

Texts in Computer Science

Wilhelm Burger
Mark J. Burge

Digital Image Processing

An Algorithmic Introduction Using Java

Second Edition



Texts in Computer Science

Series Editors

David Gries

Fred B. Schneider

More information about this series at <http://www.springer.com/series/3191>

Wilhelm Burger • Mark J. Burge

Digital Image Processing

An Algorithmic Introduction
Using Java

Second Edition



Springer

Wilhelm Burger
School of Informatics/
Communications/Media
Upper Austria University
of Applied Sciences
Hagenberg, Austria

Mark J. Burge
Noblis, Inc.
Washington, DC, USA

Series Editors

David Gries
Department of Computer Science
Cornell University
Ithaca, NY, USA

Fred B. Schneider
Department of Computer Science
Cornell University
Ithaca, NY, USA

ISSN 1868-0941
Texts in Computer Science
ISBN 978-1-4471-6683-2
DOI 10.1007/978-1-4471-6684-9

ISSN 1868-095X (electronic)
ISBN 978-1-4471-6684-9 (eBook)

Library of Congress Control Number: 2016933770

© Springer-Verlag London 2008, 2016

The author(s) has/have asserted their right(s) to be identified as the author(s) of this work in accordance with the Copyright, Design and Patents Act 1988.

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer-Verlag London Ltd.

Preface

This book provides a modern, self-contained introduction to digital image processing. We designed the book to be used both by learners desiring a firm foundation on which to build as well as practitioners in search of detailed analysis and transparent implementations of the most important techniques. This is the second English edition of the original German-language book, which has been widely used by:

- Scientists and engineers who use image processing as a tool and wish to develop a deeper understanding and create custom solutions to imaging problems in their field.
- IT professionals wanting a self-study course featuring easily adaptable code and completely worked out examples, enabling them to be productive right away.
- Faculty and students desiring an example-rich introductory textbook suitable for an advanced undergraduate or graduate level course that features exercises, projects, and examples that have been honed during our years of experience teaching this material.

While we concentrate on practical applications and concrete implementations, we do so without glossing over the important formal details and mathematics necessary for a deeper understanding of the algorithms. In preparing this text, we started from the premise that simply creating a recipe book of imaging solutions would not provide the deeper understanding needed to apply these techniques to novel problems, so instead our solutions are developed stepwise from three different perspectives: in mathematical form, as abstract pseudocode algorithms, and as complete Java programs. We use a common notation to intertwine all three perspectives—providing multiple, but linked, views of the problem and its solution.

Prerequisites

Instead of presenting digital image processing as a mathematical discipline, or strictly as signal processing, we present it from a practitioner's and programmer's perspective and with a view toward replacing many of the formalisms commonly used in other texts with constructs more readily understandable by our audience. To take full advantage of the *programming* components of this book, a knowledge of basic data structures and object-oriented programming, ideally in Java, is required. We selected Java for a number of reasons: it is the first programming language learned by students in a wide variety of engineering curricula, and professionals with knowledge of a related language, especially C# or C++, will find the programming examples easy to follow and extend.

The software in this book is designed to work with ImageJ, a widely used, programmer-extensible, imaging system developed, maintained, and distributed by the National Institutes of Health (NIH).¹ ImageJ is implemented completely in Java, and therefore runs on all major platforms, and is widely used because its “plugin”-based architecture enables it to be easily extended. While all examples run in ImageJ, they have been specifically designed to be easily ported to other environments and programming languages.

Use in research and development

This book has been especially designed for use as a textbook and as such features exercises and carefully constructed examples that supplement our detailed presentation of the fundamental concepts and techniques. As both practitioners and developers, we know that the details required to successfully understand, apply, and extend classical techniques are often difficult to find, and for this reason we have been very careful to provide the missing details, many gleaned over years of practical application. While this should make the text particularly valuable to those in research and development, it is not designed as a comprehensive, fully-cited scientific research text. On the contrary, we have carefully vetted our citations so that they can be obtained from easily accessible sources. While we have only briefly discussed the fundamentals of, or entirely omitted, topics such as hierarchical methods, wavelets, or eigenimages because of space limitations, other topics have been left out deliberately, including advanced issues such as object recognition, image understanding, and three-dimensional (3D) computer vision. So, while most techniques described in this book could be called “blind and dumb”, it is our experience that straightforward, technically clean implementations of these simpler methods are essential to the success of any further domain-specific, or even “intelligent”, approaches.

If you are only in search of a programming handbook for ImageJ or Java, there are certainly better sources. While the book includes many code examples, programming in and of itself is not our main focus. Instead Java serves as just one important element for describing each technique in a precise and immediately testable way.

Classroom use

Whether it is called signal processing, image processing, or media computation, the manipulation of digital images has been an integral part of most computer science and engineering curricula for many years. Today, with the omnipresence of all-digital work flows, it has become an integral part of the required skill set for professionals in many diverse disciplines.

Today the topic has migrated into the early stages of many curricula, where it is often a key foundation course. This migration uncovered a problem in that many of the texts relied on as standards

¹ <http://rsb.info.nih.gov/ij/>.

in the older graduate-level courses were not appropriate for beginners. The texts were usually too formal for novices, and at the same time did not provide detailed coverage of many of the most popular methods used in actual practice. The result was that educators had a difficult time selecting a single textbook or even finding a compact collection of literature to recommend to their students. Faced with this dilemma ourselves, we wrote this book in the sincere hope of filling this gap.

The contents of the following chapters can be presented in either a one- or two-semester sequence. Where feasible, we have added supporting material in order to make each chapter as independent as possible, providing the instructor with maximum flexibility when designing the course. Chapters 18–20 offer a complete introduction to the fundamental spectral techniques used in image processing and are essentially independent of the other material in the text. Depending on the goals of the instructor and the curriculum, they can be covered in as much detail as required or completely omitted. The following road map shows a possible partitioning of topics for a two-semester syllabus.

Road Map for a 1/2-Semester Syllabus	Sem.	1	2
1. Digital Images		■	□
2. ImageJ		■	□
3. Histograms and Image Statistics		■	□
4. Point Operations		■	□
5. Filters		■	□
6. Edges and Contours		■	□
7. Corner Detection		□	■
8. The Hough Transform: Finding Simple Curves		□	■
9. Morphological Filters		■	□
10. Regions in Binary Images		■	□
11. Automatic Thresholding		□	■
12. Color Images		■	□
13. Color Quantization		□	■
14. Colorimetric Color Spaces		□	■
15. Filters for Color Images		□	■
16. Edge Detection in Color Images		□	■
17. Edge-Preserving Smoothing Filters		□	■
18. Introduction to Spectral Techniques		□	■
19. The Discrete Fourier Transform in 2D		□	■
20. The Discrete Cosine Transform (DCT)		□	■
21. Geometric Operations		■	□
22. Pixel Interpolation		■	□
23. Image Matching and Registration		■	□
24. Non-Rigid Image Matching		□	■
25. Scale-Invariant Local Features (SIFT)		□	■
26. Fourier Shape Descriptors		□	■

Addendum to the second edition

This second edition is based on our completely revised German third edition and contains both additional material and several new chap-

ters including: *automatic thresholding* (Ch. 11), *filters and edge detection for color images* (Chs. 15 and 16), *edge-preserving smoothing filters* (Ch. 17), *non-rigid image matching* (Ch. 24), and *Fourier shape descriptors* (Ch. 26). Much of this new material is presented for the first time at the level of detail necessary to completely understand and create a working implementation.

The two final chapters on SIFT and Fourier shape descriptors are particularly detailed to demonstrate the actual level of granularity and the special cases which must be considered when actually implementing complex techniques. Some other chapters have been rearranged or split into multiple parts for more clarity and easier use in teaching. The mathematical notation and programming examples were completely revised and almost all illustrations were adapted or created anew for this full-color edition.

For this edition, the *ImageJ Short Reference* and ancillary source code have been relocated from the Appendix and the most recently versions are freely available in electronic form from the book's website. The complete source code, consisting of the common `imagingbook` library, sample ImageJ plugins for each book chapter, and extended documentation are available from the book's SourceForge site.²

Online resources

Visit the website for this book

www.imagingbook.com

to download supplementary materials, including the complete Java source code for all examples and the underlying software library, full-size test images, useful references, and other supplements. Comments, questions, and corrections are welcome and may be addressed to

imagingbook@gmail.com

Exercises and solutions

Each chapter of this book contains a set of sample exercises, mainly for supporting instructors to prepare their own assignments. Most of these tasks are easy to solve after studying the corresponding chapter, while some others may require more elaborated reasoning or experimental work. We assume that scholars know best how to select and adapt individual assignments in order to fit the level and interest of their students. This is the main reason why we have abstained from publishing explicit solutions in the past. However, we are happy to answer any personal request if an exercise is unclear or seems to elude a simple solution.

Thank you!

This book would not have been possible without the understanding and support of our families. Our thanks go to Wayne Rasband at NIH for developing ImageJ and for his truly outstanding support of

² <http://sourceforge.net/projects/imagingbook/>.

the community and to all our readers of the previous editions who provided valuable input, suggestions for improvement, and encouragement. The use of open source software for such a project always carries an element of risk, since the long-term acceptance and continuity is difficult to assess. Retrospectively, choosing ImageJ as the software basis for this work was a good decision, and we would consider ourselves happy if our book has indirectly contributed to the success of the ImageJ project itself. Finally, we owe a debt of gratitude to the professionals at Springer, particularly to Wayne Wheeler and Simon Reeves who were responsible for the English edition.

PREFACE

Hagenberg / Washington D.C.
Fall 2015

Contents

1	Digital Images	1
1.1	Programming with Images	2
1.2	Image Analysis and Computer Vision	2
1.3	Types of Digital Images	4
1.4	Image Acquisition	4
1.4.1	The Pinhole Camera Model	4
1.4.2	The “Thin” Lens	6
1.4.3	Going Digital	7
1.4.4	Image Size and Resolution	8
1.4.5	Image Coordinate System	9
1.4.6	Pixel Values	9
1.5	Image File Formats	11
1.5.1	Raster versus Vector Data	12
1.5.2	Tagged Image File Format (TIFF)	12
1.5.3	Graphics Interchange Format (GIF)	13
1.5.4	Portable Network Graphics (PNG)	14
1.5.5	JPEG	14
1.5.6	Windows Bitmap (BMP)	18
1.5.7	Portable Bitmap Format (PBM)	18
1.5.8	Additional File Formats	18
1.5.9	Bits and Bytes	19
1.6	Exercises	21
2	ImageJ	23
2.1	Software for Digital Imaging	24
2.2	ImageJ Overview	24
2.2.1	Key Features	25
2.2.2	Interactive Tools	26
2.2.3	ImageJ Plugins	26
2.2.4	A First Example: Inverting an Image	28
2.2.5	Plugin My_Inverter_A (using PlugInFilter)	28
2.2.6	Plugin My_Inverter_B (using PlugIn)	29
2.2.7	When to use PlugIn or PlugInFilter?	30
2.2.8	Executing ImageJ “Commands”	32
2.3	Additional Information on ImageJ and Java	34
2.3.1	Resources for ImageJ	34
2.3.2	Programming with Java	34
2.4	Exercises	34

CONTENTS	3	Histograms and Image Statistics	37
	3.1	What is a Histogram?	38
	3.2	Interpreting Histograms	39
	3.2.1	Image Acquisition	39
	3.2.2	Image Defects	41
	3.3	Calculating Histograms	43
	3.4	Histograms of Images with More than 8 Bits	45
	3.4.1	Binning	45
	3.4.2	Example	45
	3.4.3	Implementation	46
	3.5	Histograms of Color Images	46
	3.5.1	Intensity Histograms	47
	3.5.2	Individual Color Channel Histograms	47
	3.5.3	Combined Color Histograms	48
	3.6	The Cumulative Histogram	49
	3.7	Statistical Information from the Histogram	49
	3.7.1	Mean and Variance	50
	3.7.2	Median	51
	3.8	Block Statistics	51
	3.8.1	Integral Images	51
	3.8.2	Mean Intensity	53
	3.8.3	Variance	53
	3.8.4	Practical Calculation of Integral Images	53
	3.9	Exercises	54
	4	Point Operations	57
	4.1	Modifying Image Intensity	58
	4.1.1	Contrast and Brightness	58
	4.1.2	Limiting Values by Clamping	58
	4.1.3	Inverting Images	59
	4.1.4	Threshold Operation	59
	4.2	Point Operations and Histograms	59
	4.3	Automatic Contrast Adjustment	61
	4.4	Modified Auto-Contrast Operation	62
	4.5	Histogram Equalization	63
	4.6	Histogram Specification	66
	4.6.1	Frequencies and Probabilities	67
	4.6.2	Principle of Histogram Specification	67
	4.6.3	Adjusting to a Piecewise Linear Distribution	68
	4.6.4	Adjusting to a Given Histogram (Histogram Matching)	70
	4.6.5	Examples	71
	4.7	Gamma Correction	74
	4.7.1	Why Gamma?	75
	4.7.2	Mathematical Definition	77
	4.7.3	Real Gamma Values	77
	4.7.4	Applications of Gamma Correction	78
	4.7.5	Implementation	79
	4.7.6	Modified Gamma Correction	80
	4.8	Point Operations in ImageJ	82
	4.8.1	Point Operations with Lookup Tables	82
	4.8.2	Arithmetic Operations	83

4.8.3	Point Operations Involving Multiple Images	83
4.8.4	Methods for Point Operations on Two Images	84
4.8.5	ImageJ Plugins Involving Multiple Images	85
4.9	Exercises	86
5	Filters	89
5.1	What is a Filter?	89
5.2	Linear Filters	91
5.2.1	The Filter Kernel	91
5.2.2	Applying the Filter	91
5.2.3	Implementing the Filter Operation	93
5.2.4	Filter Plugin Examples	93
5.2.5	Integer Coefficients	95
5.2.6	Filters of Arbitrary Size	96
5.2.7	Types of Linear Filters	97
5.3	Formal Properties of Linear Filters	99
5.3.1	Linear Convolution	100
5.3.2	Formal Properties of Linear Convolution	101
5.3.3	Separability of Linear Filters	102
5.3.4	Impulse Response of a Filter	104
5.4	Nonlinear Filters	105
5.4.1	Minimum and Maximum Filters	105
5.4.2	Median Filter	107
5.4.3	Weighted Median Filter	109
5.4.4	Other Nonlinear Filters	111
5.5	Implementing Filters	112
5.5.1	Efficiency of Filter Programs	112
5.5.2	Handling Image Borders	113
5.5.3	Debugging Filter Programs	114
5.6	Filter Operations in ImageJ	115
5.6.1	Linear Filters	115
5.6.2	Gaussian Filters	115
5.6.3	Nonlinear Filters	116
5.7	Exercises	116
6	Edges and Contours	121
6.1	What Makes an Edge?	121
6.2	Gradient-Based Edge Detection	122
6.2.1	Partial Derivatives and the Gradient	123
6.2.2	Derivative Filters	123
6.3	Simple Edge Operators	124
6.3.1	Prewitt and Sobel Operators	125
6.3.2	Roberts Operator	127
6.3.3	Compass Operators	128
6.3.4	Edge Operators in ImageJ	130
6.4	Other Edge Operators	130
6.4.1	Edge Detection Based on Second Derivatives	130
6.4.2	Edges at Different Scales	130
6.4.3	From Edges to Contours	131
6.5	Canny Edge Operator	132
6.5.1	Pre-processing	134
6.5.2	Edge localization	134

6.5.3	Edge tracing and hysteresis thresholding	135
6.5.4	Additional Information	137
6.5.5	Implementation	138
6.6	Edge Sharpening	139
6.6.1	Edge Sharpening with the Laplacian Filter	139
6.6.2	Unsharp Masking	142
6.7	Exercises	146
7	Corner Detection	147
7.1	Points of Interest	147
7.2	Harris Corner Detector	148
7.2.1	Local Structure Matrix	148
7.2.2	Corner Response Function (CRF)	149
7.2.3	Determining Corner Points	149
7.2.4	Examples	150
7.3	Implementation	152
7.3.1	Step 1: Calculating the Corner Response Function	153
7.3.2	Step 2: Selecting “Good” Corner Points	155
7.3.3	Step 3: Cleaning up	156
7.3.4	Summary	157
7.4	Exercises	158
8	Finding Simple Curves: The Hough Transform	161
8.1	Salient Image Structures	161
8.2	The Hough Transform	162
8.2.1	Parameter Space	163
8.2.2	Accumulator Map	164
8.2.3	A Better Line Representation	165
8.3	Hough Algorithm	167
8.3.1	Processing the Accumulator Array	168
8.3.2	Hough Transform Extensions	170
8.4	Java Implementation	173
8.5	Hough Transform for Circles and Ellipses	176
8.5.1	Circles and Arcs	176
8.5.2	Ellipses	177
8.6	Exercises	179
9	Morphological Filters	181
9.1	Shrink and Let Grow	182
9.1.1	Neighborhood of Pixels	183
9.2	Basic Morphological Operations	183
9.2.1	The Structuring Element	183
9.2.2	Point Sets	184
9.2.3	Dilation	185
9.2.4	Erosion	186
9.2.5	Formal Properties of Dilation and Erosion	186
9.2.6	Designing Morphological Filters	188
9.2.7	Application Example: Outline	189
9.3	Composite Morphological Operations	192
9.3.1	Opening	192
9.3.2	Closing	192

9.3.3 Properties of Opening and Closing	193
9.4 Thinning (Skeletonization)	194
9.4.1 Thinning Algorithm by Zhang and Suen	194
9.4.2 Fast Thinning Algorithm	195
9.4.3 Java Implementation	198
9.4.4 Built-in Morphological Operations in ImageJ	201
9.5 Grayscale Morphology	202
9.5.1 Structuring Elements	202
9.5.2 Dilation and Erosion	203
9.5.3 Grayscale Opening and Closing	203
9.6 Exercises	205
10 Regions in Binary Images	209
10.1 Finding Connected Image Regions	210
10.1.1 Region Labeling by Flood Filling	210
10.1.2 Sequential Region Labeling	213
10.1.3 Region Labeling—Summary	219
10.2 Region Contours	219
10.2.1 External and Internal Contours	219
10.2.2 Combining Region Labeling and Contour Finding	220
10.2.3 Java Implementation	222
10.3 Representing Image Regions	225
10.3.1 Matrix Representation	225
10.3.2 Run Length Encoding	225
10.3.3 Chain Codes	226
10.4 Properties of Binary Regions	229
10.4.1 Shape Features	229
10.4.2 Geometric Features	230
10.5 Statistical Shape Properties	232
10.5.1 Centroid	233
10.5.2 Moments	233
10.5.3 Central Moments	234
10.5.4 Normalized Central Moments	234
10.5.5 Java Implementation	234
10.6 Moment-Based Geometric Properties	235
10.6.1 Orientation	235
10.6.2 Eccentricity	237
10.6.3 Bounding Box Aligned to the Major Axis . .	239
10.6.4 Invariant Region Moments	241
10.7 Projections	244
10.8 Topological Region Properties	244
10.9 Java Implementation	246
10.10 Exercises	246
11 Automatic Thresholding	253
11.1 Global Histogram-Based Thresholding	253
11.1.1 Image Statistics from the Histogram	255
11.1.2 Simple Threshold Selection	256
11.1.3 Iterative Threshold Selection (Isodata Algorithm)	258
11.1.4 Otsu's Method	260

11.1.5	Maximum Entropy Thresholding	263
11.1.6	Minimum Error Thresholding	266
11.2	Local Adaptive Thresholding	273
11.2.1	Bernsen's Method	274
11.2.2	Niblack's Method	275
11.3	Java Implementation	284
11.3.1	Global Thresholding Methods	285
11.3.2	Adaptive Thresholding	287
11.4	Summary and Further Reading	288
11.5	Exercises	289
12	Color Images	291
12.1	RGB Color Images	291
12.1.1	Structure of Color Images	292
12.1.2	Color Images in ImageJ	296
12.2	Color Spaces and Color Conversion	303
12.2.1	Conversion to Grayscale	304
12.2.2	Desaturating RGB Color Images	306
12.2.3	HSV/HSB and HLS Color Spaces	306
12.2.4	TV Component Color Spaces—YUV, YIQ, and YC_bC_r	317
12.2.5	Color Spaces for Printing—CMY and CMYK .	320
12.3	Statistics of Color Images	323
12.3.1	How Many Different Colors are in an Image? .	323
12.3.2	Color Histograms	324
12.4	Exercises	325
13	Color Quantization	329
13.1	Scalar Color Quantization	329
13.2	Vector Quantization	331
13.2.1	Populosity Algorithm	331
13.2.2	Median-Cut Algorithm	332
13.2.3	Octree Algorithm	333
13.2.4	Other Methods for Vector Quantization . . .	336
13.2.5	Java Implementation	337
13.3	Exercises	337
14	Colorimetric Color Spaces	341
14.1	CIE Color Spaces	341
14.1.1	CIE XYZ Color Space	342
14.1.2	CIE x, y Chromaticity	342
14.1.3	Standard Illuminants	344
14.1.4	Gamut	345
14.1.5	Variants of the CIE Color Space	345
14.2	CIELAB	346
14.2.1	CIEXYZ→CIELAB Conversion	346
14.2.2	CIELAB→CIEXYZ Conversion	347
14.3	CIELUV	348
14.3.1	CIEXYZ→CIELUV Conversion	348
14.3.2	CIELUV→CIEXYZ Conversion	350
14.3.3	Measuring Color Differences	350
14.4	Standard RGB (sRGB)	350

14.4.1	Linear vs. Nonlinear Color Components	351
14.4.2	CIEXYZ→sRGB Conversion	352
14.4.3	sRGB→CIEXYZ Conversion	353
14.4.4	Calculations with Nonlinear sRGB Values	353
14.5	Adobe RGB	354
14.6	Chromatic Adaptation	355
14.6.1	XYZ Scaling	355
14.6.2	Bradford Adaptation	356
14.7	Colorimetric Support in Java	358
14.7.1	Profile Connection Space (PCS)	358
14.7.2	Color-Related Java Classes	360
14.7.3	Implementation of the CIELAB Color Space (Example)	361
14.7.4	ICC Profiles	362
14.8	Exercises	365
15	Filters for Color Images	367
15.1	Linear Filters	367
15.1.1	Monochromatic Application of Linear Filters	368
15.1.2	Color Space Considerations	370
15.1.3	Linear Filtering with Circular Values	374
15.2	Nonlinear Color Filters	378
15.2.1	Scalar Median Filter	378
15.2.2	Vector Median Filter	378
15.2.3	Sharpening Vector Median Filter	382
15.3	Java Implementation	385
15.4	Further Reading	387
15.5	Exercises	388
16	Edge Detection in Color Images	391
16.1	Monochromatic Techniques	392
16.2	Edges in Vector-Valued Images	395
16.2.1	Multi-Dimensional Gradients	397
16.2.2	The Jacobian Matrix	397
16.2.3	Squared Local Contrast	398
16.2.4	Color Edge Magnitude	399
16.2.5	Color Edge Orientation	401
16.2.6	Grayscale Gradients Revisited	401
16.3	Canny Edge Detector for Color Images	404
16.4	Other Color Edge Operators	406
16.5	Java Implementation	410
17	Edge-Preserving Smoothing Filters	413
17.1	Kuwahara-Type Filters	414
17.1.1	Application to Color Images	416
17.2	Bilateral Filter	420
17.2.1	Domain Filter	420
17.2.2	Range Filter	421
17.2.3	Bilateral Filter—General Idea	421
17.2.4	Bilateral Filter with Gaussian Kernels	423
17.2.5	Application to Color Images	424
17.2.6	Efficient Implementation by x/y Separation	428

17.3	17.2.7 Further Reading	432
17.3	17.3 Anisotropic Diffusion Filters.....	433
17.3.1	17.3.1 Homogeneous Diffusion and the Heat Equation	434
17.3.2	17.3.2 Perona-Malik Filter	436
17.3.3	17.3.3 Perona-Malik Filter for Color Images	438
17.3.4	17.3.4 Geometry Preserving Anisotropic Diffusion ..	441
17.3.5	17.3.5 Tschumperlé-Deriche Algorithm	444
17.4	17.4 Java Implementation	448
17.5	17.5 Exercises	450
18	18 Introduction to Spectral Techniques	453
18.1	18.1 The Fourier Transform	454
18.1.1	18.1.1 Sine and Cosine Functions	454
18.1.2	18.1.2 Fourier Series Representation of Periodic Functions	457
18.1.3	18.1.3 Fourier Integral	457
18.1.4	18.1.4 Fourier Spectrum and Transformation	458
18.1.5	18.1.5 Fourier Transform Pairs	459
18.1.6	18.1.6 Important Properties of the Fourier Transform	460
18.2	18.2 Working with Discrete Signals	464
18.2.1	18.2.1 Sampling	464
18.2.2	18.2.2 Discrete and Periodic Functions	469
18.3	18.3 The Discrete Fourier Transform (DFT)	469
18.3.1	18.3.1 Definition of the DFT	469
18.3.2	18.3.2 Discrete Basis Functions	472
18.3.3	18.3.3 Aliasing Again!	472
18.3.4	18.3.4 Units in Signal and Frequency Space	475
18.3.5	18.3.5 Power Spectrum	477
18.4	18.4 Implementing the DFT	477
18.4.1	18.4.1 Direct Implementation	477
18.4.2	18.4.2 Fast Fourier Transform (FFT)	479
18.5	18.5 Exercises	479
19	19 The Discrete Fourier Transform in 2D	481
19.1	19.1 Definition of the 2D DFT	481
19.1.1	19.1.1 2D Basis Functions	481
19.1.2	19.1.2 Implementing the 2D DFT	482
19.2	19.2 Visualizing the 2D Fourier Transform	485
19.2.1	19.2.1 Range of Spectral Values	485
19.2.2	19.2.2 Centered Representation of the DFT Spectrum	485
19.3	19.3 Frequencies and Orientation in 2D	486
19.3.1	19.3.1 Effective Frequency	486
19.3.2	19.3.2 Frequency Limits and Aliasing in 2D	487
19.3.3	19.3.3 Orientation	488
19.3.4	19.3.4 Normalizing the Geometry of the 2D Spectrum	488
19.3.5	19.3.5 Effects of Periodicity	489
19.3.6	19.3.6 Windowing	490
19.3.7	19.3.7 Common Windowing Functions	491
19.4	19.4 2D Fourier Transform Examples	492

19.5	Applications of the DFT	496
19.5.1	Linear Filter Operations in Frequency Space	496
19.5.2	Linear Convolution and Correlation	499
19.5.3	Inverse Filters	499
19.6	Exercises	500
20	The Discrete Cosine Transform (DCT)	503
20.1	1D DCT	503
20.1.1	DCT Basis Functions	504
20.1.2	Implementing the 1D DCT	504
20.2	2D DCT	504
20.2.1	Examples	506
20.2.2	Separability	507
20.3	Java Implementation	509
20.4	Other Spectral Transforms	510
20.5	Exercises	510
21	Geometric Operations	513
21.1	2D Coordinate Transformations	514
21.1.1	Simple Geometric Mappings	514
21.1.2	Homogeneous Coordinates	515
21.1.3	Affine (Three-Point) Mapping	516
21.1.4	Projective (Four-Point) Mapping	519
21.1.5	Bilinear Mapping	525
21.1.6	Other Nonlinear Image Transformations	526
21.1.7	Piecewise Image Transformations	528
21.2	Resampling the Image	529
21.2.1	Source-to-Target Mapping	530
21.2.2	Target-to-Source Mapping	530
21.3	Java Implementation	531
21.3.1	General Mappings (Class Mapping)	532
21.3.2	Linear Mappings	532
21.3.3	Nonlinear Mappings	533
21.3.4	Sample Applications	533
21.4	Exercises	534
22	Pixel Interpolation	539
22.1	Simple Interpolation Methods	539
22.1.1	Ideal Low-Pass Filter	540
22.2	Interpolation by Convolution	543
22.3	Cubic Interpolation	544
22.4	Spline Interpolation	546
22.4.1	Catmull-Rom Interpolation	546
22.4.2	Cubic B-spline Approximation	547
22.4.3	Mitchell-Netravali Approximation	547
22.4.4	Lanczos Interpolation	548
22.5	Interpolation in 2D	549
22.5.1	Nearest-Neighbor Interpolation in 2D	550
22.5.2	Bilinear Interpolation	551
22.5.3	Bicubic and Spline Interpolation in 2D	553
22.5.4	Lanczos Interpolation in 2D	554
22.5.5	Examples and Discussion	555

22.6	Aliasing	556
22.6.1	Sampling the Interpolated Image	557
22.6.2	Low-Pass Filtering	558
22.7	Java Implementation	560
22.8	Exercises	563
23	Image Matching and Registration.....	565
23.1	Template Matching in Intensity Images	566
23.1.1	Distance between Image Patterns	566
23.1.2	Matching Under Rotation and Scaling	574
23.1.3	Java Implementation	574
23.2	Matching Binary Images	574
23.2.1	Direct Comparison of Binary Images	576
23.2.2	The Distance Transform	576
23.2.3	Chamfer Matching	580
23.2.4	Java Implementation	582
23.3	Exercises	583
24	Non-Rigid Image Matching	587
24.1	The Lucas-Kanade Technique	587
24.1.1	Registration in 1D	587
24.1.2	Extension to Multi-Dimensional Functions ..	589
24.2	The Lucas-Kanade Algorithm	590
24.2.1	Summary of the Algorithm.....	593
24.3	Inverse Compositional Algorithm	595
24.4	Parameter Setups for Various Linear Transformations	598
24.4.1	Pure Translation.....	598
24.4.2	Affine Transformation	599
24.4.3	Projective Transformation	601
24.4.4	Concatenating Linear Transformations ..	601
24.5	Example	602
24.6	Java Implementation	603
24.6.1	Application Example	605
24.7	Exercises	607
25	Scale-Invariant Feature Transform (SIFT)	609
25.1	Interest Points at Multiple Scales	610
25.1.1	The LoG Filter	610
25.1.2	Gaussian Scale Space.....	615
25.1.3	LoG/DoG Scale Space.....	619
25.1.4	Hierarchical Scale Space	620
25.1.5	Scale Space Structure in SIFT	624
25.2	Key Point Selection and Refinement.....	630
25.2.1	Local Extrema Detection	630
25.2.2	Position Refinement	632
25.2.3	Suppressing Responses to Edge-Like Structures	634
25.3	Creating Local Descriptors	636
25.3.1	Finding Dominant Orientations.....	637
25.3.2	SIFT Descriptor Construction	640
25.4	SIFT Algorithm Summary	647
25.5	Matching SIFT Features	648

25.5.1	Feature Distance and Match Quality	648	CONTENTS
25.5.2	Examples	654	
25.6	Efficient Feature Matching	657	
25.7	Java Implementation	661	
25.7.1	SIFT Feature Extraction	662	
25.7.2	SIFT Feature Matching	663	
25.8	Exercises	663	
26	Fourier Shape Descriptors	665	
26.1	Closed Curves in the Complex Plane	665	
26.1.1	Discrete 2D Curves	665	
26.2	Discrete Fourier Transform (DFT)	667	
26.2.1	Forward Fourier Transform	668	
26.2.2	Inverse Fourier Transform (Reconstruction) .	668	
26.2.3	Periodicity of the DFT Spectrum	670	
26.2.4	Truncating the DFT Spectrum	672	
26.3	Geometric Interpretation of Fourier Coefficients	673	
26.3.1	Coefficient G_0 Corresponds to the Contour's Centroid	673	
26.3.2	Coefficient G_1 Corresponds to a Circle	674	
26.3.3	Coefficient G_m Corresponds to a Circle with Frequency m	675	
26.3.4	Negative Frequencies	676	
26.3.5	Fourier Descriptor Pairs Correspond to Ellipses	676	
26.3.6	Shape Reconstruction from Truncated Fourier Descriptors	679	
26.3.7	Fourier Descriptors from Unsampled Polygons	682	
26.4	Effects of Geometric Transformations	687	
26.4.1	Translation	687	
26.4.2	Scale Change	688	
26.4.3	Rotation	688	
26.4.4	Shifting the Sampling Start Position	689	
26.4.5	Effects of Phase Removal	690	
26.4.6	Direction of Contour Traversal	691	
26.4.7	Reflection (Symmetry)	691	
26.5	Transformation-Invariant Fourier Descriptors	692	
26.5.1	Scale Invariance	693	
26.5.2	Start Point Invariance	694	
26.5.3	Rotation Invariance	696	
26.5.4	Other Approaches	697	
26.6	Shape Matching with Fourier Descriptors	700	
26.6.1	Magnitude-Only Matching	700	
26.6.2	Complex (Phase-Preserving) Matching	701	
26.7	Java Implementation	704	
26.8	Discussion and Further Reading	708	
26.9	Exercises	709	
A	Mathematical Symbols and Notation	713	
A.1	Symbols	713	
A.2	Set Operators	717	
A.3	Complex Numbers	717	

B Linear Algebra	719
B.1 Vectors and Matrices	719
B.1.1 Column and Row Vectors	720
B.1.2 Length (Norm) of a Vector	720
B.2 Matrix Multiplication	720
B.2.1 Scalar Multiplication	720
B.2.2 Product of Two Matrices	721
B.2.3 Matrix-Vector Products	721
B.3 Vector Products	722
B.3.1 Dot (Scalar) Product	722
B.3.2 Outer Product	723
B.3.3 Cross Product	723
B.4 Eigenvectors and Eigenvalues	723
B.4.1 Calculation of Eigenvalues	724
B.5 Homogeneous Coordinates	726
B.6 Basic Matrix-Vector Operations with the <i>Apache Commons Math Library</i>	727
B.6.1 Vectors and Matrices	727
B.6.2 Matrix-Vector Multiplication	728
B.6.3 Vector Products	728
B.6.4 Inverse of a Square Matrix	728
B.6.5 Eigenvalues and Eigenvectors	728
B.7 Solving Systems of Linear Equations	729
B.7.1 Exact Solutions	730
B.7.2 Over-Determined System (Least-Squares Solutions)	731
C Calculus	733
C.1 Parabolic Fitting	733
C.1.1 Fitting a Parabolic Function to Three Sample Points	733
C.1.2 Locating Extrema by Quadratic Interpolation	734
C.2 Scalar and Vector Fields	735
C.2.1 The Jacobian Matrix	736
C.2.2 Gradients	736
C.2.3 Maximum Gradient Direction	737
C.2.4 Divergence of a Vector Field	737
C.2.5 Laplacian Operator	738
C.2.6 The Hessian Matrix	738
C.3 Operations on Multi-Variable, Scalar Functions (Scalar Fields)	739
C.3.1 Estimating the Derivatives of a Discrete Function	739
C.3.2 Taylor Series Expansion of Functions	740
C.3.3 Finding the Continuous Extremum of a Multi-Variable Discrete Function	743
D Statistical Prerequisites	749
D.1 Mean, Variance, and Covariance	749
D.1.1 Mean	749
D.1.2 Variance and Covariance	749
D.1.3 Biased vs. Unbiased Variance	750

D.2	The Covariance Matrix	750
D.2.1	Example	751
D.2.2	Practical Calculation	752
D.3	Mahalanobis Distance	752
D.3.1	Definition	752
D.3.2	Relation to the Euclidean Distance	753
D.3.3	Numerical Aspects	753
D.3.4	Pre-Mapping Data for Efficient Mahalanobis Matching	754
D.4	The Gaussian Distribution	756
D.4.1	Maximum Likelihood Estimation	756
D.4.2	Gaussian Mixtures	758
D.4.3	Creating Gaussian Noise	758
E	Gaussian Filters	761
E.1	Cascading Gaussian Filters	761
E.2	Gaussian Filters and Scale Space	761
E.3	Effects of Gaussian Filtering in the Frequency Domain	762
E.4	LoG-Approximation by the DoG	763
F	Java Notes	765
F.1	Arithmetic	765
F.1.1	Integer Division	765
F.1.2	Modulus Operator	766
F.1.3	Unsigned Byte Data	767
F.1.4	Mathematical Functions in Class <code>Math</code>	768
F.1.5	Numerical Rounding	769
F.1.6	Inverse Tangent Function	769
F.1.7	Classes <code>Float</code> and <code>Double</code>	770
F.1.8	Testing Floating-Point Values Against Zero	770
F.2	Arrays in Java	771
F.2.1	Creating Arrays	771
F.2.2	Array Size	771
F.2.3	Accessing Array Elements	771
F.2.4	2D Arrays	772
F.2.5	Arrays of Objects	775
F.2.6	Searching for Minimum and Maximum Values	775
F.2.7	Sorting Arrays	776
References	777
Index	791

Digital Images

For a long time, using a computer to manipulate a digital image (i.e., digital image processing) was something performed by only a relatively small group of specialists who had access to expensive equipment. Usually this combination of specialists and equipment was only to be found in research labs, and so the field of digital image processing has its roots in the academic realm. Now, however, the combination of a powerful computer on every desktop and the fact that nearly everyone has some type of device for digital image acquisition, be it their cell phone camera, digital camera, or scanner, has resulted in a plethora of digital images and, with that, for many digital image processing has become as common as word processing. It was not that many years ago that digitizing a photo and saving it to a file on a computer was a time-consuming task. This is perhaps difficult to imagine given today's powerful hardware and operating system level support for all types of digital media, but it is always sobering to remember that "personal" computers in the early 1990s were not powerful enough to even load into main memory a single image from a typical digital camera of today. Now powerful hardware and software packages have made it possible for amateurs to manipulate digital images and videos just as easily as professionals.

All of these developments have resulted in a large community that works productively with digital images while having only a basic understanding of the underlying mechanics. For the typical consumer merely wanting to create a digital archive of vacation photos, a deeper understanding is not required, just as a deep understanding of the combustion engine is unnecessary to successfully drive a car.

Today, IT professionals must be more than simply familiar with digital image processing. They are expected to be able to knowledgeably manipulate images and related digital media, which are an increasingly important part of the workflow not only of those involved in medicine and media but all industries. In the same way, software engineers and computer scientists are increasingly confronted with developing programs, databases, and related systems that must correctly deal with digital images. The simple lack of practical ex-

perience with this type of material, combined with an often unclear understanding of its basic foundations and a tendency to underestimate its difficulties, frequently leads to inefficient solutions, costly errors, and personal frustration.

1.1 Programming with Images

Even though the term “image processing” is often used interchangeably with that of “image editing”, we introduce the following more precise definitions. Digital image editing, or as it is sometimes referred to, digital imaging, is the manipulation of digital images using an existing software application such as Adobe Photoshop® or Corel Paint®. Digital image processing, on the other hand, is the conception, design, development, and enhancement of digital imaging programs.

Modern programming environments, with their extensive APIs (application programming interfaces), make practically every aspect of computing, be it networking, databases, graphics, sound, or imaging, easily available to nonspecialists. The possibility of developing a program that can reach into an image and manipulate the individual elements at its very core is fascinating and seductive. You will discover that with the right knowledge, an image becomes ultimately no more than a simple array of values, that with the right tools you can manipulate in any way imaginable.

“Computer graphics”, in contrast to digital image processing, concentrates on the *synthesis* of digital images from geometrical descriptions such as three-dimensional (3D) object models [75, 87, 247]. While graphics professionals today tend to be interested in topics such as realism and, especially in terms of computer games, rendering speed, the field does draw on a number of methods that originate in image processing, such as image transformation (morphing), reconstruction of 3D models from image data, and specialized techniques such as image-based and nonphotorealistic rendering [180, 248]. Similarly, image processing makes use of a number of ideas that have their origin in computational geometry and computer graphics, such as volumetric (voxel) models in medical image processing. The two fields perhaps work closest when it comes to digital postproduction of film and video and the creation of special effects [256]. This book provides a thorough grounding in the effective processing of not only images but also sequences of images; that is, videos.

1.2 Image Analysis and Computer Vision

Often it appears at first glance that a given image-processing task will have a simple solution, especially when it is something that is easily accomplished by our own visual system. Yet in practice it turns out that developing reliable, robust, and timely solutions is difficult or simply impossible. This is especially true when the problem involves image *analysis*; that is, where the ultimate goal is not to enhance or otherwise alter the appearance of an image but instead to extract

meaningful information about its contents—be it distinguishing an object from its background, following a street on a map, or finding the bar code on a milk carton, tasks such as these often turn out to be much more difficult to accomplish than we would expect.

We expect technology to improve on what we can do by ourselves. Be it as simple as a lever to lift more weight or binoculars to see farther or as complex as an airplane to move us across continents—science has created so much that improves on, sometimes by unbelievable factors, what our biological systems are able to perform. So, it is perhaps humbling to discover that today’s technology is nowhere near as capable, when it comes to image analysis, as our own visual system. While it is possible that this will always remain true, do not let this discourage you. Instead consider it a challenge to develop creative solutions. Using the tools, techniques, and fundamental knowledge available today, it is possible not only to solve many problems but to create robust, reliable, and fast applications.

While image analysis is not the main subject of this book, it often naturally intersects with image processing and we will explore this intersection in detail in these situations: finding simple curves (Ch. 8), segmenting image regions (Ch. 10), and comparing images (Ch. 23). In these cases, we present solutions that work directly on the pixel data in a *bottom-up* way without recourse to domain-specific knowledge (i.e., blind solutions). In this way, our solutions essentially embody the distinction between image processing, *pattern recognition*, and *computer vision*, respectively. While these two disciplines are firmly grounded in, and rely heavily on, image processing, their ultimate goals are much more lofty.

Pattern recognition is primarily a mathematical discipline and has been responsible for techniques such as clustering, hidden Markov models (HMMs), decision trees, and principal component analysis (PCA), which are used to discover patterns in data and signals. Methods from pattern recognition have been applied extensively to problems arising in computer vision and image analysis. A good example of their successful application is optical character recognition (OCR), where robust, highly accurate turnkey solutions are available for recognizing scanned text. Pattern recognition methods are truly universal and have been successfully applied not only to images but also speech and audio signals, text documents, stock trades, and finding trends in large databases, where it is often called data mining. Dimensionality reduction, statistical, and syntactical methods play important roles in pattern recognition (see, e.g., [64, 169, 228]).

Computer vision tackles the problem of engineering artificial visual systems capable of somehow comprehending and interpreting our real, 3D world. Popular topics in this field include scene understanding, object recognition, motion interpretation (tracking), autonomous navigation, and the robotic manipulation of objects in a scene. Since computer vision has its roots in artificial intelligence (AI), many AI methods were originally developed to either tackle or represent a problem in computer vision (see, e.g., [51, Ch. 13]). The fields still have much in common today, especially in terms of adap-

1.2 IMAGE ANALYSIS AND COMPUTER VISION

tive methods and machine learning. Further literature on computer vision includes [15, 78, 110, 214, 222, 232].

Ultimately you will find image processing to be both intellectually challenging and professionally rewarding, as the field is ripe with problems that were originally thought to be relatively simple to solve but have to this day refused to give up their secrets. With the background and techniques presented in this text, you will not only be able to develop complete image-processing solutions but will also have the prerequisite knowledge to tackle unsolved problems and the real possibility of expanding the horizons of science: for while image processing by itself may not change the world, it is likely to be the foundation that supports marvels of the future.

1.3 Types of Digital Images

Digital images are the central theme of this book, and unlike just a few years ago, this term is now so commonly used that there is really no reason to explain it further. Yet this book is not about all types of digital images, instead it focuses on images that are made up of *picture elements*, more commonly known as *pixels*, arranged in a regular rectangular grid.

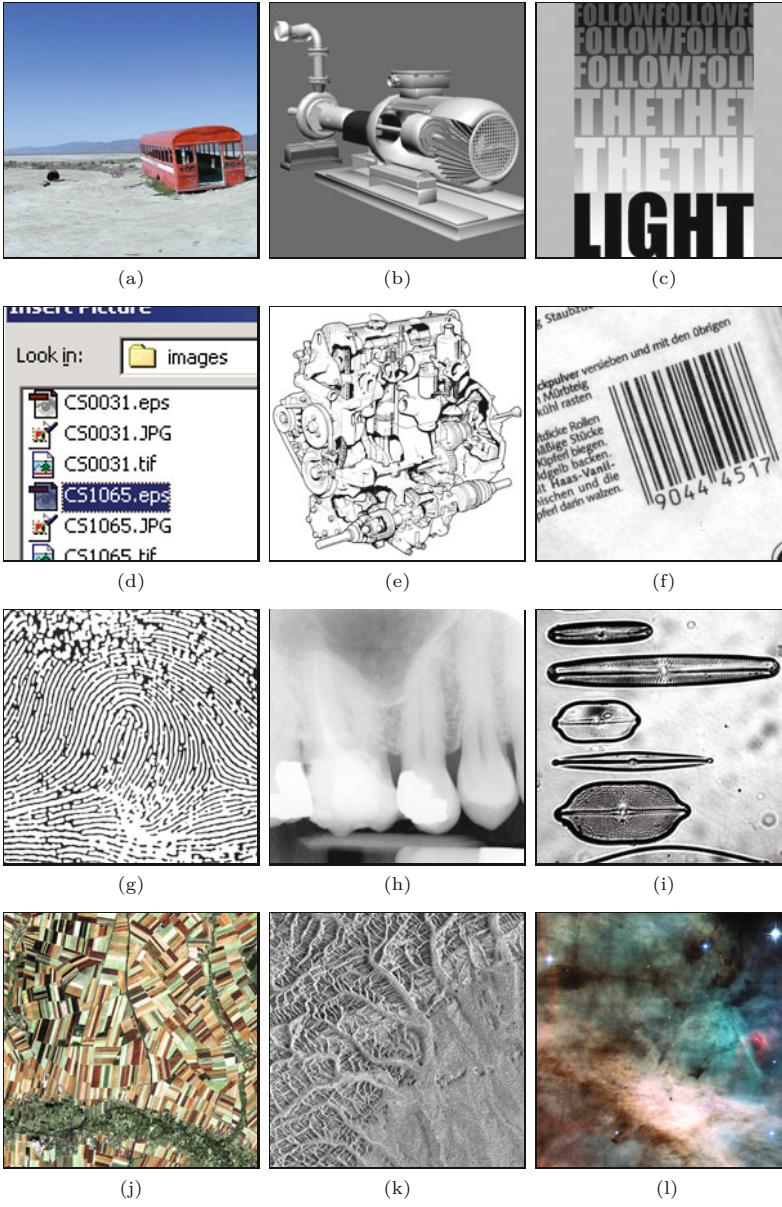
Every day, people work with a large variety of digital raster images such as color photographs of people and landscapes, grayscale scans of printed documents, building plans, faxed documents, screenshots, medical images such as x-rays and ultrasounds, and a multitude of others (see Fig. 1.1 for examples). Despite all the different sources for these images, they are all, as a rule, ultimately represented as rectangular ordered arrays of image elements.

1.4 Image Acquisition

The process by which a scene becomes a digital image is varied and complicated, and, in most cases, the images you work with will already be in digital form, so we only outline here the essential stages in the process. As most image acquisition methods are essentially variations on the classical optical camera, we will begin by examining it in more detail.

1.4.1 The Pinhole Camera Model

The pinhole camera is one of the simplest camera models and has been in use since the 13th century, when it was known as the “Camera Obscura”. While pinhole cameras have no practical use today except to hobbyists, they are a useful model for understanding the essential optical components of a simple camera. The pinhole camera consists of a closed box with a small opening on the front side through which light enters, forming an image on the opposing wall. The light forms a smaller, inverted image of the scene (Fig. 1.2).



1.4 IMAGE ACQUISITION

Fig. 1.1

Examples of digital images. Natural landscape (a), synthetically generated scene (b), poster graphic (c), computer screenshot (d), black and white illustration (e), barcode (f), fingerprint (g), x-ray (h), microscope slide (i), satellite image (j), radar image (k), astronomical object (l).

Perspective projection

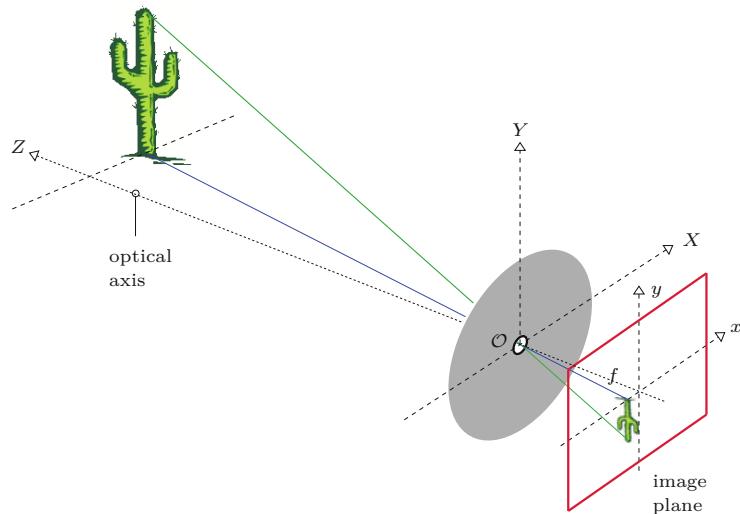
The geometric properties of the pinhole camera are very simple. The optical axis runs through the pinhole perpendicular to the image plane. We assume a visible object, in our illustration the cactus, located at a horizontal distance Z from the pinhole and vertical distance Y from the optical axis. The height of the projection y is determined by two parameters: the fixed depth of the camera box f and the distance Z to the object from the origin of the coordinate system. By comparison we see that

$$x = -f \cdot \frac{X}{Z} \quad \text{and} \quad y = -f \cdot \frac{Y}{Z} \quad (1.1)$$

Fig. 1.2

Geometry of the pinhole camera. The pinhole opening serves as the origin (\mathcal{O}) of the 3D coordinate system (X, Y, Z) for the objects in the scene.

The optical axis, which runs through the opening, is the Z axis of this coordinate system. A separate 2D coordinate system (x, y) describes the projection points on the image plane. The distance f (“focal length”) between the opening and the image plane determines the scale of the projection.



change with the scale of the resulting image in proportion to the depth of the box (i.e., the distance f) in a way similar to how the focal length does in an everyday camera. For a fixed image, a small f (i.e., short focal length) results in a small image and a large viewing angle, just as occurs when a wide-angle lens is used, while increasing the “focal length” f results in a larger image and a smaller viewing angle, just as occurs when a telephoto lens is used. The negative sign in Eqn. (1.1) means that the projected image is flipped in the horizontal and vertical directions and rotated by 180° .

Equation (1.1) describes what is commonly known today as the *perspective transformation*.¹ Important properties of this theoretical model are that straight lines in 3D space always appear straight in 2D projections and that circles appear as ellipses.

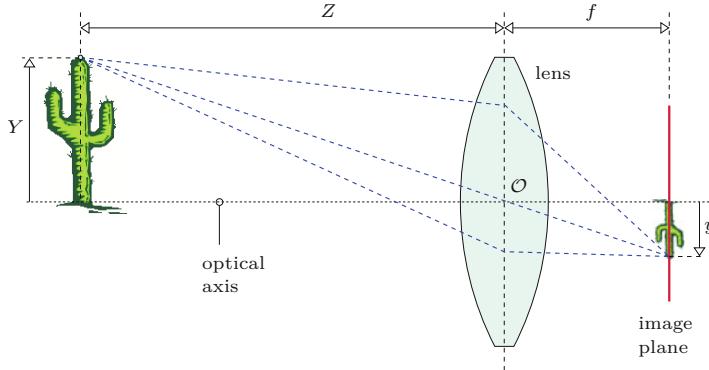
1.4.2 The “Thin” Lens

While the simple geometry of the pinhole camera makes it useful for understanding its basic principles, it is never really used in practice. One of the problems with the pinhole camera is that it requires a very small opening to produce a sharp image. This in turn reduces the amount of light passed through and thus leads to extremely long exposure times. In reality, glass lenses or systems of optical lenses are used whose optical properties are greatly superior in many aspects but of course are also much more complex. Instead we can make our model more realistic, without unduly increasing its complexity, by replacing the pinhole with a “thin lens” as in Fig. 1.3.

In this model, the lens is assumed to be symmetric and infinitely thin, such that all light rays passing through it cross through a virtual plane in the middle of the lens. The resulting image geometry is the same as that of the pinhole camera. This model is not sufficiently complex to encompass the physical details of actual lens systems, such

¹ It is hard to imagine today that the rules of perspective geometry, while known to the ancient mathematicians, were only rediscovered in 1430 by the Renaissance painter Brunelleschi.

Fig. 1.3
Thin lens projection model.



as geometrical distortions and the distinct refraction properties of different colors. So, while this simple model suffices for our purposes (i.e., understanding the mechanics of image acquisition), much more detailed models that incorporate these additional complexities can be found in the literature (see, e.g., [126]).

1.4.3 Going Digital

What is projected on the image plane of our camera is essentially a two-dimensional (2D), time-dependent, continuous distribution of light energy. In order to convert this image into a digital image on our computer, the following three main steps are necessary:

1. The continuous light distribution must be spatially sampled.
2. This resulting function must then be sampled in time to create a single (still) image.
3. Finally, the resulting values must be quantized to a finite range of integers (or floating-point values) such that they can be represented by digital numbers.

Step 1: Spatial sampling

The spatial sampling of an image (i.e., the conversion of the continuous signal to its discrete representation) depends on the geometry of the sensor elements of the acquisition device (e.g., a digital or video camera). The individual sensor elements are arranged in ordered rows, almost always at right angles to each other, along the sensor plane (Fig. 1.4). Other types of image sensors, which include hexagonal elements and circular sensor structures, can be found in specialized products.

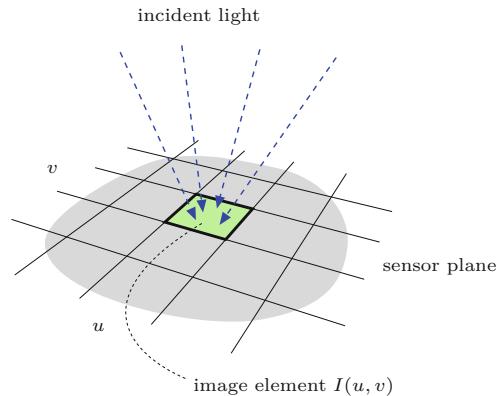
Step 2: Temporal sampling

Temporal sampling is carried out by measuring at regular intervals the amount of light incident on each individual sensor element. The CCD² in a digital camera does this by triggering the charging process and then measuring the amount of electrical charge that has built up during the specified amount of time that the CCD was illuminated.

² Charge-coupled device.

Fig. 1.4

The geometry of the sensor elements is directly responsible for the spatial sampling of the continuous image. In the simplest case, a plane of sensor elements are arranged in an evenly spaced grid, and each element measures the amount of light that falls on it.



Step 3: Quantization of pixel values

In order to store and process the image values on the computer they are commonly converted to an integer scale (e.g., $256 = 2^8$ or $4096 = 2^{12}$). Occasionally floating-point values are used in professional applications, such as medical imaging. Conversion is carried out using an analog to digital converter, which is typically embedded directly in the sensor electronics so that conversion occurs at image capture or is performed by special interface hardware.

Images as discrete functions

The result of these three stages is a description of the image in the form of a 2D, ordered matrix of integers (Fig. 1.5). Stated a bit more formally, a digital image I is a 2D function that maps from the domain of integer coordinates $\mathbb{N} \times \mathbb{N}$ to a range of possible pixel values \mathbb{P} such that

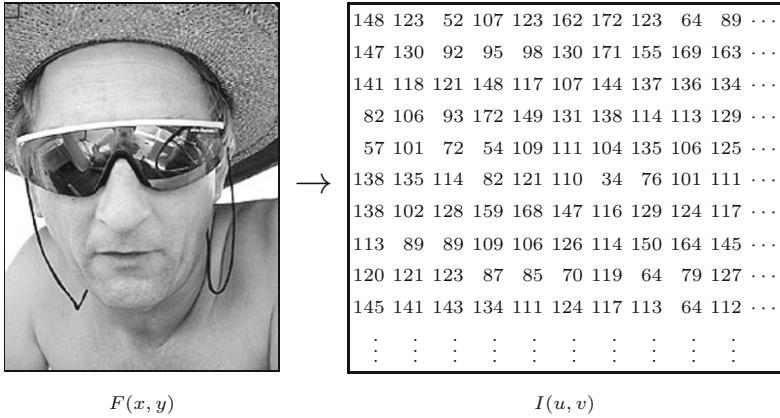
$$I(u, v) \in \mathbb{P} \quad \text{and} \quad u, v \in \mathbb{N}.$$

Now we are ready to transfer the image to our computer so that we can save, compress, and otherwise manipulate it into the file format of our choice. At this point, it is no longer important to us how the image originated since it is now a simple 2D array of numerical data. Before moving on, we need a few more important definitions.

1.4.4 Image Size and Resolution

In the following, we assume rectangular images, and while that is a relatively safe assumption, exceptions do exist. The *size* of an image is determined directly from the *width M* (number of columns) and the *height N* (number of rows) of the image matrix I .

The *resolution* of an image specifies the spatial dimensions of the image in the real world and is given as the number of image elements per measurement; for example, *dots per inch* (dpi) or *lines per inch* (lpi) for print production, or in *pixels per kilometer* for satellite images. In most cases, the resolution of an image is the same in the horizontal and vertical directions, which means that the



1.4 IMAGE ACQUISITION

Fig. 1.5

The transformation of a continuous grayscale image $F(x, y)$ to a discrete digital image $I(u, v)$ (left), image detail (below).



image elements are square. Note that this is not always the case as, for example, the image sensors of most current video cameras have non-square pixels!

The spatial resolution of an image may not be relevant in many basic image processing steps, such as point operations or filters. Precise resolution information is, however, important in cases where geometrical elements such as circles need to be drawn on an image or when distances within an image need to be measured. For these reasons, most image formats and software systems designed for professional applications rely on precise information about image resolution.

1.4.5 Image Coordinate System

In order to know which position on the image corresponds to which image element, we need to impose a coordinate system. Contrary to normal mathematical conventions, in image processing the coordinate system is usually flipped in the vertical direction; that is, the y -coordinate runs from top to bottom and the origin lies in the upper left corner (Fig. 1.6). While this system has no practical or theoretical advantage, and in fact may be a bit confusing in the context of geometrical transformations, it is used almost without exception in imaging software systems. The system supposedly has its roots in the original design of television broadcast systems, where the picture rows are numbered along the vertical deflection of the electron beam, which moves from the top to the bottom of the screen. We start the numbering of rows and columns at zero for practical reasons, since in Java array indexing also begins at zero.

1.4.6 Pixel Values

The information within an image element depends on the data type used to represent it. Pixel values are practically always binary words of length k so that a pixel can represent any of 2^k different values. The value k is called the bit depth (or just “depth”) of the image. The exact bit-level layout of an individual pixel depends on the kind of

1 DIGITAL IMAGES

Fig. 1.6

Image coordinates. In digital image processing, it is common to use a coordinate system where the origin ($u = 0, v = 0$) lies in the upper left corner. The coordinates u, v represent the columns and the rows of the image, respectively. For an image with dimensions $M \times N$, the maximum column number is $u_{\max} = M - 1$ and the maximum row number is $v_{\max} = N - 1$.

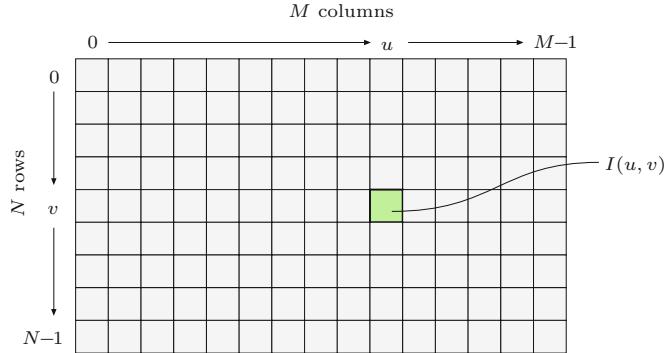


Table 1.1
Bit depths of common image types and typical application domains.

Grayscale (Intensity Images):

Chan.	Bits/Pix.	Range	Use
1	1	[0, 1]	Binary image: document, illustration, fax
1	8	[0, 255]	Universal: photo, scan, print
1	12	[0, 4095]	High quality: photo, scan, print
1	14	[0, 16383]	Professional: photo, scan, print
1	16	[0, 65535]	Highest quality: medicine, astronomy

Color Images:

Chan.	Bits/Pix.	Range	Use
3	24	$[0, 255]^3$	RGB, universal: photo, scan, print
3	36	$[0, 4095]^3$	RGB, high quality: photo, scan, print
3	42	$[0, 16383]^3$	RGB, professional: photo, scan, print
4	32	$[0, 255]^4$	CMYK, digital prepress

Special Images:

Chan.	Bits/Pix.	Range	Use
1	16	$[-32768, 32767]$	Integer values pos./neg., increased range
1	32	$\pm 3.4 \cdot 10^{38}$	Floating-point values: medicine, astronomy
1	64	$\pm 1.8 \cdot 10^{308}$	Floating-point values: internal processing

image; for example, binary, grayscale, or RGB³ color. The properties of some common image types are summarized below (also see Table 1.1).

Grayscale images (intensity images)

The image data in a grayscale image consist of a single channel that represents the intensity, brightness, or density of the image. In most cases, only positive values make sense, as the numbers represent the intensity of light energy or density of film and thus cannot be negative, so typically whole integers in the range $0, \dots, 2^k - 1$ are used. For example, a typical grayscale image uses $k = 8$ bits (1 byte) per pixel and intensity values in the range $0, \dots, 255$, where the value 0 represents the minimum brightness (black) and 255 the maximum brightness (white).

For many professional photography and print applications, as well as in medicine and astronomy, 8 bits per pixel is not sufficient. Image depths of 12, 14, and even 16 bits are often encountered in these

³ Red, green, and blue.

domains. Note that bit depth usually refers to the number of bits used to represent one color component, not the number of bits needed to represent an entire color pixel. For example, an RGB-encoded color image with an 8-bit depth would require 8 bits for each channel for a total of 24 bits, while the same image with a 12-bit depth would require a total of 36 bits.

1.5 IMAGE FILE FORMATS

Binary images

Binary images are a special type of intensity image where pixels can only take on one of two values, black or white. These values are typically encoded using a single bit (0/1) per pixel. Binary images are often used for representing line graphics, archiving documents, encoding fax transmissions, and of course in electronic printing.

Color images

Most color images are based on the primary colors red, green, and blue (RGB), typically making use of 8 bits for each color component. In these color images, each pixel requires $3 \times 8 = 24$ bits to encode all three components, and the range of each individual color component is [0, 255]. As with intensity images, color images with 30, 36, and 42 bits per pixel are commonly used in professional applications. Finally, while most color images contain three components, images with four or more color components are common in most prepress applications, typically based on the subtractive CMYK (Cyan-Magenta-Yellow-Black) color model (see Ch. 12).

Indexed or *palette* images constitute a very special class of color image. The difference between an indexed image and a *true color* image is the number of different colors (fewer for an indexed image) that can be used in a particular image. In an indexed image, the pixel values are only indices (with a maximum of 8 bits) onto a specific table of selected full-color values (see Sec. 12.1.1).

Special images

Special images are required if none of the above standard formats is sufficient for representing the image values. Two common examples of special images are those with negative values and those with floating-point values. Images with negative values arise during image-processing steps, such as filtering for edge detection (see Sec. 6.2.2), and images with floating-point values are often found in medical, biological, or astronomical applications, where extended numerical range and precision are required. These special formats are mostly application specific and thus may be difficult to use with standard image-processing tools.

1.5 Image File Formats

While in this book we almost always consider image data as being already in the form of a 2D array—ready to be accessed by a program—in practice image data must first be loaded into memory from a file. Files provide the essential mechanism for storing,

archiving, and exchanging image data, and the choice of the correct file format is an important decision. In the early days of digital image processing (i.e., before around 1985), most software developers created a new custom file format for almost every new application they developed.⁴ Today there exist a wide range of standardized file formats, and developers can almost always find at least one existing format that is suitable for their application. Using standardized file formats vastly increases the ease with which images can be exchanged and the likelihood that the images will be readable by other software in the long term. Yet for many projects the selection of the right file format is not always simple, and compromises must be made. The following sub-sections outline a few of the typical criteria that need to be considered when selecting an appropriate file format.

1.5.1 Raster versus Vector Data

In the following, we will deal exclusively with file formats for storing *raster images*; that is, images that contain pixel values arranged in a regular matrix using discrete coordinates. In contrast, *vector graphics* represent geometric objects using continuous coordinates, which are only rasterized once they need to be displayed on a physical device such as a monitor or printer.

A number of standardized file formats exist for vector images, such as the ANSI/ISO standard format CGM (Computer Graphics Metafile) and SVG (Scalable Vector Graphics),⁵ as well as proprietary formats such as DXF (Drawing Exchange Format from AutoDesk), AI (Adobe Illustrator), PICT (QuickDraw Graphics Metafile from Apple), and WMF/EMF (Windows Metafile and Enhanced Metafile from Microsoft). Most of these formats can contain both vector data and raster images in the same file. The PS (PostScript) and EPS (Encapsulated PostScript) formats from Adobe as well as the PDF (Portable Document Format) also offer this possibility, although they are typically used for printer output and archival purposes.⁶

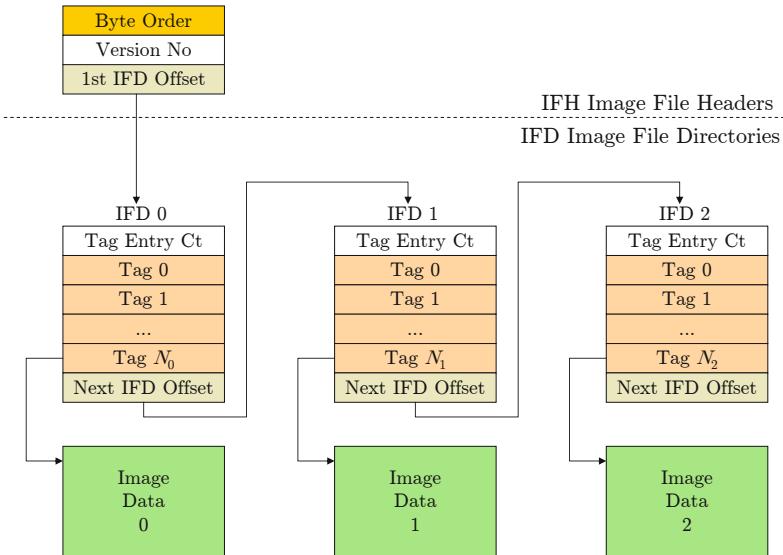
1.5.2 Tagged Image File Format (TIFF)

This is a widely used and flexible file format designed to meet the professional needs of diverse fields. It was originally developed by Aldus and later extended by Microsoft and currently Adobe. The format supports a range of grayscale, indexed, and true color images, but also special image types with large-depth integer and floating-point elements. A TIFF file can contain a number of images with different properties. The TIFF specification provides a range of different compression methods (LZW, ZIP, CCITT, and JPEG) and color spaces,

⁴ The result was a chaotic jumble of incompatible file formats that for a long time limited the practical sharing of images between research groups.

⁵ www.w3.org/TR/SVG/.

⁶ Special variations of PS, EPS, and PDF files are also used as (editable) exchange formats for raster and vector data; for example, both Adobe's Photoshop (Photoshop-EPS) and Illustrator (AI).



1.5 IMAGE FILE FORMATS

Fig. 1.7

Structure of a typical TIFF file. A TIFF file consists of a header and a linked list of image objects, three in this example. Each image object consists of a list of “tags” with their corresponding entries followed by a pointer to the actual image data.

so that it is possible, for example, to store a number of variations of an image in different sizes and representations together in a single TIFF file. The flexibility of TIFF has made it an almost universal exchange format that is widely used in archiving documents, scientific applications, digital photography, and digital video production.

The strength of this image format lies within its architecture (Fig. 1.7), which enables new image types and information blocks to be created by defining new “tags”. In this flexibility also lies the weakness of the format, namely that proprietary tags are not always supported and so the “unsupported tag” error is sometimes still encountered when loading TIFF files. ImageJ also reads only a few uncompressed variations of TIFF formats,⁷ and bear in mind that most popular Web browsers currently do not support TIFF either.

1.5.3 Graphics Interchange Format (GIF)

The Graphics Interchange Format (GIF) was originally designed by CompuServe in 1986 to efficiently encode the rich line graphics used in their dial-up Bulletin Board System (BBS). It has since grown into one of the most widely used formats for representing images on the Web. This popularity is largely due to its early support for indexed color at multiple bit depths, LZW⁸ compression, interlaced image loading, and ability to encode simple animations by storing a number of images in a single file for later sequential display. GIF is essentially an indexed image file format designed for color and grayscale images with a maximum depth of 8 bits and consequently it does not support true color images. It offers efficient support for encoding palettes containing from 2 to 256 colors, one of which can be marked for transparency. GIF supports color tables in the range

⁷ The ImageIO plugin offers support for a wider range of TIFF formats.

⁸ Lempel-Ziv-Welch

of $2, \dots, 256$, enabling pixels to be encoded using fewer bits. As an example, the pixels of an image using 16 unique colors require only 4 bits to store the 16 possible color values $0, \dots, 15$. This means that instead of storing each pixel using 1 byte, as done in other bitmap formats, GIF can encode two 4-bit pixels into each 8-bit byte. This results in a 50% storage reduction over the standard 8-bit indexed color bitmap format.

The GIF file format is designed to efficiently encode “flat” or “iconic” images consisting of large areas of the same color. It uses lossy color quantization (see Ch. 13) as well as lossless LZW compression to efficiently encode large areas of the same color. Despite the popularity of the format, when developing new software, the PNG⁹ format, presented in the next sub-section, should be preferred, as it outperforms GIF by almost every metric.

1.5.4 Portable Network Graphics (PNG)

PNG (pronounced “ping”) was originally developed as a replacement for the GIF file format when licensing issues¹⁰ arose because of its use of LZW compression. It was designed as a universal image format especially for use on the Internet, and, as such, PNG supports three different types of images:

- true color images (with up to 3×16 bits/pixel),
- grayscale images (with up to 16 bits/pixel),
- indexed color images (with up to 256 colors).

Additionally, PNG includes an *alpha* channel for transparency with a maximum depth of 16 bits. In comparison, the transparency channel of a GIF image is only a single bit deep. While the format only supports a single image per file, it is exceptional in that it allows images of up to $2^{30} \times 2^{30}$ pixels. The format supports lossless compression by means of a variation of PKZIP (Phil Katz’s ZIP). No lossy compression is available, as PNG was not designed as a replacement for JPEG. Ultimately, the PNG format meets or exceeds the capabilities of the GIF format in every way except GIF’s ability to include multiple images in a single file to create simple animations. Currently, PNG should be considered the format of choice for representing uncompressed, lossless, true color images for use on the Web.

1.5.5 JPEG

The JPEG standard defines a compression method for continuous grayscale and color images, such as those that would arise from nature photography. The format was developed by the Joint Photographic Experts Group (JPEG)¹¹ with the goal of achieving an average data reduction of a factor of 1:16 and was established in 1990 as ISO Standard IS-10918. Today it is the most widely used image file format. In practice, JPEG achieves, depending on the application, compression in the order of 1 bit per pixel (i.e., a compression factor of around

⁹ Portable network graphics

¹⁰ Unisys’s U.S. LZW Patent No. 4,558,302 expired on June 20, 2003.

¹¹ www.jpeg.org.

1:25) when compressing 24-bit color images to an acceptable quality for viewing. The JPEG standard supports images with up to 256 color components, and what has become increasingly important is its support for CMYK images (see Sec. 12.2.5).

The modular design of the JPEG compression algorithm [163] allows for variations of the “baseline” algorithm; for example, there exists an uncompressed version, though it is not often used. In the case of RGB images, the core of the algorithm consists of three main steps:

1. **Color conversion and down sampling:** A color transformation from RGB into the YC_bC_r space (see Ch. 12, Sec. 12.2.4) is used to separate the actual color components from the brightness Y component. Since the human visual system is less sensitive to rapid changes in color, it is possible to compress the color components more, resulting in a significant data reduction, without a subjective loss in image quality.
2. **Cosine transform and quantization in frequency space:** The image is divided up into a regular grid of 8 blocks, and for each independent block, the frequency spectrum is computed using the discrete cosine transformation (see Ch. 20). Next, the 64 spectral coefficients of each block are quantized into a quantization table. The size of this table largely determines the eventual compression ratio, and therefore the visual quality, of the image. In general, the high frequency coefficients, which are essential for the “sharpness” of the image, are reduced most during this step. During decompression these high frequency values will be approximated by computed values.
3. **Lossless compression:** Finally, the quantized spectral components data stream is again compressed using a lossless method, such as arithmetic or Huffman encoding, in order to remove the last remaining redundancy in the data stream.

The JPEG compression method combines a number of different compression methods and its should not be underestimated. Implementing even the baseline version is nontrivial, so application support for JPEG increased sharply once the Independent JPEG Group (IJG)¹² made available a reference implementation of the JPEG algorithm in 1991. Drawbacks of the JPEG compression algorithm include its limitation to 8-bit images, its poor performance on non-photographic images such as line art (for which it was not designed), its handling of abrupt transitions within an image, and the striking artifacts caused by the 8×8 pixel blocks at high compression rates. Figure 1.9 shows the results of compressing a section of a grayscale image using different quality factors (Photoshop $Q_{\text{JPEG}} = 10, 5, 1$).

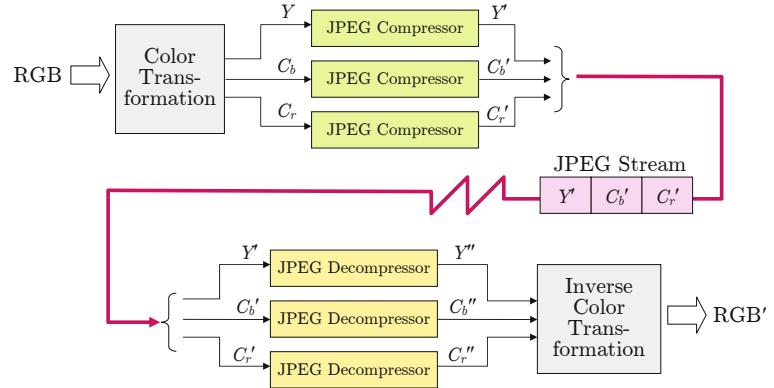
JPEG File Interchange Format (JFIF)

Despite common usage, JPEG is *not* a file format; it is “only” a method of compressing image data. The actual JPEG standard only specifies the JPEG codec (compressor and decompressor) and by de-

¹² www.ijg.org.

1 DIGITAL IMAGES

Fig. 1.8
 JPEG compression of an RGB image. Using a color space transformation, the color components C_b , C_r are separated from the Y luminance component and subjected to a higher rate of compression. Each of the three components are then run independently through the JPEG compression pipeline and are merged into a single JPEG data stream. Decompression follows the same stages in reverse order.



sign leaves the wrapping, or file format, undefined.¹³ What is normally referred to as a *JPEG file* is almost always an instance of a “JPEG File Interchange Format” (JFIF) file, originally developed by Eric Hamilton and the IJG. JFIF specifies a file format based on the JPEG standard by defining the remaining necessary elements of a file format. The JPEG standard leaves some parts of the codec undefined for generality, and in these cases JFIF makes a specific choice. As an example, in step 1 of the JPEG codec, the specific color space used in the color transformation is not part of the JPEG standard, so it is specified by the JFIF standard. As such, the use of different compression ratios for color and luminance is a practical implementation decision specified by JFIF and is not a part of the actual JPEG encoder.

Exchangeable Image File Format (EXIF)

The Exchangeable Image File Format (EXIF) is a variant of the JPEG (JFIF) format designed for storing image data originating on digital cameras, and to that end it supports storing metadata such as the type of camera, date and time, photographic parameters such as aperture and exposure time, as well as geographical (GPS) data. EXIF was developed by the Japan Electronics and Information Technology Industries Association (JEITA) as a part of the DCF¹⁴ guidelines and is used today by practically all manufacturers as the standard format for storing digital images on memory cards. Internally, EXIF uses TIFF to store the metadata information and JPEG to encode a thumbnail preview image. The file structure is designed so that it can be processed by existing JPEG/JFIF readers without a problem.

JPEG-2000

JPEG-2000, which is specified by an ISO-ITU standard (“Coding of Still Pictures”),¹⁵ was designed to overcome some of the better-known weaknesses of the traditional JPEG codec. Among the im-

¹³ To be exact, the JPEG standard only defines how to compress the individual components and the structure of the JPEG stream.

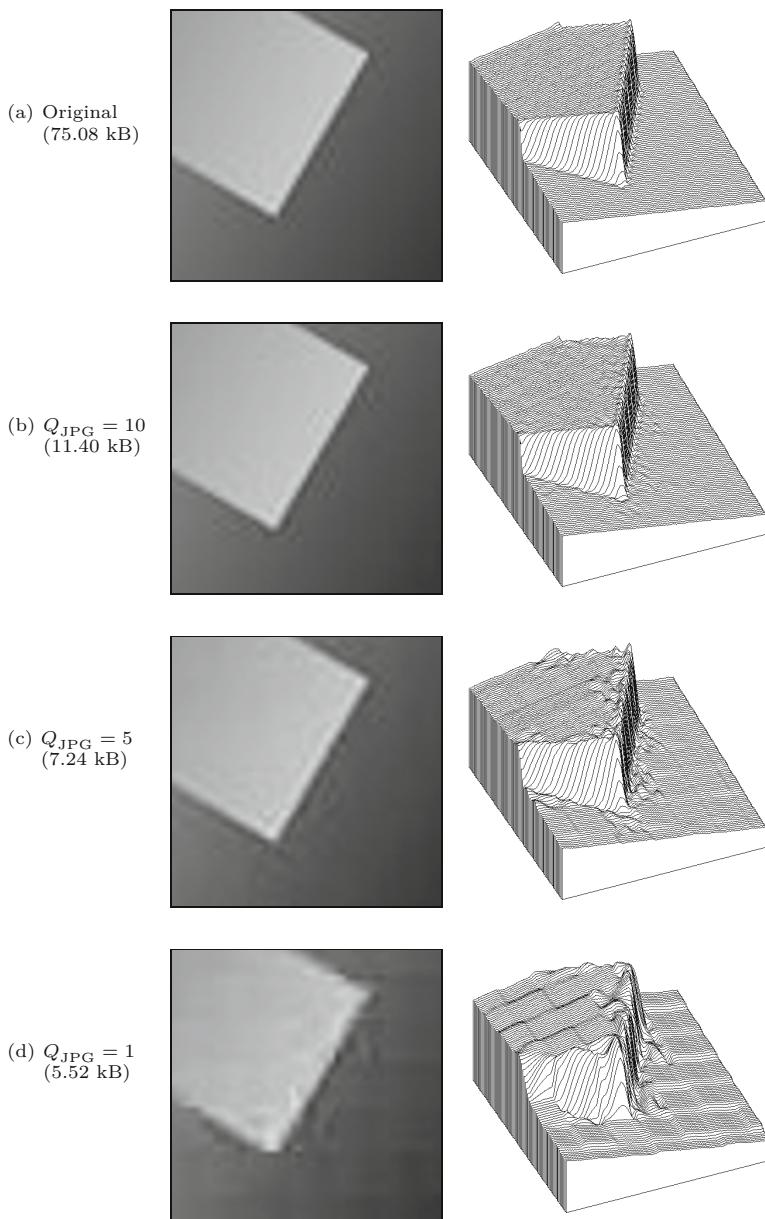
¹⁴ Design Rule for Camera File System.

¹⁵ www.jpeg.org/JPEG2000.htm.

1.5 IMAGE FILE FORMATS

Fig. 1.9

Artifacts arising from JPEG compression. A section of the original image (a) and the results of JPEG compression at different quality factors: $Q_{\text{JPEG}} = 10$ (b), $Q_{\text{JPEG}} = 5$ (c), and $Q_{\text{JPEG}} = 1$ (d). In parentheses are the resulting file sizes for the complete (dimensions 274×274) image.



provements made in JPEG-2000 are the use of larger, 64×64 pixel blocks and replacement of the discrete cosine transform by the *wavelet* transform. These and other improvements enable it to achieve significantly higher compression ratios than JPEG—up to 0.25 bits per pixel on RGB color images. Despite these advantages, JPEG-2000 is supported by only a few image-processing applications and Web browsers.¹⁶

¹⁶ At this time, ImageJ does not offer JPEG-2000 support.

1.5.6 Windows Bitmap (BMP)

The Windows Bitmap (BMP) format is a simple, and under Windows widely used, file format supporting grayscale, indexed, and true color images. It also supports binary images, but not in an efficient manner, since each pixel is stored using an entire byte. Optionally, the format supports simple lossless, run-length-based compression. While BMP offers storage for a similar range of image types as TIFF, it is a much less flexible format.

1.5.7 Portable Bitmap Format (PBM)

The Portable Bitmap Format (PBM) family¹⁷ consists of a series of very simple file formats that are exceptional in that they can be optionally saved in a human-readable text format that can be easily read in a program or simply edited using a text editor. A simple PGM image is shown in Fig. 1.10. The characters P2 in the first line indicate that the image is a PGM (“plain”) file stored in human-readable format. The next line shows how comments can be inserted directly into the file by beginning the line with the # symbol. Line three gives the image’s dimensions, in this case width 17 and height 7, and line four defines the maximum pixel value, in this case 255. The remaining lines give the actual pixel values. This format makes it easy to create and store image data without any explicit imaging API, since it requires only basic text I/O that is available in any programming environment. In addition, the format supports a much more machine-optimized “raw” output mode in which pixel values are stored as bytes. PBM is widely used under Unix and supports the following formats: PBM (*portable bitmap*) for binary *bitmaps*, PGM (*portable graymap*) for grayscale images, and PNM (*portable any map*) for color images. PGM images can be opened by ImageJ.

Fig. 1.10
Example of a PGM file in
human-readable text format
(top) and the correspond-
ing grayscale image (below).



P2
oie.pgm
17 7
255
0 13 13 13 13 13 13 13 0 0 0 0 0 0 0 0
0 13 0 0 0 0 0 13 0 7 7 0 0 81 81 81 81
0 13 0 7 7 7 0 13 0 7 7 0 0 81 0 0 0
0 13 0 7 0 7 0 13 0 7 7 0 0 81 81 81 0
0 13 0 7 7 7 0 13 0 7 7 0 0 81 0 0 0
0 13 0 0 0 0 0 13 0 7 7 0 0 81 81 81 81
0 13 13 13 13 13 13 13 0 0 0 0 0 0 0 0

1.5.8 Additional File Formats

For most practical applications, one of the following file formats is sufficient: TIFF as a universal format supporting a wide variety of uncompressed images and JPEG/JFIF for digital color photos when storage size is a concern, and there is either PNG or GIF for when an image is destined for use on the Web. In addition, there exist

¹⁷ <http://netpbm.sourceforge.net>.

countless other file formats, such as those encountered in legacy applications or in special application areas where they are traditionally used. A few of the more commonly encountered types are:

- **RGB**, a simple format from Silicon Graphics.
- **RAS** (Sun Raster Format), a simple format from Sun Microsystems.
- **TGA** (Truevision Targa File Format), the first 24-bit file format for PCs. It supports numerous image types with 8- to 32-bit depths and is still used in medicine and biology.
- **XBM/XPM** (X-Windows Bitmap/Pixmap), a group of ASCII-encoded formats used in the X-Windows system and similar to PBM/PGM.

1.5.9 Bits and Bytes

Today, opening, reading, and writing image files is mostly carried out by means of existing software libraries. Yet sometimes you still need to deal with the structure and contents of an image file at the byte level, for instance when you need to read an unsupported file format or when you receive a file where the format of the data is unknown.

Big endian and little endian

In the standard model of a computer, a file consists of a simple sequence of 8-bit bytes, and a byte is the smallest entry that can be read or written to a file. In contrast, the image elements as they are stored in memory are usually larger than a byte; for example, a 32-bit `int` value (= 4 bytes) is used for an RGB color pixel. The problem is that storing the four individual bytes that make up the image data can be done in different ways. In order to correctly recreate the original color pixel, we must naturally know the *order* in which bytes in the file are arranged.

Consider, for example, a 32-bit `int` number z with the binary and hexadecimal values¹⁸

$$z = \underbrace{00010010}_{\substack{12_H \\ (\text{MSB})}} \underbrace{00110100}_{\substack{34_H \\ (\text{MSB})}} \underbrace{01010110}_{\substack{56_H \\ (\text{MSB})}} \underbrace{01111000}_{\substack{78_H \\ (\text{LSB})}}_B \equiv 12345678_H, \quad (1.2)$$

then $00010010_B \equiv 12_H$ is the value of the *most significant byte* (MSB) and $01111000_B \equiv 78_H$ the *least significant byte* (LSB). When the individual bytes in the file are arranged in order from MSB to LSB when they are saved, we call the ordering “big endian”, and when in the opposite direction, “little endian”. Thus the 32-bit value z from Eqn. (1.2) could be stored in one of the following two modes:

Ordering	Byte Sequence	1	2	3	4
<i>big endian</i>	MSB → LSB	12_H	34_H	56_H	78_H
<i>little endian</i>	LSB → MSB	78_H	56_H	34_H	12_H

Even though correctly ordering the bytes should essentially be the responsibility of the operating and file systems, in practice it actually

¹⁸ The decimal value of z is 305419896.

1 DIGITAL IMAGES

Table 1.2

Signatures of various image file formats. Most image file formats can be identified by inspecting the first bytes of the file. These byte sequences, or signatures, are listed in hexadecimal (0x..) form and as ASCII text (█ indicates a nonprintable character).

Format	Signature	Format	Signature
PNG	0x89504e47 █PNG	BMP	0x424d BM
JPEG/JFIF	0xffd8ffe0 ████	GIF	0x4749463839 GIF89
TIFF _{little}	0x49492a00 II*█	Photoshop	0x38425053 8BPS
TIFF _{big}	0x4d4d002a MM█*	PS/EPS	0x25215053 %!PS

depends on the architecture of the processor.¹⁹ Processors from the Intel family (e.g., x86, Pentium) are traditionally little endian, and processors from other manufacturers (e.g., IBM, MIPS, Motorola, Sun) are big endian.²⁰ Big endian is also called *network byte ordering* since in the IP protocol the data bytes are arranged in MSB to LSB order during transmission.

To correctly interpret image data with multi-byte pixel values, it is necessary to know the byte ordering used when creating it. In most cases, this is fixed and defined by the file format, but in some file formats, for example TIFF, it is variable and depends on a parameter given in the file header (see [Table 1.2](#)).

File headers and signatures

Practically all image file formats contain a data header consisting of important information about the layout of the image data that follows. Values such as the size of the image and the encoding of the pixels are usually present in the file header to make it easier for programmers to allocate the correct amount of memory for the image. The size and structure of this header are usually fixed, but in some formats, such as TIFF, the header can contain pointers to additional subheaders.

In order to interpret the information in the header, it is necessary to know the file type. In many cases, this can be determined by the *file name extension* (e.g., .jpg or .tif), but since these extensions are not standardized and can be changed at any time by the user, they are not a reliable way of determining the file type. Instead, many file types can be identified by their embedded “signature”, which is often the first 2 bytes of the file. Signatures from a number of popular image formats are given in [Table 1.2](#). Most image formats can be determined by inspecting the first few bytes of the file. These bytes, or signatures, are listed in hexadecimal (0x..) form and as ASCII text. A PNG file always begins with the 4-byte sequence 0x89, 0x50, 0x4e, 0x47, which is the “magic number” 0x89 followed by the ASCII sequence “PNG”. Sometimes the signature not only identifies the type of image file but also contains information about its encoding; for instance, in TIFF the first two characters are either II for “Intel” or MM for “Motorola” and indicate the byte ordering (little endian or big endian, respectively) of the image data in the file.

¹⁹ At least the ordering of the *bits* within a byte is almost universally uniform.

²⁰ In Java, this problem does not arise since internally all implementations of the *Java Virtual Machine* use big endian ordering.

Exercise 1.1. Determine the actual physical measurement in millimeters of an image with 1400 rectangular pixels and a resolution of 72 dpi.

Exercise 1.2. A camera with a focal length of $f = 50$ mm is used to take a photo of a vertical column that is 12 m high and is 95 m away from the camera. Determine its height in the image in mm (a) and the number of pixels (b) assuming the camera has a resolution of 4000 dpi.

Exercise 1.3. The image sensor of a particular digital camera contains 2016×3024 pixels. The geometry of this sensor is identical to that of a traditional 35 mm camera (with an image size of 24×36 mm) except that it is 1.6 times smaller. Compute the resolution of this digital sensor in dpi.

Exercise 1.4. Assume the camera geometry described in Exercise 1.3 combined with a lens with focal length $f = 50$ mm. What amount of blurring (in pixels) would be caused by a uniform, 0.1° horizontal turn of the camera during exposure? Recompute this for $f = 300$ mm. Consider if the extent of the blurring also depends on the distance of the object.

Exercise 1.5. Determine the number of bytes necessary to store an uncompressed binary image of size 4000×3000 pixels.

Exercise 1.6. Determine the number of bytes necessary to store an uncompressed RGB color image of size 640×480 pixels using 8, 10, 12, and 14 bits per color channel.

Exercise 1.7. Given a black and white television with a resolution of 625×512 8-bit pixels and a frame rate of 25 images per second: (a) How many different images can this device ultimately display, and how long would you have to watch it (assuming no sleeping) in order to see every possible image at least once? (b) Perform the same calculation for a color television with 3 \times 8 bits per pixel.

Exercise 1.8. Show that the projection of a 3D straight line in a pinhole camera (assuming perspective projection as defined in Eqn. (1.1)) is again a straight line in the resulting 2D image.

Exercise 1.9. Using Fig. 1.10 as a model, use a text editor to create a PGM file, `disk.pgm`, containing an image of a bright circle. Open your image with ImageJ and then try to find other programs that can open and display the image.

ImageJ

Until a few years ago, the image-processing community was a relatively small group of people who either had access to expensive commercial image-processing tools or, out of necessity, developed their own software packages. Usually such home-brew environments started out with small software components for loading and storing images from and to disk files. This was not always easy because often one had to deal with poorly documented or even proprietary file formats. An obvious (and frequent) solution was to simply design a *new* image file format from scratch, usually optimized for a particular field, application, or even a single project, which naturally led to a myriad of different file formats, many of which did not survive and are forgotten today [163, 168]. Nevertheless, writing software for *converting* between all these file formats in the 1980s and early 1990s was an important business that occupied many people. Displaying images on computer screens was similarly difficult, because there was only marginal support from operating systems, APIs, and display hardware, and capturing images or videos into a computer was close to impossible on common hardware. It thus may have taken many weeks or even months before one could do just elementary things with images on a computer and finally do some serious image processing.

Fortunately, the situation is much different today. Only a few common image file formats have survived (see also Sec. 1.5), which are readily handled by many existing tools and software libraries. Most standard APIs for C/C++, Java, and other popular programming languages already come with at least some basic support for working with images and other types of media data. While there is still much development work going on at this level, it makes our job a lot easier and, in particular, allows us to focus on the more interesting aspects of digital imaging.

2.1 Software for Digital Imaging

Traditionally, software for digital imaging has been targeted at either *manipulating* or *processing* images, either for practitioners and designers or software programmers, with quite different requirements.

Software packages for *manipulating* images, such as Adobe Photoshop, Corel Paint, and others, usually offer a convenient user interface and a large number of readily available functions and tools for working with images interactively. Sometimes it is possible to extend the standard functionality by writing scripts or adding self-programmed components. For example, Adobe provides a special API¹ for programming Photoshop “plugins” in C++, though this is a nontrivial task and certainly too complex for nonprogrammers.

In contrast to the aforementioned category of tools, digital image *processing* software primarily aims at the requirements of algorithm and software developers, scientists, and engineers working with images, where interactivity and ease of use are not the main concerns. Instead, these environments mostly offer comprehensive and well-documented software libraries that facilitate the implementation of new image-processing algorithms, prototypes, and working applications. Popular examples are Khoros/Accusoft,² MatLab,³ ImageMagick,⁴ among many others. In addition to the support for conventional programming (typically with C/C++), many of these systems provide dedicated scripting languages or visual programming aides that can be used to construct even highly complex processes in a convenient and safe fashion.

In practice, image manipulation and image processing are of course closely related. Although Photoshop, for example, is aimed at image manipulation by nonprogrammers, the software itself implements many traditional image-processing algorithms. The same is true for many Web applications using server-side image processing, such as those based on ImageMagick. Thus image processing is really at the base of any image manipulation software and certainly not an entirely different category.

2.2 ImageJ Overview

ImageJ, the software that is used for this book, is a combination of both worlds discussed in the previous section. It offers a set of ready-made tools for viewing and interactive manipulation of images but can also be extended easily by writing new software components in a “real” programming language. ImageJ is implemented entirely in Java and is thus largely platform-independent, running without modification under Windows, MacOS, or Linux. Java’s dynamic execution model allows new modules (“plugins”) to be written as independent pieces of Java code that can be compiled, loaded, and executed “on the fly” in the running system without the need to

¹ www.adobe.com/products/photoshop/.

² www.accusoft.com.

³ www.mathworks.com.

⁴ www.imagemagick.org.

even restart ImageJ. This quick turnaround makes ImageJ an ideal platform for developing and testing new image-processing techniques and algorithms. Since Java has become extremely popular as a first programming language in many engineering curricula, it is usually quite easy for students to get started in ImageJ without having to spend much time learning another programming language. Also, ImageJ is freely available, so students, instructors, and practitioners can install and use the software legally and without license charges on any computer. ImageJ is thus an ideal platform for education and self-training in digital image processing but is also in regular use for serious research and application development at many laboratories around the world, particularly in biological and medical imaging.

ImageJ was (and still *is*) developed by Wayne Rasband [193] at the U.S. National Institutes of Health (NIH), originally as a substitute for its predecessor, NIH-Image, which was only available for the Apple Macintosh platform. The current version of ImageJ, updates, documentation, the complete source code, test images, and a continuously growing collection of third-party plugins can be downloaded from the ImageJ website.⁵ Installation is simple, with detailed instructions available online, in Werner Bailer's programming tutorial [12], and in the authors' *ImageJ Short Reference* [40].

In addition to ImageJ itself there are several popular software projects that build on or extend ImageJ. This includes in particular *Fiji*⁶ (“Fiji Is Just ImageJ”) which offers a consistent collection of numerous plugins, simple installation on various platforms and excellent documentation. All programming examples (plugins) shown in this book should also execute in Fiji without any modifications. Another important development is *ImgLib2*, which is a generic Java API for representing and processing *n*-dimensional images in a consistent fashion. ImgLib2 also provides the underlying data model for *ImageJ2*,⁷ which is a complete reimplementation of ImageJ.

2.2.1 Key Features

As a pure Java application, ImageJ should run on any computer for which a current Java runtime environment (JRE) exists. ImageJ comes with its own Java runtime, so Java need not be installed separately on the computer. Under the usual restrictions, ImageJ can be run as a Java “applet” within a Web browser, though it is mostly used as a stand-alone application. It is sometimes also used on the server side in the context of Java-based Web applications (see [12] for details). In summary, the key features of ImageJ are:

- A set of ready-to-use, interactive tools for creating, visualizing, editing, processing, analyzing, loading, and storing images, with support for several common file formats. ImageJ also provides “deep” 16-bit integer images, 32-bit floating-point images, and image sequences (“stacks”).

2.2 IMAGEJ OVERVIEW



Wayne Rasband (right) at the 1st ImageJ Conference 2006 (picture courtesy of Marc Seil, CRP Henri Tudor, Luxembourg).

⁵ <http://rsb.info.nih.gov/ij/>.

⁶ <http://fiji.sc>.

⁷ <http://imagej.net/ImageJ2>. To avoid confusion, the “classic” ImageJ platform is sometimes referred to as “ImageJ1” or simply “IJ1”.

- A simple plugin mechanism for extending the core functionality of ImageJ by writing (usually small) pieces of Java code. All coding examples shown in this book are based on such plugins.
- A macro language and the corresponding interpreter, which make it easy to implement larger processing blocks by combining existing functions without any knowledge of Java. Macros are not discussed in this book, but details can be found in ImageJ’s online documentation.⁸

2.2.2 Interactive Tools

When ImageJ starts up, it first opens its main window (Fig. 2.1), which includes the following menu entries:

- **File**: for opening, saving, and creating new images.
- **Edit**: for editing and drawing in images.
- **Image**: for modifying and converting images, geometric operations.
- **Process**: for image processing, including point operations, filters, and arithmetic operations between multiple images.
- **Analyze**: for statistical measurements on image data, histograms, and special display formats.
- **Plugin**: for editing, compiling, executing, and managing user-defined plugins.

The current version of ImageJ can open images in several common formats, including TIFF (uncompressed only), JPEG, GIF, PNG, and BMP, as well as the formats DICOM⁹ and FITS,¹⁰ which are popular in medical and astronomical image processing, respectively. As is common in most image-editing programs, all interactive operations are applied to the currently *active* image, i.e., the image most recently selected by the user. ImageJ provides a simple (single-step) “undo” mechanism for most operations, which can also revert modifications effected by user-defined plugins.

2.2.3 ImageJ Plugins

Plugins are small Java modules for extending the functionality of ImageJ by using a simple standardized interface (Fig. 2.2). Plugins can be created, edited, compiled, invoked, and organized through the **Plugin** menu in ImageJ’s main window (Fig. 2.1). Plugins can be grouped to improve modularity, and plugin commands can be arbitrarily placed inside the main menu structure. Also, many of ImageJ’s built-in functions are actually implemented as plugins themselves.

Program structure

Technically speaking, plugins are Java classes that implement a particular interface specification defined by ImageJ. There are two main types of plugins:

⁸ <http://rsb.info.nih.gov/ij/developer/macro/macros.html>.

⁹ Digital Imaging and Communications in Medicine.

¹⁰ Flexible Image Transport System.

2.2 IMAGEJ OVERVIEW

Fig. 2.1

ImageJ main window (under Windows).

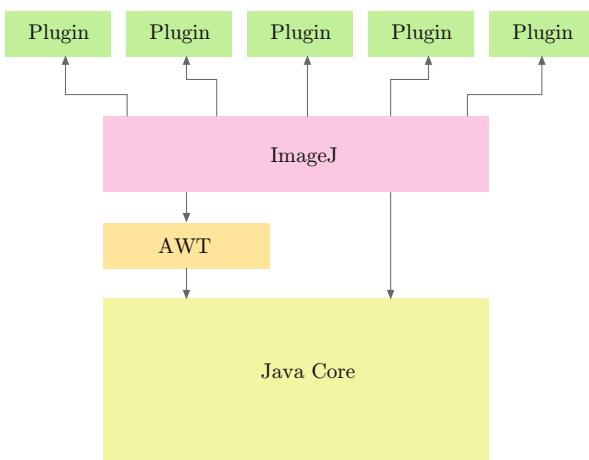
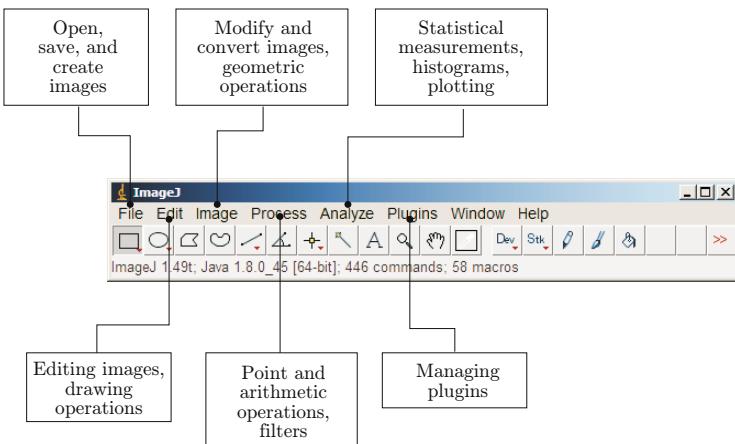


Fig. 2.2

ImageJ software structure (simplified). ImageJ is based on the Java core system and depends in particular upon Java's Advanced Windowing Toolkit (AWT) for the implementation of the user interface and the presentation of image data. Plugins are small Java classes that extend the functionality of the basic ImageJ system.

- **PlugIn:** requires no image to be open to start a plugin.
- **PlugInFilter:** the currently active image is passed to the plugin when started.

Throughout the examples in this book, we almost exclusively use plugins of the second type (i.e., **PlugInFilter**) for implementing image-processing operations. The interface specification requires that any plugin of type **PlugInFilter** must at least implement two methods, **setup()** and **run()**, with the following signatures:

```
int setup (String args, ImagePlus im)
```

When the plugin is started, ImageJ calls this method first to verify that the capabilities of this plugin match the target image. **setup()** returns a vector of binary flags (packaged as a 32-bit **int** value) that describes the plugin's properties.

```
void run (ImageProcessor ip)
```

This method does the actual work for this plugin. It is passed a single argument **ip**, an object of type **ImageProcessor**, which contains the image to be processed and all relevant information

about it. The `run()` method returns no result value (`void`) but may modify the passed image and create new images.

2.2.4 A First Example: Inverting an Image

Let us look at a real example to quickly illustrate this mechanism. The task of our first plugin is to invert any 8-bit grayscale image to turn a positive image into a negative. As we shall see later, inverting the intensity of an image is a typical *point operation*, which is discussed in detail in Chapter 4. In ImageJ, 8-bit grayscale images have pixel values ranging from 0 (black) to 255 (white), and we assume that the width and height of the image are M and N , respectively. The operation is very simple: the value of each image pixel $I(u, v)$ is replaced by its inverted value,

$$I(u, v) \leftarrow 255 - I(u, v),$$

for all image coordinates (u, v) , with $u = 0, \dots, M-1$ and $v = 0, \dots, N-1$.

2.2.5 Plugin My_Inverter_A (using PlugInFilter)

We decide to name our first plugin “`My_Inverter_A`”, which is both the name of the Java class and the name of the source file¹¹ that contains it (see Prog. 2.1). The underscore characters (“`_`”) in the name cause ImageJ to recognize this class as a plugin and to insert it automatically into the menu list at startup. The Java source code in file `My_Inverter.java` contains a few `import` statements, followed by the definition of the class `My_Inverter`, which implements the `PlugInFilter` interface (because it will be applied to an existing image).

The `setup()` method

When a plugin of type `PlugInFilter` is executed, ImageJ first invokes its `setup()` method to obtain information about the plugin itself. In this example, `setup()` only returns the value `DOES_8G` (a static `int` constant specified by the `PlugInFilter` interface), indicating that this plugin can handle 8-bit grayscale images. The parameters `arg` and `im` of the `setup()` method are not used in this example (see also Exercise 2.7).

The `run()` method

As mentioned already, the `run()` method of a `PlugInFilter` plugin receives an object (`ip`) of type `ImageProcessor`, which contains the image to be processed and all relevant information about it. First, we use the `ImageProcessor` methods `getWidth()` and `getHeight()` to query the size of the image referenced by `ip`. Then we use two nested `for` loops (with loop variables `u`, `v` for the horizontal and vertical coordinates, respectively) to iterate over all image pixels. For reading and writing the pixel values, we use two additional methods of the class `ImageProcessor`:

¹¹ File `My_Inverter_A.java`.

```

1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4
5 public class My_Inverter_A implements PlugInFilter {
6
7     public int setup(String args, ImagePlus im) {
8         return DOES_8G; // this plugin accepts 8-bit grayscale images
9     }
10
11    public void run(ImageProcessor ip) {
12        int M = ip.getWidth();
13        int N = ip.getHeight();
14
15        // iterate over all image coordinates (u,v)
16        for (int u = 0; u < M; u++) {
17            for (int v = 0; v < N; v++) {
18                int p = ip.getPixel(u, v);
19                ip.putPixel(u, v, 255 - p);
20            }
21        }
22    }
23
24 }

```

2.2 IMAGEJ OVERVIEW

Prog. 2.1

ImageJ plugin for inverting 8-bit grayscale images. This plugin implements the interface `PlugInFilter` and defines the required methods `setup()` and `run()`. The target image is received by the `run()` method as an instance of type `ImageProcessor`. ImageJ assumes that the plugin modifies the supplied image and automatically redisperses it after the plugin is executed. Program 2.2 shows an alternative implementation that is based on the `PlugIn` interface.

`int getPixel (int u, int v)`

Returns the pixel value at the given position or zero if (u, v) is outside the image bounds.

`void putPixel (int u, int v, int a)`

Sets the pixel value at position (u, v) to the new value a . Does nothing if (u, v) is outside the image bounds.

Both methods check the supplied image coordinates and pixel values to avoid unwanted errors. While this makes them more or less fail-safe it also makes them slow. If we are sure that no coordinates outside the image bounds are ever accessed (as in `My_Inverter` in Prog. 2.1) and the inserted pixel values are guaranteed not to exceed the image processor's range, we can use the significantly faster methods `get()` and `set()` in place of `getPixel()` and `putPixel()`, respectively. The most efficient way to process the image is to avoid read/write methods altogether and directly access the elements of the associated (1D) pixel array. Details on these and other methods can be found in the ImageJ API documentation.¹²

2.2.6 Plugin `My_Inverter_B` (using `PlugIn`)

Program 2.2 shows an alternative implementation of the inverter plugin based on ImageJ's `PlugIn` interface, which requires a `run()` method only. In this case the reference to the current image is not supplied directly but is obtained by invoking the (static) method

¹² <http://rsbweb.nih.gov/ij/developer/api/index.html>.

2 IMAGEJ

Prog. 2.2

Alternative implementation of the inverter plugin, based on ImageJ's `PlugIn` interface.

In contrast to Prog. 2.1 this plugin has no `setUp()` method but defines a `run()` method only. The current image (`im`) is obtained as an instance of class `ImagePlus` by invoking the `IJ.getImage()` method. After checking for the proper image type the associated `ImageProcessor` (`ip`) is retrieved from `im`. The parameter string (`args`) is not used in this example. The remaining parts of the plugin are identical to Prog. 2.1, except that the (slightly faster) pixel access methods `get()` and `set()` are used. Also note that the modified image is not re-displayed automatically but by an explicit call to `updateAndDraw()`.

```
1 import ij.IJ;
2 import ij.ImagePlus;
3 import ij.plugin.PlugIn;
4 import ij.process.ImageProcessor;
5
6 public class My_Inverter_B implements PlugIn {
7
8     public void run(String args) {
9         ImagePlus im = IJ.getImage();
10
11     if (im.getType() != ImagePlus.GRAY8) {
12         IJ.error("8-bit grayscale image required");
13         return;
14     }
15
16     ImageProcessor ip = im.getProcessor();
17     int M = ip.getWidth();
18     int N = ip.getHeight();
19
20     // iterate over all image coordinates (u,v)
21     for (int u = 0; u < M; u++) {
22         for (int v = 0; v < N; v++) {
23             int p = ip.get(u, v);
24             ip.set(u, v, 255 - p);
25         }
26     }
27
28     im.updateAndDraw();    // redraw the modified image
29 }
30 }
```

`IJ.getImage()`. If no image is currently open, `getImage()` automatically displays an error message and aborts the plugin. However, the subsequent test for the correct image type (GRAY8) and the corresponding error handling must be performed explicitly. The `run()` method accepts a single string argument that can be used to pass arbitrary information for controlling the plugin.

2.2.7 When to use `PlugIn` or `PlugInFilter`?

The choice of `PlugIn` or `PlugInFilter` is mostly a matter of taste, since both versions have their advantages and disadvantages. As a rule of thumb, we use the `PlugIn` type for tasks that do not require any image to be open but for tasks that create, load, or record images or perform operations without any images. Otherwise, if one or more open images should be processed, `PlugInFilter` is the preferred choice and thus almost all plugins in this book are of type `PlugInFilter`.

Editing, compiling, and executing the plugin

The Java source file for our plugin should be stored in directory `<iij>/plugins/`¹³ or an immediate subdirectory. New plugin files

¹³ `<iij>` denotes ImageJ's installation directory.

can be created with ImageJ's **Plugins** ▷ **New...** menu. ImageJ even provides a built-in Java editor for writing plugins, which is available through the **Plugins** ▷ **Edit...** menu but unfortunately is of little use for serious programming. A better alternative is to use a modern editor or a professional Java programming environment, such as Eclipse,¹⁴ NetBeans,¹⁵ or JBuilder,¹⁶ all of which are freely available.

For compiling plugins (to Java bytecode), ImageJ comes with its own Java compiler as part of its runtime environment. To compile and execute the new plugin, simply use the menu

Plugins ▷ **Compile and Run...**

Compilation errors are displayed in a separate log window. Once the plugin is compiled, the corresponding `.class` file is automatically loaded and the plugin is applied to the currently active image. An error message is displayed if no images are open or if the current image cannot be handled by that plugin.

At startup, ImageJ automatically loads all correctly named plugins found in the `<ij>/plugins/` directory (or any immediate subdirectory) and installs them in its **Plugins** menu. These plugins can be executed immediately without any recompilation. References to plugins can also be placed manually with the

Plugins ▷ **Shortcuts** ▷ **Install Plugin...**

command at any other position in the ImageJ menu tree. Sequences of plugin calls and other ImageJ commands may be recorded as macro programs with **Plugins** ▷ **Macros** ▷ **Record**.

Displaying and “undoing” results

Our first plugins in Prog. 2.1–2.2 did not create a new image but “destructively” modified the target image. This is not always the case, but plugins can also create additional images or compute only statistics, without modifying the original image at all. It may be surprising, though, that our plugin contains no commands for displaying the modified image. This is done automatically by ImageJ whenever it can be assumed that the image passed to a plugin was modified.¹⁷ In addition, ImageJ automatically makes a copy (“snapshot”) of the image before passing it to the `run()` method of a `PlugInFilter`-type plugin. This feature makes it possible to restore the original image (with the **Edit** ▷ **Undo** menu) after the plugin has finished without any explicit precautions in the plugin code.

Logging and debugging

The usual console output from Java via `System.out` is not available in ImageJ by default. Instead, a separate logging window can be used which facilitates simple text output by the method

`IJ.log(String s).`

¹⁴ www.eclipse.org.

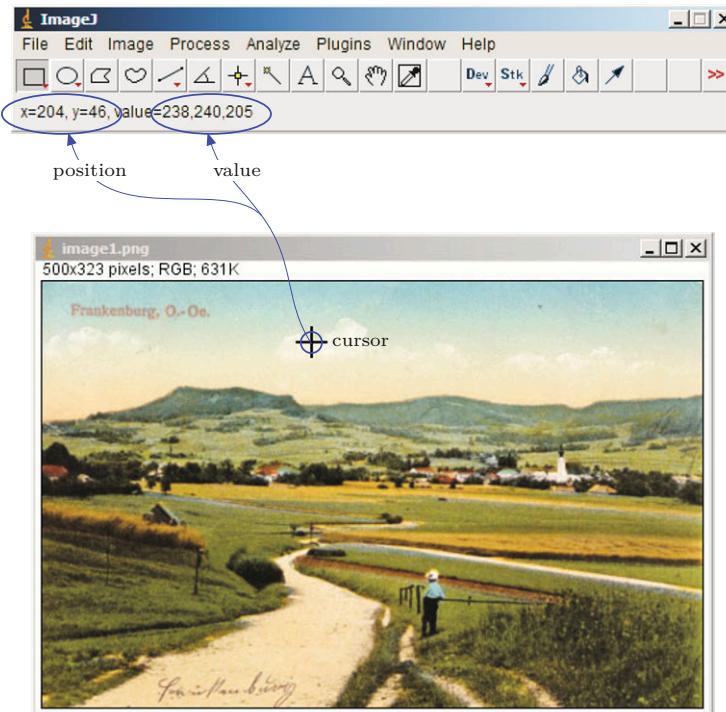
¹⁵ www.netbeans.org.

¹⁶ www.borland.com.

¹⁷ No automatic redisplay occurs if the `NO_CHANGES` flag is set in the return value of the plugin's `setup()` method.

2 IMAGEJ

Fig. 2.3
Information displayed in ImageJ's main window is extremely helpful for debugging image-processing operations. The current cursor position is displayed in pixel coordinates unless the associated image is spatially calibrated. The way pixel *values* are displayed depends on the image type; in the case of a color image (as shown here) integer RGB component values are shown.



Such calls may be placed at any position in the plugin code for quick and simple debugging at runtime. However, because of the typically large amounts of data involved, they should be used with caution in real image-processing operations. Particularly, when placed in the body of inner processing loops that could execute millions of times, text output may produce an enormous overhead compared to the time used for the actual calculations.

ImageJ itself does not offer much support for “real” debugging, i.e., for setting breakpoints, inspecting local variables etc. However, it is possible to launch ImageJ from within a programming environment (IDE) such as Eclipse or *Netbeans* and then use all debugging options that the given environment provides.¹⁸ According to experience, this is only needed in rare and exceptionally difficult situations. In most cases, inspection of pixel values displayed in ImageJ's main window (see Fig. 2.3) is much simpler and more effective. In general, many errors (in particular those related to image coordinates) can be easily avoided by careful planning in advance.

2.2.8 Executing ImageJ “Commands”

If possible, it is wise in most cases to re-use existing (and extensively tested) functionality instead of re-implementing it oneself. In particular, the Java library that comes with ImageJ covers many standard image-processing operations, many of which are used throughout this

¹⁸ For details see the “HowTo” section at <http://imagejdocu.tudor.lu>.

```

1 import ij.IJ;
2 import ij.ImagePlus;
3 import ij.plugin.PlugIn;
4
5 public class Run_Command_From_Plugin implements PlugIn {
6
7     public void run(String args) {
8         ImagePlus im = IJ.getImage();
9         IJ.run(im, "Invert", ""); // run the "Invert" command on im
10        // ... continue with this plugin
11    }
12 }

```

2.2 IMAGEJ OVERVIEW

Prog. 2.3

Executing the ImageJ command “Invert” within a Java plugin of type `PlugIn`.

```

1 public class Run_Command_From_PluginFilter implements
2     PlugInFilter {
3
4     ImagePlus im;
5
6     public int setup(String args, ImagePlus im) {
7         this.im = im;
8         return DOES_ALL;
9     }
10
11    public void run(ImageProcessor ip) {
12        im.unlock();           // unlock im to run other commands
13        IJ.run(im, "Invert", ""); // run "Invert" command on im
14        im.lock();            // lock im again (to be safe)
15        // ... continue with this plugin
16    }
17 }

```

Prog. 2.4

Executing the ImageJ command “Invert” within a Java plugin of type `PlugInFilter`. In this case the current image is automatically locked during plugin execution, such that no other operation may be applied to it. However, the image can be temporarily unlocked by calling `unlock()` and `lock()`, respectively, to run the external command.

book. Additional classes and methods for specific operations are contained in the associated (`imagingbook`) library.

In the context of ImageJ, the term “command” refers to any composite operation implemented as a (Java) plugin, a macro command or as a script.¹⁹ ImageJ itself includes numerous commands which can be listed with the menu `Plugins > Utilities > Find Commands....` They are usually referenced “by name”, i.e., by a unique string. For example, the standard operation for inverting an image (`Edit > Invert`) is implemented by the Java class `ij.plugin.filter.Filters` (with the argument `"invert"`).

An existing command can also be executed from within a Java plugin with the method `IJ.run()`, as demonstrated for the “Invert” command in Prog. 2.3. Some caution is required with plugins of type `PlugInFilter`, since these lock the current image during execution, such that no other operation can be applied to it. The example in Prog. 2.4 shows how this can be resolved by a pair of calls to `unlock()` and `lock()`, respectively, to temporarily release the current image.

A convenient tool for putting together complex commands is ImageJ’s built-in *Macro Recorder*. Started with `Plugins > Macros >`

¹⁹ Scripting languages for ImageJ currently include *JavaScript*, *BeanShell*, and *Python*.

Record..., it logs all subsequent commands in a text file for later use. It can be set up to record commands in various modes, including *Java*, *JavaScript*, *BeanShell*, or ImageJ macro code. Of course it does record the application of self-defined plugins as well.

2.3 Additional Information on ImageJ and Java

In the following chapters, we mostly use concrete plugins and Java code to describe algorithms and data structures. This not only makes these examples immediately applicable, but they should also help in acquiring additional skills for using ImageJ in a step-by-step fashion. To keep the text compact, we often describe only the `run()` method of a particular plugin and additional class and method definitions if they are relevant in the given context. The complete source code for these examples can of course be downloaded from the book’s supporting website.²⁰

2.3.1 Resources for ImageJ

The complete and most current API reference, including source code, tutorials, and many example plugins, can be found on the official ImageJ website. Another great source for any serious plugin programming is the tutorial by Werner Bailer [12].

2.3.2 Programming with Java

While this book does not require extensive Java skills from its readers, some elementary knowledge is essential for understanding or extending the given examples. There is a huge and still-growing number of introductory textbooks on Java, such as [8, 29, 66, 70, 208] and many others. For readers with programming experience who have not worked with Java before, we particularly recommend some of the tutorials on Oracle’s Java website.²¹ Also, in Appendix F of this book, readers will find a small compilation of specific Java topics that cause frequent problems or programming errors.

2.4 Exercises

Exercise 2.1. Install the current version of ImageJ on your computer and make yourself familiar with the built-in commands (open, convert, edit, and save images).

Exercise 2.2. Write a new ImageJ plugin that reflects a grayscale image horizontally (or vertically) using `My_Inverter.java` (Prog. 2.1) as a template. Test your new plugin with appropriate images of different sizes (odd, even, extremely small) and inspect the results carefully.

²⁰ www.imagingbook.com.

²¹ <http://docs.oracle.com/javase/>.

Exercise 2.3. The `run()` method of plugin `Inverter_Plugin_A` (see Prog. 2.1) iterates over all pixels of the given image. Find out in which order the pixels are visited: along the (horizontal) lines or along the (vertical) columns? Make a drawing to illustrate this process.

2.4 EXERCISES

Exercise 2.4. Create an ImageJ plugin for 8-bit grayscale images of arbitrary size that paints a white frame (with pixel value 255) 10 pixels wide *into* the image (without increasing its size). Make sure this plugin also works for very small images.

Exercise 2.5. Create a plugin for 8-bit grayscale images that calculates and prints the result (with `IJ.log()`). Use a variable of type `int` or `long` for accumulating the pixel values. What is the maximum image size for which we can be certain that the result of summing with an `int` variable is correct?

Exercise 2.6. Create a plugin for 8-bit grayscale images that calculates and prints the minimum and maximum pixel values in the current image (with `IJ.log()`). Compare your output to the results obtained with `Analyze > Measure`.

Exercise 2.7. Write a new ImageJ plugin that shifts an 8-bit grayscale image horizontally and circularly until the original state is reached again. To display the modified image after each shift, a reference to the corresponding `ImagePlus` object is required (`ImageProcessor` has no display methods). The `ImagePlus` object is only accessible to the plugin's `setup()` method, which is automatically called before the `run()` method. Modify the definition in Prog. 2.1 to keep a reference and to redraw the `ImagePlus` object as follows:

```
public class XY_Plugin implements PlugInFilter {
    ImagePlus im;           // new variable!

    public int setup(String args, ImagePlus im) {
        this.im = im;      // reference to the associated ImagePlus object
        return DOES_8G;
    }

    public void run(ImageProcessor ip) {
        // ... modify ip
        im.updateAndDraw(); // redraw the associated ImagePlus object
        // ...
    }
}
```

Histograms and Image Statistics

Histograms are used to depict image statistics in an easily interpreted visual format. With a histogram, it is easy to determine certain types of problems in an image, for example, it is simple to conclude if an image is properly exposed by visual inspection of its histogram. In fact, histograms are so useful that modern digital cameras often provide a real-time histogram overlay on the viewfinder (Fig. 3.1) to help prevent taking poorly exposed pictures. It is important to catch errors like this at the image capture stage because poor exposure results in a permanent loss of information, which it is not possible to recover later using image-processing techniques. In addition to their usefulness during image capture, histograms are also used later to improve the visual appearance of an image and as a “forensic” tool for determining what type of processing has previously been applied to an image. The final part of this chapter shows how to calculate simple image statistics from the original image, its histogram, or the so-called integral image.



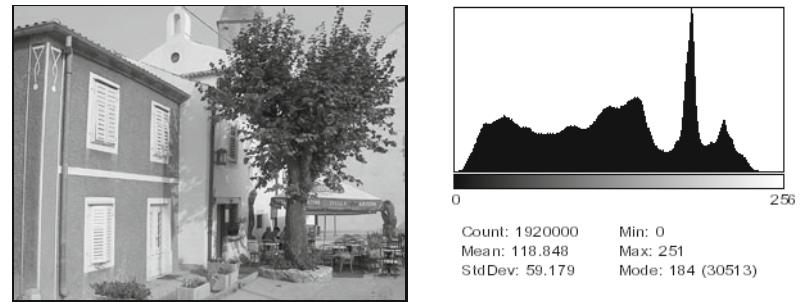
Fig. 3.1
Digital camera back display showing the associated RGB histograms.

3.1 What is a Histogram?

Histograms in general are frequency distributions, and histograms of images describe the frequency of the intensity values that occur in an image. This concept can be easily explained by considering an old-fashioned grayscale image like the one shown in Fig. 3.2.

Fig. 3.2

An 8-bit grayscale image and a histogram depicting the frequency distribution of its 256 intensity values.



The histogram \mathbf{h} for a grayscale image I with intensity values in the range $I(u, v) \in [0, K-1]$ holds exactly K entries, where $K = 2^8 = 256$ for a typical 8-bit grayscale image. Each single histogram entry is defined as

$\mathbf{h}(i)$ = the number of pixels in I with the intensity value i ,

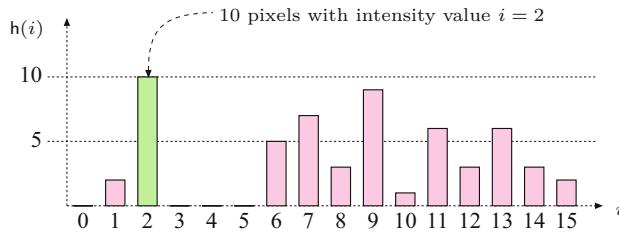
for all $0 \leq i < K$. More formally stated,¹

$$\mathbf{h}(i) = \text{card}\{(u, v) \mid I(u, v) = i\}. \quad (3.1)$$

Therefore, $\mathbf{h}(0)$ is the number of pixels with the value 0, $\mathbf{h}(1)$ the number of pixels with the value 1, and so forth. Finally, $\mathbf{h}(255)$ is the number of all white pixels with the maximum intensity value $255 = K-1$. The result of the histogram computation is a 1D vector \mathbf{h} of length K . Figure 3.3 gives an example for an image with $K = 16$ possible intensity values.

Fig. 3.3

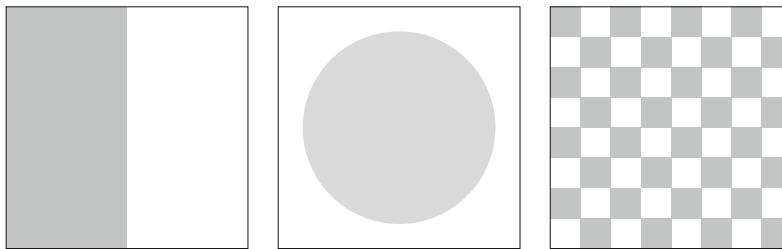
Histogram vector for an image with $K = 16$ possible intensity values. The indices of the vector element $i = 0 \dots 15$ represent intensity values. The value of 10 at index 2 means that the image contains 10 pixels of intensity value 2.



$\mathbf{h}(i)$	0	2	10	0	0	0	5	7	3	9	1	6	3	6	3	2
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Since the histogram encodes no information about *where* each of its individual entries originated in the image, it contains no information about the spatial arrangement of pixels in the image. This

¹ $\text{card}\{\dots\}$ denotes the number of elements (“cardinality”) in a set (see also Sec. A.1 in the Appendix).



3.2 INTERPRETING HISTOGRAMS

Fig. 3.4
Three very different images with identical histograms.

is intentional, since the main function of a histogram is to provide statistical information, (e.g., the distribution of intensity values) in a compact form. Is it possible to reconstruct an image using only its histogram? That is, can a histogram be somehow “inverted”? Given the loss of spatial information, in all but the most trivial cases, the answer is no. As an example, consider the wide variety of images you could construct using the same number of pixels of a specific value. These images would appear different but have exactly the same histogram (Fig. 3.4).

3.2 Interpreting Histograms

A histogram depicts problems that originate during image acquisition, such as those involving contrast and dynamic range, as well as artifacts resulting from image-processing steps that were applied to the image. Histograms are often used to determine if an image is making effective use of its intensity range (Fig. 3.5) by examining the size and uniformity of the histogram’s distribution.

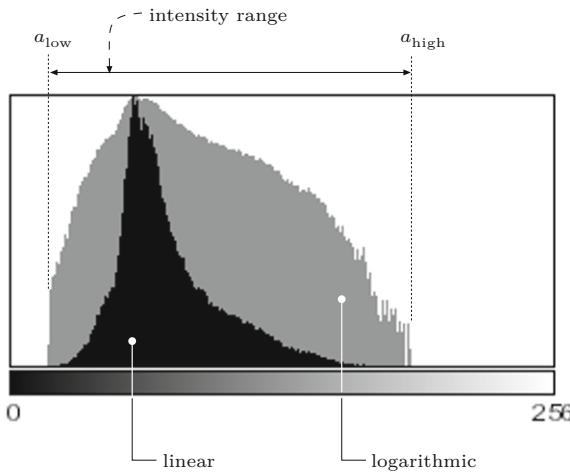


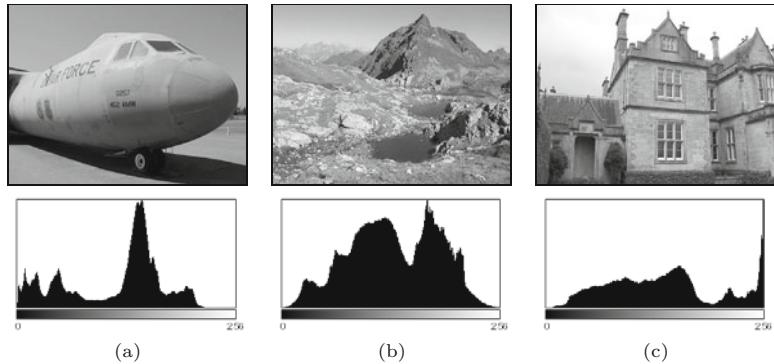
Fig. 3.5
Effective intensity range. The graph depicts the frequencies of pixel values *linearly* (black bars) and *logarithmically* (gray bars). The logarithmic form makes even relatively low occurrences, which can be very important in the image, readily apparent.

3.2.1 Image Acquisition

Histograms make typical exposure problems readily apparent. As an example, a histogram where a large section of the intensity range at one end is largely unused while the other end is crowded with

Fig. 3.6

Exposure errors are readily apparent in histograms. Underexposed (a), properly exposed (b), and overexposed (c) photographs.



high-value peaks (Fig. 3.6) is representative of an improperly exposed image.

Contrast

Contrast is understood as the range of intensity values *effectively* used within a given image, that is the difference between the image's maximum and minimum pixel values. A full-contrast image makes effective use of the entire range of available intensity values from $a = a_{\min}, \dots, a_{\max}$ with $a_{\min} = 0$, $a_{\max} = K - 1$ (black to white). Using this definition, image contrast can be easily read directly from the histogram. Figure 3.7 illustrates how varying the contrast of an image affects its histogram.

Dynamic range

The dynamic range of an image is, in principle, understood as the number of *distinct* pixel values in an image. In the ideal case, the dynamic range encompasses all K usable pixel values, in which case the value range is completely utilized. When an image has an available range of contrast $a = a_{\text{low}}, \dots, a_{\text{high}}$, with

$$a_{\min} < a_{\text{low}} \quad \text{and} \quad a_{\text{high}} < a_{\max},$$

then the maximum possible dynamic range is achieved when all the intensity values lying in this range are utilized (i.e., appear in the image; Fig. 3.8).

While the contrast of an image can be increased by transforming its existing values so that they utilize more of the underlying value range available, the dynamic range of an image can only be increased by introducing artificial (that is, not originating with the image sensor) values using methods such as interpolation (see Ch. 22). An image with a high dynamic range is desirable because it will suffer less image-quality degradation during image processing and compression. Since it is not possible to increase dynamic range after image acquisition in a practical way, professional cameras and scanners work at depths of more than 8 bits, often 12–14 bits per channel, in order to provide high dynamic range at the acquisition stage. While most output devices, such as monitors and printers, are unable to actually reproduce more than 256 different shades, a high dynamic range is always beneficial for subsequent image processing or archiving.



3.2 INTERPRETING HISTOGRAMS

Fig. 3.7

How changes in contrast affect the histogram: low contrast (a), normal contrast (b), high contrast (c).

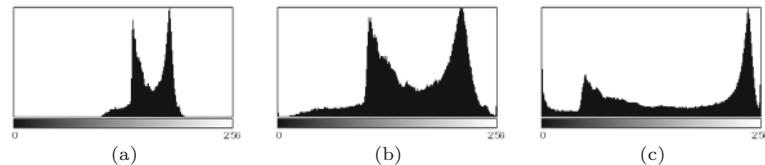
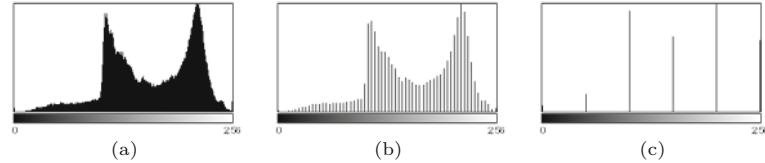


Fig. 3.8

How changes in dynamic range affect the histogram: high dynamic range (a), low dynamic range with 64 intensity values (b), extremely low dynamic range with only 6 intensity values (c).



3.2.2 Image Defects

Histograms can be used to detect a wide range of image defects that originate either during image acquisition or as the result of later image processing. Since histograms always depend on the visual characteristics of the scene captured in the image, no single “ideal” histogram exists. While a given histogram may be optimal for a specific scene, it may be entirely unacceptable for another. As an example, the ideal histogram for an astronomical image would likely be very different from that of a good landscape or portrait photo. Nevertheless, there are some general rules; for example, when taking a landscape image with a digital camera, you can expect the histogram to have evenly distributed intensity values and no isolated spikes.

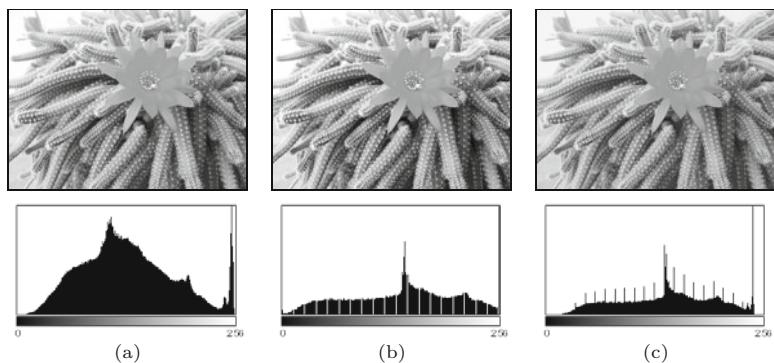
Saturation

Ideally the contrast range of a sensor, such as that used in a camera, should be greater than the range of the intensity of the light that it receives from a scene. In such a case, the resulting histogram will

be smooth at both ends because the light received from the very bright and the very dark parts of the scene will be less than the light received from the other parts of the scene. Unfortunately, this ideal is often not the case in reality, and illumination outside of the sensor's contrast range, arising for example from glossy highlights and especially dark parts of the scene, cannot be captured and is lost. The result is a histogram that is saturated at one or both ends of its range. The illumination values lying outside of the sensor's range are mapped to its minimum or maximum values and appear on the histogram as significant spikes at the tail ends. This typically occurs in an under- or overexposed image and is generally not avoidable when the inherent contrast range of the scene exceeds the range of the system's sensor ([Fig. 3.9\(a\)](#)).

Fig. 3.9

Effect of image capture errors on histograms: saturation of high intensities (a), histogram gaps caused by a slight increase in contrast (b), and histogram spikes resulting from a reduction in contrast (c).



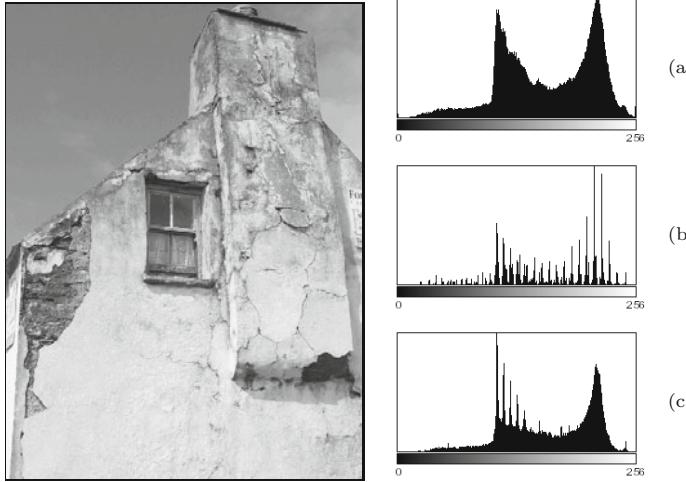
Spikes and gaps

As discussed already, the intensity value distribution for an unprocessed image is generally smooth; that is, it is unlikely that isolated spikes (except for possible saturation effects at the tails) or gaps will appear in its histogram. It is also unlikely that the count of any given intensity value will differ greatly from that of its neighbors (i.e., it is locally smooth). While artifacts like these are observed very rarely in original images, they will often be present after an image has been manipulated, for instance, by changing its contrast. Increasing the contrast (see Ch. 4) causes the histogram lines to separate from each other and, due to the discrete values, gaps are created in the histogram ([Fig. 3.9\(b\)](#)). Decreasing the contrast leads, again because of the discrete values, to the merging of values that were previously distinct. This results in increases in the corresponding histogram entries and ultimately leads to highly visible spikes in the histogram ([Fig. 3.9\(c\)](#)).²

Impacts of image compression

Image compression also changes an image in ways that are immediately evident in its histogram. As an example, during GIF compression, an image's dynamic range is reduced to only a few intensities

² Unfortunately, these types of errors are also caused by the internal contrast “optimization” routines of some image-capture devices, especially consumer-type scanners.



3.3 CALCULATING HISTOGRAMS

Fig. 3.10

Color quantization effects resulting from GIF conversion. The original image converted to a 256 color GIF image (left). Original histogram (a) and the histogram after GIF conversion (b). When the RGB image is scaled by 50%, some of the lost colors are recreated by interpolation, but the results of the GIF conversion remain clearly visible in the histogram (c).

or colors, resulting in an obvious line structure in the histogram that cannot be removed by subsequent processing (Fig. 3.10). Generally, a histogram can quickly reveal whether an image has ever been subjected to color quantization, such as occurs during conversion to a GIF image, even if the image has subsequently been converted to a full-color format such as TIFF or JPEG.

Figure 3.11 illustrates what occurs when a simple line graphic with only two gray values (128, 255) is subjected to a compression method such as JPEG, that is not designed for line graphics but instead for natural photographs. The histogram of the resulting image clearly shows that it now contains a large number of gray values that were not present in the original image, resulting in a poor-quality image³ that appears dirty, fuzzy, and blurred.

3.3 Calculating Histograms

Computing the histogram of an 8-bit grayscale image containing intensity values between 0 and 255 is a simple task. All we need is a set of 256 counters, one for each possible intensity value. First, all counters are initialized to zero. Then we iterate through the image I , determining the pixel value p at each location (u, v) , and incrementing the corresponding counter by one. At the end, each counter will contain the number of pixels in the image that have the corresponding intensity value.

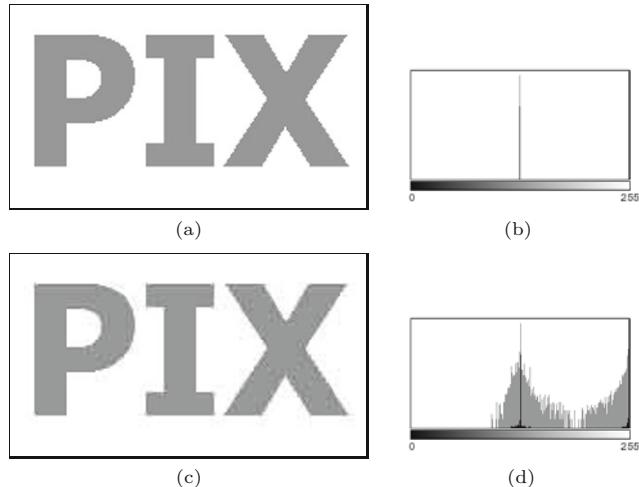
An image with K possible intensity values requires exactly K counter variables; for example, since an 8-bit grayscale image can contain at most 256 different intensity values, we require 256 counters. While individual counters make sense conceptually, an actual

³ Using JPEG compression on images like this, for which it was not designed, is one of the most egregious of imaging errors. JPEG is designed for photographs of natural scenes with smooth color transitions, and using it to compress iconic images with large areas of the same color results in strong visual artifacts (see, e.g., Fig. 1.9 on p. 17).

3 HISTOGRAMS AND IMAGE STATISTICS

Fig. 3.11

Effects of JPEG compression. The original image (a) contained only two different gray values, as its histogram (b) makes readily apparent. JPEG compression, a poor choice for this type of image, results in numerous additional gray values, which are visible in both the resulting image (c) and its histogram (d). In both histograms, the linear frequency (black bars) and the logarithmic frequency (gray bars) are shown.



Prog. 3.1

ImageJ plugin for computing the histogram of an 8-bit grayscale image. The `setup()` method returns `DOES_8G + NO_CHANGES`, which indicates that this plugin requires an 8-bit grayscale image and will not alter it (line 4).

In Java, all elements of a newly instantiated numerical array are automatically initialized to zero (line 8).

```
1 public class Compute_Histogram implements PlugInFilter {  
2  
3     public int setup(String arg, ImagePlus img) {  
4         return DOES_8G + NO_CHANGES;  
5     }  
6  
7     public void run(ImageProcessor ip) {  
8         int[] h = new int[256]; // histogram array  
9         int w = ip.getWidth();  
10        int h = ip.getHeight();  
11  
12        for (int v = 0; v < h; v++) {  
13            for (int u = 0; u < w; u++) {  
14                int i = ip.getPixel(u, v);  
15                h[i] = h[i] + 1;  
16            }  
17        }  
18        // ... histogram h can now be used  
19    }  
20}
```

implementation would not use K individual *variables* to represent the counters but instead would use an *array* with K entries (`int[256]` in Java). In this example, the actual implementation as an array is straightforward. Since the intensity values begin at zero (like arrays in Java) and are all positive, they can be used directly as the indices $i \in [0, N-1]$ of the histogram array. Program 3.1 contains the complete Java source code for computing a histogram within the `run()` method of an ImageJ plugin.

At the start of Prog. 3.1, the array `h` of type `int[]` is created (line 8) and its elements are automatically initialized⁴ to 0. It makes no difference, at least in terms of the final result, whether the array is

⁴ In Java, arrays of primitives such as `int`, `double` are initialized at creation to 0 in the case of integer types or 0.0 for floating-point types, while arrays of objects are initialized to `null`.

traversed in row or column order, as long as all pixels in the image are visited exactly once. In contrast to Prog. 2.1, in this example we traverse the array in the standard row-first order such that the outer `for` loop iterates over the *vertical* coordinates v and the inner loop over the *horizontal* coordinates u .⁵ Once the histogram has been calculated, it is available for further processing steps or for being displayed.

Of course, histogram computation is already implemented in ImageJ and is available via the method `getHistogram()` for objects of the class `ImageProcessor`. If we use this built-in method, the `run()` method of Prog. 3.1 can be simplified to

```
public void run(ImageProcessor ip) {
    int[] h = ip.getHistogram(); // built-in ImageJ method
    ... // histogram h can now be used
}
```

3.4 HISTOGRAMS OF IMAGES WITH MORE THAN 8 BITS

3.4 Histograms of Images with More than 8 Bits

Normally histograms are computed in order to visualize the image's distribution on the screen. This presents no problem when dealing with images having $2^8 = 256$ entries, but when an image uses a larger range of values, for instance 16- and 32-bit or floating-point images (see Table 1.1), then the growing number of necessary histogram entries makes this no longer practical.

3.4.1 Binning

Since it is not possible to represent each intensity value with its own entry in the histogram, we will instead let a given entry in the histogram represent a *range* of intensity values. This technique is often referred to as “binning” since you can visualize it as collecting a range of pixel values in a container such as a bin or bucket. In a binned histogram of size B , each bin $h(j)$ contains the number of image elements having values within the interval $[a_j, a_{j+1})$, and therefore (analogous to Eqn. (3.1))

$$h(j) = \text{card} \{ (u, v) \mid a_j \leq I(u, v) < a_{j+1} \}, \quad (3.2)$$

for $0 \leq j < B$. Typically the range of possible values in B is divided into bins of equal size $k_B = K/B$ such that the starting value of the interval j is

$$a_j = j \cdot \frac{K}{B} = j \cdot k_B .$$

3.4.2 Example

In order to create a typical histogram containing $B = 256$ entries from a 14-bit image, one would divide the original value range $j =$

⁵ In this way, image elements are traversed in exactly the same way that they are laid out in computer memory, resulting in more efficient memory access and with it the possibility of increased performance, especially when dealing with larger images (see also Appendix F).

$0, \dots, 2^{14}-1$ into 256 equal intervals, each of length $k_B = 2^{14}/256 = 64$, such that $a_0 = 0$, $a_1 = 64$, $a_2 = 128$, \dots , $a_{255} = 16320$ and $a_{256} = a_B = 2^{14} = 16320 = K$. This gives the following association between pixel values and histogram bins $h(0), \dots, h(255)$:

$$\begin{aligned} 0, \dots, 63 &\rightarrow h(0), \\ 64, \dots, 127 &\rightarrow h(1), \\ 128, \dots, 191 &\rightarrow h(2), \\ \vdots &\quad \vdots \quad \vdots \\ 16320, \dots, 16383 &\rightarrow h(255). \end{aligned}$$

3.4.3 Implementation

If, as in the previous example, the value range $0, \dots, K-1$ is divided into equal length intervals $k_B = K/B$, there is naturally no need to use a mapping table to find a_j since for a given pixel value $a = I(u, v)$ the correct histogram element j is easily computed. In this case, it is enough to simply divide the pixel value $I(u, v)$ by the interval length k_B ; that is,

$$\frac{I(u, v)}{k_B} = \frac{I(u, v)}{K/B} = \frac{I(u, v) \cdot B}{K}. \quad (3.3)$$

As an index to the appropriate histogram bin $h(j)$, we require an integer value

$$j = \left\lfloor \frac{I(u, v) \cdot B}{K} \right\rfloor, \quad (3.4)$$

where $\lfloor \cdot \rfloor$ denotes the *floor* operator.⁶ A Java method for computing histograms by “linear binning” is given in Prog. 3.2. Note that all the computations from Eqn. (3.4) are done with integer numbers without using any floating-point operations. Also there is no need to explicitly call the *floor* function because the expression

$$a * B / K$$

in line 11 uses integer division and in Java the fractional result of such an operation is truncated, which is equivalent to applying the floor function (assuming positive arguments).⁷ The binning method can also be applied, in a similar way, to floating-point images.

3.5 Histograms of Color Images

When referring to histograms of color images, typically what is meant is a histogram of the image intensity (luminance) or of the individual color channels. Both of these variants are supported by practically every image-processing application and are used to objectively appraise the image quality, especially directly after image acquisition.

⁶ $\lfloor x \rfloor$ rounds x down to the next whole number (see Appendix A).

⁷ For a more detailed discussion, see the section on integer division in Java in Appendix F (p. 765).

```

1 int[] binnedHistogram(ImageProcessor ip) {
2     int K = 256; // number of intensity values
3     int B = 32; // size of histogram, must be defined
4     int[] H = new int[B]; // histogram array
5     int w = ip.getWidth();
6     int h = ip.getHeight();
7
8     for (int v = 0; v < h; v++) {
9         for (int u = 0; u < w; u++) {
10            int a = ip.getPixel(u, v);
11            int i = a * B / K; // integer operations only!
12            H[i] = H[i] + 1;
13        }
14    }
15    // return binned histogram
16    return H;
17 }
```

3.5 HISTOGRAMS OF COLOR IMAGES

Prog. 3.2

Histogram computation using “binning” (Java method). Example of computing a histogram with $B = 32$ bins for an 8-bit grayscale image with $K = 256$ intensity levels. The method `binnedHistogram()` returns the histogram of the image object `ip` passed to it as an `int` array of size B .

3.5.1 Intensity Histograms

The intensity or *luminance* histogram h_{Lum} of a color image is nothing more than the histogram of the corresponding grayscale image, so naturally all aspects of the preceding discussion also apply to this type of histogram. The grayscale image is obtained by computing the luminance of the individual channels of the color image. When computing the luminance, it is not sufficient to simply average the values of each color channel; instead, a weighted sum that takes into account color perception theory should be computed. This process is explained in detail in Chapter 12 (p. 304).

3.5.2 Individual Color Channel Histograms

Even though the luminance histogram takes into account all color channels, image errors appearing in single channels can remain undiscovered. For example, the luminance histogram may appear clean even when one of the color channels is oversaturated. In RGB images, the blue channel contributes only a small amount to the total brightness and so is especially sensitive to this problem.

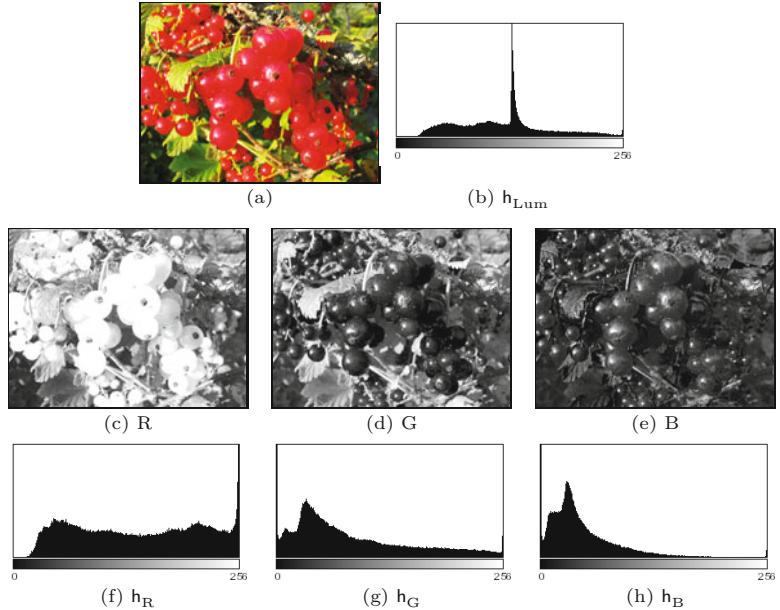
Component histograms supply additional information about the intensity distribution within the individual color channels. When computing component histograms, each color channel is considered a separate intensity image and each histogram is computed independently of the other channels. Figure 3.12 shows the luminance histogram h_{Lum} and the three component histograms h_R , h_G , and h_B of a typical RGB color image. Notice that saturation problems in all three channels (red in the upper intensity region, green and blue in the lower regions) are obvious in the component histograms but not in the luminance histogram. In this case it is striking, and not at all atypical, that the three component histograms appear completely different from the corresponding luminance histogram h_{Lum} (Fig. 3.12(b)).

3 HISTOGRAMS AND IMAGE STATISTICS

Fig. 3.12

Histograms of an RGB color image: original image (a), luminance histogram h_{Lum} (b), RGB color components as intensity images (c–e), and the associated component histograms h_R , h_G , h_B (f–h).

The fact that all three color channels have saturation problems is only apparent in the individual component histograms. The spike in the distribution resulting from this is found in the middle of the luminance histogram (b).



3.5.3 Combined Color Histograms

Luminance histograms and component histograms both provide useful information about the lighting, contrast, dynamic range, and saturation effects relative to the individual color components. It is important to remember that they provide no information about the distribution of the actual *colors* in the image because they are based on the individual color channels and not the combination of the individual channels that forms the color of an individual pixel. Consider, for example, when h_R , the component histogram for the red channel, contains the entry

$$h_R(200) = 24.$$

Then it is only known that the image has 24 pixels that have a red intensity value of 200. The entry does not tell us anything about the green and blue values of those pixels, which could be any valid value (*), that is,

$$(r, g, b) = (200, *, *).$$

Suppose further that the three component histograms included the following entries:

$$h_R(50) = 100, \quad h_G(50) = 100, \quad h_B(50) = 100.$$

Could we conclude from this that the image contains 100 pixels with the color combination

$$(r, g, b) = (50, 50, 50)$$

or that this color occurs at all? In general, no, because there is no way of ascertaining from these data if there exists a pixel in the image in which all three components have the value 50. The only thing we could really say is that the color value (50, 50, 50) can occur at most 100 times in this image.

So, although conventional (intensity or component) histograms of color images depict important properties, they do not really provide any useful information about the composition of the actual colors in an image. In fact, a collection of color images can have very similar component histograms and still contain entirely different colors. This leads to the interesting topic of the *combined* histogram, which uses statistical information about the combined color components in an attempt to determine if two images are roughly similar in their color composition. Features computed from this type of histogram often form the foundation of color-based image retrieval methods. We will return to this topic in Chapter 12, where we will explore color images in greater detail.

3.7 STATISTICAL INFORMATION FROM THE HISTOGRAM

3.6 The Cumulative Histogram

The cumulative histogram, which is derived from the ordinary histogram, is useful when performing certain image operations involving histograms; for instance, histogram equalization (see Sec. 4.5). The cumulative histogram H is defined as

$$H(i) = \sum_{j=0}^i h(j) \quad \text{for } 0 \leq i < K. \quad (3.5)$$

A particular value $H(i)$ is thus the sum of all histogram values $h(j)$, with $j \leq i$. Alternatively, we can define H recursively (as implemented in Prog. 4.2 on p. 66):

$$H(i) = \begin{cases} h(0) & \text{for } i = 0, \\ H(i-1) + h(i) & \text{for } 0 < i < K. \end{cases} \quad (3.6)$$

The cumulative histogram $H(i)$ is a monotonically increasing function with the maximum value

$$H(K-1) = \sum_{j=0}^{K-1} h(j) = M \cdot N, \quad (3.7)$$

that is, the total number of pixels in an image of width M and height N . [Figure 3.13](#) shows a concrete example of a cumulative histogram.

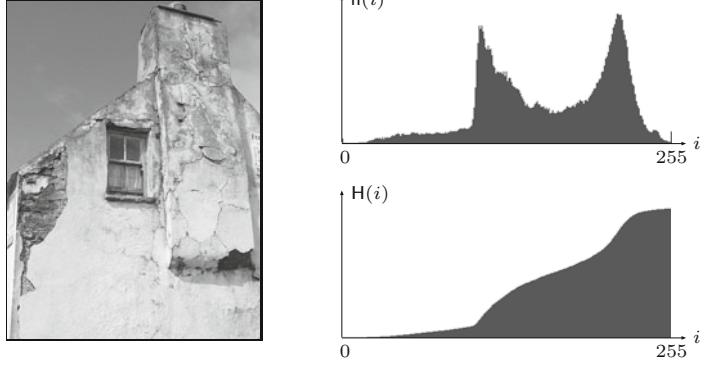
The cumulative histogram is useful not primarily for viewing but as a simple and powerful tool for capturing statistical information from an image. In particular, we will use it in the next chapter to compute the parameters for several common point operations (see Sec. 4.4–4.6).

3.7 Statistical Information from the Histogram

Some common statistical parameters of the image can be conveniently calculated directly from its histogram. For example, the minimum and maximum pixel value of an image I can be obtained by simply

Fig. 3.13

The ordinary histogram $\mathbf{h}(i)$ and its associated cumulative histogram $\mathbf{H}(i)$.



finding the smallest and largest histogram index with nonzero value, i.e.,

$$\begin{aligned}\min(I) &= \min \{ i \mid \mathbf{h}(i) > 0 \}, \\ \max(I) &= \max \{ i \mid \mathbf{h}(i) > 0 \}.\end{aligned}\quad (3.8)$$

If we assume that the histogram is already available, the advantage is that the calculation does not include the entire image but only the relatively small set of histogram elements (typ. 256).

3.7.1 Mean and Variance

The *mean* value μ of an image I (of size $M \times N$) can be calculated as

$$\mu = \frac{1}{MN} \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} I(u, v) = \frac{1}{MN} \cdot \sum_{i=0}^{K-1} \mathbf{h}(i) \cdot i, \quad (3.9)$$

i.e., either directly from the pixel values $I(u, v)$ or indirectly from the histogram \mathbf{h} (of size K), where $MN = \sum_i \mathbf{h}(i)$ is the total number of pixels.

Analogously we can also calculate the *variance* of the pixel values straight from the histogram as

$$\sigma^2 = \frac{1}{MN} \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} [I(u, v) - \mu]^2 = \frac{1}{MN} \cdot \sum_{i=0}^{K-1} (i - \mu)^2 \cdot \mathbf{h}(i). \quad (3.10)$$

As we see in the right parts of Eqns. (3.9) and (3.10), there is no need to access the original pixel values.

The formulation of the variance in Eqn. (3.10) assumes that the arithmetic mean μ has already been determined. This is not necessary though, since the mean and the variance can be calculated together in a single iteration over the image pixels or the associated histogram in the form

$$\mu = \frac{1}{MN} \cdot A \quad \text{and} \quad (3.11)$$

$$\sigma^2 = \frac{1}{MN} \cdot \left(B - \frac{1}{MN} \cdot A^2 \right), \quad (3.12)$$

with the quantities

$$A = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} I(u, v) = \sum_{i=0}^{K-1} i \cdot h(i), \quad (3.13)$$

$$B = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} I^2(u, v) = \sum_{i=0}^{K-1} i^2 \cdot h(i). \quad (3.14)$$

The above formulation has the additional numerical advantage that all summations can be performed with integer values, in contrast to Eqn. (3.10) which requires the summation of floating-point values.

3.7.2 Median

The median m of an image is defined as the smallest pixel value that is greater or equal to one half of all pixel values, i.e., lies “in the middle” of the pixel values.⁸ The median can also be easily calculated from the image’s histogram.

To determine the median of an image I from the associated histogram it is sufficient to find the index i that separates the histogram into two halves, such that the sum of the histogram entries to the left and the right of i are approximately equal. In other words, i is the smallest index where the sum of the histogram entries below (and including) i corresponds to at least half of the image size, that is,

$$m = \min \left\{ i \mid \sum_{j=0}^i h(j) \geq \frac{MN}{2} \right\}. \quad (3.15)$$

Since $\sum_{j=0}^i h(j) = H(i)$ (see Eqn. (3.5)), the median calculation can be formulated even simpler as

$$m = \min \left\{ i \mid H(i) \geq \frac{MN}{2} \right\}, \quad (3.16)$$

given the cumulative histogram H .

3.8 Block Statistics

3.8.1 Integral Images

Integral images (also known as *summed area tables* [58]) provide a simple way for quickly calculating elementary statistics of arbitrary rectangular sub-images. They have found use in several interesting applications, such as fast filtering, adaptive thresholding, image matching, local feature extraction, face detection, and stereo reconstruction [20, 142, 244].

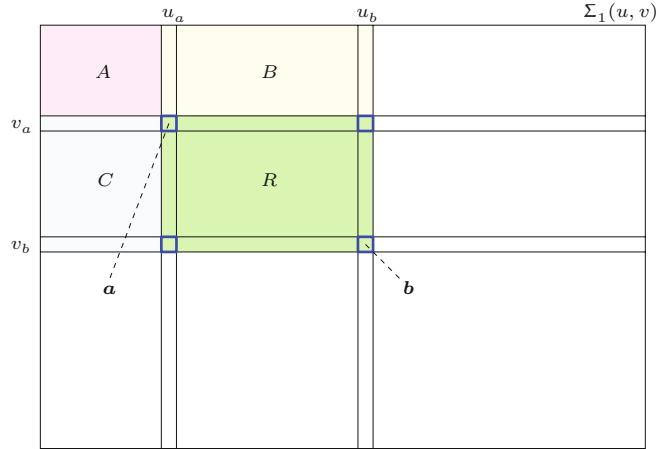
Given a scalar-valued (grayscale) image $I: M \times N \mapsto \mathbb{R}$ the associated *first-order* integral image is defined as

$$\Sigma_1(u, v) = \sum_{i=0}^u \sum_{j=0}^v I(i, j). \quad (3.17)$$

⁸ See Sec. 5.4.2 for an alternative definition of the median.

Fig. 3.14

Block-based calculations with integral images. Only four samples from the integral image Σ_1 are required to calculate the sum of the pixels inside the (green) rectangle $R = \langle \mathbf{a}, \mathbf{b} \rangle$, defined by the corner coordinates $\mathbf{a} = (u_a, v_a)$ and $\mathbf{b} = (u_b, v_b)$.



Thus a value in Σ_1 is the sum of all pixel values in the original image I located to the left and above the given position (u, v) , inclusively. The integral image can be calculated efficiently with a single pass over the image I by using the recurrence relation

$$\Sigma_1(u, v) = \begin{cases} 0 & \text{for } u < 0 \text{ or } v < 0, \\ \Sigma_1(u-1, v) + \Sigma_1(u, v-1) - \Sigma_1(u-1, v-1) + I(u, v) & \text{for } u, v \geq 0, \end{cases} \quad (3.18)$$

for positions $u = 0, \dots, M-1$ and $v = 0, \dots, N-1$ (see Alg. 3.1).

Suppose now that we wanted to calculate the sum of the pixel values in a given rectangular region R , defined by the corner positions $\mathbf{a} = (u_a, v_a)$, $\mathbf{b} = (u_b, v_b)$, that is, the *first-order block sum*

$$S_1(R) = \sum_{i=u_a}^{u_b} \sum_{j=v_a}^{v_b} I(i, j), \quad (3.19)$$

from the integral image Σ_1 . As shown in Fig. 3.14, the quantity $\Sigma_1(u_a-1, v_a-1)$ corresponds to the pixel sum within rectangle A , and $\Sigma_1(u_b, v_b)$ is the pixel sum over all four rectangles A , B , C and R , that is,

$$\begin{aligned} \Sigma_1(u_a-1, v_a-1) &= S_1(A), \\ \Sigma_1(u_b, v_a-1) &= S_1(A) + S_1(B), \\ \Sigma_1(u_a-1, v_b) &= S_1(A) + S_1(C), \\ \Sigma_1(u_b, v_b) &= S_1(A) + S_1(B) + S_1(C) + S_1(R). \end{aligned} \quad (3.20)$$

Thus $S_1(R)$ can be calculated as

$$\begin{aligned} S_1(R) &= \underbrace{S_1(A) + S_1(B) + S_1(C) + S_1(R)}_{\Sigma_1(u_b, v_b)} + \underbrace{S_1(A)}_{\Sigma_1(u_a-1, v_a-1)} \\ &\quad - \underbrace{[S_1(A) + S_1(B)]}_{\Sigma_1(u_b, v_a-1)} - \underbrace{[S_1(A) + S_1(C)]}_{\Sigma_1(u_a-1, v_b)} \\ &= \Sigma_1(u_b, v_b) + \Sigma_1(u_a-1, v_a-1) - \Sigma_1(u_b, v_a-1) - \Sigma_1(u_a-1, v_b), \end{aligned} \quad (3.21)$$

that is, by taking only *four* samples from the integral image Σ_1 .

Given the region size N_R and the sum of the pixel values $S_1(R)$, the average intensity value (*mean*) inside the rectangle R can now easily be found as

$$\mu_R = \frac{1}{N_R} \cdot S_1(R), \quad (3.22)$$

with $S_1(R)$ as defined in Eqn. (3.21) and the region size

$$N_R = |R| = (u_b - u_a + 1) \cdot (v_b - v_a + 1). \quad (3.23)$$

3.8.3 Variance

Calculating the *variance* inside a rectangular region R requires the summation of squared intensity values, that is, tabulating

$$\Sigma_2(u, v) = \sum_{i=0}^u \sum_{j=0}^v I^2(i, j), \quad (3.24)$$

which can be performed analogously to Eqn. (3.18) in the form

$$\Sigma_2(u, v) = \begin{cases} 0 & \text{for } u < 0 \text{ or } v < 0, \\ \Sigma_2(u-1, v) + \Sigma_2(u, v-1) - \\ \Sigma_2(u-1, v-1) + I^2(u, v) & \text{for } u, v \geq 0. \end{cases} \quad (3.25)$$

As in Eqns. (3.19)–(3.21), the sum of the *squared* values inside a given rectangle R (i.e., the *second-order block sum*) can be obtained as

$$\begin{aligned} S_2(R) &= \sum_{i=u_0}^{u_1} \sum_{j=v_0}^{v_1} I^2(i, j) \\ &= \Sigma_2(u_b, v_b) + \Sigma_2(u_a-1, v_a-1) - \Sigma_2(u_b, v_a-1) - \Sigma_2(u_a-1, v_b). \end{aligned} \quad (3.26)$$

From this, the variance inside the rectangular region R is finally calculated as

$$\sigma_R^2 = \frac{1}{N_R} [S_2(R) - \frac{1}{N_R} \cdot (S_1(R))^2], \quad (3.27)$$

with N_R as defined in Eqn. (3.23). In addition, certain higher-order statistics can be efficiently calculated with summation tables in a similar fashion.

3.8.4 Practical Calculation of Integral Images

Algorithm 3.1 shows how Σ_1 and Σ_2 can be calculated in a single iteration over the original image I . Note that the accumulated values in the integral images Σ_1 , Σ_2 tend to become quite large. Even with pictures of medium size and 8-bit intensity values, the range of 32-bit integers is quickly exhausted (particularly when calculating Σ_2). The use of 64-bit integers (type `long` in Java) or larger is recommended to avoid arithmetic overflow. A basic implementation of integral images is available as part of the `imagingbook` library.⁹

⁹ Class `imagingbook.lib.image.IntegralImage`.

Alg. 3.1
Joint calculation of the integral images Σ_1 and Σ_2 for a scalar-valued image I .

```

1: IntegralImage( $I$ )
   Input:  $I$ , a scalar-valued input image with  $I(u, v) \in \mathbb{R}$ .
   Returns the first and second order integral images of  $I$ .
2:  $(M, N) \leftarrow \text{Size}(I)$ 
3: Create maps  $\Sigma_1, \Sigma_2: M \times N \mapsto \mathbb{R}$ 
   Process the first image line ( $v = 0$ ):
4:  $\Sigma_1(0, 0) \leftarrow I(0, 0)$ 
5:  $\Sigma_2(0, 0) \leftarrow I^2(0, 0)$ 
6: for  $u \leftarrow 1, \dots, M-1$  do
7:    $\Sigma_1(u, 0) \leftarrow \Sigma_1(u-1, 0) + I(u, 0)$ 
8:    $\Sigma_2(u, 0) \leftarrow \Sigma_2(u-1, 0) + I^2(u, 0)$ 
   Process the remaining image lines ( $v > 0$ ):
9: for  $v \leftarrow 1, \dots, N-1$  do
10:    $\Sigma_1(0, v) \leftarrow \Sigma_1(0, v-1) + I(0, v)$ 
11:    $\Sigma_2(0, v) \leftarrow \Sigma_2(0, v-1) + I^2(0, v)$ 
12:   for  $u \leftarrow 1, \dots, M-1$  do
13:      $\Sigma_1(u, v) \leftarrow \Sigma_1(u-1, v) + \Sigma_1(u, v-1) -$ 
         $\Sigma_1(u-1, v-1) + I(u, v)$ 
14:      $\Sigma_2(u, v) \leftarrow \Sigma_2(u-1, v) + \Sigma_2(u, v-1) -$ 
         $\Sigma_2(u-1, v-1) + I^2(u, v)$ 
15: return  $(\Sigma_1, \Sigma_2)$ 
```

3.9 Exercises

Exercise 3.1. In Prog. 3.2, B and K are constants. Consider if there would be an advantage to computing the value of B/K outside of the loop, and explain your reasoning.

Exercise 3.2. Develop an ImageJ plugin that computes the cumulative histogram of an 8-bit grayscale image and displays it as a new image, similar to $H(i)$ in Fig. 3.13. *Hint:* Use the `ImageProcessor` method `int[] getHistogram()` to retrieve the original image's histogram values and then compute the cumulative histogram "in place" according to Eqn. (3.6). Create a new (blank) image of appropriate size (e.g., 256×150) and draw the scaled histogram data as black vertical bars such that the maximum entry spans the full height of the image. Program 3.3 shows how this plugin could be set up and how a new image is created and displayed.

Exercise 3.3. Develop a technique for nonlinear binning that uses a table of interval limits a_j (Eqn. (3.2)).

Exercise 3.4. Develop an ImageJ plugin that uses the Java methods `Math.random()` or `Random.nextInt(int n)` to create an image with random pixel values that are uniformly distributed in the range $[0, 255]$. Analyze the image's histogram to determine how equally distributed the pixel values truly are.

Exercise 3.5. Develop an ImageJ plugin that creates a random image with a Gaussian (normal) distribution with mean value $\mu = 128$ and standard deviation $\sigma = 50$. Use the standard Java method `double Random.nextGaussian()` to produce normally-distributed

random numbers (with $\mu = 0$ and $\sigma = 1$) and scale them appropriately to pixel values. Analyze the resulting image histogram to see if it shows a Gaussian distribution too.

3.9 EXERCISES

Exercise 3.6. Implement the calculation of the arithmetic *mean* μ and the *variance* σ^2 of a given grayscale image from its histogram h (see Sec. 3.7.1). Compare your results to those returned by ImageJ's *Analyze* \triangleright *Measure* tool (they should match *exactly*).

Exercise 3.7. Implement the first-order integral image (Σ_1) calculation described in Eqn. (3.18) and calculate the sum of pixel values $S_1(R)$ inside a given rectangle R using Eqn. (3.21). Verify numerically that the results are the same as with the naive formulation in Eqn. (3.19).

Exercise 3.8. Values of integral images tend to become quite large. Assume that 32-bit signed integers (`int`) are used to calculate the integral of the squared pixel values, that is, Σ_2 (see Eqn. (3.24)), for an 8-bit grayscale image. What is the maximum image size that is guaranteed not to cause an arithmetic overflow? Perform the same analysis for 64-bit signed integers (`long`).

Exercise 3.9. Calculate the integral image Σ_1 for a given image I , convert it to a floating-point iamge (`FloatProcessor`) and display the result. You will realize that integral images are without any apparent structure and they all look more or less the same. Come up with an efficient method for reconstructing the original image I from Σ_1 .

3 HISTOGRAMS AND IMAGE STATISTICS

Prog. 3.3

Creating and displaying a new image (ImageJ plugin). First, we create a `ByteProcessor` object (`histIp`, line 20) that is subsequently filled. At this point, `histIp` has no screen representation and is thus not visible. Then, an associated `ImagePlus` object is created (line 33) and displayed by applying the `show()` method (line 34). Notice how the title (`String`) is retrieved from the original image inside the `setup()` method (line 10) and used to compose the new image's title (lines 30 and 33). If `histIp` is changed *after* calling `show()`, then the method `updateAndDraw()` could be used to redisplay the associated image again (line 34).

```
1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ByteProcessor;
4 import ij.process.ImageProcessor;
5
6 public class Create_New_Image implements PlugInFilter {
7     ImagePlus im;
8
9     public int setup(String arg, ImagePlus im) {
10        this.im = im;
11        return DOES_8G + NO_CHANGES;
12    }
13
14    public void run(ImageProcessor ip) {
15        // obtain the histogram of ip:
16        int[] hist = ip.getHistogram();
17        int K = hist.length;
18
19        // create the histogram image:
20        ImageProcessor hip = new ByteProcessor(K, 100);
21        hip.setValue(255); // white = 255
22        hip.fill();
23
24        // draw the histogram values as black bars in hip here,
25        // for example, using hip.putpixel(u, v, 0)
26        // ...
27
28        // compose a nice title:
29        String imTitle = im.getShortTitle();
30        String histTitle = "Histogram of " + imTitle;
31
32        // display the histogram image:
33        ImagePlus him = new ImagePlus(title, hip);
34        him.show();
35    }
36 }
```

Point Operations

Point operations perform a modification of the pixel values without changing the size, geometry, or local structure of the image. Each new pixel value $b = I'(u, v)$ depends exclusively on the previous value $a = I(u, v)$ at the *same* position and is thus independent from any other pixel value, in particular from any of its neighboring pixels.¹ The original pixel values a are mapped to the new values b by some given function f , i.e.,

$$b = f(I(u, v)) \quad \text{or} \quad b = f(a). \quad (4.1)$$

If, as in this case, the function $f()$ is independent of the image coordinates (i.e., the same throughout the image), the operation is called “global” or “homogeneous”. Typical examples of homogeneous point operations include, among others:

- modifying image brightness or contrast,
- applying arbitrary intensity transformations (“curves”),
- inverting images,
- quantizing (or “posterizing”) images,
- global thresholding,
- gamma correction,
- color transformations
- etc.

We will look at some of these techniques in more detail in the following.

In contrast to Eqn. (4.1), the mapping function $g()$ for a *nonhomogeneous* point operation would also take into account the current image coordinate (u, v) , that is,

$$b = g(I(u, v), u, v) \quad \text{or} \quad b = f(a, u, v). \quad (4.2)$$

A typical nonhomogeneous operation is the local adjustment of contrast or brightness used, for example, to compensate for uneven lighting during image acquisition.

¹ If the result depends on more than one pixel value, the operation is called a “filter”, as described in Chapter 5.

4.1 Modifying Image Intensity

4.1.1 Contrast and Brightness

Let us start with a simple example. Increasing the image's contrast by 50% (i.e., by the factor 1.5) or raising the brightness by 10 units can be expressed by the mapping functions

$$f_{\text{contr}}(a) = a \cdot 1.5 \quad \text{or} \quad f_{\text{bright}}(a) = a + 10, \quad (4.3)$$

respectively. The first operation is implemented as an ImageJ plugin by the code shown in Prog. 4.1, which can easily be adapted to perform any other type of point operation. Rounding to the nearest integer values is accomplished by simply adding 0.5 before the truncation effected by the `(int)` typecast in line 8 (this only works for positive values). Also note the use of the more efficient image processor methods `get()` and `set()` (instead of `getPixel()` and `putPixel()`) in this example.

Prog. 4.1

Point operation to increase the contrast by 50% (ImageJ plugin). Note that in line 8 the result of the multiplication of the integer pixel value by the constant 1.5 (implicitly of type `double`) is of type `double`.

Thus an explicit type cast (`int`) is required to assign the value to the int variable `a`. 0.5 is added in line 8 to round to the nearest integer values.

```

1  public void run(ImageProcessor ip) {
2      int w = ip.getWidth();
3      int h = ip.getHeight();
4
5      for (int v = 0; v < h; v++) {
6          for (int u = 0; u < w; u++) {
7              int a = ip.get(u, v);
8              int b = (int) (a * 1.5 + 0.5);
9              if (b > 255)
10                  b = 255; // clamp to the maximum value (amax)
11              ip.set(u, v, b);
12          }
13      }
14  }
```

4.1.2 Limiting Values by Clamping

When implementing arithmetic operations on pixel values, we must keep in mind that the calculated results must not exceed the admissible range of pixel values for the given image type (e.g., [0, 255] in the case of 8-bit grayscale images). This is commonly called “clamping” and can be expressed in the form

$$b = \min(\max(f(a), a_{\min}), a_{\max}) = \begin{cases} a_{\min} & \text{for } f(a) < a_{\min}, \\ a_{\max} & \text{for } f(a) > a_{\max}, \\ f(a) & \text{otherwise.} \end{cases} \quad (4.4)$$

For this purpose, line 10 of Prog. 4.1 contains the statement

```
if (b > 255) b = 255;
```

which limits the result to the maximum value 255. Similarly, one may also want to limit the results to the minimum value (0) to avoid negative pixel values (which cannot be represented by this type of 8-bit image), for example, by the statement

```
if (b < 0) b = 0;
```

The above statement is not needed in Prog. 4.1 because the intermediate results can never be negative in this particular operation.

4.1.3 Inverting Images

Inverting an intensity image is a simple point operation that reverses the ordering of pixel values (by multiplying by -1) and adds a constant value to map the result to the admissible range again. Thus for a pixel value $a = I(u, v)$ in the range $[0, a_{\max}]$, the corresponding point operation is

$$f_{\text{inv}}(a) = -a + a_{\max} = a_{\max} - a. \quad (4.5)$$

The inversion of an 8-bit grayscale image with $a_{\max} = 255$ was the task of our first plugin example in Sec. 2.2.4 (Prog. 2.1). Note that in this case no clamping is required at all because the function always maps to the original range of values. In ImageJ, this operation is performed by the method `invert()` (for objects of type `ImageProcessor`) and is also available through the `Edit > Invert` menu. Obviously, inverting an image mirrors its histogram, as shown in Fig. 4.5(c).

4.1.4 Threshold Operation

Thresholding an image is a special type of quantization that separates the pixel values in two classes, depending upon a given threshold value q that is usually constant. The threshold operation maps all pixels to one of two fixed intensity values a_0 or a_1 , that is,

$$f_{\text{threshold}}(a) = \begin{cases} a_0 & \text{for } a < q, \\ a_1 & \text{for } a \geq q, \end{cases} \quad (4.6)$$

with $0 < q \leq a_{\max}$. A common application is *binarizing* an intensity image with the values $a_0 = 0$ and $a_1 = 1$.

ImageJ does provide a special image type (`BinaryProcessor`) for binary images, but these are actually implemented as 8-bit intensity images (just like ordinary intensity images) using the values 0 and 255. ImageJ also provides the `ImageProcessor` method `threshold(int level)`, with $level \equiv q$, to perform this operation, which can also be invoked through the `Image > Adjust > Threshold` menu (see Fig. 4.1 for an example). Thresholding affects the histogram by separating the distribution into two entries at positions a_0 and a_1 , as illustrated in Fig. 4.2.

4.2 Point Operations and Histograms

We have already seen that the effects of a point operation on the image's histogram are quite easy to predict in some cases. For example, increasing the brightness of an image by a constant value shifts the entire histogram to the right, raising the contrast widens

4 POINT OPERATIONS

Fig. 4.1

Threshold operation: original image (a) and corresponding histogram (c); result after thresholding with $a_{\text{th}} = 128$, $a_0 = 0$, $a_1 = 255$ (b) and corresponding histogram (d); ImageJ's interactive Threshold menu (e).

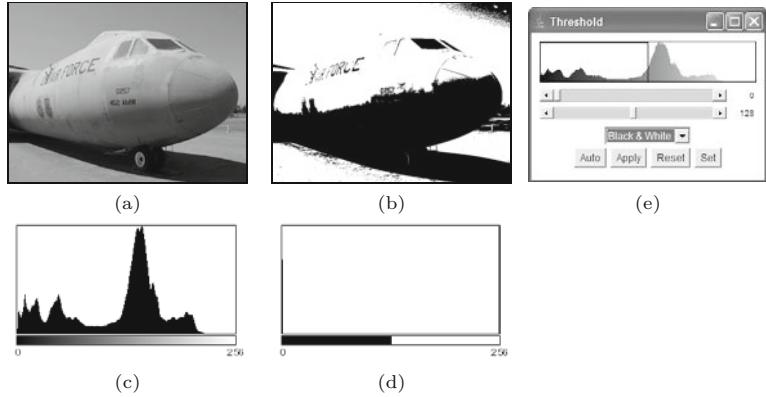
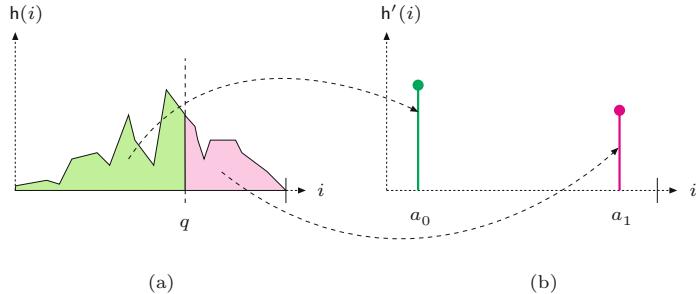


Fig. 4.2

Effects of thresholding upon the histogram. The threshold value is a_{th} . The original distribution (a) is split and merged into two isolated entries at a_0 and a_1 in the resulting histogram (b).

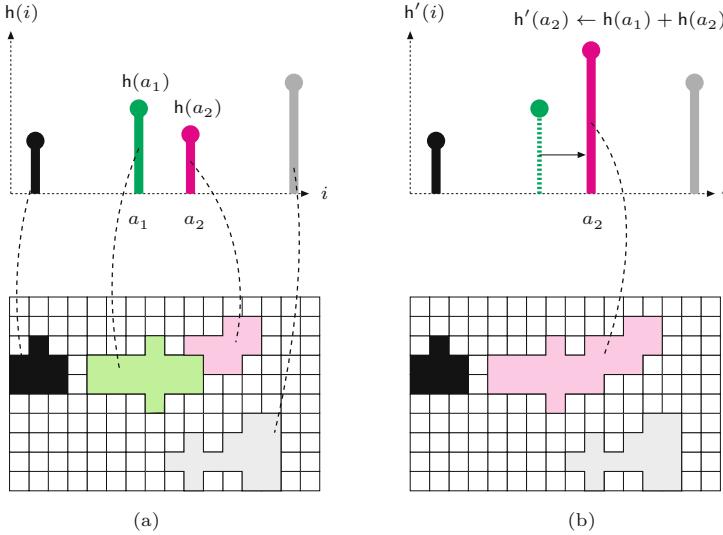


it, and inverting the image flips the histogram. Although this appears rather simple, it may be useful to look a bit more closely at the relationship between point operations and the resulting changes in the histogram.

As the illustration in Fig. 4.3 shows, every entry (bar) at some position i in the histogram maps to a *set* (of size $h(i)$) containing all image pixels whose values are exactly i .²

If a particular histogram line is *shifted* as a result of some point operation, then of course all pixels in the corresponding set are equally modified and vice versa. So what happens when a point operation (e.g., reducing image contrast) causes two previously separated histogram lines to fall together at the same position i ? The answer is that the corresponding pixel sets are *merged* and the new common histogram entry is the sum of the two (or more) contributing entries (i.e., the size of the combined set). At this point, the elements in the merged set are no longer distinguishable (or separable), so this operation may have (perhaps unintentionally) caused an irreversible reduction of dynamic range and thus a permanent loss of information in that image.

² Of course this is only true for ordinary histograms with an entry for every single intensity value. If *binning* is used (see Sec. 3.4.1), each histogram entry maps to pixels within a certain *range* of values.



4.3 AUTOMATIC CONTRAST ADJUSTMENT

Fig. 4.3

Histogram entries represent sets of pixels of the same value. If a histogram line is moved as a result of some point operation, then all pixels in the corresponding set are equally modified (a). If, due to this operation, two histogram lines $h(a_1)$, $h(a_2)$ coincide on the same index, the two corresponding pixel sets merge and the contained pixels become undiscernable (b).

4.3 Automatic Contrast Adjustment

Automatic contrast adjustment (auto-contrast) is a point operation whose task is to modify the pixels such that the available range of values is fully covered. This is done by mapping the current darkest and brightest pixels to the minimum and maximum intensity values, respectively, and linearly distributing the intermediate values.

Let us assume that a_{lo} and a_{hi} are the lowest and highest pixel values found in the current image, whose full intensity range is $[a_{min}, a_{max}]$. To stretch the image to the full intensity range (see Fig. 4.4), we first map the smallest pixel value a_{lo} to zero, subsequently increase the contrast by the factor $(a_{max} - a_{min}) / (a_{hi} - a_{lo})$, and finally shift to the target range by adding a_{min} . The mapping function for the auto-contrast operation is thus defined as

$$f_{ac}(a) = a_{min} + (a - a_{lo}) \cdot \frac{a_{max} - a_{min}}{a_{hi} - a_{lo}}, \quad (4.7)$$

provided that $a_{hi} \neq a_{lo}$; that is, the image contains at least *two* different pixel values. For an 8-bit image with $a_{min} = 0$ and $a_{max} = 255$, the function in Eqn. (4.7) simplifies to

$$f_{ac}(a) = (a - a_{lo}) \cdot \frac{255}{a_{hi} - a_{lo}}. \quad (4.8)$$

The target range $[a_{min}, a_{max}]$ need not be the maximum available range of values but can be any interval to which the image should be mapped. Of course the method can also be used to reduce the image contrast to a smaller range. Figure 4.5(b) shows the effects of an auto-contrast operation on the corresponding histogram, where the linear stretching of the intensity range results in regularly spaced gaps in the new distribution.

4 POINT OPERATIONS

Fig. 4.4

Auto-contrast operation according to Eqn. (4.7).

Original pixel values a in the range $[a_{lo}, a_{hi}]$ are mapped linearly to the target range $[a_{min}, a_{max}]$.

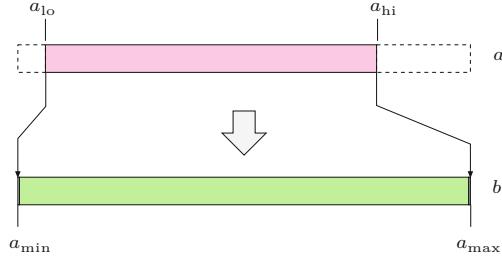
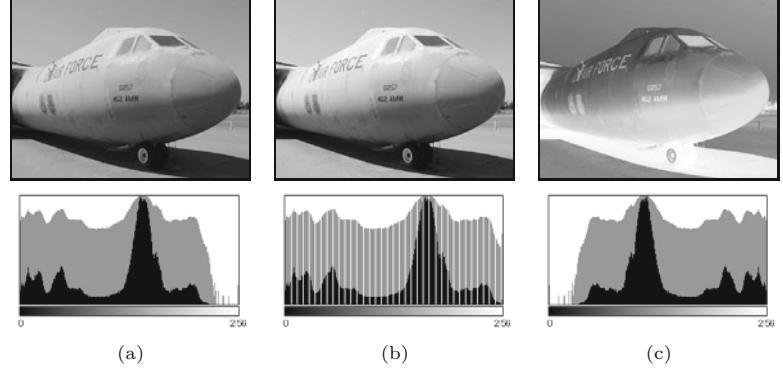


Fig. 4.5

Effects of auto-contrast and inversion operations on the resulting histograms. Original image (a), result of auto-contrast operation (b), and inversion (c). The histogram entries are shown both linearly (black bars) and logarithmically (gray bars).

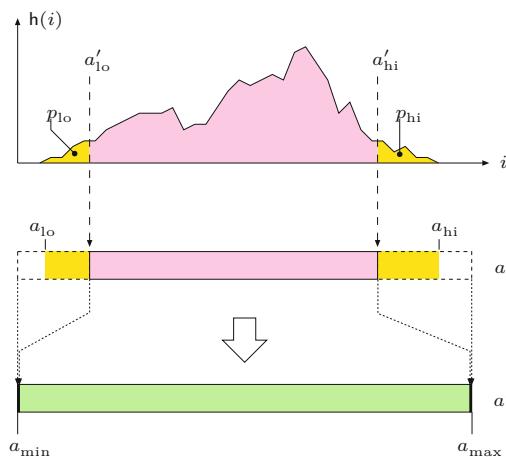


4.4 Modified Auto-Contrast Operation

In practice, the mapping function in Eqn. (4.7) could be strongly influenced by only a few extreme (low or high) pixel values, which may not be representative of the main image content. This can be avoided to a large extent by “saturating” a fixed percentage (p_{lo}, p_{hi}) of pixels at the upper and lower ends of the target intensity range. To accomplish this, we determine two limiting values a'_{lo}, a'_{hi} such that a predefined quantile q_{lo} of all pixel values in the image I are smaller than a'_{lo} and another quantile q_{hi} of the values are greater than a'_{hi} (Fig. 4.6).

Fig. 4.6

Modified auto-contrast operation (Eqn. (4.11)). Predefined quantiles (q_{lo}, q_{hi}) of image pixels—shown as dark areas at the left and right ends of the histogram $h(i)$ —are “saturated” (i.e., mapped to the extreme values of the target range). The intermediate values ($a = a'_{lo}, \dots, a'_{hi}$) are mapped linearly to the interval a_{min}, \dots, a_{max} .



The values a'_{lo} , a'_{hi} depend on the image content and can be easily obtained from the image's cumulative histogram³ H :

$$a'_{lo} = \min\{i \mid H(i) \geq M \cdot N \cdot p_{lo}\}, \quad (4.9)$$

$$a'_{hi} = \max\{i \mid H(i) \leq M \cdot N \cdot (1 - p_{hi})\}, \quad (4.10)$$

where $0 \leq p_{lo}, p_{hi} \leq 1$, $p_{lo} + p_{hi} \leq 1$, and $M \cdot N$ is the number of pixels in the image. All pixel values *outside* (and including) a'_{lo} and a'_{hi} are mapped to the extreme values a_{min} and a_{max} , respectively, and intermediate values are mapped linearly to the interval $[a_{min}, a_{max}]$. Using this formulation, the mapping to minimum and maximum intensities does not depend on singular extreme pixels only but can be based on a representative set of pixels. The mapping function for the modified auto-contrast operation can thus be defined as

$$f_{mac}(a) = \begin{cases} a_{min} & \text{for } a \leq a'_{lo}, \\ a_{min} + (a - a'_{lo}) \cdot \frac{a_{max} - a_{min}}{a'_{hi} - a'_{lo}} & \text{for } a'_{lo} < a < a'_{hi}, \\ a_{max} & \text{for } a \geq a'_{hi}. \end{cases} \quad (4.11)$$

Usually the same value is taken for both upper and lower quantiles (i.e., $p_{lo} = p_{hi} = p$), with $p = 0.005, \dots, 0.015$ (0.5, ..., 1.5 %) being common values. For example, the auto-contrast operation in Adobe Photoshop saturates 0.5 % ($p = 0.005$) of all pixels at both ends of the intensity range. Auto-contrast is a frequently used point operation and thus available in practically any image-processing software. ImageJ implements the modified auto-contrast operation as part of the Brightness/Contrast and Image ▷ Adjust menus (Auto button), shown in Fig. 4.7.

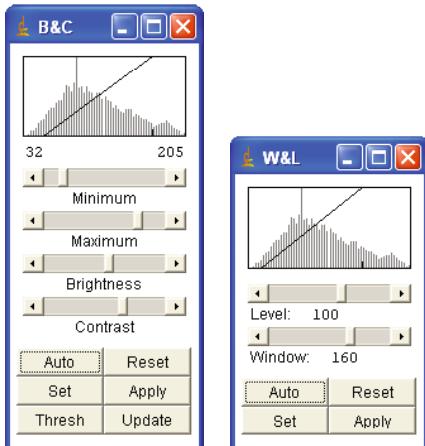


Fig. 4.7

ImageJ's Brightness/Contrast tool (left) and Window/Level tool (right) can be invoked through the Image ▷ Adjust menu. The Auto button displays the result of a modified auto-contrast operation. Apply must be hit to actually modify the image.

4.5 Histogram Equalization

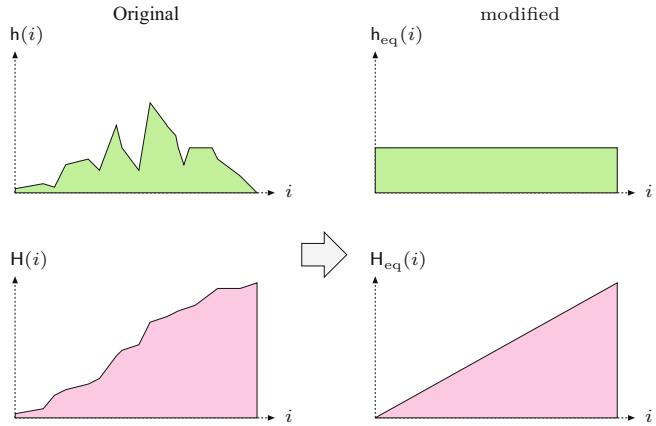
A frequent task is to adjust two different images in such a way that their resulting intensity distributions are similar, for example, to use

³ See Sec. 3.6.

4 POINT OPERATIONS

Fig. 4.8

Histogram equalization. The idea is to find and apply a point operation to the image (with original histogram h) such that the histogram h_{eq} of the modified image approximates a *uniform* distribution (top). The cumulative target histogram H_{eq} must thus be approximately wedge-shaped (bottom).

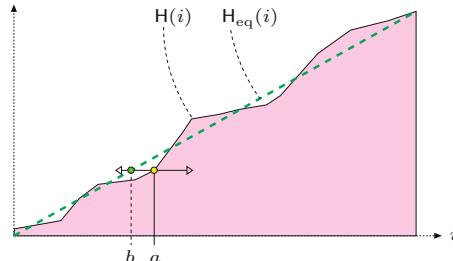


them in a print publication or to make them easier to compare. The goal of histogram equalization is to find and apply a point operation such that the histogram of the modified image approximates a *uniform* distribution (see Fig. 4.8). Since the histogram is a discrete distribution and homogeneous point operations can only shift and merge (but never split) histogram entries, we can only obtain an approximate solution in general. In particular, there is no way to eliminate or decrease individual peaks in a histogram, and a truly uniform distribution is thus impossible to reach. Based on point operations, we can thus modify the image only to the extent that the resulting histogram is *approximately* uniform. The question is how good this approximation can be and exactly which point operation (which clearly depends on the image content) we must apply to achieve this goal.

We may get a first idea by observing that the *cumulative histogram* (Sec. 3.6) of a uniformly distributed image is a linear ramp (wedge), as shown in Fig. 4.8. So we can reformulate the goal as finding a point operation that shifts the histogram lines such that the resulting cumulative histogram is approximately linear, as illustrated in Fig. 4.9.

Fig. 4.9

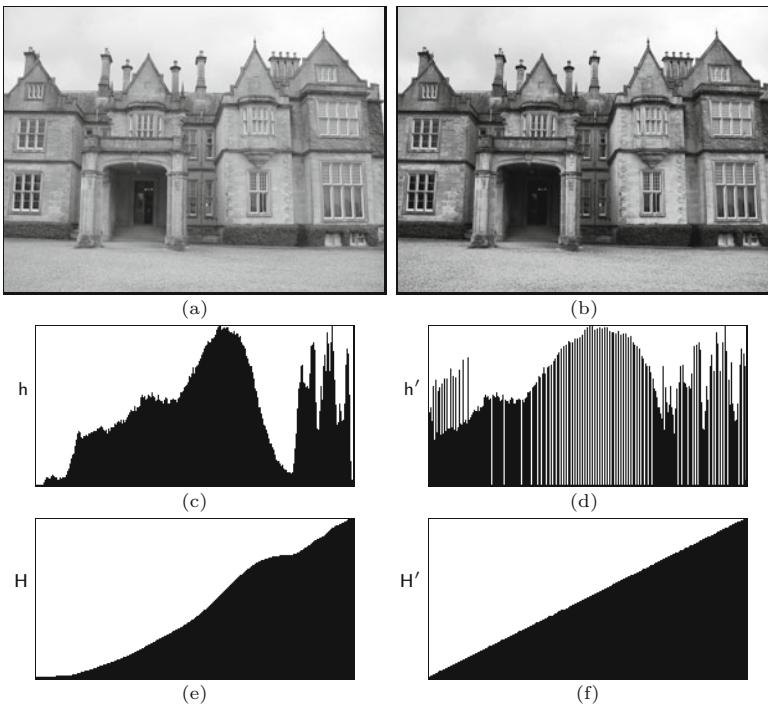
Histogram equalization on the cumulative histogram. A suitable point operation $b \leftarrow f_{eq}(a)$ shifts each histogram line from its original position a to b (left or right) such that the resulting cumulative histogram H_{eq} is approximately linear.



The desired point operation $f_{eq}()$ is simply obtained from the cumulative histogram H of the original image as⁴

$$f_{eq}(a) = \left\lfloor H(a) \cdot \frac{K-1}{M \cdot N} \right\rfloor, \quad (4.12)$$

⁴ For a derivation, see, for example, [88, p. 173].



4.5 HISTOGRAM EQUALIZATION

Fig. 4.10

Linear histogram equalization example. Original image I (a) and modified image I' (b), corresponding histograms h , h' (c, d), and cumulative histograms H , H' (e, f). The resulting cumulative histogram H' (f) approximates a uniformly distributed image. Notice that new peaks are created in the resulting histogram h' (d) by merging original histogram cells, particularly in the lower and upper intensity ranges.

for an image of size $M \times N$ with pixel values a in the range $[0, K-1]$. The resulting function $f_{\text{eq}}(a)$ in Eqn. (4.12) is monotonically increasing, because $H(a)$ is monotonic and K , M , N are all positive constants. In the (unusual) case where an image is already uniformly distributed, linear histogram equalization should not modify that image any further. Also, repeated applications of linear histogram equalization should not make any changes to the image after the first time. Both requirements are fulfilled by the formulation in Eqn. (4.12). Program 4.2 lists the Java code for a sample implementation of linear histogram equalization. An example demonstrating the effects on the image and the histograms is shown in Fig. 4.10.

Notice that for “inactive” pixel values i (i.e., pixel values that do not appear in the image, with $h(i) = 0$), the corresponding entries in the cumulative histogram $H(i)$ are either zero or identical to the neighboring entry $H(i-1)$. Consequently a contiguous range of zero values in the histogram $h(i)$ corresponds to a constant (i.e., flat) range in the cumulative histogram $H(i)$, and the function $f_{\text{eq}}(a)$ maps all “inactive” intensity values within such a range to the next lower “active” value. This effect is not relevant, however, since the image contains no such pixels anyway. Nevertheless, a linear histogram equalization may (and typically will) cause histogram lines to merge and consequently lead to a loss of dynamic range (see also Sec. 4.2).

This or a similar form of linear histogram equalization is implemented in almost any image-processing software. In ImageJ it can be invoked interactively through the **Process** \triangleright **Enhance Contrast** menu (option **Equalize**). To avoid extreme contrast effects, the histogram

4 POINT OPERATIONS

Prog. 4.2

Linear histogram equalization (ImageJ plugin). First the histogram of the image `ip` is obtained using the standard ImageJ method `ip.getHistogram()` in line 7. In line 9, the cumulative histogram is computed “in place” based on the recursive definition in Eqn. (3.6). The `int` division in line 16 implicitly performs the required floor (`l l`) operation by truncation.

```
1  public void run(ImageProcessor ip) {
2      int M = ip.getWidth();
3      int N = ip.getHeight();
4      int K = 256; // number of intensity values
5
6      // compute the cumulative histogram:
7      int[] H = ip.getHistogram();
8      for (int j = 1; j < H.length; j++) {
9          H[j] = H[j - 1] + H[j];
10     }
11
12     // equalize the image:
13     for (int v = 0; v < N; v++) {
14         for (int u = 0; u < M; u++) {
15             int a = ip.get(u, v);
16             int b = H[a] * (K - 1) / (M * N); // s. Equation (4.12)
17             ip.set(u, v, b);
18         }
19     }
20 }
```

equalization in ImageJ by default⁵ cumulates the *square root* of the histogram entries using a modified cumulative histogram of the form

$$\tilde{H}(i) = \sum_{j=0}^i \sqrt{h(j)}. \quad (4.13)$$

4.6 Histogram Specification

Although widely implemented, the goal of linear histogram equalization—a uniform distribution of intensity values (as described in the previous section)—appears rather ad hoc, since good images virtually never show such a distribution. In most real images, the distribution of the pixel values is not even remotely uniform but is usually more similar, if at all, to perhaps a Gaussian distribution. The images produced by linear equalization thus usually appear quite unnatural, which renders the technique practically useless.

Histogram *specification* is a more general technique that modifies the image to match an arbitrary intensity distribution, including the histogram of a given image. This is particularly useful, for example, for adjusting a set of images taken by different cameras or under varying exposure or lighting conditions to give a similar impression in print production or when displayed. Similar to histogram equalization, this process relies on the alignment of the cumulative histograms by applying a homogeneous point operation. To be independent of the image size (i.e., the number of pixels), we first define *normalized* distributions, which we use in place of the original histograms.

⁵ The “classic” linear approach (see Eqn. (3.5)) is used when simultaneously keeping the Alt key pressed.

4.6.1 Frequencies and Probabilities

The value in each histogram cell describes the observed frequency of the corresponding intensity value, i.e., the histogram is a discrete *frequency distribution*. For a given image I of size $M \times N$, the sum of all histogram entries $\mathbf{h}(i)$ equals the number of image pixels,

$$\sum_i \mathbf{h}(i) = M \cdot N. \quad (4.14)$$

The associated *normalized* histogram,

$$\mathbf{p}(i) = \frac{\mathbf{h}(i)}{M \cdot N}, \quad \text{for } 0 \leq i < K, \quad (4.15)$$

is usually interpreted as the *probability distribution* or *probability density function* (pdf) of a random process, where $\mathbf{p}(i)$ is the probability for the occurrence of the pixel value i . The cumulative probability of i being any possible value is 1, and the distribution \mathbf{p} must thus satisfy

$$\sum_{i=0}^{K-1} \mathbf{p}(i) = 1. \quad (4.16)$$

The statistical counterpart to the cumulative histogram \mathbf{H} (Eqn. (3.5)) is the discrete *distribution function* $\mathbf{P}()$ (also called the *cumulative distribution function* or cdf),

$$\mathbf{P}(i) = \frac{\mathbf{H}(i)}{\mathbf{H}(K-1)} = \frac{\mathbf{H}(i)}{M \cdot N} = \sum_{j=0}^i \frac{\mathbf{h}(j)}{M \cdot N} = \sum_{j=0}^i \mathbf{p}(j), \quad (4.17)$$

for $i = 0, \dots, K-1$. The computation of the cdf from a given histogram \mathbf{h} is outlined in Alg. 4.1. The resulting function $\mathbf{P}(i)$ is (as the cumulative histogram) monotonically increasing and, in particular,

$$\mathbf{P}(0) = \mathbf{p}(0) \quad \text{and} \quad \mathbf{P}(K-1) = \sum_{i=0}^{K-1} \mathbf{p}(i) = 1. \quad (4.18)$$

This statistical formulation implicitly treats the generation of images as a random process whose exact properties are mostly unknown.⁶ However, the process is usually assumed to be homogeneous (independent of the image position); that is, each pixel value is the result of a “random experiment” on a single random variable i . The observed frequency distribution given by the histogram $\mathbf{h}(i)$ serves as a (coarse) estimate of the probability distribution $\mathbf{p}(i)$ of this random variable.

4.6.2 Principle of Histogram Specification

The goal of histogram specification is to modify a given image I_A by some point operation such that its distribution function \mathbf{P}_A matches

⁶ Statistical modeling of the image generation process has a long tradition (see, e.g., [128, Ch. 2]).

4 POINT OPERATIONS

Alg. 4.1

Calculation of the cumulative distribution function (cdf) $P(i)$ from a given histogram h of length K . See Prog. 4.3 (p. 73) for the corresponding Java implementation.

```

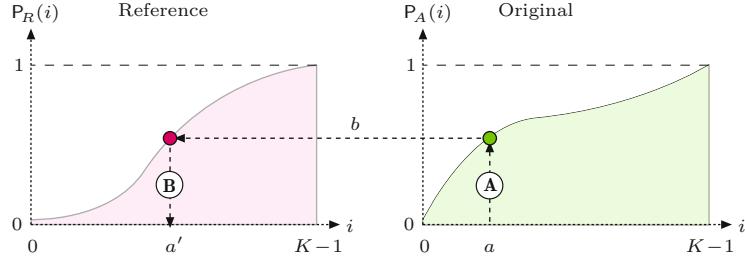
1: Cdf(h)           Returns the cumulative distribution function  $P(i) \in [0, 1]$  for a
   given histogram  $h(i)$ , with  $i = 0, \dots, K-1$ .
2: Let  $K \leftarrow \text{Size}(h)$ 
3: Let  $n \leftarrow \sum_{i=0}^{K-1} h(i)$ 
4: Create map  $P: [0, K-1] \mapsto \mathbb{R}$ 
5: Let  $c \leftarrow 0$ 
6: for  $i \leftarrow 0, \dots, K-1$  do
7:    $c \leftarrow c + h(i)$                                  $\triangleright$  cumulate histogram values
8:    $P(i) \leftarrow c/n$ 
9: return  $P$ .
```

Fig. 4.11

Principle of histogram specification. Given is the reference distribution P_R (left) and the distribution function for the original image P_A (right). The result is the mapping function $f_{hs}: a \rightarrow a'$ for a point operation, which replaces each pixel a in the original image I_A by a modified value a' . The process has two main steps:

- Ⓐ For each pixel value a , determine $b = P_A(a)$ from the right distribution function.
- Ⓑ a' is then found by inverting the left distribution function as $a' = P_R^{-1}(b)$.

In summary, the result is $f_{hs}(a) = a' = P_R^{-1}(P_A(a))$.



a *reference distribution* P_R as closely as possible. We thus look for a mapping function

$$a' = f_{hs}(a) \quad (4.19)$$

to convert the original image I_A by a point operation to a new image $I_{A'}$ with pixel values a' , such that its distribution function P'_A matches P_R , that is,

$$P'_A(i) \approx P_R(i), \quad \text{for } 0 \leq i < K. \quad (4.20)$$

As illustrated in Fig. 4.11, the desired mapping f_{hs} is found by combining the two distribution functions P_R and P_A (see [88, p. 180] for details). For a given pixel value a in the original image, we obtain the new pixel value a' as

$$a' = P_R^{-1}(P_A(a)) = P_R^{-1}(b) \quad (4.21)$$

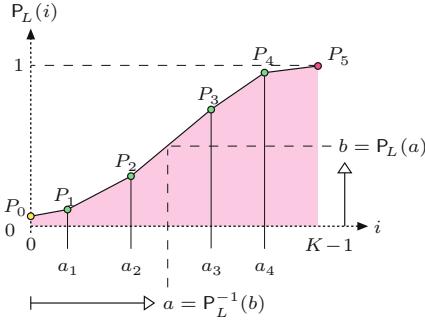
and thus the mapping f_{hs} (Eqn. (4.19)) is defined as

$$f_{hs}(a) = P_R^{-1}(P_A(a)), \quad \text{for } 0 \leq a < K. \quad (4.22)$$

This of course assumes that $P_R(i)$ is invertible, that is, that the function $P_R^{-1}(b)$ exists for $b \in [0, 1]$.

4.6.3 Adjusting to a Piecewise Linear Distribution

If the reference distribution P_R is given as a continuous, invertible function, then the mapping function f_{hs} can be obtained from Eqn. (4.22) without any difficulty. In practice, it is convenient to specify the (synthetic) reference distribution as a *piecewise linear* function $P_L(i)$; that is, as a sequence of $N+1$ coordinate pairs



4.6 HISTOGRAM SPECIFICATION

Fig. 4.12

Piecewise linear reference distribution. The function $P_L(i)$ is specified by $N = 5$ control points $(0, P_0), (a_1, P_1), \dots, (a_4, P_4)$, with $a_k < a_{k+1}$ and $P_k < P_{k+1}$. The final point P_5 is fixed at $(K-1, 1)$.

$$L = ((a_0, P_0), (a_1, P_1), \dots, (a_k, P_k), \dots, (a_N, P_N)),$$

each consisting of an intensity value a_k and the corresponding cumulative probability P_k . We assert that $0 \leq a_k < K$, $a_k < a_{k+1}$, and $0 \leq P_k < 1$. Also, the two endpoints (a_0, P_0) and (a_N, P_N) are fixed at

$$(0, P_0) \quad \text{and} \quad (K-1, 1),$$

respectively. To be invertible, the function must also be strictly monotonic, that is, $P_k < P_{k+1}$ for $0 \leq k < N$. Figure 4.12 shows an example of such a function, which is specified by $N = 5$ variable points (P_0, \dots, P_4) and a fixed end point P_5 and thus consists of $N = 5$ linear segments. The reference distribution can of course be specified at an arbitrary accuracy by inserting additional control points.

The intermediate values of $P_L(i)$ are obtained by linear interpolation between the control points as

$$P_L(i) = \begin{cases} P_m + (i - a_m) \cdot \frac{(P_{m+1} - P_m)}{(a_{m+1} - a_m)} & \text{for } 0 \leq i < K-1, \\ 1 & \text{for } i = K-1. \end{cases} \quad (4.23)$$

where $m = \max\{j \in [0, N-1] \mid a_j \leq i\}$ is the index of the line segment $(a_m, P_m) \rightarrow (a_{m+1}, P_{m+1})$, which overlaps the position i . For instance, in the example in Fig. 4.12, the point a lies within the segment that starts at point (a_2, P_2) ; i.e., $m = 2$.

For the histogram specification according to Eqn. (4.22), we also need the *inverse* distribution function $P_L^{-1}(b)$ for $b \in [0, 1]$. As we see from the example in Fig. 4.12, the function $P_L(i)$ is in general not invertible for values $b < P_L(0)$. We can fix this problem by mapping all values $b < P_L(0)$ to zero and thus obtain a “semi-inverse” of the reference distribution in Eqn. (4.23) as

$$P_L^{-1}(b) = \begin{cases} 0 & \text{for } 0 \leq b < P_L(0), \\ a_n + (b - P_n) \cdot \frac{(a_{n+1} - a_n)}{(P_{n+1} - P_n)} & \text{for } P_L(0) \leq b < 1, \\ K-1 & \text{for } b \geq 1. \end{cases} \quad (4.24)$$

Here $n = \max\{j \in \{0, \dots, N-1\} \mid P_j \leq b\}$ is the index of the line segment $(a_n, P_n) \rightarrow (a_{n+1}, P_{n+1})$, which overlaps the argument value b . The required mapping function f_{hs} for adapting a given image with intensity distribution P_A is finally specified, analogous to Eqn. (4.22), as

4 POINT OPERATIONS

Alg. 4.2

Histogram specification using a piecewise linear reference distribution. Given is the histogram \mathbf{h} of the original image and a piecewise linear reference distribution function, specified as a sequence of N control points L . The discrete mapping f_{hs} for the corresponding point operation is returned.

1: **MatchPiecewiseLinearHistogram**(\mathbf{h}, L)
Input: \mathbf{h} , histogram of the original image I ; L , reference distribution function, given as a sequence of $N + 1$ control points $L = [(a_0, P_0), (a_1, P_1), \dots, (a_N, P_N)]$, with $0 \leq a_k < K$, $0 \leq P_k \leq 1$, and $P_k < P_{k+1}$. Returns a discrete mapping $f_{\text{hs}}(a)$ to be applied to the original image I .

```

2:    $N \leftarrow \text{Size}(L) + 1$ 
3:   Let  $K \leftarrow \text{Size}(\mathbf{h})$ 
4:   Let  $\mathbf{P} \leftarrow \text{CDF}(\mathbf{h})$                                  $\triangleright \text{cdf for } \mathbf{h}$  (see Alg. 4.1)
5:   Create map  $f_{\text{hs}}: [0, K-1] \mapsto \mathbb{R}$             $\triangleright \text{function } f_{\text{hs}}$ 
6:   for  $a \leftarrow 0, \dots, K-1$  do
7:      $b \leftarrow \mathbf{P}(a)$ 
8:     if  $(b \leq P_0)$  then
9:        $a' \leftarrow 0$ 
10:      else if  $(b \geq 1)$  then
11:         $a' \leftarrow K-1$ 
12:      else
13:         $n \leftarrow N-1$ 
14:        while  $(n \geq 0) \wedge (P_n > b)$  do     $\triangleright \text{find line segment in } L$ 
15:           $n \leftarrow n - 1$ 
16:           $a' \leftarrow a_n + (b - P_n) \cdot \frac{(a_{n+1} - a_n)}{(P_{n+1} - P_n)}$      $\triangleright \text{see Eqn. 4.24}$ 
17:           $f_{\text{hs}}[a] \leftarrow a'$ 
18:   return  $f_{\text{hs}}$ .

```

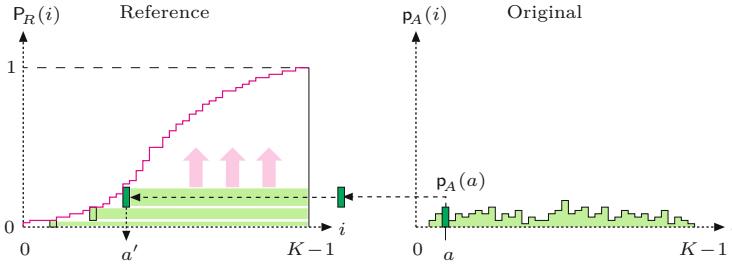
$$f_{\text{hs}}(a) = \mathbf{P}_L^{-1}(\mathbf{P}_A(a)), \quad \text{for } 0 \leq a < K. \quad (4.25)$$

The whole process of computing the pixel mapping function for a given image (histogram) and a piecewise linear target distribution is summarized in Alg. 4.2. A real example is shown in Fig. 4.14 (Sec. 4.6.5).

4.6.4 Adjusting to a Given Histogram (Histogram Matching)

If we want to adjust one image to the histogram of another image, the reference distribution function $\mathbf{P}_R(i)$ is not continuous and thus, in general, cannot be inverted (as required by Eqn. (4.22)). For example, if the reference distribution contains zero entries (i.e., pixel values k with probability $p(k) = 0$), the corresponding cumulative distribution function \mathbf{P} (just like the cumulative histogram) has intervals of constant value on which no inverse function value can be determined.

In the following, we describe a simple method for histogram matching that works with discrete reference distributions. The principal idea is graphically illustrated in Fig. 4.13. The mapping function f_{hs} is not obtained by inverting but by “filling in” the reference distribution function $\mathbf{P}_R(i)$. For each possible pixel value a , starting with $a = 0$, the corresponding probability $p_A(a)$ is stacked layer by layer “under” the reference distribution \mathbf{P}_R . The thickness of each horizontal bar for a equals the corresponding probability $p_A(a)$. The bar for a particular intensity value a with thickness $p_A(a)$ runs from



right to left, down to position a' , where it hits the reference distribution P_R . This position a' corresponds to the new pixel value to which a should be mapped.

Since the sum of all probabilities p_A and the maximum of the distribution function P_R are both 1 (i.e., $\sum_i p_A(i) = \max_i P_R(i) = 1$), all horizontal bars will exactly fit underneath the function P_R . One may also notice in Fig. 4.13 that the distribution value resulting at a' is identical to the cumulated probability $P_A(a)$. Given some intensity value a , it is therefore sufficient to find the minimum value a' , where the reference distribution $P_R(a')$ is greater than or equal to the cumulative probability $P_A(a)$, that is,

$$f_{hs}(a) = \min \{ j \mid (0 \leq j < K) \wedge (P_A(a) \leq P_R(j)) \}. \quad (4.26)$$

This results in a very simple method, which is summarized in Alg. 4.3. The corresponding Java implementation in Prog. 4.3, consists of the method `matchHistograms()`, which accepts the original histogram (`hA`) and the reference histogram (`hR`) and returns the resulting mapping function (`fhs`) specifying the required point operation.

Due to the use of normalized distribution functions, the *size* of the associated images is not relevant. The following code fragment demonstrates the use of the `matchHistograms()` method from Prog. 4.3 in ImageJ:

```
ImageProcessor ipA = ... // target image  $I_A$  (to be modified)
ImageProcessor ipR = ... // reference image  $I_R$ 

int[] hA = ipA.getHistogram(); // get histogram for  $I_A$ 
int[] hR = ipR.getHistogram(); // get histogram for  $I_R$ 

int[] fhs = matchHistograms(hA, hR); // mapping function  $f_{hs}(a)$ 

ipA.applyTable(fhs); // modify the target image  $I_A$ 
```

The original image `ipA` is modified in the last line by applying the mapping function f_{hs} (`fhs`) with the method `applyTable()` (see also p. 83).

4.6.5 Examples

Adjusting to a piecewise linear reference distribution

The first example in Fig. 4.14 shows the results of histogram specification for a continuous, piecewise linear reference distribution, as

4.6 HISTOGRAM SPECIFICATION

Fig. 4.13

Discrete histogram specification. The reference distribution P_R (left) is “filled” layer by layer from bottom to top and from right to left. For every possible intensity value a (starting from $a = 0$), the associated probability $p_A(a)$ is added as a horizontal bar to a stack accumulated ‘under’ the reference distribution P_R . The bar with thickness $p_A(a)$ is drawn from right to left down to the position a' , where the reference distribution P_R is reached. The function $f_{hs}()$ must map a to a' .

4 POINT OPERATIONS

Alg. 4.3

Histogram matching. Given are two histograms: the histogram \mathbf{h}_A of the target image I_A and a reference histogram \mathbf{h}_R , both of size K . The result is a discrete mapping function $f_{hs}()$ that, when applied to the target image, produces a new image with a distribution function similar to the reference histogram.

1: **MatchHistograms**($\mathbf{h}_A, \mathbf{h}_R$)

Input: \mathbf{h}_A , histogram of the target image I_A ; \mathbf{h}_R , reference histogram (the same size as \mathbf{h}_A). Returns a discrete mapping $f_{hs}(a)$ to be applied to the target image I_A .

```

2:    $K \leftarrow \text{Size}(\mathbf{h}_A)$ 
3:    $\mathbf{P}_A \leftarrow \text{CDF}(\mathbf{h}_A)$                                  $\triangleright$  c.d.f. for  $\mathbf{h}_A$  (Alg. 4.1)
4:    $\mathbf{P}_R \leftarrow \text{CDF}(\mathbf{h}_R)$                                  $\triangleright$  c.d.f. for  $\mathbf{h}_R$  (Alg. 4.1)
5:   Create map  $f_{hs}: [0, K-1] \mapsto \mathbb{R}$      $\triangleright$  pixel mapping function  $f_{hs}$ 
6:   for  $a \leftarrow 0, \dots, K-1$  do
7:      $j \leftarrow K-1$ 
8:     repeat
9:        $f_{hs}[a] \leftarrow j$ 
10:       $j \leftarrow j - 1$ 
11:      while ( $j \geq 0$ )  $\wedge (\mathbf{P}_A(a) \leq \mathbf{P}_R(j))$ 
12:   return  $f_{hs}$ .

```

described in Sec. 4.6.3. Analogous to Fig. 4.12, the actual distribution function \mathbf{P}_R (Fig. 4.14(f)) is specified as a polygonal line consisting of five control points $\langle a_k, q_k \rangle$ with coordinates

$$\begin{array}{ccccccc} k = & 0 & 1 & 2 & 3 & 4 & 5 \\ a_k = & 0 & 28 & 75 & 150 & 210 & 255 \\ q_k = & 0.002 & 0.050 & 0.250 & 0.750 & 0.950 & 1.000 \end{array}.$$

The resulting reference histogram (Fig. 4.14(c)) is a step function with ranges of constant values corresponding to the linear segments of the probability density function. As expected, the *cumulative* probability function for the modified image (Fig. 4.14(h)) is quite close to the reference function in Fig. 4.14(f), while the resulting *histogram* (Fig. 4.14(e)) shows little similarity with the reference histogram (Fig. 4.14(c)). However, as discussed earlier, this is all we can expect from a homogeneous point operation.

Adjusting to an arbitrary reference histogram

The example in Fig. 4.15 demonstrates this technique using synthetic reference histograms whose shape is approximately Gaussian. In this case, the reference distribution is not given as a continuous function but specified by a discrete histogram. We thus use the method described in Sec. 4.6.4 to compute the required mapping functions.

The target image used here was chosen intentionally for its poor quality, manifested by an extremely unbalanced histogram. The histograms of the modified images thus naturally show little resemblance to a Gaussian. However, the resulting *cumulative* histograms match nicely with the integral of the corresponding Gaussians, apart from the unavoidable irregularity at the center caused by the dominant peak in the original histogram.

Adjusting to another image

The third example in Fig. 4.16 demonstrates the adjustment of two images by matching their intensity histograms. One of the images is selected as the reference image I_R (Fig. 4.16(b)) and supplies the

```

1 int[] matchHistograms (int[] hA, int[] hR) {
2     // hA ... histogram  $h_A$  of the target image  $I_A$  (to be modified)
3     // hR ... reference histogram  $h_R$ 
4     // returns the mapping  $f_{hs}()$  to be applied to image  $I_A$ 
5
6     int K = hA.length;
7     double[] PA = Cdf(hA);           // get CDF of histogram  $h_A$ 
8     double[] PR = Cdf(hR);           // get CDF of histogram  $h_R$ 
9     int[] fhs = new int[K];          // mapping  $f_{hs}()$ 
10
11    // compute mapping function  $f_{hs}()$ :
12    for (int a = 0; a < K; a++) {
13        int j = K - 1;
14        do {
15            fhs[a] = j;
16            j--;
17        } while (j >= 0 && PA[a] <= PR[j]);
18    }
19    return fhs;
20 }
```

```

22 double[] Cdf (int[] h) {
23     // returns the cumul. distribution function for histogram h
24     int K = h.length;
25
26     int n = 0;                      // sum all histogram values
27     for (int i = 0; i < K; i++) {
28         n += h[i];
29     }
30
31     double[] P = new double[K];     // create CDF table P
32     int c = h[0];                  // cumulate histogram values
33     P[0] = (double) c / n;
34     for (int i = 1; i < K; i++) {
35         c += h[i];
36         P[i] = (double) c / n;
37     }
38     return P;
39 }
```

4.6 HISTOGRAM SPECIFICATION

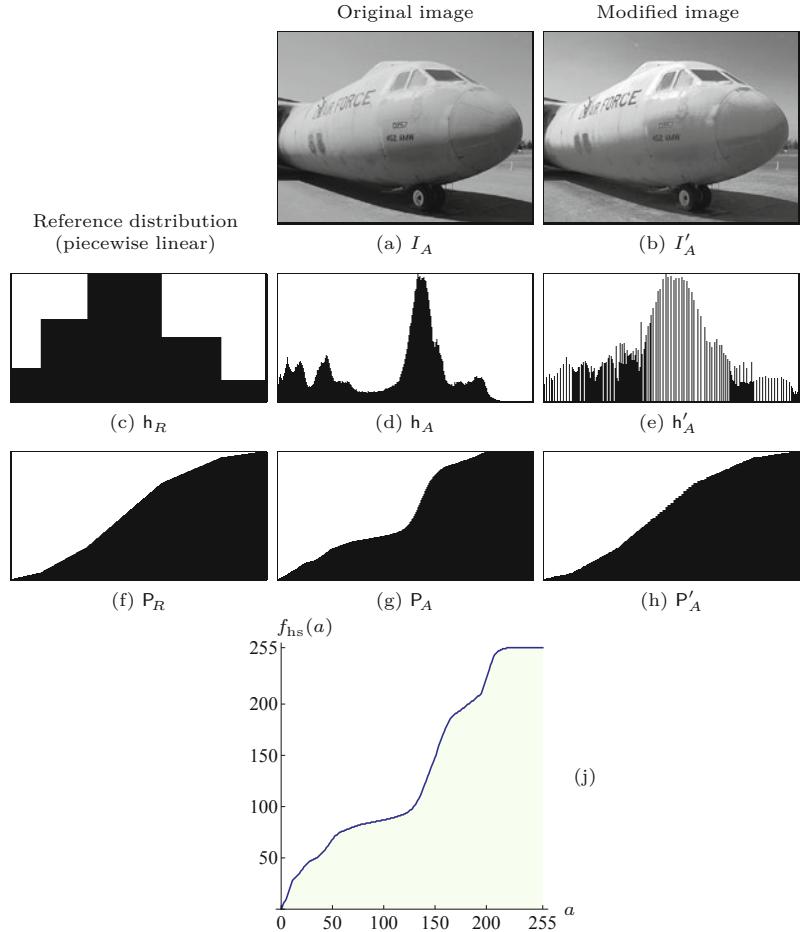
Prog. 4.3

Histogram matching (Java implementation of Alg. 4.3). The method `matchHistograms()` computes the mapping function `fhs` from the target histogram `hA` and the reference histogram `hR` (see Eqn. (4.26)). The method `Cdf()` computes the cumulative distribution function (cdf) for a given histogram (Eqn. (4.17)).

reference histogram h_R (Fig. 4.16(e)). The second (target) image I_A (Fig. 4.16(a)) is modified such that the resulting cumulative histogram matches the cumulative histogram of the reference image I_R . It can be expected that the final image $I_{A'}$ (Fig. 4.16(c)) and the reference image give a similar visual impression with regard to tonal range and distribution (assuming that both images show similar content).

Of course this method may be used to adjust multiple images to the same reference image (e.g., to prepare a series of similar photographs for a print project). For this purpose, one could either select a single representative image as a common reference or, alternatively, compute an “average” reference histogram from a set of typical images (see also Exercise 4.7).

Fig. 4.14
Histogram specification with a piecewise linear reference distribution. The target image I_A (a), its histogram (d), and distribution function P_A (g); the reference histogram h_R (c) and the corresponding distribution P_R (f); the modified image I'_A (b), its histogram h'_A (e), and the resulting distribution $P'_{A'}$ (h). Associated mapping function f_{hs} (j).

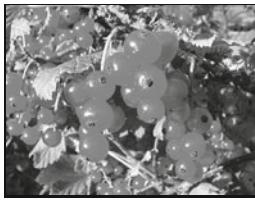
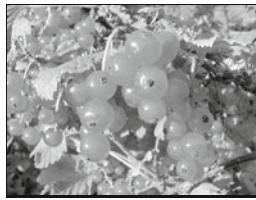


4.7 Gamma Correction

We have been using the terms “intensity” and “brightness” many times without really bothering with how the numeric pixel values in our images relate to these physical concepts, if at all. A pixel value may represent the amount of light falling onto a sensor element in a camera, the photographic density of film, the amount of light to be emitted by a monitor, the number of toner particles to be deposited by a printer, or any other relevant physical magnitude. In practice, the relationship between a pixel value and the corresponding physical quantity is usually complex and almost always nonlinear. In many imaging applications, it is important to know this relationship, at least approximately, to achieve consistent and reproducible results.

When applied to digital intensity images, the ideal is to have some kind of “calibrated intensity space” that optimally matches the human perception of intensity and requires a minimum number of bits to represent the required intensity range. Gamma correction denotes a simple point operation to compensate for the transfer characteristics of different input and output devices and to map them to a unified intensity space.

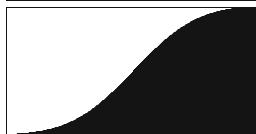
Original image

(a) I_A Gaussian ($\sigma = 50$)(b) I_{G50} Gaussian ($\sigma = 100$)(c) I_{G100}

Reference histogram

 $p_R(i)$ 

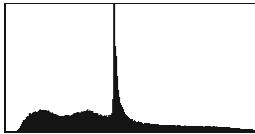
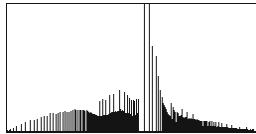
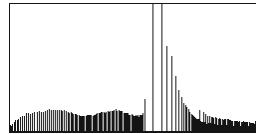
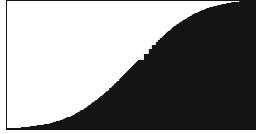
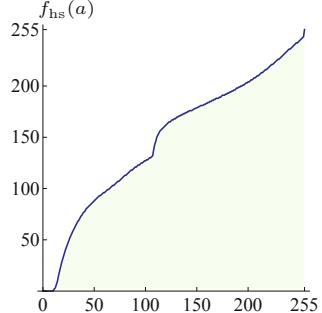
Cumulative reference histogram

 $P_R(i)$ 

(d)



(e)

(f) h_A (g) h_{G50} (h) h_{G100} (i) H_A (j) H_{G50} (k) H_{G100} 

(l)

4.7 GAMMA CORRECTION

Fig. 4.15

Histogram matching: adjusting to a synthetic histogram. Original image I_A (a), corresponding histogram (f), and cumulative histogram (i). Gaussian-shaped reference histograms with center $\mu = 128$ and $\sigma = 50$ (d) and $\sigma = 100$ (e), respectively. Resulting images after histogram matching, I_{G50} (b) and I_{G100} (c) with the corresponding histograms (g, h) and cumulative histograms (j, k). Associated mapping function f_{hs} (l).

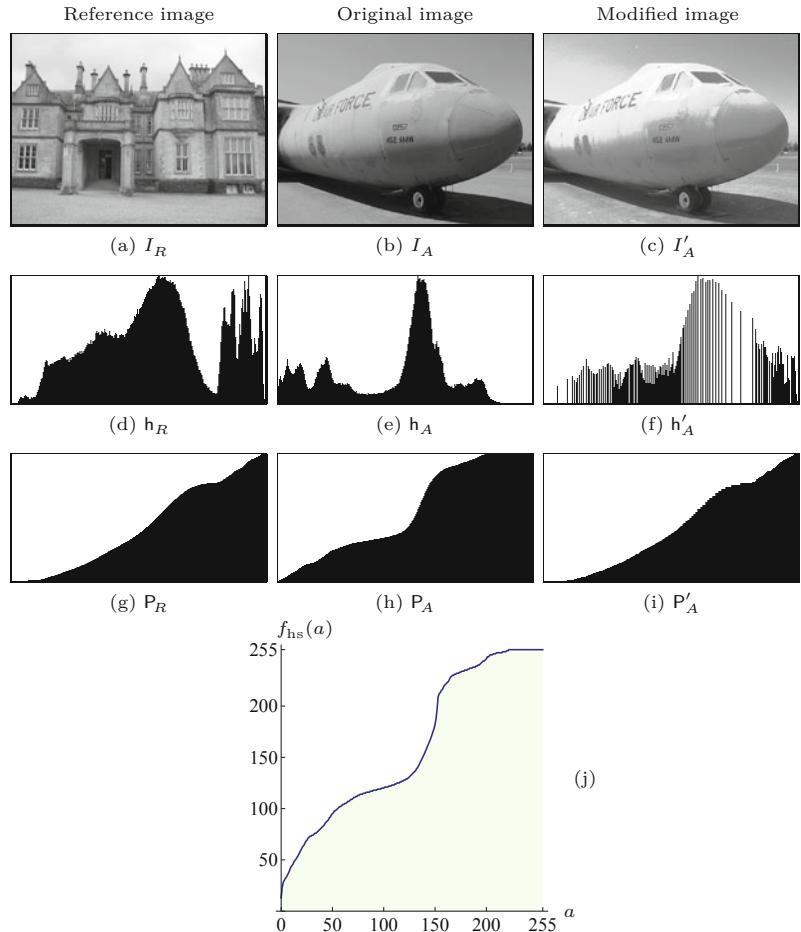
4.7.1 Why Gamma?

The term “gamma” originates from analog photography, where the relationship between the light energy and the resulting film density is approximately logarithmic. The “exposure function” (Fig. 4.17), specifying the relationship between the *logarithmic* light intensity and the resulting film density, is therefore approximately *linear* over a wide range of light intensities. The slope of this function within this linear range is traditionally referred to as the “gamma” of the photographic material. The same term was adopted later in televi-

4 POINT OPERATIONS

Fig. 4.16

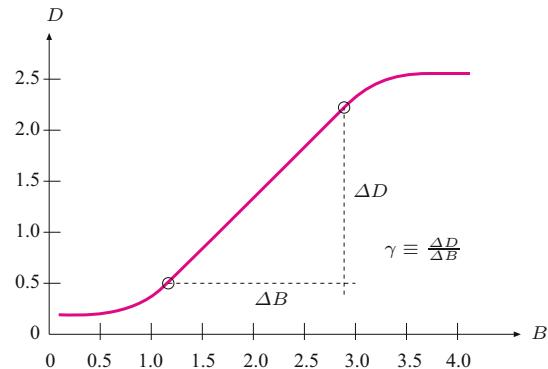
Histogram matching: adjusting to a reference image. The target image I_A (a) is modified by matching its histogram to the reference image I_R (b), resulting in the new image I'_A (c). The corresponding histograms h_A , h_R , $h'_{A'}$ (d-f) and cumulative histograms H_A , H_R , $P_{A'}$ (g-i) are shown. Notice the good agreement between the cumulative histograms of the reference and adjusted images (h,i). Associated mapping function f_{hs} (j).



sion broadcasting to describe the nonlinearities of the cathode ray tubes used in TV receivers, that is, to model the relationship between the amplitude (voltage) of the video signal and the emitted light intensity. To compensate for the nonlinearities of the receivers, a “gamma correction” was (and is) applied to the TV signal once before broadcasting in order to avoid the need for costly correction measures on the receiver side.

Fig. 4.17

Exposure function of photographic film. With respect to the logarithmic light intensity B , the resulting film density D is approximately linear over a wide intensity range. The slope ($\Delta D / \Delta B$) of this linear section of the function specifies the “gamma” (γ) value for a particular type of photographic material.



Gamma correction is based on the exponential function

$$f_\gamma(a) = a^\gamma, \quad (4.27)$$

where the parameter $\gamma \in \mathbb{R}$ is called the *gamma* value. If a is constrained to the interval $[0, 1]$, then—*independent of γ* —the value of $f_\gamma(a)$ also stays within $[0, 1]$, and the function always runs through the points $(0, 0)$ and $(1, 1)$. In particular, $f_\gamma(a)$ is the identity function for $\gamma = 1$, as shown in Fig. 4.18. The function runs *above* the diagonal for gamma values $\gamma < 1$, and *below* it for $\gamma > 1$. Controlled by a single continuous parameter (γ), the power function can thus “imitate” both logarithmic and exponential types of functions. Within the interval $[0, 1]$, the function is continuous and strictly monotonic, and also very simple to invert as

$$a = f_\gamma^{-1}(b) = b^{1/\gamma}, \quad (4.28)$$

since $b^{1/\gamma} = (a^\gamma)^{1/\gamma} = a^1 = a$. The inverse of the exponential function $f_\gamma^{-1}(b)$ is thus again an exponential function,

$$f_\gamma^{-1}(b) = f_{\bar{\gamma}}(b) = f_{1/\gamma}(b), \quad (4.29)$$

with the parameter $\bar{\gamma} = 1/\gamma$.

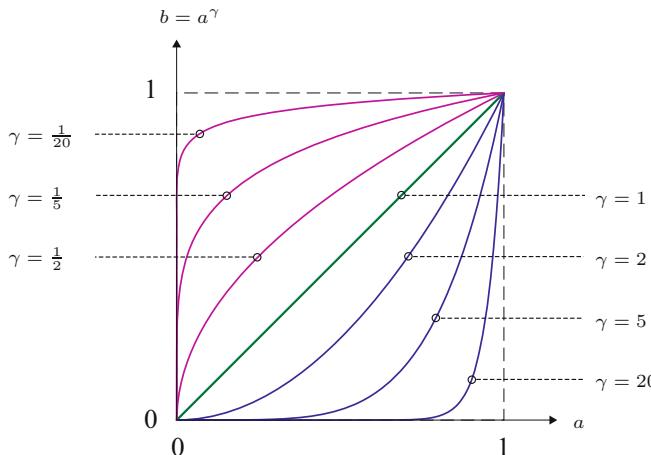


Fig. 4.18
Gamma correction function
 $f_\gamma(a) = a^\gamma$ for $a \in [0, 1]$ and
different gamma values.

4.7.3 Real Gamma Values

The actual gamma values of individual devices are usually specified by the manufacturers based on real measurements. For example, common gamma values for CRT monitors are in the range 1.8 to 2.8, with 2.4 as a typical value. Most LCD monitors are internally adjusted to similar values. Digital video and still cameras also emulate the transfer characteristics of analog film and photographic cameras by making internal corrections to give the resulting images an accustomed “look”.

In TV receivers, gamma values are standardized with 2.2 for analog NTSC and 2.8 for the PAL system (these values are theoretical; results of actual measurements are around 2.35). A gamma value of $1/2.2 \approx 0.45$ is the norm for cameras in NTSC as well as the EBU⁷ standards. The current international standard ITU-R BT.709⁸ calls for uniform gamma values of 2.5 in receivers and $1/1.956 \approx 0.51$ for cameras [76, 122]. The ITU 709 standard is based on a slightly modified version of the gamma correction (see Sec. 4.7.6).

Computers usually allow adjustment of the gamma value applied to the video output signals to adapt to a wide range of different monitors. Note, however, that the power function $f_\gamma()$ is only a coarse approximation to the actual transfer characteristics of any device, which may also not be the same for different color channels. Thus significant deviations may occur in practice, despite the careful choice of gamma settings. Critical applications, such as prepress or high-end photography, usually require additional calibration efforts based on exactly measured device profiles (see Sec. 14.7.4).

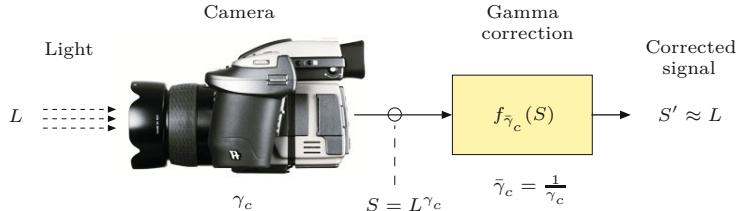
4.7.4 Applications of Gamma Correction

Let us first look at the simple example illustrated in Fig. 4.19. Assume that we use a digital camera with a nominal gamma value γ_c , meaning that its output signal s relates to the incident light intensity L as

$$S = L^{\gamma_c}. \quad (4.30)$$

Fig. 4.19

Principle of gamma correction.
To compensate the output signal S produced by a camera with nominal gamma value γ_c , a gamma correction is applied with $\bar{\gamma}_c = 1/\gamma_c$. The corrected signal S' is proportional to the received light intensity L .



To compensate the transfer characteristic of this camera (i.e., to obtain a measurement S' that is proportional to the original light intensity L), the camera signal S is subject to a gamma correction with the inverse of the camera's gamma value $\bar{\gamma}_c = 1/\gamma_c$ and thus

$$S' = f_{\bar{\gamma}_c}(S) = S^{1/\gamma_c}. \quad (4.31)$$

The resulting signal

$$S' = S^{1/\gamma_c} = (L^{\gamma_c})^{1/\gamma_c} = L^{(\gamma_c \frac{1}{\gamma_c})} = L^1$$

is obviously proportional (in theory even identical) to the original light intensity L . Although this example is quite simplistic, it still demonstrates the general rule, which holds for output devices as well:

⁷ European Broadcast Union (EBU).

⁸ International Telecommunications Union (ITU).

The transfer characteristic of an input or output device with specified gamma value γ is compensated for by a gamma correction with $\bar{\gamma} = 1/\gamma$.

4.7 GAMMA CORRECTION

In the aforementioned, we have implicitly assumed that all values are strictly in the range $[0, 1]$, which usually is not the case in practice. When working with digital images, we have to deal with discrete pixel values, for example, in the range $[0, 255]$ for 8-bit images. In general, performing a gamma correction

$$b \leftarrow f_{\text{gc}}(a, \gamma),$$

on a pixel value $a \in [0, a_{\max}]$ and a gamma value $\gamma > 0$ requires the following three steps:

1. Scale a linearly to $\hat{a} \in [0, 1]$.
2. Apply the gamma correction function to \hat{a} : $\hat{b} \leftarrow \hat{a}^\gamma$.
3. Scale $\hat{b} \in [0, 1]$ linearly back to $b \in [0, a_{\max}]$.

Formulated in a more compact way, the corrected pixel value b is obtained from the original value a as

$$b \leftarrow \left(\frac{a}{a_{\max}} \right)^\gamma \cdot a_{\max}. \quad (4.32)$$

[Figure 4.20](#) illustrates the typical role of gamma correction in the digital work flow with two input (camera, scanner) and two output devices (monitor, printer), each with its individual gamma value. The central idea is to correct all images to be processed and stored in a device-independent, standardized intensity space.

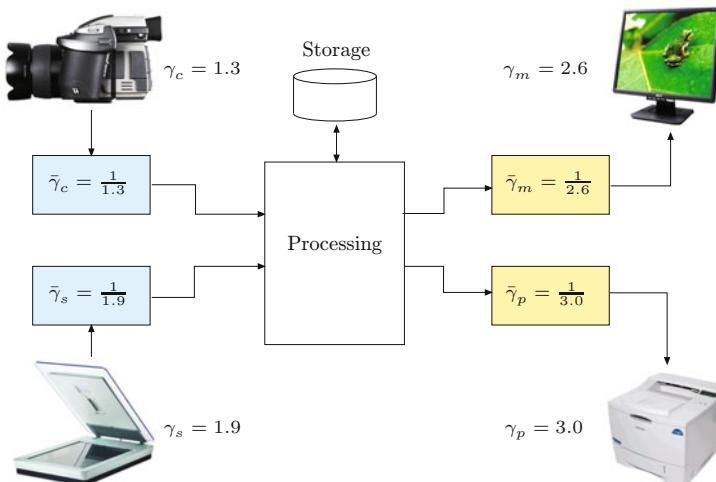


Fig. 4.20

Gamma correction in the digital imaging work flow. Images are processed and stored in a “linear” intensity space, where gamma correction is used to compensate for the transfer characteristic of each input and output device. (The gamma values shown are examples only.)

4.7.5 Implementation

Program 4.4 shows the implementation of gamma correction as an ImageJ plugin for 8-bit grayscale images. The mapping function $f_{\text{gc}}(a, \gamma)$ is computed as a lookup table (Fgc), which is then applied to the image using the method `applyTable()` to perform the actual point operation (see also Sec. 4.8.1).

4 POINT OPERATIONS

Prog. 4.4

Implementation of gamma correction in the `run()` method of an ImageJ plugin. The corrected intensity values `b` are only computed once and stored in the lookup table `Fgc` (line 15). The gamma value `GAMMA` is constant. The actual point operation is performed by calling the ImageJ method `applyTable(Fgc)` on the image object `ip` (line 18).

```

1  public void run(ImageProcessor ip) {
2      // works for 8-bit images only
3      int K = 256;
4      int aMax = K - 1;
5      double GAMMA = 2.8;
6
7      // create and fill the lookup table:
8      int[] Fgc = new int[K];
9
10     for (int a = 0; a < K; a++) {
11         double aa = (double) a / aMax;           // scale to [0, 1]
12         double bb = Math.pow(aa, GAMMA);        // power function
13         // scale back to [0, 255]:
14         int b = (int) Math.round(bb * aMax);
15         Fgc[a] = b;
16     }
17
18     ip.applyTable(Fgc); // modify the image
19 }
```

4.7.6 Modified Gamma Correction

A subtle problem with the simple power function $f_\gamma(a) = a^\gamma$ (Eqn. (4.27)) appears if we take a closer look at the *slope* of this function, expressed by its first derivative,

$$f'_\gamma(a) = \gamma \cdot a^{(\gamma-1)},$$

which for $a = 0$ has the values

$$f'_\gamma(0) = \begin{cases} 0 & \text{for } \gamma > 1, \\ 1 & \text{for } \gamma = 1, \\ \infty & \text{for } \gamma < 1. \end{cases} \quad (4.33)$$

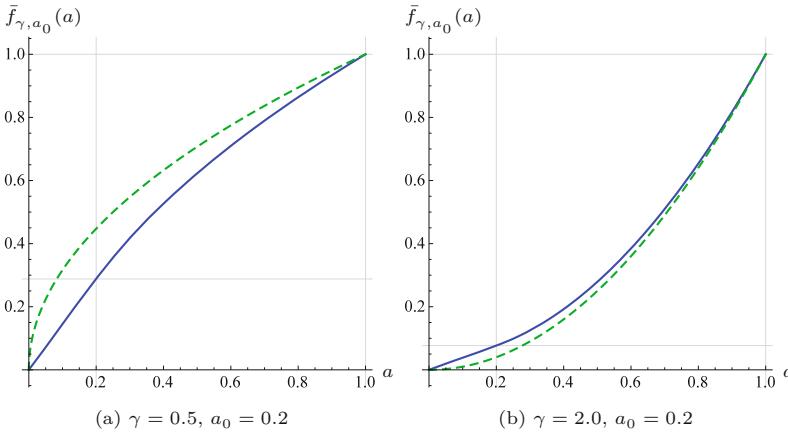
The tangent to the function at the origin is thus horizontal ($\gamma > 1$), diagonal ($\gamma = 1$), or vertical ($\gamma < 1$), with no intermediate values. For $\gamma < 1$, this causes extremely high amplification of small intensity values and thus increased noise in dark image regions. Theoretically, this also means that the power function is generally not invertible at the origin.

A common solution to this problem is to replace the lower part ($0 \leq a \leq a_0$) of the power function by a *linear* segment with constant slope and to continue with the ordinary power function for $a > a_0$. The resulting modified gamma correction function,

$$\bar{f}_{\gamma, a_0}(a) = \begin{cases} s \cdot a & \text{for } 0 \leq a \leq a_0, \\ (1+d) \cdot a^\gamma - d & \text{for } a_0 < a \leq 1, \end{cases} \quad (4.34)$$

$$\text{with } s = \frac{\gamma}{a_0(\gamma-1) + a_0^{(1-\gamma)}} \quad \text{and} \quad d = \frac{1}{a_0^\gamma(\gamma-1) + 1} - 1 \quad (4.35)$$

thus consists of a *linear* section (for $0 \leq a \leq a_0$) and a *nonlinear* section (for $a_0 < a \leq 1$) that connect smoothly at the transition point



4.7 GAMMA CORRECTION

Fig. 4.21
Modified gamma correction.
The mapping $\bar{f}_{\gamma,a_0}(a)$ consists
of a linear segment with fixed
slope s between $a = 0$ and
 $a = a_0$, followed by a power
function with parameter γ
(Eqn. (4.34)). The dashed
lines show the ordinary power
functions for the same gamma
values.

$a = a_0$. The linear slope s and the parameter d are determined by the requirement that the two function segments must have identical values as well as identical slopes (first derivatives) at $a = a_0$ to produce a continuous function. The function in Eqn. (4.34) is thus fully specified by the two parameters a_0 and γ .

Figure 4.21 shows two examples of the modified gamma correction $\bar{f}_{\gamma,a_0}()$ with values $\gamma = 0.5$ and $\gamma = 2.0$, respectively. In both cases, the transition point is at $a_0 = 0.2$. For comparison, the figure also shows the ordinary gamma correction $f_\gamma(a)$ for the same gamma values (dashed lines), whose slope at the origin is ∞ (Fig. 4.21(a)) and zero (Fig. 4.21(b)), respectively.

Gamma correction in common standards

The modified gamma correction is part of several modern imaging standards. In practice, however, the values of a_0 are considerably smaller than the ones used for the illustrative examples in Fig. 4.21, and γ is chosen to obtain a good overall match to the desired correction function. For example, the ITU-BT.709 specification [122] mentioned in Sec. 4.7.3 specifies the parameters

$$\gamma = \frac{1}{2.222} \approx 0.45 \quad \text{and} \quad a_0 = 0.018, \quad (4.36)$$

with the corresponding slope and offset values $s = 4.50681$ and $d = 0.0991499$, respectively (Eqn. (4.35)). The resulting correction function $\bar{f}_{\text{ITU}}(a)$ has a *nominal* gamma value of 0.45, which corresponds to the *effective* gamma value $\gamma_{\text{eff}} = 1/1.956 \approx 0.511$. The gamma correction in the sRGB standard [224] is specified on the same basis (with different parameters; see Sec. 14.4).

Figure 4.22 shows the actual correction functions for the ITU and sRGB standards, respectively, each in comparison with the equivalent ordinary gamma correction. The ITU function (Fig. 4.22(a)) with $\gamma = 0.45$ and $a_0 = 0.018$ corresponds to an ordinary gamma correction with effective gamma value $\gamma_{\text{eff}} = 0.511$ (dashed line). The curves for sRGB (Fig. 4.22(b)) differ only by the parameters γ and a_0 , as summarized in Table 4.1.

4 POINT OPERATIONS

Fig. 4.22

Gamma correction functions specified by the ITU-R BT.709 (a) and sRGB (b) standards. The continuous plot shows the modified gamma correction with the nominal γ values and transition points a_0 .

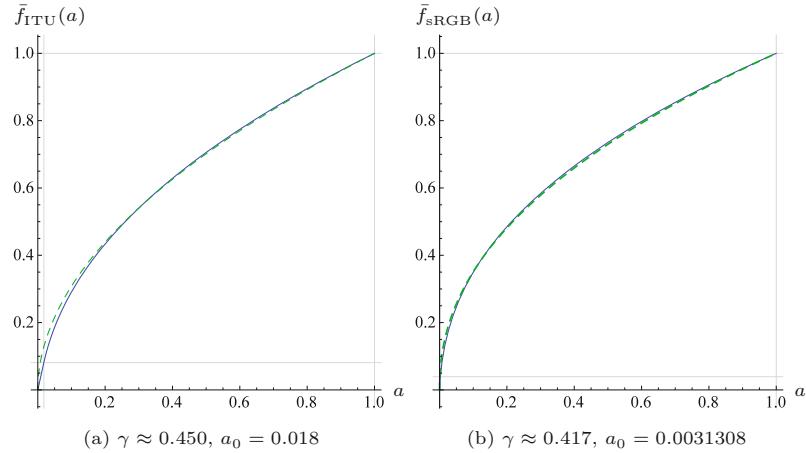


Table 4.1

Gamma correction parameters for the ITU and sRGB standards based on the modified mapping in Eqns. (4.34) and (4.35).

Standard	Nominal gamma value γ	a_0	s	d	Effective gamma value γ_{eff}
ITU-R BT.709	$1/2.222 \approx 0.450$	0.018	4.50	0.099	$1/1.956 \approx 0.511$
sRGB	$1/2.400 \approx 0.417$	0.0031308	12.92	0.055	$1/2.200 \approx 0.455$

Inverting the modified gamma correction

To invert the modified gamma correction of the form $b = \bar{f}_{\gamma, a_0}(a)$ (Eqn. (4.34)), we need the inverse of the function $\bar{f}_{\gamma, a_0}()$, which is again defined in two parts,

$$\bar{f}_{\gamma, a_0}^{-1}(b) = \begin{cases} b/s & \text{for } 0 \leq b \leq s \cdot a_0, \\ \left(\frac{b+d}{1+d}\right)^{1/\gamma} & \text{for } s \cdot a_0 < b \leq 1. \end{cases} \quad (4.37)$$

s and d are the quantities defined in Eqn. (4.35) and thus

$$a = \bar{f}_{\gamma, a_0}^{-1}(\bar{f}_{\gamma, a_0}(a)) \quad \text{for } a \in [0, 1], \quad (4.38)$$

with the *same* value γ being used in both functions. The inverse gamma correction function is required in particular for transforming between different color spaces if nonlinear (i.e., gamma-corrected) component values are involved (see also Sec. 14.2).

4.8 Point Operations in ImageJ

Several important types of point operations are already implemented in ImageJ, so there is no need to program every operation manually (as shown in Prog. 4.4). In particular, it is possible in ImageJ to apply point operations efficiently by using tabulated functions, to use built-in standard functions for point operations on single images, and to apply arithmetic operations on pairs of images. These issues are described briefly in the remaining parts of this section.

4.8.1 Point Operations with Lookup Tables

Some point operations require complex computations for each pixel, and the processing of large images may be quite time-consuming. If

the point operation is *homogeneous* (i.e., independent of the pixel coordinates), the value of the mapping function can be precomputed for every possible pixel value and stored in a lookup table, which may then be applied very efficiently to the image. A lookup table \mathbf{L} represents a discrete mapping (function f) from the original to the new pixel values,

$$\mathbf{F} : [0, K-1] \xrightarrow{f} [0, K-1]. \quad (4.39)$$

For a point operation specified by a particular pixel mapping function $a' = f(a)$, the table \mathbf{L} is initialized with the values

$$\mathbf{F}[a] \leftarrow f(a), \quad \text{for } 0 \leq a < K. \quad (4.40)$$

Thus the K table elements of \mathbf{F} need only be computed once, where typically $K = 256$. Performing the actual point operation only requires a simple (and quick) table lookup in \mathbf{F} at each pixel, that is,

$$I'(u, v) \leftarrow \mathbf{F}[I(u, v)], \quad (4.41)$$

which is much more efficient than any individual function call. ImageJ provides the method

```
void applyTable(int[] F)
```

for objects of type `ImageProcessor`, which requires a lookup table F as a 1D `int` array of size K (see Prog. 4.4 on page 80 for an example). The advantage of this approach is obvious: for an 8-bit image, for example, the mapping function is evaluated only 256 times (independent of the image size) and not a million times or more as in the case of a large image. The use of lookup tables for implementing point operations thus always makes sense if the number of image pixels ($M \times N$) is greater than the number of possible pixel values K (which is usually the case).

4.8.2 Arithmetic Operations

ImageJ implements a set of common arithmetic operations as methods for the class `ImageProcessor`, which are summarized in Table 4.2. In the following example, the image is multiplied by a scalar constant (1.5) to increase its contrast:

```
ImageProcessor ip = ... //some image
ip.multiply(1.5);
```

The image `ip` is destructively modified by all of these methods, with the results being limited (clamped) to the minimum and maximum pixel values, respectively.

4.8.3 Point Operations Involving Multiple Images

Point operations may involve more than one image at once, with arithmetic operations on the pixels of *pairs* of images being a special but important case. For example, we can express the pointwise *addition* of two images I_1 and I_2 (of identical size) to create a new image I' as

4 POINT OPERATIONS

Table 4.2

ImageJ methods for arithmetic operations applicable to objects of type `ImageProcessor`.

<code>void abs()</code>	$I(u, v) \leftarrow I(u, v) $
<code>void add(int p)</code>	$I(u, v) \leftarrow I(u, v) + p$
<code>void gamma(double g)</code>	$I(u, v) \leftarrow (I(u, v)/255)^g \cdot 255$
<code>void invert(int p)</code>	$I(u, v) \leftarrow 255 - I(u, v)$
<code>void log()</code>	$I(u, v) \leftarrow \log_{10}(I(u, v))$
<code>void max(double s)</code>	$I(u, v) \leftarrow \max(I(u, v), s)$
<code>void min(double s)</code>	$I(u, v) \leftarrow \min(I(u, v), s)$
<code>void multiply(double s)</code>	$I(u, v) \leftarrow \text{round}(I(u, v) \cdot s)$
<code>void sqr()</code>	$I(u, v) \leftarrow I(u, v)^2$
<code>void sqrt()</code>	$I(u, v) \leftarrow \sqrt{I(u, v)}$

$$I'(u, v) \leftarrow I_1(u, v) + I_2(u, v) \quad (4.42)$$

for all positions (u, v) . In general, any function $f(a_1, a_2, \dots, a_n)$ over n pixel values a_i may be defined to perform pointwise combinations of n images, that is,

$$I'(u, v) \leftarrow f(I_1(u, v), I_2(u, v), \dots, I_n(u, v)). \quad (4.43)$$

Of course, most arithmetic operations on multiple images can also be implemented as successive binary operations on pairs of images.

4.8.4 Methods for Point Operations on Two Images

ImageJ supplies a single method for implementing arithmetic operations on pairs of images,

```
copyBits(ImageProcessor ip2, int u, int v, int mode),
```

which applies the binary operation specified by the transfer mode parameter `mode` to all pixel pairs taken from the *source image* `ip2` and the *target image* (the image on which this method is invoked) and stores the result in the target image. u, v are the coordinates where the source image is inserted into the target image (usually $u = v = 0$). The following code segment demonstrates the addition of two images:

```
ImageProcessor ip1 = ... // target image ( $I_1$ )
ImageProcessor ip2 = ... // source image ( $I_2$ )
...
ip1.copyBits(ip2, 0, 0, Blitter.ADD); //  $I_1 \leftarrow I_1 + I_2$ 
// ip1 holds the result, ip2 is unchanged
...
```

In this operation, the target image `ip1` is destructively modified, while the source image `ip2` remains unchanged. The constant `ADD` is one of several arithmetic transfer modes defined by the `Blitter` interface (see [Table 4.3](#)). In addition, `Blitter` defines (bitwise) logical operations, such as `OR` and `AND`. For arithmetic operations, the `copyBits()` method limits the results to the admissible range of pixel values (of the target image). Also note that (except for target images of type `FloatProcessor`) the results are *not* rounded but truncated to integer values.

ADD	$I_1(u, v) \leftarrow I_1(u, v) + I_2(u, v)$
AVERAGE	$I_1(u, v) \leftarrow (I_1(u, v) + I_2(u, v)) / 2$
COPY	$I_1(u, v) \leftarrow I_2(u, v)$
DIFFERENCE	$I_1(u, v) \leftarrow I_1(u, v) - I_2(u, v) $
DIVIDE	$I_1(u, v) \leftarrow I_1(u, v) / I_2(u, v)$
MAX	$I_1(u, v) \leftarrow \max(I_1(u, v), I_2(u, v))$
MIN	$I_1(u, v) \leftarrow \min(I_1(u, v), I_2(u, v))$
MULTIPLY	$I_1(u, v) \leftarrow I_1(u, v) \cdot I_2(u, v)$
SUBTRACT	$I_1(u, v) \leftarrow I_1(u, v) - I_2(u, v)$

4.8 POINT OPERATIONS IN IMAGEJ

Table 4.3

Arithmetic operations and corresponding transfer mode constants for `ImageProcessor`'s `copyBits()` method. Example: `ip1.copyBits(ip2, 0, 0, Blitter.ADD)`.

4.8.5 ImageJ Plugins Involving Multiple Images

ImageJ provides two types of plugin: a generic plugin (`PlugIn`), which can be run without any open image, and plugins of type `PlugInFilter`, which apply to a single image. In the latter case, the currently active image is passed as an object of type `ImageProcessor` (or any of its subclasses) to the plugin's `run()` method (see also Sec. 2.2.3).

If two or more images I_1, I_2, \dots, I_k are to be combined by a plugin program, only a single image I_1 can be passed directly to the plugin's `run()` method, but not the additional images I_2, \dots, I_k . The usual solution is to make the plugin open a dialog window to let the user select the remaining images interactively. This is demonstrated in the following example plugin for transparently blending two images.

Example: Linear blending

Linear blending is a simple method for continuously mixing two images, I_{BG} and I_{FG} . The background image I_{BG} is covered by the foreground image I_{FG} , whose transparency is controlled by the value α in the form

$$I'(u, v) = \alpha \cdot I_{\text{BG}}(u, v) + (1-\alpha) \cdot I_{\text{FG}}(u, v), \quad (4.44)$$

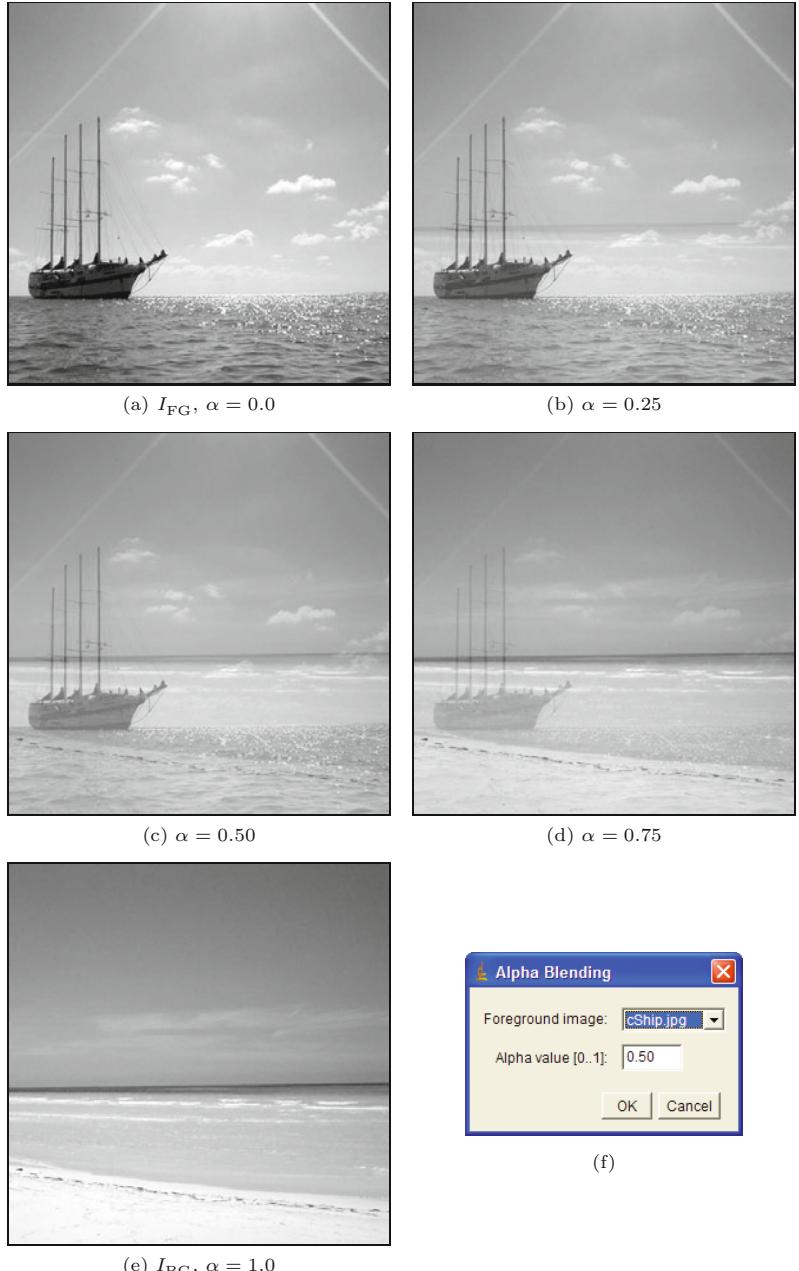
with $0 \leq \alpha \leq 1$. For $\alpha = 0$, the foreground image I_{FG} is nontransparent (opaque) and thus entirely hides the background image I_{BG} . Conversely, the image I_{FG} is fully transparent for $\alpha = 1$ and only I_{BG} is visible. All α values between 0 and 1 result in a weighted sum of the corresponding pixel values taken from I_{BG} and I_{FG} (Eqn. (4.44)).

Figure 4.23 shows the results of linear blending for different α values. The Java code for the corresponding implementation (as an ImageJ plugin) is listed in Prog. 4.5. The background image (`bgIp`) is passed directly to the plugin's `run()` method. The second (foreground) image and the α value are specified interactively by creating an instance of the ImageJ class `GenericDialog`, which allows the simple implementation of dialog windows with various types of input fields.

4 POINT OPERATIONS

Fig. 4.23

Linear blending example. Foreground image I_{FG} (a) and background image (I_{BG}) (e); blended images for transparency values $\alpha = 0.25, 0.50$, and 0.75 (b-d) and dialog window (f) produced by GenericDialog (see Prog. 4.5).



4.9 Exercises

Exercise 4.1. Implement the auto-contrast operation as defined in Eqns. (4.9)–(4.11) as an ImageJ plugin for an 8-bit grayscale image. Set the quantile p of pixels to be saturated at both ends of the intensity range (0 and 255) to $p = p_{lo} = p_{hi} = 1\%$.

Exercise 4.2. Modify the histogram equalization plugin in Prog. 4.2 to use a lookup table (Sec. 4.8.1) for computing the point operation.

Exercise 4.3. Implement the histogram equalization as defined in Eqn. (4.12), but use the *modified* cumulative histogram defined in Eqn. (4.13), cumulating the square root of the histogram entries. Compare the results to the standard (linear) approach by plotting the resulting histograms and cumulative histograms as shown in Fig. 4.10.

Exercise 4.4. Show formally that (a) a linear histogram equalization (Eqn. (4.12)) does not change an image that already has a uniform intensity distribution and (b) that any repeated application of histogram equalization to the same image causes no more changes.

Exercise 4.5. Show that the linear histogram equalization (Sec. 4.5) is only a special case of histogram specification (Sec. 4.6).

Exercise 4.6. Implement the histogram specification using a piecewise linear reference distribution function, as described in Sec. 4.6.3. Define a new object class with all necessary instance variables to represent the distribution function and implement the required functions $P_L(i)$ (Eqn. (4.23)) and $P_L^{-1}(b)$ (Eqn. (4.24)) as methods of this class.

Exercise 4.7. Using a histogram specification for adjusting *multiple* images (Sec. 4.6.4), one could either use one typical image as the reference or compute an “average” reference histogram from a set of images. Implement the second approach and discuss its possible advantages (or disadvantages).

Exercise 4.8. Implement the modified gamma correction (see Eqn. (4.34)) as an ImageJ plugin with variable values for γ and a_0 using a lookup table as shown in Prog. 4.4.

Exercise 4.9. Show that the modified gamma correction function $f_{\gamma, a_0}(a)$, with the parameters defined in Eqns. (4.34)–(4.35), is C1-continuous (i.e., both the function itself and its first derivative are continuous).

4 POINT OPERATIONS

Prog. 4.5

ImageJ-Plugin (Linear Blending). A background image is transparently blended with a selected foreground image. The plugin is applied to the (currently active) background image, and the foreground image must also be open when the plugin is started. The background image (`bgIp`), which is passed to the plugin's `run()` method, is multiplied with α (line 22).

The foreground image (`fgIP`, selected in part 2) is first duplicated (line 20) and then multiplied with $(1 - \alpha)$ (line 21). Thus the original foreground image is not modified.

The final result is obtained by adding the two weighted images (line 23). To select the foreground image, a list of currently open images and image titles is obtained (lines 30–32). Then a dialog object (of type `GenericDialog`) is created and opened for specifying the foreground image (`fgIm`) and the α value (lines 36–46).

```
1 import ij.ImagePlus;
2 import ij.gui.GenericDialog;
3 import ij.plugin.filter.PlugInFilter;
4 import ij.process.Blitter;
5 import ij.process.ImageProcessor;
6 import imagingbook.lib.ij.IjUtils;
7
8 public class Linear_Blending implements PlugInFilter {
9     static double alpha = 0.5; // transparency of foreground image
10    ImagePlus fgIm; // foreground image (to be selected)
11
12    public int setup(String arg, ImagePlus im) {
13        return DOES_8G;
14    }
15
16    public void run(ImageProcessor ipBG) { // ipBG =  $I_{BG}$ 
17        if(runDialog()) {
18            ImageProcessor ipFG = // ipFG =  $I_{FG}$ 
19                fgIm.getProcessor().convertToByte(false);
20            ipFG = ipFG.duplicate();
21            ipFG.multiply(1 - alpha); //  $I_{FG} \leftarrow I_{FG} \cdot (1 - \alpha)$ 
22            ipBG.multiply(alpha); //  $I_{BG} \leftarrow I_{BG} \cdot \alpha$ 
23            ipBG.copyBits(ipFG, 0, 0, Blitter.ADD); //  $I_{BG} \leftarrow I_{BG} + I_{FG}$ 
24        }
25    }
26
27    boolean runDialog() {
28        // get list of open images and their titles:
29        ImagePlus[] openImages = IjUtils.getOpenImages(true);
30        String[] imageTitles = new String[openImages.length];
31        for (int i = 0; i < openImages.length; i++) {
32            imageTitles[i] = openImages[i].getShortTitle();
33        }
34        // create the dialog and show:
35        GenericDialog gd =
36            new GenericDialog("Linear Blending");
37        gd.addChoice("Foreground image:",
38            imageTitles, imageTitles[0]);
39        gd.addNumericField("Alpha value [0..1]:", alpha, 2);
40        gd.showDialog();
41
42        if (gd.wasCanceled())
43            return false;
44        else {
45            fgIm = openImages[gd.getNextChoiceIndex()];
46            alpha = gd.getNextNumber();
47            return true;
48        }
49    }
50 }
```

Filters

The essential property of point operations (discussed in the previous chapter) is that each new pixel value only depends on the original pixel at the *same* position. The capabilities of point operations are limited, however. For example, they cannot accomplish the task of *sharpening* or *smoothing* an image (Fig. 5.1). This is what filters can do. They are similar to point operations in the sense that they also produce a 1:1 mapping of the image coordinates, that is, the geometry of the image does not change.

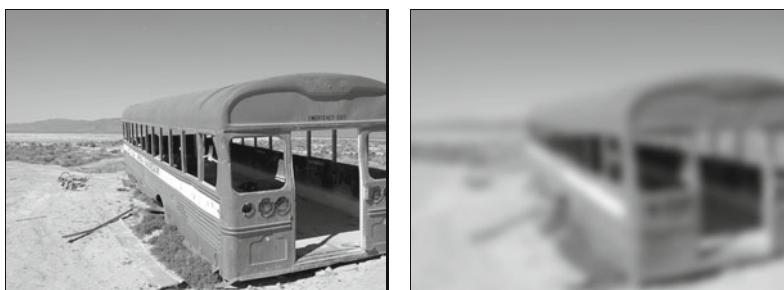


Fig. 5.1

No point operation can blur or sharpen an image. This is an example of what filters can do. Like point operations, filters do not modify the geometry of an image.

5.1 What is a Filter?

The main difference between filters and point operations is that filters generally use more than one pixel from the source image for computing each new pixel value. Let us first take a closer look at the task of smoothing an image. Images look sharp primarily at places where the local intensity rises or drops sharply (i.e., where the difference between neighboring pixels is large). On the other hand, we perceive an image as blurred or fuzzy where the local intensity function is smooth.

A first idea for smoothing an image could thus be to simply replace every pixel by the *average* of its neighboring pixels. To determine the new pixel value in the smoothed image $I'(u, v)$, we use the

original pixel $I(u, v) = p_0$ at the same position plus its eight neighboring pixels p_1, p_2, \dots, p_8 to compute the arithmetic mean of these nine values,

$$I'(u, v) \leftarrow \frac{p_0 + p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + p_8}{9}. \quad (5.1)$$

Expressed in relative image coordinates this is

$$\begin{aligned} I'(u, v) \leftarrow \frac{1}{9} \cdot [& I(u-1, v-1) + I(u, v-1) + I(u+1, v-1) + \\ & I(u-1, v) + I(u, v) + I(u+1, v) + \\ & I(u-1, v+1) + I(u, v+1) + I(u+1, v+1)], \end{aligned} \quad (5.2)$$

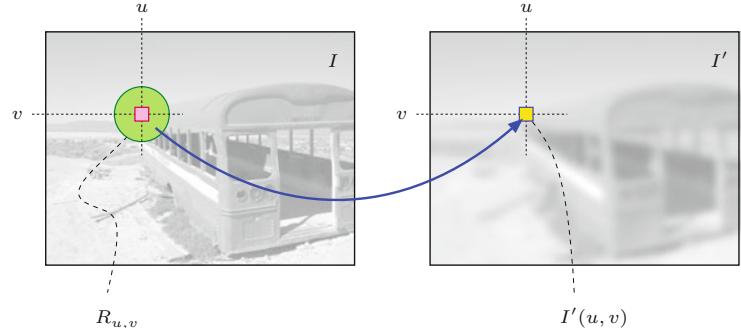
which we can write more compactly in the form

$$I'(u, v) \leftarrow \frac{1}{9} \cdot \sum_{j=-1}^1 \sum_{i=-1}^1 I(u+i, v+j). \quad (5.3)$$

This simple local averaging already exhibits all the important elements of a typical filter. In particular, it is a so-called *linear* filter, which is a very important class of filters. But how are filters defined in general? First they differ from point operations mainly by using not a single source pixel but a *set* of them for computing each resulting pixel. The coordinates of the source pixels are fixed relative to the current image position (u, v) and usually form a contiguous region, as illustrated in Fig. 5.2.

Fig. 5.2

Principal filter operation. Each new pixel value $I'(u, v)$ is calculated as a function of the pixel values within a specified region of source pixels $R_{u,v}$ in the original image I .



The *size* of the filter region is an important parameter of the filter because it specifies how many original pixels contribute to each resulting pixel value and thus determines the spatial extent (support) of the filter. For example, the smoothing filter in Eqn. (5.2) uses a 3×3 region of support that is centered at the current coordinate (u, v) . Similar filters with larger support, such as 5×5 , 7×7 , or even 21×21 pixels, would obviously have stronger smoothing effects.

The *shape* of the filter region is not necessarily quadratic or even rectangular. In fact, a circular (disk-shaped) region would be preferred to obtain an *isotropic* blur effect (i.e., one that is the same in all image directions). Another option is to assign different *weights* to the pixels in the support region, such as to give stronger emphasis to pixels that are closer to the center of the region. Furthermore, the support region of a filter does not need to be contiguous and may

not even contain the original pixel itself (imagine a ring-shaped filter region, for example). Theoretically the filter region could even be of infinite size.

It is probably confusing to have so many options—a more systematic method is needed for specifying and applying filters in a targeted manner. The traditional and proven classification into *linear* and *nonlinear* filters is based on the mathematical properties of the filter function; that is, whether the result is computed from the source pixels by a *linear* or a *nonlinear* expression. In the following, we discuss both classes of filters and show several practical examples.

5.2 LINEAR FILTERS

5.2 Linear Filters

Linear filters are denoted that way because they combine the pixel values in the support region in a linear fashion, that is, as a weighted summation. The local averaging process discussed in the beginning (Eqn. (5.3)) is a special example, where all nine pixels in the 3×3 support region are added with identical weights ($1/9$). With the same mechanism, a multitude of filters with different properties can be defined by simply modifying the distribution of the individual weights.

5.2.1 The Filter Kernel

For any linear filter, the size and shape of the support region, as well as the individual pixel weights, are specified by the “filter kernel” or “filter matrix” $H(i, j)$. The size of the kernel H equals the size of the filter region, and every element $H(i, j)$ specifies the weight of the corresponding pixel in the summation. For the 3×3 smoothing filter in Eqn. (5.3), the filter kernel is

$$H = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} = \frac{1}{9} \cdot \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \quad (5.4)$$

because each of the nine pixels contributes one-ninth of its value to the result.

In principle, the filter kernel $H(i, j)$ is, just like the image itself, a discrete, 2D, real-valued function, $H: \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{R}$. The filter has its own coordinate system with the origin—often referred to as the “hot spot”—mostly (but not necessarily) located at the center. Thus, filter coordinates are generally positive and negative (Fig. 5.3). The filter function is of infinite extent and considered zero outside the region defined by the matrix H .

5.2.2 Applying the Filter

For a linear filter, the result is unambiguously and completely specified by the coefficients of the filter matrix. Applying the filter to an image is a simple process that is illustrated in Fig. 5.4. The following steps are performed at each image position (u, v) :

5 FILTERS

Fig. 5.3

Filter matrix and its coordinate system. i is the horizontal (column) index, j is the vertical (row) index.

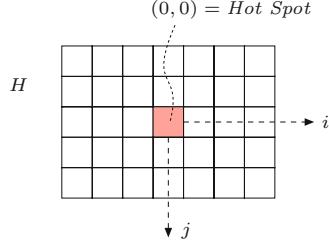
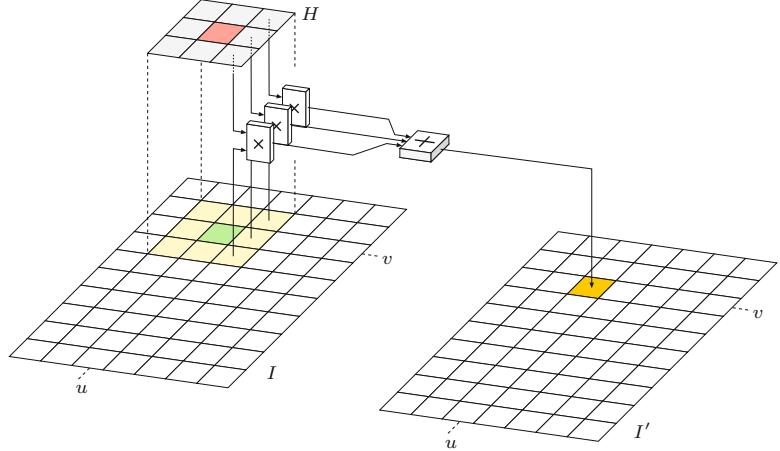


Fig. 5.4

Linear filter operation. The filter kernel H is placed with its origin at position (u, v) on the image I . Each filter coefficient $H(i, j)$ is multiplied with the corresponding image pixel $I(u + i, v + j)$, the results are added, and the final sum is inserted as the new pixel value $I'(u, v)$.



1. The filter kernel H is moved over the original image I such that its origin $H(0, 0)$ coincides with the current image position (u, v) .
2. All filter coefficients $H(i, j)$ are multiplied with the corresponding image element $I(u + i, v + j)$, and the results are added up.
3. Finally, the resulting sum is stored at the current position in the new image $I'(u, v)$.

Described formally, the pixel values of the new image $I'(u, v)$ are computed by the operation

$$I'(u, v) \leftarrow \sum_{(i,j) \in R_H} I(u + i, v + j) \cdot H(i, j), \quad (5.5)$$

where R_H denotes the set of coordinates covered by the filter H . For a typical 3×3 filter with centered origin, this is

$$I'(u, v) \leftarrow \sum_{i=-1}^{i=1} \sum_{j=-1}^{j=1} I(u + i, v + j) \cdot H(i, j), \quad (5.6)$$

for all image coordinates (u, v) . Not quite for *all* coordinates, to be exact. There is an obvious problem at the image borders where the filter reaches outside the image and finds no corresponding pixel values to use in computing a result. For the moment, we ignore this border problem, but we will attend to it again in Sec. 5.5.2.

5.2.3 Implementing the Filter Operation

5.2 LINEAR FILTERS

Now that we understand the principal operation of a filter (Fig. 5.4) and know that the borders need special attention, we go ahead and program a simple linear filter in ImageJ. But before we do this, we may want to consider one more detail. In a point operation (e.g., in Progs. 4.1 and 4.2), each new pixel value depends only on the corresponding pixel value in the original image, and it was thus no problem simply to store the results back to the same image—the computation is done “in place” without the need for any intermediate storage. In-place computation is generally not possible for a filter since any original pixel contributes to more than one resulting pixel and thus may not be modified before all operations are complete.

We therefore require additional storage space for the resulting image, which subsequently could be copied back to the source image again (if desired). Thus the complete filter operation can be implemented in two different ways (Fig. 5.5):

- The result of the filter computation is initially stored in a new image whose content is eventually copied back to the original image.
- The original image is first copied to an intermediate image that serves as the source for the actual filter operation. The result replaces the pixels in the original image.

The same amount of storage is required for both versions, and thus none of them offers a particular advantage. In the following examples, we generally use version B.

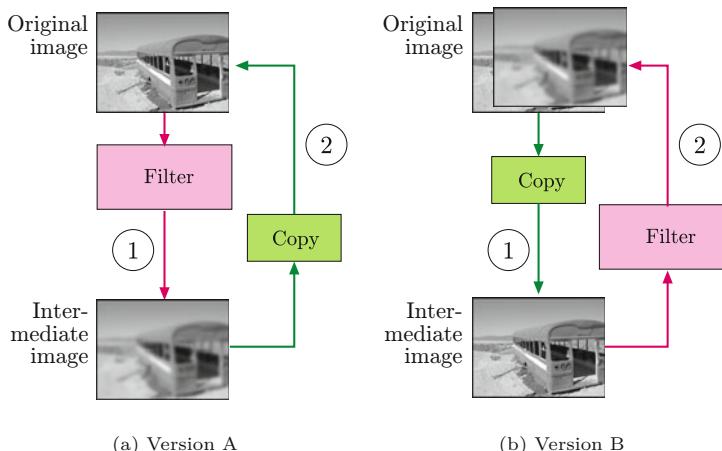


Fig. 5.5
Practical implementation of in-place filter operations.
Version A: The result of the filter operation is first stored in an intermediate image and subsequently copied back to the original image (a).
Version B: The original image is first copied to an intermediate image that serves as the source for the filter operation. The results are placed in the original image (b).

5.2.4 Filter Plugin Examples

The following examples demonstrate the implementation of two very basic filters that are nevertheless often used in practice.

Simple 3×3 averaging filter (“box” filter)

Program 5.1 shows the ImageJ code for a simple 3×3 smoothing filter based on local averaging (Eqn. (5.4)), which is often called a

5 FILTERS

Prog. 5.1

3×3 averaging “box” filter (`Filter_Box_3x3`). First (in line 10) a duplicate (`copy`) of the original image (`orig`) is created, which is used as the source image in the subsequent filter computation (line 18). In line 23, the resulting value is placed in the original image (line 23). Notice that the border pixels remain unchanged because they are not reached by the iteration over (u, v) .

```
1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4
5 public class Filter_Box_3x3 implements PlugInFilter {
6     ...
7     public void run(ImageProcessor ip) {
8         int M = ip.getWidth();
9         int N = ip.getHeight();
10        ImageProcessor copy = ip.duplicate();
11
12        for (int u = 1; u <= M - 2; u++) {
13            for (int v = 1; v <= N - 2; v++) {
14                //compute filter result for position (u, v):
15                int sum = 0;
16                for (int i = -1; i <= 1; i++) {
17                    for (int j = -1; j <= 1; j++) {
18                        int p = copy.getPixel(u + i, v + j);
19                        sum = sum + p;
20                    }
21                }
22                int q = (int) (sum / 9.0);
23                ip.putPixel(u, v, q);
24            }
25        }
26    }
27 }
```

“box” filter because of its box-like shape. No explicit filter matrix is required in this case, since all filter coefficients are identical ($1/9$). Also, no *clamping* (see Sec. 4.1.2) of the results is needed because the sum of the filter coefficients is 1 and thus no pixel values outside the admissible range can be created.

Although this example implements an extremely simple filter, it nevertheless demonstrates the general structure of a 2D filter program. In particular, *four* nested loops are needed: *two* (outer) loops for moving the filter over the image coordinates (u, v) and *two* (inner) loops to iterate over the (i, j) coordinates within the rectangular filter region. The required amount of computation thus depends not only upon the size of the image but equally on the size of the filter.

Another 3×3 smoothing filter

Instead of the constant weights applied in the previous example, we now use a real filter matrix with variable coefficients. For this purpose, we apply a bell-shaped 3×3 filter function $H(i, j)$, which puts more emphasis on the center pixel than the surrounding pixels:

$$H = \begin{bmatrix} 0.075 & 0.125 & 0.075 \\ 0.125 & \textcolor{red}{0.200} & 0.125 \\ 0.075 & 0.125 & 0.075 \end{bmatrix}. \quad (5.7)$$

Notice that all coefficients in H are positive and sum to 1 (i.e., the matrix is normalized) such that all results remain within the origi-

```

1   ...
2   public void run(ImageProcessor ip) {
3       int M = ip.getWidth();
4       int N = ip.getHeight();
5
6       //3x3 filter matrix:
7       double[][] H = {
8           {0.075, 0.125, 0.075},
9           {0.125, 0.200, 0.125},
10          {0.075, 0.125, 0.075}};
11
12      ImageProcessor copy = ip.duplicate();
13
14      for (int u = 1; u <= M - 2; u++) {
15          for (int v = 1; v <= N - 2; v++) {
16              // compute filter result for position (u,v):
17              double sum = 0;
18              for (int i = -1; i <= 1; i++) {
19                  for (int j = -1; j <= 1; j++) {
20                      int p = copy.getPixel(u + i, v + j);
21                      // get the corresponding filter coefficient:
22                      double c = H[j + 1][i + 1];
23                      sum = sum + c * p;
24                  }
25              }
26              int q = (int) Math.round(sum);
27              ip.putPixel(u, v, q);
28          }
29      }
30  }

```

5.2 LINEAR FILTERS

Prog. 5.2

3×3 smoothing filter (*Filter_Smooth_3x3*). The filter matrix is defined as a 2D array of type `double` (line 7). The coordinate origin of the filter is assumed to be at the center of the matrix (i.e., at the array position [1, 1]), which is accounted for by an offset of 1 for the i, j coordinates in line 22. The results are rounded (line 26) and stored in the original image (line 27).

nal range of pixel values. Again no clamping is necessary and the program structure in Prog. 5.2 is virtually identical to the previous example. The filter matrix (`filter`) is represented by a 2D array¹ of type `double`. Each pixel is multiplied by the corresponding coefficient of the filter matrix, the resulting sum being also of type `double`. Accessing the filter coefficients, it must be considered that the coordinate origin of the filter matrix is assumed to be at its center (i.e., at position (1, 1)) in the case of a 3×3 matrix. This explains the offset of 1 for the i and j coordinates (see Prog. 5.2, line 22).

5.2.5 Integer Coefficients

Instead of using floating-point coefficients (as in the previous examples), it is often simpler and usually more efficient to work with integer coefficients in combination with some common scale factor s , that is,

$$H(i, j) = s \cdot H'(i, j), \quad (5.8)$$

with $H'(i, j) \in \mathbb{Z}$ and $s \in \mathbb{R}$. If all filter coefficients are positive (which is the case for any smoothing filter), then s is usually taken

¹ See the additional comments regarding 2D arrays in Java in Sec. F.2.4 in the Appendix.

as the reciprocal of the sum of the coefficients,

$$s = \frac{1}{\sum_{i,j} H'(i,j)}, \quad (5.9)$$

to obtain a normalized filter matrix. In this case, the results are bounded to the original range of pixel values. For example, the filter matrix in Eqn. (5.7) could be defined equivalently as

$$H = \begin{bmatrix} 0.075 & 0.125 & 0.075 \\ 0.125 & \textcolor{red}{0.200} & 0.125 \\ 0.075 & 0.125 & 0.075 \end{bmatrix} = \frac{1}{40} \cdot \begin{bmatrix} 3 & 5 & 3 \\ 5 & \textcolor{red}{8} & 5 \\ 3 & 5 & 3 \end{bmatrix} \quad (5.10)$$

with the common scale factor $s = \frac{1}{40} = 0.025$. A similar scaling is used for the filter operation in Prog. 5.3.

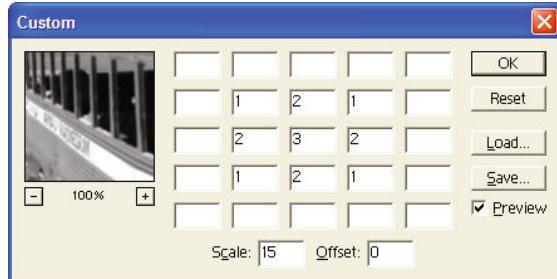
In Adobe Photoshop, linear filters can be specified with the “Custom Filter” tool (Fig. 5.6) using integer coefficients and a common scale factor *Scale* (which corresponds to the reciprocal of s). In addition, a constant *Offset* value can be specified; for example, to shift negative results (caused by negative coefficients) into the visible range of values. In summary, the operation performed by the 5×5 Photoshop custom filter can be expressed as

$$I'(u, v) \leftarrow \text{Offset} + \frac{1}{\text{Scale}} \cdot \sum_{j=-2}^{j=2} \sum_{i=-2}^{i=2} I(u+i, v+j) \cdot H(i, j). \quad (5.11)$$

Fig. 5.6

Adobe Photoshop’s “Custom Filter” implements linear filters up to a size of 5×5 .

The filter’s coordinate origin (“hot spot”) is assumed to be at the center (value set to 3 in this example), and empty cells correspond to zero coefficients. In addition to the (integer) coefficients, common *Scale* and *Offset* values can be specified (see Eqn. (5.11)).



5.2.6 Filters of Arbitrary Size

Small filters of size 3×3 are frequently used in practice, but sometimes much larger filters are required. Let us assume that the filter matrix H is centered and has an odd number of $(2K+1)$ columns and $(2L+1)$ rows, with $K, L \geq 0$. If the image is of size $M \times N$, that is

$$I(u, v) \quad \text{with} \quad 0 \leq u < M \quad \text{and} \quad 0 \leq v < N, \quad (5.12)$$

then the result of the filter can be calculated for all image coordinates (u', v') with

$$K \leq u' \leq (M-K-1) \quad \text{and} \quad L \leq v' \leq (N-L-1), \quad (5.13)$$

as illustrated in Fig. 5.7. Program 5.3 (which is adapted from Prog. 5.2) shows a 7×5 smoothing filter as an example for implementing

```

1  public void run(ImageProcessor ip) {
2      int M = ip.getWidth();
3      int N = ip.getHeight();
4
5      // filter matrix H of size  $(2K + 1) \times (2L + 1)$ 
6      int[][] H = {
7          {0,0,1,1,1,0,0},
8          {0,1,1,1,1,1,0},
9          {1,1,1,1,1,1,1},
10         {0,1,1,1,1,1,0},
11         {0,0,1,1,1,0,0}};
12
13     double s = 1.0 / 23; // sum of filter coefficients is 23
14
15     // H[L][K] is the center element of H:
16     int K = H[0].length / 2; // K = 3
17     int L = H.length / 2; // L = 2
18
19     ImageProcessor copy = ip.duplicate();
20
21     for (int u = K; u <= M - K - 1; u++) {
22         for (int v = L; v <= N - L - 1; v++) {
23             // compute filter result for position (u, v):
24             int sum = 0;
25             for (int i = -K; i <= K; i++) {
26                 for (int j = -L; j <= L; j++) {
27                     int p = copy.getPixel(u + i, v + j);
28                     int c = H[j + L][i + K];
29                     sum = sum + c * p;
30                 }
31             }
32             int q = (int) Math.round(s * sum);
33             // clamp result:
34             if (q < 0) q = 0;
35             if (q > 255) q = 255;
36             ip.putPixel(u, v, q);
37         }
38     }
39 }
```

5.2 LINEAR FILTERS

Prog. 5.3

Linear filter of arbitrary size using integer coefficients ([Filter_Arbitrary](#)). The filter matrix is an integer array of size $(2K + 1) \times (2L + 1)$ with the origin at the center element. The summation variable `sum` is also defined as an integer (`int`), which is scaled by a constant factor `s` and rounded in line 32. The border pixels are not modified.

linear filters of arbitrary size. This example uses integer-valued filter coefficients (line 6) in combination with a common scale factor s , as described already. As usual, the “hot spot” of the filter is assumed to be at the matrix center, and the range of all iterations depends on the dimensions of the filter matrix. In this case, clamping of the results is included (in lines 34–35) as a preventive measure.

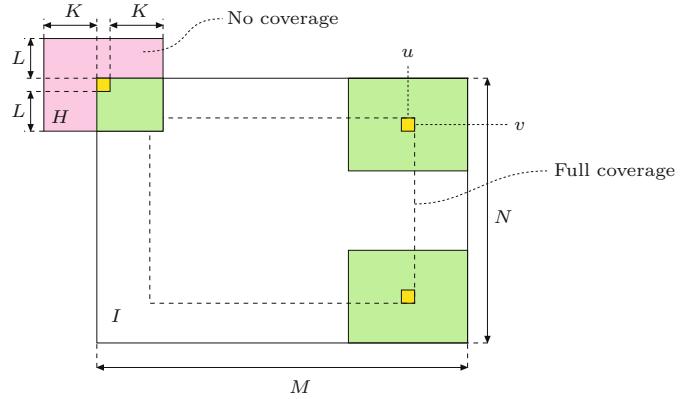
5.2.7 Types of Linear Filters

Since the effects of a linear filter are solely specified by the filter matrix (which can take on arbitrary values), an infinite number of different linear filters exists, at least in principle. So how can these filters be used and which filters are suited for a given task? In the following, we briefly discuss two broad classes of linear filters that are

5 FILTERS

Fig. 5.7

Border geometry. The filter can be applied only at locations where the kernel H of size $(2K+1) \times (2L+1)$ is fully contained in the image (inner rectangle).



of key importance in practice: smoothing filters and difference filters (Fig. 5.8).

Smoothing filters

Every filter we have discussed so far causes some kind of smoothing. In fact, any linear filter with positive-only coefficients is a smoothing filter in a sense, because such a filter computes merely a weighted average of the image pixels within a certain image region.

Box filter

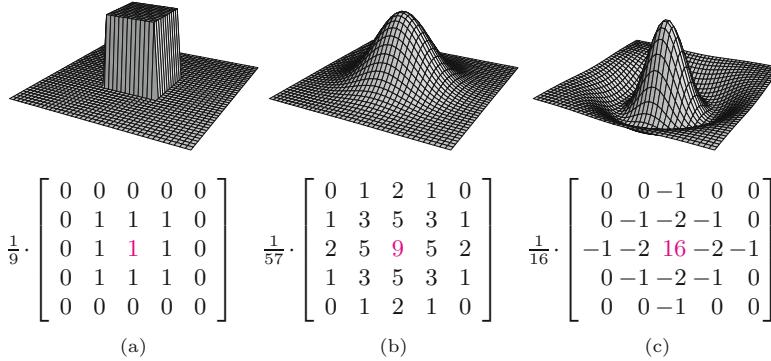
This simplest of all smoothing filters, whose 3D shape resembles a box (Fig. 5.8(a)), is a well-known friend already. Unfortunately, the box filter is far from an optimal smoothing filter due to its wild behavior in frequency space, which is caused by the sharp cutoff around its sides. Described in frequency terms, smoothing corresponds to low-pass filtering, that is, effectively attenuating all signal components above a given cutoff frequency (see also Chs. 18–19). The box filter, however, produces strong “ringing” in frequency space and is therefore not considered a high-quality smoothing filter. It may also appear rather ad hoc to assign the same weight to all image pixels in the filter region. Instead, one would probably expect to have stronger emphasis given to pixels near the center of the filter than to the more distant ones. Furthermore, smoothing filters should possibly operate “isotropically” (i.e., uniformly in each direction), which is certainly not the case for the rectangular box filter.

Gaussian filter

The filter matrix (Fig. 5.8(b)) of this smoothing filter corresponds to a 2D Gaussian function,

$$H^{G,\sigma}(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}}, \quad (5.14)$$

where σ denotes the width (standard deviation) of the bell-shaped function and r is the distance (radius) from the center. The pixel at the center receives the maximum weight (1.0, which is scaled to the integer value 9 in the matrix shown in Fig. 5.8(b)), and the remaining coefficients drop off smoothly with increasing distance from the



center. The Gaussian filter is isotropic if the discrete filter matrix is large enough for a sufficient approximation (at least 5×5). As a low-pass filter, the Gaussian is “well-behaved” in frequency space and thus clearly superior to the box filter. The 2D Gaussian filter is separable into a pair of 1D filters (see Sec. 5.3.3), which facilitates its efficient implementation.²

Difference filters

If some of the filter coefficients are negative, the filter calculation can be interpreted as the difference of two sums: the weighted sum of all pixels with associated positive coefficients minus the weighted sum of pixels with negative coefficients in the filter region R_H , that is,

$$I'(u, v) = \sum_{(i,j) \in R^+} I(u+i, v+j) \cdot |H(i, j)| - \sum_{(i,j) \in R^-} I(u+i, v+j) \cdot |H(i, j)|, \quad (5.15)$$

where R_H^+ and R_H^- denote the partitions of the filter with positive coefficients $H(i, j) > 0$ and negative coefficients $H(i, j) < 0$, respectively. For example, the 5×5 Laplace filter in Fig. 5.8(c) computes the difference between the center pixel (with weight 16) and the weighted sum of 12 surrounding pixels (with weights -1 or -2). The remaining 12 pixels have associated zero coefficients and are thus ignored in the computation.

While local intensity variations are *smoothed* by averaging, we can expect the exact contrary to happen when differences are taken: local intensity changes are *enhanced*. Important applications of difference filters thus include edge detection (Sec. 6.2) and image sharpening (Sec. 6.6).

5.3 Formal Properties of Linear Filters

In the previous sections, we have approached the concept of filters in a rather casual manner to quickly get a grasp of how filters are defined and used. While such a level of treatment may be sufficient for most practical purposes, the power of linear filters may not really

Fig. 5.8

Typical examples of linear filters, illustrated as 3D plots (top), profiles (center), and approximations by discrete filter matrices (bottom). The “box” filter (a) and the Gauss filter (b) are both *smoothing filters* with all-positive coefficients. The “Laplacian” or “Mexican hat” filter (c) is a *difference filter*. It computes the weighted difference between the center pixel and the surrounding pixels and thus reacts most strongly to local intensity peaks.

² See also Sec. E in the Appendix.

be apparent yet considering the limited range of (simple) applications seen so far.

The real importance of linear filters (and perhaps their formal elegance) only becomes visible when taking a closer look at some of the underlying theoretical details. At this point, it may be surprising to the experienced reader that we have not mentioned the term “convolution” in this context yet. We make up for this in the remaining parts of this section.

5.3.1 Linear Convolution

The operation associated with a linear filter, as described in the previous section, is not an invention of digital image processing but has been known in mathematics for a long time. It is called *linear convolution*³ and in general combines two functions of the same dimensionality, either continuous or discrete. For discrete, 2D functions I and H , the convolution operation is defined as

$$I'(u, v) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(u-i, v-j) \cdot H(i, j), \quad (5.16)$$

or, expressed with the designated *convolution operator* (*) in the form

$$I' = I * H. \quad (5.17)$$

This almost looks the same as Eqn. (5.5), with two differences: the range of the variables i, j in the summation and the negative signs in the coordinates of $I(u - i, v - j)$. The first point is easy to explain: because the coefficients outside the filter matrix $H(i, j)$, also referred to as a filter *kernel*, are assumed to be zero, the positions outside the matrix are irrelevant in the summation. To resolve the coordinate issue, we modify Eqn. (5.16) by replacing the summation variables i, j to

$$I'(u, v) = \sum_{(i,j) \in R_H} I(u-i, v-j) \cdot H(i, j) \quad (5.18)$$

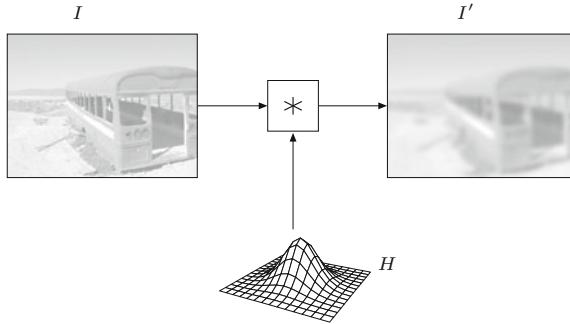
$$= \sum_{(i,j) \in R_H} I(u+i, v+j) \cdot H(-i, -j) \quad (5.19)$$

$$= \sum_{(i,j) \in R_H} I(u+i, v+j) \cdot H^*(i, j). \quad (5.20)$$

The result is identical to the linear filter in Eqn. (5.5), with the $H^*(i, j) = H(-i, -j)$ being the horizontally and vertically *reflected* (i.e., rotated by 180°) kernel H . To be precise, the operation in Eqn. (5.5) actually defines the linear *correlation*, which is merely a convolution with a reflected filter matrix.⁴

³ Oddly enough the simple concept of convolution is often (though unjustly) feared as an intractable mystery.

⁴ Of course this is the same in the 1D case. Linear correlation is typically used for comparing images or subpatterns (see Sec. 23.1 for details).



5.3 FORMAL PROPERTIES OF LINEAR FILTERS

Fig. 5.9
Convolution as a “black box” operation. The original image I is subjected to a linear convolution ($*$) with the convolution kernel H , producing the output image I' .

Thus the mathematical concept underlying all linear filters is the convolution operation ($*$) and its results are completely and sufficiently specified by the convolution matrix (or kernel) H . To illustrate this relationship, the convolution is often pictured as a “black box” operation, as shown in Fig. 5.9.

5.3.2 Formal Properties of Linear Convolution

The importance of linear convolution is based on its simple mathematical properties as well as its multitude of manifestations and applications. Linear convolution is a suitable model for many types of natural phenomena, including mechanical, acoustic, and optical systems. In particular (as shown in Ch. 18), there are strong formal links to the Fourier representation of signals in the frequency domain that are extremely valuable for understanding complex phenomena, such as sampling and aliasing. In the following, however, we first look at some important properties of linear convolution in the accustomed “signal” or image space.

Commutativity

Linear convolution is *commutative*; that is, for any image I and filter kernel H ,

$$I * H = H * I. \quad (5.21)$$

Thus the result is the same if the image and filter kernel are interchanged, and it makes no difference if we convolve the image I with the kernel H or the other way around. The two functions I and H are interchangeable and may assume either role.

Linearity

Linear filters are so called because of the linearity properties of the convolution operation, which manifests itself in various aspects. For example, if an image is multiplied by a scalar constant $s \in \mathbb{R}$, then the result of the convolution multiplies by the same factor, that is,

$$(s \cdot I) * H = I * (s \cdot H) = s \cdot (I * H). \quad (5.22)$$

Similarly, if we add two images I_1, I_2 pixel by pixel and convolve the resulting image with some kernel H , the same outcome is obtained

by convolving each image individually and adding the two results afterward, that is,

$$(I_1 + I_2) * H = (I_1 * H) + (I_2 * H). \quad (5.23)$$

It may be surprising, however, that simply *adding* a constant (scalar) value b to the image does *not* add to the convolved result by the same amount,

$$(b + I) * H \neq b + (I * H), \quad (5.24)$$

and is thus not part of the linearity property. While linearity is an important theoretical property, one should note that in practice “linear” filters are often only partially linear because of rounding errors or a limited range of output values.

Associativity

Linear convolution is associative, meaning that the order of successive filter operations is irrelevant, that is,

$$(I * H_1) * H_2 = I * (H_1 * H_2). \quad (5.25)$$

Thus multiple successive filters can be applied in any order, and multiple filters can be arbitrarily combined into new filters.

5.3.3 Separability of Linear Filters

A direct consequence of associativity is the separability of linear filters. If a convolution kernel H can be expressed as the convolution of multiple kernels H_i in the form

$$H = H_1 * H_2 * \dots * H_n, \quad (5.26)$$

then (as a consequence of Eqn. (5.25)) the filter operation $I * H$ may be performed as a sequence of convolutions with the constituting kernels H_i ,

$$\begin{aligned} I * H &= I * (H_1 * H_2 * \dots * H_n) \\ &= (\dots ((I * H_1) * H_2) * \dots * H_n). \end{aligned} \quad (5.27)$$

Depending upon the type of decomposition, this may result in significant computational savings.

x/y separability

The possibility of separating a 2D kernel H into a pair of 1D kernels h_x , h_y is of particular relevance and is used in many practical applications. Let us assume, as a simple example, that the filter is composed of the 1D kernels h_x and h_y , with

$$h_x = [1 \ 1 \ \textcolor{red}{1} \ 1 \ 1] \quad \text{and} \quad h_y = \begin{bmatrix} 1 \\ \textcolor{red}{1} \\ 1 \end{bmatrix}, \quad (5.28)$$

respectively. If these filters are applied sequentially to the image I ,

$$I' = (I * h_x) * h_y, \quad (5.29)$$

then (according to Eqn. (5.27)) this is equivalent to applying the composite filter

$$H = h_x * h_y = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & \textcolor{red}{1} & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}. \quad (5.30)$$

Thus the 2D 5×3 “box” filter H can be constructed from two 1D filters of lengths 5 and 3, respectively (which is obviously true for box filters of any size). But what is the advantage of this? In the aforementioned case, the required amount of processing is $5 \cdot 3 = 15$ steps per image pixel for the 2D filter H as compared with $5 + 3 = 8$ steps for the two separate 1D filters, a reduction of almost 50 %. In general, the number of operations for a 2D filter grows *quadratically* with the filter size (side length) but only *linearly* if the filter is x/y -separable. Clearly, separability is an eminent bonus for the implementation of large linear filters (see also Sec. 5.5.1).

Separable Gaussian filters

In general, a 2D filter is x/y -separable if (as in the earlier example) the filter function $H(i, j)$ can be expressed as the outer product (\otimes) of two 1D functions,

$$H(i, j) = h_x(i) \cdot h_y(j), \quad (5.31)$$

because in this case the resulting function also corresponds to the convolution product $H = H_x * H_y$. A prominent example is the widely employed 2D Gaussian function $G_\sigma(x, y)$ (Eqn. (5.14)), which can be expressed as the product

$$G_\sigma(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (5.32)$$

$$= \exp(-\frac{x^2}{2\sigma^2}) \cdot \exp(-\frac{y^2}{2\sigma^2}) = g_\sigma(x) \cdot g_\sigma(y). \quad (5.33)$$

Thus a 2D Gaussian filter H_σ^G can be implemented by a pair of 1D Gaussian filters $h_{x,\sigma}^G$ and $h_{y,\sigma}^G$ as

$$I * H_\sigma^G = I * h_{x,\sigma}^G * h_{y,\sigma}^G. \quad (5.34)$$

The ordering of the two 1D filters is not relevant in this case. With different σ -values along the x and y axes, elliptical 2D Gaussians can be realized as separable filters in the same fashion.

The Gaussian function decays relatively slowly with increasing distance from the center. To avoid visible truncation errors, discrete approximations of the Gaussian should have a sufficiently large extent of about $\pm 2.5\sigma$ to $\pm 3.5\sigma$ samples. For example, a discrete 2D Gaussian with “radius” $\sigma = 10$ requires a minimum filter size of 51×51 pixels, in which case the x/y -separable version can be expected to run about 50 times faster than the full 2D filter. The Java method `makeGaussKernel1d()` in Prog. 5.4 shows how to dynamically create a 1D Gaussian filter kernel with an extent of $\pm 3\sigma$ (i.e., a vector of odd length $6\sigma + 1$). As an example, this method is used for implementing “unsharp masking” filters where relatively large Gaussian kernels may be required (see Prog. 6.1 in Sec. 6.6.2).

5 FILTERS

Prog. 5.4

Dynamic creation of 1D Gaussian filter kernels. For a given σ , the Java method `makeGaussKernel1d()` returns a discrete 1D Gaussian filter kernel (float array) large enough to avoid truncation effects.

```

1  float[] makeGaussKernel1d(double sigma) {
2      // create the 1D kernel h:
3      int center = (int) (3.0 * sigma);
4      float[] h = new float[2 * center + 1]; // odd size
5      // fill the 1D kernel h:
6      double sigma2 = sigma * sigma;      //  $\sigma^2$ 
7      for (int i = 0; i < h.length; i++) {
8          double r = center - i;
9          h[i] = (float) Math.exp(-0.5 * (r * r) / sigma2);
10     }
11     return h;
12 }
```

5.3.4 Impulse Response of a Filter

Linear convolution is a binary operation involving two functions as its operands; it also has a “neutral element”, which of course is a function, too. The *impulse* or *Dirac* function $\delta()$ is neutral under convolution, that is,

$$I * \delta = I. \quad (5.35)$$

In the 2D, discrete case, the impulse function is defined as

$$\delta(u, v) = \begin{cases} 1 & \text{for } u = v = 0, \\ 0 & \text{otherwise.} \end{cases} \quad (5.36)$$

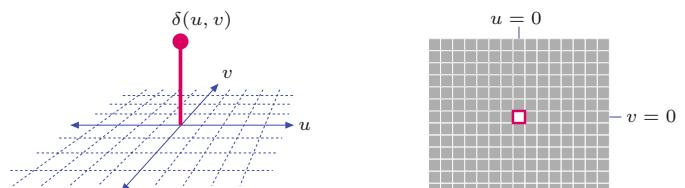
Interpreted as an image, this function is merely a single bright pixel (with value 1) at the coordinate origin contained in a dark (zero value) plane of infinite extent ([Fig. 5.10](#)).

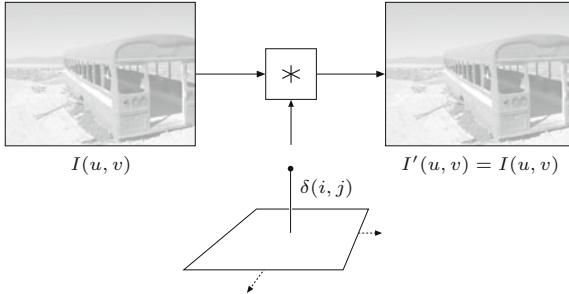
When the Dirac function is used as the filter kernel in a linear convolution as in Eqn. (5.35), the result is identical to the original image ([Fig. 5.11](#)). The reverse situation is more interesting, however, where some filter H is applied to the impulse δ as the input function. What happens? Since convolution is commutative (Eqn. (5.21)) it is evident that

$$H * \delta = \delta * H = H \quad (5.37)$$

and thus the result of this filter operation is identical to the filter H itself ([Fig. 5.12](#))! While sending an impulse into a linear filter to obtain its filter function may seem paradoxical at first, it makes sense if the properties (coefficients) of the filter H are unknown. Assuming that the filter is actually linear, complete information about this filter is obtained by injecting only a single impulse and measuring the result, which is called the “impulse response” of the filter. Among

Fig. 5.10
Discrete 2D *impulse* or
Dirac function $\delta(u, v)$.





5.4 NONLINEAR FILTERS

Fig. 5.11

Convolving the image I with the impulse δ returns the original (unmodified) image.

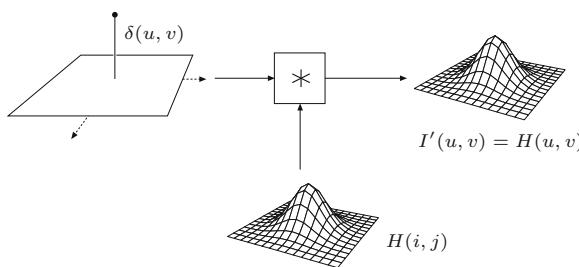


Fig. 5.12

The linear filter H with the impulse δ as the input yields the filter kernel H as the result.

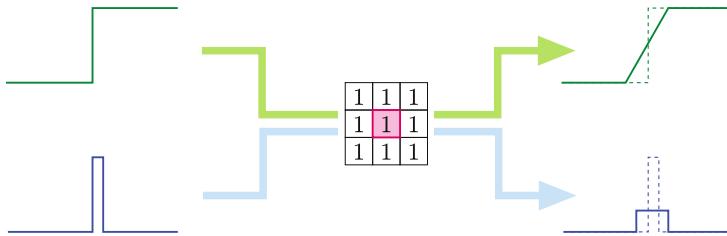


Fig. 5.13

Any image structure is blurred by a linear smoothing filter—important image structures such as step edges (top) or thin lines (bottom) are widened, and local contrast is reduced.

other applications, this technique is used for measuring the behavior of optical systems (e.g., lenses), where a point light source serves as the impulse and the result—a distribution of light energy—is called the “point spread function” (PSF) of the system.

5.4 Nonlinear Filters

5.4.1 Minimum and Maximum Filters

Like all other filters, nonlinear filters calculate the result at a given image position (u, v) from the pixels inside the moving region $R_{u,v}$ of the original image. The filters are called “nonlinear” because the source pixel values are combined by some nonlinear function. The simplest of all nonlinear filters are the *minimum* and *maximum* filters, defined as

$$I'(u, v) = \min_{(i,j) \in R} \{I(u + i, v + j)\}, \quad (5.38)$$

$$I'(u, v) = \max_{(i,j) \in R} \{I(u + i, v + j)\}, \quad (5.39)$$

5 FILTERS

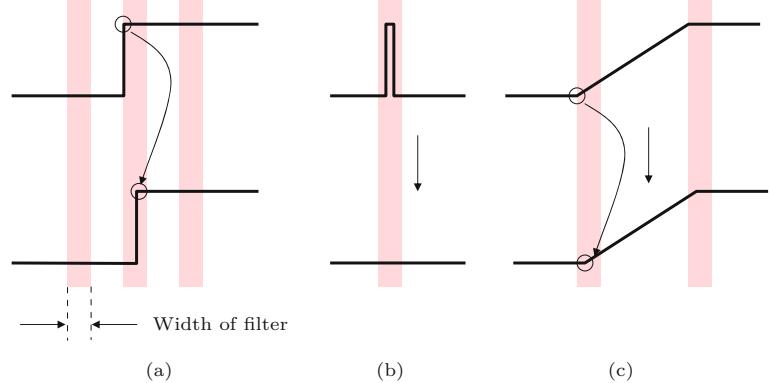
Fig. 5.14

3×3 linear box filter applied to a grayscale image corrupted with salt-and-pepper noise. Original (a), filtered image (b), enlarged details (c, d). Note that the individual noise pixels are only flattened but not removed.



Fig. 5.15

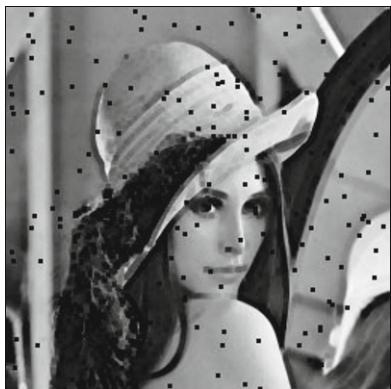
Effects of a 1D minimum filter on different local signal structures. Original signal (top) and result after filtering (bottom), where the colored bars indicate the extent of the filter. The step edge (a) and the linear ramp (c) are shifted to the right by half the filter width, and the narrow pulse (b) is completely removed.



where R denotes the filter region (set of filter coordinates, usually a square of size 3×3 pixels). **Figure 5.15** illustrates the effects of a 1D minimum filter on various local signal structures.

Figure 5.16 shows the results of applying 3×3 pixel minimum and maximum filters to a grayscale image corrupted with “salt-and-pepper” noise (i.e., randomly placed white and black dots), respectively. Obviously the minimum filter removes the white (salt) dots, because any single white pixel within the 3×3 filter region is replaced

Minimum filter



(a)

Maximum filter



(b)



(c)



(d)

5.4 NONLINEAR FILTERS

Fig. 5.16

Minimum and maximum filters applied to a grayscale image corrupted with “salt-and-pepper” noise (see original in Fig. 5.14(a)). The 3×3 *minimum* filter eliminates the bright dots and widens all dark image structures (a, c). The *maximum* filter shows the exact opposite effects (b, d).

by one of its surrounding pixels with a smaller value. Notice, however, that the minimum filter at the same time widens all the dark structures in the image.

The reverse effects can be expected from the *maximum* filter. Any single bright pixel is a local maximum as soon as it is contained in the filter region R . White dots (and all other bright image structures) are thus widened to the size of the filter, while now the dark (“pepper”) dots disappear.⁵

5.4.2 Median Filter

It is impossible of course to design a filter that removes any noise but keeps all the important image structures intact, because no filter can discriminate which image content is important to the viewer and which is not. The popular median filter is at least a good step in this direction.

⁵ The image shown in Figs. 5.14 and 5.16, called “Lena” (or “Lenna”), is one of the most popular test images in digital image processing ever and thus of historic interest. The picture of the Swedish “playmate” Lena Sjööblom (Söderberg?), published in *Playboy* in 1972, was included in a collection of test images at the University of Southern California and was subsequently used by researchers throughout the world (presumably without knowledge of its delicate origin) [115].

The median filter replaces every image pixel by the *median* of the pixels in the current filter region R , that is,

$$I'(u, v) = \underset{(i,j) \in R}{\text{median}}\{I(u + i, v + j)\}. \quad (5.40)$$

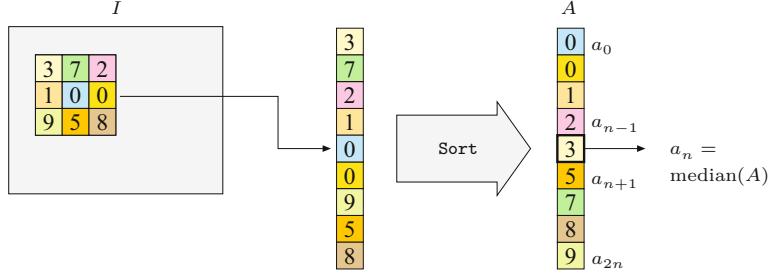
The median of a set of $2n + 1$ values $A = \{a_0, \dots, a_{2n}\}$ can be defined as the *center* value a_n after arranging (sorting) A to an ordered sequence, that is,

$$\text{median}(\underbrace{a_0, a_1, \dots, a_{n-1}}_{n \text{ values}}, \underbrace{a_n, a_{n+1}, \dots, a_{2n}}_{n \text{ values}}) = a_n, \quad (5.41)$$

where $a_i \leq a_{i+1}$. Figure 5.17 demonstrates the calculation of the median filter of size 3×3 (i.e., $n = 4$).

Fig. 5.17

Calculation of the median. The nine pixel values collected from the 3×3 image region are arranged as a vector that is subsequently sorted (A). The center value of A is taken as the median.



Equation (5.41) defines the median of an *odd*-sized set of values, and if the side length of the rectangular filters is odd (which is usually the case), then the number of elements in the filter region is odd as well. In this case, the median filter does not create any new pixel values that did not exist in the original image. If, however, the number of elements is *even*, then the median of the sorted sequence $A = (a_0, \dots, a_{2n-1})$ is defined as the arithmetic mean of the two adjacent center values a_{n-1} and a_n ,

$$\text{median}(\underbrace{a_0, \dots, a_{n-1}}_{n \text{ values}}, \underbrace{a_n, \dots, a_{2n-1}}_{n \text{ values}}) = \frac{a_{n-1} + a_n}{2}. \quad (5.42)$$

By averaging a_{n-1} and a_n , new pixel values are generally introduced by the median filter if the region is of even size.

Figure 5.18 compares the results of median filtering with a linear-smoothing filter. Finally, Fig. 5.19 illustrates the effects of a 3×3 pixel median filter on selected 2D image structures. In particular, very small structures (smaller than half the filter size) are eliminated, but all other structures remain largely unchanged. A sample Java implementation of the median filter of arbitrary size is shown in Prog. 5.5. The constant K specifies the side length of the filter region R of size $(2r + 1) \times (2r + 1)$. The number of elements in R (equal to the length of the vector A) is

$$(2r + 1)^2 = 4(r^2 + r) + 1, \quad (5.43)$$

and thus the index of the middle vector element is $n = 2(r^2 + r)$. Setting $r = 1$ gives a 3×3 median filter ($n = 4$), $r = 2$ gives a 5×5

Box filter (linear)



(a)

Median filter (nonlinear)



(b)



(c)

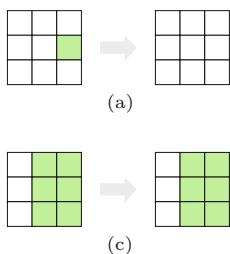


(d)

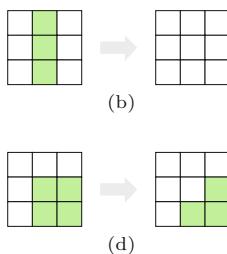
5.4 NONLINEAR FILTERS

Fig. 5.18

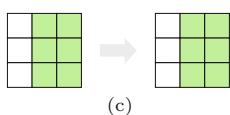
Linear smoothing filter vs. median filter applied to a grayscale image corrupted with salt-and-pepper noise (see original in Fig. 5.14(a)). The 3×3 linear box filter (a, c) reduces the bright and dark peaks to some extent but is unable to remove them completely. In addition, the entire image is blurred. The median filter (b, d) effectively eliminates the noise dots and also keeps the remaining structures largely intact. However, it also creates small spots of flat intensity that noticeably affect the sharpness.



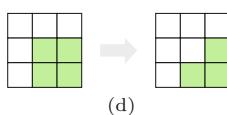
(a)



(b)



(c)



(d)

Fig. 5.19

Effects of a 3×3 pixel median filter on different 2D image structures. Isolated dots are eliminated (a), as are thin lines (b). The step edge remains unchanged (c), while a corner is rounded off (d).

filter ($n = 12$), etc. The structure of this plugin is similar to the arbitrary size linear filter in Prog. 5.3.

5.4.3 Weighted Median Filter

The median is a rank order statistic, and in a sense the “majority” of the pixel values involved determine the result. A single exceptionally high or low value (an “outlier”) cannot influence the result much but only shift the result up or down to the next value. Thus the median (in contrast to the linear average) is considered a “robust” measure. In an ordinary median filter, each pixel in the filter region has the same influence, regardless of its distance from the center.

5 FILTERS

Prog. 5.5

Median filter of arbitrary size (Plugin Filter_Median). An array A of type `int` is defined (line 16) to hold the region's pixel values for each filter position (u, v) . This array is sorted by using the Java utility method `Arrays.sort()` in line 32. The center element of the sorted vector ($A[n]$) is taken as the median value and stored in the original image (line 33).

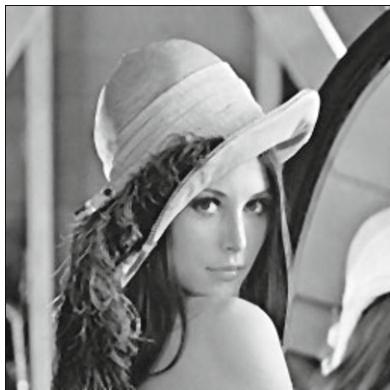
```
1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4 import java.util.Arrays;
5
6 public class Filter_Median implements PlugInFilter {
7
8     final int r = 4; // specifies the size of the filter
9
10    public void run(ImageProcessor ip) {
11        int M = ip.getWidth();
12        int N = ip.getHeight();
13        ImageProcessor copy = ip.duplicate();
14
15        // vector to hold pixels from  $(2r+1) \times (2r+1)$  neighborhood:
16        int[] A = new int[(2 * r + 1) * (2 * r + 1)];
17
18        // index of center vector element  $n = 2(r^2 + r)$ :
19        int n = 2 * (r * r + r);
20
21        for (int u = r; u <= M - r - 2; u++) {
22            for (int v = r; v <= N - r - 2; v++) {
23                // fill the pixel vector A for filter position (u,v):
24                int k = 0;
25                for (int i = -r; i <= r; i++) {
26                    for (int j = -r; j <= r; j++) {
27                        A[k] = copy.getPixel(u + i, v + j);
28                        k++;
29                    }
30                }
31                // sort vector A and take the center element A[n]:
32                Arrays.sort(A);
33                ip.putPixel(u, v, A[n]);
34            }
35        }
36    }
37 }
```

The weighted median filter assigns individual weights to the positions in the filter region, which can be interpreted as the “number of votes” for the corresponding pixel values. Similar to the coefficient matrix H of a linear filter, the distribution of weights is specified by a *weight matrix* W , with $W(i, j) \in \mathbb{N}$. To compute the result of the modified filter, each pixel value $I(u + i, v + j)$ involved is inserted $W(i, j)$ times into the extended pixel vector

$$A = (a_0, \dots, a_{L-1}) \quad \text{of length} \quad L = \sum_{(i,j) \in R} W(i, j). \quad (5.44)$$

This vector is again sorted, and the resulting center value is taken as the median, as in the standard median filter. [Figure 5.21](#) illustrates the computation of the weighted median filter using the 3×3 weight matrix

Median Filter



(a)

Weighted Median Filter



(b)



(c)



(d)

5.4 NONLINEAR FILTERS

Fig. 5.20

Ordinary vs. weighted median filter. Compared to the ordinary median filter (a, c), the weighted median (b, d) shows superior preservation of structural details. Both filters are of size 3×3 ; the weight matrix in Eqn. (5.45) was used for the weighted median filter.

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 2 & \textcolor{red}{3} & 2 \\ 1 & 2 & 1 \end{bmatrix}, \quad (5.45)$$

which requires an extended pixel vector of length $L = 15$, equal to the sum of the weights in W . If properly used, the weighted median filter yields effective noise removal with good preservation of structural details (see Fig. 5.20 for an example).

Of course this method may also be used to implement ordinary median filters of nonrectangular shape; for example, a *cross-shaped* median filter can be defined with the weight matrix

$$W^+ = \begin{bmatrix} 0 & 1 & 0 \\ 1 & \textcolor{red}{1} & 1 \\ 0 & 1 & 0 \end{bmatrix}. \quad (5.46)$$

Not every arrangement of weights is useful, however. In particular, if the weight assigned to the center pixel is greater than the sum of all other weights, then that pixel would always have the “majority vote” and dictate the resulting value, thus inhibiting any filter effect.

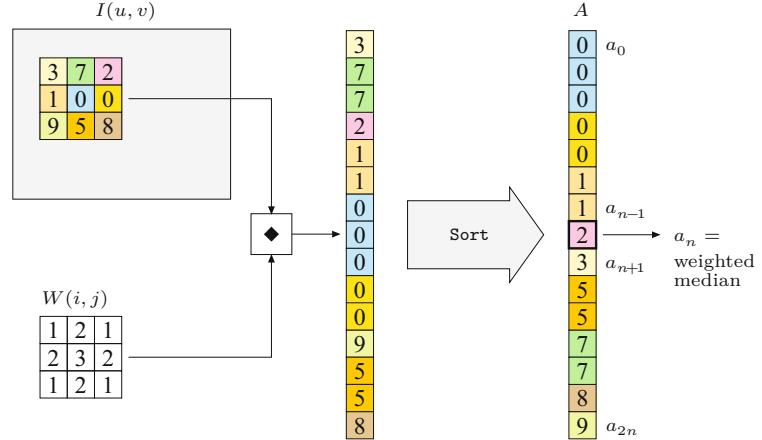
5.4.4 Other Nonlinear Filters

Median and weighted median filters are two examples of nonlinear filters that are easy to describe and frequently used. Since “nonlin-

5 FILTERS

Fig. 5.21

Weighted median example. Each pixel value is inserted into the extended pixel vector multiple times, as specified by the weight matrix W . For example, the value 0 from the center pixel is inserted three times (since $W(0, 0) = 3$) and the pixel value 7 twice. The pixel vector is sorted and the center value (2) is taken as the median.



“ear” refers to anything that is not linear, there are a multitude of filters that fall into this category, including the morphological filters for binary and grayscale images, which are discussed in Ch. 9. Other types of nonlinear filters, such as the corner detector described in Ch. 7, are often described algorithmically and thus defy a simple, compact description.

In contrast to the linear case, there is usually no “strong theory” for nonlinear filters that could, for example, describe the relationship between the sum of two images and the results of a median filter, as does Eqn. (5.23) for linear convolution. Similarly, not much (if anything) can be stated in general about the effects of nonlinear filters in frequency space.

5.5 Implementing Filters

5.5.1 Efficiency of Filter Programs

Computing the results of filters is computationally expensive in most cases, especially with large images, large filter kernels, or both. Given an image of size $M \times N$ and a filter kernel of size $(2K + 1) \times (2L + 1)$, a direct implementation requires

$$2K \cdot 2L \cdot M \cdot N = 4 KLMN \quad (5.47)$$

operations, namely multiplications and additions (in the case of a linear filter). Thus if both the image and the filter are simply assumed to be of size $N \times N$, the time complexity of direct filtering is $\mathcal{O}(N^4)$. As described in Sec. 5.3.3, substantial savings are possible when large, 2D filters can be decomposed (separated) into smaller, possibly 1D filters.

The programming examples in this chapter are deliberately designed to be simple and easy to understand, and none of the solutions shown is particularly efficient. Possibilities for tuning and code optimization exist in many places. It is particularly important to move all unnecessary instructions out of inner loops if possible because

these are executed most often. This applies especially to “expensive” instructions, such as method invocations, which may be relatively time-consuming.

In the examples, we have intentionally used the ImageJ standard methods `getPixel()` for reading and `putPixel()` for writing image pixels, which is the simplest and safest approach to accessing image data but also the slowest, of course. Substantial speed can be gained by using the quicker read and write methods `get()` and `set()` defined for class `ImageProcessor` and its subclasses. Note, however, that these methods do not check if the passed image coordinates are valid. Maximum performance can be obtained by accessing the pixel arrays directly.

5.5 IMPLEMENTING FILTERS

5.5.2 Handling Image Borders

As mentioned briefly in Sec. 5.2.2, the image borders require special attention in most filter implementations. We have argued that theoretically no filter results can be computed at positions where the filter matrix is not fully contained in the image array. Thus any filter operation would reduce the size of the resulting image, which is not acceptable in most applications. While no formally correct remedy exists, there are several more or less practical methods for handling the remaining border regions:

Method 1: Set the unprocessed pixels at the borders to some constant value (e.g., “black”). This is certainly the simplest method, but not acceptable in many situations because the image size is incrementally reduced by every filter operation.

Method 2: Set the unprocessed pixels to the original (unfiltered) image values. Usually the results are unacceptable, too, due to the noticeable difference between filtered and unprocessed image parts.

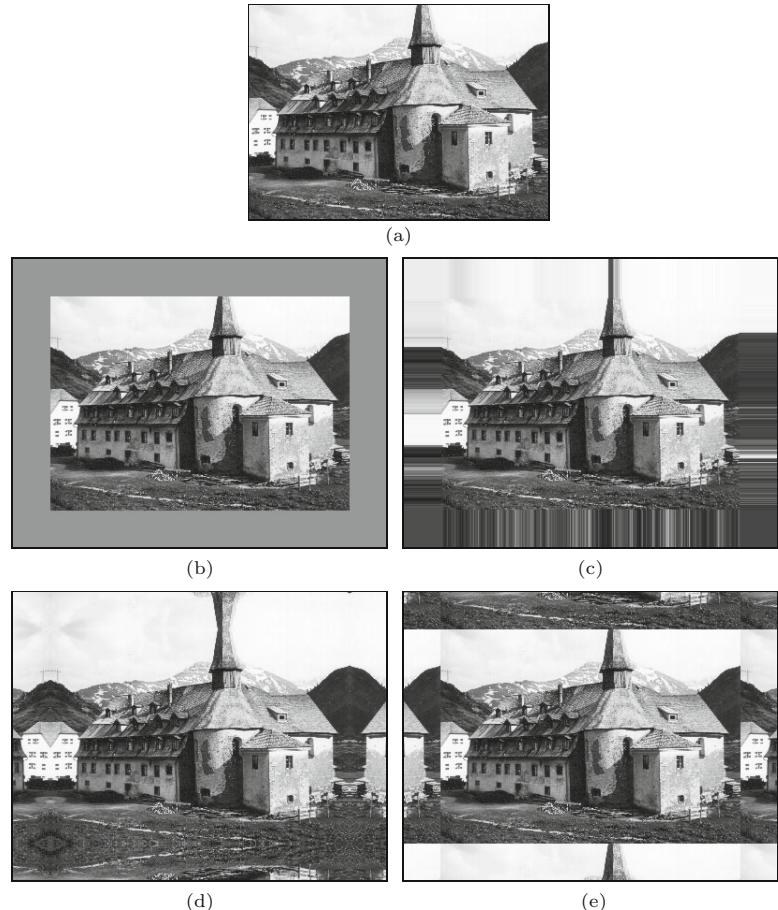
Method 3: Expand the image by “padding” additional pixels around it and apply the filter to the border regions as well. Fig. 5.22 shows different options for padding images.

- A. The pixels outside the image have a *constant value* (e.g., “black” or “gray”, see Fig. 5.22(a)). This may produce strong artifacts at the image borders, particularly when large filters are used.
- B. The *border pixels extend* beyond the image boundaries (Fig. 5.22(b)). Only minor artifacts can be expected at the borders. The method is also simple to compute and is thus often considered the method of choice.
- C. The *image is mirrored* at each of its four boundaries (Fig. 5.22(c)). The results will be similar to those of the previous method unless very large filters are used.
- D. The *image repeats periodically* in the horizontal and vertical directions (Fig. 5.22(d)). This may seem strange at first, and the results are generally not satisfactory. However, in discrete spectral analysis, the image is implicitly treated as a periodic function, too (see Ch. 18). Thus, if the image is filtered in the frequency domain, the results will be equal to filtering in the space domain under this repetitive model.

5 FILTERS

Fig. 5.22

Methods for padding the image to facilitate filtering along the borders. The assumption is that the (nonexisting) pixels outside the original image are either set to some constant value (a), take on the value of the closest border pixel (b), are mirrored at the image boundaries (c), or repeat periodically along the coordinate axes (d).



None of these methods is perfect and, as usual, the right choice depends upon the type of image and the filter applied. Notice also that the special treatment of the image borders may sometimes require more programming effort (and computing time) than the processing of the interior image.

5.5.3 Debugging Filter Programs

Experience shows that programming errors can hardly ever be avoided, even by experienced practitioners. Unless errors occur during execution (usually caused by trying to access nonexistent array elements), filter programs always “do something” to the image that may be similar but not identical to the expected result. To assure that the code operates correctly, it is not advisable to start with full, large images but first to experiment with small test cases for which the outcome can easily be predicted. Particularly when implementing linear filters, a first “litmus test” should always be to inspect the impulse response of the filter (as described in Sec. 5.3.4) before processing any real images.

5.6 Filter Operations in ImageJ

5.6 FILTER OPERATIONS IN IMAGEJ

ImageJ offers a collection of readily available filter operations, many of them contributed by other authors using different styles of implementation. Most of the available operations can also be invoked via ImageJ's **Process** menu.

5.6.1 Linear Filters

Filters based on linear convolution are implemented by the ImageJ plugin class `ij.plugin.filter.Convolver`, which offers useful “public” methods in addition to the standard `run()` method. Usage of this class is illustrated by the following example that convolves an 8-bit grayscale image with the filter kernel from Eqn. (5.7):

$$H = \begin{bmatrix} 0.075 & 0.125 & 0.075 \\ 0.125 & \textcolor{magenta}{0.200} & 0.125 \\ 0.075 & 0.125 & 0.075 \end{bmatrix}.$$

In the following `run()` method, we first define the filter matrix H as a 1D `float` array (notice the syntax for the `float` constants “`0.075f`”, etc.) and then create a new instance (`cv`) of class `Convolver` in line 8:

```
import ij.plugin.filter.Convolver;
...
public void run(ImageProcessor I) {
    float[] H = { // coefficient array H is one-dimensional!
        0.075f, 0.125f, 0.075f,
        0.125f, 0.200f, 0.125f,
        0.075f, 0.125f, 0.075f };
    Convolver cv = new Convolver();
    cv.setNormalize(true);      // turn on filter normalization
    cv.convolve(I, H, 3, 3);   // apply the filter H to I
}
```

The invocation of the method `convolve()` applies the filter H to the image I . It requires two additional arguments for the dimensions of the filter matrix since H is passed as a 1D array. The image I is destructively modified by the `convolve` operation.

In this case, one could have also used the nonnormalized, integer-valued filter matrix given in Eqn. (5.10) because `convolve()` normalizes the given filter automatically (after `cv.setNormalize(true)`).

5.6.2 Gaussian Filters

The ImageJ class `ij.plugin.filter.GaussianBlur` implements a simple Gaussian blur filter with arbitrary radius (σ). The filter uses separable 1D Gaussians as described in Sec. 5.3.3. Here is an example showing its application with $\sigma = 2.5$:

```
import ij.plugin.filter.GaussianBlur;
...
public void run(ImageProcessor I) {
    GaussianBlur gb = new GaussianBlur();
```

```

    double sigmaX = 2.5;
    double sigmaY = sigmaX;
    double accuracy = 0.01;
    gb.blurGaussian(I, sigmaX, sigmaY, accuracy);
    ...
}

```

The `accuracy` value specifies the size of the discrete filter kernel. Higher accuracy reduces truncation errors but requires larger kernels and more processing time.

An alternative implementation of separable Gaussian filters can be found in Prog. 6.1 (see p. 145), which uses the method `makeGaussKernel1d()` defined in Prog. 5.4 (page 104) for dynamically calculating the required 1D filter kernels.

5.6.3 Nonlinear Filters

A small set of nonlinear filters is implemented in the ImageJ class `ij.plugin.filter.RankFilters`, including the minimum, maximum, and standard median filters. The filter region is (approximately) circular with variable radius. Here is an example that applies three different filters with the same radius in sequence:

```

import ij.plugin.filter.RankFilters;
...
public void run(ImageProcessor I) {
    RankFilters rf = new RankFilters();
    double radius = 3.5;
    rf.rank(I, radius, RankFilters.MIN);           // minimum filter
    rf.rank(I, radius, RankFilters.MAX);           // maximum filter
    rf.rank(I, radius, RankFilters.MEDIAN);        // median filter
}

```

5.7 Exercises

Exercise 5.1. Explain why the “custom filter” in Adobe Photoshop (Fig. 5.6) is not strictly a linear filter.

Exercise 5.2. Determine the possible maximum and minimum results (pixel values) for the following linear filter, when applied to an 8-bit grayscale image (with pixel values in the range [0, 255]):

$$H = \begin{bmatrix} -1 & -2 & 0 \\ -2 & 0 & 2 \\ 0 & 2 & 1 \end{bmatrix}.$$

Assume that no clamping of the results occurs.

Exercise 5.3. Modify the ImageJ plugin shown in Prog. 5.3 such that the image borders are processed as well. Use one of the methods for extending the image outside its boundaries as described in Sec. 5.5.2.

Exercise 5.4. Show that a standard box filter is not isotropic (i.e., does not smooth the image identically in all directions).

Exercise 5.5. Explain why the clamping of results to a limited range of pixel values may violate the linearity property (Sec. 5.3.2) of linear filters.

Exercise 5.6. Verify the properties of the *impulse* function with respect to linear filters (see Eqn. (5.37)). Create a black image with a white pixel at its center and use this image as the 2D impulse. See if linear filters really deliver the filter matrix H as their impulse response.

Exercise 5.7. Describe the effects of the linear filters with the following kernels:

$$H_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \textcolor{red}{1} & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad H_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \textcolor{red}{2} & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad H_3 = \frac{1}{3} \cdot \begin{bmatrix} 0 & 0 & 1 \\ 0 & \textcolor{red}{1} & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

Exercise 5.8. Design a linear filter (kernel) that creates a horizontal blur over a length of 7 pixels, thus simulating the effect of camera movement during exposure.

Exercise 5.9. Compare the number of processing steps required for non-separable linear filters and x/y -separable filters sized 5×5 , 11×11 , 25×25 , and 51×51 pixels. Compute the speed gain resulting from separability in each case.

Exercise 5.10. Program your own ImageJ plugin that implements a Gaussian smoothing filter with variable filter width (radius σ). The plugin should dynamically create the required filter kernels with a size of at least 5σ in both directions. Make use of the fact that the Gaussian function is x/y -separable (see Sec. 5.3.3).

Exercise 5.11. The *Laplacian of Gaussian* (LoG) filter (see Fig. 5.8) is based on the sum of the second derivatives of the 2D Gaussian. It is defined as

$$L_\sigma(x, y) = -\left(\frac{x^2 + y^2 - 2 \cdot \sigma^2}{\sigma^4}\right) \cdot e^{-\frac{x^2+y^2}{2 \cdot \sigma^2}}. \quad (5.48)$$

Implement the LoG filter as an ImageJ plugin of variable width (σ), analogous to Exercise 5.10. Find out if the LoG function is x/y -separable.

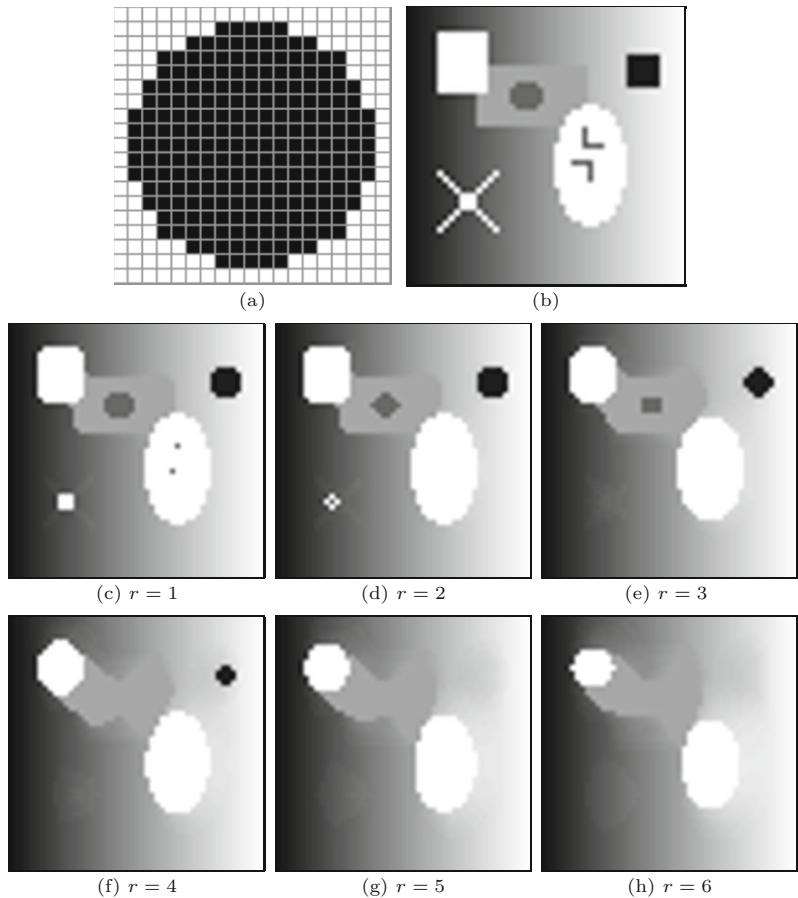
Exercise 5.12. Implement a circular (i.e., disk-shaped) median filter for grayscale images. Make the filter's radius r adjustable in the range from 1 to 10 pixels (e.g., using ImageJ's `GenericDialog` class). Use a binary (0/1) disk-shaped *mask* to represent the filter's support region R , with a minimum size of $(2r+1) \times (2r+1)$, as shown in Fig. 5.23(a). Create this mask dynamically for the chosen filter radius r (see Fig. 5.23(c-h) for typical results).

Exercise 5.13. Implement a weighted median filter (see Sec. 5.4.3) as an ImageJ plugin, specifying the weights as a constant, 2D `int` array. Test the filter on suitable images and compare the results with those from a standard median filter. Explain why, for example, the following weight matrix does *not* make sense:

Fig. 5.23

Disk-shaped median filter.
Example of a binary mask to represent the support region R with radius $r = 8$ (a).
The origin of the filter region is located at its center.

Synthetic test image (b).
Results of the median filter for $r = 1, \dots, 6$ (c-h).



$$W = \begin{bmatrix} 0 & 1 & 0 \\ 1 & \textcolor{blue}{5} & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

Exercise 5.14. The “jitter” filter is a (quite exotic) example for a *nonhomogeneous filter*. For each image position, it selects a space-variant filter kernel (of size $2r + 1$) containing a single, randomly placed impulse (1); for example,

$$H_{u,v} = \begin{bmatrix} 0 & 0 & 0 & \textcolor{blue}{1} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \textcolor{red}{0} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (5.49)$$

for $r = 2$. The position of the 1-value in the kernel $H_{u,v}$ is uniformly distributed in the range $i, j \in [-r, r]$; thus the filter effectively picks a random pixel value from the surrounding $(2r + 1) \times (2r + 1)$ neighborhood. Implement this filter for $r = 3, 5, 10$, as shown in Fig. 5.24. Is this filter linear or nonlinear? Develop another version using a Gaussian random distribution.



Original



$r = 3$



$r = 5$



$r = 10$

5.7 EXERCISES

Fig. 5.24
Jitter filter example.

Edges and Contours

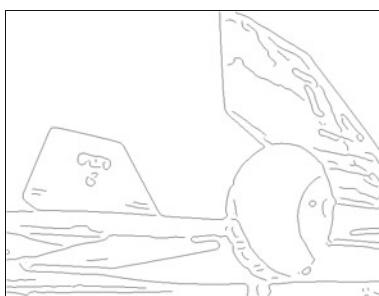
Prominent image “events” originating from local changes in intensity or color, such as edges and contours, are of high importance for the visual perception and interpretation of images. The perceived amount of information in an image appears to be directly related to the distinctiveness of the contained structures and discontinuities. In fact, edge-like structures and contours seem to be so important for our human visual system that a few lines in a caricature or illustration are often sufficient to unambiguously describe an object or a scene. It is thus no surprise that the enhancement and detection of edges has been a traditional and important topic in image processing as well. In this chapter, we first look at simple methods for localizing edges and then attend to the related issue of image sharpening.

6.1 What Makes an Edge?

Edges and contours play a dominant role in human vision and probably in many other biological vision systems as well. Not only are edges visually striking, but it is often possible to describe or reconstruct a complete figure from a few key lines, as the example in Fig. 6.1 shows. But how do edges arise, and how can they be technically localized in an image?



(a)



(b)

Fig. 6.1
Edges play an important role in human vision. Original image (a) and edge image (b).

Edges can roughly be described as image positions where the local intensity changes distinctly along a particular orientation. The stronger the local intensity change, the higher is the evidence for an edge at that position. In mathematics, the amount of change with respect to spatial distance is known as the first derivative of a function, and we thus start with this concept to develop our first simple edge detector.

6.2 Gradient-Based Edge Detection

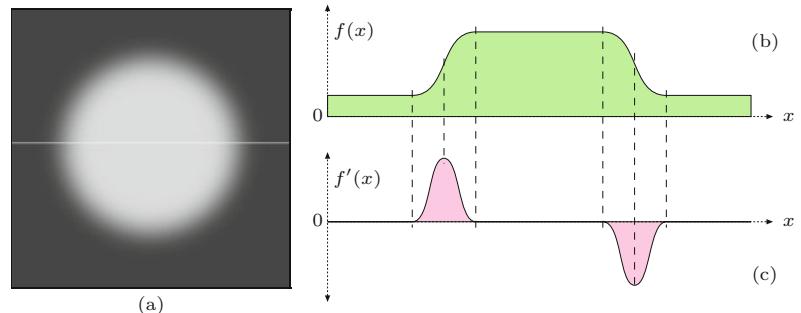
For simplicity, we first investigate the situation in only one dimension, assuming that the image contains a single bright region at the center surrounded by a dark background (Fig. 6.2(a)). In this case, the intensity profile along one image line would look like the 1D function $f(x)$, as shown in Fig. 6.2(b). Taking the first derivative of the function f ,

$$f'(x) = \frac{df}{dx}(x), \quad (6.1)$$

results in a positive swing at those positions where the intensity rises and a negative swing where the value of the function drops (Fig. 6.2(c)).

Fig. 6.2

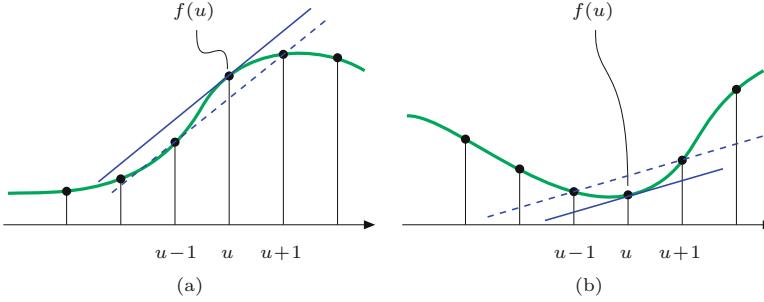
Sample image and first derivative in one dimension: original image (a), horizontal intensity profile $f(x)$ along the center image line (b), and first derivative $f'(x)$ (c).



Unlike in the continuous case, however, the first derivative is undefined for a *discrete* function $f(u)$ (such as the line profile of a real image), and some method is needed to estimate it. Figure 6.3 shows the basic idea, again for the 1D case: the first derivative of a continuous function at position x can be interpreted as the slope of its *tangent* at this position. One simple method for roughly approximating the slope of the tangent for a *discrete* function $f(u)$ at position u is to fit a straight line through the neighboring function values $f(u-1)$ and $f(u+1)$,

$$\frac{df}{dx}(u) \approx \frac{f(u+1) - f(u-1)}{(u+1) - (u-1)} = \frac{f(u+1) - f(u-1)}{2}. \quad (6.2)$$

Of course, the same method can be applied in the vertical direction to estimate the first derivative along the y -axis, that is, along the image columns.



6.2 GRADIENT-BASED EDGE DETECTION

Fig. 6.3

Estimating the first derivative of a discrete function. The slope of the straight (dashed) line between the neighboring function values $f(u-1)$ and $f(u+1)$ is taken as the estimate for the slope of the tangent (i.e., the first derivative) at $f(u)$.

6.2.1 Partial Derivatives and the Gradient

A derivative of a multi-dimensional function taken along one of its coordinate axes is called a *partial derivative*; for example,

$$I_x = \frac{\partial I}{\partial x}(u, v) \quad \text{and} \quad I_y = \frac{\partial I}{\partial y}(u, v) \quad (6.3)$$

are the partial derivatives of the 2D image function $I(u, v)$ along the u and v axes, respectively.¹ The vector

$$\nabla I(u, v) = \begin{pmatrix} I_x(u, v) \\ I_y(u, v) \end{pmatrix} = \begin{pmatrix} \frac{\partial I}{\partial x}(u, v) \\ \frac{\partial I}{\partial y}(u, v) \end{pmatrix} \quad (6.4)$$

is called the *gradient* of the function I at position (u, v) . The *magnitude* of the gradient,

$$|\nabla I| = \sqrt{I_x^2 + I_y^2}, \quad (6.5)$$

is invariant under image rotation and thus independent of the orientation of the underlying image structures. This property is important for isotropic localization of edges, and thus $|\nabla I|$ is the basis of many practical edge detection methods.

6.2.2 Derivative Filters

The components of the gradient function (Eqn. (6.4)) are simply the first derivatives of the image lines (Eqn. (6.1)) and columns along the horizontal and vertical axes, respectively. The approximation of the first horizontal derivatives (Eqn. (6.2)) can be easily implemented by a linear filter (see Sec. 5.2) with the 1D kernel

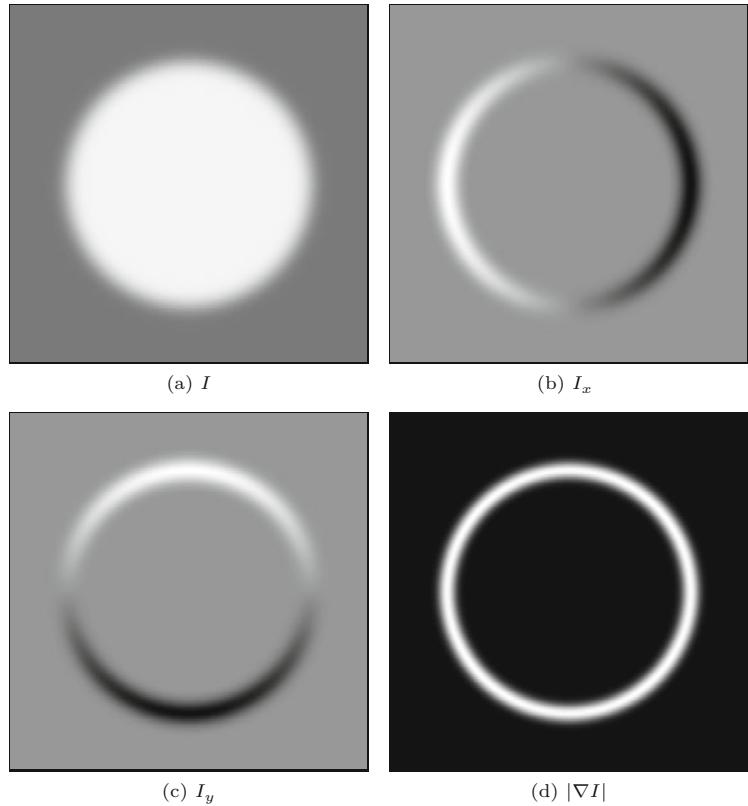
$$H_x^D = [-0.5 \ 0 \ 0.5] = 0.5 \cdot [-1 \ 0 \ 1], \quad (6.6)$$

where the coefficients -0.5 and $+0.5$ apply to the image elements $I(u-1, v)$ and $I(u+1, v)$, respectively. Notice that the center pixel $I(u, v)$ itself is weighted with the zero coefficient and is thus ignored. Analogously, the vertical component of the gradient is obtained with the linear filter

¹ ∂ denotes the *partial derivative* or “del” operator.

Fig. 6.4

Partial derivatives of a 2D function: synthetic image function I (a); approximate first derivatives in the horizontal direction $\partial I/\partial u$ (b) and the vertical direction $\partial I/\partial v$ (c); magnitude of the resulting gradient $|\nabla I|$ (d). In (b) and (c), the lowest (negative) values are shown black, the maximum (positive) values are white, and zero values are gray.



$$H_y^D = \begin{bmatrix} -0.5 \\ 0 \\ 0.5 \end{bmatrix} = 0.5 \cdot \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}. \quad (6.7)$$

Figure 6.4 shows the results of applying the gradient filters defined in Eqn. (6.6) and Eqn. (6.7) to a synthetic test image.

The orientation dependence of the filter responses can be seen clearly. The horizontal gradient filter H_x^D reacts most strongly to rapid changes along the horizontal direction, (i.e., to *vertical* edges); analogously the vertical gradient filter H_y^D reacts most strongly to *horizontal* edges. The filter response is zero in flat image regions (shown as gray in Fig. 6.4(b, c)).

6.3 Simple Edge Operators

The local gradient of the image function is the basis of many classical edge-detection operators. Practically, they only differ in the type of filter used for estimating the gradient components and the way these components are combined. In many situations, one is not only interested in the *strength* of edge points but also in the local *direction* of the edge. Both types of information are contained in the gradient function and can be easily computed from the directional components. The following small collection describes some frequently used, simple edge operators that have been around for many years and are thus interesting from a historic perspective as well.

6.3.1 Prewitt and Sobel Operators

6.3 SIMPLE EDGE OPERATORS

The edge operators by Prewitt [191] and Sobel [61] are two classic methods that differ only marginally in the derivative filters they use.

Gradient filters

Both operators use linear filters that extend over three adjacent lines and columns, respectively, to counteract the noise sensitivity of the simple (single line/column) gradient operators (Eqns. (6.6) and (6.7)). The Prewitt operator uses the filter kernels

$$H_x^P = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad H_y^P = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}, \quad (6.8)$$

which compute the average gradient components across three neighboring lines or columns, respectively. When the filters are written in separated form,

$$H_x^P = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad H_y^P = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}, \quad (6.9)$$

respectively, it becomes obvious that H_x^P performs a simple (box) smoothing over three lines before computing the x gradient (Eqn. (6.6)), and analogously H_y^P smooths over three columns before computing the y gradient (Eqn. (6.7)).² Because of the commutativity property of linear convolution, this could equally be described the other way around, with smoothing being applied *after* the computation of the gradients.

The filters for the Sobel operator are almost identical; however, the smoothing part assigns higher weight to the current center line and column, respectively:

$$H_x^S = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad H_y^S = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}. \quad (6.10)$$

The estimates for the local gradient components are obtained from the filter results by appropriate scaling, that is,

$$\nabla I(u, v) \approx \frac{1}{6} \cdot \begin{pmatrix} (I * H_x^P)(u, v) \\ (I * H_y^P)(u, v) \end{pmatrix} \quad (6.11)$$

for the *Prewitt* operator and

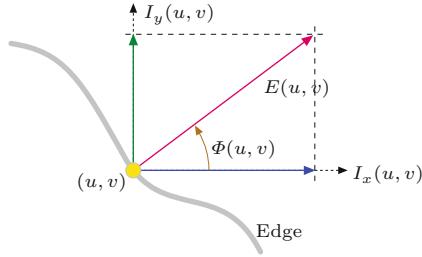
$$\nabla I(u, v) \approx \frac{1}{8} \cdot \begin{pmatrix} (I * H_x^S)(u, v) \\ (I * H_y^S)(u, v) \end{pmatrix} \quad (6.12)$$

for the *Sobel* operator.

² In Eqn. (6.9), $*$ is the linear convolution operator (see Sec. 5.3.1).

Fig. 6.5

Calculation of edge magnitude and orientation (geometry).



Edge strength and orientation

In the following, we denote the scaled filter results (obtained with either the Prewitt or Sobel operator) as

$$I_x = I * H_x \quad \text{and} \quad I_y = I * H_y.$$

In both cases, the local edge strength $E(u, v)$ is defined as the gradient magnitude

$$E(u, v) = \sqrt{I_x^2(u, v) + I_y^2(u, v)} \quad (6.13)$$

and the local edge orientation angle $\Phi(u, v)$ is³

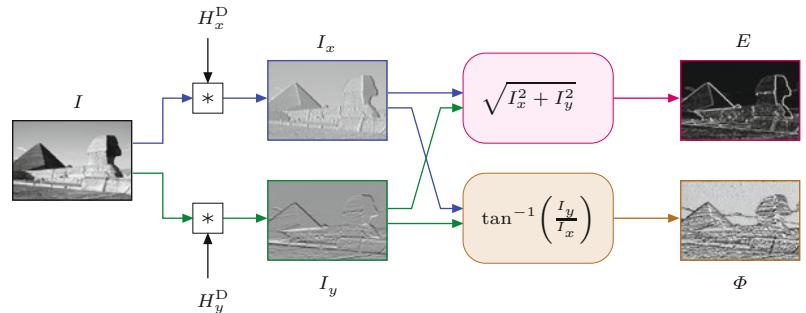
$$\Phi(u, v) = \tan^{-1}\left(\frac{I_y(u, v)}{I_x(u, v)}\right) = \text{ArcTan}(I_x(u, v), I_y(u, v)), \quad (6.14)$$

as illustrated in Fig. 6.5.

The whole process of extracting the edge magnitude and orientation is summarized in Fig. 6.6. First, the original image I is independently convolved with the two gradient filters H_x and H_y , and subsequently the edge strength E and orientation Φ are computed from the filter results. Figure 6.7 shows the edge strength and orientation for two test images, obtained with the Sobel filters in Eqn. (6.10).

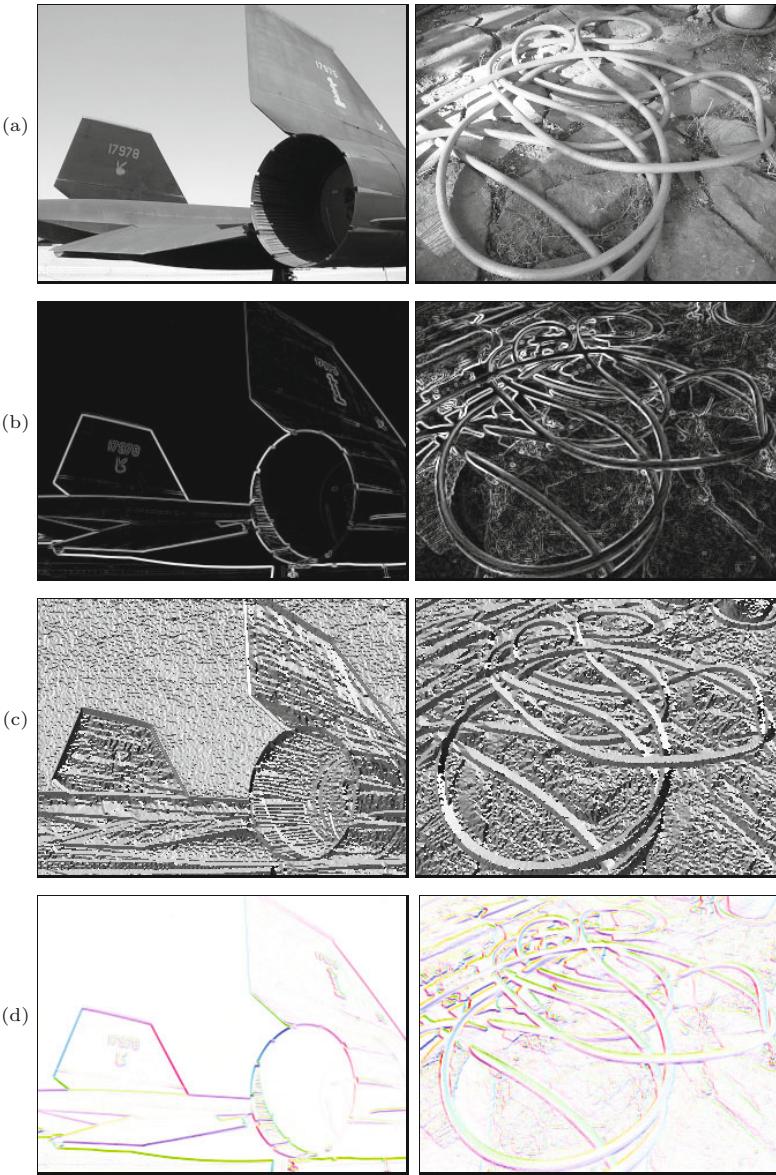
Fig. 6.6

Typical process of gradient-based edge extraction. The linear derivative filters H_x^D and H_y^D produce two gradient images, I_x and I_y , respectively. They are used to compute the edge strength E and orientation Φ for each image position (u, v) .



The estimate of the edge orientation based on the original Prewitt and Sobel filters is relatively inaccurate, and improved versions of the Sobel filters were proposed in [126, p. 353] to minimize the orientation errors:

³ See the hints in Sec. F.1.6 in the Appendix for computing the inverse tangent $\tan^{-1}(y/x)$ with the $\text{ArcTan}(x, y)$ function.



6.3 SIMPLE EDGE OPERATORS

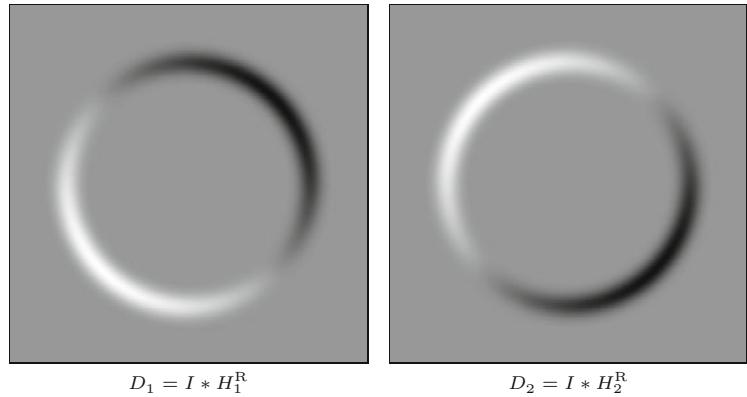
Fig. 6.7

Edge strength and orientation obtained with a Sobel operator. Original images (a), the edge strength $E(u, v)$ (b), and the local edge orientation $\Phi(u, v)$ (c). The images in (d) show the orientation angles coded as color hues, with the edge strength controlling the color saturation (see Sec. 12.2.3 for the corresponding definitions).

6.3.2 Roberts Operator

As one of the simplest and oldest edge finders, the Roberts operator [199] today is mainly of historic interest. It employs two extremely small filters of size 2×2 for estimating the directional gradient along

Fig. 6.8
Diagonal gradient components produced by the two Roberts filters.

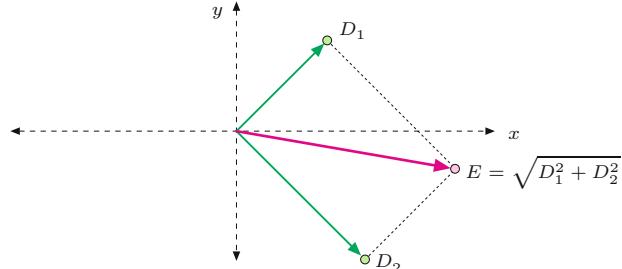


the image diagonals:

$$H_1^R = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \quad \text{and} \quad H_2^R = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}. \quad (6.16)$$

These filters naturally respond to diagonal edges but are not highly selective to orientation; that is, both filters show strong results over a relatively wide range of angles (Fig. 6.8). The local edge strength is calculated by measuring the length of the resulting 2D vector, similar to the gradient computation but with its components rotated 45° (Fig. 6.9).

Fig. 6.9
Definition of edge strength for the Roberts operator. The edge strength $E(u, v)$ corresponds to the length of the vector obtained by adding the two orthogonal gradient components (filter results) $D_1(u, v)$ and $D_2(u, v)$.



6.3.3 Compass Operators

The design of linear edge filters involves a trade-off: the stronger a filter responds to edge-like structures, the more sensitive it is to orientation. In other words, filters that are orientation insensitive tend to respond to nonedge structures, while the most discriminating edge filters only respond to edges in a narrow range of orientations. One solution is to use not only a single pair of relatively “wide” filters for two directions (such as the Prewitt and the simple Sobel operator discussed in Sec. 6.3.1) but a larger set of filters with narrowly spaced orientations.

Extended Sobel operator

Classic examples are the edge operator proposed by *Kirsch* [136] and the “extended Sobel” or *Robinson* operator [200], which employs the following eight filters with orientations spaced at 45°:

$$H_0^{\text{ES}} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad H_1^{\text{ES}} = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}, \quad (6.17)$$

$$H_2^{\text{ES}} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}, \quad H_3^{\text{ES}} = \begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix}, \quad (6.18)$$

$$H_4^{\text{ES}} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \quad H_5^{\text{ES}} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}, \quad (6.19)$$

$$H_6^{\text{ES}} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, \quad H_7^{\text{ES}} = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}. \quad (6.20)$$

Only the results of four of these eight filters ($H_0^{\text{ES}}, H_1^{\text{ES}}, \dots, H_7^{\text{ES}}$) must actually be computed since the remaining four are identical except for the reversed sign. For example, from the fact that $H_4^{\text{ES}} = -H_0^{\text{ES}}$ and the convolution being linear (Eqn. (5.22)), it follows that

$$I * H_4^{\text{ES}} = I * -H_0^{\text{ES}} = -(I * H_0^{\text{ES}}), \quad (6.21)$$

that is, the result for filter H_4^S is simply the negative result for filter H_0^S . The directional outputs D_0, D_1, \dots, D_7 for the eight Sobel filters can thus be computed as follows:

$$\begin{aligned} D_0 &\leftarrow I * H_0^{\text{ES}}, \quad D_1 \leftarrow I * H_1^{\text{ES}}, \quad D_2 \leftarrow I * H_2^{\text{ES}}, \quad D_3 \leftarrow I * H_3^{\text{ES}}, \\ D_4 &\leftarrow -D_0, \quad D_5 \leftarrow -D_1, \quad D_6 \leftarrow -D_2, \quad D_7 \leftarrow -D_3. \end{aligned} \quad (6.22)$$

The edge strength E^S at position (u, v) is defined as the maximum of the eight filter outputs; that is,

$$\begin{aligned} E^{\text{ES}}(u, v) &= \max(D_0(u, v), D_1(u, v), \dots, D_7(u, v)) \\ &= \max(|D_0(u, v)|, |D_1(u, v)|, |D_2(u, v)|, |D_3(u, v)|), \end{aligned} \quad (6.23)$$

and the strongest-responding filter also determines the local edge orientation as

$$\Phi^{\text{ES}}(u, v) = \frac{\pi}{4} j, \quad \text{with } j = \underset{0 \leq i \leq 7}{\operatorname{argmax}} D_i(u, v). \quad (6.24)$$

Kirsch operator

Another classic compass operator is the one proposed by Kirsch [136], which also employs eight oriented filters with the following kernels:

$$H_0^{\text{K}} = \begin{bmatrix} -5 & 3 & 3 \\ -5 & 0 & 3 \\ -5 & 3 & 3 \end{bmatrix}, \quad H_4^{\text{K}} = \begin{bmatrix} 3 & 3 & -5 \\ 3 & 0 & -5 \\ 3 & 3 & -5 \end{bmatrix}, \quad (6.25)$$

$$H_1^{\text{K}} = \begin{bmatrix} -5 & -5 & 3 \\ -5 & 0 & 3 \\ 3 & 3 & 3 \end{bmatrix}, \quad H_5^{\text{K}} = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 0 & -5 \\ 3 & -5 & -5 \end{bmatrix}, \quad (6.26)$$

$$H_2^{\text{K}} = \begin{bmatrix} -5 & -5 & -5 \\ 3 & 0 & 3 \\ 3 & 3 & 3 \end{bmatrix}, \quad H_6^{\text{K}} = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 0 & 3 \\ -5 & -5 & -5 \end{bmatrix}, \quad (6.27)$$

$$H_3^{\text{K}} = \begin{bmatrix} 3 & -5 & -5 \\ 3 & 0 & -5 \\ 3 & 3 & 3 \end{bmatrix}, \quad H_7^{\text{K}} = \begin{bmatrix} 3 & 3 & 3 \\ -5 & 0 & 3 \\ -5 & -5 & 3 \end{bmatrix}. \quad (6.28)$$

Again, because of the symmetries, only four of the eight filters need to be applied and the results may be combined in the same way as already described for the extended Sobel operator.

In practice, this and other “compass operators” show only minor benefits over the simpler operators described earlier, including the small advantage of not requiring the computation of square roots (which is considered a relatively “expensive” operation).

6.3.4 Edge Operators in ImageJ

The current version of ImageJ implements the Sobel operator (as described in Eqn. (6.10)) for practically any type of image. It can be invoked via the

Process ▷ Find Edges

menu and is also available through the method `void findEdges()` for objects of type `ImageProcessor`.

6.4 Other Edge Operators

One problem with edge operators based on first derivatives (as described in the previous section) is that each resulting edge is as wide as the underlying intensity transition and thus edges may be difficult to localize precisely. An alternative class of edge operators makes use of the second derivatives of the image function, including some popular modern edge operators that also address the problem of edges appearing at various levels of scale. These issues are briefly discussed in the following.

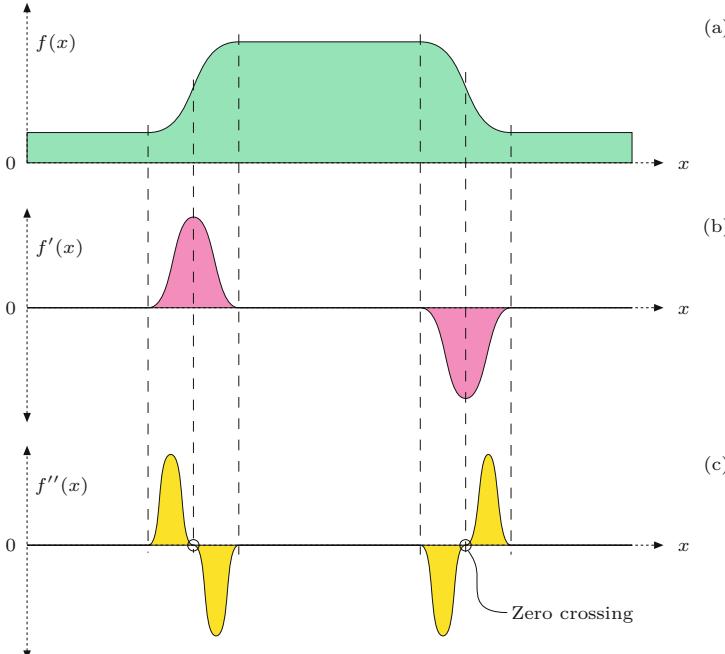
6.4.1 Edge Detection Based on Second Derivatives

The second derivative of a function measures its local curvature. The idea is that edges can be found at zero positions or—even better—at the zero crossings of the second derivatives of the image function, as illustrated in Fig. 6.10 for the 1D case. Since second derivatives generally tend to amplify image noise, some sort of presmoothing is usually applied with suitable low-pass filters.

A popular example is the “Laplacian-of-Gaussian” (LoG) operator [161], which combines gGaussian smoothing and computing the second derivatives (see the *Laplace Filter* in Sec. 6.6.1) into a single linear filter. The example in Fig. 6.11 shows that the edges produced by the LoG operator are more precisely localized than the ones delivered by the Prewitt and Sobel operators, and the amount of “clutter” is comparably small. Details about the LoG operator and a comprehensive survey of common edge operators can be found in [203, Ch. 4] and [165].

6.4.2 Edges at Different Scales

Unfortunately, the results of the simple edge operators we have discussed so far often deviate from what we as humans perceive as important edges. The two main reasons for this are:



6.4 OTHER EDGE OPERATORS

Fig. 6.10

Principle of edge detection with the second derivative: original function (a), first derivative (b), and second derivative (c). Edge points are located where the second derivative crosses through zero and the first derivative has a high magnitude.

- First, edge operators only respond to local intensity differences, while our visual system is able to extend edges across areas of minimal or vanishing contrast.
- Second, edges exist not at a single fixed resolution or at a certain scale but over a whole range of different scales.

Typical small edge operators, such as the Sobel operator, can only respond to intensity differences that occur within their 3×3 pixel filter regions. To recognize edge-like events over a greater horizon, we would either need larger edge operators (with correspondingly large filters) or to use the original (small) operators on reduced (i.e., scaled) images. This is the principal idea of “multiresolution” techniques (also referred to as “hierarchical” or “pyramid” techniques), which have traditionally been used in many image-processing applications [41, 151]. In the context of edge detection, this typically amounts to detecting edges at various scale levels first and then deciding which edge (if any) at which scale level is dominant at each image position.

6.4.3 From Edges to Contours

Whatever method is used for edge detection, the result is usually a continuous value for the edge strength for each image position and possibly also the angle of local edge orientation. How can this information be used, for example, to find larger image structures and contours of objects in particular?

Binary edge maps

In many situations, the next step after edge enhancement (by some edge operator) is the selection of edge points, a binary decision about

whether an image pixel is an edge point or not. The simplest method is to apply a *threshold* operation to the edge strength delivered by the edge operator using either a fixed or adaptive threshold value, which results in a binary edge image or “edge map”.

In practice, edge maps hardly ever contain perfect contours but instead many small, unconnected contour fragments, interrupted at positions of insufficient edge strength. After thresholding, the empty positions of course contain no edge information at all that could possibly be used in a subsequent step, such as for linking adjacent edge segments. Despite this weakness, global thresholding is often used at this point because of its simplicity, and some common postprocessing methods, such as the Hough transform (see Ch. 8), can cope well with incomplete edge maps.

Contour following

The idea of tracing contours sequentially along the discovered edge points is not uncommon and appears quite simple in principle. Starting from an image point with high edge strength, the edge is followed iteratively in both directions until the two traces meet and a closed contour is formed. Unfortunately, there are several obstacles that make this task more difficult than it seems at first, including the following:

- edges may end in regions of vanishing intensity gradient,
- crossing edges lead to ambiguities, and
- contours may branch into several directions.

Because of these problems, contour following usually is not applied to original images or continuous-valued edge images except in very simple situations, such as when there is a clear separation between objects (foreground) and the background. Tracing contours in segmented binary images is much simpler, of course (see Ch. 10).

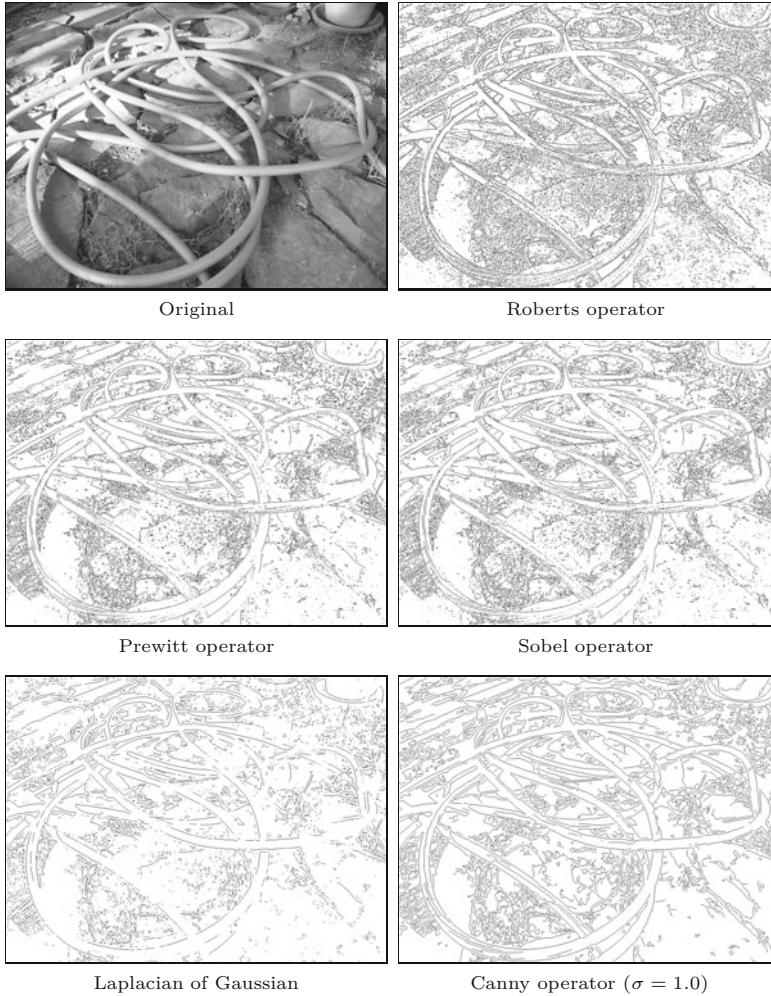
6.5 Canny Edge Operator

The operator proposed by Canny [42] is widely used and still considered “state of the art” in edge detection. The method tries to reach three main goals: (a) to minimize the number of false edge points, (b) achieve good localization of edges, and (c) deliver only a single mark on each edge. These properties are usually not achieved with simple edge operators (mostly based on first derivatives and subsequent thresholding).

At its core, the Canny “filter” is a gradient method (based on first derivatives; see Sec. 6.2), but it uses the zero crossings of second derivatives for precise edge localization.⁴ In this regard, the method is similar to edge detectors that are based on the second derivatives of the image function [161].

Fully implemented, the Canny detector uses a set of relatively large, oriented filters at multiple image resolutions and merges the

⁴ The zero crossings of a function’s second derivative are found where the first derivatives exhibit a local maximum or minimum.



6.5 CANNY EDGE OPERATOR

Fig. 6.11

Comparison of various edge operators. Important criteria for the quality of edge results are the amount of “clutter” (irrelevant edge elements) and the connectedness of dominant edges. The Roberts operator responds to very small edge structures because of the small size of its filters. The similarity of the Prewitt and Sobel operators is manifested in the corresponding results. The edge map produced by the Canny operator is substantially cleaner than those of the simpler operators, even for a fixed and relatively small scale value σ .

individual results into a common *edge map*. It is quite common, however, to use only a single-scale implementation of the algorithm with an adjustable filter radius (smoothing parameter σ), which is nevertheless superior to most of the simple edge operators (see Fig. 6.11). In addition, the algorithm not only yields a binary edge map but connected chains of edge pixels, which greatly simplifies the subsequent processing steps. Thus, even in its basic (single-scale) form, the Canny operator is often preferred over other edge detection methods.

In its basic (single-scale) form, the Canny operator performs the following steps (stated more precisely in Algs. 6.1–6.2):

1. **Pre-processing:** Smooth the image with a Gaussian filter of width σ , which specifies the scale level of the edge detector. Calculate the x/y gradient vector at each position of the filtered image and determine the local gradient magnitude and orientation.
2. **Edge localization:** Isolate local maxima of gradient magnitude by “non-maximum suppression” along the local gradient direction.

- 3. Edge tracing and hysteresis thresholding:** Collect sets of connected edge pixels from the local maxima by applying “hysteresis thresholding”.

6.5.1 Pre-processing

The original intensity image I is first smoothed with a Gaussian filter kernel $H^{G,\sigma}$; its width σ specifies the spatial scale at which edges are to be detected (see Alg. 6.1, lines 2–10). Subsequently, first-order difference filters are applied to the smoothed image \bar{I} to calculate the components \bar{I}_x, \bar{I}_y of the local gradient vectors (Alg. 6.1, line 3–3).⁵ Then the local magnitude E_{mag} is calculated as the norm of the corresponding gradient vector (Alg. 6.1, line 11). In view of the subsequent thresholding it may be helpful to normalize the edge magnitude values to a standard range (e.g., to $[0, 100]$).

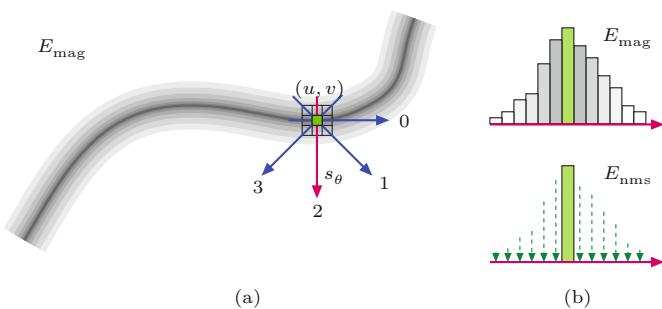
6.5.2 Edge localization

Candidate edge pixels are isolated by local “non-maximum suppression” of the edge magnitude E_{mag} . In this step, only those pixels are preserved that represent a local maximum along the 1D profile in the direction of the gradient, that is, perpendicular to the edge tangent (see Fig. 6.12). While the gradient may point in any continuous direction, only *four* discrete directions are typically used to facilitate efficient processing. The pixel at position (u, v) is only retained as an edge candidate if its gradient magnitude is greater than both its immediate neighbors in the direction specified by the gradient vector (d_x, d_y) at position (u, v) . If a pixel is not a local maximum, its edge magnitude value is set to zero (i.e., “suppressed”). In Alg. 6.1, the non-maximum suppressed edge values are stored in the map E_{nms} .

Fig. 6.12

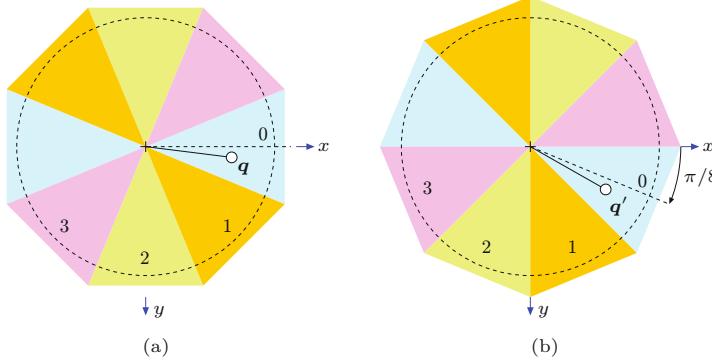
Non-maximum suppression of gradient magnitude. The gradient direction at position (u, v) is coarsely quantized to four discrete orientations $s_\theta \in \{0, 1, 2, 3\}$ (a).

Only pixels where the gradient magnitude $E_{\text{mag}}(u, v)$ is a local maximum in the gradient direction (i.e., perpendicular to the edge tangent) are taken as candidate edge points (b). The gradient magnitude at all other points is set (suppressed) to zero.



The problem of finding the discrete orientation $s_\theta = 0, \dots, 3$ for a given gradient vector $\mathbf{q} = (d_x, d_y)$ is illustrated in Fig. 6.13. This task is simple if the corresponding angle $\theta = \tan^{-1}(d_y/d_x)$ is known, but at this point the use of the trigonometric functions is typically avoided for efficiency reasons. The octant that corresponds to \mathbf{q} can be inferred directly from the signs and magnitude of the components d_x, d_y , however, the necessary decision rules are quite complex. Much simpler rules apply if the coordinate system and gradient vector \mathbf{q} are

⁵ See also Sec. C.3.1 in the Appendix.



6.5 CANNY EDGE OPERATOR

Fig. 6.13

Discrete gradient directions. In (a), calculating the octant for a given orientation vector $\mathbf{q} = (d_x, d_y)$ requires a relatively complex decision. Alternatively (b), if \mathbf{q} is rotated by $\frac{\pi}{8}$ to \mathbf{q}' , the corresponding octant can be found directly from the components of $\mathbf{q}' = (d'_x, d'_y)$ without the need to calculate the actual angle. Orientation vectors in the other octants are mirrored to octants $s_\theta = 0, 1, 2, 3$.

1: **CannyEdgeDetector**($I, \sigma, t_{hi}, t_{lo}$)

Input: I , a grayscale image of size $M \times N$; σ , scale (radius of Gaussian filter $H^{G,\sigma}$); t_{hi} , t_{lo} , hysteresis thresholds ($t_{hi} > t_{lo}$). Returns a binary edge map of size $M \times N$.

```

2:    $\bar{I} \leftarrow I * H^{G,\sigma}$                                  $\triangleright$  blur with Gaussian of width  $\sigma$ 
3:    $\bar{I}_x \leftarrow \bar{I} * [-0.5 \ 0 \ 0.5]$                    $\triangleright$   $x$ -gradient
4:    $\bar{I}_y \leftarrow \bar{I} * [-0.5 \ 0 \ 0.5]^\top$              $\triangleright$   $y$ -gradient
5:    $(M, N) \leftarrow \text{Size}(I)$ 
6:   Create maps:
7:      $E_{\text{mag}} : M \times N \mapsto \mathbb{R}$                  $\triangleright$  gradient magnitude
8:      $E_{\text{nms}} : M \times N \mapsto \mathbb{R}$                  $\triangleright$  maximum magnitude
9:      $E_{\text{bin}} : M \times N \mapsto \{0, 1\}$                $\triangleright$  binary edge pixels
10:  for all image coordinates  $(u, v) \in M \times N$  do
11:     $E_{\text{mag}}(u, v) \leftarrow [\bar{I}_x^2(u, v) + \bar{I}_y^2(u, v)]^{1/2}$ 
12:     $E_{\text{nms}}(u, v) \leftarrow 0$ 
13:     $E_{\text{bin}}(u, v) \leftarrow 0$ 
14:    for  $u \leftarrow 1, \dots, M-2$  do
15:      for  $v \leftarrow 1, \dots, N-2$  do
16:         $d_x \leftarrow \bar{I}_x(u, v)$ ,  $d_y \leftarrow \bar{I}_y(u, v)$ 
17:         $s_\theta \leftarrow \text{GetOrientationSector}(d_x, d_y)$            $\triangleright$  Alg. 6.2
18:        if  $\text{IsLocalMax}(E_{\text{mag}}, u, v, s_\theta, t_{lo})$  then       $\triangleright$  Alg. 6.2
19:           $E_{\text{nms}}(u, v) \leftarrow E_{\text{mag}}(u, v)$             $\triangleright$  only keep local maxima
20:    for  $u \leftarrow 1, \dots, M-2$  do
21:      for  $v \leftarrow 1, \dots, N-2$  do
22:        if  $(E_{\text{nms}}(u, v) \geq t_{hi}) \wedge (E_{\text{bin}}(u, v) = 0)$  then
23:           $\text{TraceAndThreshold}(E_{\text{nms}}, E_{\text{bin}}, u, v, t_{lo})$            $\triangleright$  Alg. 6.2
24:    return  $E_{\text{bin}}$ .

```

Alg. 6.1

Canny edge detector for grayscale images.

rotated by $\frac{\pi}{8}$, as illustrated in Fig. 6.13(b). This step is implemented by the function `GetOrientationSector()` in Alg. 6.2.⁶

6.5.3 Edge tracing and hysteresis thresholding

In the final step, sets of connected edge points are collected from the magnitude values that remained unsuppressed in the previous oper-

⁶ Note that the elements of the rotation matrix in Alg. 6.2 (line 2) are constants and thus no repeated use of trigonometric functions is required.

6 EDGES AND CONTOURS

Alg. 6.2

Procedures used in Alg. 6.1 (Canny edge detector).

```

1: GetOrientationSector( $d_x, d_y$ )
   Returns the discrete octant  $s_\theta$  for the orientation vector  $(d_x, d_y)^\top$ .
   See Fig. 6.13 for an illustration.

2:  $\begin{pmatrix} d'_x \\ d'_y \end{pmatrix} \leftarrow \begin{pmatrix} \cos(\pi/8) & -\sin(\pi/8) \\ \sin(\pi/8) & \cos(\pi/8) \end{pmatrix} \cdot \begin{pmatrix} d_x \\ d_y \end{pmatrix}$   $\triangleright$  rotate  $\begin{pmatrix} d_x \\ d_y \end{pmatrix}$  by  $\pi/8$ 

3: if  $d'_y < 0$  then
4:    $d'_x \leftarrow -d'_x, \quad d'_y \leftarrow -d'_y$   $\triangleright$  mirror to octants  $0, \dots, 3$ 

5:  $s_\theta \leftarrow \begin{cases} 0 & \text{if } (d'_x \geq 0) \wedge (d'_x \geq d'_y) \\ 1 & \text{if } (d'_x \geq 0) \wedge (d'_x < d'_y) \\ 2 & \text{if } (d'_x < 0) \wedge (-d'_x < d'_y) \\ 3 & \text{if } (d'_x < 0) \wedge (-d'_x \geq d'_y) \end{cases}$ 
6: return  $s_\theta$ .  $\triangleright$  sector index  $s_\theta \in \{0, 1, 2, 3\}$ 

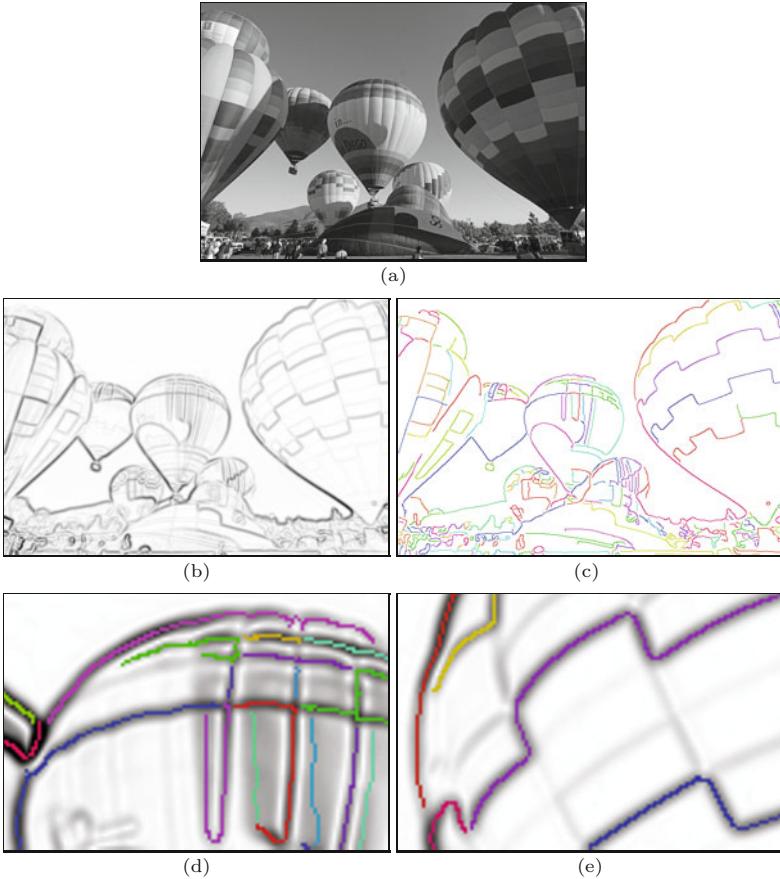
7: IsLocalMax( $E_{\text{mag}}, u, v, s_\theta, t_{\text{lo}}$ )
   Determines if the gradient magnitude  $E_{\text{mag}}$  is a local maximum
   at position  $(u, v)$  in direction  $s_\theta \in \{0, 1, 2, 3\}$ .

8:  $m_C \leftarrow E_{\text{mag}}(u, v)$ 
9: if  $m_C < t_{\text{lo}}$  then
10:   return false
11: else
12:    $m_L \leftarrow \begin{cases} E_{\text{mag}}(u-1, v) & \text{if } s_\theta = 0 \\ E_{\text{mag}}(u-1, v-1) & \text{if } s_\theta = 1 \\ E_{\text{mag}}(u, v-1) & \text{if } s_\theta = 2 \\ E_{\text{mag}}(u-1, v+1) & \text{if } s_\theta = 3 \end{cases}$ 
13:    $m_R \leftarrow \begin{cases} E_{\text{mag}}(u+1, v) & \text{if } s_\theta = 0 \\ E_{\text{mag}}(u+1, v+1) & \text{if } s_\theta = 1 \\ E_{\text{mag}}(u, v+1) & \text{if } s_\theta = 2 \\ E_{\text{mag}}(u+1, v-1) & \text{if } s_\theta = 3 \end{cases}$ 
14: return  $(m_L \leq m_C) \wedge (m_C \geq m_R)$ 

15: TraceAndThreshold( $E_{\text{nms}}, E_{\text{bin}}, u_0, v_0, t_{\text{lo}}$ )
   Recursively collects and marks all pixels of an edge that are 8-
   connected to  $(u_0, v_0)$  and have a gradient magnitude above  $t_{\text{lo}}$ .

16:  $E_{\text{bin}}(u_0, v_0) \leftarrow 1$   $\triangleright$  mark  $(u_0, v_0)$  as an edge pixel
17:  $u_L \leftarrow \max(u_0-1, 0)$   $\triangleright$  limit to image bounds
18:  $u_R \leftarrow \min(u_0+1, M-1)$ 
19:  $v_T \leftarrow \max(v_0-1, 0)$ 
20:  $v_B \leftarrow \min(v_0+1, N-1)$ 
21: for  $u \leftarrow u_L, \dots, u_R$  do
22:   for  $v \leftarrow v_T, \dots, v_B$  do
23:     if  $(E_{\text{nms}}(u, v) \geq t_{\text{lo}}) \wedge (E_{\text{bin}}(u, v) = 0)$  then
24:       TraceAndThreshold( $E_{\text{nms}}, E_{\text{bin}}, u, v, t_{\text{lo}}$ )
25: return
```

ation. This is done with a technique called “hysteresis thresholding” using two different threshold values, t_{lo} (with $t_{\text{hi}} > t_{\text{lo}}$). The image is scanned for pixels with edge magnitude $E_{\text{nms}}(u, v) \geq t_{\text{hi}}$. Whenever such a (previously unvisited) location is found, a new *edge trace* is started and all connected edge pixels (u', v') are added to it as long as $E_{\text{nms}}(u', v') \geq t_{\text{lo}}$. Only those edge traces remain that contain at least one pixel with edge magnitude greater than t_{hi} and no pixels with edge magnitude less than t_{lo} . This process (which is similar to



6.5 CANNY EDGE OPERATOR

Fig. 6.14
Grayscale Canny edge operator details. Inverted gradient magnitude (a), detected edge points with connected edge tracks shown in distinctive colors (b). Details with gradient magnitude and detected edge points overlaid (c, d). Settings: $\sigma = 2.0$, $t_{hi} = 20\%$, $t_{lo} = 5\%$ (of the max. edge magnitude).

flood-fill region growing) is detailed in procedure `GetOrientationSector` in Alg. 6.2. Typical threshold values for 8-bit grayscale images are $t_{hi} = 5.0$ and $t_{lo} = 2.5$.

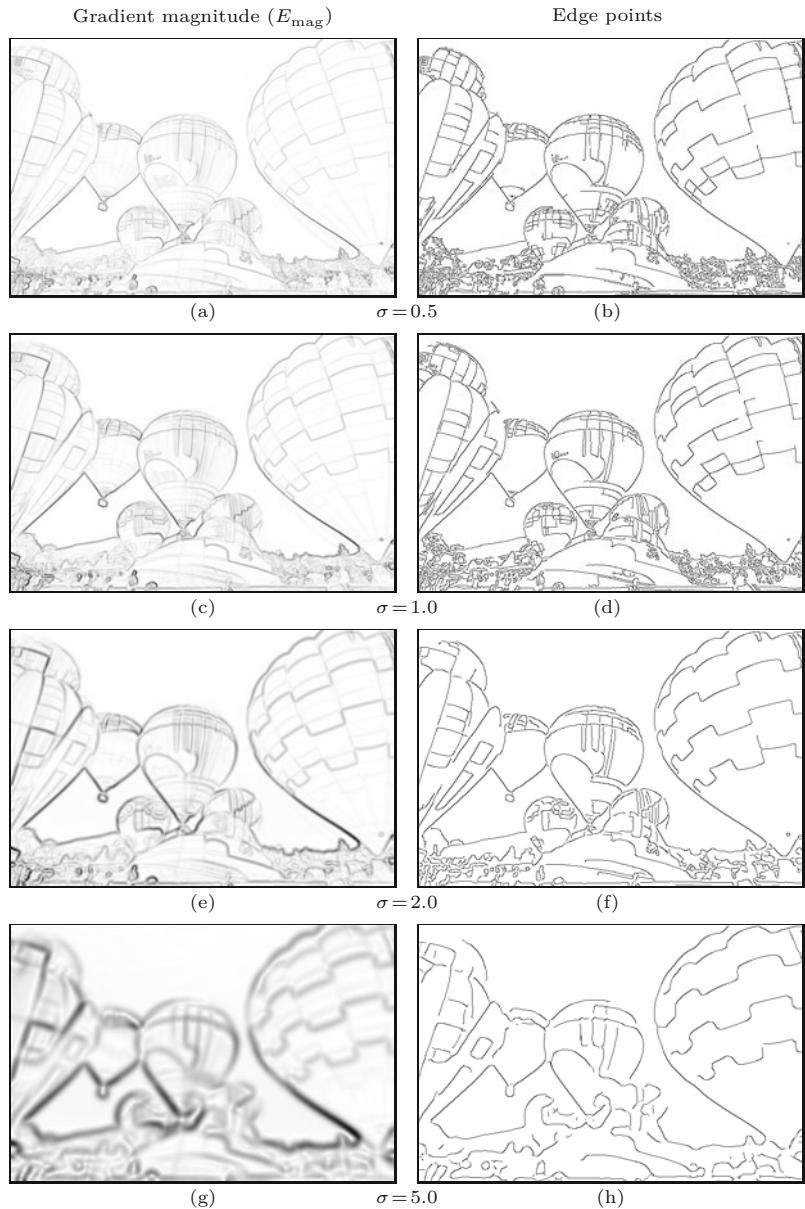
Figure 6.14 illustrates the effectiveness of non-maximum suppression for localizing the edge centers and edge-linking with hysteresis thresholding. Results from the single-scale Canny detector are shown in Fig. 6.15 for different settings of σ and fixed upper/lower threshold values $t_{hi} = 20\%$, $t_{lo} = 5\%$ (relative to the maximum gradient magnitude).

6.5.4 Additional Information

Due to the long-lasting popularity of the Canny operator, additional descriptions and some excellent illustrations can be found at various places in the literature, including [89, p. 719], [232, pp. 71–80], and [166, pp. 548–549]. An edge operator similar to the Canny detector, but based on a set of recursive filters, is described in [62]. While the Canny detector was originally designed for grayscale images, modified versions for color images exist, including the one we describe in the next section.

Fig. 6.15

Results from the single-scale grayscale Canny edge operator (Algs. 6.1–6.2) for different values of $\sigma = 0.5, \dots, 5.0$. Inverted gradient magnitude (left column) and detected edge points (right column). The detected edge points (right column) are linked to connected edge chains.



6.5.5 Implementation

A complete implementation of the Canny edge detector for both grayscale and RGB color images can be found in the Java library for this book.⁷ A basic usage example Prog. 16.1 is shown in Prog. 16.1 on p. 411.

⁷ Class `CannyEdgeDetector` in package `imagingbook.pub.coloredge`.

Making images look sharper is a frequent task, such as to make up for a lack of sharpness after scanning or scaling an image or to pre-compensate for a subsequent loss of sharpness in the course of printing or displaying an image. A common approach to image sharpening is to amplify the high-frequency image components, which are mainly responsible for the perceived sharpness of an image and for which the strongest occur at rapid intensity transitions. In the following, we describe two methods for artificial image sharpening that are based on techniques similar to edge detection and thus fit well in this chapter. In the following, we describe two methods for artificial image sharpening that are based on techniques similar to edge detection and thus fit well in this chapter.

6.6.1 Edge Sharpening with the Laplacian Filter

A common method for localizing rapid intensity changes are filters based on the second derivatives of the image function. Figure 6.16 illustrates this idea on a 1D, continuous function $f(x)$. The second derivative $f''(x)$ of the step function shows a positive pulse at the lower end of the transition and a negative pulse at the upper end. The edge is sharpened by subtracting a certain fraction w of the second derivative $f''(x)$ from the original function $f(x)$,

$$\hat{f}(x) = f(x) - w \cdot f''(x). \quad (6.29)$$

Depending upon the weight factor $w \geq 0$, the expression in Eqn. (6.29) causes the intensity function to overshoot at both sides of an edge, thus exaggerating edges and increasing the perceived sharpness.

Laplacian operator

Sharpening of a 2D function can be accomplished with the second derivatives in the horizontal and vertical directions combined by the so-called Laplacian operator. The Laplacian operator ∇^2 of a 2D function $f(x, y)$ is defined as the sum of the second partial derivatives along the x and y directions:

$$(\nabla^2 f)(x, y) = \frac{\partial^2 f}{\partial^2 x}(x, y) + \frac{\partial^2 f}{\partial^2 y}(x, y). \quad (6.30)$$

Similar to the first derivatives (see Sec. 6.2.2), the second derivatives of a discrete image function can also be estimated with a set of simple linear filters. Again, several versions, have been proposed. For example, the two 1D filters

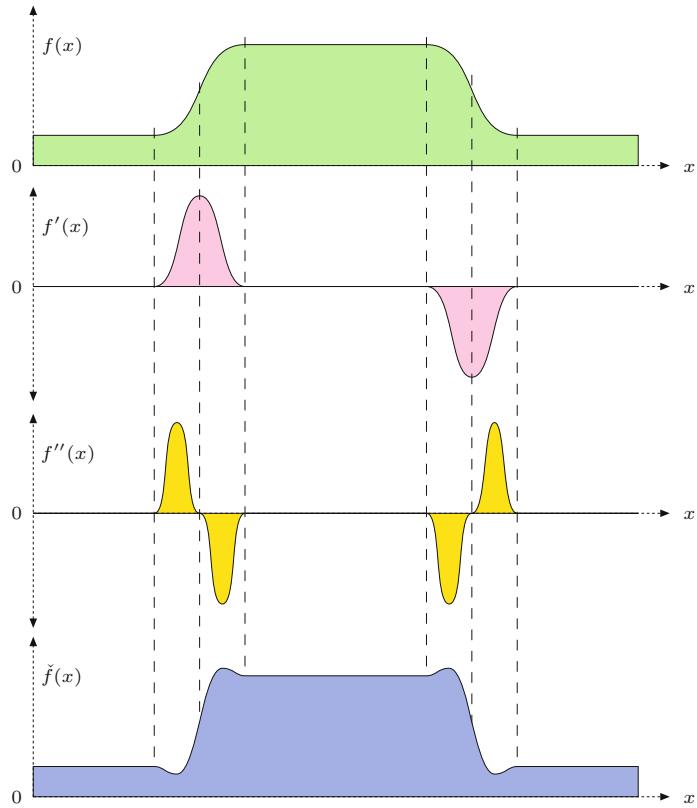
$$\frac{\partial^2 f}{\partial^2 x} \approx H_x^L = [1 \textcolor{red}{-}2 \ 1] \quad \text{and} \quad \frac{\partial^2 f}{\partial^2 y} \approx H_y^L = \begin{bmatrix} 1 \\ \textcolor{red}{-}2 \\ 1 \end{bmatrix}, \quad (6.31)$$

for estimating the second derivatives along the x and y directions, respectively, combine to make the 2D Laplacian filter

6 EDGES AND CONTOURS

Fig. 6.16

Edge sharpening with the second derivative. The original intensity function $f(x)$, first derivative $f'(x)$, second derivative $f''(x)$, and sharpened intensity function $\hat{f}(x) = f(x) - w \cdot f''(x)$ are shown (w is a weighting factor).



$$H^L = H_x^L + H_y^L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}. \quad (6.32)$$

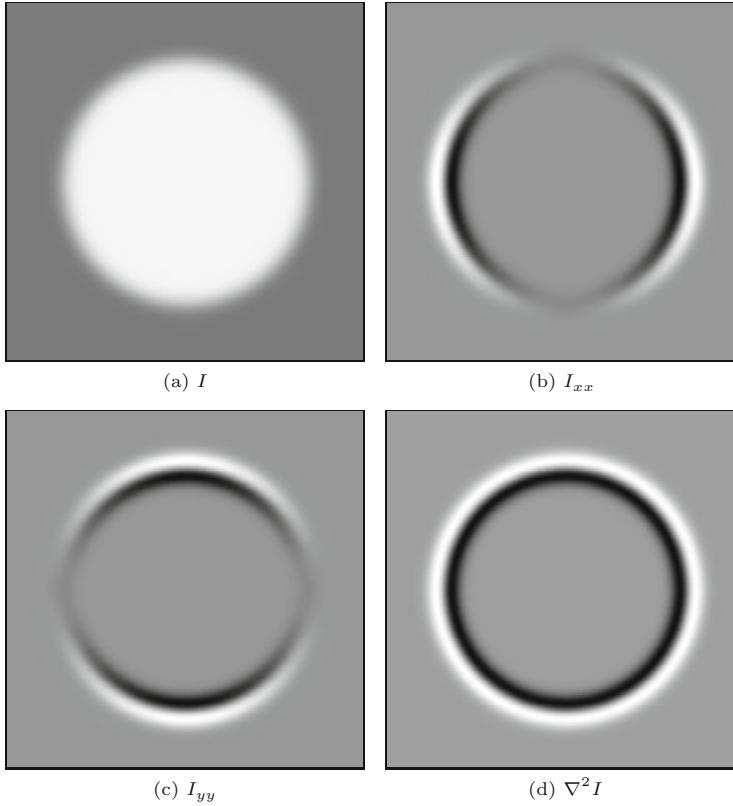
Figure 6.17 shows an example of applying the Laplacian filter H^L to a grayscale image, where the pairs of positive-negative peaks at both sides of each edge are clearly visible. The filter appears almost isotropic despite the coarse approximation with the small filter kernels.

Notice that H^L in Eqn. (6.32) is not *separable* in the usual sense (as described in Sec. 5.3.3) but, because of the linearity property of convolution (Eqns. (5.21) and (5.23)), it can be expressed (and computed) as the *sum* of two 1D filters,

$$I * H^L = I * (H_x^L + H_y^L) = (I * H_x^L) + (I * H_y^L) = I_{xx} + I_{yy}. \quad (6.33)$$

Analogous to the gradient filters (for estimating the first derivatives), the sum of the coefficients is zero in any Laplace filter, such that its response is zero in areas of constant (flat) intensity (Fig. 6.17). Other common variants of 3×3 pixel Laplace filters are

$$H_8^L = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{oder} \quad H_{12}^L = \begin{bmatrix} 1 & 2 & 1 \\ 2 & -12 & 2 \\ 1 & 2 & 1 \end{bmatrix}. \quad (6.34)$$



6.6 EDGE SHARPENING

Fig. 6.17

Results of Laplace filter H^L : synthetic test image I (a), second partial derivative $I_{xx} = \partial^2 I / \partial^2 x$ in the horizontal direction (b), second partial derivative $I_{yy} = \partial^2 I / \partial^2 y$ in the vertical direction (c), and Laplace filter $\nabla^2 I = I_{xx} + I_{yy}$ (d). Intensities in (b-d) are scaled such that maximally negative and positive values are shown as black and white, respectively, and zero values are gray.

Sharpening

To perform the actual sharpening, as described by Eqn. (6.29) for the 1D case, we first apply a Laplacian filter H^L to the image I and then subtract a fraction of the result from the original image,

$$I' \leftarrow I - w \cdot (H^L * I). \quad (6.35)$$

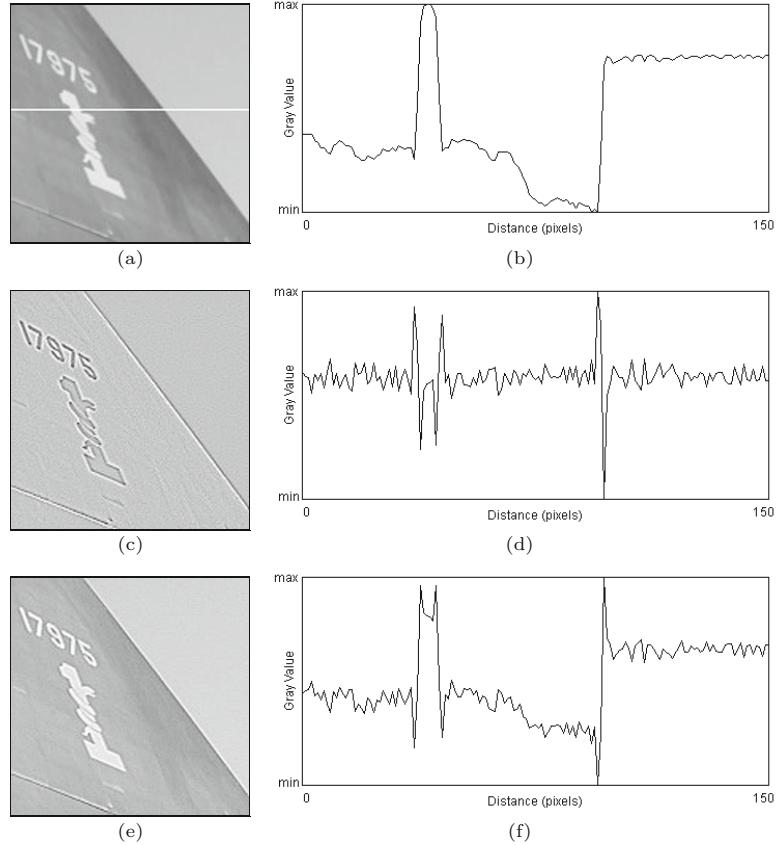
The factor w specifies the proportion of the Laplacian component and thus the sharpening strength. The proper choice of w also depends on the specific Laplacian filter used in Eqn. (6.35) since none of the aforementioned filters is normalized.

Figure 6.17 shows the result of applying a Laplacian filter (with the kernel given in Eqn. (6.32)) to a synthetic test image where the pairs of positive/negative peaks at both sides of each edge are clearly visible. The filter appears almost isotropic despite the coarse approximation with the small filter kernels. The application to a real grayscale image using the filter H^L (Eqn. (6.32)) and $w = 1.0$ is shown in Fig. 6.18.

As we can expect from second-order derivatives, the Laplacian filter is fairly sensitive to image noise, which can be reduced (as is commonly done in edge detection with first derivatives) by previous smoothing, such as with a Gaussian filter (see also Sec. 6.4.1).

Fig. 6.18

Edge sharpening with the Laplacian filter. Original image with a horizontal profile taken from the marked line (a, b), result of Laplacian filter H^L (c, d), and sharpened image with sharpening factor $w = 1.0$ (e, f).



6.6.2 Unsharp Masking

“Unsharp masking” (USM) is a technique for edge sharpening that is particularly popular in astronomy, digital printing, and many other areas of image processing. The term originates from classical photography, where the sharpness of an image was optically enhanced by combining it with a smoothed (“unsharp”) copy. This process is in principle the same for digital images.

Process

The first step in the USM filter is to subtract a smoothed version of the image from the original, which enhances the edges. The result is called the “mask”. In analog photography, the required smoothing was achieved by simply defocusing the lens. Subsequently, the mask is again added to the original, such that the edges in the image are sharpened. In summary, the steps involved in USM filtering are:

1. The mask image M is generated by subtracting (from the original image I) a smoothed version of I , obtained by filtering with \tilde{H} , that is,

$$M \leftarrow I - (I * \tilde{H}) = I - \tilde{I}. \quad (6.36)$$

The kernel \tilde{H} of the smoothing filter is assumed to be normalized (see Sec. 5.2.5).

2. To obtain the sharpened image \check{I} , the mask M is added to the original image I , weighted by the factor a , which controls the amount of sharpening,

$$\check{I} \leftarrow I + a \cdot M, \quad (6.37)$$

and thus (by inserting from Eqn. (6.36))

$$\check{I} \leftarrow I + a \cdot (I - \tilde{I}) = (1 + a) \cdot I - a \cdot \tilde{I}. \quad (6.38)$$

Smoothing filter

In principle, any smoothing filter could be used for the kernel \tilde{H} in Eqn. (6.36), but Gaussian filters $H^{G,\sigma}$ with variable radius σ are most common (see also Sec. 5.2.7). Typical parameter values are 1 to 20 for σ and 0.2 to 4.0 (equivalent to 20% to 400%) for the sharpening factor a .

Figure 6.19 shows two examples of USM filters using Gaussian smoothing filters with different radii σ .

Extensions

The advantages of the USM filter over the Laplace filter are reduced noise sensitivity due to the involved smoothing and improved controllability through the parameters σ (spatial extent) and a (sharpening strength).

Of course the USM filter responds not only to real edges but to some extent to any intensity transition, and thus potentially increases any visible noise in continuous image regions. Some implementations (e.g., Adobe Photoshop) therefore provide an additional *threshold* parameter t_c to specify the *minimum local contrast* required to perform edge sharpening. Sharpening is only applied if the local contrast at position (u, v) , expressed, for example, by the gradient magnitude $|\nabla I|$ (Eqn. (6.5)), is greater than that threshold. Otherwise, that pixel remains unmodified, that is,

$$\check{I}(u, v) \leftarrow \begin{cases} I(u, v) + a \cdot M(u, v) & \text{for } |\nabla I|(u, v) \geq t_c, \\ I(u, v) & \text{otherwise.} \end{cases} \quad (6.39)$$

Different to the original USM filter (Eqn. (6.37)), this extended version is no longer a *linear* filter. On color images, the USM filter is usually applied to all color channels with identical parameter settings.

Implementation

The USM filter is available in virtually any image-processing software and, due to its simplicity and flexibility, has become an indispensable tool for many professional users. In ImageJ, the USM filter is implemented by the plugin class `UnsharpMask`⁸ and can be applied through the menu

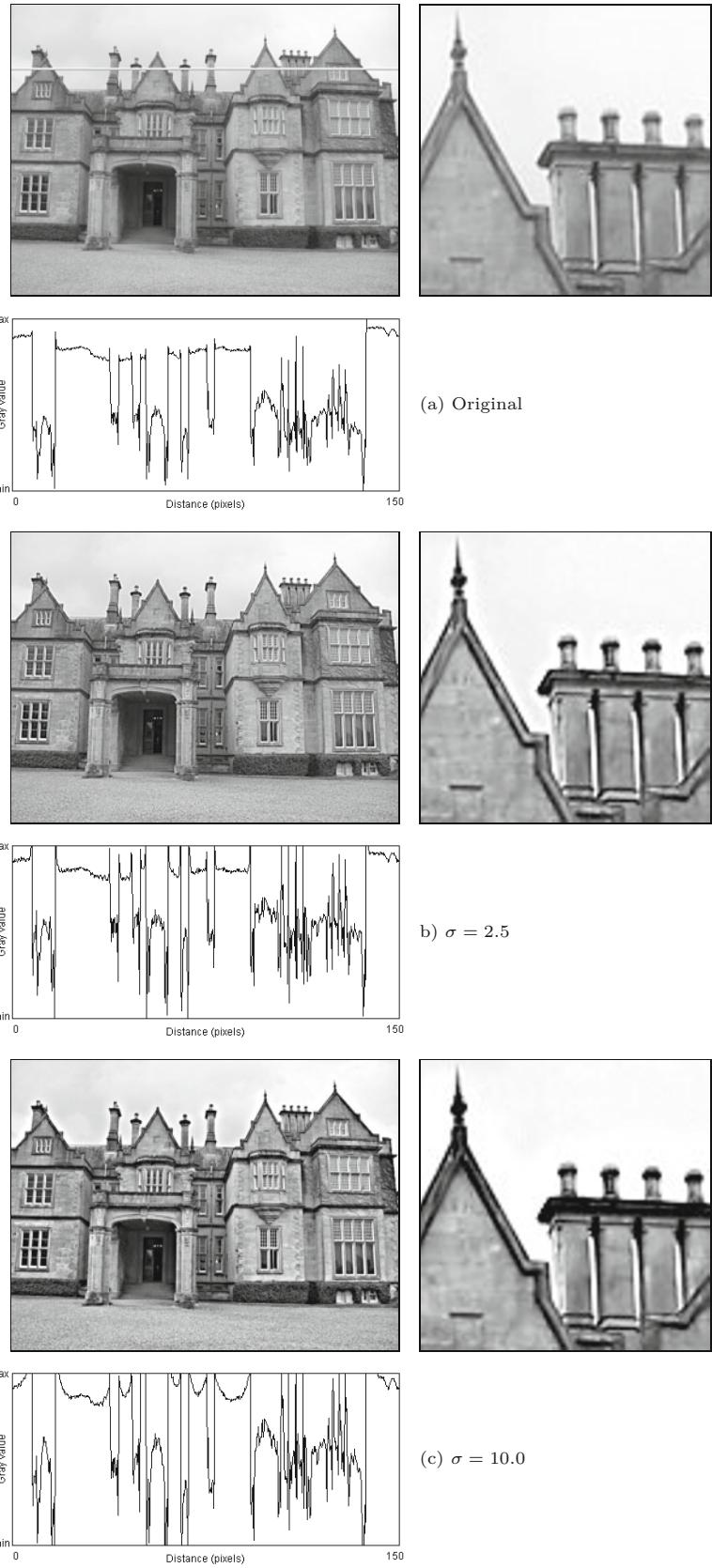
Process ▷ Filter ▷ Unsharp Mask...

⁸ In package `ij.plugin.filter`.

6 EDGES AND CONTOURS

Fig. 6.19

Unsharp masking filters with varying smoothing radii $\sigma = 2.5$ and 10.0 . The sharpening strength a is set to 1.0 (100%). The profiles show the intensity function for the image line marked in the original image (top-left).



This filter can also be used from other plugin classes, for example, in the following way:

```
import ij.plugin.filter.UnsharpMask;
...
public void run(ImageProcessor ip) {
    UnsharpMask usm = new UnsharpMask();
    double r = 2.0; // standard settings for radius
    double a = 0.6; // standard settings for weight
    usm.sharpen(ip, r, a);
...
}
```

ImageJ's `UnsharpMask` implementation uses the class `GaussianBlur` for the required smoothing operation. The alternative implementation shown in Prog. 6.1 follows the definition in Eqn. (6.38) and uses Gaussian filter kernels that are created with the method `makeGaussKernel1d()`, as defined in Prog. 5.4.

```
1  double radius = 1.0; // radius (sigma of Gaussian)
2  double amount = 1.0; // amount of sharpening (1 = 100%)
3  ...
4  public void run(ImageProcessor ip) {
5      ImageProcessor I = ip.convertToFloat(); // I
6
7      // create a blurred version of the image:
8      ImageProcessor J = I.duplicate(); // J
9      float[] H = GaussianFilter.makeGaussKernel1d(sigma);
10     Convolver cv = new Convolver();
11     cv.setNormalize(true);
12     cv.convolve(J, H, 1, H.length);
13     cv.convolve(J, H, H.length, 1);
14
15     I.multiply(1 + a); // I ← (1 + a) · I
16     J.multiply(a); // J ← a · J
17     I.copyBits(J, 0, 0, Blitter.SUBTRACT); // I ← (1+a) · I - a · J
18
19     // copy result back into original byte image
20     ip.insert(I.convertToByte(false), 0, 0);
21 }
```

6.6 EDGE SHARPENING

Prog. 6.1

Unsharp masking (Java implementation). First the original image is converted to a `FloatProcessor` object `I` (I) in line 5, which is duplicated to hold the blurred image `J` (\tilde{I}) in line 8. The method `makeGaussKernel1d()`, defined in Prog. 5.4, is used to create the 1D Gaussian filter kernel applied in the horizontal and vertical directions (lines 12–13). The remaining calculations follow Eqn. (6.38).

Laplace vs. USM filter

A closer look at these two methods reveals that sharpening with the Laplace filter (Sec. 6.6.1) can be viewed as a special case of the USM filter. If the Laplace filter in Eqn. (6.32) is decomposed as

$$H^L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} - 5 \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = 5 \cdot (\tilde{H}^L - \delta), \quad (6.40)$$

one can see that H^L consists of a simple 3×3 pixel smoothing filter \tilde{H} minus the impulse function δ . Laplace sharpening with the weight factor w as defined in Eqn. (6.35) can therefore (by a little manipulation) be expressed as

$$\begin{aligned}
\check{I}_L &\leftarrow I - w \cdot (H^L * I) = I - w \cdot (5(\tilde{H}^L - \delta) * I) \\
&= I - 5w \cdot (\tilde{H}^L * I - I) = I + 5w \cdot (I - \tilde{H}^L * I) \\
&= I + 5w \cdot M^L,
\end{aligned} \tag{6.41}$$

that is, in the form of a USM filter $\check{I} \leftarrow I + a \cdot M$ (Eqn. (6.37)). Laplacian sharpening is thus a special case of a USM filter with the mask $M = M^L = (I - \tilde{H}^L * I)$, the specific smoothing filter

$$\tilde{H}^L = \frac{1}{5} \begin{bmatrix} 0 & 1 & 0 \\ 1 & \textcolor{red}{1} & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

and the sharpening factor $a = 5w$.

6.7 Exercises

Exercise 6.1. Calculate (manually) the gradient and the Laplacian (using the discrete approximations in Eqn. (6.2) and Eqn. (6.32), respectively) for the following “image”:

$$I = \begin{bmatrix} 14 & 10 & 19 & 16 & 14 & 12 \\ 18 & 9 & 11 & 12 & 10 & 19 \\ 9 & 14 & 15 & 26 & 13 & 6 \\ 21 & 27 & 17 & 17 & 19 & 16 \\ 11 & 18 & 18 & 19 & 16 & 14 \\ 16 & 10 & 13 & 7 & 22 & 21 \end{bmatrix}.$$

Exercise 6.2. Implement the Sobel edge operator as defined in Eqn. (6.10) (and illustrated in Fig. 6.6) as an ImageJ plugin. The plugin should generate two new images for the edge magnitude $E(u, v)$ and the edge orientation $\Phi(u, v)$. Come up with a suitable way to display local edge orientation.

Exercise 6.3. Express the Sobel operator (Eqn. (6.10)) in x/y -separable form analogous to the decomposition of the Prewitt operator in Eqn. (6.9).

Exercise 6.4. Implement the Kirsch operator (Eqns. (6.25)–(6.28)) analogous to the two-directional Sobel operator in Exercise 6.2 and compare the results from both methods, particularly the edge orientation estimates.

Exercise 6.5. Devise and implement a compass edge operator with more than eight (16?) differently oriented filters.

Exercise 6.6. Compare the results of the unsharp masking filters in ImageJ and Adobe Photoshop using a suitable test image. How should the parameters for σ (*radius*) and a (*weight*) be defined in both implementations to obtain similar results?

Corner Detection

Corners are prominent structural elements in an image and are therefore useful in a wide variety of applications, including following objects across related images (*tracking*), determining the correspondence between stereo images, serving as reference points for precise geometrical measurements, and calibrating camera systems for machine vision applications. Thus corner points are not only important in human vision but they are also “robust” in the sense that they do not arise accidentally in 3D scenes and, furthermore, can be located quite reliably under a wide range of viewing angles and lighting conditions.

7.1 Points of Interest

Despite being easily recognized by our visual system, accurately and precisely detecting corners automatically is not a trivial task. A good corner detector must satisfy a number of criteria, including distinguishing between true and accidental corners, reliably detecting corners in the presence of realistic image noise, and precisely and accurately determining the locations of corners. Finally, it should also be possible to implement the detector efficiently enough so that it can be utilized in real-time applications such as video tracking.

Numerous methods for finding corners or similar interest points have been proposed and most of them take advantage of the following basic principle. While an *edge* is usually defined as a location in the image at which the gradient is especially high in *one* direction and low in the direction normal to it, a *corner point* is defined as a location that exhibits a strong gradient value in *multiple* directions at the same time.

Most methods take advantage of this observation by examining the first or second derivative of the image in the x and y directions to find corners (e.g., [77, 102, 137, 154]). In the next section, we describe in detail the Harris detector, also known as the “Plessey feature point detector” [102], since it turns out that even though more efficient

detectors are known (see, e.g., [210, 220]), the Harris detector, and other detectors based on it, are the most widely used in practice.

7.2 Harris Corner Detector

This operator, developed by Harris and Stephens [102], is one of a group of related methods based on the same premise: a corner point exists where the gradient of the image is especially strong in more than one direction at the same time. In addition, locations along edges, where the gradient is strong in only one direction, should not be considered as corners, and the detector should be isotropic, that is, independent of the orientation of the local gradients.

7.2.1 Local Structure Matrix

The Harris corner detector is based on the first partial derivatives (gradient) of the image function $I(u, v)$, that is,

$$I_x(u, v) = \frac{\partial I}{\partial x}(u, v) \quad \text{and} \quad I_y(u, v) = \frac{\partial I}{\partial y}(u, v). \quad (7.1)$$

For each image position (u, v) , we first calculate the three quantities

$$A(u, v) = I_x^2(u, v), \quad (7.2)$$

$$B(u, v) = I_y^2(u, v), \quad (7.3)$$

$$C(u, v) = I_x(u, v) \cdot I_y(u, v) \quad (7.4)$$

that constitute the elements of the *local structure matrix* $\mathbf{M}(u, v)$:¹

$$\mathbf{M} = \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix} = \begin{pmatrix} A & C \\ C & B \end{pmatrix}. \quad (7.5)$$

Next, each of the three scalar fields $A(u, v)$, $B(u, v)$, $C(u, v)$ is individually smoothed by convolution with a linear Gaussian filter $H^{G, \sigma}$ (see Sec. 5.2.7),

$$\bar{\mathbf{M}} = \begin{pmatrix} A * H_\sigma^G & C * H_\sigma^G \\ C * H_\sigma^G & B * H_\sigma^G \end{pmatrix} = \begin{pmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{pmatrix}. \quad (7.6)$$

The *eigenvalues*² of the matrix $\bar{\mathbf{M}}$, defined as³

$$\begin{aligned} \lambda_{1,2} &= \frac{\text{trace}(\bar{\mathbf{M}})}{2} \pm \sqrt{\left(\frac{\text{trace}(\bar{\mathbf{M}})}{2}\right)^2 - \det(\bar{\mathbf{M}})} \\ &= \frac{1}{2} \cdot \left(\bar{A} + \bar{B} \pm \sqrt{\bar{A}^2 - 2 \cdot \bar{A} \cdot \bar{B} + \bar{B}^2 + 4 \cdot \bar{C}^2} \right), \end{aligned} \quad (7.7)$$

¹ For improved legibility, we simplify the notation used in the following by omitting the function coordinates (u, v) ; for example, the function $I_x(u, v)$ is abbreviated as I_x or $A(u, v)$ is simply denoted A etc.

² See also Sec. B.4 in the Appendix.

³ $\det(\bar{\mathbf{M}})$ denotes the *determinant* and $\text{trace}(\bar{\mathbf{M}})$ denotes the *trace* of the matrix $\bar{\mathbf{M}}$ (see, e.g., [35, pp. 252 and 259]).

are (because the matrix is symmetric) positive and real. They contain essential information about the local image structure. Within an image region that is uniform (that is, appears flat), $\bar{\mathbf{M}} = 0$ and therefore $\lambda_1 = \lambda_2 = 0$. On an ideal ramp, however, the eigenvalues are $\lambda_1 > 0$ and $\lambda_2 = 0$, independent of the orientation of the edge. The eigenvalues thus encode an edge's *strength*, and their associated *eigenvectors* correspond to the local edge *orientation*.

A corner should have a strong edge in the main direction (corresponding to the larger of the two eigenvalues), another edge normal to the first (corresponding to the smaller eigenvalues), and both eigenvalues must be significant. Since $\bar{A}, \bar{B} \geq 0$, we can assume that $\text{trace}(\bar{\mathbf{M}}) > 0$ and thus $|\lambda_1| \geq |\lambda_2|$. Therefore only the smaller of the two eigenvalues, $\lambda_2 = \text{trace}(\bar{\mathbf{M}})/2 - \sqrt{\dots}$, is relevant when determining a corner.

7.2.2 Corner Response Function (CRF)

From Eqn. (7.7) we see that the difference between the two eigenvalues of the local structure matrix is

$$\lambda_1 - \lambda_2 = 2 \cdot \sqrt{0.25 \cdot (\text{trace}(\bar{\mathbf{M}}))^2 - \det(\bar{\mathbf{M}})}, \quad (7.8)$$

where the expression under the square root is always non-negative. At a good corner position, the difference between the two eigenvalues λ_1, λ_2 should be as small as possible and thus the expression under the root in Eqn. (7.8) should be a minimum. To avoid the explicit calculation of the eigenvalues (and the square root) the Harris detector defines the function

$$\begin{aligned} Q(u, v) &= \det(\bar{\mathbf{M}}(u, v)) - \alpha \cdot (\text{trace}(\bar{\mathbf{M}}(u, v)))^2 \\ &= \bar{A}(u, v) \cdot \bar{B}(u, v) - \bar{C}^2(u, v) - \alpha \cdot [\bar{A}(u, v) + \bar{B}(u, v)]^2 \end{aligned} \quad (7.9)$$

as a measure of “corner strength”, where the parameter α determines the sensitivity of the detector. $Q(u, v)$ is called the “corner response function” and returns maximum values at isolated corners. In practice, α is assigned a fixed value in the range of 0.04 to 0.06 (max. $0.25 = \frac{1}{4}$). The larger the value of α , the less sensitive the detector is and the fewer corners detected.

7.2.3 Determining Corner Points

An image location (u, v) is selected as a potential candidate for a corner point if

$$Q(u, v) > t_H,$$

where the threshold t_H is selected based on image content and typically lies within the range of 10,000 to 1,000,000. Once selected, the corners $\mathbf{c}_i = \langle u_i, v_i, q_i \rangle$ are inserted into the sequence

$$\mathcal{C} = (\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_N),$$

which is then sorted in descending order (i.e., $q_i \geq q_{i+1}$) according to *corner strength* $q_i = Q(u_i, v_i)$, as defined in Eqn. (7.9). To suppress

Table 7.1
Harris corner detector—typical parameter settings for Alg. 7.1.

Prefilter (Alg. 7.1, line 2–3): Smoothing with a small xy -separable filter $H_p = H_{px} * H_{py}$, where

$$H_{px} = \frac{1}{9} \cdot \begin{bmatrix} 2 & 5 & 2 \end{bmatrix} \quad \text{and} \quad H_{py} = H_{px}^\top = \frac{1}{9} \cdot \begin{bmatrix} 2 \\ 5 \\ 2 \end{bmatrix}.$$

Gradient filter (Alg. 7.1, line 3): Computing the first partial derivative in the x and y directions with

$$h_{dx} = \begin{bmatrix} -0.5 & 0.5 \end{bmatrix} \quad \text{and} \quad h_{dy} = h_{dx}^\top = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}.$$

Blur filter (Alg. 7.1, line 10): Smoothing the individual components of the structure matrix M with separable Gaussian filters

$$H_b = H_{bx} * H_{by} \text{ with}$$

$$h_{bx} = \frac{1}{64} \cdot \begin{bmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{bmatrix} \quad \text{and} \quad h_{by} = h_{bx}^\top = \frac{1}{64} \cdot \begin{bmatrix} 1 \\ 6 \\ 15 \\ 20 \\ 15 \\ 6 \\ 1 \end{bmatrix}.$$

Control parameter (Alg. 7.1, line 14): $\alpha = 0.04, \dots, 0.06$ (default 0.05).

Response threshold (Alg. 7.1, line 19): $t_H = 10\,000, \dots, 1\,000\,000$ (default 20 000).

Neighborhood radius (Alg. 7.1, line 37): $d_{\min} = 10$ Pixel.

the false corners that tend to arise in densely packed groups around true corners, all except the strongest corner in a specified vicinity are eliminated. To accomplish this, the list \mathcal{C} is traversed from the front to the back, and the weaker corners toward the end of the list, which lie in the surrounding neighborhood of a stronger corner, are deleted.

The complete algorithm for the Harris detector is summarized again in Alg. 7.1; the associated parameters are listed in Table 7.1.

7.2.4 Examples

Figure 7.1 uses a simple synthetic image to illustrate the most important steps in corner detection using the Harris detector. The figure shows the result of the gradient computation, the three components of the structure matrix $M(u, v) = \begin{pmatrix} A & C \\ C & B \end{pmatrix}$, and the values of the *corner response function* $Q(u, v)$ for each image position (u, v) . This example was calculated with the standard settings as given in Table 7.1.

The second example (Fig. 7.2) illustrates the detection of corner points in a grayscale representation of a natural scene. It demonstrates how weak corners are eliminated in favor of the strongest corner in a region.

1: **HarrisCorners**(I, α, t_H, d_{\min})

Input: I , the source image; α , sensitivity parameter (typ. 0.05); t_H , response threshold (typ. 20 000); d_{\min} , minimum distance between final corners. Returns a sequence of the strongest corners detected in I .

Step 1 – calculate the corner response function:

- 2: $I_x \leftarrow (I * h_{px}) * h_{dx}$ \triangleright horizontal prefilter and derivative
- 3: $I_y \leftarrow (I * h_{py}) * h_{dy}$ \triangleright vertical prefilter and derivative
- 4: $(M, N) \leftarrow \text{Size}(I)$
- 5: Create maps $A, B, C, Q: M \times N \mapsto \mathbb{R}$
- 6: **for all** image coordinates (u, v) **do**
 Compute the local structure matrix $\mathbf{M} = \begin{pmatrix} A & C \\ C & B \end{pmatrix}$:
- 7: $A(u, v) \leftarrow (I_x(u, v))^2$
- 8: $B(u, v) \leftarrow (I_y(u, v))^2$
- 9: $C(u, v) \leftarrow I_x(u, v) \cdot I_y(u, v)$

Blur the components of the local structure matrix ($\bar{\mathbf{M}}$):

- 10: $\bar{A} \leftarrow A * H_b$
- 11: $\bar{B} \leftarrow B * H_b$
- 12: $\bar{C} \leftarrow C * H_b$
- 13: **for all** image coordinates (u, v) **do** \triangleright calc. corner response:
- 14: $Q(u, v) \leftarrow \bar{A}(u, v) \cdot \bar{B}(u, v) - \bar{C}^2(u, v) - \alpha \cdot [\bar{A}(u, v) + \bar{B}(u, v)]^2$
- 15: **Step 2** – collect the corner points:
- 16: $\mathcal{C} \leftarrow ()$ \triangleright start with an empty corner sequence
- 17: **for all** image coordinates (u, v) **do**
 if $Q(u, v) > t_H \wedge \text{IsLocalMax}(Q, u, v)$ **then**
 $c \leftarrow \langle u, v, Q(u, v) \rangle$ \triangleright create a new corner c
 $\mathcal{C} \leftarrow \mathcal{C} \cup (c)$ \triangleright add c to corner sequence \mathcal{C}
- 20: $\mathcal{C}_{\text{clean}} \leftarrow \text{CleanUpCorners}(\mathcal{C}, d_{\min})$
- 21: **return** $\mathcal{C}_{\text{clean}}$

-
- 22: **IsLocalMax**(Q, u, v) \triangleright determine if $Q(u, v)$ is a local maximum
 - 23: $\mathcal{N} \leftarrow \text{GetNeighbors}(Q, u, v)$ \triangleright se below
 - 24: **return** $Q(u, v) > \max(\mathcal{N})$ \triangleright true or false

25: **GetNeighbors**(Q, u, v)

Returns the 8 neighboring values around $Q(u, v)$.

- 26: $\mathcal{N} \leftarrow (Q(u+1, v), Q(u+1, v-1), Q(u, v-1), Q(u-1, v-1),$
 $Q(u-1, v), Q(u-1, v+1), Q(u, v+1), Q(u+1, v+1))$
- 27: **return** \mathcal{N}

28: **CleanUpCorners**(\mathcal{C}, d_{\min})

- 29: $\text{Sort}(\mathcal{C})$ \triangleright sort \mathcal{C} by desc. q_i (strongest corners first)
- 30: $\mathcal{C}_{\text{clean}} \leftarrow ()$ \triangleright empty “clean” corner sequence
- 31: **while** \mathcal{C} is not empty **do**
 $c_0 \leftarrow \text{GetFirst}(\mathcal{C})$ \triangleright get the strongest corner from \mathcal{C}
 $\mathcal{C} \leftarrow \text{Delete}(c_0, \mathcal{C})$ \triangleright the 1st element is removed from \mathcal{C}
 $\mathcal{C}_{\text{clean}} \leftarrow \mathcal{C}_{\text{clean}} \cup (c_0)$ \triangleright add c_0 to $\mathcal{C}_{\text{clean}}$
 for all c_j in \mathcal{C} **do**
 if $\text{Dist}(c_0, c_j) < d_{\min}$ **then**
 $\mathcal{C} \leftarrow \text{Delete}(c_j, \mathcal{C})$ \triangleright remove element c_j from \mathcal{C}
- 38: **return** $\mathcal{C}_{\text{clean}}$

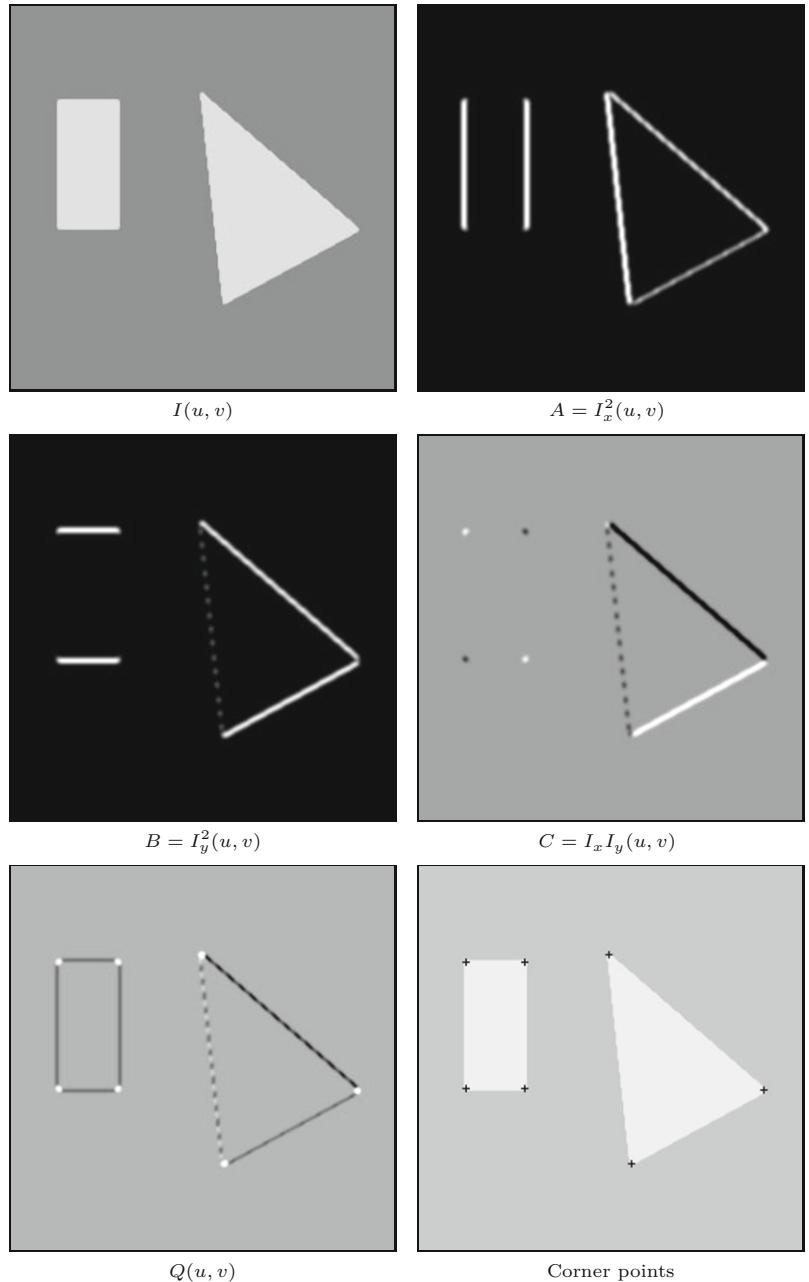
7.2 HARRIS CORNER DETECTOR

Alg. 7.1

Harris corner detector. This algorithm takes an intensity image I and creates a sorted list of detected corner points. $*$ is the convolution operator used for linear filter operations. Details for the parameters H_p , H_{dx} , H_{dy} , H_b , α , and t_H can be found in Table 7.1.

Fig. 7.1

Harris corner detector—
Example 1. Starting with the original image $I(u, v)$, the first derivative is computed, and then from it the components of the structure matrix $M(u, v)$, with $A(u, v) = I_x^2(u, v)$, $B = I_y^2(u, v)$, $C = I_x(u, v) \cdot I_y(u, v)$. $A(u, v)$ and $B(u, v)$ represent, respectively, the strength of the horizontal and vertical edges. In $C(u, v)$, the values are strongly positive (white) or strongly negative (black) only where the edges are strong in both directions (null values are shown in gray). The corner response function, $Q(u, v)$, exhibits noticeable positive peaks at the corner positions.



7.3 Implementation

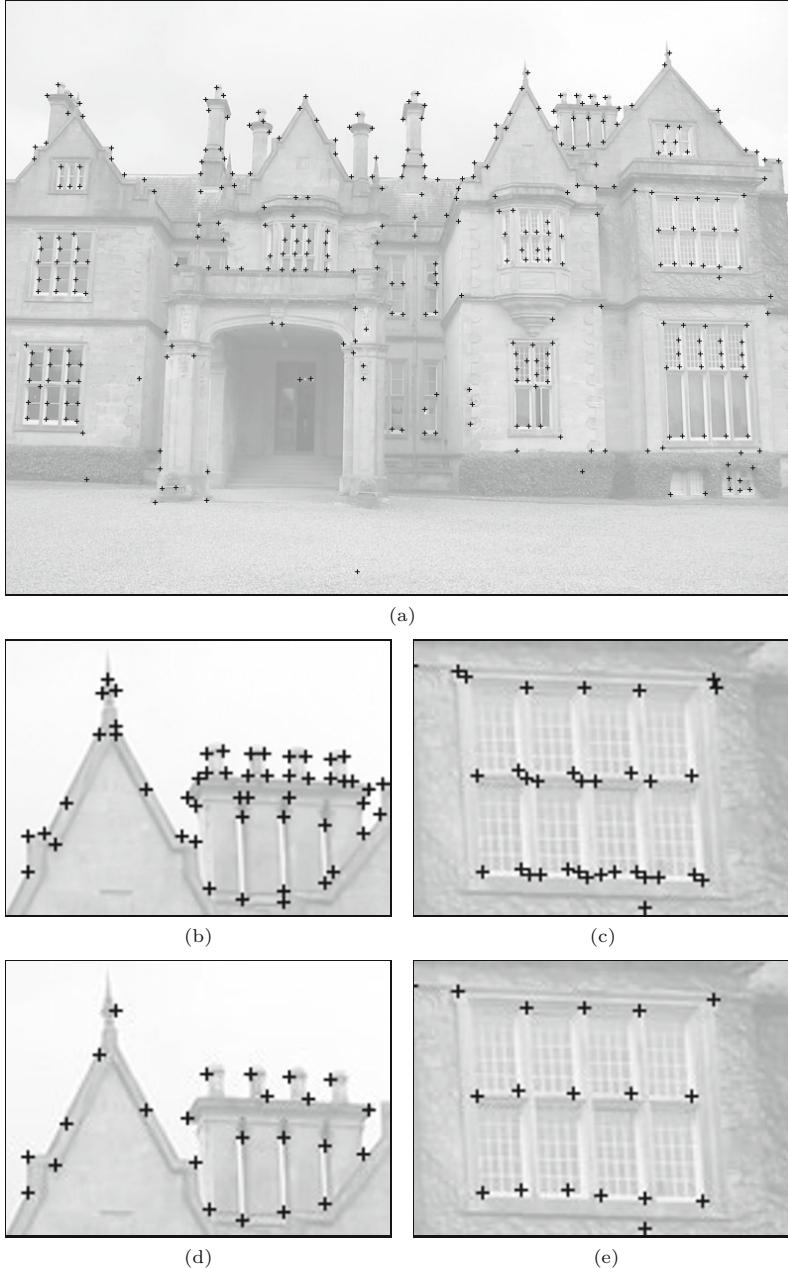
Since the Harris detector algorithm is more complex than the algorithms we presented earlier, in the following sections we explain its implementation in greater detail. While reading the following you may wish to refer to the complete source code for the class `HarrisCornerDetector`, which is available online as part of the `imagingbook` library.⁴

⁴ Package `imagingbook.pub.corners`.

7.3 IMPLEMENTATION

Fig. 7.2

Harris corner detector—Example 2. A complete result with the final corner points marked (a). After selecting the strongest corner points within a 10-pixel radius, only 335 of the original 615 candidate corners remain. Details before (b, c) and after selection (d, e).



7.3.1 Step 1: Calculating the Corner Response Function

To handle the range of the positive and negative values generated by the filters used in this step, we will need to use floating-point images to store the intermediate results, which also assures sufficient range and precision for small values. The kernels of the required filters, that is, the presmoothing filter H_p , the gradient filters H_{dx} , H_{dy} , and the smoothing filter for the structure matrix H_b , are defined as 1D float arrays:

```
1 float[] hp = {2f/9, 5f/9, 2f/9};
```

```

2 float[] hd = {0.5f, 0, -0.5f};
3 float[] hb =
4     {1f/64, 6f/64, 15f/64, 20f/64, 15f/64, 6f/64, 1f/64};

```

From the original 8-bit image (of type `ByteProcessor`), we first create two copies, `Ix` and `Iy`, of type `FloatProcessor`:

```

5  FloatProcessor Ix = I.convertToFloatProcessor();
6  FloatProcessor Iy = I.convertToFloatProcessor();

```

The first processing step is to presmooth the image with the 1D filter kernel hp ($= h_{px} = h_{py}^T$, see Alg. 7.1, line 2). Subsequently the 1D gradient filter hd ($= h_{dx} = h_{dy}^T$) is used to calculate the horizontal and vertical derivatives (see Alg. 7.1, line 3). To perform the convolution with the corresponding 1D kernels we use the (static) methods `convolveX()` and `convolveY()` defined in class `Filter`:⁵

```

7  Filter.convolveX(Ix, hp);           //  $I_x \leftarrow I_x * h_{px}$ 
8  Filter.convolveX(Ix, hd);           //  $I_x \leftarrow I_x * h_{dx}$ 
9  Filter.convolveY(Iy, hp);           //  $I_y \leftarrow I_y * h_{py}$ 
10 Filter.convolveY(Iy, hd);          //  $I_y \leftarrow I_y * h_{dy}$ 

```

Now the components $A(u, v)$, $B(u, v)$, $C(u, v)$ of the structure matrix \mathbf{M} are calculated for all image positions (u, v) :

```

11 A = ImageMath.sqr(Ix);           //  $A(u, v) \leftarrow I_x^2(u, v)$ 
12 B = ImageMath.sqr(Iy);           //  $B(u, v) \leftarrow I_y^2(u, v)$ 
13 C = ImageMath.mult(Ix, Iy);      //  $C(u, v) \leftarrow I_x(u, v) \cdot I_y(u, v)$ 
14

```

The components of the structure matrix are then smoothed with a separable filter kernel $H_b = h_{bx} * h_{by}$:

```

15 Filter.convolveXY(A, hb);        //  $A \leftarrow (A * h_{bx}) * h_{by}$ 
16 Filter.convolveXY(B, hb);        //  $B \leftarrow (B * h_{bx}) * h_{by}$ 
17 Filter.convolveXY(C, hb);        //  $C \leftarrow (C * h_{bx}) * h_{by}$ 

```

The variables `A`, `B`, `C` of type `FloatProcessor` are declared in the class `HarrisCornerDetector`. `sqr()` and `mult()` are static methods of class `ImageMath` for squaring an image and multiplying two images, respectively. The method `convolveXY(I, h)` is used to apply a x/y -separable 2D convolution with the 1D kernel `h` to the image `I`.

Finally, the corner response function (Alg. 7.1, line 14) is calculated by the method `makeCrf()` and a new image (of type `FloatProcessor`) is created:

```

18 private FloatProcessor makeCrf(float alpha) {
19     FloatProcessor Q = new FloatProcessor(M, N);
20     final float[] pA = (float[]) A.getPixels();
21     final float[] pB = (float[]) B.getPixels();
22     final float[] pC = (float[]) C.getPixels();
23     final float[] pQ = (float[]) Q.getPixels();
24     for (int i = 0; i < M * N; i++) {
25         float a = pA[i], b = pB[i], c = pC[i];
26         float det = a * b - c * c; //  $\det(\bar{\mathbf{M}})$ 
27         float trace = a + b;       //  $\text{trace}(\bar{\mathbf{M}})$ 
28         pQ[i] = det - alpha * (trace * trace);

```

⁵ Package `imagingbook.lib.image`.

```
29     }
30     return Q;
31 }
```

7.3.2 Step 2: Selecting “Good” Corner Points

The result of the first stage of Alg. 7.1 is the corner response function $Q(u, v)$, which in our implementation is stored as a floating-point image (`FloatProcessor`). In the second stage, the dominant corner points are selected from Q . For this we need (a) an object type to describe the corners and (b) a flexible container, in which to store these objects. In this case, the container should be a dynamic data structure since the number of objects to be stored is not known beforehand.

The `Corner` class

Next we define a new class `Corner`⁶ to represent individual corner points $c = \langle x, y, q \rangle$ and a single constructor (in line 35) with `float` parameters x, y for the position and corner strength q :

```
32 public class Corner implements Comparable<Corner> {
33     final float x, y, q;
34
35     public Corner (float x, float y, float q) {
36         this.x = x;
37         this.y = y;
38         this.q = q;
39     }
40
41     public int compareTo (Corner c2) {
42         if (this.q > c2.q) return -1;
43         if (this.q < c2.q) return 1;
44         else return 0;
45     }
46     ...
47 }
```

The class `Corner` implements Java’s `Comparable` interface, such that objects of type `Corner` can be compared with each other and thereby sorted into an ordered sequence. The `compareTo()` method required by the `Comparable` interface is defined (in line 41) to sort corners by descending `q` values.

Choosing a suitable container

In Alg. 7.1, we used the notion of a *sequence* or *lists* to organize and manipulate the collections of potential corner points generated at various stages. One solution would be to utilize *arrays*, but since the size of arrays must be declared before they are used, we would have to allocate memory for extremely large arrays in order to store all the possible corner points that might be identified. Instead, we make use of the `ArrayList` class, which is one of many dynamic data structures conveniently provided by Java’s *Collections Framework*.⁷

⁶ Package `imagingbook.pub.corners`.

⁷ Package `java.util`.

The `collectCorners()` method

The method `collectCorners()` outlined here selects the dominant corner points from the corner response function $Q(u, v)$. The parameter *border* specifies the width of the image's border, within which corner points should be ignored.

```

48 List<Corner> collectCorners(FloatProcessor Q, float tH, int
        border) {
49     List<Corner> C = new ArrayList<Corner>();
50     for (int v = border; v < N - border; v++) {
51         for (int u = border; u < M - border; u++) {
52             float q = Q.getf(u, v);
53             if (q > tH && isLocalMax(Q, u, v)) {
54                 Corner c = new Corner(u, v, q);
55                 C.add(c);
56             }
57         }
58     }
59     return C;
60 }
```

First (in line 49), a new instance of `ArrayList`⁸ is created and assigned to the variable `C`. Then the CRF image `Q` is traversed, and when a potential corner point is located, a new `Corner` is instantiated (line 54) and added to `C` (line 55). The Boolean method `isLocalMax()` (defined in class `HarrisCornerDetector`) determines if the 2D function `Q` is a local maximum at the given position `u, v`:

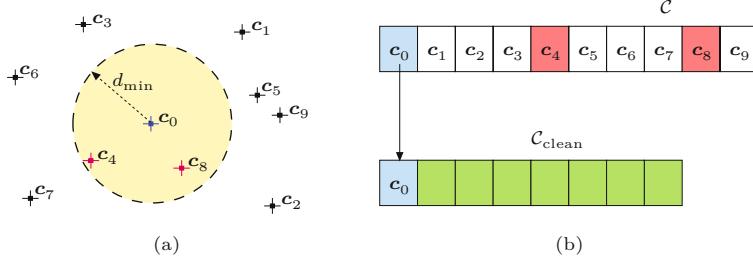
```

61 boolean isLocalMax (FloatProcessor Q, int u, int v) {
62     if (u <= 0 || u >= M - 1 || v <= 0 || v >= N - 1) {
63         return false;
64     }
65     else {
66         float[] q = (float[]) Q.getPixels();
67         int i0 = (v - 1) * M + u;
68         int i1 = v * M + u;
69         int i2 = (v + 1) * M + u;
70         float q0 = q[i1];
71         return // check 8 neighbors of q0:
72             q0 >= q[i0 - 1] && q0 >= q[i0] && q0 >= q[i0 + 1] &&
73             q0 >= q[i1 - 1] &&
74             q0 >= q[i2 - 1] && q0 >= q[i2] && q0 >= q[i2 + 1] ;
75     }
76 }
```

7.3.3 Step 3: Cleaning up

The final step is to remove the weakest corners in a limited area where the size of this area is specified by the radius d_{\min} (Alg. 7.1, lines 29–38). This process is outlined in Fig. 7.3 and implemented by the following method `cleanupCorners()`.

⁸ The specification `ArrayList<Corner>` indicates that the list `C` may only contain objects of type `Corner`.



```

77 List<Corner> cleanupCorners(List<Corner> C, double dmin) {
78     double dmin2 = dmin * dmin;
79     // sort corners by descending q-value:
80     Collections.sort(C);
81     // we use an array of corners for efficiency reasons:
82     Corner[] Ca = C.toArray(new Corner[C.size()]);
83     List<Corner> Cclean = new ArrayList<Corner>(C.size());
84     for (int i = 0; i < Ca.length; i++) {
85         Corner c0 = Ca[i];      // get next strongest corner
86         if (c0 != null) {
87             Cclean.add(c0);
88             // delete all remaining corners cj too close to c0:
89             for (int j = i + 1; j < Ca.length; j++) {
90                 Corner cj = Ca[j];
91                 if (cj != null && c0.dist2(cj) < dmin2)
92                     Ca[j] = null;    //delete corner cj from Ca
93             }
94         }
95     }
96     return Cclean;
97 }
```

Initially (in line 80) the corner list C is sorted by decreasing corner strength q by calling the static method `sort()`.⁹ The sorted sequence is then converted to an array (line 82) which is traversed from start to end (line 84–95). For each selected corner (c_0), all subsequent corners (c_j) with a distance d_{\min} are deleted from the sequence (line 92). The “surviving” corners are then transferred to the final corner sequence C_{clean} .

Note that the call `c0.dist2(cj)` in line 91 returns the *squared* Euclidean distance between the corner points c_0 and c_j , that is, the quantity $d^2 = (x_0 - x_j)^2 + (y_0 - y_j)^2$. Since the square of the distance suffices for the comparison, we do not need to compute the actual distance, and consequently we avoid calling the expensive square root function. This is a common trick when comparing distances.

7.3.4 Summary

Most of the implementation steps we have just described are initiated through calls from the method `findCorners()` in class `HarrisCornerDetector`:

```
98 public List<Corner> findCorners() {
```

7.3 IMPLEMENTATION

Fig. 7.3

Selecting the strongest corners within a given spatial distance. (a) Sample corner positions in the 2D plane. (b) The original list of corners (C) is sorted by “corner strength” (q) in descending order; that is, c_0 is the strongest corner. First, corner c_0 is added to a new list C_{clean} , while the weaker corners c_1, c_2, \dots are treated similarly until no more elements remain in C . None of the corners in the resulting list C_{clean} is closer to another corner than d_{\min} .

⁹ Defined in class `java.util.Collections`.

```

99     FloatProcessor Q = makeCrf((float)params.alpha);
100    List<Corner> corners =
101        collectCorners(Q, (float)params.tH, params.border);
102    if (params.doCleanUp) {
103        corners = cleanupCorners(corners, params.dmin);
104    }
105    return corners;
106 }
```

An example of how to use the class `HarrisCornerDetector` is shown by the associated ImageJ plugin `Find_Corners` whose `run()` consists of only a few lines of code. This method simply creates a new object of the class `HarrisCornerDetector`, calls the `findCorners()` method, and finally displays the results in a new image (R):

```

107 public class Find_Corners implements PlugInFilter {
108
109     public void run(ImageProcessor ip) {
110         HarrisCornerDetector cd = new HarrisCornerDetector(ip);
111         List<Corner> corners = cd.findCorners();
112         ColorProcessor R = ip.convertToColorProcessor();
113         drawCorners(R, corners);
114         (new ImagePlus("Result", R)).show();
115     }
116
117     void drawCorners(ImageProcessor ip,
118                      List<Corner> corners) {
119         ip.setColor(cornerColor);
120         for (Corner c : corners) {
121             drawCorner(ip, c);
122         }
123     }
124
125     void drawCorner(ImageProcessor ip, Corner c) {
126         int size = cornerSize;
127         int x = Math.round(c.getX());
128         int y = Math.round(c.getY());
129         ip.drawLine(x - size, y, x + size, y);
130         ip.drawLine(x, y - size, x, y + size);
131     }
132 }
```

For completeness, the definition of the `drawCorners()` method has been included here; the complete source code can be found online. Again, when writing this code, the focus is on understandability and not necessarily speed and memory usage. Many elements of the code can be optimized with relatively little effort (perhaps as an exercise?) if efficiency becomes important.

7.4 Exercises

Exercise 7.1. Adapt the `draw()` method in the class `Corner` (see p. 155) so that the strength (*q*-value) of the corner points can also be visualized. This could be done, for example, by manipulating

the size, color, or intensity of the markers drawn in relation to the strength of the corner.

7.4 EXERCISES

Exercise 7.2. Conduct a series of experiments to determine how image contrast affects the performance of the Harris detector, and then develop an idea for how you might automatically determine the parameter t_H depending on image content.

Exercise 7.3. Explore how rotation and distortion of the image affect the performance of the Harris corner detector. Based on your experiments, is the operator truly isotropic?

Exercise 7.4. Determine how image noise affects the performance of the Harris detector in terms of the positional accuracy of the detected corners and the omission of actual corners. Remark: ImageJ's menu command **Process ▷ Noise ▷ Add Specified Noise...** can be used to easily add certain types of random noise to a given image.

Finding Simple Curves: The Hough Transform

In Chapter 6 we demonstrated how to use appropriately designed filters to detect edges in images. These filters compute both the edge strength and orientation at every position in the image. In the following sections, we explain how to decide (e.g., by using a threshold operation on the edge strength) if a curve is actually present at a given image location. The result of this process is generally represented as a binary *edge map*. Edge maps are considered preliminary results, since with an edge filter's limited ("myopic") view it is not possible to accurately ascertain if a point belongs to a true edge. Edge maps created using simple threshold operations contain many edge points that do not belong to true edges (false positives), and, on the other hand, many edge points are not detected and hence are missing from the map (false negatives).

8.1 Salient Image Structures

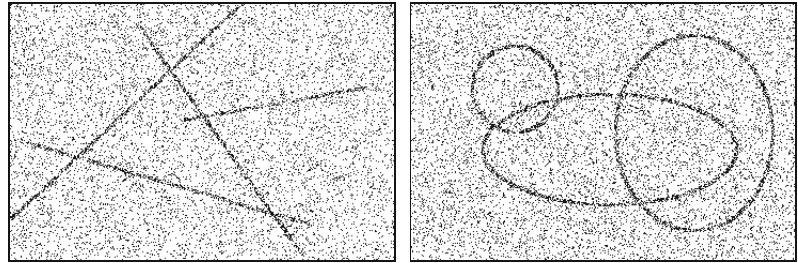
An intuitive approach to locating large image structures is to first select an arbitrary edge point, systematically examine its neighboring pixels and add them if they belong to the object's contour, and repeat. In principle, such an approach could be applied to either a continuous edge map consisting of edge strengths and orientations or a simple binary *edge map*. Unfortunately, with either input, such an approach is likely to fail due to image noise and ambiguities that arise when trying to follow the contours. Additional constraints and information about the type of object sought are needed in order to handle pixel-level problems such as branching, as well as interruptions. This type of local sequential *contour tracing* makes for an interesting optimization problem [128] (see also Sec. 10.2).

A completely different approach is to search for globally apparent structures that consist of certain simple shape features. As an example, Fig. 8.1 shows that certain structures are readily apparent to the human visual system, even when they overlap in noisy images. The biological basis for why the human visual system spontaneously

8 FINDING SIMPLE CURVES: THE HOUGH TRANSFORM

Fig. 8.1

The human visual system is capable of instantly recognizing prominent image structures even under difficult conditions.



recognizes four lines or three ellipses in Fig. 8.1 instead of a larger number of disjoint segments and arcs is not completely known. At the cognitive level, theories such as “Gestalt” grouping have been proposed to address this behavior. The next sections explore one technique, the Hough transform, that provides an algorithmic solution to this problem.

8.2 The Hough Transform

The method from Paul Hough—originally published as a US Patent [111] and often referred to as the “Hough transform” (HT)—is a general approach to localizing any shape that can be defined parametrically within a distribution of points [64, 117]. For example, many geometrical shapes, such as lines, circles, and ellipses, can be readily described using simple equations with only a few parameters. Since simple geometric forms often occur as part of man-made objects, they are especially useful features for analysis of these types of images (Fig. 8.2).

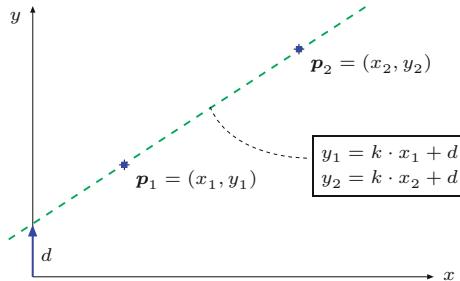
The Hough transform is perhaps most often used for detecting straight line segments in edge maps. A line segment in 2D can be described with two real-valued parameters using the classic slope-intercept form

$$y = k \cdot x + d, \quad (8.1)$$

Fig. 8.2

Simple geometrical forms such as sections of lines, circles, and ellipses are often found in man-made objects.





8.2 THE HOUGH TRANSFORM

Fig. 8.3

Two points, p_1 and p_2 , lie on the same line when $y_1 = kx_1 + d$ and $y_2 = kx_2 + d$ for a particular pair of parameters k and d .

where k is the slope and d the intercept—that is, the height at which the line would intercept the y axis (Fig. 8.3). A line segment that passes through two given edge points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ must satisfy the conditions

$$y_1 = k \cdot x_1 + d \quad \text{and} \quad y_2 = k \cdot x_2 + d, \quad (8.2)$$

for $k, d \in \mathbb{R}$. The goal is to find values of k and d such that as many edge points as possible lie on the line they describe; in other words, the line that fits the most edge points. But how can you determine the number of edge points that lie on a given line segment? One possibility is to exhaustively “draw” every possible line segment into the image while counting the number of points that lie exactly on each of these. Even though the discrete nature of pixel images (with only a finite number of different lines) makes this approach possible in theory, generating such a large number of lines is infeasible in practice.

8.2.1 Parameter Space

The Hough transform approaches the problem from another direction. It examines all the possible line segments that run through a single given point in the image. Every line $L_j = \langle k_j, d_j \rangle$ that runs through a point $p_0 = (x_0, y_0)$ must satisfy the condition

$$L_j : y_0 = k_j x_0 + d_j \quad (8.3)$$

for suitable values k_j, d_j . Equation 8.3 is underdetermined and the possible solutions for k_j, d_j correspond to an infinite set of lines passing through the given point p_0 (Fig. 8.4). Note that for a given k_j , the solution for d_j in Eqn. (8.3) is

$$d_j = -x_0 \cdot k_j + y_0, \quad (8.4)$$

which is another equation for a line, where now k_j, d_j are the *variables* and x_0, y_0 are the constant *parameters* of the equation. The solution set $\{(k_j, d_j)\}$ of Eqn. (8.4) describes the parameters of all possible lines L_j passing through the image point $p_0 = (x_0, y_0)$.

For an *arbitrary* image point $p_i = (x_i, y_i)$, Eqn. (8.4) describes the line

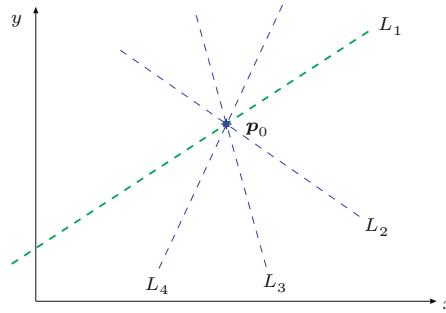
$$M_i : d = -x_i \cdot k + y_i \quad (8.5)$$

with the parameters $-x_i, y_i$ in the so-called *parameter* or *Hough space*, spanned by the coordinates k, d . The relationship between

8 FINDING SIMPLE CURVES: THE HOUGH TRANSFORM

Fig. 8.4

A set of lines passing through an image point. For all possible lines L_j passing through the point $\mathbf{p}_0 = (x_0, y_0)$, the equation $y_0 = k_j x_0 + d_j$ holds for appropriate values of the parameters k_j, d_j .



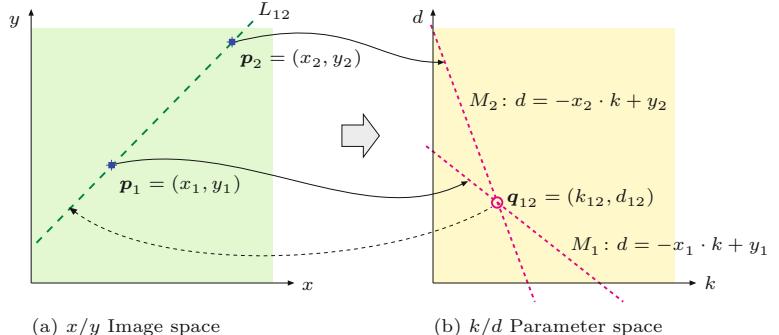
(x, y) *image space* and (k, d) *parameter space* can be summarized as follows:

Image Space (x, y)		Parameter Space (k, d)	
Point	$\mathbf{p}_i = (x_i, y_i)$	\longleftrightarrow	$M_i: d = -x_i \cdot k + y_i$ Line
Line	$L_j: y = k_j \cdot x + d_j$	\longleftrightarrow	$\mathbf{q}_j = (k_j, d_j)$ Point

Each image point \mathbf{p}_i and its associated line bundle correspond to exactly one line M_i in parameter space. Therefore we are interested in those places in the parameter space where lines *intersect*. The example in Fig. 8.5 illustrates how the lines M_1 and M_2 intersect at the position $\mathbf{q}_{12} = (k_{12}, d_{12})$ in the parameter space, which means (k_{12}, d_{12}) are the parameters of the line in the image space that runs through both image points \mathbf{p}_1 and \mathbf{p}_2 . The more lines M_i that intersect at a single point in the parameter space, the more image space points lie on the corresponding line in the image! In general, we can state:

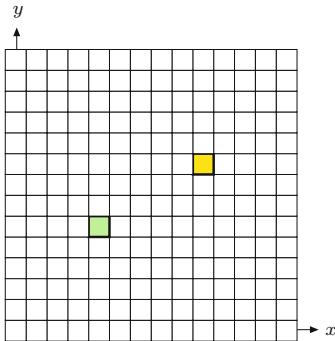
If N lines intersect at position (k', d') in *parameter space*, then N image points lie on the corresponding line $y = k'x + d'$ in *image space*.

Fig. 8.5
Relationship between image space and parameter space. The parameter values for all possible lines passing through the image point $\mathbf{p}_i = (x_i, y_i)$ in image space (a) lie on a single line M_i in parameter space (b). This means that each point $\mathbf{q}_j = (k_j, d_j)$ in parameter space corresponds to a single line L_j in image space. The intersection of the two lines M_1, M_2 at the point $\mathbf{q}_{12} = (k_{12}, d_{12})$ in parameter space indicates that a line L_{12} through the two points k_{12} and d_{12} exists in the image space.

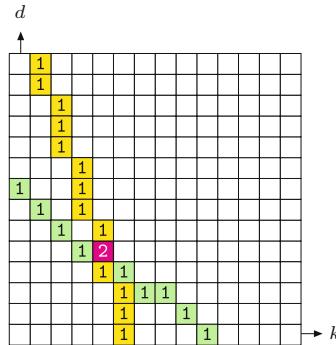


8.2.2 Accumulator Map

Finding the dominant lines in the image can now be reformulated as finding all the locations in parameter space where a significant number of lines intersect. This is basically the goal of the HT. In order



(a) Image space



(b) Accumulator map

8.2 THE HOUGH TRANSFORM

Fig. 8.6

The accumulator map is a discrete representation of the parameter space (k, d). For each image point found (a), a discrete line in the parameter space (b) is drawn. This operation is performed *additively* so that the values of the array through which the line passes are incremented by 1. The value at each cell of the accumulator array is the number of parameter space lines that intersect it (in this case 2).

to compute the HT, we must first decide on a discrete representation of the continuous parameter space by selecting an appropriate step size for the k and d axes. Once we have selected step sizes for the coordinates, we can represent the space naturally using a 2D array. Since the array will be used to keep track of the number of times parameter space lines intersect, it is called an “accumulator” array. Each parameter space line is painted into the accumulator array and the cells through which it passes are incremented, so that ultimately each cell accumulates the total number of lines that intersect at that cell (Fig. 8.6).

8.2.3 A Better Line Representation

The line representation in Eqn. (8.1) is not used in practice because for vertical lines the slope is infinite, that is, $k = \infty$. A more practical representation is the so-called *Hessian normal form* (HNF)¹ for representing lines,

$$x \cdot \cos(\theta) + y \cdot \sin(\theta) = r, \quad (8.6)$$

which does not exhibit such singularities and also provides a natural linear quantization for its parameters, the angle θ and the radius r (Fig. 8.7).

With the HNF representation, the parameter space is defined by the coordinates θ, r , and a point $\mathbf{p} = (x, y)$ in image space corresponds to the relation

$$r(\theta) = x \cdot \cos(\theta) + y \cdot \sin(\theta), \quad (8.7)$$

for angles in the range $0 \leq \theta < \pi$ (see Fig. 8.8). Thus, for a given image point \mathbf{p} , the associated radius r is simply a function of the angle θ . If we use the center of the image (of size $M \times N$),

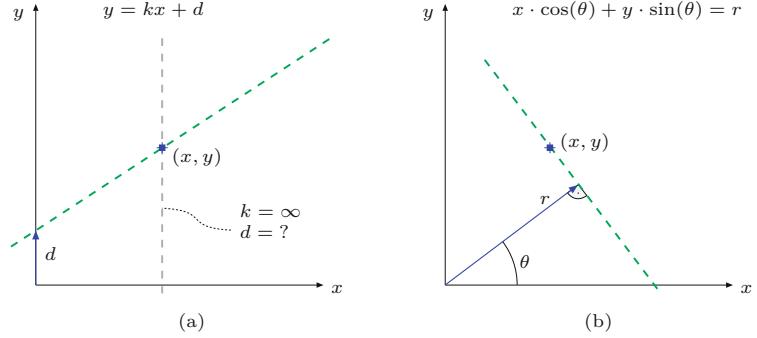
$$\mathbf{x}_r = \begin{pmatrix} x_r \\ y_r \end{pmatrix} = \frac{1}{2} \cdot \begin{pmatrix} M \\ N \end{pmatrix}, \quad (8.8)$$

¹ The Hessian normal form is a normalized version of the general (“algebraic”) line equation $Ax + By + C = 0$, with $A = \cos(\theta)$, $B = \sin(\theta)$, and $C = -r$ (see, e.g., [35, p. 194]).

8 FINDING SIMPLE CURVES: THE HOUGH TRANSFORM

Fig. 8.7

Representation of lines in 2D. In the common k, d representation (a), vertical lines pose a problem because $k = \infty$. The Hessian normal form (b) avoids this problem by representing a line by its angle θ and distance r from the origin.



as the reference point for the x/y image coordinates, then it is possible to limit the range of the radius to half the diagonal of the image, that is,

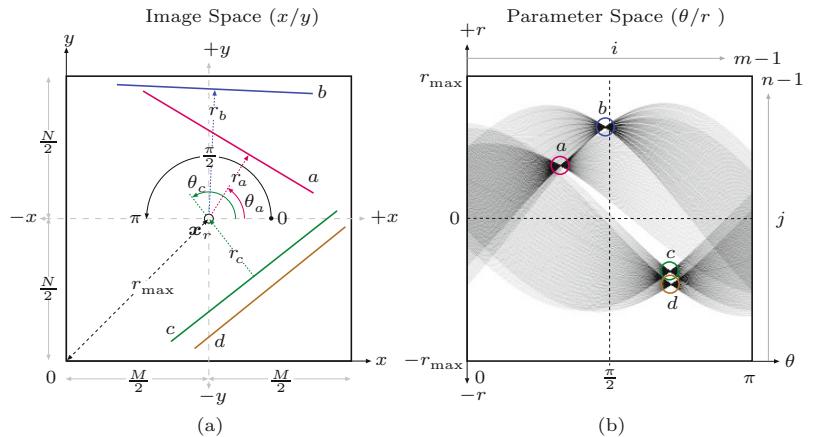
$$-r_{\max} \leq r(\theta) \leq r_{\max}, \quad \text{with} \quad r_{\max} = \frac{1}{2}\sqrt{M^2 + N^2}. \quad (8.9)$$

We can see that the function $r(\theta)$ in Eqn. (8.7) is the sum of a cosine and a sine function on θ , each being weighted by the x and y coordinates of the image point (assumed to be constant for the moment). The result is again a sinusoidal function whose magnitude and phase depend only on the weights (coefficients) x, y . Thus, with the Hessian parameterization θ/r , an image point (x, y) does not create a straight line in the accumulator map $A(i, j)$ but a unique sinusoidal curve, as shown in Fig. 8.8. Again, each image point adds a curve to the accumulator and each resulting cluster point corresponds to a dominant line in the image with a proportional number of points on it.²

Fig. 8.8

Image space and parameter space using the HNF representation. The image (a) of size $M \times N$ contains four straight lines L_a, \dots, L_d . Each point on an image line creates a sinusoidal curve in the θ/r parameter space (b) and the corresponding line parameters are indicated by the clearly visible cluster points in the accumulator map. The reference point \mathbf{x}_r for the x/y coordinates lies at the center of the image. The line angles θ_i are in the range $[0, \pi)$ and the associated radii r_i are in $[-r_{\max}, r_{\max}]$ (the length r_{\max} is half of the image diagonal). For example, the angle θ_a of line L_a is approximately $\pi/3$, with the (positive) radius $r_a \approx 0.4 r_{\max}$.

Note that, with this parameterization, line L_c has the angle $\theta_c \approx 2\pi/3$ and the negative radius $r_c \approx -0.4 r_{\max}$.



² Note that, in Fig. 8.8(a), the positive direction of the y -coordinate runs *upwards* (unlike our usual convention for image coordinates) to stay in line with the previous illustrations (and high school geometry). In practice, the consequences are minor: only the rotation angle runs in the opposite direction and thus the accumulator image in Fig. 8.8(b) was mirrored horizontally for proper display.

8.3 Hough Algorithm

8.3 HOUGH ALGORITHM

The fundamental Hough algorithm using the HNF line representation (Eqn. (8.6)) is given in Alg. 8.1. Starting with a binary image $I(u, v)$ where the edge pixels have been assigned a value of 1, the first stage creates a 2D accumulator array and then iterates over the image to fill it. The resulting increments are

$$d_\theta = \pi/m \quad \text{and} \quad d_r = \sqrt{M^2 + N^2}/n \quad (8.10)$$

for the angle θ and the radius r , respectively. The discrete indices of the accumulators cells are denoted i and j , with $j_0 = n \div 2$ as the center index (for $r = 0$).

For each relevant image point (u, v) , a sinusoidal curve is added to the accumulator map by stepping over the discrete angles $\theta_i = \theta_0, \dots, \theta_{m-1}$, calculating the corresponding radius³

$$r(\theta_i) = (u - x_r) \cdot \cos(\theta_i) + (v - y_r) \cdot \sin(\theta_i) \quad (8.11)$$

(see Eqn. (8.7)) and its discrete index

$$j = j_0 + \text{round} \left(\frac{r(\theta_i)}{d_r} \right), \quad (8.12)$$

and subsequently incrementing the accumulator cell $A(i, j)$ by one (see Alg. 8.1, lines 10–17). The line parameters θ_i and r_j for a given accumulator position (i, j) can be calculated as

$$\theta_i = i \cdot d_\theta \quad \text{and} \quad r_j = (j - j_0) \cdot d_r. \quad (8.13)$$

In the second stage of Alg. 8.1, the accumulator array is searched for local peaks above a given minimum Values a_{\min} . For each detected peak, a line object is created of the form

$$L_k = \langle \theta_k, r_k, a_k \rangle, \quad (8.14)$$

consisting of the angle θ_k , the radius r_k (relative to the reference point x_r), and the corresponding accumulator value a_k . The resulting sequence of lines $\mathcal{L} = (L_1, L_2, \dots)$ is then sorted by descending a_k and returned.

[Figure 8.9](#) shows the result of applying the Hough transform to a very noisy binary image, which obviously contains four straight lines. They appear clearly as cluster points in the corresponding accumulator map in [Fig. 8.9 \(b\)](#). [Figure 8.9 \(c\)](#) shows the reconstruction of these lines from the extracted parameters. In this example, the resolution of the discrete parameter space is set to 256×256 .⁴

³ The frequent (and expensive) calculation of $\cos(\theta_i)$ and $\sin(\theta_i)$ in Eqn. (8.11) and Alg. 8.1 (line 15) can be easily avoided by initially tabulating the function values for all m possible angles $\theta_i = \theta_0, \dots, \theta_{m-1}$, which should yield a significant performance gain.

⁴ Note that *drawing* a straight line given in Hessian normal form is not really a trivial task (see Exercises 8.1–8.2 for details).

8 FINDING SIMPLE CURVES: THE HOUGH TRANSFORM

Alg. 8.1

Hough algorithm for detecting straight lines. The algorithm returns a sorted list of straight lines of the form $L_k = \langle \theta_k, r_k, a_k \rangle$ for the binary input image I of size $M \times N$. The resolution of the discrete Hough accumulator map (and thus the step size for the angle and radius) is specified by parameters m and n , respectively. a_{\min} defines the minimum accumulator value, that is, the minimum number of image point on any detected line. The function `IsLocalMax()` used in line 20 is the same as in Alg. 7.1 (see p. 151).

```

1: HoughTransformLines( $I, m, n, a_{\min}$ )
   Input:  $I$ , a binary image of size  $M \times N$ ;  $m$ , angular accumulator steps;  $n$ , radial accumulator steps;  $a_{\min}$ , minimum accumulator count per line. Returns a sorted sequence  $\mathcal{L} = (L_1, L_2, \dots)$  of the most dominant lines found.
2:  $(M, N) \leftarrow \text{Size}(I)$ 
3:  $(x_r, y_r) \leftarrow \frac{1}{2} \cdot (M, N)$             $\triangleright$  reference point  $x_r$  (image center)
4:  $d_\theta \leftarrow \pi/m$                           $\triangleright$  angular step size
5:  $d_r \leftarrow \sqrt{M^2 + N^2}/n$               $\triangleright$  radial step size
6:  $j_0 \leftarrow n \div 2$                           $\triangleright$  map index for  $r = 0$ 

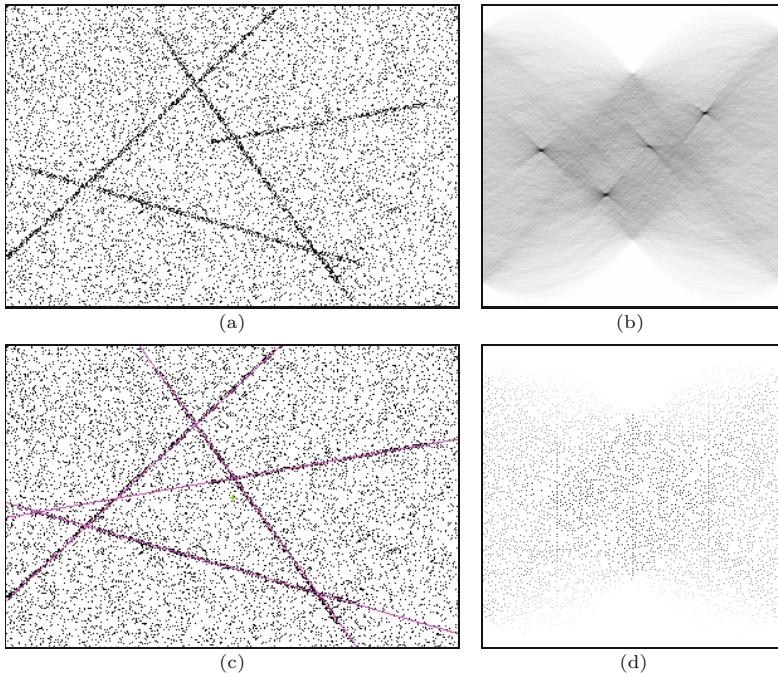
Step 1 – set up and fill the Hough accumulator:
7: Create map  $A: [0, m-1] \times [0, n-1] \mapsto \mathbb{Z}$             $\triangleright$  accumulator
8: for all accumulator cells  $(i, j)$  do
9:    $A(i, j) \leftarrow 0$                                  $\triangleright$  initialize accumulator

10: for all  $(u, v) \in M \times N$  do                       $\triangleright$  scan the image
11:   if  $I(u, v) > 0$  then                           $\triangleright I(u, v)$  is a foreground pixel
12:      $(x, y) \leftarrow (u - x_r, v - y_r)$            $\triangleright$  shift to reference
13:     for  $i \leftarrow 0, \dots, m-1$  do             $\triangleright$  angular coordinate  $i$ 
14:        $\theta \leftarrow d_\theta \cdot i$                    $\triangleright$  angle,  $0 \leq \theta < \pi$ 
15:        $r \leftarrow x \cdot \cos(\theta) + y \cdot \sin(\theta)$      $\triangleright$  see Eqn. 8.7
16:        $j \leftarrow j_0 + \text{round}(r/d_r)$            $\triangleright$  radial coordinate  $j$ 
17:        $A(i, j) \leftarrow A(i, j) + 1$                  $\triangleright$  increment  $A(i, j)$ 

Step 2 – extract the most dominant lines:
18:  $\mathcal{L} \leftarrow ()$                                  $\triangleright$  start with empty sequence of lines
19: for all accumulator cells  $(i, j)$  do           $\triangleright$  collect local maxima
20:   if  $(A(i, j) \geq a_{\min}) \wedge \text{IsLocalMax}(A, i, j)$  then
21:      $\theta \leftarrow i \cdot d_\theta$                        $\triangleright$  angle  $\theta$ 
22:      $r \leftarrow (j - j_0) \cdot d_r$                  $\triangleright$  radius  $r$ 
23:      $a \leftarrow A(i, j)$                            $\triangleright$  accumulated value  $a$ 
24:      $L \leftarrow \langle \theta, r, a \rangle$              $\triangleright$  create a new line  $L$ 
25:      $\mathcal{L} \leftarrow \mathcal{L} \cup (L)$              $\triangleright$  add line  $L$  to sequence  $\mathcal{L}$ 
26:    $\text{Sort}(\mathcal{L})$                              $\triangleright$  sort  $\mathcal{L}$  by descending accumulator count  $a$ 
27:   return  $\mathcal{L}$ 
```

8.3.1 Processing the Accumulator Array

The reliable detection and precise localization of peaks in the accumulator map $A(i, j)$ is not a trivial problem. As can readily be seen in Fig. 8.9(b), even in the case where the lines in the image are geometrically “straight”, the parameter space curves associated with them do not intersect at *exactly* one point in the accumulator array but rather their intersection points are distributed within a small area. This is primarily caused by the rounding errors introduced by the discrete coordinate grid used for the accumulator array. Since the maximum points are really maximum *areas* in the accumulator array, simply traversing the array and returning the positions of its largest values is not sufficient. Since this is a critical step in the algorithm, we examine two different approaches below (see Fig. 8.10).



8.3 HOUGH ALGORITHM

Fig. 8.9 Hough transform for straight lines. The dimensions of the original image (a) are 360×240 pixels, so the maximal radius (measured from the image center) is $r_{\max} \approx 216$. For the parameter space (b), a step size of 256 is used for both the angle $\theta = 0, \dots, \pi$ (horizontal axis) and the radius $r = -r_{\max}, \dots, r_{\max}$ (vertical axis). The four (dark) clusters in (b) surround the maximum values in the accumulator array, and their parameters correspond to the four lines in the original image. Intensities are shown inverted in all images to improve legibility.

Approach A: Thresholding

First the accumulator is thresholded to the value of t_a by setting all accumulator values $A(i, j) < t_a$ to 0. The resulting scattering of points, or point clouds, are first coalesced into regions (Fig. 8.10(b)) using a technique such as a morphological *closing* operation (see Sec. 9.3.2). Next the remaining regions must be localized, for instance using the region-finding technique from Sec. 10.1, and then each region's centroid (see Sec. 10.5) can be utilized as the (noninteger) coordinates for the potential image space line. Often the sum of the accumulator's values within a region is used as a measure of the strength (number of image points) of the line it represents.

Approach B: Nonmaximum suppression

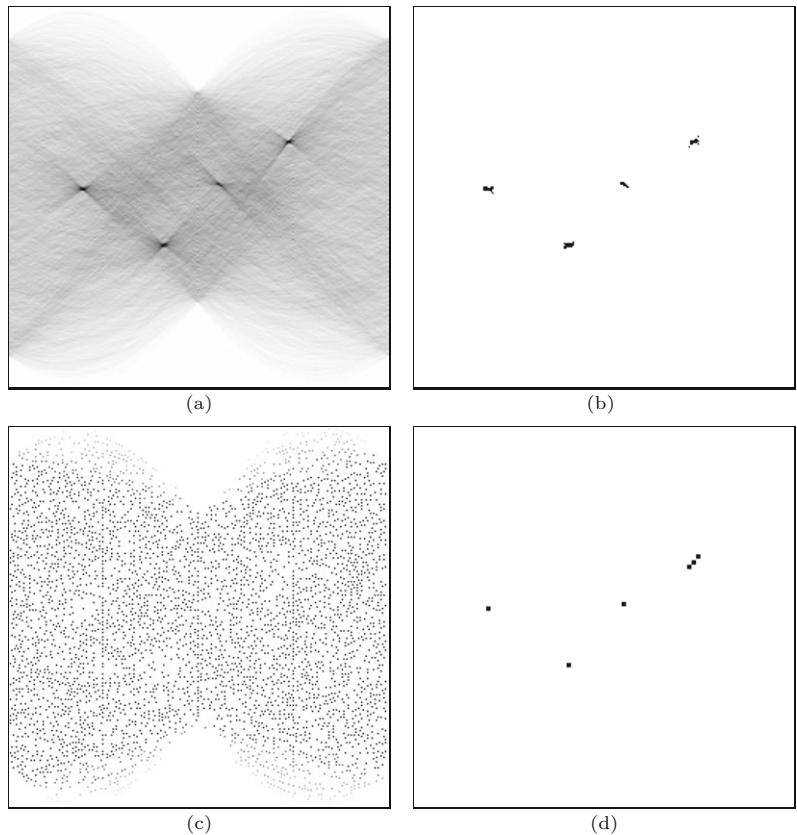
In this method, local maxima in the accumulator array are found by suppressing nonmaximal values.⁵ This is carried out by determining for every accumulator cell $A(i, j)$ whether the value is higher than the value of all of its neighboring cells. If this is the case, then the value remains the same; otherwise it is set to 0 (Fig. 8.10(c)). The (integer) coordinates of the remaining peaks are potential line parameters, and their respective heights correlate with the strength of the image space line they represent. This method can be used in conjunction with a threshold operation to reduce the number of candidate points that must be considered. The result for Fig. 8.9(a) is shown in Fig. 8.10(d).

⁵ Nonmaximum suppression is also used in Sec. 7.2.3 for isolating corner points.

8 FINDING SIMPLE CURVES: THE HOUGH TRANSFORM

Fig. 8.10

Finding local maximum values in the accumulator array. Original distribution of the values in the Hough accumulator (a). **Variant A:** Threshold operation using 50% of the maximum value (b). The remaining regions represent the four dominant lines in the image, and the coordinates of their centroids are a good approximation to the line parameters. **Variant B:** Using non-maximum suppression results in a large number of local maxima (c) that must then be reduced using a threshold operation (d).



Mind the vertical lines!

Special consideration should be given to *vertical* lines (once more!) when processing the contents of the accumulator map. The parameter pairs for these lines lie near $\theta = 0$ and $\theta = \pi$ at the left and right borders, respectively, of the accumulator map (see Fig. 8.8(b)). Thus, to locate peak clusters in this part of the parameter space, the horizontal coordinate along the θ axis must be treated circularly, that is, modulo m . However, as can be seen clearly in Fig. 8.8(b), the sinusoidal traces in the parameter space do not continue smoothly at the transition $\theta = \pi \rightarrow 0$, but are vertically mirrored! Evaluating such neighborhoods near the borders of the parameter space thus requires special treatment of the vertical (r) accumulator coordinate.

8.3.2 Hough Transform Extensions

So far, we have presented the Hough transform only in its most basic formulation. The following is a list of some of the more common methods of improving and refining the method.

Modified accumulation

The purpose of the accumulator map is to locate the intersections of multiple 2D curves. Due to the discrete nature of the image and accumulator coordinates, rounding errors usually cause the parameter curves not to intersect in a single accumulator cell, even when the

associated image lines are exactly straight. A common remedy is, for a given angle $\theta = i_\theta \cdot \Delta_\theta$ (Alg. 8.1), to increment not only the *main* accumulator cell $A(i, j)$ but also the *neighboring* cells $A(i, j-1)$ and $A(i, j+1)$, possibly with different weights. This makes the Hough transform more tolerant against inaccurate point coordinates and rounding errors.

8.3 HOUGH ALGORITHM

Considering edge strength and orientation

Until now, the raw data for the Hough transform was typically an edge map that was interpreted as a binary image with ones at potential edge points. Yet edge maps contain additional information, such as the edge strength $E(u, v)$ and local edge orientation $\Phi(u, v)$ (see Sec. 6.3), which can be used to improve the results of the HT.

The *edge strength* $E(u, v)$ is especially easy to take into consideration. Instead of incrementing visited accumulator cells by 1, add the strength of the respective edge, that is,

$$A(i, j) \leftarrow A(i, j) + E(u, v). \quad (8.15)$$

In this way, strong edge points will contribute more to the accumulated values than weak ones (see also Exercise 8.6).

The local *edge orientation* $\Phi(u, v)$ is also useful for limiting the range of possible orientation angles for the line at (u, v) . The angle $\Phi(u, v)$ can be used to increase the efficiency of the algorithm by reducing the number of accumulator cells to be considered along the θ axis. Since this also reduces the number of irrelevant “votes” in the accumulator, it increases the overall sensitivity of the Hough transform (see, e.g., [125, p. 483]).

Bias compensation

Since the value of a cell in the Hough accumulator represents the number of image points falling on a line, longer lines naturally have higher values than shorter lines. This may seem like an obvious point to make, but consider when the image only contains a small section of a “long” line. For instance, if a line only passes through the corner of an image then the cells representing it in the accumulator array will naturally have lower values than a “shorter” line that lies entirely within the image (Fig. 8.11). It follows then that if we only search the accumulator array for maximal values, it is likely that we will completely miss short line segments. One way to compensate for

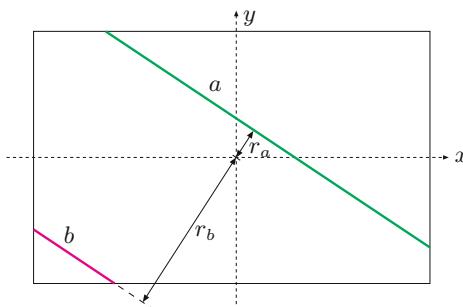


Fig. 8.11
Hough transform bias problem.
When an image represents only a finite section of an object, then those lines nearer the center (smaller r values) will have higher values than those farther away (larger r values). As an example, the maximum value of the accumulator for line a will be higher than that of line b .

this inherent bias is to compute for each accumulator entry $A(i, j)$ the maximum number of image points $A_{\max}(i, j)$ possible for a line with the corresponding parameters and then normalize the result, for example, in the form

$$A(i, j) \leftarrow \frac{A(i, j)}{\max(1, A_{\max}(i, j))}. \quad (8.16)$$

The normalization map $A_{\max}(i, j)$ can be determined analytically (by calculating the intersecting length of each line) or by simulation; for example, by computing the Hough transform of an image with the same dimensions in which all pixels are edge pixels or by using a random image in which the pixels are uniformly distributed.

Line endpoints

Our simple version of the Hough transform determines the parameters of the line in the image but not their endpoints. These could be found in a subsequent step by determining which image points belong to any detected line (e.g., by applying a threshold to the perpendicular distance between the ideal line—defined by its parameters—and the actual image points). An alternative solution is to calculate the extreme point of the line during the computation of the accumulator array. For this, every cell of the accumulator array is supplemented with four addition coordinates to

$$A(i, j) = (a, u_{\min}, v_{\min}, u_{\max}, v_{\max}), \quad (8.17)$$

where component a denotes the original accumulator value and u_{\min} , v_{\min} , u_{\max} , v_{\max} are the coordinates of the line's bounding box. After the additional coordinates are initialized, they are updated simultaneously with the positions along the parameter trace for every image point (u, v) . After completion of the process, the accumulator cell (i, j) contains the bounding box for all image points that contributed it. When finding the maximum values in the second stage, care should be taken so that the merged cells contain the correct endpoints (see also Exercise 8.4).

Hierarchical Hough transform

The accuracy of the results increases with the size of the parameter space used; for example, a step size of 256 along the θ axis is equivalent to searching for lines at every $\frac{\pi}{256} \approx 0.7^\circ$. While increasing the number of accumulator cells provides a finer result, bear in mind that it also increases the computation time and especially the amount of memory required.

Instead of increasing the resolution of the entire parameter space, the idea of the hierarchical HT is to gradually “zoom” in and refine the parameter space. First, the regions containing the most important lines are found using a relatively low-resolution parameter space, and then the parameter spaces of those regions are recursively passed to the HT and examined at a higher resolution. In this way, a relatively exact determination of the parameters can be found using a limited (in comparison) parameter space.

Line intersections

It may be useful in certain applications not to find the lines themselves but their intersections, for example, for precisely locating the corner points of a polygon-shaped object. The Hough transform delivers the parameters of the recovered lines in Hessian normal form (that is, as pairs $L_k = \langle \theta_k, r_k \rangle$). To compute the point of intersection $\mathbf{x}_{12} = (x_{12}, y_{12})^\top$ for two lines $L_1 = \langle \theta_1, r_1 \rangle$ and $L_2 = \langle \theta_2, r_2 \rangle$ we need to solve the system of linear equations

$$\begin{aligned} x_{12} \cdot \cos(\theta_1) + y_{12} \cdot \sin(\theta_1) &= r_1, \\ x_{12} \cdot \cos(\theta_2) + y_{12} \cdot \sin(\theta_2) &= r_2, \end{aligned} \quad (8.18)$$

for the unknowns x_{12}, y_{12} . The solution is

$$\begin{aligned} \begin{pmatrix} x_{12} \\ y_{12} \end{pmatrix} &= \frac{1}{\cos(\theta_1)\sin(\theta_2) - \cos(\theta_2)\sin(\theta_1)} \cdot \begin{pmatrix} r_1 \sin(\theta_2) - r_2 \sin(\theta_1) \\ r_2 \cos(\theta_1) - r_1 \cos(\theta_2) \end{pmatrix} \\ &= \frac{1}{\sin(\theta_2 - \theta_1)} \cdot \begin{pmatrix} r_1 \sin(\theta_2) - r_2 \sin(\theta_1) \\ r_2 \cos(\theta_1) - r_1 \cos(\theta_2) \end{pmatrix}, \end{aligned} \quad (8.19)$$

for $\sin(\theta_2 - \theta_1) \neq 0$. Obviously \mathbf{x}_0 is undefined (no intersection point exists) if the lines L_1, L_2 are parallel to each other (i.e., if $\theta_1 \equiv \theta_2$).

Figure 8.12 shows an illustrative example using *ARToolkit*⁶ markers. After automatic thresholding (see Ch. 11) the straight line segments along the outer boundary of the largest binary region are analyzed with the Hough transform. Subsequently, the corners of the marker are calculated precisely as the intersection points of the involved line segments.

8.4 Java Implementation

The complete Java source code for the straight line Hough transform is available online in class `HoughTransformLines`.⁷ Detailed usage of this class is shown in the ImageJ plugin `Find_Straight_Lines` (see also Prog. 8.1 for a minimal example).⁸

`HoughTransformLines` (class)

This class is a direct implementation of the Hough transform for straight lines, as outlined in Alg. 8.1. The sin/cos function calls (see Alg. 8.1, line 15) are substituted by precalculated tables for improved efficiency. The class defines the following constructors:

```
HoughTransformLines (ImageProcessor I, Parameters
params)
I denotes the input image, where all pixel values > 0 are
assumed to be relevant (edge) points; params is an instance of
the (inner) class HoughTransformLines.Parameters, which
allows to specify the accumulator size (nAng, nRad) etc.
```

⁶ Used for augmented reality applications, see www.hitl.washington.edu/artoolkit/.

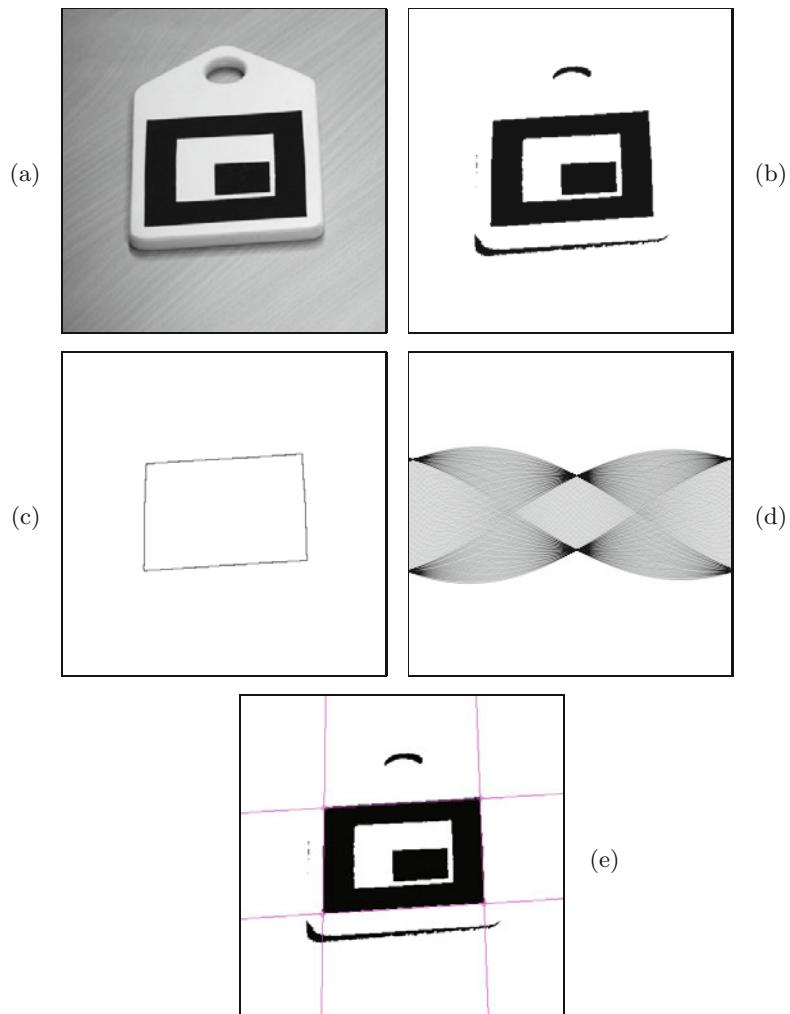
⁷ Package `imagingbook.pub.hough`.

⁸ Note that the current implementation has no bias compensation (see Sec. 8.3.2, Fig. 8.11).

8 FINDING SIMPLE CURVES: THE HOUGH TRANSFORM

Fig. 8.12

Hough transform used for precise calculation of corner points. Original image showing a typical ARToolkit marker (a), result after automatic thresholding (b). The outer contour pixels of the largest binary region (c) are used as input points to the Hough transform. Hough accumulator map (d), detected lines and marked intersection points (e).



```
HoughTransformLines (Point2D[] points, int M, int N,  
Parameters params)
```

In this case the Hough transform is calculated for a sequence of 2D points (`points`); `M`, `N` specify the associated coordinate frame (for calculating the reference point x_r), which is typically the original image size; `params` is a parameter object (as described before).

The most important public methods of the class `ClassHoughTransformLines` are:

```
HoughLine[] getLines (int amin, int maxLines)
```

Returns a sorted sequence of line objects⁹ whose accumulator value is `amin` or greater. The sequence is sorted by accumulator values and contains up to `maxLines` elements

```
int[][] getAccumulator ()
```

Returns a reference to the accumulator map `A` (of size $m \times n$ for angles and radii, respectively).

⁹ Of type `HoughTransformLines.HoughLine`.

```

1 import imagingbook... .HoughTransformLines;
2 import imagingbook... .HoughTransformLines.HoughLine;
3 import imagingbook... .HoughTransformLines.Parameters;
4 ...
5
6 public void run(ImageProcessor ip) {
7     Parameters params = new Parameters();
8     params.nAng = 256;      // = m
9     params.nRad = 256;      // = n
10
11    // compute the Hough Transform:
12    HoughTransformLines ht =
13        new HoughTransformLines(ip, params);
14
15    // retrieve the 5 strongest lines with min. 50 accumulator votes
16    HoughLine[] lines = ht.getLines(50, 5);
17
18    if (lines.length > 0) {
19        IJ.log("Lines found:");
20        for (HoughLine L : lines) {
21            IJ.log(L.toString()); // list the resulting lines
22        }
23    } else
24        IJ.log("No lines found!");
25
26 }

```

8.4 JAVA IMPLEMENTATION

Prog. 8.1

Minimal example for the usage of class `HoughTransformLines` (`run()` method for an `ImageProcessor` plugin of type `PlugInFilter`). First (in lines 7–9) a parameter object is created and configured; `nAng` (= m) and `nRad` (= n) specify the number of discrete angular and radial steps in the Hough accumulator map. In lines 12–13 an instance of `HoughTransformLines` is created for the image `ip`. The accumulator map is calculated in this step. In line 16, `getLines()` is called to retrieve the sequence of the 5 strongest detected lines, with at least 50 image points each. Unless empty, this sequence is subsequently listed.

`int[][] getAccumulatorMax ()`

Returns a copy of accumulator array in which all non-maxima are replaced by zero values.

`FloatProcessor getAccumulatorImage ()`

Returns a floating-point image of the accumulator array, analogous to `getAccumulator()`. Angles θ_i run horizontally, radii r_j vertically.

`FloatProcessor getAccumulatorMaxImage ()`

Returns a floating-point image of the accumulator array with suppressed non-maximum values, analogous to `getAccumulatorMax()`.

`double angleFromIndex (int i)`

Returns the angle $\theta_i \in [0, \pi]$ for the given index i in the range $0, \dots, m-1$.

`double radiusFromIndex (int j)`

Returns the radius $r_j \in [-r_{\max}, r_{\max}]$ for the given index j in the range $0, \dots, n-1$.

`Point2D getReferencePoint ()`

Returns the (fixed) reference point x_r , for this Hough transform instance.

HoughLine (class)

`HoughLine` represents a straight line in Hessian normal form. It is implemented as an inner class of `HoughTransformLines`. It offers no public constructor but the following methods:

```
double getAngle ()
    Returns the angle  $\theta \in [0, \pi)$  of this line.

double getRadius ()
    Returns the radius  $r \in [-r_{\max}, r_{\max}]$  of this line, relative to
    the associated Hough transform's reference point  $x_r$ .

int getCount ()
    Returns the Hough transform's accumulator value (number of
    registered image points) for this line.

Point2D getReferencePoint ()
    Returns the (fixed) reference point  $x_r$  for this line. Note that
    all lines associated with a given Hough transform share the
    same reference point.

double getDistance (Point2D p)
    Returns the Euclidean distance of point p to this line. The
    result may be positive or negative, depending on which side of
    the line p is located.
```

8.5 Hough Transform for Circles and Ellipses

8.5.1 Circles and Arcs

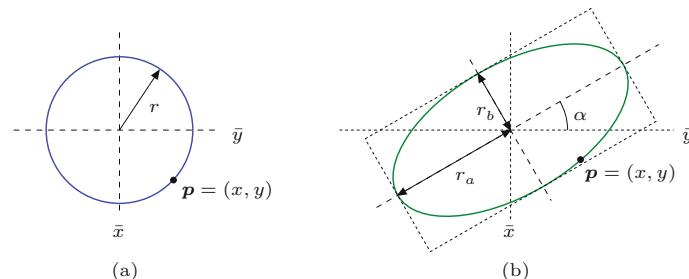
Since lines in 2D have two degrees of freedom, they could be completely specified using two real-valued parameters. In a similar fashion, representing a circle in 2D requires *three* parameters, for example

$$C = \langle \bar{x}, \bar{y}, r \rangle,$$

where \bar{x} , \bar{y} are the coordinates of the center and r is the radius of the circle (Fig. 8.13).

Fig. 8.13

Representation of circles and ellipses in 2D. A circle (a) requires three parameters (e.g., \bar{x}, \bar{y}, r). An arbitrary ellipse (b) takes five parameters (e.g., $\bar{x}, \bar{y}, r_a, r_b, \alpha$).



A point $p = (x, y)$ lies exactly on the circle C if the condition

$$(x - \bar{x})^2 + (y - \bar{y})^2 = r^2 \quad (8.20)$$

holds. Therefore the Hough transform for circles requires a 3D parameter space $A(i, j, k)$ to find the position and radius of circles (and

circular arcs) in an image. Unlike the HT for lines, there does not exist a simple functional dependency between the coordinates in parameter space; so how can we find every parameter combination (\bar{x}, \bar{y}, r) that satisfies Eqn. (8.20) for a given image point (x, y) ? A “brute force” is to exhaustively test all cells of the parameter space to see if the relation in Eqn. (8.20) holds, which is computationally quite expensive, of course.

If we examine Fig. 8.14, we can see that a better idea might be to make use of the fact that the coordinates of the center points also form a circle in Hough space. It is not necessary therefore to search the entire 3D parameter space for each image point. Instead we need only increase the cell values along the edge of the appropriate circle on each r plane of the accumulator array. To do this, we can adapt any of the standard algorithms for generating circles. In this case, the integer math version of the well-known *Bresenham* algorithm [33] is particularly well-suited.

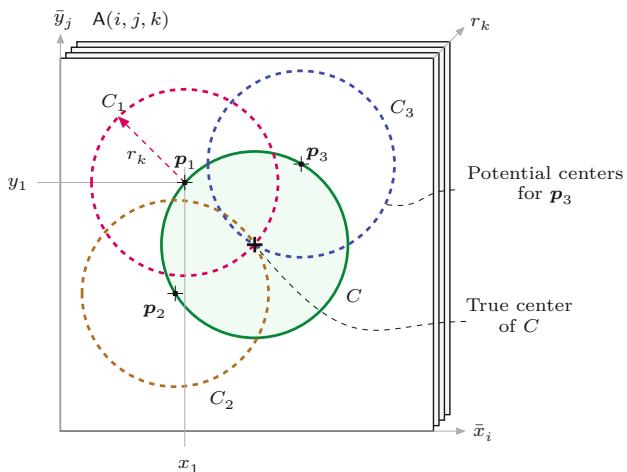


Fig. 8.14
Hough transform for circles. The illustration depicts a single slice of the 3D accumulator array $A(i, j, k)$ at a given circle radius r_k . The center points of all the circles running through a given image point $p_1 = (x_1, y_1)$ form a circle C_1 with a radius of r_k centered around p_1 , just as the center points of the circles that pass through p_2 and p_3 lie on the circles C_2, C_3 . The cells along the edges of the three circles C_1, C_2, C_3 of radius r_k are traversed and their values in the accumulator array incremented. The cell in the accumulator array contains a value of 3 where the circles intersect at the true center of the image circle C .

Figure 8.15 shows the spatial structure of the 3D parameter space for circles. For a given image point $p_m = (u_m, v_m)$, at each plane along the r axis (for $r_k = r_{\min}, \dots, r_{\max}$), a circle centered at (u_m, v_m) with the radius r_k is traversed, ultimately creating a 3D cone-shaped surface in the parameter space. The coordinates of the dominant circles can be found by searching the accumulator space for the cells with the highest values; that is, the cells where the most cones intersect. Just as in the linear HT, the *bias* problem (see Sec. 8.3.2) also occurs in the circle HT. Sections of circles (i.e., arcs) can be found in a similar way, in which case the maximum value possible for a given cell is proportional to the arc length.

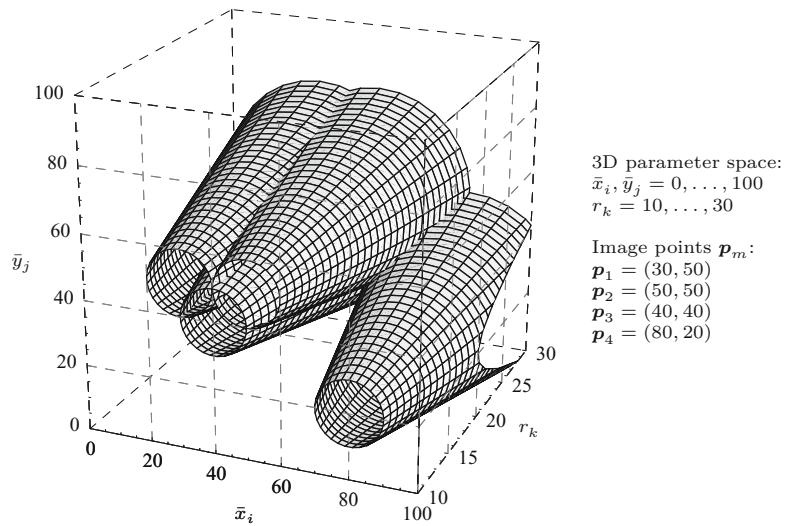
8.5.2 Ellipses

In a perspective image, most circular objects originating in our real, 3D world will actually appear in 2D images as ellipses, except in the case where the object lies on the optical axis and is observed from the front. For this reason, perfectly circular structures seldom occur

8 FINDING SIMPLE CURVES: THE HOUGH TRANSFORM

Fig. 8.15

3D parameter space for circles. For each image point $\mathbf{p} = (u, v)$, the cells lying on a cone (with its axis at (u, v) and varying radius r_k) in the 3D accumulator $\mathbf{A}(i, j, k)$ are traversed and incremented. The size of the discrete accumulator is set to $100 \times 100 \times 30$. Candidate center points are found where many of the 3D surfaces intersect.



in photographs. While the Hough transform can still be used to find ellipses, the larger parameter space required makes it substantially more expensive.

A general ellipse in 2D has five degrees of freedom and therefore requires five parameters to represent it, for example,

$$E = \langle \bar{x}, \bar{y}, r_a, r_b, \alpha \rangle, \quad (8.21)$$

where (\bar{x}, \bar{y}) are the coordinates of the center points, (r_a, r_b) are the two radii, and α is the orientation of the principal axis (Fig. 8.13).¹⁰ In order to find ellipses of any size, position, and orientation using the Hough transform, a 5D parameter space with a suitable resolution in each dimension is required. A simple calculation illustrates the enormous expense of representing this space: using a resolution of only $128 = 2^7$ steps in every dimension results in 2^{35} accumulator cells, and implementing these using 4-byte `int` values thus requires 2^{37} bytes (128 gigabytes) of memory. Moreover, the amount of processing required for filling and evaluating such a huge parameter space makes this method unattractive for real applications.

An interesting alternative in this case is the *generalized Hough transform*, which in principle can be used for detecting any arbitrary 2D shape [15, 117]. Using the generalized Hough transform, the shape of the sought-after contour is first encoded point by point in a table and then the associated parameter space is related to the position (x_c, y_c) , scale S , and orientation θ of the shape. This requires a 4D space, which is smaller than that of the Hough method for ellipses described earlier.

¹⁰ See Chapter 10, Eqn. (10.39) for a parametric equation of this ellipse.

Exercise 8.1. Drawing a straight line given in Hessian normal (HNF) form is not directly possible because typical graphics environments can only draw lines between two specified end points.¹¹ An HNF line $L = \langle\theta, r\rangle$, specified relative to a reference point $\mathbf{x}_r = (x_r, y_r)$, can be drawn into an image I in several ways (implement both versions):

Version 1: Iterate over all image points (u, v) ; if Eqn. (8.11), that is,

$$r = (u - x_r) \cdot \cos(\theta) + (v - y_r) \cdot \sin(\theta), \quad (8.22)$$

is satisfied for position (u, v) , then mark the pixel $I(u, v)$. Of course, this “brute force” method will only show those (few) line pixels whose positions satisfy the line equation *exactly*. To obtain a more “tolerant” drawing method, we first reformulate Eqn. (8.22) to

$$(u - x_r) \cdot \cos(\theta) + (v - y_r) \cdot \sin(\theta) - r = d. \quad (8.23)$$

Obviously, Eqn. (8.22) is only then exactly satisfied if $d = 0$ in Eqn. (8.23). If, however, Eqn. (8.22) is *not* satisfied, then the magnitude of $d \neq 0$ equals the distance of the point (u, v) from the line. Note that d itself may be positive or negative, depending on which side of the line (u, v) is located. This suggests the following version.

Version 2: Define a constant $w > 0$. Iterate over all image positions (u, v) ; whenever the inequality

$$|(u - x_r) \cdot \cos(\theta) + (v - y_r) \cdot \sin(\theta) - r| \leq w \quad (8.24)$$

is satisfied for position (u, v) , mark the pixel $I(u, v)$. For example, all line points should show with $w = 1$. What is the geometric meaning of w ?

Exercise 8.2. Develop a less “brutal” method (compared to Exercise 8.1) for drawing a straight line $L = \langle\theta, r\rangle$ in Hessian normal form (HNF). First, set up the HNF equations for the four border lines of the image, A, B, C, D . Now determine the intersection points of the given line L with each border line A, \dots, D and use the built-in `drawLine()` method or a similar routine to draw L by connecting the intersection points. Consider which special situations may appear and how they could be handled.

Exercise 8.3. Implement (or extend) the Hough transform for straight lines by including measures against the bias problem, as discussed in Sec. 8.3.2 (Eqn. (8.16)).

Exercise 8.4. Implement (or extend) the Hough transform for finding lines that takes into account line endpoints, as described in Sec. 8.3.2 (Eqn. (8.17)).

Exercise 8.5. Calculate the pairwise intersection points of all detected lines (see Eqns. (8.18)–(8.19)) and show the results graphically.

¹¹ For example, with `drawLine(x1, y1, x2, y2)` in ImageJ.

Exercise 8.6. Extend the Hough transform for straight lines so that updating the accumulator map takes into account the intensity (edge magnitude) of the current pixel, as described in Eqn. (8.15).

Exercise 8.7. Implement a *hierarchical* Hough transform for straight lines (see p. 172) capable of accurately determining line parameters.

Exercise 8.8. Implement the Hough transform for finding circles and circular arcs with varying radii. Make use of a fast algorithm for drawing circles in the accumulator array, such as described in Sec. 8.5.

Morphological Filters

In the discussion of the median filter in Chapter 5 (Sec. 5.4.2), we noticed that this type of filter can somehow alter 2D image structures. **Figure 9.1** illustrates once more how corners are rounded off, holes of a certain size are filled, and small structures, such as single dots or thin lines, are removed. The median filter thus responds selectively to the local shape of image structures, a property that might be useful for other purposes if it can be applied not just randomly but in a controlled fashion. Altering the local structure in a predictable way is exactly what “morphological” filters can do, which we focus on in this chapter.

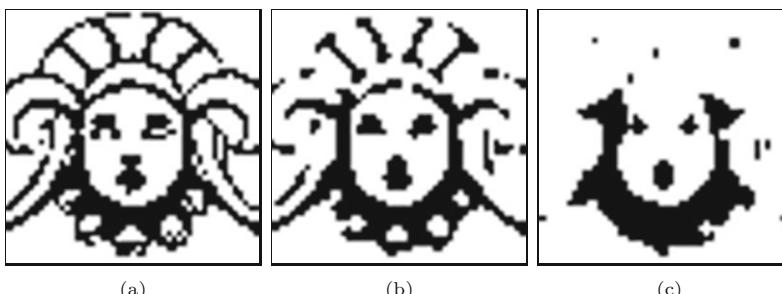


Fig. 9.1
Median filter applied to a binary image: original image (a) and results from a 3×3 pixel median filter (b) and a 5×5 pixel median filter (c).

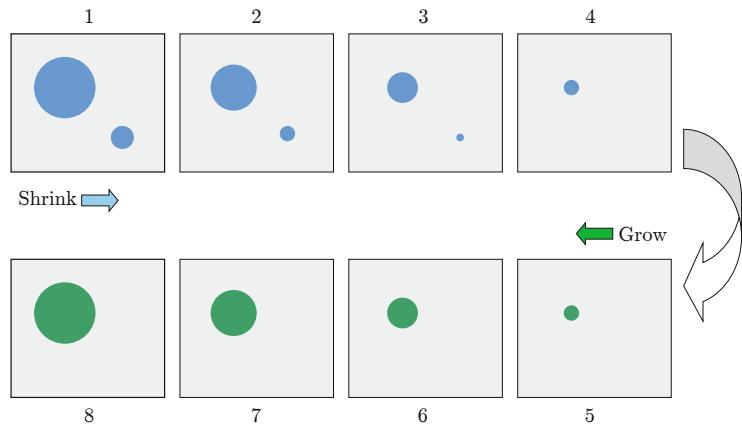
In their original form, morphological filters are aimed at binary images, images with only two possible pixel values, 0 and 1 or *black* and *white*, respectively. Binary images are found in many places, in particular in digital printing, document transmission (FAX) and storage, or as selection masks in image and video editing. Binary images can be obtained from grayscale images by simple thresholding (see Sec. 4.1.4) using either a global or a locally varying threshold value. We denote binary pixels with values 1 and 0 as *foreground* and *background* pixels, respectively. In most of the following examples, the foreground pixels are shown in black and background pixels are shown in white, as is common in printing.

At the end of this chapter, we will see that morphological filters are applicable not only to binary images but also to grayscale and

9 MORPHOLOGICAL FILTERS

Fig. 9.2

Basic idea of size-dependent removal of image structures. Small structures may be eliminated by iterative shrinking and subsequent growing. Ideally, the “surviving” structures should be restored to their original shape.



even color images, though these operations differ significantly from their binary counterparts.

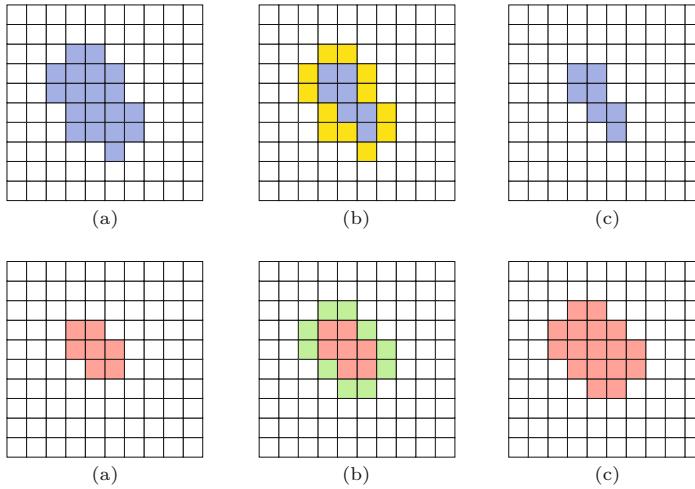
9.1 Shrink and Let Grow

Our starting point was the observation that a simple 3×3 pixel median filter can round off larger image structures and remove smaller structures, such as points and thin lines, in a binary image. This could be useful to eliminate structures that are below a certain size (e.g., to clean an image from noise or dirt). But how can we control the size and possibly the shape of the structures affected by such an operation?

Although its structural effects may be interesting, we disregard the median filter at this point and start with this task again from the beginning. Let's assume that we want to remove small structures from a binary image without significantly altering the remaining larger structures. The key idea for accomplishing this could be the following (Fig. 9.2):

1. First, all structures in the image are iteratively “shrunk” by peeling off a layer of a certain thickness around the boundaries.
2. Shrinking removes the smaller structures step by step, and only the larger structures remain.
3. The remaining structures are then grown back by the same amount.
4. Eventually the larger regions should have returned to approximately their original shapes, while the smaller regions have disappeared from the image.

All we need for this are two types of operations. “Shrinking” means to remove a layer of pixels from a foreground region around all its borders against the background (Fig. 9.3). The other way around, “growing”, adds a layer of pixels around the border of a foreground region (Fig. 9.4).



9.2 BASIC MORPHOLOGICAL OPERATIONS

Fig. 9.3

“Shrinking” a foreground region by removing a layer of border pixels: original image (a), identified foreground pixels that are in direct contact with the background (b), and result after shrinking (c).

Fig. 9.4

“Growing” a foreground region by attaching a layer of pixels: original image (a), identified background pixels that are in direct contact with the region (b), and result after growing (c).

9.1.1 Neighborhood of Pixels

For both operations, we must define the meaning of two pixels being adjacent (i.e., being “neighbors”). Two definitions of “neighborhood” are commonly used for rectangular pixel grids (Fig. 9.5):

- **4-neighborhood (\mathcal{N}_4):** the four pixels adjacent to a given pixel in the horizontal and vertical directions;
- **8-neighborhood (\mathcal{N}_8):** the pixels contained in \mathcal{N}_4 plus the four adjacent pixels along the diagonals.

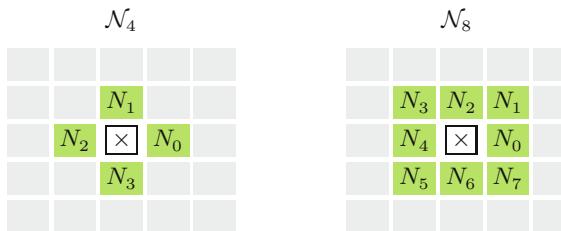


Fig. 9.5
Definitions of “neighborhood” on a rectangular pixel grid: 4-neighborhood $\mathcal{N}_4 = \{N_1, \dots, N_4\}$ and 8-neighborhood $\mathcal{N}_8 = \mathcal{N}_4 \cup \{N_5, \dots, N_8\}$.

9.2 Basic Morphological Operations

Shrinking and growing are indeed the two most basic morphological operations, which are referred to as “erosion” and “dilation”, respectively. These morphological operations, however, are much more general than illustrated in the example in Sec. 9.1. They go well beyond removing or attaching single pixel layers and—in combination—can perform much more complex operations.

9.2.1 The Structuring Element

Similar to the coefficient matrix of a linear filter (see Sec. 5.2), the properties of a morphological filter are specified by elements in a matrix called a “structuring element”. In binary morphology, the structuring element (just like the image itself) contains only the values 0

and 1,

$$H(i, j) \in \{0, 1\},$$

and the *hot spot* marks the origin of the coordinate system of H (Fig. 9.6). Notice that the hot spot is not necessarily located at the center of the structuring element, nor must its value be 1.

Fig. 9.6
Binary structuring element (example). 1-elements are marked with \bullet ; 0-cells are empty. The hot spot (boxed) is not necessarily located at the center.



9.2.2 Point Sets

For the formal specification of morphological operations, it is sometimes helpful to describe binary images as *sets* of 2D coordinate points.¹

For a binary image $I(u, v) \in \{0, 1\}$, the corresponding point set \mathcal{Q}_I consists of the coordinate pairs $\mathbf{p} = (u, v)$ of all foreground pixels,

$$\mathcal{Q}_I = \{\mathbf{p} \mid I(\mathbf{p}) = 1\}. \quad (9.1)$$

Of course, as shown in Fig. 9.7, not only a binary image I but also a structuring element H can be described as a point set.

Fig. 9.7
A binary image I or a structuring element H can each be described as a set of coordinate pairs, \mathcal{Q}_I and \mathcal{Q}_H , respectively. The dark shaded element in H marks the coordinate origin (hot spot).



$$I \equiv \mathcal{Q}_I = \{(1, 1), (2, 1), (2, 2)\}$$

$$H \equiv \mathcal{Q}_H = \{(0, 0), (1, 0)\}$$

With the description as point sets, fundamental operations on binary images can also be expressed as simple set operations. For example, *inverting* a binary image $I \rightarrow \bar{I}$ (i.e., exchanging foreground and background) is equivalent to building the *complementary* set

$$\mathcal{Q}_{\bar{I}} = \bar{\mathcal{Q}}_I = \{\mathbf{p} \in \mathbb{Z}^2 \mid \mathbf{p} \notin \mathcal{Q}_I\}. \quad (9.2)$$

Combining two binary images I_1 and I_2 by an OR operation between corresponding pixels, the resulting point set is the *union* of the individual point sets \mathcal{Q}_{I_1} and \mathcal{Q}_{I_2} ; that is,

$$\mathcal{Q}_{I_1 \vee I_2} = \mathcal{Q}_{I_1} \cup \mathcal{Q}_{I_2}. \quad (9.3)$$

Since a point set \mathcal{Q}_I is only an alternative representation of the binary image I (i.e., $I \equiv \mathcal{Q}_I$), we will use both image and set notations synonymously in the following. For example, we simply write \bar{I} instead of $\bar{\mathcal{Q}}_I$ for an inverted image as in Eqn. (9.2) or $I_1 \cup I_2$ instead of $\mathcal{Q}_{I_1} \cup \mathcal{Q}_{I_2}$ in Eqn. (9.3). The meaning should always be clear in the given context.

¹ *Morphology* is a mathematical discipline dealing with the algebraic analysis of geometrical structures and shapes, with strong roots in set theory.

Translating (shifting) a binary image I by some coordinate vector \mathbf{d} creates a new image with the content

$$I_{\mathbf{d}}(\mathbf{p} + \mathbf{d}) = I(\mathbf{p}) \quad \text{oder} \quad I_{\mathbf{d}}(\mathbf{p}) = I(\mathbf{p} - \mathbf{d}), \quad (9.4)$$

which is equivalent to changing the coordinates of the original point set in the form

$$I_{\mathbf{d}} \equiv \{(\mathbf{p} + \mathbf{d}) \mid \mathbf{p} \in I\}. \quad (9.5)$$

In some cases, it is also necessary to *reflect* (mirror) a binary image or point set about its origin, which we denote as

$$I^* \equiv \{-\mathbf{p} \mid \mathbf{p} \in I\}. \quad (9.6)$$

9.2.3 Dilation

A *dilation* is the morphological operation that corresponds to our intuitive concept of “growing” as discussed already. As a set operation, it is defined as

$$I \oplus H \equiv \{(\mathbf{p} + \mathbf{q}) \mid \text{for all } \mathbf{p} \in I, \mathbf{q} \in H\}. \quad (9.7)$$

Thus the point set produced by a dilation is the (vector) sum of all possible pairs of coordinate points from the original sets I and H , as illustrated by a simple example in Fig. 9.8. Alternatively, one could view the dilation as the structuring element H being *replicated* at each foreground pixel of the image I or, conversely, the image I being replicated at each foreground element of H . Expressed in set notation,² this is

$$I \oplus H \equiv \bigcup_{\mathbf{p} \in I} H_{\mathbf{p}} = \bigcup_{\mathbf{q} \in H} I_{\mathbf{q}}, \quad (9.8)$$

with $H_{\mathbf{p}}$, $I_{\mathbf{q}}$ denoting the sets H , I shifted by \mathbf{p} and \mathbf{q} , respectively (see Eqn. (9.5)).

I	H	$I \oplus H$
0 1 2 3	-1 0 1	0 1 2 3
0    	-1    0    1 	0     1    2  3 
⊕	=	

$$I \equiv \{(1, 1), (2, 1), (2, 2)\}, \quad H \equiv \{(\mathbf{0}, \mathbf{0}), (\mathbf{1}, \mathbf{0})\}$$

$$\begin{aligned} I \oplus H &\equiv \{ (1, 1) + (\mathbf{0}, \mathbf{0}), (1, 1) + (\mathbf{1}, \mathbf{0}), \\ &\quad (2, 1) + (\mathbf{0}, \mathbf{0}), (2, 1) + (\mathbf{1}, \mathbf{0}), \\ &\quad (2, 2) + (\mathbf{0}, \mathbf{0}), (2, 2) + (\mathbf{1}, \mathbf{0}) \} \end{aligned}$$

Fig. 9.8
Binary dilation example. The binary image I is subject to dilation with the structuring element H . In the result $I \oplus H$ the structuring element H is replicated at every foreground pixel of the original image I .

² See also Sec. A.2 in the Appendix.

9.2.4 Erosion

The quasi-inverse of dilation is the *erosion* operation, again defined in set notation as

$$I \ominus H \equiv \{\mathbf{p} \in \mathbb{Z}^2 \mid (\mathbf{p} + \mathbf{q}) \in I, \text{ for all } \mathbf{q} \in H\}. \quad (9.9)$$

This operation can be interpreted as follows. A position \mathbf{p} is contained in the result $I \ominus H$ if (and only if) the structuring element H —when placed at this position \mathbf{p} —is *fully contained* in the foreground pixels of the original image; that is, if $H_{\mathbf{p}}$ is a subset of I . Equivalent to Eqn. (9.9), we could thus define binary erosion as

$$I \ominus H \equiv \{\mathbf{p} \in \mathbb{Z}^2 \mid H_{\mathbf{p}} \subseteq I\}. \quad (9.10)$$

Figure 9.9 shows a simple example for binary erosion.

Fig. 9.9
Binary erosion example. The binary image I is subject to erosion with H as the structuring element. H is only covered by I when placed at position $\mathbf{p} = (1, 1)$, thus the resulting points set contains only the single coordinate $(1, 1)$.

I				H			$I \ominus H$			
0	1	2	3	-1	0	1	0	1	2	3
0	■			-1			0	■		
1	●	●		0	■	●	1		●	
2		●		1			2			
3							3			

$$I \equiv \{(1, 1), (2, 1), (2, 2)\}, \quad H \equiv \{(\mathbf{0}, \mathbf{0}), (1, 0)\}$$

$I \ominus H \equiv \{(1, 1)\}$ because
$(1, 1) + (\mathbf{0}, \mathbf{0}) = (1, 1) \in I \quad \text{and} \quad (1, 1) + (1, 0) = (2, 1) \in I$

9.2.5 Formal Properties of Dilation and Erosion

The dilation operation is *commutative*,

$$I \oplus H = H \oplus I, \quad (9.11)$$

and therefore—just as in linear convolution—the image and the structuring element (filter) can be exchanged to get the same result. Dilation is also *associative*, that is,

$$(I_1 \oplus I_2) \oplus I_3 = I_1 \oplus (I_2 \oplus I_3), \quad (9.12)$$

and therefore the ordering of multiple dilations is not relevant. This also means—analogous to linear filters (cf. Eqn. (5.25))—that a dilation with a large structuring element of the form $H_{\text{big}} = H_1 \oplus H_2 \oplus \dots \oplus H_K$ can be efficiently implemented as a sequence of multiple dilations with smaller structuring elements by

$$I \oplus H_{\text{big}} = (\dots ((I \oplus H_1) \oplus H_2) \oplus \dots \oplus H_K) \quad (9.13)$$

There is also a *neutral element* (δ) for the dilation operation, similar to the Dirac function for the linear convolution (see Sec. 5.3.4),

$$I \oplus \delta = \delta \oplus I = I, \quad \text{with } \delta = \{(0, 0)\}. \quad (9.14)$$

The *erosion* operation is, in contrast to dilation (but similar to arithmetic subtraction), *not* commutative, that is,

$$I \ominus H \neq H \ominus I, \quad (9.15)$$

9.2 BASIC MORPHOLOGICAL OPERATIONS

in general. However, if erosion and dilation are combined, then—again in analogy with arithmetic subtraction and addition—the following chain rule holds:

$$(I_1 \ominus I_2) \ominus I_3 = I_1 \ominus (I_2 \oplus I_3). \quad (9.16)$$

Although dilation and erosion are not mutually inverse (in general, the effects of dilation cannot be undone by a subsequent erosion), there are still some strong formal relations between these two operations. For one, dilation and erosion are *dual* in the sense that a dilation of the *foreground* (I) can be accomplished by an erosion of the *background* (\bar{I}) and subsequent inversion of the result,

$$I \oplus H = \overline{(\bar{I} \ominus H^*)}, \quad (9.17)$$

where H^* denotes the *reflection* of H (Eqn. (9.6)). This works similarly the other way, too, namely

$$\bar{I} \ominus H = \overline{(I \oplus H^*)}, \quad (9.18)$$

effectively eroding the foreground by dilating the background with the mirrored structuring element, as illustrated by the example in Fig. 9.10 (see [88, pp. 521–524] for a formal proof).

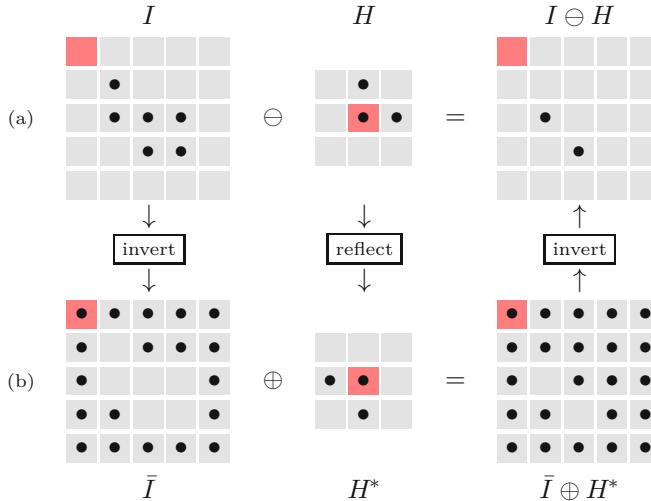


Fig. 9.10
Implementing erosion via dilation. The binary erosion of the foreground $I \ominus H$ (a) can be implemented by dilating the inverted (background) image \bar{I} with the reflected structuring element H^* and subsequently inverting the result again (b).

Equation (9.18) is interesting because it shows that we only need to implement either dilation or erosion for computing both, considering that the foreground–background inversion is a very simple task. Algorithm 9.1 gives a simple algorithmic description of dilation and erosion based on the aforementioned relationships.

9 MORPHOLOGICAL FILTERS

Alg. 9.1

Binary dilation and erosion.

Procedure DILATE() implements the binary dilation as suggested by Eqn. (9.8). The original image I is displaced to each foreground coordinate of H and then copied into the resulting image I' . The hot spot of the structuring element H is assumed to be at coordinate $(0, 0)$. Procedure ERODE() implements the binary erosion by dilating the inverted image \bar{I} with the reflected structuring element H^* , as described by Eqn. (9.18).

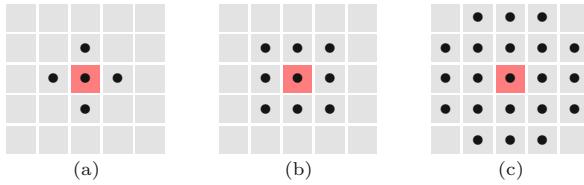
```

1: Dilate( $I, H$ )
   Input:  $I$ , a binary image of size  $M \times N$ ;
           $H$ , a binary structuring element.
   Returns the dilated image  $I' = I \oplus H$ .
2: Create map  $I': M \times N \mapsto \{0, 1\}$             $\triangleright$  new binary image  $I'$ 
3: for all  $(p) \in M \times N$  do
4:    $I'(p) \leftarrow 0$                                  $\triangleright I' \leftarrow \{ \}$ 
5:   for all  $q \in H$  do
6:     for all  $p \in I$  do
7:        $I'(p + q) \leftarrow 1$                        $\triangleright I' \leftarrow I' \cup \{(p+q)\}$ 
8:   return  $I'$                                    $\triangleright I' = I \oplus H$ 

9: Erode( $I, H$ )
   Input:  $I$ , a binary image of size  $M \times N$ ;
           $H$ , a binary structuring element.
   Returns the eroded image  $I' = I \ominus H$ .
10:  $\bar{I} \leftarrow \text{Invert}(I)$                       $\triangleright \bar{I} \leftarrow \neg I$ 
11:  $H^* \leftarrow \text{Reflect}(H)$ 
12:  $I' \leftarrow \text{Invert}(\text{Dilate}(\bar{I}, H^*))$        $\triangleright I' = I \ominus H = \overline{(\bar{I} \oplus H^*)}$ 
13: return  $I'$ 
```

Fig. 9.11

Typical binary structuring elements of various sizes. 4-neighborhood (a), 8-neighborhood (b), “small disk” (c).



9.2.6 Designing Morphological Filters

A morphological filter is unambiguously specified by (a) the type of operation and (b) the contents of the structuring element. The appropriate size and shape of the structuring element depends upon the application, image resolution, etc. In practice, structuring elements of quasi-circular shape are frequently used, such as the examples shown in Fig. 9.11.

A dilation with a circular (disk-shaped) structuring element with radius r adds a layer of thickness r to any foreground structure in the image. Conversely, an erosion with that structuring element peels off layers of the same thickness. Figure 9.13 shows the results of dilation and erosion with disk-shaped structuring elements of different diameters applied to the original image in Fig. 9.12. Dilation and erosion results for various other structuring elements are shown in Fig. 9.14.

Disk-shaped structuring elements are commonly used to implement *isotropic* filters, morphological operations that have the same effect in every direction. Unlike linear filters (e.g., the 2D Gaussian filter in Sec. 5.3.3), it is generally not possible to compose an isotropic 2D structuring element H° from 1D structuring elements H_x and H_y since the dilation $H_x \oplus H_y$ always results in a rectangular (i.e., non-isotropic) structure. A remedy for approximating large disk-shaped filters is to alternately apply smaller disk-shaped operators of differ-



9.2 BASIC MORPHOLOGICAL OPERATIONS

Fig. 9.12

Original binary image and the section used in the following examples (illustration by Albrecht Dürer, 1515).

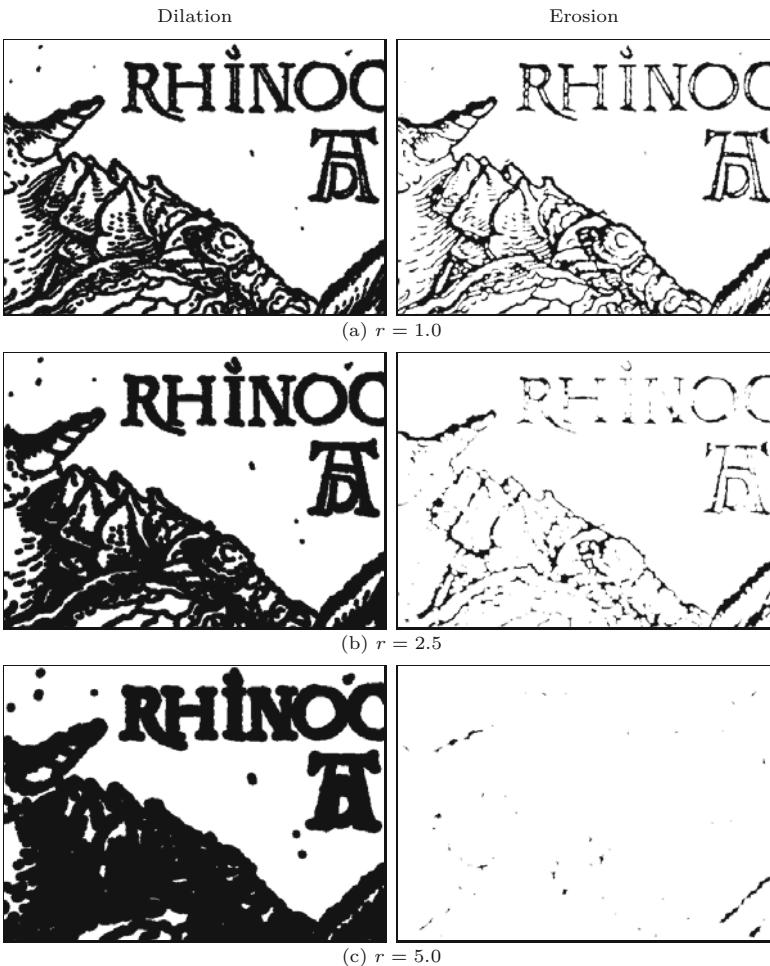


Fig. 9.13

Results of binary dilation and erosion with disk-shaped structuring elements. The radius of the disk (r) is 1.0 (a), 2.5 (b), and 5.0 (c).

ent shapes, as illustrated in Fig. 9.15. The resulting filter is generally not fully isotropic but can be implemented efficiently as a sequence of small filters.

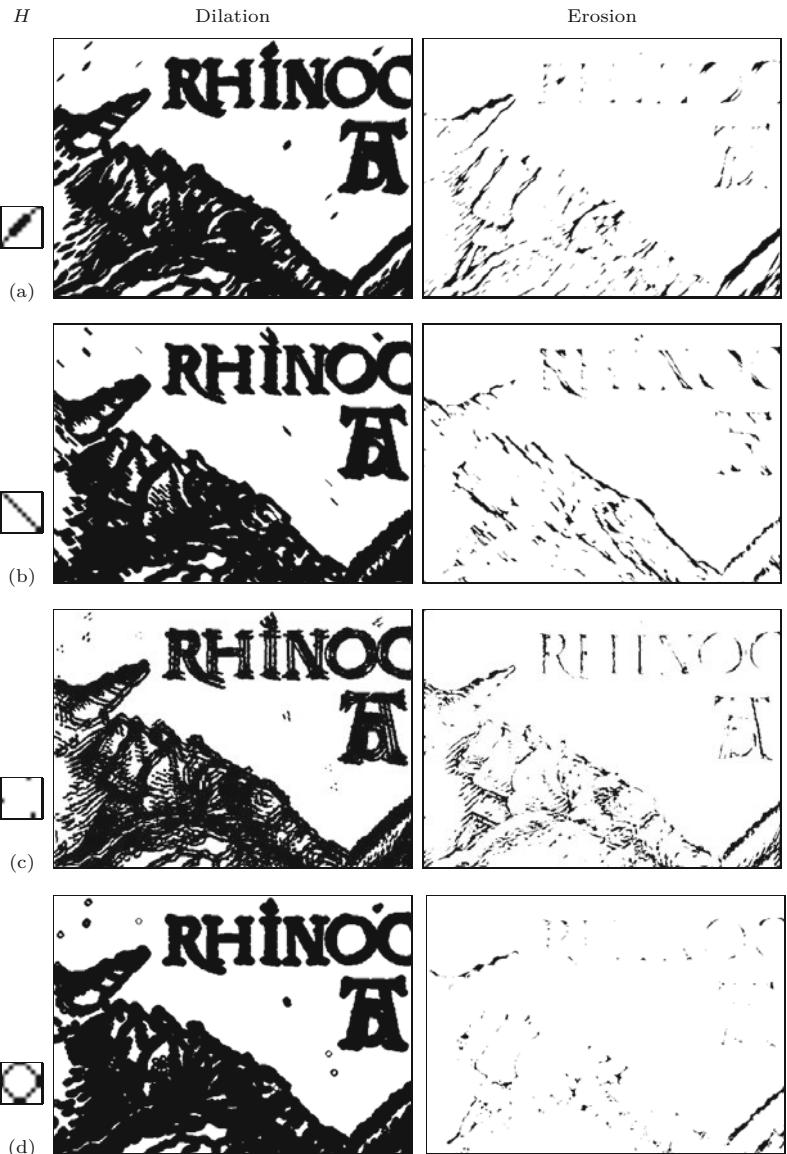
9.2.7 Application Example: Outline

A typical application of morphological operations is to extract the boundary pixels of the foreground structures. The process is very simple. First, we apply an erosion on the original image I to remove the boundary pixels of the foreground,

9 MORPHOLOGICAL FILTERS

Fig. 9.14

Examples of binary dilation and erosion with various free-form structuring elements. The structuring elements H are shown in the left column (enlarged). Notice that the dilation expands every isolated foreground point to the shape of the structuring element, analogous to the *impulse response* of a linear filter. Under erosion, only those elements where the structuring element is fully contained in the original image survive.



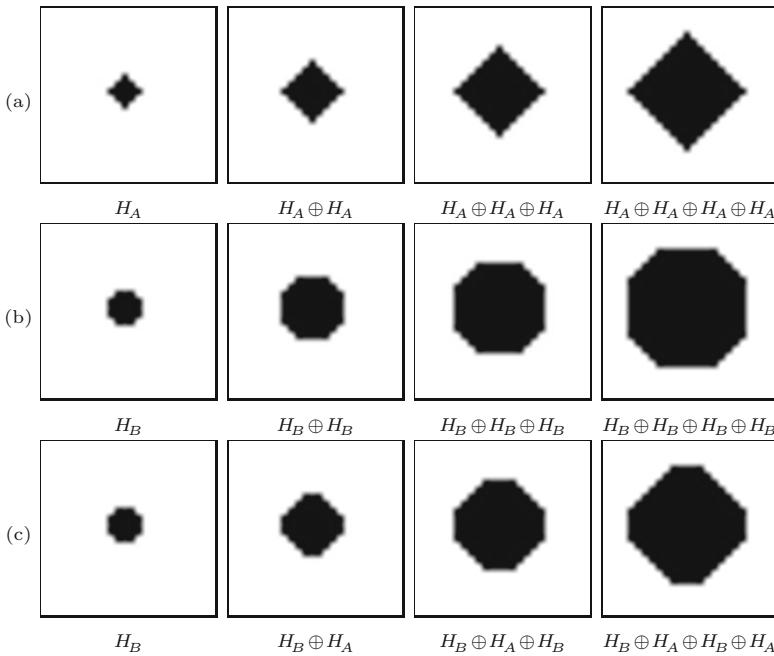
$$I' = I \ominus H_n,$$

where H_n is a structuring element, for example, for a 4- or 8-neighborhood (Fig. 9.11) as the structuring element H_n . The actual boundary pixels B are those contained in the original image but *not* in the eroded image, that is, the *intersection* of the original image I and the inverted result \bar{I}' , or

$$B \leftarrow I \cap \bar{I}' = I \cap \overline{(I \ominus H_n)}. \quad (9.19)$$

Figure 9.17 shows an example for the extraction of region boundaries. Notice that using the 4-neighborhood as the structuring element H_n produces “8-connected” contours and vice versa [125, p. 504].

The process of boundary extraction is illustrated on a simple example in **Fig. 9.16**. As can be observed in this figure, the result B



9.2 BASIC MORPHOLOGICAL OPERATIONS

Fig. 9.15

Composition of large morphological filters by repeated application of smaller filters: repeated application of the structuring element H_A (a) and structuring element H_B (b); alternating application of H_B and H_A (c).

contains exactly those pixels that are *different* in the original image I and the eroded image $I' = I \ominus H_n$, which can also be obtained by an exclusive-OR (XOR) operation between pairs of pixels; that is, boundary extraction from a binary image can be implemented as

$$B(\mathbf{p}) \leftarrow I(\mathbf{p}) \text{ XOR } (I \ominus H_n)(\mathbf{p}), \quad \text{for all } \mathbf{p}. \quad (9.20)$$

Figure 9.17 shows a more complex example for isolating the boundary pixels in a real image.

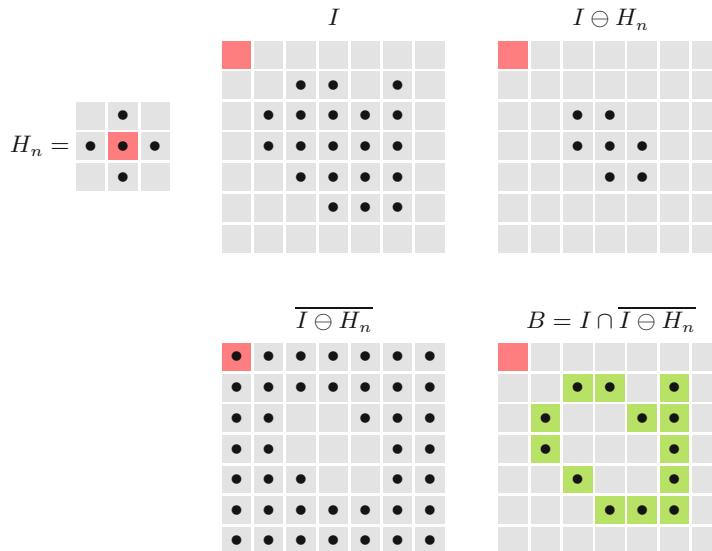
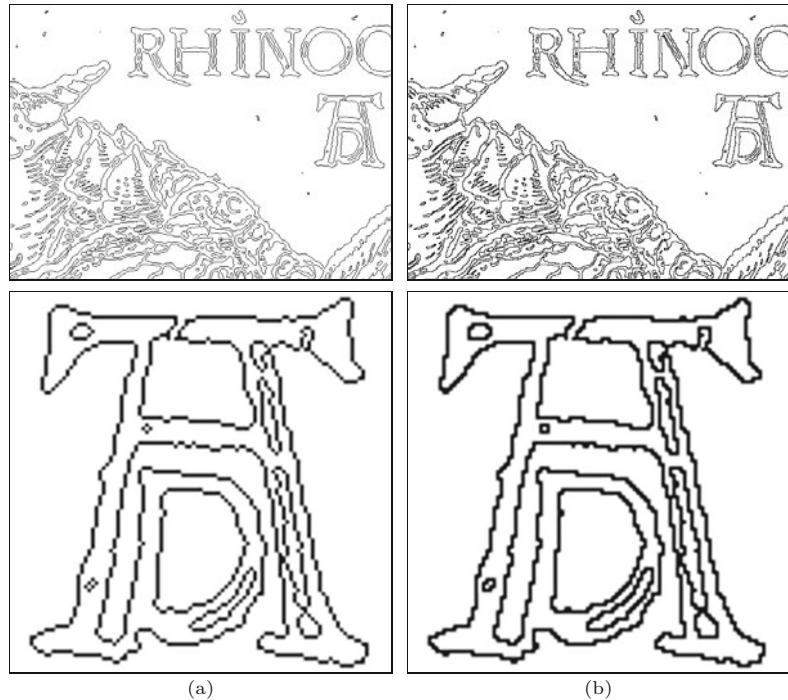


Fig. 9.16

Outline example using a 4-neighborhood structuring element H_n . The image I is first eroded ($I \ominus H_n$) and subsequently inverted ($\overline{I \ominus H_n}$). The boundary pixels are finally obtained as the intersection $I \cap \overline{I \ominus H_n}$.

Fig. 9.17

Extraction of boundary pixels using morphological operations. The 4-neighborhood structuring element used in (a) produces 8-connected contours. Conversely, using the 8-neighborhood as the structuring element gives 4-connected contours (b).



9.3 Composite Morphological Operations

Due to their semiduality, dilation and erosion are often used together in composite operations, two of which are so important that they even carry their own names and symbols: “opening” and “closing”. They are probably the most frequently used morphological operations in practice.

9.3.1 Opening

A binary opening $I \circ H$ denotes an erosion followed by a dilation with the *same* structuring element H ,

$$I \circ H = (I \ominus H) \oplus H. \quad (9.21)$$

The main effect of an opening is that all foreground structures that are smaller than the structuring element are eliminated in the first step (erosion). The remaining structures are smoothed by the subsequent dilation and grown back to approximately their original size, as demonstrated by the examples in Fig. 9.18. This process of shrinking and subsequent growing corresponds to the idea for eliminating small structures that we had initially sketched in Sec. 9.1.

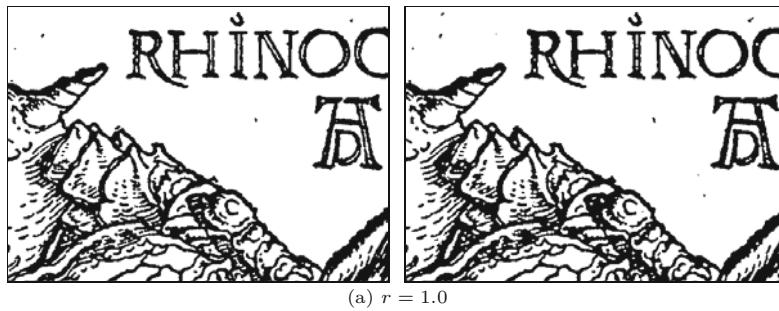
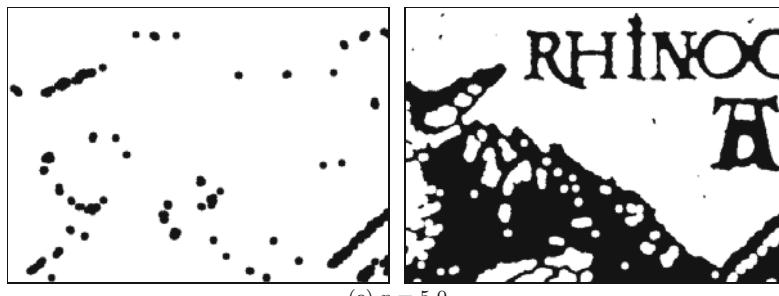
9.3.2 Closing

When the sequence of erosion and dilation is reversed, the resulting operation is called a closing and denoted $I \bullet H$,

$$I \bullet H = (I \oplus H) \ominus H. \quad (9.22)$$

Opening

Closing

(a) $r = 1.0$ (b) $r = 2.5$ (c) $r = 5.0$

9.3 COMPOSITE MORPHOLOGICAL OPERATIONS

Fig. 9.18

Binary opening and closing with disk-shaped structuring elements. The radius r of the structuring element H is 1.0 (top), 2.5 (center), or 5.0 (bottom).

A *closing* removes (closes) holes and fissures in the foreground structures that are smaller than the structuring element H . Some examples with typical disk-shaped structuring elements are shown in Fig. 9.18.

9.3.3 Properties of Opening and Closing

Both operations, opening as well as closing, are *idempotent*, meaning that their results are “final” in the sense that any subsequent application of the same operation no longer changes the result, that is,

$$\begin{aligned} I \circ H &= (I \circ H) \circ H = ((I \circ H) \circ H) \circ H = \dots, \\ I \bullet H &= (I \bullet H) \bullet H = ((I \bullet H) \bullet H) \bullet H = \dots. \end{aligned} \quad (9.23)$$

Also, opening and closing are “duals” in the sense that opening the foreground is equivalent to closing the background and vice versa, that is,

$$I \circ H = \overline{\overline{I} \bullet H} \quad \text{and} \quad I \bullet H = \overline{\overline{I} \circ H}. \quad (9.24)$$

9.4 Thinning (Skeletonization)

Thinning is a common morphological technique which aims at shrinking binary structures down to a maximum thickness of one pixel without splitting them into multiple parts. This is accomplished by iterative “conditional” erosion. It is applied to a local neighborhood only if a sufficiently thick structure remains and the operation does not cause a separation to occur. This requires that, depending on the local image structure, a decision must be made at every image position whether another erosion step may be applied or not. The operation continues until no more changes appear in the resulting image. It follows that, compared to the ordinary (“homogeneous”) morphological discussed earlier, thinning is computationally expensive in general. A frequent application of thinning is to calculate the “skeleton” of a binary region, for example, for structural matching of 2D shapes.

Thinning is also known by the terms *center line detection* and *medial axis transform*. Many different implementations of varied complexity and efficiency exist (see, e.g., [2, 7, 68, 108, 201]). In the following, we describe the classic algorithm by Zhang and Suen [265] and its implementation as a representative example.³

9.4.1 Thinning Algorithm by Zhang and Suen

The input to this algorithm is a binary image I , with foreground pixels carrying the value 1 and background pixels with value 0. The algorithm scans the image and at each position (u, v) examines a 3×3 neighborhood with the central element P and the surrounding values $\mathbf{N} = (N_0, N_1, \dots, N_7)$, as illustrated in Fig. 9.5(b). The complete process is summarized in Alg. 9.2.

For classifying the contents of the local neighborhood \mathbf{N} we first define the function

$$B(\mathbf{N}) = N_0 + N_1 + \dots + N_7 = \sum_{i=0}^7 N_i, \quad (9.25)$$

which simply counts surrounding foreground pixels. We also define the so-called “connectivity number” to express how many binary components are connected via the current center pixel at position (u, v) . This quantity is equivalent to the number of $1 \rightarrow 0$ transitions in the sequence (N_0, \dots, N_7, N_0) , or expressed in arithmetic terms,

$$C(\mathbf{N}) = \sum_{i=0}^7 N_i \cdot [N_i - N_{(i+1) \bmod 8}]. \quad (9.26)$$

Figure 9.19 shows some selected examples for the neighborhood \mathbf{N} and the associated values for the functions $B(\mathbf{N})$ and $C(\mathbf{N})$. Based on the above functions, we finally define two Boolean predicates R_1, R_2 on the neighborhood \mathbf{N} ,

³ The built-in thinning operation in ImageJ is also based on this algorithm.

$$R_1(N) := [2 \leq B(N) \leq 6] \wedge [C(N) = 1] \wedge [N_6 \cdot N_0 \cdot N_2 = 0] \wedge [N_4 \cdot N_6 \cdot N_0 = 0], \quad (9.27)$$

$$R_2(N) := [2 \leq B(N) \leq 6] \wedge [C(N) = 1] \wedge [N_0 \cdot N_2 \cdot N_4 = 0] \wedge [N_2 \cdot N_4 \cdot N_6 = 0]. \quad (9.28)$$

$B=0$	$B=7$	$B=6$	$B=3$	$B=4$	$B=5$	$B=6$
$C=0$	$C=1$	$C=2$	$C=3$	$C=4$	$C=5$	$C=6$

9.4 THINNING (SKELETONIZATION)

Fig. 9.19
Selected binary neighborhood patterns N and associated function values $B(N)$ and $C(N)$ (see Eqns. (9.25)–(9.26)).

Depending on the outcome of $R_1(N)$ and $R_2(N)$, the foreground pixel at the center position of N is either deleted (i.e., eroded) or marked as non-removable (see Alg. 9.2, lines 16 and 27).

Figure 9.20 illustrates the effect of layer-by-layer thinning performed by procedure `ThinOnce()`. In every iteration, only one “layer” of foreground pixels is selectively deleted. An example of thinning applied to a larger binary image is shown in Fig. 9.21.

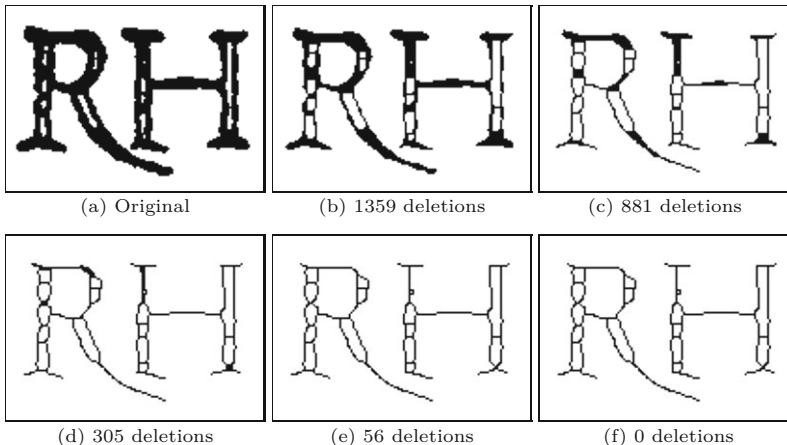


Fig. 9.20
Iterative application of the `ThinOnce()` procedure. The “deletions” indicated in (b–f) denote the number of pixels that were removed from the previous image. No deletions occurred in the final iteration (from (e) to (f)). Thus five iterations were required to thin this image.

9.4.2 Fast Thinning Algorithm

In a binary image, only $2^8 = 256$ different combinations of zeros and ones are possible inside any 8-neighborhood. Since the expressions in Eqns. (9.27)–(9.27) are relatively costly to evaluate it makes sense to pre-calculate and tabulate all 256 instances (see Fig. 9.22). This is the basis of the fast version of Zhang and Suen’s algorithm, summarized in Alg. 9.3. It uses a decision table Q , which is constant and calculated only once by procedure `MakeDeletionCodeTable()` in Alg. 9.3 (lines 34–45). The table contains the binary codes

$$Q(i) \in \{0, 1, 2, 3\} = \{00_b, 01_b, 10_b, 11_b\}, \quad (9.29)$$

for $i = 0, \dots, 255$, where the two bits correspond to the predicates R_1 and R_2 , respectively. The associated test is found in procedure `ThinOnceFast()` in line 19. The two passes are in this case controlled by a separate loop variable ($p = 1, 2$). In the concrete implementation, the map Q is not calculated at the start but defined as a constant array (see Prog. 9.1 for the actual Java code).

9 MORPHOLOGICAL FILTERS

Alg. 9.2

Iterative thinning algorithm by Zhang und Suen [265]. Procedure `ThinOnce()` performs a single thinning step on the supplied binary image I_b and returns the number of deleted foreground pixels. It is iteratively invoked by `Thin()` until no more pixels are deleted. The required pixel deletions are only registered in the binary map D and executed en-bloc at the end of every iteration. Lines 40–42 define the functions $R_1()$, $R_2()$, $B()$ and $C()$ used to characterize the local pixel neighborhoods. Note that the order of processing the image positions (u, v) in the `for all` loops in **Pass 1** and **Pass 2** is completely arbitrary. In particular, positions could be processed simultaneously, so the algorithm may be easily parallelized (and thereby accelerated).

```

1: Thin( $I_b, i_{\max}$ )
   Input:  $I_b$ , binary image with background = 0, foreground > 0;
           $i_{\max}$ , max. number of iterations. Returns the number of iterations
          performed and modifies  $I_b$ .
2:  $(M, N) \leftarrow \text{Size}(I_b)$ 
3: Create a binary map  $D: M \times N \mapsto \{0, 1\}$ 
4:  $i \leftarrow 0$ 
5: do
6:    $n_d \leftarrow \text{ThinOnce}(I_b, D)$ 
7:    $i \leftarrow i + 1$ 
8: while ( $n_d > 0 \wedge i < i_{\max}$ )  $\triangleright$  do ... while more deletions required
9: return  $i$ 

10: ThinOnce( $I_b, D$ )
    Pass 1:
11:    $n_1 \leftarrow 0$   $\triangleright$  deletion counter
12:   for all image positions  $(u, v) \in M \times N$  do
13:      $D(u, v) \leftarrow 0$ 
14:     if  $I_b(u, v) > 0$  then
15:        $N \leftarrow \text{GetNeighborhood}(I_b, u, v)$ 
16:       if  $R_1(N)$  then  $\triangleright$  see Eq. 9.27
17:          $D(u, v) \leftarrow 1$   $\triangleright$  mark pixel  $(u, v)$  for deletion
18:          $n_1 \leftarrow n_1 + 1$ 
19:     if  $n_1 > 0$  then  $\triangleright$  at least 1 deletion required
20:       for all image positions  $(u, v) \in M \times N$  do
21:          $I_b(u, v) \leftarrow I_b(u, v) - D(u, v)$   $\triangleright$  delete all marked pixels

    Pass 2:
22:    $n_2 \leftarrow 0$ 
23:   for all image positions  $(u, v) \in M \times N$  do
24:      $D(u, v) \leftarrow 0$ 
25:     if  $I_b(u, v) > 0$  then
26:        $N \leftarrow \text{GetNeighborhood}(I_b, u, v)$ 
27:       if  $R_2(N)$  then  $\triangleright$  see Eq. 9.28
28:          $D(u, v) \leftarrow 1$   $\triangleright$  mark pixel  $(u, v)$  for deletion
29:          $n_2 \leftarrow n_2 + 1$ 
30:     if  $n_2 > 0$  then  $\triangleright$  at least 1 deletion required
31:       for all image positions  $(u, v) \in M \times N$  do
32:          $I_b(u, v) \leftarrow I_b(u, v) - D(u, v)$   $\triangleright$  delete all marked pixels
33:   return  $n_1 + n_2$ 

34: GetNeighborhood( $I_b, u, v$ )
35:    $N_0 \leftarrow I_b(u + 1, v), \quad N_1 \leftarrow I_b(u + 1, v - 1)$ 
36:    $N_2 \leftarrow I_b(u, v - 1), \quad N_3 \leftarrow I_b(u - 1, v - 1)$ 
37:    $N_4 \leftarrow I_b(u - 1, v), \quad N_5 \leftarrow I_b(u - 1, v + 1)$ 
38:    $N_6 \leftarrow I_b(u, v + 1), \quad N_7 \leftarrow I_b(u + 1, v + 1)$ 
39:   return  $(N_0, N_1, \dots, N_7)$ 

40:  $R_1(N) := [2 \leq B(N) \leq 6] \wedge [C(N) = 1] \wedge [N_6 \cdot N_0 \cdot N_2 = 0] \wedge [N_4 \cdot N_6 \cdot N_0 = 0]$ 
41:  $R_2(N) := [2 \leq B(N) \leq 6] \wedge [C(N) = 1] \wedge [N_0 \cdot N_2 \cdot N_4 = 0] \wedge [N_2 \cdot N_4 \cdot N_6 = 0]$ 

42:  $B(N) := \sum_{i=0}^7 N_i, \quad C(N) := \sum_{i=0}^7 N_i \cdot [N_i - N_{(i+1) \bmod 8}]$ 

```

```

1: ThinFast( $I_b, i_{\max}$ )
   Input:  $I_b$ , binary image with background = 0, foreground > 0;
           $i_{\max}$ , max. number of iterations. Returns the number of iterations
          performed and modifies  $I_b$ .
2:  $(M, N) \leftarrow \text{Size}(I_b)$ 
3:  $Q \leftarrow \text{MakeDeletionCodeTable}()$ 
4: Create a binary map  $D: M \times N \mapsto \{0, 1\}$ 
5:  $i \leftarrow 0$ 
6: do
7:    $n_d \leftarrow \text{ThinOnce}(I_b, D)$ 
8:   while  $(n_d > 0 \wedge i < i_{\max})$   $\triangleright$  do ... while more deletions required
9: return  $i$ 

10: ThinOnceFast( $I_b, D$ )            $\triangleright$  performs a single thinning iteration
11:    $n_d \leftarrow 0$                    $\triangleright$  number of deletions in both passes
12:   for  $p \leftarrow 1, 2$  do           $\triangleright$  pass counter (2 passes)
13:      $n \leftarrow 0$                    $\triangleright$  number of deletions in current pass
14:     for all image positions  $(u, v)$  do
15:        $D(u, v) \leftarrow 0$ 
16:       if  $I_b(u, v) = 1$  then       $\triangleright I_b(u, v) = P$ 
17:          $c \leftarrow \text{GetNeighborhoodIndex}(I_b, u, v)$ 
18:          $q \leftarrow Q(c)$              $\triangleright q \in \{0, 1, 2, 3\} = \{00_b, 01_b, 10_b, 11_b\}$ 
19:         if  $(p \text{ and } q) \neq 0$  then     $\triangleright$  bitwise ‘and’ operation
20:            $D(u, v) \leftarrow 1$          $\triangleright$  mark pixel  $(u, v)$  for deletion
21:            $n \leftarrow n + 1$ 
22:       if  $n > 0$  then           $\triangleright$  at least 1 deletion is required
23:          $n_d \leftarrow n_d + n$ 
24:         for all image positions  $(u, v)$  do
25:            $I_b(u, v) \leftarrow I_b(u, v) - D(u, v)$      $\triangleright$  delete all marked
               pixels
26: return  $n_d$ 

27: GetNeighborhoodIndex( $I_b, u, v$ )
28:    $N_0 \leftarrow I_b(u + 1, v), \quad N_1 \leftarrow I_b(u + 1, v - 1)$ 
29:    $N_2 \leftarrow I_b(u, v - 1), \quad N_3 \leftarrow I_b(u - 1, v - 1)$ 
30:    $N_4 \leftarrow I_b(u - 1, v), \quad N_5 \leftarrow I_b(u - 1, v + 1)$ 
31:    $N_6 \leftarrow I_b(u, v + 1), \quad N_7 \leftarrow I_b(u + 1, v + 1)$ 
32:    $c \leftarrow N_0 + N_1 \cdot 2 + N_2 \cdot 4 + N_3 \cdot 8 + N_4 \cdot 16 + N_5 \cdot 32 + N_6 \cdot 64 + N_7 \cdot 128$ 
33: return  $c$                        $\triangleright c \in [0, 255]$ 

34: MakeDeletionCodeTable()
35: Create maps  $Q: [0, 255] \mapsto \{0, 1, 2, 3\}, \quad N: [0, 7] \mapsto \{0, 1\}$ 
36: for  $i \leftarrow 0, \dots, 255$  do           $\triangleright$  list all possible neighborhoods
37:   for  $k \leftarrow 0, \dots, 7$  do           $\triangleright$  check neighbors  $0, \dots, 7$ 
38:      $N(k) \leftarrow \begin{cases} 1 & \text{if } (i \text{ and } 2^k) \neq 0 \\ 0 & \text{otherwise} \end{cases}$      $\triangleright$  test the  $k^{\text{th}}$  bit of  $i$ 
39:      $q \leftarrow 0$ 
40:     if  $R_1(N)$  then           $\triangleright$  see Alg. 9.2, line 40
41:        $q \leftarrow q + 1$            $\triangleright$  set bit 0 of  $q$ 
42:     if  $R_2(N)$  then           $\triangleright$  see Alg. 9.2, line 41
43:        $q \leftarrow q + 2$            $\triangleright$  set bit 1 of  $q$ 
44:      $Q(i) \leftarrow q$            $\triangleright q \in \{0, 1, 2, 3\} = \{00_b, 01_b, 10_b, 11_b\}$ 
45: return  $Q$ 

```

9.4 THINNING (SKELETONIZATION)

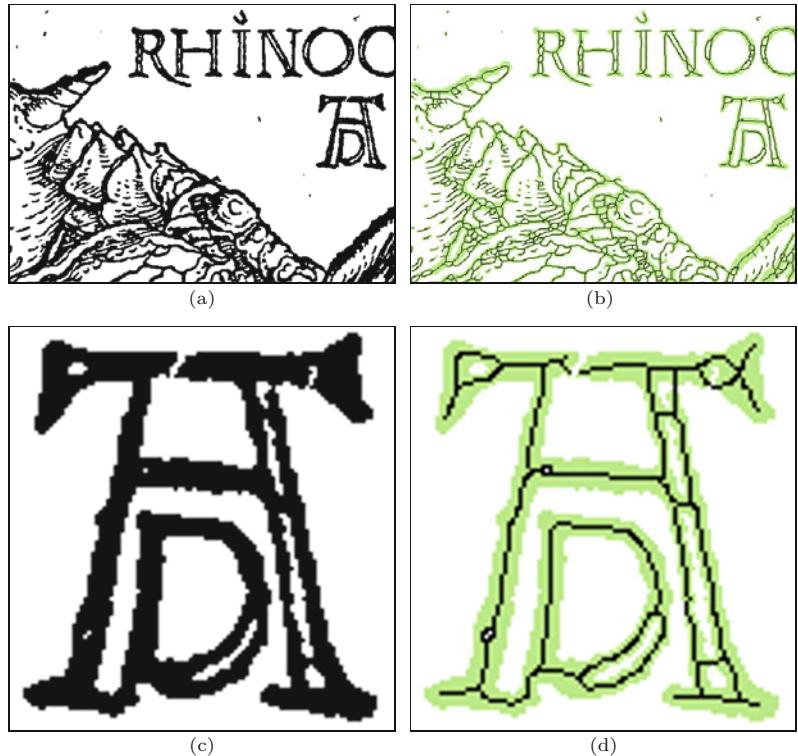
Alg. 9.3

Thinning algorithm by Zhang und Suen (accelerated version of Alg. 9.2). This algorithm employs a pre-calculated table of “deletion codes” (Q). Procedure $\text{GetNeighborhood}()$ has been replaced by $\text{GetNeighborhoodIndex}()$, which does not return the neighboring pixel values themselves but the associated 8-bit index c with possible values in $0, \dots, 255$ (see Fig. 9.22). For completeness, the calculation of table Q is included in procedure $\text{MakeDeletionCodeTable}()$, although this table is fixed and may be simply defined as a constant array (see Prog. 9.1).

9 MORPHOLOGICAL FILTERS

Fig. 9.21

Thinning a binary image (Alg. 9.2 or 9.3). Original image with enlarged detail (a, c) and results after thinning (b, d). The original foreground pixels are marked green, the resulting pixels are black.



Prog. 9.1

Java definition for the “deletion code” table Q (see Fig. 9.22).

```
1 static final byte[] Q = {  
2     0, 0, 0, 3, 0, 0, 3, 3, 0, 0, 0, 0, 0, 3, 0, 3, 3,  
3     0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 3, 0, 3, 1,  
4     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
5     3, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 3, 0, 3, 1,  
6     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
7     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
8     3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
9     3, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 1, 0, 1, 0,  
10    0, 3, 0, 3, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 3,  
11    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,  
12    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
13    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
14    3, 3, 0, 3, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2,  
15    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
16    3, 3, 0, 3, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
17    3, 2, 0, 2, 0, 0, 0, 0, 3, 2, 0, 0, 1, 0, 0, 0  
18};
```

9.4.3 Java Implementation

The complete Java source code for the morphological operations on binary images is available online as part of the `imagingbook`⁴ library.

⁴ Package `imagingbook.pub.morphology`.



9.4 THINNING (SKELETONIZATION)

Fig. 9.22

“Deletion codes” for the 256 possible binary 8-neighborhoods tabulated in map $Q(c)$ of Alg. 9.3. $\square = 0$ and $\blacksquare = 1$ denote background and foreground pixels, respectively. The 2-bit codes are color coded as indicated at the bottom.

BinaryMorphologyFilter class

This class implements several morphological operators for binary images of type `ByteProcessor`. It defines the sub-classes `Box` and `Disk` with different structuring elements. The class provides the following constructors:

BinaryMorphologyFilter ()
Creates a morphological filter with a (default) structuring element of size 3×3 as depicted in Fig. 9.11(b).

BinaryMorphologyFilter (int [] [] H)
Creates a morphological filter with a structuring element specified by the 2D array H, which may contain 0/1 values only (all values > 0 are treated as 1).

BinaryMorphologyFilter.Box (int rad)
Creates a morphological filter with a square structuring element of radius $\text{rad} \geq 1$ and side length $2 \cdot \text{rad} + 1$ pixels.

BinaryMorphologyFilter.Disk (double rad)
Creates a morphological filter with a disk-shaped structuring element with radius $\text{rad} \geq 1$ and diameter $2 \cdot \text{round}(\text{rad}) + 1$ pixels.

The key methods⁵ of `BinaryMorphologyFilter` are:

void applyTo (ByteProcessor I, OpType op)
Destructively applies the morphological operator op to the image I. Possible arguments for op are `Dilate`, `Erode`, `Open`, `Close`, `Outline`, `Thin`.

void dilate (ByteProcessor I)
Performs (destructive) *dilation* on the binary image I with the initial structuring element of this filter.

void erode (ByteProcessor I)
Performs (destructive) *erosion* on the binary image I.

void open (ByteProcessor I)
Performs (destructive) *opening* on the binary image I.

void close (ByteProcessor I)
Performs (destructive) *closing* on the binary image I.

void outline (ByteProcessor I)
Performs a (destructive) *outline* operation on the binary image I using a 3×3 structuring element (see Sec. 9.2.7).

void thin (ByteProcessor I)
Performs a (destructive) *thinning* operation on the binary image I using a 3×3 structuring element (with at most $i_{\max} = 1500$ iterations, see Alg. 9.3).

void thin (ByteProcessor I, int iMax)
Performs a thinning operation with at most iMax iterations (see Alg. 9.3).

int thinOnce (ByteProcessor I)
Performs a single iteration of the thinning operation and returns the number of pixel deletions (see Alg. 9.3).

The methods listed here *always* treat image pixels with value 0 as background and values > 0 as foreground. Unlike ImageJ's built-in implementation of morphological operations (described in Sec. 9.4.4), the display lookup table (LUT, typically only used for display purposes) of the image is *not* taken into account at all.

⁵ See the online documentation for additional methods.

```

1 import ij.ImagePlus;
2 import ij.plugin.filter.PluginFilter;
3 import ij.process.ByteProcessor;
4 import ij.process.ImageProcessor;
5 import imagingbook.pub.morphology.BinaryMorphologyFilter;
6 import imagingbook.pub.morphology.BinaryMorphologyFilter.
    OpType;
7
8 public class Bin_Dilate_Disk_Demo implements PluginFilter {
9     static double radius = 5.0;
10    static OpType op = OpType.Dilate; // Erode, Open, Close, ...
11
12    public int setup(String arg, ImagePlus imp) {
13        return DOES_8G;
14    }
15
16    public void run(ImageProcessor ip) {
17        BinaryMorphologyFilter bmf =
18            new BinaryMorphologyFilter.Disk(radius);
19        bmf.applyTo((ByteProcessor) ip, op);
20    }
21 }

```

9.4 THINNING (SKELETONIZATION)

Prog. 9.2

Example for using class `BinaryMorphologyFilter` (see Sec. 9.4.3) inside a ImageJ plugin. The actual filter operator is instantiated in line 18 and subsequently (in line 19) applied to the image `ip` of type `ByteProcessor`. Available operations (`OpType`) are `Dilate`, `Erode`, `Open`, `Close`, `Outline` and `Thin`. Note that the results depend strictly on the pixel values of the input image, with values 0 taken as background and values > 0 taken as foreground. The display lookup-table (LUT) is irrelevant.

The example in Prog. 9.2 shows the use of class `BinaryMorphologyFilter` in a complete ImageJ plugin that performs dilation with a disk-shaped structuring element of radius 5 (pixel units). Other examples can be found in the online code repository.

9.4.4 Built-in Morphological Operations in ImageJ

Apart from the implementation described in the previous section, the ImageJ API provides built-in methods for basic morphological operations, such as `dilate()` and `erode()`. These methods use a 3×3 structuring element (analogous to Fig. 9.11(b)) and are only defined for images of type `ByteProcessor` and `ColorProcessor`. In the case of RGB color images (`ColorProcessor`) the morphological operation is applied individually to the three color channels. All these and other morphological operations can be applied interactively through ImageJ's `Process` \triangleright `Binary` menu (see Fig. 9.23(a)).

Note that ImageJ's `dilate()` and `erode()` methods use the current settings of display lookup table (LUT) to discriminate between background and foreground pixels. Thus the results of morphological operations depend not only on the stored pixel values but how they are being displayed (in addition to the settings in `Process` \triangleright `Binary` \triangleright `Options...`, see Fig. 9.23(b)).⁶ It is therefore recommended to use the methods (defined for `ByteProcessor` only)

```

dilate(int count, int background),
erode(int count, int background)

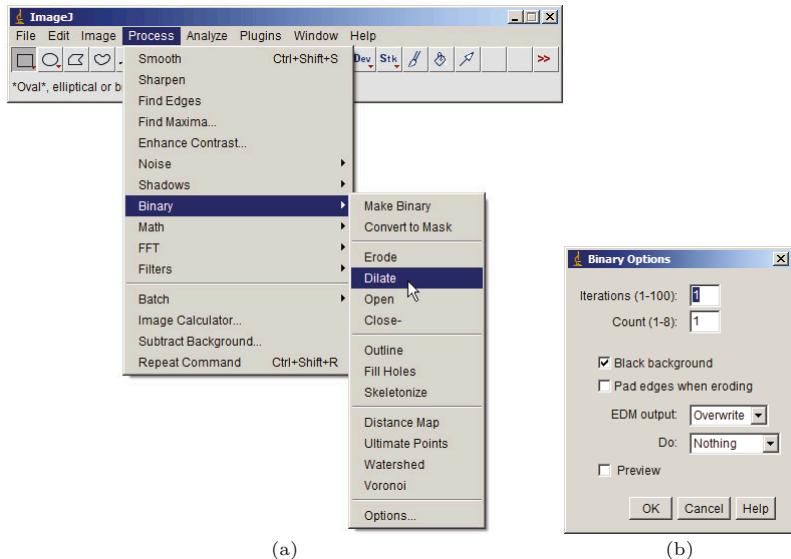
```

⁶ These dependencies may be quite confusing because the same program will produce different results under different user setups.

9 MORPHOLOGICAL FILTERS

Fig. 9.23

Morphological operations in ImageJ's built-in standard menu **Process** ▷ **Binary** (a) and optional settings with **Process** ▷ **Binary** ▷ **Options...** (b). The choice “Black background” specifies if background pixels are bright or dark, which is taken into account by ImageJ's morphological operations.



instead, since they provide explicit control of the background pixel value and are thus independent from other settings. ImageJ's `ByteProcessor` class defines additional methods for morphological operations on binary images, such as `outline()` and `skeletonize()`. The method `outline()` implements the extraction of region boundaries using an 8-neighborhood structuring element, as described in Sec. 9.2.7. The method `skeletonize()`, on the other hand, implements a thinning process similar to Alg. 9.3.

9.5 Grayscale Morphology

Morphological operations are not confined to binary images but are also for intensity (grayscale) images. In fact, the definition of grayscale morphology is a *generalization* of binary morphology, with the binary OR and AND operators replaced by the arithmetic MAX and MIN operators, respectively. As a consequence, procedures designed for grayscale morphology can also perform binary morphology (but not the other way around).⁷ In the case of color images, the grayscale operations are usually applied individually to each color channel.

9.5.1 Structuring Elements

Unlike in the binary scheme, the structuring elements for grayscale morphology are not defined as point sets but as real-valued 2D functions, that is,

$$H(i, j) \in \mathbb{R}, \quad \text{for } (i, j) \in \mathbb{Z}^2. \quad (9.30)$$

The values in H may be negative or zero. Notice, however, that, in contrast to linear convolution (Sec. 5.3.1), zero elements in grayscale

⁷ ImageJ provides a single implementation of morphological operations that handles both binary and grayscale images (see Sec. 9.4.4).

morphology generally *do* contribute to the result.⁸ The design of structuring elements for grayscale morphology must therefore distinguish explicitly between cells containing the value 0 and empty (“don’t care”) cells, for example,

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 0 \\ \hline 1 & 2 & 1 \\ \hline 0 & 1 & 0 \\ \hline \end{array} \neq \begin{array}{|c|c|c|} \hline & 1 & \\ \hline 1 & 2 & 1 \\ \hline & 1 & \\ \hline \end{array}. \quad (9.31)$$

9.5 GRayscale MORPHOLOGY

9.5.2 Dilation and Erosion

The result of grayscale *dilation* $I \oplus H$ is defined as the *maximum* of the values in H added to the values of the current subimage of I , that is,

$$(I \oplus H)(u, v) = \max_{(i,j) \in H} (I(u+i, v+j) + H(i, j)). \quad (9.32)$$

Similarly, the result of grayscale *erosion* is the *minimum* of the differences,

$$(I \ominus H)(u, v) = \min_{(i,j) \in H} (I(u+i, v+j) - H(i, j)). \quad (9.33)$$

[Figures 9.24](#) and [9.25](#) demonstrate the basic process of grayscale dilation and erosion, respectively, on a simple example.

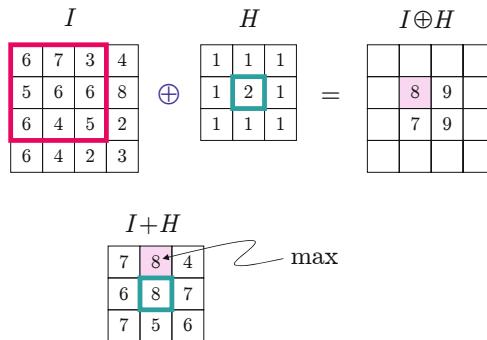


Fig. 9.24

Grayscale dilation $I \oplus H$. The 3×3 pixel structuring element H is placed on the image I in the upper left position. Each value of H is added to the corresponding element of I ; the intermediate result $(I + H)$ for this particular position is shown below. Its maximum value $8 = 7 + 1$ is inserted into the result $(I \oplus H)$ at the current position of the filter origin. The results for three other filter positions are also shown.

In general, either operation may produce *negative* results that must be considered if the range of pixel values is restricted, for example, by clamping the results (see Ch. 4, Sec. 4.1.2). Some examples of grayscale dilation and erosion on natural images using disk-shaped structuring elements of various sizes are shown in [Fig. 9.26](#). Figure 9.28 demonstrates the same operations with some freely designed structuring elements.

9.5.3 Grayscale Opening and Closing

Opening and closing on grayscale images are defined, identical to the binary case (Eqns. (9.21) and (9.22)), as operations composed

⁸ While a zero coefficient in a linear convolution matrix simply means that the corresponding image pixel is ignored.

9 MORPHOLOGICAL FILTERS

Fig. 9.25

Grayscale erosion $I \ominus H$. The 3×3 pixel structuring element H is placed on the image I in the upper left position. Each value of H is subtracted from the corresponding element of I ; the intermediate result $(I - H)$ for this particular position is shown below. Its minimum value $3 - 1 = 2$ is inserted into the result $(I \ominus H)$ at the current position of the filter origin.

The results for three other filter positions are also shown.

$$\begin{array}{c}
 I \\
 \begin{array}{|c|c|c|c|} \hline
 6 & 7 & 3 & 4 \\ \hline
 5 & 6 & 6 & 8 \\ \hline
 6 & 4 & 5 & 2 \\ \hline
 6 & 4 & 2 & 3 \\ \hline
 \end{array}
 \end{array}
 \ominus
 \begin{array}{c}
 H \\
 \begin{array}{|c|c|c|} \hline
 1 & 1 & 1 \\ \hline
 1 & 2 & 1 \\ \hline
 1 & 1 & 1 \\ \hline
 \end{array}
 \end{array}
 =
 \begin{array}{c}
 I \ominus H \\
 \begin{array}{|c|c|c|c|} \hline
 & & & \\ \hline
 & 2 & 1 & \\ \hline
 & 1 & 1 & \\ \hline
 & & & \\ \hline
 \end{array}
 \end{array}$$

$$\begin{array}{c}
 I - H \\
 \begin{array}{|c|c|c|} \hline
 5 & 6 & 2 \\ \hline
 4 & 4 & 5 \\ \hline
 5 & 3 & 4 \\ \hline
 \end{array}
 \end{array}
 \xrightarrow{\text{min}}$$

Original



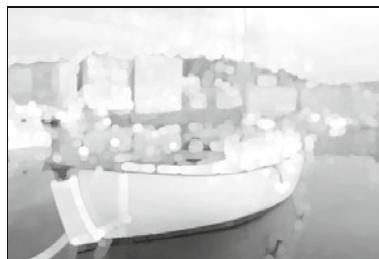
Dilation



Erosion



(a) $r = 2.5$



(b) $r = 5.0$



(c) $r = 10.0$

Opening



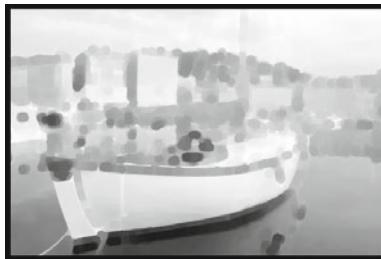
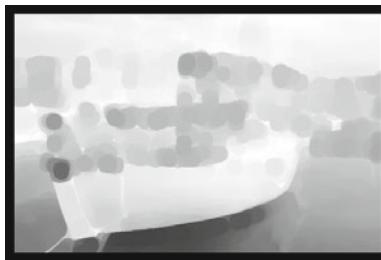
Closing



9.6 EXERCISES

Fig. 9.27

Grayscale opening and closing with disk-shaped structuring elements. The radius r of the structuring element is 2.5 (a), 5.0 (b), and 10.0 (c).

(a) $r = 2.5$ (b) $r = 5.0$ (c) $r = 10.0$ 

of dilation and erosion with the same structuring element. Some examples are shown in Fig. 9.27 for disk-shaped structuring elements and in Fig. 9.29 for various nonstandard structuring elements. Notice that interesting effects can be obtained, particularly from structuring elements resembling the shape of brush or other stroke patterns.

As mentioned in Sec. 9.4.4, the morphological operations available in ImageJ can be applied to binary images as well as grayscale images. In addition, several additional plugins and complete morphology packages are available online,⁹ including the morphology operators by Gabriel Landini and the Grayscale Morphology package by Dimitar Prodanov, which allows structuring elements to be interactively specified (a modified version was used for some examples in this chapter).

9.6 Exercises

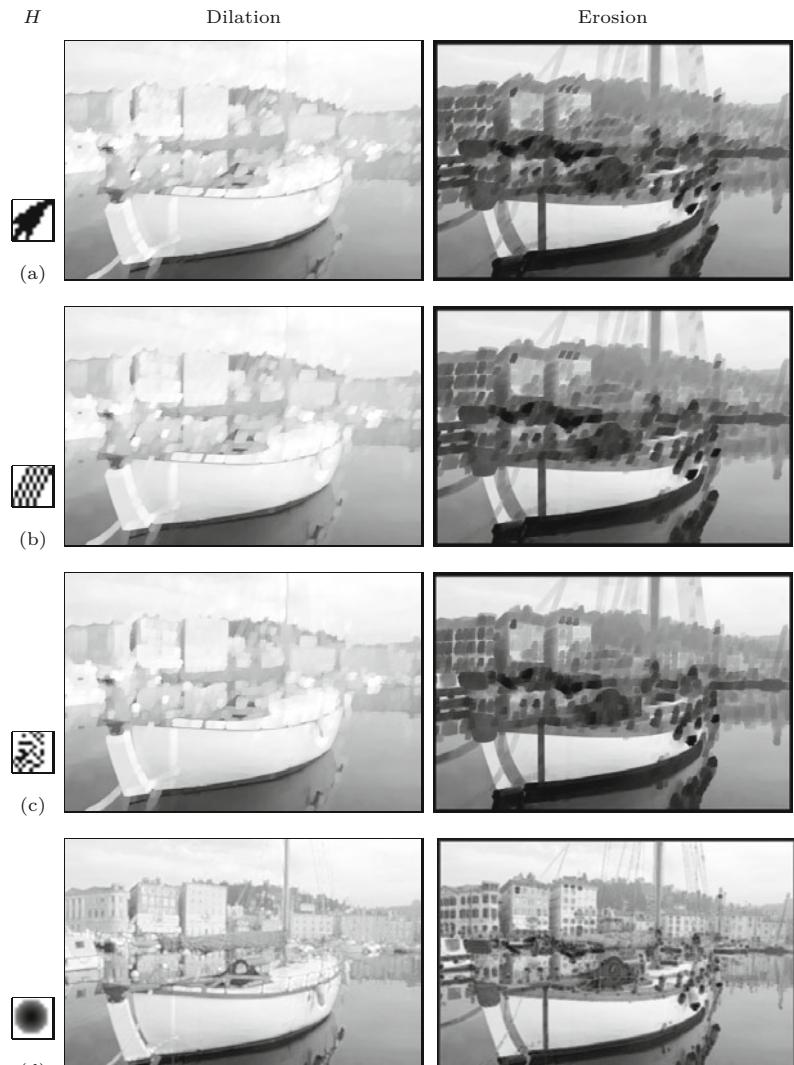
Exercise 9.1. Manually calculate the results of dilation and erosion for the following image I and the structuring elements H_1 and H_2 :

⁹ See <http://rsb.info.nih.gov/ij/plugins/>.

9 MORPHOLOGICAL FILTERS

Fig. 9.28

Grayscale dilation and erosion with various free-form structuring elements.



$$I = \begin{array}{|c|c|c|c|c|c|} \hline & \color{red}{\bullet} & \color{gray}{\bullet} & \color{gray}{\bullet} & \color{gray}{\bullet} & \color{black}{\bullet} \\ \hline \color{gray}{\bullet} & \color{gray}{\bullet} & \color{black}{\bullet} & \color{black}{\bullet} & \color{black}{\bullet} & \color{black}{\bullet} \\ \hline \color{black}{\bullet} & \color{black}{\bullet} & \color{black}{\bullet} & \color{black}{\bullet} & \color{black}{\bullet} & \color{black}{\bullet} \\ \hline & \color{gray}{\bullet} & \color{gray}{\bullet} & \color{black}{\bullet} & \color{black}{\bullet} & \color{gray}{\bullet} \\ \hline & & \color{gray}{\bullet} & & \color{gray}{\bullet} & \\ \hline \end{array}$$

$$H_1 = \begin{array}{|c|c|c|} \hline \color{gray}{\bullet} & & \color{black}{\bullet} \\ \hline & \color{red}{\bullet} & \color{gray}{\bullet} \\ \hline & & \color{gray}{\bullet} \\ \hline \end{array}$$

$$H_2 = \begin{array}{|c|c|c|} \hline & \color{black}{\bullet} & \\ \hline \color{black}{\bullet} & \color{red}{\bullet} & \color{black}{\bullet} \\ \hline & & \color{black}{\bullet} \\ \hline \end{array}$$

Exercise 9.2. Assume that a binary image I contains unwanted foreground spots with a maximum diameter of 5 pixels that should be removed without damaging the remaining structures. Design a suitable morphological procedure, and evaluate its performance on appropriate test images.

Exercise 9.3. Investigate if the results of the thinning operation described in Alg. 9.2 (and implemented by the `thin()` method of class `BinaryMorphologyFilter`) are invariant against rotating the image

H

Opening

Closing

9.6 EXERCISES



Fig. 9.29

Grayscale opening and closing with various free-form structuring elements.

by 90° and horizontal or vertical mirroring. Use appropriate test images to see if the results are identical.

Exercise 9.4. Show that, in the special case of the structuring elements with the contents

$$\begin{array}{|c|c|c|} \hline \bullet & \bullet & \bullet \\ \hline \bullet & \textcolor{red}{\bullet} & \bullet \\ \hline \bullet & \bullet & \bullet \\ \hline \end{array}$$

for *binary*

and

$$\begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & \textcolor{red}{0} & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$$

for *grayscale* images,

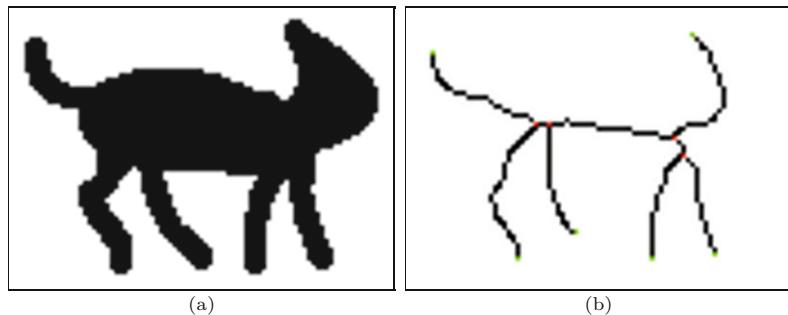
dilation is equivalent to a 3×3 pixel maximum filter and erosion is equivalent to a 3×3 pixel minimum filter (see Ch. 5, Sec. 5.4.1).

Exercise 9.5. Thinning can be applied to extract the “skeleton” of a binary region, which in turn can be used to characterize the shape of the region. A common approach is to partition the skeleton into a graph, consisting of nodes and connecting segments, as a

9 MORPHOLOGICAL FILTERS

Fig. 9.30

Segmentation of a region skeleton. Original binary image (a) and the skeleton obtained by thinning (b). Terminal nodes are marked green, connecting (inner) nodes are marked red.



shape representation (see [Fig. 9.30](#) for an example). Use ImageJ’s `skeletonize()` method or the `thin()` methode of class `BinaryMorphologyFilter` (see Sec. 9.4.3) to generate the skeleton, then locate and mark the connecting and terminal nodes of this structure. Define precisely the properties of each type of node and use this definition in your implementation. Test your implementation on different examples. How would you generally judge the robustness of this approach as a 2D shape representation?

Regions in Binary Images

In a binary image, pixels can take on exactly one of two values. These values are often thought of as representing the “foreground” and “background” in the image, even though these concepts often are not applicable to natural scenes. In this chapter we focus on connected regions in images and how to isolate and describe such structures.

Let us assume that our task is to devise a procedure for finding the number and type of objects contained in an image as shown in Fig. 10.1. As long as we continue to consider each pixel in isolation, we will not be able to determine how many objects there are overall in the image, where they are located, and which pixels belong to which objects. Therefore our first step is to find each object by grouping together all the pixels that belong to it. In the simplest case, an object is a group of touching foreground pixels, that is, a connected *binary region* or “component”.

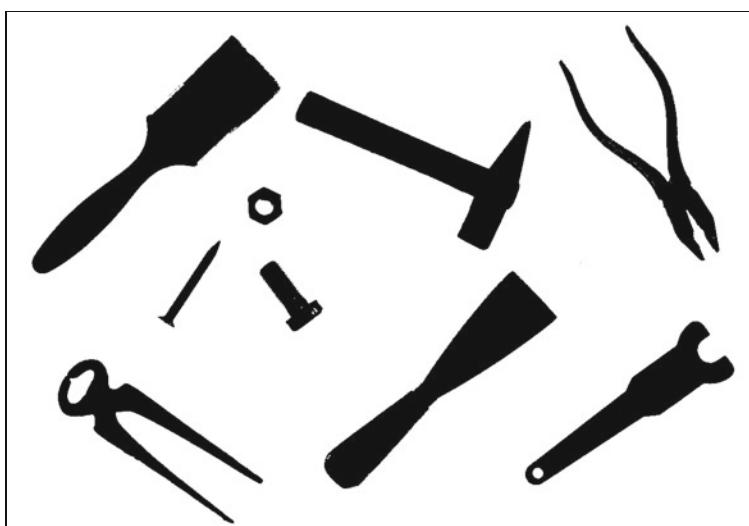


Fig. 10.1
Binary image with nine components. Each component corresponds to a connected region of (black) foreground pixels.

10.1 Finding Connected Image Regions

In the search for binary regions, the most important tasks are to find out which pixels belong to which regions, how many regions are in the image, and where these regions are located. These steps usually take place as part of a process called *region labeling* or *region coloring*. During this process, neighboring pixels are pieced together in a stepwise manner to build regions in which all pixels within that region are assigned a unique number (“label”) for identification. In the following sections, we describe two variations on this idea. In the first method, region marking through *flood filling*, a region is filled in all directions starting from a single point or “seed” within the region. In the second method, *sequential region marking*, the image is traversed from top to bottom, marking regions as they are encountered. In Sec. 10.2.2, we describe a third method that combines two useful processes, region labeling and contour finding, in a single algorithm.

Independent of which of these methods we use, we must first settle on either the 4- or 8-connected definition of neighboring (see Ch. 9, Fig. 9.5) for determining when two pixels are “connected” to each other, since under each definition we can end up with different results. In the following region-marking algorithms, we use the following convention: the original binary image $I(u, v)$ contains the values 0 and 1 to mark the *background* and *foreground*, respectively; any other value is used for numbering (labeling) the regions, that is, the pixel values are

$$I(u, v) = \begin{cases} 0 & \text{background,} \\ 1 & \text{foreground,} \\ 2, 3, \dots & \text{region label.} \end{cases} \quad (10.1)$$

10.1.1 Region Labeling by Flood Filling

The underlying algorithm for region marking by *flood filling* is simple: search for an unmarked foreground pixel and then fill (visit and mark) all the rest of the neighboring pixels in its region. This operation is called a “flood fill” because it is as if a flood of water erupts at the start pixel and flows out across a flat region. There are various methods for carrying out the fill operation that ultimately differ in how to select the coordinates of the next pixel to be visited during the fill. We present three different ways of performing the `FloodFill()` procedure: a recursive version, an iterative *depth-first* version, and an iterative *breadth-first* version (see Alg. 10.1):

A. Recursive Flood Filling: The recursive version (Alg. 10.1, line 8) does not make use of explicit data structures to keep track of the image coordinates but uses the local variables that are implicitly allocated by recursive procedure calls.¹ Within each region, a tree structure, rooted at the starting point, is defined by the neighborhood relation between pixels. The recursive step corresponds to a *depth-first traversal* [54] of this tree and results

¹ In Java, and similar imperative programming languages such as C and C++, local variables are automatically stored on the *call stack* at each procedure call and restored from the stack when the procedure returns.

1: **RegionLabeling**(I)

Input: I , an integer-valued image with initial values $0 = \text{background}$, $1 = \text{foreground}$. Returns nothing but modifies the image I .

```

2:    $label \leftarrow 2$             $\triangleright$  value of the next label to be assigned
3:   for all image coordinates  $u, v$  do
4:     if  $I(u, v) = 1$  then            $\triangleright$  a foreground pixel
5:       FloodFill( $I, u, v, label$ )     $\triangleright$  any of the 3 versions below
6:        $label \leftarrow label + 1$ .
7:   return



---


8: FloodFill( $I, u, v, label$ )           $\triangleright$  Recursive Version
9:   if  $u, v$  is within the image boundaries and  $I(u, v) = 1$  then
10:     $I(u, v) \leftarrow label$ 
11:    FloodFill( $I, u+1, v, label$ )     $\triangleright$  recursive call to FloodFill()
12:    FloodFill( $I, u, v+1, label$ )
13:    FloodFill( $I, u, v-1, label$ )
14:    FloodFill( $I, u-1, v, label$ )
15:   return



---


16: FloodFill( $I, u, v, label$ )           $\triangleright$  Depth-First Version
17:    $S \leftarrow ()$                     $\triangleright$  create an empty stack  $S$ 
18:    $S \leftarrow (u, v) \cup S$          $\triangleright$  push seed coordinate  $(u, v)$  onto  $S$ 
19:   while  $S \neq ()$  do            $\triangleright$  while  $S$  is not empty
20:      $(x, y) \leftarrow \text{GetFirst}(S)$ 
21:      $S \leftarrow \text{Delete}((x, y), S)$      $\triangleright$  pop first coordinate off the stack
22:     if  $x, y$  is within the image boundaries and  $I(x, y) = 1$  then
23:        $I(x, y) \leftarrow label$ 
24:        $S \leftarrow (x+1, y) \cup S$          $\triangleright$  push  $(x+1, y)$  onto  $S$ 
25:        $S \leftarrow (x, y+1) \cup S$          $\triangleright$  push  $(x, y+1)$  onto  $S$ 
26:        $S \leftarrow (x, y-1) \cup S$          $\triangleright$  push  $(x, y-1)$  onto  $S$ 
27:        $S \leftarrow (x-1, y) \cup S$          $\triangleright$  push  $(x-1, y)$  onto  $S$ 
28:   return



---


29: FloodFill( $I, u, v, label$ )           $\triangleright$  Breadth-First Version
30:    $Q \leftarrow ()$                     $\triangleright$  create an empty queue  $Q$ 
31:    $Q \leftarrow Q \cup (u, v)$          $\triangleright$  append seed coordinate  $(u, v)$  to  $Q$ 
32:   while  $Q \neq ()$  do            $\triangleright$  while  $Q$  is not empty
33:      $(x, y) \leftarrow \text{GetFirst}(Q)$ 
34:      $Q \leftarrow \text{Delete}((x, y), Q)$      $\triangleright$  dequeue first coordinate
35:     if  $x, y$  is within the image boundaries and  $I(x, y) = 1$  then
36:        $I(x, y) \leftarrow label$ 
37:        $Q \leftarrow Q \cup (x+1, y)$          $\triangleright$  append  $(x+1, y)$  to  $Q$ 
38:        $Q \leftarrow Q \cup (x, y+1)$          $\triangleright$  append  $(x, y+1)$  to  $Q$ 
39:        $Q \leftarrow Q \cup (x, y-1)$          $\triangleright$  append  $(x, y-1)$  to  $Q$ 
40:        $Q \leftarrow Q \cup (x-1, y)$          $\triangleright$  append  $(x-1, y)$  to  $Q$ 
41:   return
```

10.1 FINDING CONNECTED IMAGE REGIONS

Alg. 10.1

Region marking by *flood filling*. The binary input image I uses the value 0 for background pixels and 1 for foreground pixels. Unmarked foreground pixels are searched for, and then the region to which they belong is filled. Procedure **FloodFill()** is defined in three different versions: *recursive*, *depth-first* and *breadth-first*.

in very short and elegant program code. Unfortunately, since the maximum depth of the recursion—and thus the size of the required stack memory—is proportional to the size of the region, stack memory is quickly exhausted. Therefore this method is risky and really only practical for very small images.

- B. Iterative Flood Filling (*depth-first*):** Every recursive algorithm can also be reformulated as an iterative algorithm (Alg. 10.1, line 16) by implementing and managing its own *stacks*. In this case, the stack records the “open” (that is, the adjacent but not yet visited) elements. As in the recursive version (A), the corresponding tree of pixels is traversed in *depth-first* order. By making use of its own dedicated stack (which is created in the much larger *heap* memory), the depth of the tree is no longer limited to the size of the call stack.
- C. Iterative Flood Filling (*breadth-first*):** In this version, pixels are traversed in a way that resembles an expanding wave front propagating out from the starting point (Alg. 10.1, line 29). The data structure used to hold the as yet unvisited pixel coordinates is in this case a *queue* instead of a stack, but otherwise it is identical to version B.

Java implementation

The recursive version (A) of the algorithm is an exact blueprint of the Java implementation. However, a normal Java runtime environment does not support more than about 10,000 recursive calls of the `FloodFill()` procedure (Alg. 10.1, line 8) before the memory allocated for the call stack is exhausted. This is only sufficient for relatively small images with fewer than approximately 200×200 pixels.

Program 10.1 (line 1–17) gives the complete Java implementation for both variants of the iterative `FloodFill()` procedure. The stack (S) in the *depth-first* Version (B) and the queue (Q) in the *breadth-first* variant (C) are both implemented as instances of type `LinkedList`.² `<Point>` is specified as a type parameter for both generic container classes so they can only contain objects of type `Point`.³

Figure 10.2 illustrates the progress of the region marking in both variants within an example region, where the start point (i.e., seed point), which would normally lie on a contour edge, has been placed arbitrarily within the region in order to better illustrate the process. It is clearly visible that the *depth-first* method first explores *one* direction (in this case horizontally to the left) completely (that is, until it reaches the edge of the region) and only then examines the remaining directions. In contrast the *breadth-first* method markings proceed outward, layer by layer, equally in all directions.

Due to the way exploration takes place, the memory requirement of the *breadth-first* variant of the *flood-fill* version is generally much lower than that of the *depth-first* variant. For example, when flood filling the region in Fig. 10.2 (using the implementation given Prog. 10.1), the stack in the *depth-first* variant grows to a maximum of 28,822 elements, while the queue used by the *breadth-first* variant never exceeds a maximum of 438 nodes.

² The class `LinkedList` is part of Java’s *collections framework*.

³ Note that the depth-first and breadth-first implementations in Prog. 10.1 typically run slower than the recursive version described in Alg. 10.1, since they allocate (and immediately discard) large numbers of `Point` objects. A better solution is to use a queue or stack with elements of a primitive type (e.g., `int`) instead. See also Exercise 10.3.

Depth-first version (using a *stack*):

```
1 void floodFill(int u, int v, int label) {
2     Deque<Point> S = new LinkedList<Point>(); // stack S
3     S.push(new Point(u, v));
4     while (!S.isEmpty()) {
5         Point p = S.pop();
6         int x = p.x;
7         int y = p.y;
8         if ((x >= 0) && (x < width) && (y >= 0) && (y < height)
9             && ip.getPixel(x, y) == 1) {
10            ip.putPixel(x, y, label);
11            S.push(new Point(x + 1, y));
12            S.push(new Point(x, y + 1));
13            S.push(new Point(x, y - 1));
14            S.push(new Point(x - 1, y));
15        }
16    }
17 }
```

10.1 FINDING CONNECTED IMAGE REGIONS

Prog. 10.1

Java implementation of iterative flood filling (*depth-first* and *breadth-first* variants).

Breadth-first version (using a *queue*):

```
18 void floodFill(int u, int v, int label) {
19     Queue<Point> Q = new LinkedList<Point>(); // queue Q
20     Q.add(new Point(u, v));
21     while (!Q.isEmpty()) {
22         Point p = Q.remove(); // get the next point to process
23         int x = p.x;
24         int y = p.y;
25         if ((x >= 0) && (x < width) && (y >= 0) && (y < height)
26             && ip.getPixel(x, y) == 1) {
27            ip.putPixel(x, y, label);
28            Q.add(new Point(x + 1, y));
29            Q.add(new Point(x, y + 1));
30            Q.add(new Point(x, y - 1));
31            Q.add(new Point(x - 1, y));
32        }
33    }
34 }
```

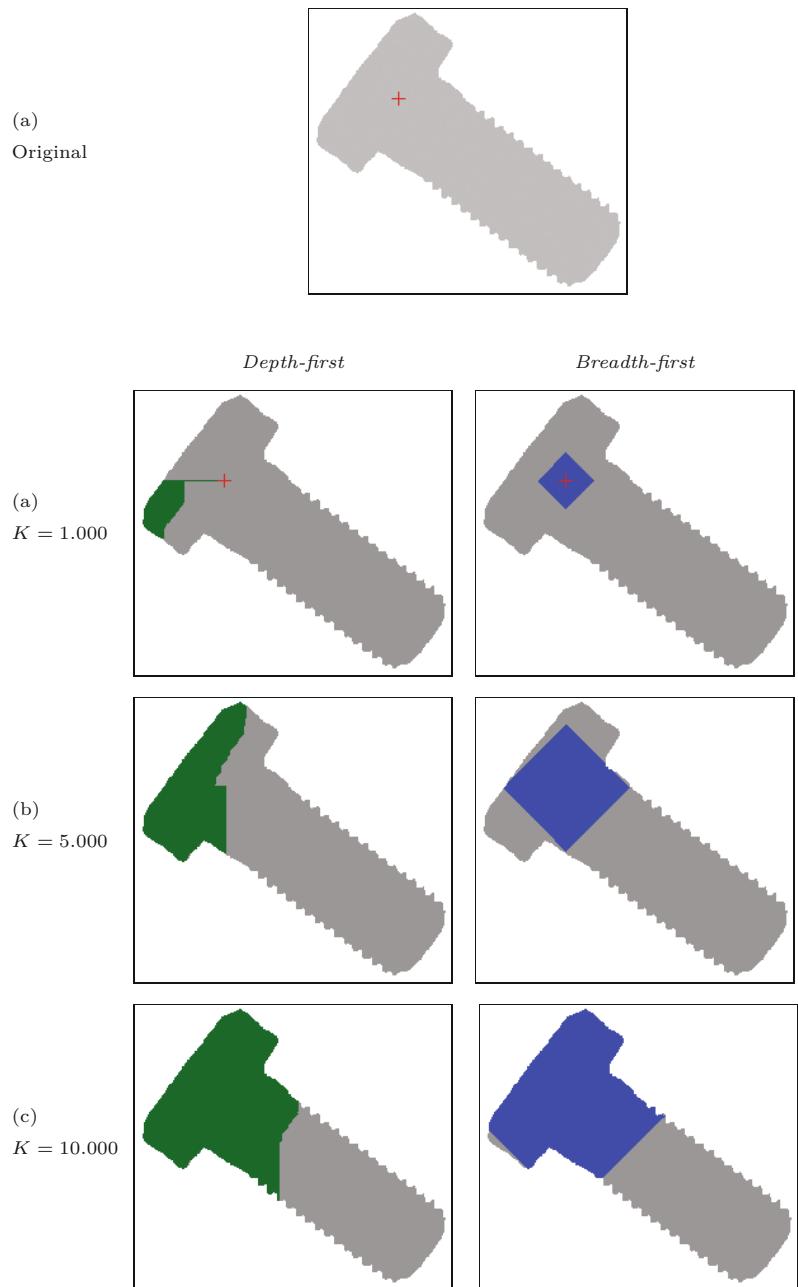
10.1.2 Sequential Region Labeling

Sequential region marking is a classical, nonrecursive technique that is known in the literature as “region labeling”. The algorithm consists of two steps: (1) preliminary labeling of the image regions and (2) resolving cases where more than one label occurs (i.e., has been assigned in the previous step) in the same connected region. Even though this algorithm is relatively complex, especially its second stage, its moderate memory requirements make it a good choice under limited memory conditions. However, this is not a major issue on modern computers and thus, in terms of overall efficiency, sequential labeling offers no clear advantage over the simpler methods described earlier. The sequential technique is nevertheless interesting (not only from a historic perspective) and inspiring. The complete process is summarized in Alg. 10.2, with the following main steps:

10 REGIONS IN BINARY IMAGES

Fig. 10.2

Iterative flood filling—comparison between the *depth-first* and *breadth-first* approach. The starting point, marked + in the top two image (a), was arbitrarily chosen. Intermediate results of the *flood fill* process after 1000 (a), 5000 (b), and 10,000 (c) marked pixels are shown. The image size is 250×242 pixels.



Step 1: Initial labeling

In the first stage of region labeling, the image is traversed from top left to bottom right sequentially to assign a preliminary label to every foreground pixel. Depending on the definition of neighborhood (either 4- or 8-connected) used, the following neighbors in the direct vicinity of each pixel must be examined (\times marks the current pixel at the position (u, v)):

1: **SequentialLabeling(I)**

Input: I , an integer-valued image with initial values $0 = \text{background}$, $1 = \text{foreground}$. Returns nothing but modifies the image I .

Step 1 – Assign initial labels:

```

2:  $(M, N) \leftarrow \text{Size}(I)$ 
3:  $\text{label} \leftarrow 2$                                  $\triangleright$  value of the next label to be assigned
4:  $C \leftarrow ()$                                   $\triangleright$  empty list of label collisions
5: for  $v \leftarrow 0, \dots, N - 1$  do
6:   for  $u \leftarrow 0, \dots, M - 1$  do
7:     if  $I(u, v) = 1$  then       $\triangleright I(u, v)$  is a foreground pixel
8:        $\mathcal{N} \leftarrow \text{GetNeighbors}(I, u, v)$            $\triangleright$  see Eqn. 10.2
9:       if  $N_i = 0$  for all  $N_i \in \mathcal{N}$  then
10:         $I(u, v) \leftarrow \text{label}.$ 
11:         $\text{label} \leftarrow \text{label} + 1.$ 
12:       else if exactly one  $N_j \in \mathcal{N}$  has a value  $> 1$  then
13:         set  $I(u, v) \leftarrow N_j$ 
14:       else if more than one  $N_k \in \mathcal{N}$  have values  $> 1$  then
15:          $I(u, v) \leftarrow N_k$        $\triangleright$  select one  $N_k > 1$  as the new
16:           label
17:         for all  $N_l \in \mathcal{N}$ , with  $l \neq k$  and  $N_l > 1$  do
18:            $C \leftarrow C \cup (N_k, N_l)$   $\triangleright$  register collision  $(N_k, N_l)$ 

```

Remark: The image I now contains labels $0, 2, \dots, \text{label}-1$.

Step 2 – Resolve label collisions:

Create a partitioning of the label set (sequence of 1-element sets):

```

18:  $R \leftarrow (\{2\}, \{3\}, \{4\}, \dots, \{\text{label}-1\})$ 
19: for all collisions  $(A, B)$  in  $C$  do
20:   Find the sets  $R(a), R(b)$  holding labels  $A, B$ :
21:    $a \leftarrow$  index of the set  $R(a)$  that contains label  $A$ 
22:    $b \leftarrow$  index of the set  $R(b)$  that contains label  $B$ 
23:   if  $a \neq b$  then       $\triangleright A$  and  $B$  are contained in different sets
24:      $R(a) \leftarrow R(a) \cup R(b)$      $\triangleright$  merge elements of  $R(b)$  into  $R(a)$ 
      $R(b) \leftarrow \{\}$ 

```

Remark: All *equivalent* labels (i.e., all labels of pixels in the same connected component) are now contained in the same subset of R .

25: **Step 3: Relabel the image:**

```

26: for all  $(u, v) \in M \times N$  do
27:   if  $I(u, v) > 1$  then       $\triangleright$  this is a labeled foreground pixel
28:      $j \leftarrow$  index of the set  $R(j)$  that contains label  $I(u, v)$ 
29:     Choose a representative element  $k$  from the set  $R(j)$ :
30:      $k \leftarrow \min(R(j))$        $\triangleright$  e.g., pick the minimum value
31:      $I(u, v) \leftarrow k$            $\triangleright$  replace the image label

```

31: **return**

10.1 FINDING CONNECTED IMAGE REGIONS

Alg. 10.2

Sequential region labeling. The binary input image I uses the value $I(u, v) = 0$ for background pixels and $I(u, v) = 1$ for foreground (region) pixels. The resulting labels have the values $2, \dots, \text{label}-1$.

$$\mathcal{N}_4 = \begin{array}{|c|c|c|} \hline & N_1 & \\ \hline N_2 & \times & N_0 \\ \hline & N_3 & \\ \hline \end{array} \quad \text{or} \quad \mathcal{N}_8 = \begin{array}{|c|c|c|} \hline & N_3 & N_2 & N_1 \\ \hline N_4 & \times & N_0 \\ \hline & N_5 & N_6 & N_7 \\ \hline \end{array}. \quad (10.2)$$

When using the 4-connected neighborhood \mathcal{N}_4 , only the two neighbors $N_1 = I(u-1, v)$ and $N_2 = I(u, v-1)$ need to be considered, but when using the 8-connected neighborhood \mathcal{N}_8 , all four neighbors $N_1 \dots N_4$ must be examined. In the following examples (Figs. 10.3–10.5), we use an 8-connected neighborhood and a very simple test image (Fig. 10.3(a)) to demonstrate the sequential region labeling process.

Propagating region labels

Again we assume that, in the image, the value $I(u, v) = 0$ represents background pixels and the value $I(u, v) = 1$ represents foreground pixels. We will also consider neighboring pixels that lie outside of the image matrix (e.g., on the array borders) to be part of the background. The neighborhood region $\mathcal{N}(u, v)$ is slid over the image horizontally and then vertically, starting from the top left corner. When the current image element $I(u, v)$ is a foreground pixel, it is either assigned a new region number or, in the case where one of its previously examined neighbors in $\mathcal{N}(u, v)$ was a foreground pixel, it takes on the region number of the neighbor. In this way, existing region numbers propagate in the image from the left to the right and from the top to the bottom, as shown in (Fig. 10.3(b–c)).

Label collisions

In the case where two or more neighbors have labels belonging to *different* regions, then a label collision has occurred; that is, pixels within a single connected region have different labels. For example, in a U-shaped region, the pixels in the left and right arms are at first assigned different labels since it is not immediately apparent that they are actually part of a single region. The two labels will propagate down independently from each other until they eventually collide in the lower part of the “U” (Fig. 10.3(d)).

When two labels a, b collide, then we know that they are actually “equivalent”; that is, they are contained in the same image region. These collisions are registered but otherwise not dealt with during the first step. Once all collisions have been registered, they are then resolved in the second step of the algorithm. The number of collisions depends on the content of the image. There can be only a few or very many collisions, and the exact number is only known at the end of the first step, once the whole image has been traversed. For this reason, collision management must make use of dynamic data structures such as lists or hash tables.

Upon the completion of the first steps, all the original foreground pixels have been provisionally marked, and all the collisions between labels within the same regions have been registered for subsequent processing. The example in Fig. 10.4 illustrates the state upon completion of step 1: all foreground pixels have been assigned preliminary labels (Fig. 10.4(a)), and the following collisions (depicted by circles) between the labels (2, 4), (2, 5), and (2, 6) have been registered. The labels $\mathcal{L} = \{2, 3, 4, 5, 6, 7\}$ and collisions $\mathcal{C} = \{(2, 4), (2, 5), (2, 6)\}$ correspond to the nodes and edges of an undirected graph (Fig. 10.4(b)).

(a)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	1	1	0	1	0	0
0	1	1	1	1	1	1	0	0	1	0	0	1	0	0
0	0	0	0	1	0	1	0	0	0	0	0	1	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	0	0
0	1	1	0	0	0	1	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0 Background

1 Foreground

(b) Background neighbors only

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	1	1	0	1	0	0
0	1	1	1	1	1	1	0	0	1	0	0	1	0	0
0	0	0	0	1	0	1	0	0	0	0	0	1	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	0	0
0	1	1	0	0	0	1	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

New label (2)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	2	1	0	0	1	1	0	1	0	0
0	1	1	1	1	1	1	0	0	1	0	0	1	0	0
0	0	0	0	1	0	1	0	0	0	0	0	1	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
0	1	1	0	0	0	1	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(c) Exactly one neighbor label

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	2	1	0	0	1	1	0	1	0	0
0	1	1	1	1	1	1	0	0	1	0	0	1	0	0
0	0	0	0	1	0	1	0	0	0	0	0	1	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
0	1	1	0	0	0	1	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Neighbor label is propagated

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	2	2	0	0	1	1	0	1	0	0
0	1	1	1	1	1	1	1	0	0	1	0	0	1	0
0	0	0	0	1	0	1	0	0	0	0	0	0	1	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
0	1	1	0	0	0	1	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(d) Two different neighbor labels

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	2	2	0	0	3	3	0	4	0	0
0	5	5	5	1	1	1	0	0	1	0	0	1	0	0
0	0	0	0	1	0	1	0	0	0	0	0	1	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
0	1	1	0	0	0	1	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

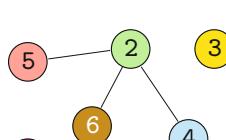
One of the labels (2) is propagated

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	2	2	0	0	3	3	0	4	0	0
0	5	5	5	2	1	1	0	0	1	0	0	1	0	0
0	0	0	0	1	0	1	0	0	0	0	0	1	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
0	1	1	0	0	0	1	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	2	2	0	0	3	3	0	4	0	0
0	5	5	5	2	2	0	0	3	0	0	4	0	0	0
0	0	0	0	2	2	0	0	0	0	0	4	0	0	0
0	6	6	2	2	2	2	2	2	2	2	2	0	0	0
0	0	0	0	2	2	2	2	2	2	2	2	2	0	0
0	7	7	0	0	0	2	0	2	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a)



Step 2: Resolving label collisions

The task in the second step is to resolve the label collisions that arose in the first step in order to merge the corresponding “partial” regions. This process is nontrivial since it is possible for two regions with dif-

10.1 FINDING CONNECTED IMAGE REGIONS

Fig. 10.3
Sequential region labeling—label propagation. Original image (a). The first foreground pixel (marked 1) is found in (b): all neighbors are background pixels (marked 0), and the pixel is assigned the first label (2). In the next step (c), there is exactly one neighbor pixel marked with the label 2, so this value is propagated. In (d) there are two neighboring pixels, and they have differing labels (2 and 5); one of these values is propagated, and the collision (2, 5) is registered.

Fig. 10.4
Sequential region labeling—intermediate result after step 1. Label collisions indicated by circles (a); the nodes of the undirected graph (b) correspond to the labels, and its edges correspond to the collisions.

ferent labels to be connected transitively (e.g., $(a, b) \cap (b, c) \Rightarrow (a, c)$) through a third region or, more generally, through a series of regions. In fact, this problem is identical to the problem of finding the *connected components* of a graph [54], where the labels \mathcal{L} determined in step 1 constitute the “nodes” of the graph and the registered collisions \mathcal{C} make up its “edges” (**Fig. 10.4(b)**).

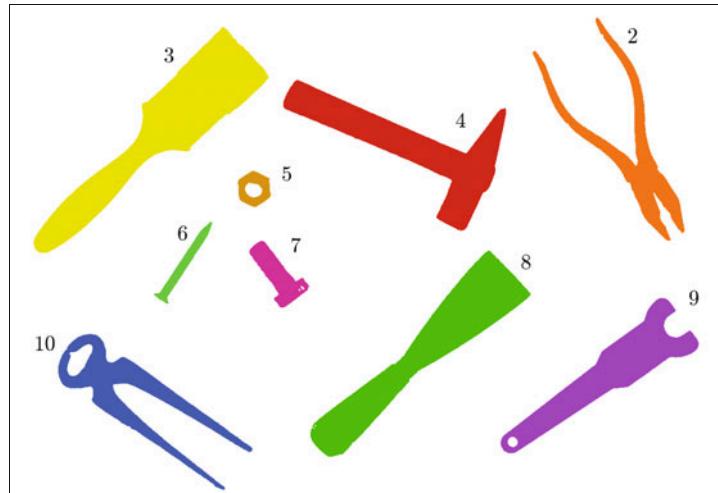
Once all the distinct labels within a single region have been collected, the labels of all the pixels in the region are updated so they carry the same label (e.g., choosing the smallest label number in the region), as depicted in Fig. 10.5. Figure 10.6 shows the complete segmentation with some region statistics that can be easily calculated from the labeling data.

Fig. 10.5

Sequential region labeling—final result after step 2. All equivalent labels have been replaced by the smallest label within that region.

Fig. 10.6

Example of a complete region labeling. The pixels within each region have been colored according to the consecutive label values 2, 3, ..., 10 they were assigned. The corresponding region statistics are shown in the table (total image size is 1212×836).



Label	Area (pixels)	Bounding Box (left, top, right, bottom)	Centroid (x_c , y_c)
2	14978	(887, 21, 1144, 399)	(1049.7, 242.8)
3	36156	(40, 37, 438, 419)	(261.9, 209.5)
4	25904	(464, 126, 841, 382)	(680.6, 240.6)
5	2024	(387, 281, 442, 341)	(414.2, 310.6)
6	2293	(244, 367, 342, 506)	(294.4, 439.0)
7	4394	(406, 400, 507, 512)	(454.1, 457.3)
8	29777	(510, 416, 883, 765)	(704.9, 583.9)
9	20724	(833, 497, 1168, 759)	(1016.0, 624.1)
10	16566	(82, 558, 411, 821)	(208.7, 661.6)

In this section, we have described a selection of algorithms for finding and labeling connected regions in images. We discovered that the elegant idea of labeling individual regions using a simple recursive flood-filling method (Sec. 10.1.1) was not useful because of practical limitations on the depth of recursion and the high memory costs associated with it. We also saw that classical sequential region labeling (Sec. 10.1.2) is relatively complex and offers no real advantage over iterative implementations of the *depth-first* and *breadth-first* methods. In practice, the iterative breadth-first method is generally the best choice for large and complex images. In the following section we present a modern and efficient algorithm that performs region labeling and also delineates the regions' contours. Since contours are required in many applications, this combined approach is highly practical.

10.2 Region Contours

Once the regions in a binary image have been found, the next step is often to find the contours (that is, the outlines) of the regions. Like so many other tasks in image processing, at first glance this appears to be an easy one: simply follow along the edge of the region. We will see that, in actuality, describing this apparently simple process algorithmically requires careful thought, which has made contour finding one of the classic problems in image analysis.

10.2.1 External and Internal Contours

As we discussed in Chapter 9, Sec. 9.2.7, the pixels along the edge of a binary region (i.e., its border) can be identified using simple morphological operations and difference images. It must be stressed, however, that this process only *marks* the pixels along the contour, which is useful, for instance, for display purposes. In this section, we will go one step further and develop an algorithm for obtaining an *ordered sequence* of border pixel coordinates for describing a region's contour. Note that connected image regions contain exactly one *outer* contour, yet, due to holes, they can contain arbitrarily many *inner* contours. Within such holes, smaller regions may be found, which will again have their own outer contours, and in turn these regions may themselves contain further holes with even smaller regions, and so on in a recursive manner (Fig. 10.7). An additional complication arises when regions are connected by parts that taper down to the width of a single pixel. In such cases, the contour can run through the same pixel more than once and from different directions (Fig. 10.8). Therefore, when tracing a contour from a start point x_s , returning to the start point is *not* a sufficient condition for terminating the contour-tracing process. Other factors, such as the current direction along which contour points are being traversed, must be taken into account.

One apparently simple way of determining a contour is to proceed in analogy to the two-stage process presented in Sec. 10.1; that is,

10 REGIONS IN BINARY IMAGES

Fig. 10.7

Binary image with outer and inner contours. The outer contour lies along the outside of the foreground region (dark). The inner contour surrounds the space within the region, which may contain further regions (holes), and so on.

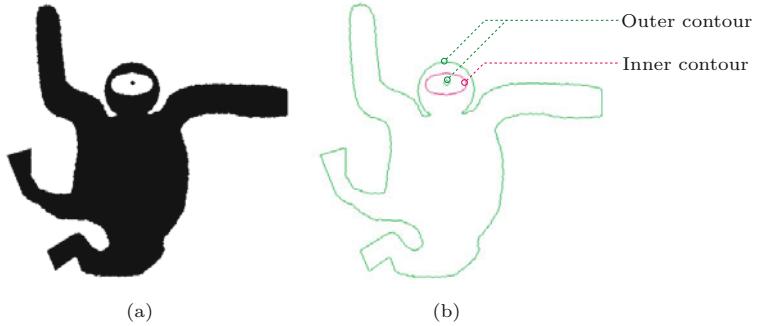
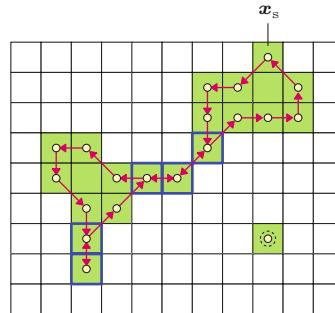


Fig. 10.8

The path along a contour as an ordered sequence of pixel coordinates with a given start point \mathbf{x}_s . Individual pixels may occur (be visited) more than once within the path, and a region consisting of a single isolated pixel will also have a contour (bottom right).



to *first* identify the connected regions in the image and *second*, for each region, proceed around it, starting from a pixel selected from its border. In the same way, an internal contour can be found by starting at a border pixel of a region's hole. A wide range of algorithms based on first finding the regions and then following along their contours have been published, including [202], [180, pp. 142–148], and [214, p. 296].

As a modern alternative, we present the following *combined* algorithm that, in contrast to the aforementioned classical methods, combines contour finding and region labeling in a single process.

10.2.2 Combining Region Labeling and Contour Finding

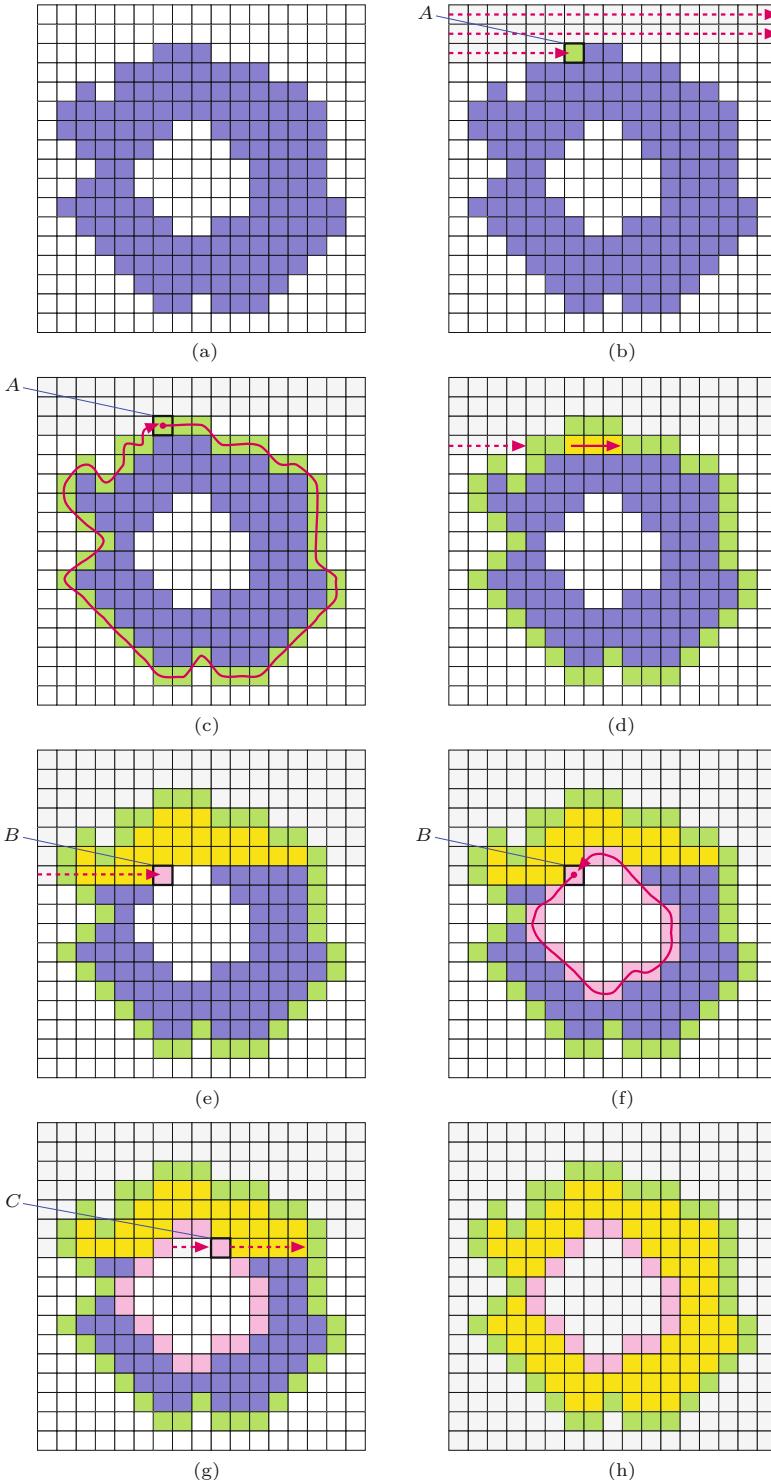
This method, based on [47], combines the concepts of sequential region labeling (Sec. 10.1) and traditional contour tracing into a single algorithm able to perform both tasks simultaneously during a single pass through the image. It identifies and labels regions and at the same time traces both their inner and outer contours. The algorithm does not require any complicated data structures and is relatively efficient when compared to other methods with similar capabilities. The key steps of this method are described here and illustrated in Fig. 10.9:

1. As in the sequential region labeling (Alg. 10.2), the binary image I is traversed from the top left to the bottom right. Such a traversal ensures that all pixels in the image are eventually examined and assigned an appropriate label.

10.2 REGION CONTOURS

Fig. 10.9

Combined region labeling and contour following (after [47]). The image in (a) is traversed from the top left to the lower right, one row at a time. In (b), the first foreground pixel A on the outer edge of the region is found. Starting from point A , the pixels on the edge along the outer contour are visited and labeled until A is reached again (c). Labels picked up at the outer contour are propagated along the image line inside the region (d). In (e), B was found as the first point on the *inner contour*. Now the inner contour is traversed in clock-wise direction, marking the contour pixels until point B is reached again (f). The same tracing process is used as in step (c), with the inside of the region always lying to the right of the contour path. In (g) a previously marked point C on an inner contour is detected. Its label is again propagated along the image line inside the region. The final result is shown in (h).



2. At a given position in the image, the following cases may occur:

Case A: The transition from a background pixel to a previously unmarked foreground pixel means that this pixel lies on the outer edge of a new region. A new *label* is assigned and the associated *outer* contour is traversed and marked by calling the method `TraceContour` (see Alg. 10.3 and Fig. 10.9(a)). Furthermore, all background pixels directly bordering the region are marked with the special label -1 .

Case B: The transition from a foreground pixel B to an unmarked background pixel means that this pixel lies on an *inner* contour (Fig. 10.9(b)). Starting from B , the inner contour is traversed and its pixels are marked with labels from the surrounding region (Fig. 10.9(c)). Also, all bordering background pixels are again assigned the special label value -1 .

Case C: When a foreground pixel does not lie on a contour, then the neighboring pixel to the left has already been labeled (Fig. 10.9(d)) and this label is propagated to the current pixel.

In Algs. 10.3–10.4, the entire procedure is presented again and explained precisely. Procedure `RegionContourLabeling` traverses the image line-by-line and calls procedure `TraceContour` whenever a new inner or outer contour must be traced. The labels of the image elements along the contour, as well as the neighboring foreground pixels, are stored in the “label map” L (a rectangular array of the same size as the image) by procedure `FindNextContourPoint` in Alg. 10.4.

10.2.3 Java Implementation

The Java implementation of the combined region labeling and contour tracing algorithm can be found online in class `RegionContourLabeling`⁴ (for details see Sec. 10.9). It almost exactly follows Algs. 10.3–10.4, only the image I and the associated *label map* L are initially *padded* (i.e., enlarged) by a surrounding layer of background pixels. This simplifies the process of tracing the outer region contours, since no special treatment is needed at the image borders. Program 10.2 shows a minimal example of its usage within the `run()` method of an ImageJ plugin (class `Trace_Contours`).

Examples

This combined algorithm for region marking and contour following is particularly well suited for processing large binary images since it is efficient and has only modest memory requirements. Figure 10.10 shows a synthetic test image that illustrates a number of special situations, such as isolated pixels and thin sections, which the algorithm must deal with correctly when following the contours. In the resulting plot, outer contours are shown as black polygon lines running through the centers of the contour pixels, and inner contours are drawn white. Contours of single-pixel regions are marked by small circles filled with the corresponding color. Figure 10.11 shows the results for a larger section taken from a real image (Fig. 9.12).

⁴ Package `imagingbook.pub.regions`.

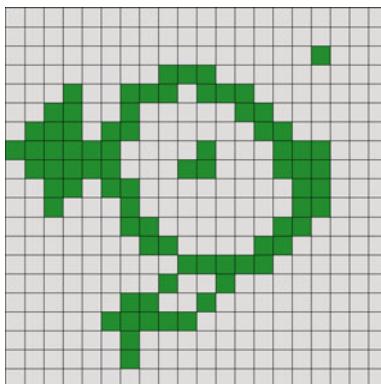
1: RegionContourLabeling(I)

Input: I , a binary image with $0 = \text{background}$, $1 = \text{foreground}$.
Returns sequences of outer and inner contours and a map of region labels.

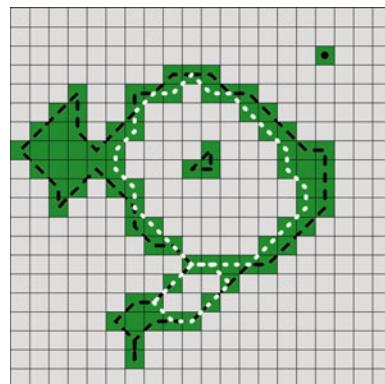
```

2:  $(M, N) \leftarrow \text{Size}(I)$ 
3:  $C_{\text{out}} \leftarrow ()$                                  $\triangleright$  empty list of outer contours
4:  $C_{\text{in}} \leftarrow ()$                                  $\triangleright$  empty list of inner contours
5: Create map  $L: M \times N \mapsto \mathbb{Z}$                  $\triangleright$  create the label map  $L$ 
6: for all  $(u, v)$  do
7:    $L(u, v) \leftarrow 0$                                  $\triangleright$  initialize  $L$  to zero
8:    $r \leftarrow 0$                                           $\triangleright$  region counter
9: for  $v \leftarrow 0, \dots, N-1$  do                       $\triangleright$  scan the image top to bottom
10:    $label \leftarrow 0$ 
11:   for  $u \leftarrow 0, \dots, M-1$  do                   $\triangleright$  scan the image left to right
12:     if  $I(u, v) > 0$  then                          $\triangleright I(u, v)$  is a foreground pixel
13:       if  $(label \neq 0)$  then                      $\triangleright$  continue existing region
14:          $L(u, v) \leftarrow label$ 
15:       else
16:          $label \leftarrow L(u, v)$ 
17:       if  $(label = 0)$  then           $\triangleright$  hit a new outer contour
18:          $r \leftarrow r + 1$ 
19:          $label \leftarrow r$ 
20:          $x_s \leftarrow (u, v)$ 
21:          $C \leftarrow \text{TraceContour}(x_s, 0, label, I, L)$   $\triangleright$  outer c.
22:          $C_{\text{out}} \leftarrow C_{\text{out}} \cup (C)$             $\triangleright$  collect outer contour
23:          $L(u, v) \leftarrow label$ 
24:       else                                      $\triangleright I(u, v)$  is a background pixel
25:         if  $(label \neq 0)$  then
26:           if  $(L(u, v) = 0)$  then           $\triangleright$  hit new inner contour
27:              $x_s \leftarrow (u-1, v)$ 
28:              $C \leftarrow \text{TraceContour}(x_s, 1, label, I, L)$   $\triangleright$  inner cntr.
29:              $C_{\text{in}} \leftarrow C_{\text{in}} \cup (C)$             $\triangleright$  collect inner contour
30:            $label \leftarrow 0$ 
31: return  $(C_{\text{out}}, C_{\text{in}}, L)$ 
```

continued in Alg. 10.4 \bowtie



(a)



(b)

10.2 REGION CONTOURS

Alg. 10.3

Combined contour tracing and region labeling (part 1). Given a binary image I , the application of $\text{RegionContourLabeling}(I)$ returns a set of contours and an array containing region labels for all pixels in the image. When a new point on either an outer or inner contour is found, then an ordered list of the contour's points is constructed by calling procedure TraceContour (line 21 and line 28). TraceContour itself is described in Alg. 10.4.

Fig. 10.10

Combined contour and region marking. Original image, with foreground pixels marked green (a); located contours with black lines for *outer* and white lines for *inner* contours (b). Contour polygons pass through the pixel centers. Outer contours of single-pixel regions (e.g., in the upper-right of (b)) are marked by a single dot.

10 REGIONS IN BINARY IMAGES

Alg. 10.4

Combined contour finding and region labeling (part 2, continued from Alg. 10.3). Starting from x_s , the procedure `TraceContour` traces along the contour in the direction $d_s = 0$ for outer contours or $d_s = 1$ for inner contours. During this process, all contour points as well as neighboring background points are marked in the label array L . Given a point x_c , `TraceContour` uses `FindNextContourPoint()` to determine the next point along the contour (line 9). The function `Delta()` returns the next coordinate in the sequence, taking into account the search direction d .

1: **TraceContour**($x_s, d_s, label, I, L$)
Input: x_s , start position; d_s , initial search direction; $label$, the label assigned to this contour; I , the binary input image; L , label map. Returns a new outer or inner contour (sequence of points) starting at x_s .

2: $(x, d) \leftarrow \text{FindNextContourPoint}(x_s, d_s, I, L)$
3: $c \leftarrow (x)$ ▷ new contour with the single point x
4: $x_p \leftarrow x_s$ ▷ previous position $x_p = (u_p, v_p)$
5: $x_c \leftarrow x$ ▷ current position $x_c = (u_c, v_c)$
6: $done \leftarrow (x_s \equiv x)$ ▷ isolated pixel?
7: **while** ($\neg done$) **do**
8: $L(u_c, v_c) \leftarrow label$
9: $(x_n, d) \leftarrow \text{FindNextContourPoint}(x_c, (d + 6) \bmod 8, I, L)$
10: $x_p \leftarrow x_c$
11: $x_c \leftarrow x_n$
12: $done \leftarrow (x_p \equiv x_s \wedge x_c \equiv x)$ ▷ back at starting position?
13: **if** ($\neg done$) **then**
14: $c \leftarrow c \cup (x_n)$ ▷ add point x_n to contour c
15: **return** c ▷ return this contour

16: **FindNextContourPoint**(x, d, I, L)
Input: x , initial position; $d \in [0, 7]$, search direction, I , binary input image; L , the label map.
Returns the next point on the contour and the modified search direction.

17: **for** $i \leftarrow 0, \dots, 6$ **do** ▷ search in 7 directions
18: $x_n \leftarrow x + \text{Delta}(d)$
19: **if** $I(x_n) = 0$ **then** ▷ $I(u_n, v_n)$ is a background pixel
20: $L(x_n) \leftarrow -1$ ▷ mark background as visited (-1)
21: $d \leftarrow (d + 1) \bmod 8$
22: **else** ▷ found a non-background pixel at x_n
23: **return** (x_n, d)
24: **return** (x, d) ▷ found no next node, return start position

25: **Delta**(d) := $\begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}$, with
$$\begin{array}{c|ccccccc} d & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline \Delta x & 1 & 1 & 0 & -1 & -1 & -1 & 0 & 1 \\ \Delta y & 0 & 1 & 1 & 1 & 0 & -1 & -1 & -1 \end{array}$$

Prog. 10.2

Example of using the class `ContourTracer`. (plugin `Trace_Contours`). First (in line 9) a new instance of `RegionContourLabeling` is created for the input image I .

The segmentation into regions and contours is done by the constructor. In lines 11–12 the outer and inner contours are retrieved as (possibly empty) lists of type `Contour`. Finally, the list of connected regions is obtained in line 14.

```

1 import imagingbook.pub.regions.BinaryRegion;
2 import imagingbook.pub.regions.Contour;
3 import imagingbook.pub.regions.RegionContourLabeling;
4 import java.util.List;
5 ...
6 public void run(ImageProcessor ip) {
7     // Make sure we have a proper byte image:
8     ByteProcessor I = ip.convertToByteProcessor();
9     // Create the region labeler / contour tracer:
10    RegionContourLabeling seg = new RegionContourLabeling(I);
11    // Get all outer/inner contours and connected regions:
12    List<Contour> outerContours = seg.getOuterContours();
13    List<Contour> innerContours = seg.getInnerContours();
14    List<BinaryRegion> regions = seg.getRegions();
15    ...
16 }
```



10.3 REPRESENTING IMAGE REGIONS

Fig. 10.11

Example of a complex contour (original image in Ch. 9, Fig. 9.12). Outer contours are marked in black and inner contours in white.

10.3 Representing Image Regions

10.3.1 Matrix Representation

A natural representation for images is a matrix (i.e., a two-dimensional array) in which elements represent the intensity or the color at a corresponding position in the image. This representation lends itself, in most programming languages, to a simple and elegant mapping onto two-dimensional arrays, which makes possible a very natural way to work with raster images. One possible disadvantage with this representation is that it does not depend on the content of the image. In other words, it makes no difference whether the image contains only a pair of lines or is of a complex scene because the amount of memory required is constant and depends only on the dimensions of the image.

Regions in an image can be represented using a logical mask in which the area within the region is assigned the value *true* and the area without the value *false* (Fig. 10.12). Since these values can be represented by a single bit, such a matrix is often referred to as a “bitmap”.⁵

10.3.2 Run Length Encoding

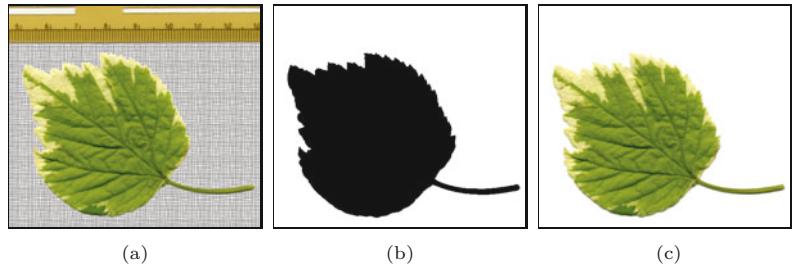
In *run length encoding* (RLE), sequences of adjacent foreground pixels can be represented compactly as “runs”. A run, or contiguous

⁵ Java does not provide a genuine 1-bit data type. Even variables of type `boolean` are represented internally (i.e., within the Java virtual machine) as 32-bit `ints`.

10 REGIONS IN BINARY IMAGES

Fig. 10.12

Use of a binary mask to specify a region of an image: original image (a), logical (bit) mask (b), and masked image (c).



block, is a maximal length sequence of adjacent pixels of the same type within either a row or a column. Runs of arbitrary length can be encoded compactly using three integers,

$$Run_i = \langle row_i, column_i, length_i \rangle,$$

as illustrated in [Fig. 10.13](#). When representing a sequence of runs within the same row, the number of the row is redundant and can be left out. Also, in some applications, it is more useful to record the coordinate of the end column instead of the length of the run.

Fig. 10.13

Run length encoding in row direction. A run of pixels can be represented by its starting point $(1, 2)$ and its length (6).

Bitmap									RLE
0	1	2	3	4	5	6	7	8	$\langle row, column, length \rangle$
0									
1		•	•	•	•	•	•	•	$\langle 1, 2, 6 \rangle$
2									$\langle 3, 4, 4 \rangle$
3				•	•	•	•	•	$\langle 4, 1, 3 \rangle$
4	•	•	•	•	•	•	•	•	$\langle 4, 5, 3 \rangle$
5	•	•	•	•	•	•	•	•	$\langle 5, 0, 9 \rangle$
6									

Since the RLE representation can be easily implemented and efficiently computed, it has long been used as a simple lossless compression method. It forms the foundation for fax transmission and can be found in a number of other important codecs, including TIFF, GIF, and JPEG. In addition, RLE provides precomputed information about the image that can be used directly when computing certain properties of the image (for example, statistical moments; see Sec. 10.5.2).

10.3.3 Chain Codes

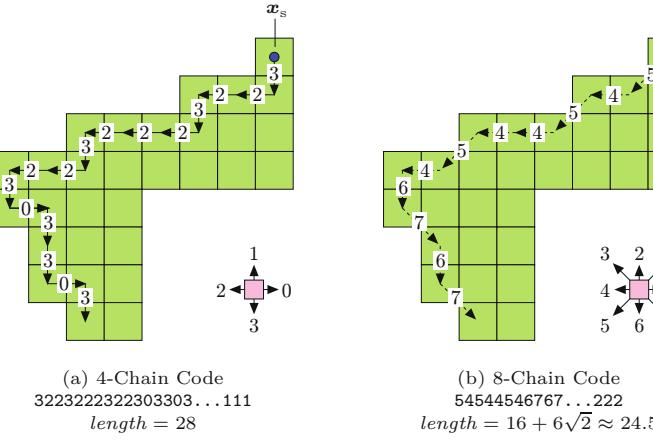
Regions can be represented not only using their interiors but also by their contours. Chain codes, which are often referred to as Freeman codes [79], are a classical method of contour encoding. In this encoding, the contour beginning at a given start point x_s is represented by the sequence of directional changes it describes on the discrete image grid ([Fig. 10.14](#)).

Absolute chain code

For a closed contour of a region \mathcal{R} , described by the sequence of points $\mathbf{c}_{\mathcal{R}} = (\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{M-1})$ with $\mathbf{x}_i = \langle u_i, v_i \rangle$, we create the elements of its chain code sequence $\mathbf{c}'_{\mathcal{R}} = (c'_0, c'_1, \dots, c'_{M-1})$ with

Fig. 10.14

Chain codes with 4- and 8-connected neighborhoods. To compute a chain code, begin traversing the contour from a given starting point x_s . Encode the relative position between adjacent contour points using the directional code for either 4-connected (left) or 8-connected (right) neighborhoods. The length of the resulting path, calculated as the sum of the individual segments, can be used to approximate the true length of the contour.



$$c'_i = \text{Code}(u', v'), \quad (10.3)$$

where

$$(u', v') = \begin{cases} (u_{i+1} - u_i, v_{i+1} - v_i) & \text{for } 0 \leq i < M-1, \\ (u_0 - u_i, v_0 - v_i) & \text{for } i = M-1, \end{cases} \quad (10.4)$$

and $\text{Code}(u', v')$ being defined (assuming an 8-connected neighborhood) by the following table:

u'	1	1	0	-1	-1	-1	0	1
v'	0	1	1	1	0	-1	-1	-1
$\text{Code}(u', v')$	0	1	2	3	4	5	6	7

Chain codes are compact since instead of storing the absolute coordinates for every point on the contour, only that of the starting point is recorded. The remaining points are encoded relative to the starting point by indicating in which of the eight possible directions the next point lies. Since only 3 bits are required to encode these eight directions the values can be stored using a smaller numeric type.

Differential chain code

Directly comparing two regions represented using chain codes is difficult since the description depends on the starting point selected x_s , and for instance simply rotating the region by 90° results in a completely different chain code. When using a *differential* chain code, the situation improves slightly. Instead of encoding the difference in the *position* of the next contour point, the change in the *direction* along the discrete contour is encoded. A given *absolute* chain code $\mathbf{c}_R' = (c'_0, c'_1, \dots, c'_{M-1})$ can be converted element by element to a *differential* chain code $\mathbf{c}_R'' = (c''_0, c''_1, \dots, c''_{M-1})$, with⁶

$$c''_i = \begin{cases} (c'_{i+1} - c'_i) \bmod 8 & \text{for } 0 \leq i < M-1, \\ (c'_0 - c'_i) \bmod 8 & \text{for } i = M-1, \end{cases} \quad (10.5)$$

⁶ For the implementation of the mod operator see Sec. F.1.2 in the Appendix.

again under the assumption of an 8-connected neighborhood. The element c''_i thus describes the change in direction (curvature) of the contour between two successive segments c'_i and c'_{i+1} of the original chain code $\mathbf{c}'_{\mathcal{R}}$. For the contour in Fig. 10.14(b), for example, the result is

$$\begin{aligned}\mathbf{c}'_{\mathcal{R}} &= (5, 4, 5, 4, 4, 5, 4, 6, 7, 6, 7, \dots, 2, 2, 2), \\ \mathbf{c}''_{\mathcal{R}} &= (7, 1, 7, 0, 1, 7, 2, 1, 7, 1, 1, \dots, 0, 0, 3).\end{aligned}$$

Given the start position \mathbf{x}_s and the (absolute) initial direction c_0 , the original contour can be unambiguously reconstructed from the differential chain code.

Shape numbers

While the differential chain code remains the same when a region is rotated by 90° , the encoding is still dependent on the selected starting point. If we want to determine the similarity of two contours of the same length M using their differential chain codes \mathbf{c}''_1 , \mathbf{c}''_2 , we must first ensure that the same start point was used when computing the codes. A method that is often used [15, 88] is to interpret the elements c''_i in the differential chain code as the digits of a number to the base b ($b = 8$ for an 8-connected contour or $b = 4$ for a 4-connected contour) and the numeric value

$$\text{Val}(\mathbf{c}''_{\mathcal{R}}) = c''_0 \cdot b^0 + c''_1 \cdot b^1 + \dots + c''_{M-1} \cdot b^{M-1} = \sum_{i=0}^{M-1} c''_i \cdot b^i. \quad (10.6)$$

Then the sequence $\mathbf{c}''_{\mathcal{R}}$ is shifted circularly until the numeric value of the corresponding number reaches a maximum. We use the expression $\mathbf{c}''_{\mathcal{R}} \triangleright k$ to denote the sequence $\mathbf{c}''_{\mathcal{R}}$ being circularly shifted by k positions to the right.⁷ For example, for $k = 2$ this is

$$\begin{aligned}\mathbf{c}''_{\mathcal{R}} &= (0, 1, 3, 2, \dots, 5, 3, 7, 4), \\ \mathbf{c}''_{\mathcal{R}} \triangleright 2 &= (7, 4, 0, 1, 3, 2, \dots, 5, 3),\end{aligned}$$

and

$$k_{\max} = \underset{0 \leq k < M}{\operatorname{argmax}} \text{Val}(\mathbf{c}''_{\mathcal{R}} \triangleright k), \quad (10.7)$$

denotes the shift required to maximize the corresponding arithmetic value. The resulting code sequence or *shape number*,

$$\mathbf{s}_{\mathcal{R}} = \mathbf{c}''_{\mathcal{R}} \triangleright k_{\max}, \quad (10.8)$$

is *normalized* with respect to the starting point and can thus be directly compared element by element with other normalized code sequences. Since the function $\text{Val}()$ in Eqn. (10.6) produces values that are in general too large to be actually computed, in practice the relation

$$\text{Val}(\mathbf{c}''_1) > \text{Val}(\mathbf{c}''_2)$$

⁷ That is, $(\mathbf{c}''_{\mathcal{R}} \triangleright k)(i) = \mathbf{c}''_{\mathcal{R}}((i - k) \bmod M)$.

is determined by comparing the *lexicographic ordering* between the sequences \mathbf{c}_1'' and \mathbf{c}_2'' so that the arithmetic values need not be computed at all.

Unfortunately, comparisons based on chain codes are generally not very useful for determining the similarity between regions simply because rotations at arbitrary angles ($\neq 90^\circ$) have too great of an impact (change) on a region's code. In addition, chain codes are not capable of handling changes in size (scaling) or other distortions. Section 10.4 presents a number of tools that are more appropriate in these types of cases.

Fourier shape descriptors

An elegant approach to describing contours are so-called Fourier shape descriptors, which interpret the two-dimensional contour $C = (\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{M-1})$ with $\mathbf{x}_i = (u_i, v_i)$ as a sequence of values in the complex plane, where

$$z_i = (u_i + i \cdot v_i) \in \mathbb{C}. \quad (10.9)$$

From this sequence, one obtains (using a suitable method of interpolation in case of an 8-connected contour), a discrete, one-dimensional periodic function $f(s) \in \mathbb{C}$ with a constant sampling interval over s , the path length around the contour. The coefficients of the 1D *Fourier spectrum* (see Sec. 18.3) of this function $f(s)$ provide a shape description of the contour in frequency space, where the lower spectral coefficients deliver a gross description of the shape. The details of this classical method can be found, for example, in [88, 97, 126, 128, 222]. This technique is described in considerable detail in Chapter 26.

10.4 Properties of Binary Regions

Imagine that you have to describe the contents of a digital image to another person over the telephone. One possibility would be to call out the value of each pixel in some agreed upon order. A much simpler way of course would be to describe the image on the basis of its properties—for example, “a red rectangle on a blue background”, or at an even higher level such as “a sunset at the beach with two dogs playing in the sand”. While using such a description is simple and natural for us, it is not (yet) possible for a computer to generate these types of descriptions without human intervention. For computers, it is of course simpler to calculate the mathematical properties of an image or region and to use these as the basis for further classification. Using features to classify, be they images or other items, is a fundamental part of the field of pattern recognition, a research area with many applications in image processing and computer vision [64, 169, 228].

10.4.1 Shape Features

The comparison and classification of binary regions is widely used, for example, in optical character recognition (OCR) and for automating

processes ranging from blood cell counting to quality control inspection of manufactured products on assembly lines. The analysis of binary regions turns out to be one of the simpler tasks for which many efficient algorithms have been developed and used to implement reliable applications that are in use every day.

By a *feature* of a region, we mean a specific numerical or qualitative measure that is computable from the values and coordinates of the pixels that make up the region. As an example, one of the simplest features is its *size* or *area*; that is the number of pixels that make up a region. In order to describe a region in a compact form, different features are often combined into a *feature vector*. This vector is then used as a sort of “signature” for the region that can be used for classification or comparison with other regions. The best features are those that are simple to calculate and are not easily influenced (robust) by irrelevant changes, particularly translation, rotation, and scaling.

10.4.2 Geometric Features

A region \mathcal{R} of a binary image can be interpreted as a two-dimensional distribution of foreground points $\mathbf{p}_i = (u_i, v_i)$ on the discrete plane \mathbb{Z}^2 , that is, as a set

$$\mathcal{R} = \{\mathbf{x}_0, \dots, \mathbf{x}_{N-1}\} = \{(u_0, v_0), (u_1, v_1), \dots, (u_{N-1}, v_{N-1})\}.$$

Most geometric properties are defined in such a way that a region is considered to be a set of pixels that, in contrast to the definition in Sec. 10.1, does not necessarily have to be connected.

Perimeter

The perimeter (or circumference) of a region \mathcal{R} is defined as the length of its outer contour, where \mathcal{R} must be connected. As illustrated in Fig. 10.14, the type of neighborhood relation must be taken into account for this calculation. When using a 4-neighborhood, the measured length of the contour (except when that length is 1) will be larger than its actual length.

In the case of 8-neighborhoods, a good approximation is reached by weighing the horizontal and vertical segments with 1 and diagonal segments with $\sqrt{2}$. Given an 8-connected chain code $\mathbf{c}'_{\mathcal{R}} = (c'_0, c'_1, \dots, c'_{M-1})$, the perimeter of the region is arrived at by

$$\text{Perimeter}(\mathcal{R}) = \sum_{i=0}^{M-1} \text{length}(c'_i), \quad (10.10)$$

with

$$\text{length}(c) = \begin{cases} 1 & \text{for } c = 0, 2, 4, 6, \\ \sqrt{2} & \text{for } c = 1, 3, 5, 7. \end{cases} \quad (10.11)$$

However, with this conventional method of calculation, the real perimeter $P(\mathcal{R})$ is systematically overestimated. As a simple remedy, an empirical correction factor of 0.95 works satisfactorily even for relatively small regions, that is,

$$P(\mathcal{R}) \approx 0.95 \cdot \text{Perimeter}(\mathcal{R}). \quad (10.12)$$

The area of a binary region \mathcal{R} can be found by simply counting the image pixels that make up the region, that is,

$$A(\mathcal{R}) = N = |\mathcal{R}|. \quad (10.13)$$

The area of a connected region without holes can also be approximated from its closed contour, defined by M coordinate points $(\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{M-1})$, where $\mathbf{x}_i = (u_i, v_i)$, using the Gaussian area formula for polygons:

$$A(\mathcal{R}) \approx \frac{1}{2} \cdot \left| \sum_{i=0}^{M-1} (u_i \cdot v_{(i+1) \bmod M} - u_{(i+1) \bmod M} \cdot v_i) \right|. \quad (10.14)$$

When the contour is already encoded as a chain code $\mathbf{c}'_{\mathcal{R}} = (c'_0, c'_1, \dots, c'_{M-1})$, then the region's area can be computed (trivially) with Eqn. (10.14) by expanding C_{abs} into a sequence of contour points from an arbitrary starting point (e.g., $(0, 0)$). However, the area can also be calculated directly from the chain code representation without expanding the contour [263] (see also Exercise 10.12).

While simple region properties such as area and perimeter are not influenced (except for quantization errors) by translation and rotation of the region, they are definitely affected by changes in size; for example, when the object to which the region corresponds is imaged from different distances. However, as will be described, it is possible to specify combined features that are *invariant* to translation, rotation, and scaling as well.

Compactness and roundness

Compactness is understood as the relation between a region's area and its perimeter. We can use the fact that a region's perimeter P increases linearly with the enlargement factor while the area A increases quadratically to see that, for a particular shape, the ratio A/P^2 should be the same at any scale. This ratio can thus be used as a feature that is invariant under translation, rotation, and scaling. When applied to a circular region of any diameter, this ratio has a value of $\frac{1}{4\pi}$, so by normalizing it against a filled circle, we create a feature that is sensitive to the *roundness* or *circularity* of a region,

$$\text{Circularity}(\mathcal{R}) = 4\pi \cdot \frac{A(\mathcal{R})}{P^2(\mathcal{R})}, \quad (10.15)$$

which results in a maximum value of 1 for a perfectly round region \mathcal{R} and a value in the range $[0, 1]$ for all other shapes (Fig. 10.15). If an absolute value for a region's roundness is required, the corrected perimeter estimate (Eqn. (10.12)) should be employed. Figure 10.15 shows the circularity values of different regions as computed with the formulation in Eqn. (10.15).

Bounding box

The bounding box of a region \mathcal{R} is the minimal axis-parallel rectangle that encloses all points of \mathcal{R} ,

10 REGIONS IN BINARY IMAGES

Fig. 10.15

Circularity values for different shapes. Shown are the corresponding estimates for $\text{Circularity}(\mathcal{R})$ as defined in Eqn. (10.15). Corrected values calculated with Eqn. (10.12) are shown in parentheses.

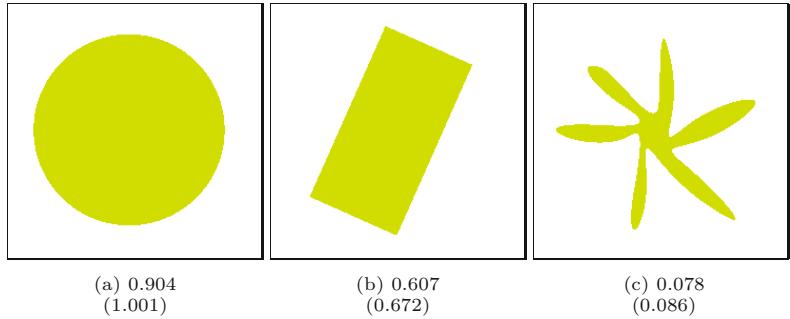
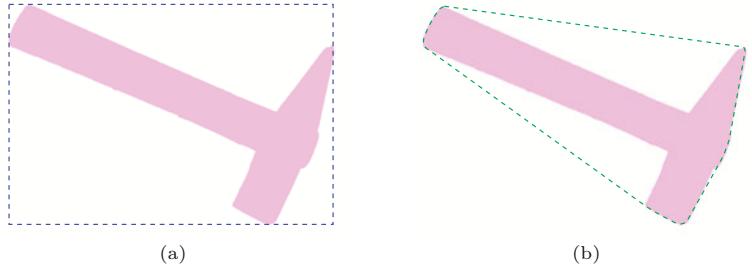


Fig. 10.16

Example bounding box (a) and convex hull (b) of a binary image region.



$$\text{BoundingBox}(\mathcal{R}) = \langle u_{\min}, u_{\max}, v_{\min}, v_{\max} \rangle, \quad (10.16)$$

where u_{\min}, u_{\max} and v_{\min}, v_{\max} are the minimal and maximal coordinate values of all points $(u_i, v_i) \in \mathcal{R}$ in the x and y directions, respectively (Fig. 10.16(a)).

Convex hull

The convex hull is the smallest convex polygon that contains all points of the region \mathcal{R} . A physical analogy is a board in which nails stick out in correspondence to each of the points in the region. If you were to place an elastic band around *all* the nails, then, when you release it, it will contract into a convex hull around the nails (see Figs. 10.16(b) and 10.21(c)). Given N contour points, the convex hull can be computed in time $\mathcal{O}(N \log V)$, where V is the number vertices in the polygon of the resulting convex hull [17].

The convex hull is useful, for example, for determining the convexity or the *density* of a region. The *convexity* is defined as the relationship between the length of the convex hull and the original perimeter of the region. *Density* is then defined as the ratio between the area of the region and the area of its convex hull. The *diameter*, on the other hand, is the maximal distance between any two nodes on the convex hull.

10.5 Statistical Shape Properties

When computing statistical shape properties, we consider a region \mathcal{R} to be a collection of coordinate points distributed within a two-dimensional space. Since statistical properties can be computed for point distributions that do not form a connected region, they can

be applied before segmentation. An important concept in this context are the *central moments* of the region's point distribution, which measure characteristic properties with respect to its midpoint or *centroid*.

10.5.1 Centroid

The centroid or center of gravity of a connected region can be easily visualized. Imagine drawing the region on a piece of cardboard or tin and then cutting it out and attempting to balance it on the tip of your finger. The location on the region where you must place your finger in order for the region to balance is the *centroid* of the region.⁸

The centroid $\bar{\mathbf{x}} = (\bar{x}, \bar{y})^\top$ of a binary (not necessarily connected) region is the arithmetic mean of the point coordinates $\mathbf{x}_i = (u_i, v_i)$, that is,

$$\bar{\mathbf{x}} = \frac{1}{|\mathcal{R}|} \cdot \sum_{\mathbf{x}_i \in \mathcal{R}} \mathbf{x}_i \quad (10.17)$$

or

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u_i, v_i)} u_i \quad \text{and} \quad \bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u_i, v_i)} v_i. \quad (10.18)$$

10.5.2 Moments

The formulation of the region's centroid in Eqn. (10.18) is only a special case of the more general statistical concept of a *moment*. Specifically, the expression

$$m_{pq}(\mathcal{R}) = \sum_{(u, v) \in \mathcal{R}} I(u, v) \cdot u^p \cdot v^q \quad (10.19)$$

describes the (ordinary) moment of order p, q for a discrete (image) function $I(u, v) \in \mathbb{R}$; for example, a grayscale image. All the following definitions are also generally applicable to regions in grayscale images. The moments of connected binary regions can also be calculated directly from the coordinates of the contour points [212, p. 148].

In the special case of a binary image $I(u, v) \in \{0, 1\}$, only the foreground pixels with $I(u, v) = 1$ in the region \mathcal{R} need to be considered, and therefore Eqn. (10.19) can be simplified to

$$m_{pq}(\mathcal{R}) = \sum_{(u, v) \in \mathcal{R}} u^p \cdot v^q. \quad (10.20)$$

In this way, the *area* of a binary region can be expressed as its zero-order moment,

$$A(\mathcal{R}) = |\mathcal{R}| = \sum_{(u, v)} 1 = \sum_{(u, v)} u^0 \cdot v^0 = m_{00}(\mathcal{R}) \quad (10.21)$$

and similarly the *centroid* $\bar{\mathbf{x}}$ Eqn. (10.18) can be written as

⁸ Assuming you did not imagine a region where the centroid lies outside of the region or within a hole in the region, which is of course possible.

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)} u^1 \cdot v^0 = \frac{m_{10}(\mathcal{R})}{m_{00}(\mathcal{R})},$$

$$\bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)} u^0 \cdot v^1 = \frac{m_{01}(\mathcal{R})}{m_{00}(\mathcal{R})}.$$
(10.22)

These moments thus represent concrete physical properties of a region. Specifically, the area m_{00} is in practice an important basis for characterizing regions, and the centroid (\bar{x}, \bar{y}) permits the reliable and (within a fraction of a pixel) exact specification of a region's position.

10.5.3 Central Moments

To compute position-independent (translation-invariant) region features, the region's centroid, which can be determined precisely in any situation, can be used as a reference point. In other words, we can shift the origin of the coordinate system to the region's centroid $\bar{x} = (\bar{x}, \bar{y})$ to obtain the *central* moments of order p, q :

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v) \in \mathcal{R}} I(u, v) \cdot (u - \bar{x})^p \cdot (v - \bar{y})^q.$$
(10.23)

For a binary image (with $I(u, v) = 1$ within the region \mathcal{R}), Eqn. (10.23) can be simplified to

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v) \in \mathcal{R}} (u - \bar{x})^p \cdot (v - \bar{y})^q.$$
(10.24)

10.5.4 Normalized Central Moments

Central moment values of course depend on the absolute size of the region since the value depends directly on the distance of all region points to its centroid. So, if a 2D shape is scaled uniformly by some factor $s \in \mathbb{R}$, its central moments multiply by the factor

$$s^{(p+q+2)}.$$
(10.25)

Thus size-invariant “normalized” moments are obtained by scaling with the reciprocal of the area $A = \mu_{00} = m_{00}$ raised to the required power in the form

$$\bar{\mu}_{pq}(\mathcal{R}) = \mu_{pq} \cdot \left(\frac{1}{\mu_{00}(\mathcal{R})} \right)^{(p+q+2)/2},$$
(10.26)

for $(p + q) \geq 2$ [126, p. 529].

10.5.5 Java Implementation

Program 10.3 gives a direct (brute force) Java implementation for computing the ordinary, central, and normalized central moments for binary images (`BACKGROUND = 0`). This implementation is only meant to clarify the computation, and naturally much more efficient implementations are possible (see, e.g., [131]).

```

1 // Ordinary moment:
2
3 double moment(ImageProcessor I, int p, int q) {
4     double Mpq = 0.0;
5     for (int v = 0; v < I.getHeight(); v++) {
6         for (int u = 0; u < I.getWidth(); u++) {
7             if (I.getPixel(u, v) > 0) {
8                 Mpq+= Math.pow(u, p) * Math.pow(v, q);
9             }
10        }
11    }
12    return Mpq;
13 }
14
15 // Central moments:
16
17 double centralMoment(ImageProcessor I, int p, int q) {
18     double m00 = moment(I, 0, 0); //region area
19     double xCtr = moment(I, 1, 0) / m00;
20     double yCtr = moment(I, 0, 1) / m00;
21     double cMpq = 0.0;
22     for (int v = 0; v < I.getHeight(); v++) {
23         for (int u = 0; u < I.getWidth(); u++) {
24             if (I.getPixel(u, v) > 0) {
25                 cMpq+= Math.pow(u-xCtr, p) * Math.pow(v-yCtr, q);
26             }
27        }
28    }
29    return cMpq;
30 }
31
32 // Normalized central moments:
33
34 double nCentralMoment(ImageProcessor I, int p, int q) {
35     double m00 = moment(I, 0, 0);
36     double norm = Math.pow(m00, 0.5 * (p + q + 2));
37     return centralMoment(I, p, q) / norm;
38 }

```

10.6 MOMENT-BASED GEOMETRIC PROPERTIES

Prog. 10.3

Example of directly computing moments in Java. The methods `moment()`, `centralMoment()`, and `nCentralMoment()` compute for a binary image the moments m_{pq} , μ_{pq} , and $\bar{\mu}_{pq}$ (Eqns. (10.20), (10.24), and (10.26)).

10.6 Moment-Based Geometric Properties

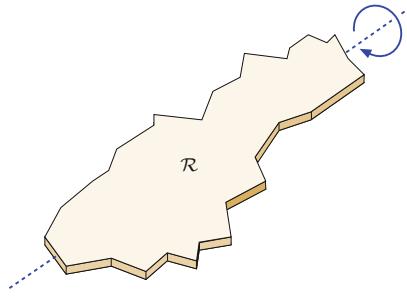
While normalized moments can be directly applied for classifying regions, further interesting and geometrically relevant features can be elegantly derived from statistical region moments.

10.6.1 Orientation

Orientation describes the direction of the major axis, that is, the axis that runs through the centroid and along the widest part of the region ([Fig. 10.18\(a\)](#)). Since rotating the region around the major axis requires less effort (smaller moment of inertia) than spinning it around any other axis, it is sometimes referred to as the major axis of rotation. As an example, when you hold a pencil between your hands and twist it around its major axis (that is, around the lead),

Fig. 10.17

Major axis of a region. Rotating an elongated region \mathcal{R} , interpreted as a physical body, around its major axis requires less effort (least moment of inertia) than rotating it around any other axis.



the pencil exhibits the least mass inertia (Fig. 10.17). As long as a region exhibits an orientation at all ($\mu_{20}(\mathcal{R}) \neq \mu_{02}(\mathcal{R})$), the direction $\theta_{\mathcal{R}}$ of the major axis can be found directly from the central moments μ_{pq} as

$$\tan(2\theta_{\mathcal{R}}) = \frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})} \quad (10.27)$$

and thus the corresponding angle is

$$\theta_{\mathcal{R}} = \frac{1}{2} \cdot \tan^{-1} \left(\frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})} \right) \quad (10.28)$$

$$= \frac{1}{2} \cdot \text{ArcTan}(\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R}), 2 \cdot \mu_{11}(\mathcal{R})). \quad (10.29)$$

The resulting angle $\theta_{\mathcal{R}}$ is in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.⁹ Orientation measurements based on region moments are very accurate in general.

Calculating orientation vectors

When visualizing region properties, a frequent task is to plot the region's orientation as a line or arrow, usually anchored at the center of gravity $\bar{\mathbf{x}} = (\bar{x}, \bar{y})^T$; for example, by a parametric line of the form

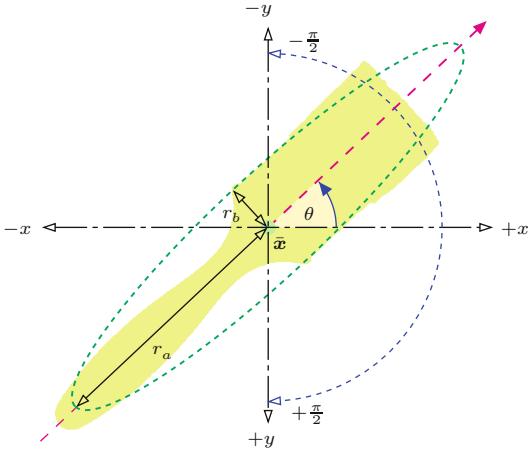
$$\mathbf{x} = \bar{\mathbf{x}} + \lambda \cdot \mathbf{x}_d = \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} + \lambda \cdot \begin{pmatrix} \cos(\theta_{\mathcal{R}}) \\ \sin(\theta_{\mathcal{R}}) \end{pmatrix}, \quad (10.30)$$

with the normalized orientation vector \mathbf{x}_d and the length variable $\lambda > 0$. To find the unit orientation vector $\mathbf{x}_d = (\cos \theta, \sin \theta)^T$, we could first compute the inverse tangent to get 2θ (Eqn. (10.28)) and then compute the cosine and sine of θ . However, the vector \mathbf{x}_d can also be obtained without using trigonometric functions as follows. Rewriting Eqn. (10.27) as

$$\tan(2\theta_{\mathcal{R}}) = \frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})} = \frac{a}{b} = \frac{\sin(2\theta_{\mathcal{R}})}{\cos(2\theta_{\mathcal{R}})}, \quad (10.31)$$

we get (by Pythagora's theorem)

⁹ See Sec. A.1 in the Appendix for the computation of angles with the `ArcTan()` (inverse tangent) function and Sec. F.1.6 for the corresponding Java method `Math.atan2()`.


Fig. 10.18

Region orientation and eccentricity. The major axis of the region extends through its center of gravity \bar{x} at the orientation θ . Note that angles are in the range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ and increment in the *clockwise* direction because the y axis of the image coordinate system points downward (in this example, $\theta \approx -0.759 \approx -43.5^\circ$). The eccentricity of the region is defined as the ratio between the lengths of the major axis (r_a) and the minor axis (r_b) of the “equivalent” ellipse.

$$\sin(2\theta_{\mathcal{R}}) = \frac{a}{\sqrt{a^2+b^2}} \quad \text{and} \quad \cos(2\theta_{\mathcal{R}}) = \frac{b}{\sqrt{a^2+b^2}},$$

where $A = 2\mu_{11}(\mathcal{R})$ and $B = \mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})$. Using the relations $\cos^2\alpha = \frac{1}{2}[1 + \cos(2\alpha)]$ and $\sin^2\alpha = \frac{1}{2}[1 - \cos(2\alpha)]$, we can compute the normalized orientation vector $\mathbf{x}_d = (x_d, y_d)^T$ as

$$x_d = \cos(\theta_{\mathcal{R}}) = \begin{cases} 0 & \text{for } a = b = 0, \\ \left[\frac{1}{2} \cdot \left(1 + \frac{b}{\sqrt{a^2+b^2}} \right) \right]^{\frac{1}{2}} & \text{otherwise,} \end{cases} \quad (10.32)$$

$$y_d = \sin(\theta_{\mathcal{R}}) = \begin{cases} 0 & \text{for } a = b = 0, \\ \left[\frac{1}{2} \cdot \left(1 - \frac{b}{\sqrt{a^2+b^2}} \right) \right]^{\frac{1}{2}} & \text{for } a \geq 0, \\ -\left[\frac{1}{2} \cdot \left(1 - \frac{b}{\sqrt{a^2+b^2}} \right) \right]^{\frac{1}{2}} & \text{for } a < 0, \end{cases} \quad (10.33)$$

straight from the central region moments $\mu_{11}(\mathcal{R})$, $\mu_{20}(\mathcal{R})$, and $\mu_{02}(\mathcal{R})$, as defined in Eqn. (10.31). The horizontal component (x_d) in Eqn. (10.32) is always positive, while the case switch in Eqn. (10.33) corrects the sign of the vertical component (y_d) to map to the same angular range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ as Eqn. (10.28). The resulting vector \mathbf{x}_d is normalized (i.e., $\|(x_d, y_d)\| = 1$) and could be scaled arbitrarily for display purposes by a suitable length λ , for example, using the region’s eccentricity value described in Sec. 10.6.2 (see also Fig. 10.19).

10.6.2 Eccentricity

Similar to the region orientation, moments can also be used to determine the “elongatedness” or *eccentricity* of a region. A naive approach for computing the eccentricity could be to rotate the region until we can fit a bounding box (or enclosing ellipse) with a maximum aspect ratio. Of course this process would be computationally intensive simply because of the many rotations required. If we know the orientation of the region (Eqn. (10.28)), then we may fit a bounding box that is parallel to the region’s major axis. In general, the proportions of the region’s bounding box is not a good eccentricity measure

anyway because it does not consider the distribution of pixels inside the box.

Based on region moments, highly accurate and stable measures can be obtained without any iterative search or optimization. Also, moment-based methods do not require knowledge of the boundary length (as required for computing the circularity feature in Sec. 10.4.2), and they can also handle nonconnected regions or point clouds. Several different formulations of region eccentricity can be found in the literature [15, 126, 128] (see also Exercise 10.17). We adopt the following definition because of its simple geometrical interpretation:

$$\text{Ecc}(\mathcal{R}) = \frac{a_1}{a_2} = \frac{\mu_{20} + \mu_{02} + \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}}{\mu_{20} + \mu_{02} - \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}}, \quad (10.34)$$

where $a_1 = 2\lambda_1$, $a_2 = 2\lambda_2$ are proportional to the eigenvalues λ_1, λ_2 (with $\lambda_1 \geq \lambda_2$) of the symmetric 2×2 matrix

$$\mathbf{A} = \begin{pmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{pmatrix}, \quad (10.35)$$

with the region's central moments $\mu_{11}, \mu_{20}, \mu_{02}$ (see Eqn. (10.23)).¹⁰ The values of Ecc are in the range $[1, \infty)$, where $\text{Ecc} = 1$ corresponds to a circular disk and elongated regions have values > 1 .

The value returned by Ecc(\mathcal{R}) is invariant to the region's orientation and size, that is, this quantity has the important property of being rotation and scale invariant. However, the values a_1, a_2 contain relevant information about the spatial structure of the region. Geometrically, the eigenvalues λ_1, λ_2 (and thus a_1, a_2) directly relate to the proportions of the “equivalent” ellipse, positioned at the region's center of gravity (\bar{x}, \bar{y}) and oriented at $\theta = \theta_{\mathcal{R}}$ Eqn. (10.28). The lengths of the major and minor axes, r_a and r_b , are

$$r_a = 2 \cdot \left(\frac{\lambda_1}{|\mathcal{R}|} \right)^{\frac{1}{2}} = \left(\frac{2a_1}{|\mathcal{R}|} \right)^{\frac{1}{2}}, \quad (10.36)$$

$$r_b = 2 \cdot \left(\frac{\lambda_2}{|\mathcal{R}|} \right)^{\frac{1}{2}} = \left(\frac{2a_2}{|\mathcal{R}|} \right)^{\frac{1}{2}}, \quad (10.37)$$

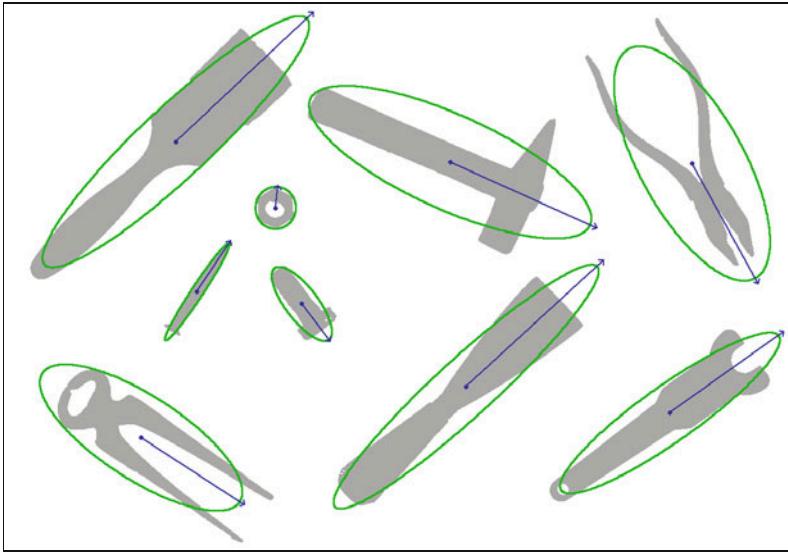
respectively, with a_1, a_2 as defined in Eqn. (10.34) and $|\mathcal{R}|$ being the number of pixels in the region. Given the axes' lengths r_a, r_b and the centroid (\bar{x}, \bar{y}) , the parametric equation of this ellipse is

$$\mathbf{x}(t) = \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} + \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} r_a \cdot \cos(t) \\ r_b \cdot \sin(t) \end{pmatrix} \quad (10.38)$$

$$= \begin{pmatrix} \bar{x} + \cos(\theta) \cdot r_a \cdot \cos(t) - \sin(\theta) \cdot r_b \cdot \sin(t) \\ \bar{y} + \sin(\theta) \cdot r_a \cdot \cos(t) + \cos(\theta) \cdot r_b \cdot \sin(t) \end{pmatrix}, \quad (10.39)$$

for $0 \leq t < 2\pi$. If entirely *filled*, the region described by this ellipse would have the same central moments as the original region \mathcal{R} . Figure 10.19 shows a set of regions with overlaid orientation and eccentricity results.

¹⁰ \mathbf{A} is actually the *covariance matrix* for the distribution of pixel positions inside the region (see Sec. D.2 in the Appendix).



10.6 MOMENT-BASED GEOMETRIC PROPERTIES

Fig. 10.19

Orientation and eccentricity examples. The orientation θ (Eqn. (10.28)) is displayed for each connected region as a vector with the length proportional to the region's eccentricity value $Ecc(\mathcal{R})$ (Eqn. (10.34)). Also shown are the ellipses (Eqns. (10.36) and (10.37)) corresponding to the orientation and eccentricity parameters.

10.6.3 Bounding Box Aligned to the Major Axis

While the ordinary, x/y axis-aligned bounding box (see Sec. 10.4.2) is of little practical use (because it is sensitive to rotation), it may be interesting to see how to find a region's bounding box that is aligned with its major axis, as defined in Sec. 10.6.1. Given a region's orientation angle $\theta_{\mathcal{R}}$,

$$\mathbf{e}_a = \begin{pmatrix} x_a \\ y_a \end{pmatrix} = \begin{pmatrix} \cos(\theta_{\mathcal{R}}) \\ \sin(\theta_{\mathcal{R}}) \end{pmatrix} \quad (10.40)$$

is the unit vector parallel to its major axis; thus

$$\mathbf{e}_b = \mathbf{e}_a^\perp = \begin{pmatrix} y_a \\ -x_a \end{pmatrix} \quad (10.41)$$

is the unit vector orthogonal to \mathbf{e}_a .¹¹ The bounding box can now be determined as follows (see Fig. 10.20):

1. Project each region point¹² $\mathbf{u}_i = (u_i, v_i)$ onto the vector \mathbf{e}_a (parallel to the region's major axis) by calculating the dot product¹³

$$a_i = \mathbf{u}_i \cdot \mathbf{e}_a \quad (10.42)$$

and keeping the minimum and maximum values

$$a_{\min} = \min_{\mathbf{u}_i \in \mathcal{R}} a_i, \quad a_{\max} = \max_{\mathbf{u}_i \in \mathcal{R}} a_i. \quad (10.43)$$

2. Analogously, project each region point \mathbf{u}_i onto the *orthogonal axis* (specified by the vector \mathbf{e}_b) by

¹¹ $\mathbf{x}^\perp = \text{perp}(\mathbf{x}) = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \cdot \mathbf{x}$.

¹² Of course, if the region's contour is available, it is sufficient to iterate over the contour points only.

¹³ See Sec. B.3.1, Eqn. (B.19) in the Appendix.

10 REGIONS IN BINARY IMAGES

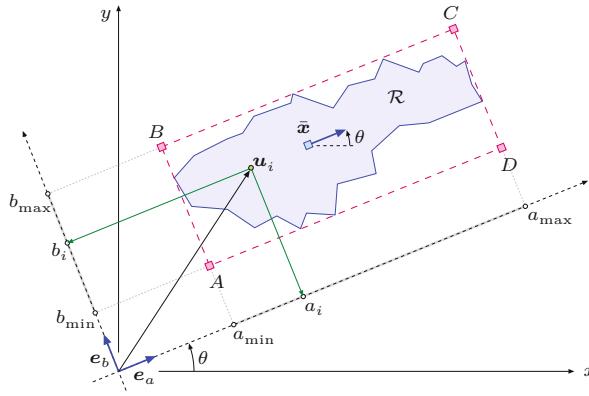
Fig. 10.20

Calculation of a region's major axis-aligned bounding box.

The unit vector e_a is parallel to the region's major axis (oriented at angle θ); e_b is perpendicular to e_a . The projection of a region point u_i onto the lines defined by e_a and e_b yields the lengths a_i and b_i , respectively (measured from the coordinate origin).

The resulting quantities a_{\min} , a_{\max} , b_{\min} , b_{\max} define the corner points (A, B, C, D) of the axis-aligned bounding box.

Note that the position of the region's centroid (\bar{x}) is not required in this calculation.



$$b_i = \mathbf{u}_i \cdot \mathbf{e}_b \quad (10.44)$$

and keeping the minimum and maximum values, that is,

$$b_{\min} = \min_{\mathbf{u}_i \in \mathcal{R}} b_i, \quad b_{\max} = \max_{\mathbf{u}_i \in \mathcal{R}} b_i. \quad (10.45)$$

Note that steps 1 and 2 can be performed in a single iteration over all region points.

3. Finally, from the resulting quantities a_{\min} , a_{\max} , b_{\min} , b_{\max} , calculate the four corner points A, B, C, D of the bounding box as

$$\begin{aligned} A &= a_{\min} \cdot \mathbf{e}_a + b_{\min} \cdot \mathbf{e}_b, & B &= a_{\min} \cdot \mathbf{e}_a + b_{\max} \cdot \mathbf{e}_b, \\ C &= a_{\max} \cdot \mathbf{e}_a + b_{\max} \cdot \mathbf{e}_b, & D &= a_{\max} \cdot \mathbf{e}_a + b_{\min} \cdot \mathbf{e}_b. \end{aligned} \quad (10.46)$$

The complete calculation is summarized in Alg. 10.20; a typical example is shown in Fig. 10.21(d).

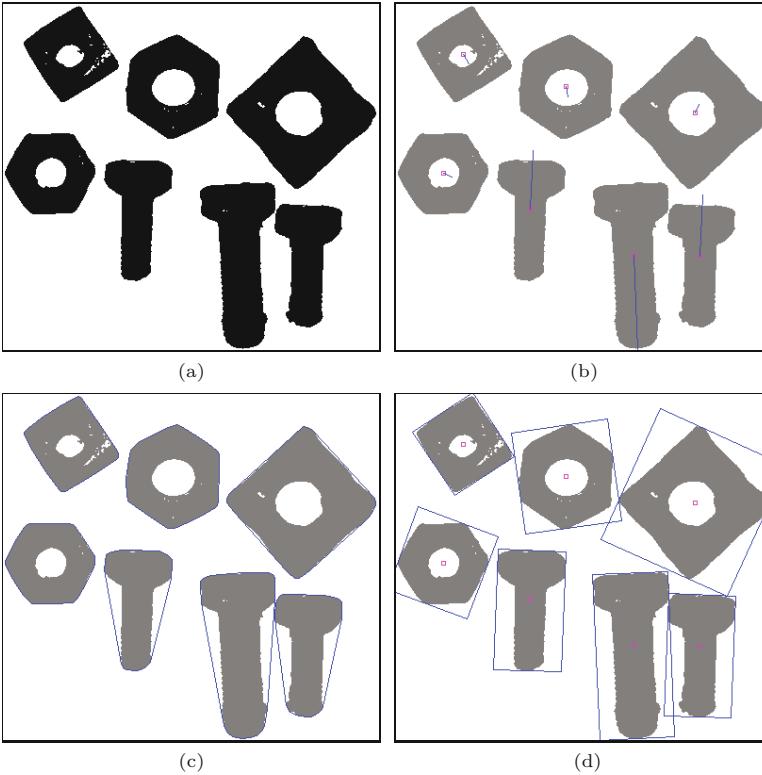
Alg. 10.5

Calculation of the major axis-aligned bounding box for a binary region \mathcal{R} . If the region's contour is available, it is sufficient to use the contour points only.

```

1: MajorAxisAlignedBoundingBox( $\mathcal{R}$ )
   Input:  $\mathcal{R} = \{\mathbf{u}_i\}$ , a binary region containing points  $\mathbf{u}_i \in \mathbb{R}^2$ .
   Returns the four corner points of the region's bounding box.
2:  $\theta \leftarrow 0.5 \cdot \text{ArcTan}(\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R}), 2 \cdot \mu_{11}(\mathcal{R}))$   $\triangleright$  see Eq. 10.28
3:  $\mathbf{e}_a \leftarrow (\cos(\theta), \sin(\theta))^T$   $\triangleright$  unit vector paralle. to region's major axis
4:  $\mathbf{e}_b \leftarrow (\sin(\theta), -\cos(\theta))^T$   $\triangleright$  unit vector perpendic. to major axis
5:  $a_{\min} \leftarrow \infty, \quad a_{\max} \leftarrow -\infty$ 
6:  $b_{\min} \leftarrow \infty, \quad b_{\max} \leftarrow -\infty$ 
7: for all  $\mathbf{u} \in \mathcal{R}$  do
8:    $a \leftarrow \mathbf{u} \cdot \mathbf{e}_a$   $\triangleright$  project  $\mathbf{u}$  onto  $\mathbf{e}_a$  (Eq. 10.42)
9:    $a_{\min} \leftarrow \min(a_{\min}, a)$ 
10:   $a_{\max} \leftarrow \max(a_{\max}, a)$ 
11:   $b \leftarrow \mathbf{u} \cdot \mathbf{e}_b$   $\triangleright$  project  $\mathbf{u}$  onto  $\mathbf{e}_b$  (Eq. 10.44)
12:   $b_{\min} \leftarrow \min(b_{\min}, b)$ 
13:   $b_{\max} \leftarrow \max(b_{\max}, b)$ 
14:   $A \leftarrow a_{\min} \cdot \mathbf{e}_a + b_{\min} \cdot \mathbf{e}_b$ 
15:   $B \leftarrow a_{\min} \cdot \mathbf{e}_a + b_{\max} \cdot \mathbf{e}_b$ 
16:   $C \leftarrow a_{\max} \cdot \mathbf{e}_a + b_{\max} \cdot \mathbf{e}_b$ 
17:   $D \leftarrow a_{\max} \cdot \mathbf{e}_a + b_{\min} \cdot \mathbf{e}_b$ 
18: return  $(A, B, C, D)$   $\triangleright$  corners of the bounding box

```



10.6 MOMENT-BASED GEOMETRIC PROPERTIES

Fig. 10.21

Geometric region properties. Original binary image (a), centroid and orientation vector (length determined by the region's eccentricity) of the major axis (b), convex hull (c), and major axis-aligned bounding box (d).

10.6.4 Invariant Region Moments

Normalized central moments are not affected by the translation or uniform scaling of a region (i.e., the values are invariant), but in general rotating the image will change these values.

Hu's invariant moments

A classical solution to this problem is a clever combination of simpler features known as “Hu’s Moments” [112]:¹⁴

$$\begin{aligned}
 \phi_1 &= \bar{\mu}_{20} + \bar{\mu}_{02}, \\
 \phi_2 &= (\bar{\mu}_{20} - \bar{\mu}_{02})^2 + 4\bar{\mu}_{11}^2, \\
 \phi_3 &= (\bar{\mu}_{30} - 3\bar{\mu}_{12})^2 + (3\bar{\mu}_{21} - \bar{\mu}_{03})^2, \\
 \phi_4 &= (\bar{\mu}_{30} + \bar{\mu}_{12})^2 + (\bar{\mu}_{21} + \bar{\mu}_{03})^2, \\
 \phi_5 &= (\bar{\mu}_{30} - 3\bar{\mu}_{12}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot [(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3(\bar{\mu}_{21} + \bar{\mu}_{03})^2] + \\
 &\quad (3\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot [3(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2], \\
 \phi_6 &= (\bar{\mu}_{20} - \bar{\mu}_{02}) \cdot [(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2] + \\
 &\quad 4\bar{\mu}_{11} \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}), \\
 \phi_7 &= (3\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot [(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3(\bar{\mu}_{21} + \bar{\mu}_{03})^2] + \\
 &\quad (3\bar{\mu}_{12} - \bar{\mu}_{30}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot [3(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2].
 \end{aligned} \tag{10.47}$$

¹⁴ In order to improve the legibility of Eqn. (10.47) the argument for the region (\mathcal{R}) has been dropped; as an example, with the region argument, the first line would read $H_1(\mathcal{R}) = \bar{\mu}_{20}(\mathcal{R}) + \bar{\mu}_{02}(\mathcal{R})$, and so on.

In practice, the logarithm of these quantities (that is, $\log(\phi_k)$) is used since the raw values may have a very large range. These features are also known as *moment invariants* since they are invariant under translation, rotation, and scaling. While defined here for binary images, they are also applicable to parts of grayscale images; examples can be found in [88, p. 517].

Flusser's invariant moments

It was shown in [72, 73] that Hu's moments, as listed in Eqn. (10.47), are partially redundant and incomplete. Based on so-called *complex moments* $c_{pq} \in \mathbb{C}$, Flusser designed an improved set of 11 rotation and scale-invariant features ψ_1, \dots, ψ_{11} (see Eqn. (10.51)) for characterizing 2D shapes. For grayscale images (with $I(u, v) \in \mathbb{R}$), the complex moments of order p, q are defined as

$$c_{pq}(\mathcal{R}) = \sum_{(u,v) \in \mathcal{R}} I(u, v) \cdot (x + i \cdot y)^p \cdot (x - i \cdot y)^q, \quad (10.48)$$

with centered positions $x = u - \bar{x}$ and $y = v - \bar{y}$, and (\bar{x}, \bar{y}) being the *centroid* of \mathcal{R} (i denotes the imaginary unit). In the case of binary images (with $I(u, v) \in [0, 1]$) Eqn. (10.48) simplifies to

$$c_{pq}(\mathcal{R}) = \sum_{(u,v) \in \mathcal{R}} (x + i \cdot y)^p \cdot (x - i \cdot y)^q. \quad (10.49)$$

Analogous to Eqn. (10.26), the complex moments can be *scale-normalized* to

$$\hat{c}_{p,q}(\mathcal{R}) = \frac{1}{A^{(p+q+2)/2}} \cdot c_{p,q}, \quad (10.50)$$

with A being the area of \mathcal{R} [74, p. 29]. Finally, the derived rotation and scale invariant region moments of 2nd to 4th order are¹⁵

$$\begin{aligned} \psi_1 &= \operatorname{Re}(\hat{c}_{1,1}), & \psi_2 &= \operatorname{Re}(\hat{c}_{2,1} \cdot \hat{c}_{1,2}), & \psi_3 &= \operatorname{Re}(\hat{c}_{2,0} \cdot \hat{c}_{1,2}^2), \\ \psi_4 &= \operatorname{Im}(\hat{c}_{2,0} \cdot \hat{c}_{1,2}^2), & \psi_5 &= \operatorname{Re}(\hat{c}_{3,0} \cdot \hat{c}_{1,2}^3), & \psi_6 &= \operatorname{Im}(\hat{c}_{3,0} \cdot \hat{c}_{1,2}^3), \\ \psi_7 &= \operatorname{Re}(\hat{c}_{2,2}), & \psi_8 &= \operatorname{Re}(\hat{c}_{3,1} \cdot \hat{c}_{1,2}^2), & \psi_9 &= \operatorname{Im}(\hat{c}_{3,1} \cdot \hat{c}_{1,2}^2), \\ \psi_{10} &= \operatorname{Re}(\hat{c}_{4,0} \cdot \hat{c}_{1,2}^4), & \psi_{11} &= \operatorname{Im}(\hat{c}_{4,0} \cdot \hat{c}_{1,2}^4). \end{aligned} \quad (10.51)$$

Table 10.1 lists the normalized Flusser moments for five binary shapes taken from the Kimia dataset [134].

Shape matching with region moments

One obvious use of invariant region moments is shape matching and classification. Given two binary shapes A and B , with associated moment (“feature”) vectors

$$\mathbf{f}_A = (\psi_1(A), \dots, \psi_{11}(A)) \quad \text{and} \quad \mathbf{f}_B = (\psi_1(B), \dots, \psi_{11}(B)),$$

respectively, one approach could be to simply measure the difference between shapes by the Euclidean distance of these vectors in the form

¹⁵ In Eqn. (10.51), the use of $\operatorname{Re}()$ for the quantities ψ_1, ψ_2, ψ_7 (which are real-valued *per se*) is redundant.



	ψ_1	ψ_2	ψ_3	ψ_4	ψ_5	ψ_6	ψ_7	ψ_8	ψ_9	ψ_{10}	ψ_{11}
	0.3730017575	0.2545476083	0.2154034257	0.2124041195	0.3600613700						
ψ_2	0.0012699373	0.0004247053	0.0002068089	0.0001089652	0.0017187073						
ψ_3	0.0004041515	0.0000644829	0.0000274491	0.0000014248	-0.0003853999						
ψ_4	0.0000097827	-0.0000076547	0.0000071688	-0.0000022103	-0.0001944121						
ψ_5	0.0000012672	0.0000002327	0.0000000637	0.0000000083	-0.0000078073						
ψ_6	0.00000001090	-0.0000000483	0.0000000041	0.0000000153	-0.0000061997						
ψ_7	0.2687922057	0.1289708408	0.0814034374	0.0712567626	0.2340886626						
ψ_8	0.0003192443	0.0000414818	0.0000134036	0.0000003020	-0.0002878997						
ψ_9	0.0000053208	-0.0000032541	0.0000030880	-0.0000008365	-0.0001628669						
ψ_{10}	0.00000103461	0.0000000991	0.0000000019	-0.0000000003	0.0000001922						
ψ_{11}	0.0000000120	-0.0000000020	0.0000000008	-0.0000000000	0.0000003015						

	0.000	0.183	0.245	0.255	0.037
	0.183	0.000	0.062	0.071	0.149
	0.245	0.062	0.000	0.011	0.210
	0.255	0.071	0.011	0.000	0.220
	0.037	0.149	0.210	0.220	0.000

$$d_E(A, B) = \|\mathbf{f}_A - \mathbf{f}_B\| = \left[\sum_{i=1}^{11} |\psi_i(A) - \psi_i(B)|^2 \right]^{1/2}. \quad (10.52)$$

Concrete distances between the five sample shapes are listed in Table 10.2. Since the moment vectors are rotation and scale invariant,¹⁶ shape comparisons should remain unaffected by such transformations. Note, however, that the magnitude of the individual moments varies over a very large range. Thus, if the Euclidean distance is used as we have just suggested, the comparison (matching) of shapes is typically dominated by a few moments (or even a single moment) of relatively large magnitude, while the small-valued moments play virtually no role in the distance calculation. This is because the Euclidean distance treats the multi-dimensional feature space uniformly along all dimensions.

As a consequence, moment-based shape discrimination with the ordinary Euclidean distance is typically not very selective. A simple solution is to replace Eqn. (10.52) by a *weighted distance* measure of the form

$$d'_E(A, B) = \left[\sum_{i=1}^{11} w_i \cdot |\psi_i(A) - \psi_i(B)|^2 \right]^{1/2}, \quad (10.53)$$

with fixed weights $w_1, \dots, w_{11} \geq 0$ assigned to each each moment feature to compensate for the differences in magnitude.

A more elegant approach is to use of the *Mahalanobis* distance [24, 157] for comparing the moment vectors, which accounts for the statistical distribution of each vector component and avoids large-magnitude components dominating the smaller ones. In this case,

¹⁶ Although the invariance property holds perfectly for continuous shapes, rotating and scaling *discrete* binary images may significantly affect the associated region moments.

10.6 MOMENT-BASED GEOMETRIC PROPERTIES

Table 10.1

Binary shapes and associated normalized Flusser moments ψ_1, \dots, ψ_{11} . Notice the magnitude of the moments varies by a large factor.

Table 10.2

Inter-class (Euclidean) distances $d_E(A, B)$ between normalized shape feature vectors for the five reference shapes (see Eqn. (10.52)). Off-diagonal values should be consistently large to allow good shape discrimination.

the distance calculation becomes

$$d_M(A, B) = [(\mathbf{f}_A - \mathbf{f}_B)^T \cdot \Sigma^{-1} \cdot (\mathbf{f}_A - \mathbf{f}_B)]^{1/2}, \quad (10.54)$$

where Σ is the 11×11 covariance matrix for the moment vectors \mathbf{f} . Note that the expression under the root in Eqn. (10.54) is the dot product of a row vector and a column vector, that is, the result is a non-negative scalar value. The Mahalanobis distance can be viewed as a special form of the weighted Euclidean distance (Eqn. (10.53)), where the weights are determined by the variability of the individual vector components. See Sec. D.3 in the Appendix and Exercise 10.16 for additional details.

10.7 Projections

Image projections are 1D representations of the image contents, usually calculated parallel to the coordinate axis. In this case, the horizontal and vertical projections of a scalar-valued image $I(u, v)$ of size $M \times N$ are defined as

$$P_{\text{hor}}(v) = \sum_{u=0}^{M-1} I(u, v) \quad \text{for } 0 < v < N, \quad (10.55)$$

$$P_{\text{ver}}(u) = \sum_{v=0}^{N-1} I(u, v) \quad \text{for } 0 < u < M. \quad (10.56)$$

The *horizontal* projection $P_{\text{hor}}(v_0)$ (Eqn. (10.55)) is the sum of the pixel values in the image *row* v_0 and has length N corresponding to the height of the image. On the other hand, a *vertical* projection P_{ver} of length M is the sum of all the values in the image *column* u_0 (Eqn. (10.56)). In the case of a binary image with $I(u, v) \in \{0, 1\}$, the projection contains the count of the foreground pixels in the corresponding image row or column.

Program 10.4 gives a direct implementation of the projection calculations as the `run()` method for an ImageJ plugin, where projections in both directions are computed during a single traversal of the image.

Projections in the direction of the coordinate axis are often utilized to quickly analyze the structure of an image and isolate its component parts; for example, in document images it is used to separate graphic elements from text blocks as well as to isolate individual lines (see the example in Fig. 10.22). In practice, especially to account for document skew, projections are often computed along the major axis of an image region Eqn. (10.28). When the projection vectors of a region are computed in reference to the centroid of the region along the major axis, the result is a rotation-invariant vector description (often referred to as a “signature”) of the region.

10.8 Topological Region Properties

Topological features do not describe the shape of a region in continuous terms; instead, they capture its structural properties. Topological

```

1  public void run(ImageProcessor I) {
2      int M = I.getWidth();
3      int N = I.getHeight();
4      int[] pHor = new int[N]; // = Phor(v)
5      int[] pVer = new int[M]; // = Pver(u)
6      for (int v = 0; v < N; v++) {
7          for (int u = 0; u < M; u++) {
8              int p = I.getPixel(u, v);
9              pHor[v] += p;
10             pVer[u] += p;
11         }
12     } // use projections pHor, pVer now
13     // ...
14 }
```

10.8 TOPOLOGICAL REGION PROPERTIES

Prog. 10.4

Calculation of horizontal and vertical projections. The `run()` method for an ImageJ plugin (`ip` is of type `ByteProcessor` or `ShortProcessor`) computes the projections in x and y directions simultaneously in a single traversal of the image. The projections are represented by the one-dimensional arrays `horProj` and `verProj` with elements of type `int`.



Fig. 10.22
Horizontal and vertical projections of a binary image.

properties are typically invariant even under strong image transformations. The convexity of a region, which can be calculated from the convex hull (Sec. 10.4.2), is also a topological property.

A simple and robust topological feature is the *number of holes* $N_L(\mathcal{R})$ in a region. This feature is easily determined while finding the inner contours of a region, as described in Sec. 10.2.2.

A useful topological feature that can be derived directly from the number of holes is the so-called *Euler number* N_E , which is the difference between the number of connected regions N_R and the number of their holes N_L , that is,

$$N_E(\mathcal{R}) = N_R(\mathcal{R}) - N_L(\mathcal{R}). \quad (10.57)$$

In the case of a single connected region this is simply $1 - N_L$. For a picture of the number “8”, for example, $N_E = 1 - 2 = -1$ and for the letter “D” we get $N_E = 1 - 1 = 0$.

Topological features are often used in combination with numerical features for classification. A classic example of this combination is OCR (optical character recognition) [38]. Figure 10.23 shows an

10 REGIONS IN BINARY IMAGES

Fig. 10.23
Visual identification markers composed of recursively nested regions [22].



interesting use of topological structures for coding optical markers used in augmented reality applications [22].¹⁷ The recursive nesting of outer and inner regions is equivalent to a tree structure that allows fast and unique identification of a larger number of known patterns (see also Exercise 10.21).

10.9 Java Implementation

Most algorithms described in this chapter are implemented as part of the `imagingbook` library.¹⁸ The key classes are `BinaryRegion` and `Contour`, the abstract class `RegionLabeling` and its concrete subclasses `RecursiveLabeling`, `BreadthFirstLabeling`, `DepthFirstLabeling` (Alg. 10.1) and `SequentialLabeling` (Alg. 10.2). The combined region labeling and contour tracing method (Algs. 10.3 and 10.4) is implemented by class `RegionContourLabeling`. Additional details can be found in the online documentation.

Example

A complete example for the use of this API is shown in Prog. 10.5. Particularly useful is the facility for visiting all positions of a specific region using the iterator returned by method `getRegionPoints()`, as demonstrated by this code segment:

```
RegionLabeling segmenter = ....
// Get the largest region:
BinaryRegion r = segmenter.getRegions(true).get(0);
// Loop over all points of region r:
for (Point p : r.getRegionPoints()) {
    int u = p.x;
    int v = p.y;
    // do something with position u, v
}
```

10.10 Exercises

Exercise 10.1. Manually simulate the execution of both variations (*depth-first* and *breadth-first*) of the flood-fill algorithm using the image in Fig. 10.24 and starting at position (5, 1).

¹⁷ <http://reactivision.sourceforge.net/>.

¹⁸ Package `imagingbook.pub.regions`.

```

1 ...
2 import imagingbook.pub.regions.BinaryRegion;
3 import imagingbook.pub.regions.Contour;
4 import imagingbook.pub.regions.ContourOverlay;
5 import imagingbook.pub.regions.RegionContourLabeling;
6 import java.awt.geom.Point2D;
7 import java.util.List;
8
9 public class Region_Countours_Demo implements PlugInFilter {
10
11    public int setup(String arg, ImagePlus im) {
12        return DOES_8G + NO_CHANGES;
13    }
14
15    public void run(ImageProcessor ip) {
16        // Make sure we have a proper byte image:
17        ByteProcessor bp = ip.convertToByteProcessor();
18
19        // Create the region labeler / contour tracer:
20        RegionContourLabeling segmenter =
21            new RegionContourLabeling(bp);
22
23        // Get the list of detected regions (sort by size):
24        List<BinaryRegion> regions =
25            segmenter.getRegions(true);
26        if (regions.isEmpty()) {
27            IJ.error("No regions detected!");
28            return;
29        }
30
31        // List all regions:
32        IJ.log("Detected regions: " + regions.size());
33        for (BinaryRegion r: regions) {
34            IJ.log(r.toString());
35        }
36
37        // Get the outer contour of the largest region:
38        BinaryRegion largestRegion = regions.get(0);
39        Contour oc = largestRegion.getOuterContour();
40        IJ.log("Points on outer contour of largest region:");
41        Point2D[] points = oc.getPointArray();
42        for (int i = 0; i < points.length; i++) {
43            Point2D p = points[i];
44            IJ.log("Point " + i + ": " + p.toString());
45        }
46
47        // Get all inner contours of the largest region:
48        List<Contour> ics = largestRegion.getInnerContours();
49        IJ.log("Inner regions (holes): " + ics.size());
50    }
51 }

```

10.10 EXERCISES

Prog. 10.5

Complete example for the use of the `regions` API. The ImageJ plugin `Region_Countours_Demo` segments the binary (8-bit grayscale) image `ip` into connected components. This is done with an instance of class `RegionContourLabeling` (see line 21), which also extracts the regions' contours. In line 25, a list of regions (sorted by size) is produced which is subsequently traversed (line 33). The treatment of outer and inner contours as well as the iteration over individual contour points is shown in lines 38–49.

10 REGIONS IN BINARY IMAGES

Fig. 10.24
Binary image for Exercise 10.1.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	1	1	0	0	1	1	0	1	0	
2	0	1	1	1	1	1	1	0	0	1	0	0	1	0	
3	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0
4	0	1	1	1	1	1	1	1	1	1	1	1	1	0	
5	0	0	0	0	1	1	1	1	1	1	1	1	1	0	
6	0	1	1	0	0	0	1	0	1	0	0	0	0	0	
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

0 Background
1 Foreground

Exercise 10.2. The implementation of the flood-fill algorithm in Prog. 10.1 places all the neighboring pixels of each visited pixel into either the *stack* or the *queue* without ensuring they are foreground pixels and that they lie within the image boundaries. The number of items in the stack or the queue can be reduced by ignoring (not inserting) those neighboring pixels that do not meet the two conditions given. Modify the *depth-first* and *breadth-first* variants given in Prog. 10.1 accordingly and compare the new running times.

Exercise 10.3. The implementations of depth-first and breadth-first labeling shown in Prog. 10.1 will run significantly slower than the recursive version because the frequent creation of new `Point` objects is quite time consuming. Modify the *depth-first* version of Prog. 10.1 to use a stack with elements of a *primitive type* (e.g., `int`) instead. Note that (at least in Java)¹⁹ it is not possible to specify a built-in list structure (such as `Deque` or `LinkedList`) for a primitive element type. Implement your own stack class that internally uses an `int`-array to store the (u, v) coordinates. What is the maximum number of stack entries needed for a given image of size $M \times N$? Compare the performance of your solution to the original version in Prog. 10.1.

Exercise 10.4. Implement an ImageJ plugin that encodes a given binary image by run length encoding (Sec. 10.3.2) and stores it in a file. Develop a second plugin that reads the file and reconstructs the image.

Exercise 10.5. Calculate the amount of memory required to represent a contour with 1000 points in the following ways: (a) as a sequence of coordinate points stored as pairs of `int` values; (b) as an 8-chain code using Java `byte` elements, and (c) as an 8-chain code using only 3 bits per element.

Exercise 10.6. Implement a Java class for describing a binary image region using chain codes. It is up to you, whether you want to use an absolute or differential chain code. The implementation should be able to encode closed contours as chain codes and also reconstruct the contours given a chain code.

Exercise 10.7. The *Graham Scan* method [91] is an efficient algorithm for calculating the convex hull of a 2D point set (of size n), with time complexity $\mathcal{O}(n \cdot \log(n))$.²⁰ Implement this algorithm and show that it is sufficient to consider only the outer contour points of a region to calculate its convex hull.

¹⁹ Other languages like C# allow this.

²⁰ See also http://en.wikipedia.org/wiki/Graham_scan.

Exercise 10.8. While computing the convex hull of a region, the maximal diameter (maximum distance between two arbitrary points) can also be simply found. Devise an alternative method for computing this feature without using the convex hull. Determine the running time of your algorithm in terms of the number of points in the region.

10.10 EXERCISES

Exercise 10.9. Implement an algorithm for comparing contours using their shape numbers Eqn. (10.6). For this purpose, develop a metric for measuring the distance between two normalized chain codes. Describe if, and under which conditions, the results will be reliable.

Exercise 10.10. Sketch the contour equivalent to the *absolute* chain code sequence $c'_R = (6, 7, 7, 1, 2, 0, 2, 3, 5, 4, 4)$. (a) Choose an arbitrary starting point and determine if the resulting contour is closed. (b) Find the associated *differential* chain code c''_R (Eqn. (10.5)).

Exercise 10.11. Calculate (under assumed 8-neighborhood) the *shape number* of base $b = 8$ (see Eqn. (10.6)) for the differential chain code $c''_R = (1, 0, 2, 1, 6, 2, 1, 2, 7, 0, 2)$ and all possible circular shifts of this code. Which shift yields the maximum arithmetic value?

Exercise 10.12. Using Eqn. (10.14) as the basis, develop and implement an algorithm that computes the area of a region from its 8-chain-encoded contour (see also [263], [127, Sec. 19.5]).

Exercise 10.13. Modify Alg. 10.3 such that the outer and inner contours are not returned as individual lists ($C_{\text{out}}, C_{\text{in}}$) but as a composite tree structure. An outer contour thus represents a region that may contain zero, one, or more inner contours (i.e., holes). Each inner contour may again contain other regions (i.e., outer contours), and so on.

Exercise 10.14. Sketch an example binary region where the centroid does not lie inside the region itself.

Exercise 10.15. Implement the binary region moment features proposed by *Hu* (Eqn. (10.47)) and/or *Flusser* (Eqn. (10.51)) and verify that they are invariant under image scaling and rotation. Use the test image in Fig. 10.25²¹ (or create your own), which contains rotated and mirrored instances of the reference shapes, in addition to other (unknown) shapes.

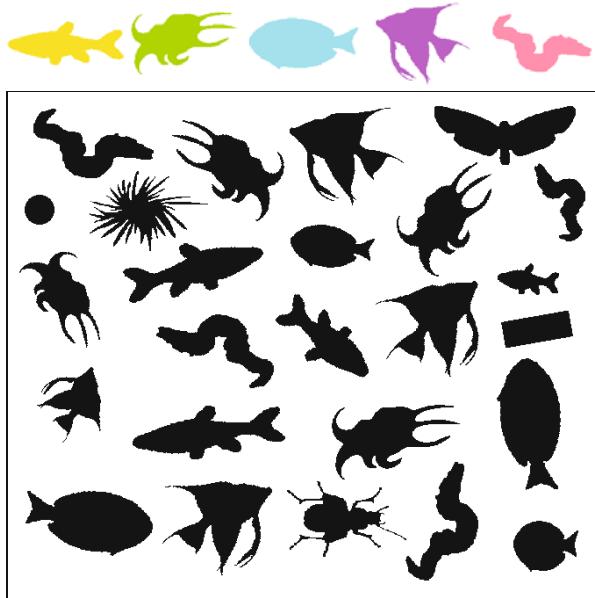
Exercise 10.16. Implement the Mahalanobis distance calculation, as defined in Eqn. (10.54), for measuring the similarity between shape moment vectors.

- A. Compute the covariance matrix Σ (see Sec. D.3 in the Appendix) for the $m = 11$ Flusser shape features ψ_1, \dots, ψ_{11} of the reference images in Table 10.1. Calculate and tabulate the inter-class Mahalanobis distances for the reference shapes, analogous to the example in Table 10.2.

²¹ Images are available on the book's website.

Fig. 10.25

Test image for moment-based shape matching. Reference shapes (top) and test image (bottom) composed of rotated and/or scaled shapes from the Kimia database and additional (unclassified) shapes.



- B.** Extend your analysis to a larger set of 500–1000 shapes (e.g., from the Kimia dataset [134], which contains more than 20 000 binary shape images). Calculate the normalized moment features and the covariance matrix Σ for the entire image set. Calculate the inter-class distance matrices for (a) the Euclidean and (b) the Mahalanobis distance. Display the distance matrices as grayscale images (`FloatProcessor`) and interpret them.

Exercise 10.17. There are alternative definitions for the *eccentricity* of a region Eqn. (10.34); for example [128, p. 394],

$$\text{Ecc}_2(\mathcal{R}) = \frac{[\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})]^2 + 4 \cdot \mu_{11}^2(\mathcal{R})}{[\mu_{20}(\mathcal{R}) + \mu_{02}(\mathcal{R})]^2}. \quad (10.58)$$

Implement this version as well as the one in Eqn. (10.34) and contrast the results using suitably designed regions. Determine the numeric range of these quantities and test if they are really rotation and scale-invariant.

Exercise 10.18. Write an ImageJ plugin that (a) finds (labels) all regions in a binary image, (b) computes the orientation and eccentricity for each region, and (c) shows the results as a direction vector and the equivalent ellipse on top of each region (as exemplified in Fig. 10.19). Hint: Use Eqn. (10.39) to develop a method for drawing ellipses at arbitrary orientations (not available in ImageJ).

Exercise 10.19. The Java method in Prog. 10.4 computes an image's horizontal and vertical projections. The scheme described in Sec. 10.6.3 and illustrated in Fig. 10.20 can be used to calculate projections along arbitrary directions θ . Develop and implement such a process and display the resulting projections.

Exercise 10.20. Text recognition (OCR) methods are likely to fail if the document image is not perfectly axis-aligned. One method for estimating the skew angle of a text document is to perform binary segmentation and connected components analysis (see Fig. 10.26):

10.10 EXERCISES

- *Smear* the original binary image by applying a disk-shaped morphological dilation with a specified radius (see Chapter 9, Sec. 9.2.3). The aim is to close the gaps between neighboring glyphs without closing the space between adjacent text lines (Fig. 10.26(b))
- Apply region segmentation to the resulting image and calculate the orientation $\theta(\mathcal{R})$ and the eccentricity $E(\mathcal{R})$ of each region \mathcal{R} (see Secs. 10.6.1 and 10.6.2). Ignore all regions that are either too small or not sufficiently elongated.
- Estimate the global skew angle by averaging the regions' orientations θ_i . Note that, since angles are *circular*, they cannot be averaged in the usual way (see Chapter 15, Eqn. (15.14) for how to calculate the mean of a circular quantity). Consider using the eccentricity as a weight for the contribution of the associated region to the global average.
- Obviously, this scheme is sensitive to *outliers*, that is, against angles that deviate strongly from the average orientation. Try to improve this estimate (i.e., make it more robust and accurate) by iteratively removing angles that are “too far” from the average orientation and then recalculating the result.

Exercise 10.21. Draw the tree structure, defined by the recursive nesting of outer and inner regions, for each of the markers shown in Fig. 10.23. Based on this graph structure, suggest an algorithm for matching pairs of markers or, alternatively, for retrieving the best-matching marker from a database of markers.

10 REGIONS IN BINARY IMAGES

Fig. 10.26

Document skew estimation example (see Exercise 10.20). Original binary image (a); result of applying a disk-shaped morphological dilation with radius 3.0 (b); region orientation vectors (c); histogram of the orientation angle θ (d).

The real skew angle in this scan is approximately 1.1° .

As President Eisenhower once said, nuclear weapons are the only thing that can destroy the United States. Americans want to know how the next president plans to control the thousands of these weapons of mass destruction that exist in the world.

It's worth remembering that in October 1986 President Ronald Reagan was meeting with Soviet President Mikhail Gorbachev in Reykjavik, Iceland, to discuss eliminating nuclear weapons.

The two leaders focused on nuclear weapons testing. If you are serious about total nuclear disarmament, you have to end testing first. As Reagan wrote then, "I am committed to the ultimate attainment of a total ban on nuclear testing, a goal that has been endorsed by every U.S. president since President Eisenhower."

But Reagan had some prerequisites. In 1986 the United States Senate had yet to ratify two treaties that had been negotiated with the Soviets: the Threshold Test Ban, which limited the size of underground

buried underground tests for peaceful purposes. Reagan wanted to get these treaties ratified first, and that meant making sure the agreements could not be cheated on by secret tests. As Reagan like to say "Trust, but verify."

In 1990, after Reagan had left office, both the Threshold Test Ban and the Peaceful Nuclear Explosions Treaty were ratified by the Senate after satisfactory review of the verification provisions. Reagan's first requirement on the road to a nuclear test ban was complete.

Reagan's second requirement for ending nuclear testing was that the Soviets and the Americans should reduce their nuclear stockpiles. That effort started with the 1987 Intermediate-Range Nuclear Forces Treaty, which eliminated medium- and short-range nuclear missiles. The Strategic Arms Reduction Talks (START) treaties subsequently continued U.S. and Russian reductions, although thousands still remain.

In 1996 the Comprehensive Nuclear Test Ban Treaty was crafted to ban all nuclear test

and underground tests for peaceful purposes. Reagan wanted to get these treaties ratified first, and that meant making sure the agreements could not be cheated on by secret tests. As Reagan like to say "Trust, but verify."

In 1990, after Reagan had left office, both the Threshold Test Ban and the Peaceful Nuclear Explosions Treaty were ratified by the Senate after satisfactory review of the verification provisions. Reagan's first requirement on the road to a nuclear test ban was complete.

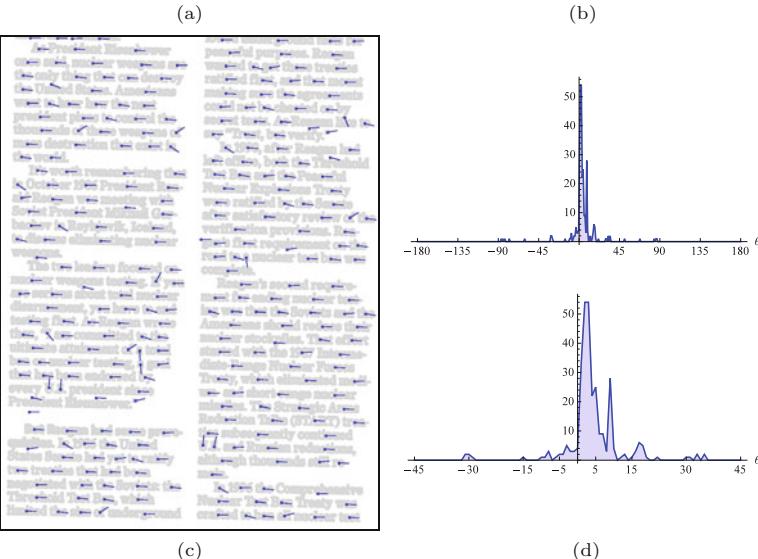
Reagan's second requirement for ending nuclear testing was that the Soviets and the Americans should reduce their nuclear stockpiles. That effort started with the 1987 Intermediate-Range Nuclear Forces Treaty, which eliminated medium- and short-range nuclear missiles. The Strategic Arms Reduction Talks (START) treaties subsequently continued U.S. and Russian reductions, although thousands still remain.

Reagan's second requirement for ending nuclear testing was that the Soviets and the Americans should reduce their nuclear stockpiles. That effort started with the 1987 Intermediate-Range Nuclear Forces Treaty, which eliminated medium- and short-range nuclear missiles. The Strategic Arms Reduction Talks (START) treaties subsequently continued U.S. and Russian reductions, although thousands still remain.

In 1990, after Reagan had left office, both the Threshold Test Ban and the Peaceful Nuclear Explosions Treaty were ratified by the Senate after satisfactory review of the verification provisions. Reagan's first requirement on the road to a nuclear test ban was complete.

Reagan's second requirement for ending nuclear testing was that the Soviets and the Americans should reduce their nuclear stockpiles. That effort started with the 1987 Intermediate-Range Nuclear Forces Treaty, which eliminated medium- and short-range nuclear missiles. The Strategic Arms Reduction Talks (START) treaties subsequently continued U.S. and Russian reductions, although thousands still remain.

In 1996 the Comprehensive Nuclear Test Ban Treaty was crafted to ban all nuclear test



Automatic Thresholding

Although techniques based on binary image regions have been used for a very long time, they still play a major role in many practical image processing applications today because of their simplicity and efficiency. To obtain a binary image, the first and perhaps most critical step is to convert the initial grayscale (or color) image to a binary image, in most cases by performing some form of thresholding operation, as described in Chapter 4, Sec. 4.1.4.

Anyone who has ever tried to convert a scanned document image to a readable binary image has experienced how sensitively the result depends on the proper choice of the threshold value. This chapter deals with finding the best threshold automatically only from the information contained in the image, i.e., in an “unsupervised” fashion. This may be a single, “global” threshold that is applied to the whole image or different thresholds for different parts of the image. In the latter case we talk about “adaptive” thresholding, which is particularly useful when the image exhibits a varying background due to uneven lighting, exposure, or viewing conditions.

Automatic thresholding is a traditional and still very active area of research that had its peak in the 1980s and 1990s. Numerous techniques have been developed for this task, ranging from simple ad-hoc solutions to complex algorithms with firm theoretical foundations, as documented in several reviews and evaluation studies [86, 178, 204, 213, 231]. Binarization of images is also considered a “segmentation” technique and thus often categorized under this term. In the following, we describe some representative and popular techniques in greater detail, starting in Sec. 11.1 with global thresholding methods and continuing with adaptive methods in Sec. 11.2.

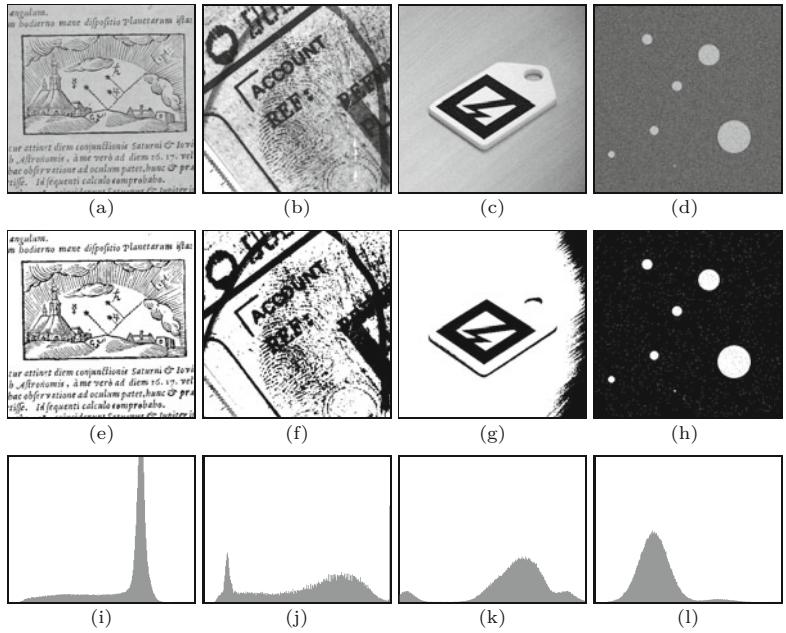
11.1 Global Histogram-Based Thresholding

Given a grayscale image I , the task is to find a single “optimal” threshold value for binarizing this image. Applying a particular threshold q is equivalent to classifying each pixel as being either part

11 AUTOMATIC THRESHOLDING

Fig. 11.1

Test images used for subsequent thresholding experiments. Detail from a manuscript by Johannes Kepler (a), document with fingerprint (b), ARTToolkit marker (c), synthetic two-level Gaussian mixture image (d). Results of thresholding with the fixed threshold value $q = 128$ (e–h). Histograms of the original images (i–l) with intensity values from 0 (left) to 255 (right).



of the *background* or the *foreground*. Thus the set of all image pixels is partitioned into two disjoint sets \mathcal{C}_0 and \mathcal{C}_1 , where \mathcal{C}_0 contains all elements with values in $[0, 1, \dots, q]$ and \mathcal{C}_1 collects the remaining elements with values in $[q+1, \dots, K-1]$, that is,

$$(u, v) \in \begin{cases} \mathcal{C}_0 & \text{if } I(u, v) \leq q \text{ (background),} \\ \mathcal{C}_1 & \text{if } I(u, v) > q \text{ (foreground).} \end{cases} \quad (11.1)$$

Of course, the meaning of *background* and *foreground* may differ from one application to another. For example, the aforementioned scheme is quite natural for astronomical or thermal images, where the relevant “foreground” pixels are bright and the background is dark. Conversely, in document analysis, for example, the objects of interest are usually the *dark* letters or artwork printed on a bright background. This should not be confusing and of course one can always *invert* the image to adapt to this scheme, so there is no loss of generality here.

Figure 11.1 shows several test images used in this chapter and the result of thresholding with a fixed threshold value. The synthetic image in Fig. 11.1(d) is the mixture of two Gaussian random distributions $\mathcal{N}_0, \mathcal{N}_1$ for the background and foreground, respectively, with $\mu_0 = 80$, $\mu_1 = 170$, $\sigma_0 = \sigma_1 = 20$. The corresponding histograms of the test images are shown in Fig. 11.1(i–l). Note that all histograms are normalized to constant area (not to maximum values, as usual), with intensity values ranging from 0 (left) to 255 (right).

The key question is how to find a suitable (or even “optimal”) threshold value for binarizing the image. As the name implies, histogram-based methods calculate the threshold primarily from the information contained in the image’s histogram, without inspecting the actual image pixels. Other methods process individual pixels for finding the threshold and there are also hybrid methods that rely both on the histogram and the local image content. Histogram-based

techniques are usually simple and efficient, because they operate on a small set of data (256 values in case of an 8-bit histogram); they can be grouped into two main categories: *shape-based* and *statistical* methods.

Shape-based methods analyze the structure of the histogram's distribution, for example by trying to locate peaks, valleys and other "shape" features. Usually the histogram is first smoothed to eliminate narrow peaks and gaps. While shape-based methods were quite popular early on, they are usually not as robust as their statistical counterparts or at least do not seem to offer any distinct advantages. A classic representative of this category is the "triangle" (or "chord") algorithm described in [261]. References to numerous other shape-based methods can be found in [213].

Statistical methods, as their name suggests, rely on statistical information derived from the image's histogram (which of course is a statistic itself), such as the mean, variance, or entropy. In the next section, we discuss a few elementary parameters that can be obtained from the histogram, followed by a description of concrete algorithms that use this information. Again there are a vast number of similar methods and we have selected four representative algorithms to be described in more detail: (a) iterative threshold selection by Ridler and Calvard [198], (b) Otsu's clustering method [177], (c) the minimum error method by Kittler and Illingworth [116], and (d) the maximum entropy thresholding method by Kapur, Sahoo, and Wong [133].

11.1.1 Image Statistics from the Histogram

As described in Chapter 3, Sec. 3.7, several statistical quantities, such as the arithmetic mean, variance and median, can be calculated directly from the histogram, without reverting to the original image data. If we *threshold* the image at level q ($0 \leq q < K$), the set of pixels is partitioned into the disjoint subsets $\mathcal{C}_0, \mathcal{C}_1$, corresponding to the background and the foreground. The number of pixels assigned to each subset is

$$n_0(q) = |\mathcal{C}_0| = \sum_{g=0}^q h(g) \quad \text{and} \quad n_1(q) = |\mathcal{C}_1| = \sum_{g=q+1}^{K-1} h(g), \quad (11.2)$$

respectively. Also, because all pixels are assigned to either the *background* set \mathcal{C}_0 or the *foreground* set \mathcal{C}_1 ,

$$n_0(q) + n_1(q) = |\mathcal{C}_0| + |\mathcal{C}_1| = |\mathcal{C}_0 \cup \mathcal{C}_1| = MN. \quad (11.3)$$

For any threshold q , the *mean* values of the associated partitions $\mathcal{C}_0, \mathcal{C}_1$ can be calculated from the image histogram as

$$\mu_0(q) = \frac{1}{n_0(q)} \cdot \sum_{g=0}^q g \cdot h(g), \quad (11.4)$$

$$\mu_1(q) = \frac{1}{n_1(q)} \cdot \sum_{g=q+1}^{K-1} g \cdot h(g) \quad (11.5)$$

11.1 GLOBAL HISTOGRAM-BASED THRESHOLDING

and these quantities relate to the image's overall mean μ_I (Eqn. (3.9)) by¹

$$\mu_I = \frac{1}{MN} \cdot [n_0(q) \cdot \mu_0(q) + n_1(q) \cdot \mu_1(q)] = \mu_0(K-1). \quad (11.6)$$

Analogously, the *variances* of the background and foreground partitions can be extracted from the histogram as²

$$\begin{aligned} \sigma_0^2(q) &= \frac{1}{n_0(q)} \cdot \sum_{g=0}^q (g - \mu_0(q))^2 \cdot h(g) \\ \sigma_1^2(q) &= \frac{1}{n_1(q)} \cdot \sum_{g=q+1}^{K-1} (g - \mu_1(q))^2 \cdot h(g). \end{aligned} \quad (11.7)$$

(Of course, as in Eqn. (3.12), this calculation can also be performed in a single iteration and without knowing $\mu_0(q), \mu_1(q)$ in advance.) The overall variance σ_I^2 for the whole image is identical to the variance of the background for $q = K-1$,

$$\sigma_I^2 = \frac{1}{MN} \cdot \sum_{g=0}^{K-1} (g - \mu_I)^2 \cdot h(g) = \sigma_0^2(K-1), \quad (11.8)$$

that is, for all pixels being assigned to the background partition. Note that, unlike the simple relation of the means given in Eqn. (11.6),

$$\sigma_I^2 \neq \frac{1}{MN} [n_0(q) \cdot \sigma_0^2(q) + n_1(q) \cdot \sigma_1^2(q)] \quad (11.9)$$

in general (see also Eqn. (11.20)).

We will use these basic relations in the discussion of histogram-based threshold selection algorithms in the following and add more specific ones as we go along.

11.1.2 Simple Threshold Selection

Clearly, the choice of the threshold value should not be fixed but somehow based on the content of the image. In the simplest case, we could use the *mean* of all image pixels,

$$q \leftarrow \text{mean}(I) = \mu_I, \quad (11.10)$$

as the threshold value q , or the *median*, (see Sec. 3.7.2),

$$q \leftarrow \text{median}(I) = m_I, \quad (11.11)$$

or, alternatively, the average of the *minimum* and the *maximum* (mid-range value), that is,

$$q \leftarrow \frac{\max(I) + \min(I)}{2}. \quad (11.12)$$

¹ Note that $\mu_0(q), \mu_1(q)$ are meant to be functions over q and thus $\mu_0(K-1)$ in Eqn. (11.6) denotes the mean of partition C_0 for the threshold $K-1$.

² $\sigma_0^2(q)$ and $\sigma_1^2(q)$ in Eqn. (11.7) are also functions over q .

```

1: QuantileThreshold( $\mathbf{h}, p$ )
   Input:  $\mathbf{h} : [0, K-1] \mapsto \mathbb{N}$ , a grayscale histogram.  $p$ , the proportion
          of expected background pixels ( $0 < p < 1$ ). Returns the optimal
          threshold value or  $-1$  if no threshold is found.

2:  $K \leftarrow \text{Size}(\mathbf{h})$                                  $\triangleright$  number of intensity levels
3:  $MN \leftarrow \sum_{i=0}^{K-1} \mathbf{h}(i)$                  $\triangleright$  number of image pixels
4:  $i \leftarrow 0$ 
5:  $c \leftarrow \mathbf{h}(0)$ 
6: while  $(i < K) \wedge (c < MN \cdot p)$  do     $\triangleright$  quantile calc. (Eq. 11.13)
7:    $i \leftarrow i + 1$ 
8:    $c \leftarrow c + \mathbf{h}(j)$ 
9: if  $c < MN$  then                       $\triangleright$  foreground is non-empty
10:   $q \leftarrow i$ 
11: else                                 $\triangleright$  foreground is empty, all pixels are background
12:   $q \leftarrow -1$ 
13: return  $q$ 

```

Like the image mean μ_I (see Eqn. (3.9)), all these quantities can be obtained directly from the histogram \mathbf{h} .

Thresholding at the median segments the image into approximately equal-sized background and foreground sets, that is, $|\mathcal{C}_0| \approx |\mathcal{C}_1|$, which assumes that the “interesting” (foreground) pixels cover about half of the image. This may be appropriate for certain images, but completely wrong for others. For example, a scanned text image will typically contain a lot more white than black pixels, so using the median threshold would probably be unsatisfactory in this case. If the approximate fraction p ($0 < p < 1$) of expected background pixels is known in advance, the threshold could be set to that *quantile* instead. In this case, q is simply chosen as

$$q \leftarrow \min \left\{ i \mid \sum_{j=0}^i \mathbf{h}(j) \geq M \cdot N \cdot p \right\}, \quad (11.13)$$

where N is the total number of pixels. We see that the *median* is only a special case of a quantile measure, with $p = 0.5$. This simple thresholding method is summarized in Alg. 11.1.

For the *mid-range* technique (Eqn. (11.12)), the limiting intensity values $\min(I)$ and $\max(I)$ can be found by searching for the smallest and largest non-zero entries, respectively, in the histogram \mathbf{h} . The mid-range threshold segments the image at 50 % (or any other percentile) of the contrast range. In this case, nothing can be said in general about the relative sizes of the resulting background and foreground partitions. Because a single extreme pixel value (outlier) may change the contrast range dramatically, this approach is not very robust. Here too it is advantageous to define the contrast range by specifying pixel *quantiles*, analogous to the calculation of the quantities a'_{low} and a'_{high} in the modified auto-contrast function (see Ch. 4, Sec. 4.4).

In the pathological (but nevertheless possible) case that all pixels in the image have the *same* intensity g , all the aforementioned meth-

11.1 GLOBAL HISTOGRAM-BASED THRESHOLDING

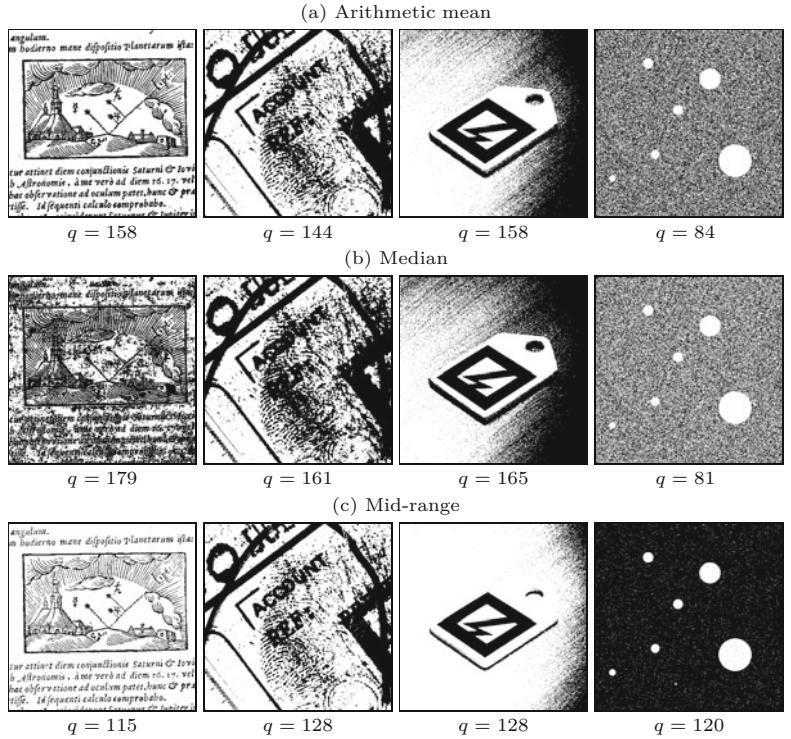
Alg. 11.1

Quantile thresholding. The optimal threshold value $q \in [0, K-2]$ is returned, or -1 if no valid threshold was found. Note the test in line 9 to check if the foreground is empty or not (the background is always non-empty by definition).

11 AUTOMATIC THRESHOLDING

Fig. 11.2

Results from various simple thresholding schemes. Mean (a–d), median (e–h), and mid-range (i–l) threshold, as specified in Eqns. (11.10)–(11.12).



ods will return the threshold $q = g$, which assigns all pixels to the background partition and leaves the foreground empty. Algorithms should try to detect this situation, because thresholding a uniform image obviously makes no sense. Results obtained with these simple thresholding techniques are shown in Fig. 11.2. Despite the obvious limitations, even a simple automatic threshold selection (such as the quantile technique in Alg. 11.1) will typically yield more reliable results than the use of a fixed threshold.

11.1.3 Iterative Threshold Selection (Isodata Algorithm)

This classic iterative algorithm for finding an optimal threshold is attributed to Ridler and Calvard [198] and was related to Isodata clustering by Velasco [242]. It is thus sometimes referred to as the “isodata” or “intermeans” algorithm. Like in many other global thresholding schemes it is assumed that the image’s histogram is a mixture of two separate distributions, one for the intensities of the background pixels and the other for the foreground pixels. In this case, the two distributions are assumed to be Gaussian with approximately identical spreads (variances).

The algorithm starts by making an initial guess for the threshold, for example, by taking the mean or the median of the whole image. This splits the set of pixels into a background and a foreground set, both of which should be non-empty. Next, the means of both sets are calculated and the threshold is repositioned to their average, that is, centered between the two means. The means are then re-calculated for the resulting background and foreground sets, and so on, until

```

1: IsodataThreshold( $h$ )
   Input:  $h : [0, K-1] \mapsto \mathbb{N}$ , a grayscale histogram.
   Returns the optimal threshold value or  $-1$  if no threshold is
   found.

2:  $K \leftarrow \text{Size}(h)$                                  $\triangleright$  number of intensity levels
3:  $q \leftarrow \text{Mean}(h, 0, K-1)$        $\triangleright$  set initial threshold to overall mean
4: repeat
5:    $n_0 \leftarrow \text{Count}(h, 0, q)$            $\triangleright$  background population
6:    $n_1 \leftarrow \text{Count}(h, q+1, K-1)$        $\triangleright$  foreground population
7:   if  $(n_0 = 0) \vee (n_1 = 0)$  then  $\triangleright$  backgrd. or foregrd. is empty
8:     return  $-1$ 
9:    $\mu_0 \leftarrow \text{Mean}(h, 0, q)$            $\triangleright$  background mean
10:   $\mu_1 \leftarrow \text{Mean}(h, q+1, K-1)$        $\triangleright$  foreground mean
11:   $q' \leftarrow q$                            $\triangleright$  keep previous threshold
12:   $q \leftarrow \left\lfloor \frac{\mu_0 + \mu_1}{2} \right\rfloor$      $\triangleright$  calculate the new threshold
13: until  $q = q'$                        $\triangleright$  terminate if no change
14: return  $q$ 

```

$$15: \text{Count}(h, a, b) := \sum_{g=a}^b h(g)$$

$$16: \text{Mean}(h, a, b) := \left[\sum_{g=a}^b g \cdot h(g) \right] / \left[\sum_{g=a}^b h(g) \right]$$

11.1 GLOBAL HISTOGRAM-BASED THRESHOLDING

Alg. 11.2

“Isodata” threshold selection based on the iterative method by Ridler and Calvard [198].

the threshold does not change any longer. In practice, it takes only a few iterations for the threshold to converge.

This procedure is summarized in Alg. 11.2. The initial threshold is set to the overall mean (line 3). For each threshold q , separate mean values μ_0, μ_1 are computed for the corresponding foreground and background partitions. The threshold is repeatedly set to the average of the two means until no more change occurs. The clause in line 7 tests if either the background or the foreground partition is empty, which will happen, for example, if the image contains only a single intensity value. In this case, no valid threshold exists and the procedure returns -1 . The functions $\text{Count}(h, a, b)$ and $\text{Mean}(h, a, b)$ in lines 15–16 return the number of pixels and the mean, respectively, of the image pixels with intensity values in the range $[a, b]$. Both can be computed directly from the histogram h without inspecting the image itself.

The performance of this algorithm can be easily improved by using tables $\mu_0(q), \mu_1(q)$ for the background and foreground means, respectively. The modified, table-based version of the iterative threshold selection procedure is shown in Alg. 11.3. It requires two passes over the histogram to initialize the tables μ_0, μ_1 and only a small, constant number of computations for each iteration in its main loop. Note that the image’s overall mean μ_I , used as the initial guess for the threshold q (Alg. 11.3, line 4), need not be calculated separately but can be obtained as $\mu_I = \mu_0(K-1)$, given that threshold $q = K-1$ assigns all image pixels to the background. The time complexity of this algorithm is thus $\mathcal{O}(K)$, that is, linear w.r.t. the size of the

11 AUTOMATIC THRESHOLDING

Alg. 11.3

Fast version of “isodata” threshold selection. Pre-calculated tables are used for the foreground and background means μ_0 and μ_1 , respectively.

```

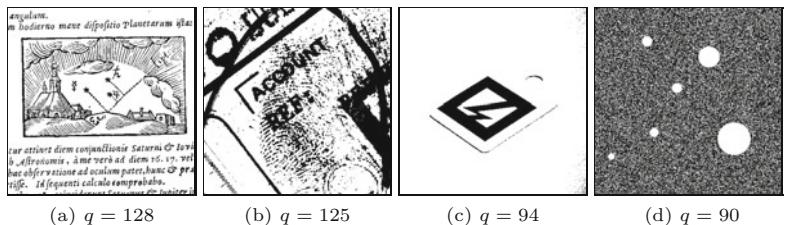
1: FastIsodataThreshold( $h$ )
2:   Input:  $h : [0, K-1] \mapsto \mathbb{N}$ , a grayscale histogram.
3:   Returns the optimal threshold value or  $-1$  if no threshold is
4:   found.
5:   repeat
6:     if  $(\mu_0(q) < 0) \vee (\mu_1(q) < 0)$  then
7:       return  $-1$                                  $\triangleright$  background or foreground is empty
8:      $q' \leftarrow q$                                  $\triangleright$  keep previous threshold
9:      $q \leftarrow \lfloor \frac{\mu_0(q) + \mu_1(q)}{2} \rfloor$        $\triangleright$  calculate the new threshold
10:    until  $q = q'$                              $\triangleright$  terminate if no change
11:    return  $q$ 

12: MakeMeanTables( $h$ )
13:    $K \leftarrow \text{Size}(h)$ 
14:   Create maps  $\mu_0, \mu_1 : [0, K-1] \mapsto \mathbb{R}$ 
15:    $n_0 \leftarrow 0, s_0 \leftarrow 0$ 
16:   for  $q \leftarrow 0, \dots, K-1$  do       $\triangleright$  tabulate background means  $\mu_0(q)$ 
17:      $n_0 \leftarrow n_0 + h(q)$ 
18:      $s_0 \leftarrow s_0 + q \cdot h(q)$ 
19:      $\mu_0(q) \leftarrow \begin{cases} s_0/n_0 & \text{if } n_0 > 0 \\ -1 & \text{otherwise} \end{cases}$ 
20:    $N \leftarrow n_0$ 
21:    $n_1 \leftarrow 0, s_1 \leftarrow 0$ 
22:    $\mu_1(K-1) \leftarrow 0$ 
23:   for  $q \leftarrow K-2, \dots, 0$  do       $\triangleright$  tabulate foreground means  $\mu_1(q)$ 
24:      $n_1 \leftarrow n_1 + h(q+1)$ 
25:      $s_1 \leftarrow s_1 + (q+1) \cdot h(q+1)$ 
26:      $\mu_1(q) \leftarrow \begin{cases} s_1/n_1 & \text{if } n_1 > 0 \\ -1 & \text{otherwise} \end{cases}$ 
27:   return  $\langle \mu_0, \mu_1, N \rangle$ 

```

Fig. 11.3

Thresholding with the isodata algorithm. Binarized images and the corresponding optimal threshold values (q).



histogram. Figure 11.3 shows the results of thresholding with the isodata algorithm applied to the test images in Fig. 11.1.

11.1.4 Otsu’s Method

The method proposed by Otsu [147, 177] also assumes that the original image contains pixels from two classes, whose intensity distributions are unknown. The goal is to find a threshold q such that the resulting background and foreground distributions are maximally separated, which means that they are (a) each as narrow as possi-

ble (have minimal variances) and (b) their centers (means) are most distant from each other.

For a given threshold q , the variances of the corresponding background and foreground partitions can be calculated straight from the image's histogram (see Eqn. (11.7)). The combined width of the two distributions is measured by the *within-class* variance

$$\sigma_w^2(q) = P_0(q) \cdot \sigma_0^2(q) + P_1(q) \cdot \sigma_1^2(q) \quad (11.14)$$

$$= \frac{1}{MN} \cdot [n_0(q) \cdot \sigma_0^2(q) + n_1(q) \cdot \sigma_1^2(q)], \quad (11.15)$$

where

$$P_0(q) = \sum_{i=0}^q p(i) = \frac{1}{MN} \cdot \sum_{i=0}^q h(i) = \frac{n_0(q)}{MN}, \quad (11.16)$$

$$P_1(q) = \sum_{i=q+1}^{K-1} p(i) = \frac{1}{MN} \cdot \sum_{i=q+1}^{K-1} h(i) = \frac{n_1(q)}{MN} \quad (11.17)$$

are the class probabilities for \mathcal{C}_0 , \mathcal{C}_1 , respectively. Thus the within-class variance in Eqn. (11.15) is simply the sum of the individual variances weighted by the corresponding class probabilities or “populations”. Analogously, the *between-class* variance,

$$\sigma_b^2(q) = P_0(q) \cdot (\mu_0(q) - \mu_I)^2 + P_1(q) \cdot (\mu_1(q) - \mu_I)^2 \quad (11.18)$$

$$= \frac{1}{MN} [n_0(q) \cdot (\mu_0(q) - \mu_I)^2 + n_1(q) \cdot (\mu_1(q) - \mu_I)^2] \quad (11.19)$$

measures the distances between the cluster means μ_0 , μ_1 and the overall mean μ_I . The total image variance σ_I^2 is the sum of the within-class variance and the between-class variance, that is,

$$\sigma_I^2 = \sigma_w^2(q) + \sigma_b^2(q), \quad (11.20)$$

for $q = 0, \dots, K-1$. Since σ_I^2 is constant for a given image, the threshold q can be found by either *minimizing* the within-variance σ_w^2 or *maximizing* the between-variance σ_b^2 . The natural choice is to maximize σ_b^2 , because it only relies on first-order statistics (i.e., the within-class means μ_0, μ_1). Since the overall mean μ_I can be expressed as the weighted sum of the partition means μ_0 and μ_1 (Eqn. (11.6)), we can simplify Eqn. (11.19) to

$$\sigma_b^2(q) = P_0(q) \cdot P_1(q) \cdot [\mu_0(q) - \mu_1(q)]^2 \quad (11.21)$$

$$= \frac{1}{(MN)^2} \cdot n_0(q) \cdot n_1(q) \cdot [\mu_0(q) - \mu_1(q)]^2. \quad (11.22)$$

The optimal threshold is finally found by *maximizing* the expression for the between-class variance in Eqn. (11.22) with respect to q , thereby *minimizing* the within-class variance in Eqn. (11.15).

Noting that $\sigma_b^2(q)$ only depends on the means (and *not* on the variances) of the two partitions for a given threshold q allows for a very efficient implementation, as outlined in Alg. 11.4. The algorithm assumes a grayscale image with a total of N pixels and K intensity

11.1 GLOBAL HISTOGRAM-BASED THRESHOLDING

11 AUTOMATIC THRESHOLDING

Alg. 11.4

Finding the optimal threshold using Otsu's method [177]. Initially (outside the **for**-loop), the threshold q is assumed to be -1 , which corresponds to the background class being empty ($n_0 = 0$) and all pixels are assigned to the foreground class ($n_1 = N$). The **for**-loop (lines 7–14) examines each possible threshold $q = 0, \dots, K-2$.

The factor $1/(MN)^2$ in line 11 is constant and thus not relevant for the optimization. The optimal threshold value is returned, or -1 if no valid threshold was found. The function `MakeMeanTables()` is defined in Alg. 11.3.

```

1: OtsuThreshold( $\mathbf{h}$ )
   Input:  $\mathbf{h} : [0, K-1] \mapsto \mathbb{N}$ , a grayscale histogram. Returns the
         optimal threshold value or  $-1$  if no threshold is found.
2:    $K \leftarrow \text{Size}(\mathbf{h})$                                  $\triangleright$  number of intensity levels
3:    $(\boldsymbol{\mu}_0, \boldsymbol{\mu}_1, MN) \leftarrow \text{MakeMeanTables}(\mathbf{h})$        $\triangleright$  see Alg. 11.3
4:    $\sigma_{b\max}^2 \leftarrow 0$ 
5:    $q_{\max} \leftarrow -1$ 
6:    $n_0 \leftarrow 0$ 
7:   for  $q \leftarrow 0, \dots, K-2$  do  $\triangleright$  examine all possible threshold values  $q$ 
8:      $n_0 \leftarrow n_0 + \mathbf{h}(q)$ 
9:      $n_1 \leftarrow MN - n_0$ 
10:    if  $(n_0 > 0) \wedge (n_1 > 0)$  then
11:       $\sigma_b^2 \leftarrow \frac{1}{(MN)^2} \cdot n_0 \cdot n_1 \cdot [\boldsymbol{\mu}_0(q) - \boldsymbol{\mu}_1(q)]^2$      $\triangleright$  see Eq. 11.22
12:      if  $\sigma_b^2 > \sigma_{b\max}^2$  then                                 $\triangleright$  maximize  $\sigma_b^2$ 
13:         $\sigma_{b\max}^2 \leftarrow \sigma_b^2$ 
14:         $q_{\max} \leftarrow q$ 
15:   return  $q_{\max}$ 
```

levels. As in Alg. 11.3, precalculated tables $\boldsymbol{\mu}_0(q), \boldsymbol{\mu}_1(q)$ are used for the background and foreground means for all possible threshold values $q = 0, \dots, K-1$.

Possible threshold values are $q = 0, \dots, K-2$ (with $q = K-1$, all pixels are assigned to the background). Initially (before entering the main **for**-loop in line 7) $q = -1$; at this point, the set of background pixels ($\leq q$) is empty and all pixels are classified as foreground ($n_0 = 0$ and $n_1 = N$). Each possible threshold value is examined inside the body of the **for**-loop.

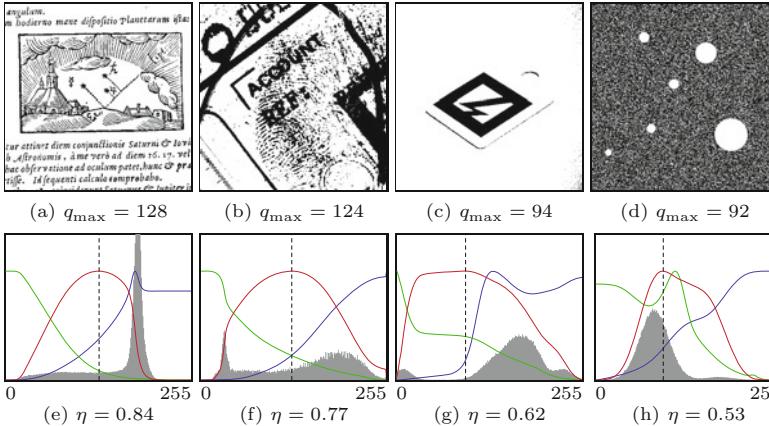
As long as any of the two classes is empty ($n_0(q) = 0$ or $n_1(q) = 0$),³ the resulting between-class variance $\sigma_b^2(q)$ is zero. The threshold that yields the maximum between-class variance ($\sigma_{b\max}^2$) is returned, or -1 if no valid threshold could be found. This occurs when all image pixels have the same intensity, that all pixels are in either the background or the foreground class.

Note that in line 11 of Alg. 11.4, the factor $\frac{1}{N^2}$ is constant (independent of q) and can thus be ignored in the optimization. However, care must be taken at this point because the computation of σ_b^2 may produce intermediate values that exceed the range of typical (32-bit) integer variables, even for medium-size images. Variables of type `long` should be used or the computation be performed with floating-point values.

The absolute “goodness” of the final thresholding by q_{\max} could be measured as the ratio

$$\eta = \frac{\sigma_b^2(q_{\max})}{\sigma_I^2} \in [0, 1] \quad (11.23)$$

³ This is the case if the image contains no pixels with values $I(u, v) \leq q$ or $I(u, v) > q$, that is, the histogram \mathbf{h} is empty either below or above the index q .



(see Eqn. (11.8)), which is invariant under linear changes of contrast and brightness [177]. Greater values of η indicate better thresholding.

Results of automatic threshold selection with Otsu’s method are shown in Fig. 11.4, where q_{\max} denotes the optimal threshold and η is the corresponding “goodness” estimate, as defined in Eqn. (11.23). The graph underneath each image shows the original histogram (gray) overlaid with the variance within the background σ_0^2 (green), the variance within the foreground σ_1^2 (blue), and the between-class variance σ_b^2 (red) for varying threshold values q . The dashed vertical line marks the position of the optimal threshold q_{\max} .

Due to the pre-calculation of the mean values, Otsu’s method requires only three passes over the histogram and is thus very fast ($\mathcal{O}(K)$), in contrast to opposite accounts in the literature. The method is frequently quoted and performs well in comparison to other approaches [213], despite its long history and its simplicity. In general, the results are very similar to the ones produced by the iterative threshold selection (“isodata”) algorithm described in Sec. 11.1.3.

11.1.5 Maximum Entropy Thresholding

Entropy is an important concept in information theory and particularly in data compression. It is a statistical measure that quantifies the average amount of information contained in the “messages” generated by a stochastic data source [99, 101]. For example, the MN pixels in an image I can be interpreted as a message of MN symbols, each taken independently from a finite alphabet of K (e.g., 256) different intensity values. Every pixel is assumed to be statistically independent. Knowing the probability of each intensity value g to occur, entropy measures how likely it is to observe a particular image, or, in other words, how much we should be surprised to see such an image. Before going into further details, we briefly review the notion of probabilities in the context of images and histograms (see also Ch. 4, Sec. 4.6.1).

For modeling the image generation as a random process, we first need to define an “alphabet”, that is, a set of symbols

11.1 GLOBAL HISTOGRAM-BASED THRESHOLDING

Fig. 11.4

Results of thresholding with Otsu’s method. Calculated threshold values q and resulting binary images (a–d). Graphs in (e–h) show the corresponding within-background variance σ_0^2 (green), the within-foreground variance σ_1^2 (blue), and the between-class variance σ_b^2 (red), for varying threshold values $q = 0, \dots, 255$. The optimal threshold q_{\max} (dashed vertical line) is positioned at the maximum of σ_b^2 . The value η denotes the “goodness” estimate for the thresholding, as defined in Eqn. (11.23).

$$Z = \{0, 1, \dots, K-1\}, \quad (11.24)$$

which in this case is simply the set of possible intensity values $g = 0, \dots, K-1$, together with the probability $p(g)$ that a particular intensity value g occurs. These probabilities are supposed to be known in advance, which is why they are called *a priori* (or *prior*) probabilities. The vector of probabilities,

$$(p(0), p(1), \dots, p(K-1)),$$

is a *probability distribution* or *probability density function* (pdf). In practice, the *a priori* probabilities are usually *unknown*, but they can be estimated by observing how often the intensity values actually occur in one or more images, assuming that these are representative instances of the images typically produced by that source. An estimate $\mathbf{p}(g)$ of the image's probability density function $p(g)$ is obtained by normalizing its histogram \mathbf{h} in the form

$$p(g) \approx \mathbf{p}(g) = \frac{\mathbf{h}(g)}{MN}, \quad (11.25)$$

for $0 \leq g < K$, such that $0 \leq \mathbf{p}(g) \leq 1$ and $\sum_{g=0}^{K-1} \mathbf{p}(g) = 1$. The associated *cumulative distribution function* (cdf) is

$$\mathbf{P}(g) = \sum_{i=0}^g \frac{\mathbf{h}(i)}{MN} = \sum_{i=0}^g \mathbf{p}(i), \quad (11.26)$$

where $\mathbf{P}(0) = \mathbf{p}(0)$ and $\mathbf{P}(K-1) = 1$. This is simply the normalized *cumulative histogram*.⁴

Entropy of images

Given an estimate of its intensity probability distribution $\mathbf{p}(g)$, the *entropy* of an image is defined as⁵

$$H(Z) = \sum_{g \in Z} \mathbf{p}(g) \cdot \log_b \left(\frac{1}{\mathbf{p}(g)} \right) = - \sum_{g \in Z} \mathbf{p}(g) \cdot \log_b (\mathbf{p}(g)), \quad (11.27)$$

where $g = I(u, v)$ and $\log_b(x)$ denotes the logarithm of x to the base b . If $b = 2$, the entropy (or “information content”) is measured in *bits*, but proportional results are obtained with any other logarithm (such as \ln or \log_{10}). Note that the value of $H()$ is always positive, because the probabilities $\mathbf{p}()$ are in $[0, 1]$ and thus the terms $\log_b [\mathbf{p}()]$ are negative or zero for any b .

Some other properties of the entropy are also quite intuitive. For example, if all probabilities $\mathbf{p}(g)$ are zero except for one intensity g' , then the entropy $H(I)$ is *zero*, indicating that there is no uncertainty (or “surprise”) in the messages produced by the corresponding data source. The (rather boring) images generated by this source will contain nothing but pixels of intensity g' , since all other intensities are

⁴ See also Chapter 3, Sec. 3.6.

⁵ Note the subtle difference in notation for the cumulative histogram \mathbf{H} and the entropy H .

impossible. Conversely, the entropy is a maximum if all K intensities have the same probability (uniform distribution),

$$p(g) = \frac{1}{K}, \quad \text{for } 0 \leq g < K, \quad (11.28)$$

and therefore (from Eqn. (11.27)) the entropy in this case is

$$H(Z) = - \sum_{i=0}^{K-1} \frac{1}{K} \cdot \log_b \left(\frac{1}{K} \right) = \frac{1}{K} \cdot \underbrace{\sum_{i=0}^{K-1} \log_b(K)}_{K \cdot \log_b(K)} \quad (11.29)$$

$$= \frac{1}{K} \cdot (K \cdot \log_b(K)) = \log_b(K). \quad (11.30)$$

This is the maximum possible entropy of a stochastic source with an alphabet Z of size K . Thus the entropy $H(Z)$ is always in the range $[0, \log(K)]$.

Using image entropy for threshold selection

The use of image entropy as a criterion for threshold selection has a long tradition and numerous methods have been proposed. In the following, we describe the early (but still popular) technique by Kapur et al. [100, 133] as a representative example.

Given a particular threshold q (with $0 \leq q < K-1$), the estimated probability distributions for the resulting partitions \mathcal{C}_0 and \mathcal{C}_1 are

$$\begin{aligned} \mathcal{C}_0 : & \left(\frac{p(0)}{P_0(q)} \frac{p(1)}{P_0(q)} \cdots \frac{p(q)}{P_0(q)} \quad 0 \quad 0 \quad \dots \quad 0 \quad \right), \\ \mathcal{C}_1 : & \left(\quad 0 \quad 0 \quad \dots \quad 0 \quad \frac{p(q+1)}{P_1(q)} \frac{p(q+2)}{P_1(q)} \cdots \frac{p(K-1)}{P_1(q)} \right), \end{aligned} \quad (11.31)$$

with the associated cumulated probabilities (see Eqn. (11.26))

$$P_0(q) = \sum_{i=0}^q p(i) = P(q) \quad \text{and} \quad P_1(q) = \sum_{i=q+1}^{K-1} p(i) = 1 - P(q). \quad (11.32)$$

Note that $P_0(q) + P_1(q) = 1$, since the background and foreground partitions are disjoint. The entropies *within* each partition are defined as

$$H_0(q) = - \sum_{i=0}^q \frac{p(i)}{P_0(q)} \cdot \log \left(\frac{p(i)}{P_0(q)} \right), \quad (11.33)$$

$$H_1(q) = - \sum_{i=q+1}^{K-1} \frac{p(i)}{P_1(q)} \cdot \log \left(\frac{p(i)}{P_1(q)} \right), \quad (11.34)$$

and the *overall* entropy for the threshold q is

$$H_{01}(q) = H_0(q) + H_1(q). \quad (11.35)$$

This expression is to be maximized over q , also called the “information between the classes” \mathcal{C}_0 and \mathcal{C}_1 . To allow for an efficient computation, the expression for $H_0(q)$ in Eqn. (11.33) can be rearranged to

$$H_0(q) = - \sum_{i=0}^q \frac{p(i)}{P_0(q)} \cdot [\log(p(i)) - \log(P_0(q))] \quad (11.36)$$

$$= - \frac{1}{P_0(q)} \cdot \sum_{i=0}^q p(i) \cdot [\log(p(i)) - \log(P_0(q))] \quad (11.37)$$

$$\begin{aligned} &= - \frac{1}{P_0(q)} \cdot \underbrace{\sum_{i=0}^q p(i) \cdot \log(p(i))}_{S_0(q)} + \frac{1}{P_0(q)} \cdot \underbrace{\sum_{i=0}^q p(i) \cdot \log(P_0(q))}_{= P_0(q)} \\ &= - \frac{1}{P_0(q)} \cdot S_0(q) + \log(P_0(q)). \end{aligned} \quad (11.38)$$

Similarly $H_1(q)$ in Eqn. (11.34) becomes

$$H_1(q) = - \sum_{i=q+1}^{K-1} \frac{p(i)}{P_1(q)} \cdot [\log(p(i)) - \log(P_1(q))] \quad (11.39)$$

$$= - \frac{1}{1-P_0(q)} \cdot S_1(q) + \log(1-P_0(q)). \quad (11.40)$$

Given the estimated probability distribution $p(i)$, the cumulative probability P_0 and the summation terms S_0, S_1 (see Eqns. (11.38)–(11.40)) can be calculated from the recurrence relations

$$\begin{aligned} P_0(q) &= \begin{cases} p(0) & \text{for } q = 0, \\ P_0(q-1) + p(q) & \text{for } 0 < q < K, \end{cases} \\ S_0(q) &= \begin{cases} p(0) \cdot \log(p(0)) & \text{for } q = 0, \\ S_0(q-1) + p(q) \cdot \log(p(q)) & \text{for } 0 < q < K, \end{cases} \\ S_1(q) &= \begin{cases} 0 & \text{for } q = K-1, \\ S_1(q+1) + p(q+1) \cdot \log(p(q+1)) & \text{for } 0 \leq q < K-1. \end{cases} \end{aligned} \quad (11.41)$$

The complete procedure is summarized in Alg. 11.5, where the values $S_0(q), S_1(q)$ are obtained from precalculated tables S_0, S_1 . The algorithm performs three passes over the histogram of length K (two for filling the tables S_0, S_1 and one in the main loop), so its time complexity is $\mathcal{O}(K)$, like the algorithms described before.

Results obtained with this technique are shown in Fig. 11.5. The technique described in this section is simple and efficient, because it again relies entirely on the image's histogram. More advanced entropy-based thresholding techniques exist that, among other improvements, take into account the spatial structure of the original image. An extensive review of entropy-based methods can be found in [46].

11.1.6 Minimum Error Thresholding

The goal of minimum error thresholding is to optimally fit a combination (mixture) of Gaussian distributions to the image's histogram. Before we proceed, we briefly look at some additional concepts from statistics. Note, however, that the following material is only intended

1: **MaximumEntropyThreshold**(h)

Input: $h : [0, K-1] \mapsto \mathbb{N}$, a grayscale histogram. Returns the optimal threshold value or -1 if no threshold is found.

```

2:  $K \leftarrow \text{Size}(h)$                                  $\triangleright$  number of intensity levels
3:  $p \leftarrow \text{Normalize}(h)$                            $\triangleright$  normalize histogram
4:  $(S_0, S_1) \leftarrow \text{MakeTables}(p, K)$            $\triangleright$  tables for  $S_0(q), S_1(q)$ 
5:  $P_0 \leftarrow 0$                                       $\triangleright P_0 \in [0, 1]$ 
6:  $q_{\max} \leftarrow -1$ 
7:  $H_{\max} \leftarrow -\infty$                              $\triangleright$  maximum joint entropy
8: for  $q \leftarrow 0, \dots, K-2$  do     $\triangleright$  check all possible threshold values  $q$ 
9:    $P_0 \leftarrow P_0 + p(q)$ 
10:   $P_1 \leftarrow 1 - P_0$                                  $\triangleright P_1 \in [0, 1]$ 
11:   $H_0 \leftarrow \begin{cases} -\frac{1}{P_0} \cdot S_0(q) + \log(P_0) & \text{if } P_0 > 0 \\ 0 & \text{otherwise} \end{cases}$   $\triangleright BG$  entropy
12:   $H_1 \leftarrow \begin{cases} -\frac{1}{P_1} \cdot S_1(q) + \log(P_1) & \text{if } P_1 > 0 \\ 0 & \text{otherwise} \end{cases}$   $\triangleright FG$  entropy
13:   $H_{01} = H_0 + H_1$                                  $\triangleright$  overall entropy for  $q$ 
14:  if  $H_{01} > H_{\max}$  then                          $\triangleright$  maximize  $H_{01}(q)$ 
15:     $H_{\max} \leftarrow H_{01}$ 
16:     $q_{\max} \leftarrow q$ 
17: return  $q_{\max}$ 
```

18: **MakeTables**(p, K)

```

19: Create maps  $S_0, S_1 : [0, K-1] \mapsto \mathbb{R}$ 
20:  $s_0 \leftarrow 0$ 
21: for  $i \leftarrow 0, \dots, K-1$  do                       $\triangleright$  initialize table  $S_0$ 
22:   if  $p(i) > 0$  then
23:      $s_0 \leftarrow s_0 + p(i) \cdot \log(p(i))$ 
24:    $S_0(i) \leftarrow s_0$ 
25:  $s_1 \leftarrow 0$ 
26: for  $i \leftarrow K-1, \dots, 0$  do                       $\triangleright$  initialize table  $S_1$ 
27:    $S_1(i) \leftarrow s_1$ 
28:   if  $p(i) > 0$  then
29:      $s_1 \leftarrow s_1 + p(i) \cdot \log(p(i))$ 
30: return  $(S_0, S_1)$ 
```

11.1 GLOBAL HISTOGRAM-BASED THRESHOLDING

Alg. 11.5

Maximum entropy threshold selection after Kapur et al. [133]. Initially (outside the **for**-loop), the threshold q is assumed to be -1 , which corresponds to the background class being empty ($n_0 = 0$) and all pixels assigned to the foreground class ($n_1 = N$). The **for**-loop (lines 8–16) examines each possible threshold $q = 0, \dots, K-2$. The optimal threshold value $(0, \dots, K-2)$ is returned, or -1 if no valid threshold was found.

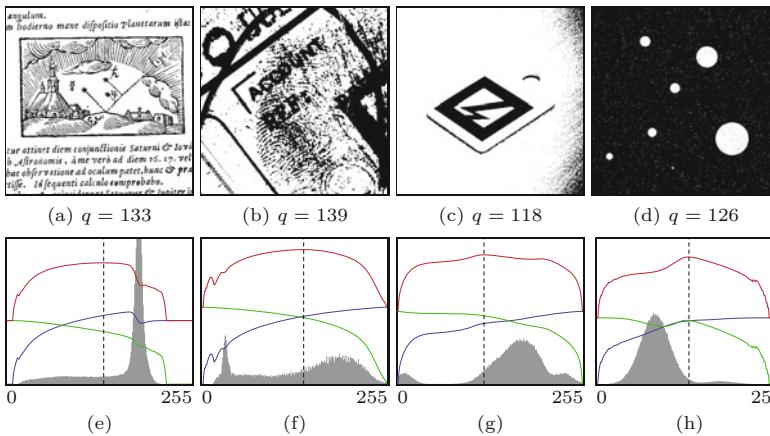


Fig. 11.5
Thresholding with the Maximum-entropy method. Calculated threshold values q and resulting binary images (a–d). Graphs in (e–h) show the background entropy $H_0(q)$ (green), foreground entropy $H_1(q)$ (blue) and overall entropy $H_{01}(q) = H_0(q) + H_1(q)$ (red), for varying threshold values q . The optimal threshold q_{\max} is found at the maximum of H_{01} (dashed vertical line).

as a superficial outline to explain the elementary concepts. For a solid grounding of these and related topics readers are referred to the excellent texts available on statistical pattern recognition, such as [24, 64].

Bayesian decision making

The assumption is again that the image pixels originate from one of two classes, \mathcal{C}_0 and \mathcal{C}_1 , or background and foreground, respectively. Both classes generate random intensity values following unknown statistical distributions. Typically, these are modeled as Gaussian distributions with unknown parameters μ and σ^2 , as will be described. The task is to decide for each pixel value x to which of the two classes it most likely belongs. Bayesian reasoning is a classic technique for making such decisions in a probabilistic context.

The *probability*, that a certain intensity value x originates from a background pixel is denoted

$$p(x | \mathcal{C}_0).$$

This is called a “conditional probability”.⁶ It tells us how likely it is to observe the gray value x when a pixel is a member of the background class \mathcal{C}_0 . Analogously, $p(x | \mathcal{C}_1)$ is the conditional probability of observing the value x when a pixel is known to be of the foreground class \mathcal{C}_1 .

For the moment, let us assume that the conditional probability functions $p(x | \mathcal{C}_0)$ and $p(x | \mathcal{C}_1)$ are known. Our problem is reversed though, namely to decide which class a pixel most likely belongs to, given that its intensity is x . This means that we are actually interested in the conditional probabilities

$$p(\mathcal{C}_0 | x) \quad \text{and} \quad p(\mathcal{C}_1 | x), \quad (11.42)$$

also called *a posteriori* (or *posterior*) probabilities. If we knew these, we could simply select the class with the higher probability in the form

$$\mathcal{C} = \begin{cases} \mathcal{C}_0 & \text{if } p(\mathcal{C}_0 | x) > p(\mathcal{C}_1 | x), \\ \mathcal{C}_1 & \text{otherwise.} \end{cases} \quad (11.43)$$

Bayes’ theorem provides a method for estimating these *posterior* probabilities, that is,

$$p(\mathcal{C}_j | x) = \frac{p(x | \mathcal{C}_j) \cdot p(\mathcal{C}_j)}{p(x)}, \quad (11.44)$$

where $p(\mathcal{C}_j)$ is the *prior* probability of class \mathcal{C}_j . While, in theory, the prior probabilities are also unknown, they can be easily estimated from the image histogram (see also Sec. 11.1.5). Finally, $p(x)$ in Eqn. (11.44) is the overall probability of observing the intensity value x ,

⁶ In general, $p(A | B)$ denotes the (conditional) probability of observing the event A in a given situation B . It is usually read as “the probability of A , given B ”.

which is typically estimated from its relative frequency in one or more images.⁷

Note that for a particular intensity x , the corresponding evidence $p(x)$ only *scales* the posterior probabilities and is thus not relevant for the classification itself. Consequently, we can reformulate the binary decision rule in Eqn. (11.43) to

$$\mathcal{C} = \begin{cases} \mathcal{C}_0 & \text{if } p(x | \mathcal{C}_0) \cdot p(\mathcal{C}_0) > p(x | \mathcal{C}_1) \cdot p(\mathcal{C}_1), \\ \mathcal{C}_1 & \text{otherwise.} \end{cases} \quad (11.45)$$

This is called Bayes' decision rule. It minimizes the probability of making a classification error if the involved probabilities are known and is also called the “minimum error” criterion.

Gaussian probability distributions

If the probability distributions $p(x | \mathcal{C}_j)$ are modeled as *Gaussian*⁸ distributions $\mathcal{N}(x | \mu_j, \sigma_j^2)$, where μ_j, σ_j^2 denote the *mean* and *variance* of class \mathcal{C}_j , we can rewrite the scaled posterior probabilities in Eqn. (11.45) as

$$p(x | \mathcal{C}_j) \cdot p(\mathcal{C}_j) = \frac{1}{\sqrt{2\pi\sigma_j^2}} \cdot \exp\left(-\frac{(x - \mu_j)^2}{2\sigma_j^2}\right) \cdot p(\mathcal{C}_j). \quad (11.46)$$

As long as the ordering between the resulting class scores remains unchanged, these quantities can be scaled or transformed arbitrarily. In particular, it is common to use the *logarithm* of the above expression to avoid repeated multiplications of small numbers. For example, applying the natural logarithm⁹ to both sides of Eqn. (11.46) yields

$$\ln(p(x | \mathcal{C}_j) \cdot p(\mathcal{C}_j)) = \ln(p(x | \mathcal{C}_j)) + \ln(p(\mathcal{C}_j)) \quad (11.47)$$

$$= \ln\left(\frac{1}{\sqrt{2\pi\sigma_j^2}}\right) + \ln\left(\exp\left(-\frac{(x - \mu_j)^2}{2\sigma_j^2}\right)\right) + \ln(p(\mathcal{C}_j)) \quad (11.48)$$

$$= -\frac{1}{2} \cdot \ln(2\pi) - \frac{1}{2} \cdot \ln(\sigma_j^2) - \frac{(x - \mu_j)^2}{2\sigma_j^2} + \ln(p(\mathcal{C}_j)) \quad (11.49)$$

$$= -\frac{1}{2} \cdot \left[\ln(2\pi) + \frac{(x - \mu_j)^2}{\sigma_j^2} + \ln(\sigma_j^2) - 2 \cdot \ln(p(\mathcal{C}_j)) \right]. \quad (11.50)$$

Since $\ln(2\pi)$ in Eqn. (11.50) is constant, it can be ignored for the classification decision, as well as the factor $\frac{1}{2}$ at the front. Thus, to find the class \mathcal{C}_j that maximizes $p(x | \mathcal{C}_j) \cdot p(\mathcal{C}_j)$ for a given intensity value x , it is sufficient to *maximize* the quantity

$$-\left[\frac{(x - \mu_j)^2}{\sigma_j^2} + 2 \cdot [\ln(\sigma_j^2) - \ln(p(\mathcal{C}_j))]\right] \quad (11.51)$$

or, alternatively, to *minimize*

⁷ $p(x)$ is also called the “evidence” for the event x .

⁸ See also Sec. D.4 in the Appendix.

⁹ Any logarithm could be used but the natural logarithm complements the exponential function of the Gaussian.

$$\varepsilon_j(x) = \frac{(x - \mu_j)^2}{\sigma_j^2} + 2 \cdot [\ln(\sigma_j) - \ln(p(\mathcal{C}_j))]. \quad (11.52)$$

The quantity $\varepsilon_j(x)$ can be viewed as a *measure of the potential error* involved in classifying the observed value x as being of class \mathcal{C}_j . To obtain the decision associated with the minimum risk, we can modify the binary decision rule in Eqn. (11.45) to

$$\mathcal{C} = \begin{cases} \mathcal{C}_0 & \text{if } \varepsilon_0(x) \leq \varepsilon_1(x), \\ \mathcal{C}_1 & \text{otherwise.} \end{cases} \quad (11.53)$$

Remember that this rule tells us how to correctly classify the observed intensity value x as being either of the background class \mathcal{C}_0 or the foreground class \mathcal{C}_1 , assuming that the underlying distributions are really Gaussian and their parameters are well estimated.

Goodness of classification

If we apply a threshold q , all pixel values $g \leq q$ are implicitly classified as \mathcal{C}_0 (background) and all $g > q$ as \mathcal{C}_1 (foreground). The goodness of this classification by q over all N image pixels $I(u, v)$ can be measured with the criterion function

$$e(q) = \frac{1}{MN} \cdot \sum_{u,v} \begin{cases} \varepsilon_0(I(u, v)) & \text{for } I(u, v) \leq q \\ \varepsilon_1(I(u, v)) & \text{for } I(u, v) > q \end{cases} \quad (11.54)$$

$$= \frac{1}{MN} \cdot \sum_{g=0}^q h(g) \cdot \varepsilon_0(g) + \frac{1}{MN} \cdot \sum_{g=q+1}^{K-1} h(g) \cdot \varepsilon_1(g) \quad (11.55)$$

$$= \sum_{g=0}^q p(g) \cdot \varepsilon_0(g) + \sum_{g=q+1}^{K-1} p(g) \cdot \varepsilon_1(g), \quad (11.56)$$

with the normalized frequencies $p(g) = h(g)/N$ and the function $\varepsilon_j(g)$ as defined in Eqn. (11.52). By substituting $\varepsilon_j(g)$ from Eqn. (11.52) and some mathematical gymnastics, $e(q)$ can be written as

$$\begin{aligned} e(q) &= 1 + P_0(q) \cdot \ln(\sigma_0^2(q)) + P_1(q) \cdot \ln(\sigma_1^2(q)) \\ &\quad - 2 \cdot P_0(q) \cdot \ln(P_0(q)) - 2 \cdot P_1(q) \cdot \ln(P_1(q)). \end{aligned} \quad (11.57)$$

The remaining task is to find the threshold q that *minimizes* $e(q)$ (where the constant 1 in Eqn. (11.57) can be omitted, of course). For each possible threshold q , we only need to estimate (from the image's histogram, as in Eqn. (11.31)) the “prior” probabilities $P_0(q)$, $P_1(q)$ and the corresponding within-class variances $\sigma_0(q)$, $\sigma_1(q)$. The *prior* probabilities for the background and foreground classes are estimated as

$$P_0(q) \approx \sum_{g=0}^q p(g) = \frac{1}{MN} \cdot \sum_{g=0}^q h(g) = \frac{n_0(q)}{MN}, \quad (11.58)$$

$$P_1(q) \approx \sum_{g=q+1}^{K-1} p(g) = \frac{1}{MN} \cdot \sum_{g=q+1}^{K-1} h(g) = \frac{n_1(q)}{MN}, \quad (11.59)$$

where $n_0(q) = \sum_{i=0}^q h(i)$, $n_1(q) = \sum_{i=q+1}^{K-1} h(i)$, and $MN = n_0(q) + n_1(q)$ is the total number of image pixels. Estimates for background and foreground variances ($\sigma_0^2(q)$ and $\sigma_1^2(q)$, respectively) defined in Eqn. (11.7), can be calculated efficiently by expressing them in the form

$$\begin{aligned}\sigma_0^2(q) &\approx \frac{1}{n_0(q)} \cdot \left[\underbrace{\sum_{g=0}^q h(g) \cdot g^2}_{B_0(q)} - \frac{1}{n_0(q)} \cdot \left(\underbrace{\sum_{g=0}^q h(g) \cdot g}_{A_0(q)} \right)^2 \right] \\ &= \frac{1}{n_0(q)} \cdot \left[B_0(q) - \frac{1}{n_0(q)} \cdot A_0^2(q) \right],\end{aligned}\quad (11.60)$$

$$\begin{aligned}\sigma_1^2(q) &\approx \frac{1}{n_1(q)} \cdot \left[\underbrace{\sum_{g=q+1}^{K-1} h(g) \cdot g^2}_{B_1(q)} - \frac{1}{n_1(q)} \cdot \left(\underbrace{\sum_{g=q+1}^{K-1} h(g) \cdot g}_{A_1(q)} \right)^2 \right] \\ &= \frac{1}{n_1(q)} \cdot \left[B_1(q) - \frac{1}{n_1(q)} \cdot A_1^2(q) \right],\end{aligned}\quad (11.61)$$

with the quantities

$$\begin{aligned}A_0(q) &= \sum_{g=0}^q h(g) \cdot g, & B_0(q) &= \sum_{g=0}^q h(g) \cdot g^2, \\ A_1(q) &= \sum_{g=q+1}^{K-1} h(g) \cdot g, & B_1(q) &= \sum_{g=q+1}^{K-1} h(g) \cdot g^2.\end{aligned}\quad (11.62)$$

Furthermore, the values $\sigma_0^2(q)$, $\sigma_1^2(q)$ can be tabulated for every possible q in only two passes over the histogram, using the recurrence relations

$$A_0(q) = \begin{cases} 0 & \text{for } q = 0, \\ A_0(q-1) + h(q) \cdot q & \text{for } 1 \leq q \leq K-1, \end{cases}\quad (11.63)$$

$$B_0(q) = \begin{cases} 0 & \text{for } q = 0, \\ B_0(q-1) + h(q) \cdot q^2 & \text{for } 1 \leq q \leq K-1, \end{cases}\quad (11.64)$$

$$A_1(q) = \begin{cases} 0 & \text{for } q = K-1, \\ A_1(q+1) + h(q+1) \cdot (q+1) & \text{for } 0 \leq q \leq K-2, \end{cases}\quad (11.65)$$

$$B_1(q) = \begin{cases} 0 & \text{for } q = K-1, \\ B_1(q+1) + h(q+1) \cdot (q+1)^2 & \text{for } 0 \leq q \leq K-2. \end{cases}\quad (11.66)$$

The complete minimum-error threshold calculation is summarized in Alg. 11.6. First, the tables S_0, S_1 are set up and initialized with the values of $\sigma_0^2(q), \sigma_1^2(q)$, respectively, for $0 \leq q < K$, following the recursive scheme in Eqns. (11.63–11.66). Subsequently, the error value $e(q)$ is calculated for every possible threshold value q to find the global minimum. Again $e(q)$ can only be calculated for those values of q , for which both resulting partitions are non-empty (i.e., with $n_0(q), n_1(q) > 0$). Note that, in lines 27 and 37 of Alg. 11.6, a small constant ($\frac{1}{12}$) is added to the variance to avoid zero values when the corresponding class population is homogeneous (i.e., only

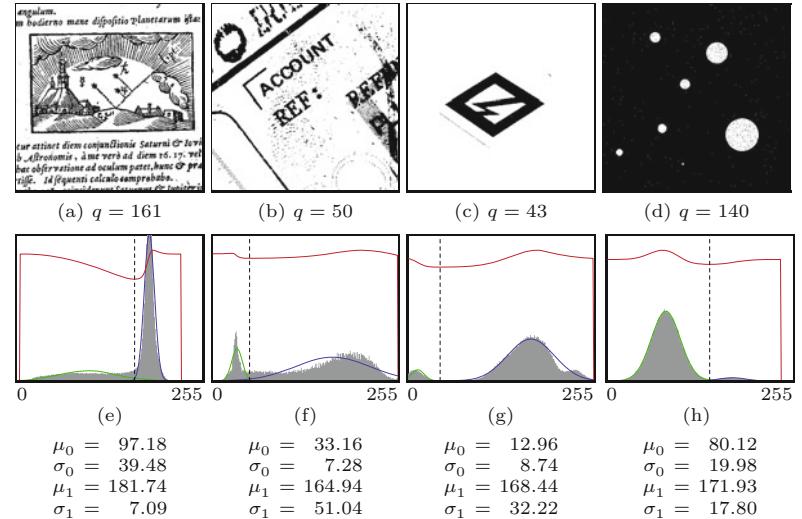
11.1 GLOBAL HISTOGRAM-BASED THRESHOLDING

contains a single intensity value).¹⁰ This ensures that the algorithm works properly on images with only two distinct gray values. The algorithm computes the optimal threshold by performing three passes over the histogram (two for initializing the tables and one for finding the minimum); it thus has the same time complexity of $\mathcal{O}(K)$ as the algorithms described before.

[Figure 11.6](#) shows the results of minimum-error thresholding on our set of test images. It also shows the fitted pair of Gaussian distributions for the background and the foreground pixels, respectively, for the optimal threshold as well as the graphs of the error function $e(q)$, which is minimized over all threshold values q . Obviously the error function is quite flat in certain cases, indicating that similar scores are obtained for a wide range of threshold values and the optimal threshold is not very distinct. We can also see that the estimate is quite accurate in case of the synthetic test image in [Fig. 11.6\(d\)](#), which is actually generated as a mixture of two Gaussians (with parameters $\mu_0 = 80$, $\mu_1 = 170$ and $\sigma_0 = \sigma_1 = 20$). Note that the histograms in [Fig. 11.6](#) have been properly normalized (to constant area) to illustrate the curves of the Gaussians, that is, properly scaled by their prior probabilities (P_0, P_1), while the original histograms are scaled with respect to their maximum values.

Fig. 11.6

Results from minimum-error thresholding. Calculated threshold values q and resulting binary images (a–d). The green and blue graphs in (e–h) show the fitted Gaussian background and foreground distributions $\mathcal{N}_0 = (\mu_0, \sigma_0)$ and $\mathcal{N}_1 = (\mu_1, \sigma_1)$, respectively. The red graph corresponds to the error quantity $e(q)$ for varying threshold values $q = 0, \dots, 255$ (see Eqn. (11.57)). The optimal threshold q_{\min} is located at the minimum of $e(q)$ (dashed vertical line). The estimated parameters of the background/foreground Gaussians are listed at the bottom.



A minor theoretical problem with the minimum error technique is that the parameters of the Gaussian distributions are always estimated from *truncated* samples. This means that, for any threshold q , only the intensity values smaller than q are used to estimate the parameters of the background distribution, and only the intensities greater than q contribute to the foreground parameters. In practice, this problem is of minor importance, since the distributions are typically not strictly Gaussian either.

¹⁰ This is explained by the fact that each histogram bin $h(i)$ represents intensities in the continuous range $[i \pm 0.5]$ and the variance of uniformly distributed values in the unit interval is $\frac{1}{12}$.

1: **MinimumErrorThreshold(h)**

Input: $h : [0, K-1] \mapsto \mathbb{N}$, a grayscale histogram. Returns the optimal threshold value or -1 if no threshold is found.

```

2:    $K \leftarrow \text{Size}(h)$ 
3:    $(S_0, S_1, N) \leftarrow \text{MakeSigmaTables}(h, K)$ 
4:    $n_0 \leftarrow 0$ 
5:    $q_{\min} \leftarrow -1$ 
6:    $e_{\min} \leftarrow \infty$ 
7:   for  $q \leftarrow 0, \dots, K-2$  do       $\triangleright$  evaluate all possible thresholds  $q$ 
8:      $n_0 \leftarrow n_0 + h(q)$            $\triangleright$  background population
9:      $n_1 \leftarrow N - n_0$            $\triangleright$  foreground population
10:    if  $(n_0 > 0) \wedge (n_1 > 0)$  then
11:       $P_0 \leftarrow n_0/N$            $\triangleright$  prior probability of  $C_0$ 
12:       $P_1 \leftarrow n_1/N$            $\triangleright$  prior probability of  $C_1$ 
13:       $e \leftarrow P_0 \cdot \ln(S_0(q)) + P_1 \cdot \ln(S_1(q))$ 
           $- 2 \cdot (P_0 \cdot \ln(P_0) + P_1 \cdot \ln(P_1))$        $\triangleright$  Eq. 11.57
14:    if  $e < e_{\min}$  then           $\triangleright$  minimize error for  $q$ 
15:       $e_{\min} \leftarrow e$ 
16:       $q_{\min} \leftarrow q$ 
17:   return  $q_{\min}$ 
```

18: **MakeSigmaTables(h, K)**

Create maps $S_0, S_1 : [0, K-1] \mapsto \mathbb{R}$

```

19:    $n_0 \leftarrow 0$ 
20:    $A_0 \leftarrow 0$ 
21:    $B_0 \leftarrow 0$ 
22:   for  $q \leftarrow 0, \dots, K-1$  do           $\triangleright$  tabulate  $\sigma_0^2(q)$ 
23:      $n_0 \leftarrow n_0 + h(q)$ 
24:      $A_0 \leftarrow A_0 + h(q) \cdot q$            $\triangleright$  Eq. 11.63
25:      $B_0 \leftarrow B_0 + h(q) \cdot q^2$            $\triangleright$  Eq. 11.64
26:      $S_0(q) \leftarrow \begin{cases} \frac{1}{12} + (B_0 - A_0^2/n_0)/n_0 & \text{for } n_0 > 0 \\ 0 & \text{otherwise} \end{cases}$        $\triangleright$  Eq. 11.60
27:    $N \leftarrow n_0$ 
28:    $n_1 \leftarrow 0$ 
29:    $A_1 \leftarrow 0$ 
30:    $B_1 \leftarrow 0$ 
31:    $S_1(K-1) \leftarrow 0$ 
32:   for  $q \leftarrow K-2, \dots, 0$  do           $\triangleright$  tabulate  $\sigma_1^2(q)$ 
33:      $n_1 \leftarrow n_1 + h(q+1)$ 
34:      $A_1 \leftarrow A_1 + h(q+1) \cdot (q+1)$            $\triangleright$  Eq. 11.65
35:      $B_1 \leftarrow B_1 + h(q+1) \cdot (q+1)^2$            $\triangleright$  Eq. 11.66
36:      $S_1(q) \leftarrow \begin{cases} \frac{1}{12} + (B_1 - A_1^2/n_1)/n_1 & \text{for } n_1 > 0 \\ 0 & \text{otherwise} \end{cases}$        $\triangleright$  Eq. 11.61
37:   return  $(S_0, S_1, N)$ 
```

11.2 LOCAL ADAPTIVE THRESHOLDING

Alg. 11.6

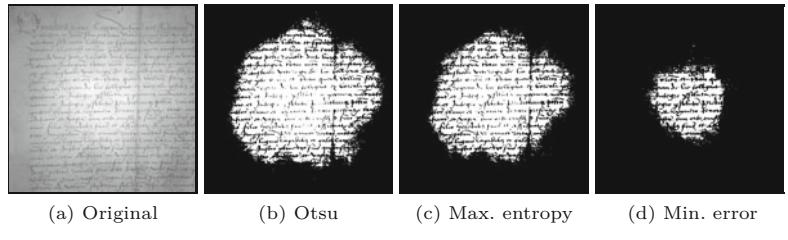
Minimum error threshold selection based on a Gaussian mixture model (after [116]). Tables S_0, S_1 are initialized with values $\sigma_0^2(q)$ and $\sigma_1^2(q)$, respectively (see Eqns. (11.60)–(11.61)), for all possible threshold values $q = 0, \dots, K-1$. N is the number of image pixels. Initially (outside the **for**-loop), the threshold q is assumed to be -1 , which corresponds to the background class being empty ($n_0 = 0$) and all pixels assigned to the foreground class ($n_1 = N$). The **for**-loop (lines 8–16) examines each possible threshold $q = 0, \dots, K-2$. The optimal threshold is returned, or -1 if no valid threshold was found.

11.2 Local Adaptive Thresholding

In many situations, a fixed threshold is not appropriate to classify the pixels in the entire image, for example, when confronted with stained backgrounds or uneven lighting or exposure. Figure 11.7 shows a typical, unevenly exposed document image and the results obtained with some global thresholding methods described in the previous sections.

Fig. 11.7

Global thresholding methods fail under uneven lighting or exposure. Original image (a), results from global thresholding with various methods described above (b-d).



Instead of using a single threshold value for the whole image, adaptive thresholding specifies a *varying* threshold value $Q(u, v)$ for each image position that is used to classify the corresponding pixel $I(u, v)$ in the same way as described in Eqn. (11.1) for a global threshold. The following approaches differ only with regard to how the threshold “surface” Q is derived from the input image.

11.2.1 Bernsen’s Method

The method proposed by Bernsen [23] specifies a dynamic threshold for each image position (u, v) , based on the minimum and maximum intensity found in a local neighborhood $R(u, v)$. If

$$I_{\min}(u, v) = \min_{\substack{(i, j) \in \\ R(u, v)}} I(i, j), \quad (11.67)$$

$$I_{\max}(u, v) = \max_{\substack{(i, j) \in \\ R(u, v)}} I(i, j) \quad (11.68)$$

are the minimum and maximum intensity values within a fixed-size neighborhood region R centered at position (u, v) , the space-varying threshold is simply calculated as the *mid-range* value

$$Q(u, v) = \frac{I_{\min}(u, v) + I_{\max}(u, v)}{2}. \quad (11.69)$$

This is done as long as the local contrast $c(u, v) = I_{\max}(u, v) - I_{\min}(u, v)$ is above some predefined limit c_{\min} . If $c(u, v) < c_{\min}$, the pixels in the corresponding image region are assumed to belong to a single class and are (by default) assigned to the background.

The whole process is summarized in Alg. 11.7. Note that the meaning of “background” in terms of intensity levels depends on the application. For example, in astronomy, the image background is usually darker than the objects of interest. In typical OCR applications, however, the background (paper) is brighter than the foreground objects (print). The main function provides a control parameter bg to select the proper default threshold \bar{q} , which is set to K in case of a dark background ($bg = \text{dark}$) and to 0 for a bright background ($bg = \text{bright}$). The support region R may be square or circular, typically with a radius $r = 15$. The choice of the minimum contrast limit c_{\min} depends on the type of imagery and the noise level ($c_{\min} = 15$ is a suitable value to start with).

Figure 11.8 shows the results of Bernsen’s method on the uneven test image used in Fig. 11.7 for different settings of the region’s radius r . Due to the nonlinear min- and max-operation, the resulting

```

1: BernsenThreshold( $I, r, c_{\min}, bg$ )
   Input:  $I$ , intensity image of size  $M \times N$ ;  $r$ , radius of support
   region;  $c_{\min}$ , minimum contrast;  $bg$ , background type (dark or
   bright). Returns a map with an individual threshold value for
   each image position.
2:  $(M, N) \leftarrow \text{Size}(I)$ 
3: Create map  $Q : M \times N \mapsto \mathbb{R}$ 
4:  $\bar{q} \leftarrow \begin{cases} K & \text{if } bg = \text{dark} \\ 0 & \text{if } bg = \text{bright} \end{cases}$ 
5: for all image coordinates  $(u, v) \in M \times N$  do
6:    $R \leftarrow \text{MakeCircularRegion}(u, v, r)$ 
7:    $I_{\min} \leftarrow \min_{(i,j) \in R} I(i, j)$ 
8:    $I_{\max} \leftarrow \max_{(i,j) \in R} I(i, j)$ 
9:    $c \leftarrow I_{\max} - I_{\min}$ 
10:   $Q(u, v) \leftarrow \begin{cases} (I_{\min} + I_{\max})/2 & \text{if } c \geq c_{\min} \\ \bar{q} & \text{otherwise} \end{cases}$ 
11: return  $Q$ 

```

12: **MakeCircularRegion**(u, v, r)
 Returns the set of pixel coordinates within the circle of radius r ,
 centered at (u, v)

13: **return** $\{(i, j) \in \mathbb{Z}^2 \mid (u - i)^2 + (v - j)^2 \leq r^2\}$

11.2 LOCAL ADAPTIVE THRESHOLDING

Alg. 11.7

Adaptive thresholding using local contrast (after Bernsen [23]). The argument to bg should be set to `dark` if the image background is darker than the structures of interest, and to `bright` if the background is brighter than the objects.

threshold surface is not smooth. The minimum contrast is set to $c_{\min} = 15$, which is too low to avoid thresholding low-contrast noise visible along the left image margin. By increasing the minimum contrast c_{\min} , more neighborhoods are considered “flat” and thus ignored, that is, classified as background. This is demonstrated in Fig. 11.9. While larger values of c_{\min} effectively eliminate low-contrast noise, relevant structures are also lost, which illustrates the difficulty of finding a suitable global value for c_{\min} . Additional examples, using the test images previously used for global thresholding, are shown in Fig. 11.10.

What Alg. 11.7 describes formally can be implemented quite efficiently, noting that the calculation of local minima and maxima over a sliding window (lines 6–8) corresponds to a simple nonlinear filter operation (see Ch. 5, Sec. 5.4). To perform these calculations, we can use a *minimum* and *maximum* filter with radius r , as provided by virtually every image processing environment. For example, the Java implementation of the Bernsen thresholder in Prog. 11.1 uses ImageJ’s built-in `RankFilters` class for this purpose. The complete implementation can be found on the book’s website (see Sec. 11.3 for additional details on the corresponding API).

11.2.2 Niblack’s Method

In this approach, originally presented in [172, Sec. 5.1], the threshold $Q(u, v)$ is varied across the image as a function of the local intensity average $\mu_R(u, v)$ and standard deviation¹¹ $\sigma_R(u, v)$ in the form

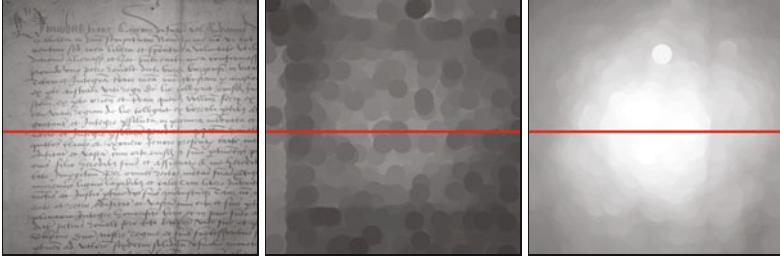
¹¹ The standard deviation σ is the square root of the variance σ^2 .

11 AUTOMATIC THRESHOLDING

Prog. 11.1

Bernsen's thresholder (ImageJ plugin implementation of Alg. 11.7). Note the use of ImageJ's `RankFilters` class (lines 30–32) for calculating the local minimum (I_{\min}) and maximum (I_{\max}) maps inside the `getThreshold()` method. The resulting threshold surface $Q(u, v)$ is returned as an 8-bit image of type `ByteProcessor`.

```
1 package imagingbook.pub.threshold.adaptive;
2 import ij.plugin.filter.RankFilters;
3 import ij.process.ByteProcessor;
4 import imagingbook.pub.threshold.BackgroundMode;
5
6 public class BernsenThresholder extends AdaptiveThresholder {
7
8     public static class Parameters {
9         public int radius = 15;
10        public int cmin = 15;
11        public BackgroundMode bgMode = BackgroundMode.DARK;
12    }
13
14    private final Parameters params;
15
16    public BernsenThresholder() {
17        this.params = new Parameters();
18    }
19
20    public BernsenThresholder(Parameters params) {
21        this.params = params;
22    }
23
24    public ByteProcessor getThreshold(ByteProcessor I) {
25        int M = I.getWidth();
26        int N = I.getHeight();
27        ByteProcessor Imin = (ByteProcessor) I.duplicate();
28        ByteProcessor Imax = (ByteProcessor) I.duplicate();
29
30        RankFilters rf = new RankFilters();
31        rf.rank(Imin,params.radius,RankFilters.MIN); //  $I_{\min}(u, v)$ 
32        rf.rank(Imax,params.radius,RankFilters.MAX); //  $I_{\max}(u, v)$ 
33
34        int q = (params.bgMode == BackgroundMode.DARK) ?
35                    256 : 0;
36        ByteProcessor Q = new ByteProcessor(M, N); //  $Q(u, v)$ 
37
38        for (int v = 0; v < N; v++) {
39            for (int u = 0; u < M; u++) {
40                int gMin = Imin.get(u, v);
41                int gMax = Imax.get(u, v);
42                int c = gMax - gMin;
43                if (c >= params.cmin)
44                    Q.set(u, v, (gMin + gMax) / 2);
45                else
46                    Q.set(u, v, q);
47            }
48        }
49        return Q;
50    }
51 }
```

(a) $I(u, v)$ (b) $I_{\min}(u, v)$ (c) $I_{\max}(u, v)$ (d) $r = 7$ (e) $r = 15$ (f) $r = 30$ (g) $r = 7$ (h) $r = 15$ (i) $r = 30$

11.2 LOCAL ADAPTIVE THRESHOLDING

Fig. 11.8

Adaptive thresholding using Bernsen's method. Original image (a), local minimum (b), and maximum (c). The center row shows the binarized images for different settings of r (d–f). The corresponding curves in (g–i) show the original intensity (gray), local minimum (green), maximum (red), and the actual threshold (blue) along the horizontal line marked in (a–c). The region radius r is 15 pixels, the minimum contrast c_{\min} is 15 intensity units.

(a) $c_{\min} = 15$ (b) $c_{\min} = 30$ (c) $c_{\min} = 60$ **Fig. 11.9**

Adaptive thresholding using Bernsen's method with different settings of c_{\min} . Binarized images (top row) and threshold surface $Q(u, v)$ (bottom row). Black areas in the threshold functions indicate that the local contrast is below c_{\min} ; the corresponding pixels are classified as background (white in this case).

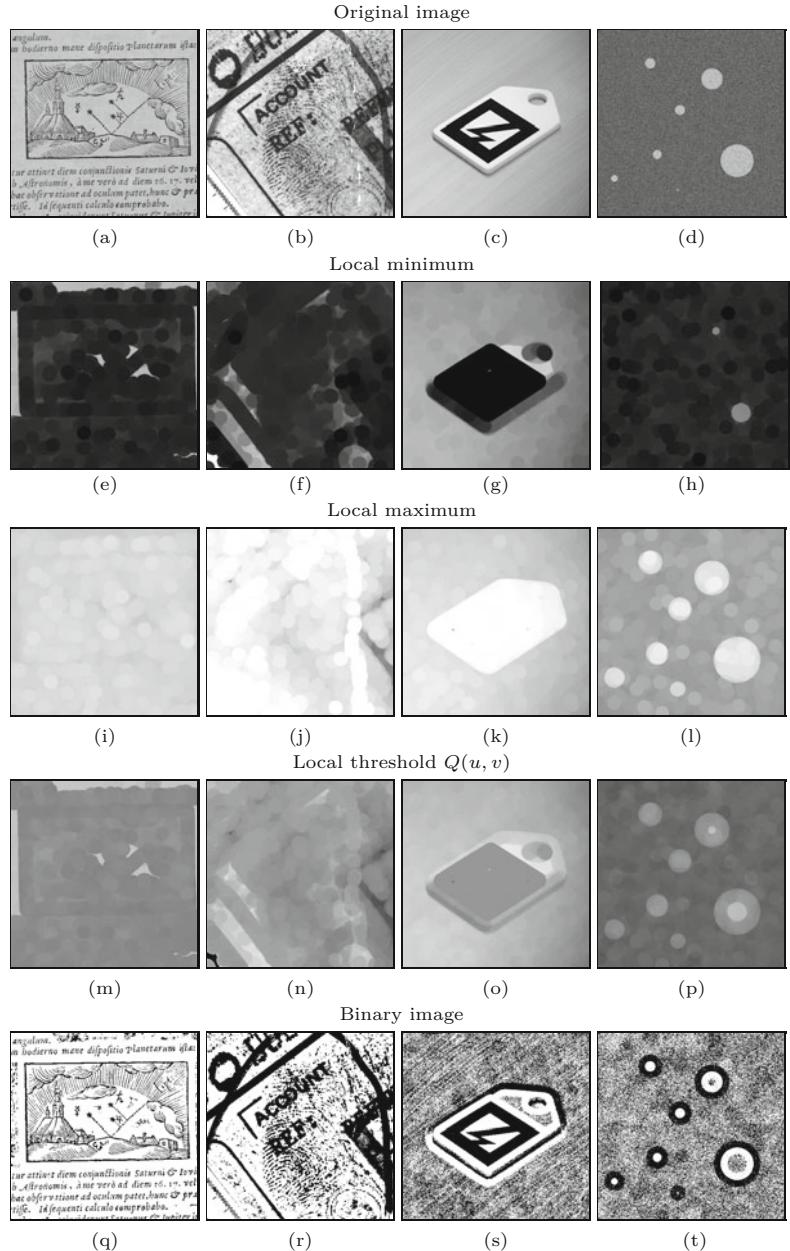
$$Q(u, v) := \mu_R(u, v) + \kappa \cdot \sigma_R(u, v). \quad (11.70)$$

Thus the local threshold $Q(u, v)$ is determined by adding a constant portion ($\kappa \geq 0$) of the local standard deviation σ_R to the local mean μ_R . μ_R and σ_R are calculated over a square support region R centered at (u, v) . The size (radius) of the averaging region R should be as large as possible, at least larger than the size of the structures to be detected, but small enough to capture the variations (unevenness)

11 AUTOMATIC THRESHOLDING

Fig. 11.10

Additional examples for Bernsen's method. Original images (a–d), local minimum I_{\min} (e–h), maximum I_{\max} (i–l), and threshold map Q (m–p); results after thresholding the images (q–t). Settings are $r = 15$, $c_{\min} = 15$. A bright background is assumed for all images ($bq = \text{bright}$), except for image (d).



of the background. A size of 31×31 pixels (or radius $r = 15$) is suggested in [172] and $\kappa = 0.18$, though the latter does not seem to be critical.

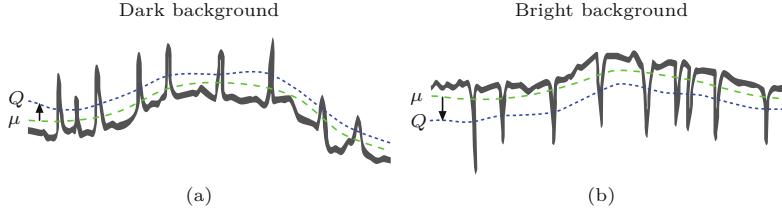
One problem is that, for small values of σ_R (as obtained in “flat” image regions of approximately constant intensity), the threshold will be close to the local average, which makes the segmentation quite sensitive to low-amplitude noise (“ghosting”). A simple improvement is to secure a minimum distance from the mean by adding a constant offset d , that is, replacing Eqn. (11.70) by

$$Q(u, v) := \mu_R(u, v) + \kappa \cdot \sigma_R(u, v) + d, \quad (11.71)$$

with $d \geq 0$, in the range $2, \dots, 20$ for typical 8-bit images.

The original formulation (Eqn. (11.70)) is aimed at situations where the foreground structures are *brighter* than the background ([Fig. 11.11\(a\)](#)) but does not work if the images are set up the other way round ([Fig. 11.11\(b\)](#)). In the case that the structures of interest are *darker* than the background (as, e.g., in typical OCR applications), one could either work with inverted images or modify the calculation of the threshold to

$$Q(u, v) := \begin{cases} \mu_R(u, v) + (\kappa \cdot \sigma_R(u, v) + d) & \text{for dark BG,} \\ \mu_R(u, v) - (\kappa \cdot \sigma_R(u, v) + d) & \text{for bright BG.} \end{cases} \quad (11.72)$$



The modified procedure is detailed in Alg. 11.8. The example in [Fig. 11.12](#) shows results obtained with this method on an image with a bright background containing dark structures, for $\kappa = 0.3$ and varying settings of d . Note that setting $d = 0$ ([Fig. 11.12\(d, g\)](#)) corresponds to Niblack's original method. For these examples, a circular window of radius $r = 15$ was used to compute the local mean $\mu_R(u, v)$ and variance $\sigma_R(u, v)$. Additional examples are shown in [Fig. 11.13](#). Note that the selected radius r is obviously too small for the structures in the images in [Fig. 11.13\(c, d\)](#), which are thus not segmented cleanly. Better results can be expected with a larger radius.

With the intent to improve upon Niblack's method, particularly for thresholding deteriorated text images, Sauvola and Pietikäinen [207] proposed setting the threshold to

$$Q(u, v) := \begin{cases} \mu_R(u, v) \cdot [1 - \kappa \cdot (\frac{\sigma_R(u, v)}{\sigma_{\max}} - 1)] & \text{for dark BG,} \\ \mu_R(u, v) \cdot [1 + \kappa \cdot (\frac{\sigma_R(u, v)}{\sigma_{\max}} - 1)] & \text{for bright BG,} \end{cases} \quad (11.73)$$

with $\kappa = 0.5$ and $\sigma_{\max} = 128$ (the “dynamic range of the standard deviation” for 8-bit images) as suggested parameter values. In this approach, the offset between the threshold and the local average not only depends on the local variation σ_R (as in Eqn. (11.70)), but also on the magnitude of the local *mean* μ_R ! Thus, changes in absolute brightness lead to modified relative threshold values, even when the image contrast remains constant. Though this technique is frequently referenced in the literature, it appears questionable if this behavior is generally desirable.

Calculating local mean and variance

Algorithm 11.8 shows the principle operation of Niblack's method and also illustrates how to efficiently calculate the local average and

11.2 LOCAL ADAPTIVE THRESHOLDING

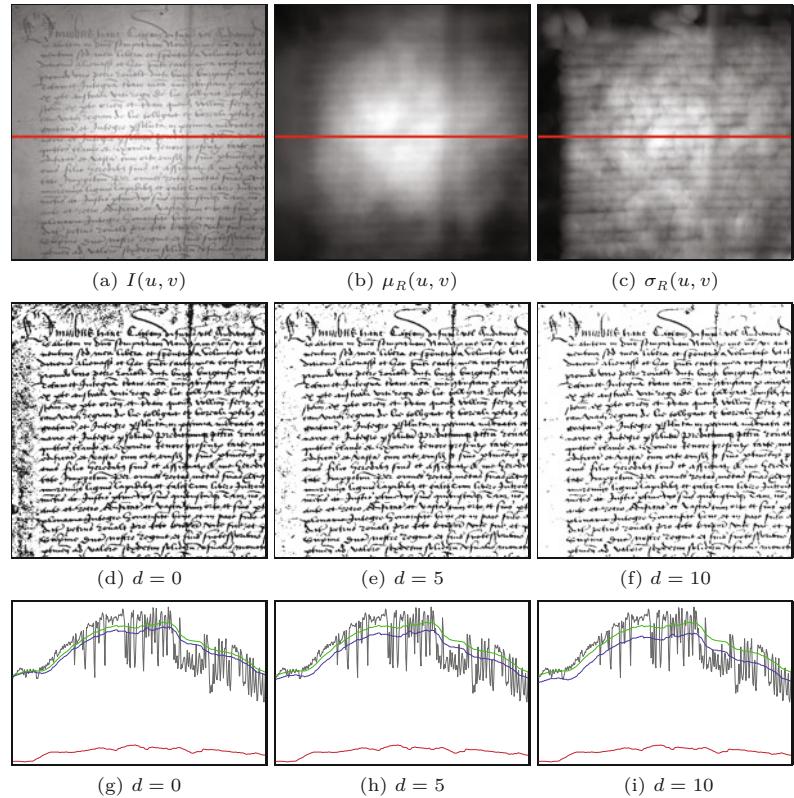
Fig. 11.11

Adaptive thresholding based on average local intensity. The illustration shows a line profile as typically found in document imaging. The space-variant threshold Q (dotted blue line) is chosen as the local average μ_R (dashed green line) offset by a multiple of the local intensity variation σ_R . The offset is chosen to be *positive* for images with a dark background and bright structures (a) and *negative* if the background is brighter than the contained structures (b).

11 AUTOMATIC THRESHOLDING

Fig. 11.12

Adaptive thresholding using Niblack's method (with $r = 15$, $\kappa = 0.3$). Original image (a), local mean μ_R (b), and standard deviation σ_R (c). The result for $d = 0$ in (d) corresponds to Niblack's original formulation. Increasing the value of d reduces the amount of clutter in regions with low variance (e, f). The curves in (g–i) show the local intensity (gray), mean (green), variance (red), and the actual threshold (blue) along the horizontal line marked in (a–c).



variance. Given the image I and the averaging region R , we can use the shortcut suggested in Eqn. (3.12) to obtain these quantities as

$$\mu_R = \frac{1}{n} \cdot A \quad \text{and} \quad \sigma_R^2 = \frac{1}{n} \cdot (B - \frac{1}{n} \cdot A^2), \quad (11.74)$$

with

$$A = \sum_{(i,j) \in R} I(i,j), \quad B = \sum_{(i,j) \in R} I^2(i,j), \quad n = |R|. \quad (11.75)$$

Procedure `GetLocalMeanAndVariance()` in Alg. 11.8 shows this calculation in full detail.

When computing the local average and variance, attention must be paid to the situation at the image borders, as illustrated in Fig. 11.14. Two approaches are frequently used. In the first approach (following the common practice for implementing filter operations), all outside pixel values are replaced by the closest inside pixel, which is always a border pixel. Thus the border pixel values are effectively replicated outside the image boundaries and thus these pixels have a strong influence on the local results. The second approach is to perform the calculation of the average and variance on only those image pixels that are actually covered by the support region. In this case, the number of pixels (N) is reduced at the image borders to about 1/4 of the full region size.

Although the calculation of the local mean and variance outlined by function `GetLocalMeanAndVariance()` in Alg. 11.8 is definitely more

```

1: NiblackThreshold( $I, r, \kappa, d, bg$ )
   Input:  $I$ , intensity image of size  $M \times N$ ;  $r$ , radius of support region;  $\kappa$ , variance control parameter;  $d$ , minimum offset;  $bg \in \{\text{dark}, \text{bright}\}$ , background type. Returns a map with an individual threshold value for each image position.
2:  $(M, N) \leftarrow \text{Size}(I)$ 
3: Create map  $Q : M \times N \mapsto \mathbb{R}$ 
4: for all image coordinates  $(u, v) \in M \times N$  do
   Define a support region of radius  $r$ , centered at  $(u, v)$ :
5:  $(\mu, \sigma^2) \leftarrow \text{GetLocalMeanAndVariance}(I, u, v, r)$ 
6:  $\sigma \leftarrow \sqrt{\sigma^2}$  ▷ local std. deviation  $\sigma_R$ 
7:  $Q(u, v) \leftarrow \begin{cases} \mu + (\kappa \cdot \sigma + d) & \text{if } bg = \text{dark} \\ \mu - (\kappa \cdot \sigma + d) & \text{if } bg = \text{bright} \end{cases}$  ▷ Eq. 11.72
8: return  $Q$ 
9: GetLocalMeanAndVariance( $I, u, v, r$ )
   Returns the local mean and variance of the image pixels  $I(i, j)$  within the disk-shaped region with radius  $r$  around position  $(u, v)$ .
10:  $R \leftarrow \text{MakeCircularRegion}(u, v, r)$  ▷ see Alg. 11.7
11:  $n \leftarrow 0$ 
12:  $A \leftarrow 0$ 
13:  $B \leftarrow 0$ 
14: for all  $(i, j) \in R$  do
15:    $n \leftarrow n + 1$ 
16:    $A \leftarrow A + I(i, j)$ 
17:    $B \leftarrow B + I^2(i, j)$ 
18:    $\mu \leftarrow \frac{1}{n} \cdot A$ 
19:    $\sigma^2 \leftarrow \frac{1}{n} \cdot (B - \frac{1}{n} \cdot A^2)$ 
20: return  $(\mu, \sigma^2)$ 

```

11.2 LOCAL ADAPTIVE THRESHOLDING

Alg. 11.8

Adaptive thresholding using local mean and variance (modified version of Niblack's method [172]). The argument to bg should be `dark` if the image background is darker than the structures of interest, `bright` if the background is brighter than the objects. The function `MakeCircularRegion()` is defined in Alg. 11.7.

efficient than a brute-force approach, additional optimizations are possible. Most image processing environments have suitable routines already built in. With ImageJ, for example, we can again use the `RankFilters` class (as with the *min-* and *max-filters* in the *Bernsen* approach, see Sec. 11.2.1). Instead of performing the computation for each pixel individually, the following ImageJ code segment uses pre-defined filters to compute two separate images `Imean` (μ_R) and `Ivar` (σ_R^2) containing the local mean and variance values, respectively, with a disk-shaped support region of radius 15:

```

ByteProcessor I; // original image  $I(u, v)$ 
int radius = 15;

FloatProcessor Imean = I.convertToFloatProcessor();
FloatProcessor Ivar = Imean.duplicate();

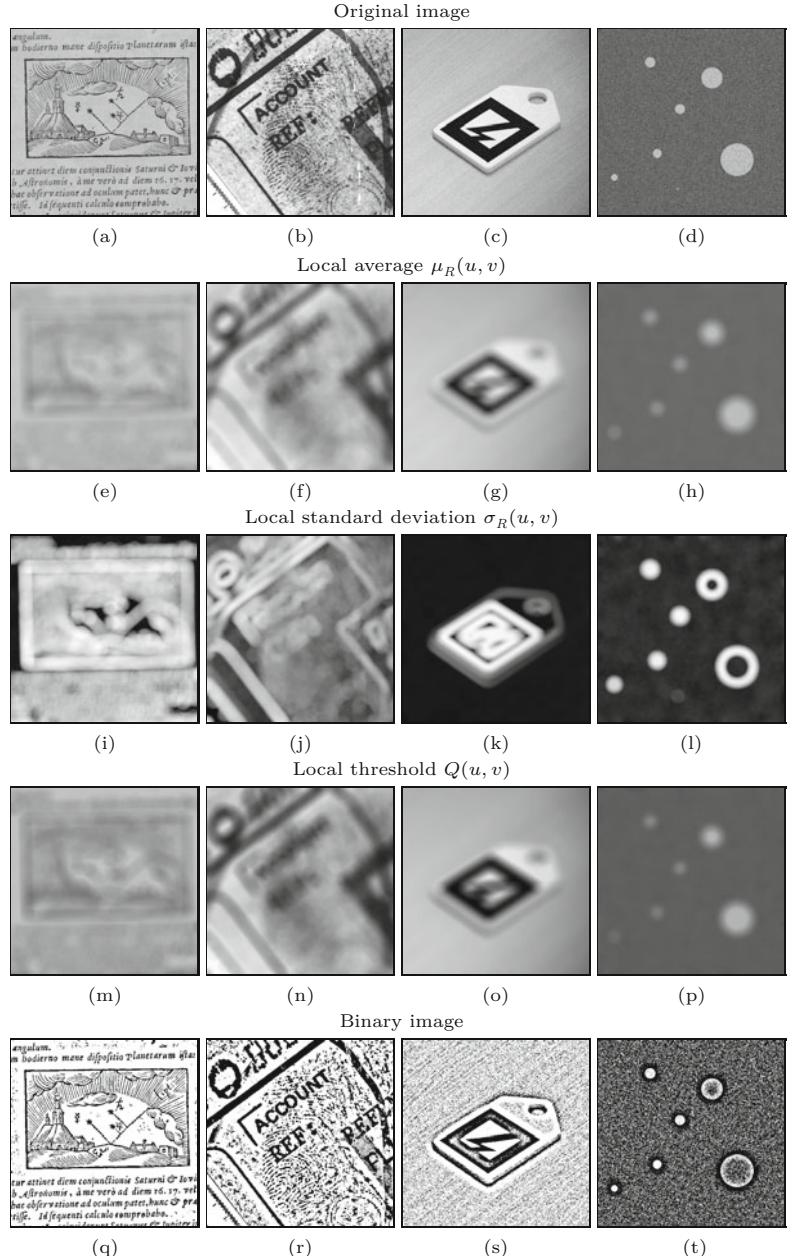
RankFilters rf = new RankFilters();
rf.rank(Imean, radius, RankFilters.MEAN);           //  $\mu_R(u, v)$ 
rf.rank(Ivar, radius, RankFilters.VARIANCE);        //  $\sigma_R^2(u, v)$ 
...

```

11 AUTOMATIC THRESHOLDING

Fig. 11.13

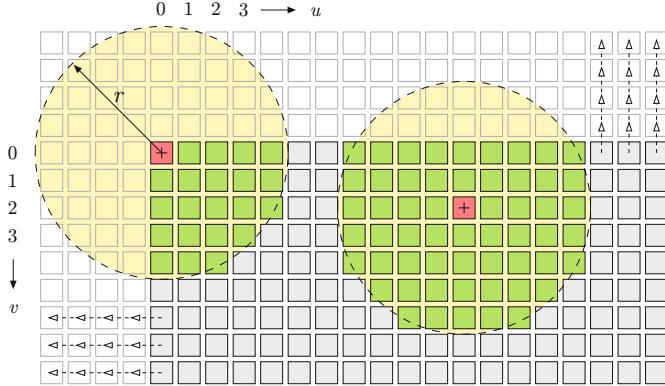
Additional examples for thresholding with Niblack's method using a disk-shaped support region of radius $r = 15$. Original images (a–d), local mean μ_R (e–h), std. deviation σ_R (i–l), and threshold Q (m–p); results after thresholding the images (q–t). The background is assumed to be brighter than the structures of interest, except for image (d), which has a dark background. Settings are $\kappa = 0.3$, $d = 5$.



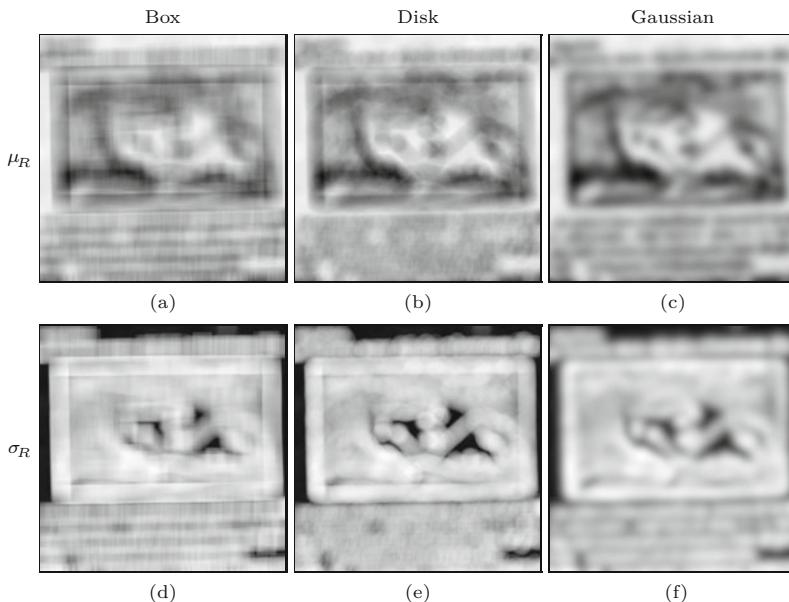
See Sec. 11.3 and the online code for additional implementation details. Note that the filter methods implemented in `RankFilters` perform replication of border pixels as the border handling strategy, as discussed earlier.

Local average and variance with Gaussian kernels

The purpose of taking the local average is to smooth the image to obtain an estimate of the varying background intensity. In case of a square or circular region, this is equivalent to convolving the image with a box- or disk-shaped kernel, respectively. Kernels of this



type, however, are not well suited for image smoothing, because they create strong ringing and truncating effects, as demonstrated in Fig. 11.15. Moreover, convolution with a box-shaped (rectangular) kernel is a non-isotropic operation, that is, the results are orientation-dependent. From this perspective alone it seems appropriate to consider other smoothing kernels, Gaussian kernels in particular.



Using a Gaussian kernel H^G for smoothing is equivalent to calculating a weighted average of the corresponding image pixels, with the weights being the coefficients of the kernel. Thus calculating this weighted local average can be expressed by

$$\mu_G(u, v) = \frac{1}{\sum H^G} \cdot (I * H^G)(u, v), \quad (11.76)$$

where $\sum H^G$ is the sum of the coefficients in the kernel H^G and $*$ denotes the linear convolution operator.¹² Analogously, there is also

11.2 LOCAL ADAPTIVE THRESHOLDING

Fig. 11.14

Calculating local statistics at image boundaries. The illustration shows a disk-shaped support region with radius r , placed at the image border. Pixel values outside the image can be replaced (“filled-in”) by the closest border pixel, as is common in many filter operations. Alternatively, the calculation of the local statistics can be confined to include only those pixels inside the image that are actually covered by the support region. At any border pixel, the number of covered elements (N) is still more than $\approx 1/4$ of the full region size. In this particular case, the circular region covers a maximum of $N = 69$ pixels when fully embedded and $N = 22$ when positioned at an image corner.

Fig. 11.15

Local average (a–c) and variance (d–f) obtained with different smoothing kernels. 31×31 box filter (a, d), disk filter with radius $r = 15$ (b, e), Gaussian kernel with $\sigma = 0.6 \cdot 15 = 9.0$ (c, f). Both the box and disk filter show strong truncation effects (ringing), the box filter is also highly non-isotropic. All images are contrast-enhanced for better visibility.

¹² See Chapter 5, Sec. 5.3.1.

a weighted variance σ_G^2 which can be calculated jointly with the local average μ_G (as in Eqn. (11.74)) in the form

$$\mu_G(u, v) = \frac{1}{\sum H^G} \cdot A_G(u, v), \quad (11.77)$$

$$\sigma_G^2(u, v) = \frac{1}{\sum H^G} \cdot (B_G(u, v) - \frac{1}{\sum H^G} \cdot A_G^2(u, v)), \quad (11.78)$$

with $A_G = I * H^G$ and $B_G = I^2 * H^G$.

Thus all we need is two filter operations, one applied to the original image ($I * H^G$) and another applied to the squared image ($I^2 * H^G$), using the same 2D Gaussian kernel H^G (or any other suitable smoothing kernel). If the kernel H^G is *normalized* (i.e., $\sum H^G = 1$), Eqns. (11.77)–(11.78) reduce to

$$\mu_G(u, v) = A_G(u, v), \quad (11.79)$$

$$\sigma_G^2(u, v) = B_G(u, v) - A_G^2(u, v), \quad (11.80)$$

with A_G, B_G as defined already.

This suggests a very simple process for computing the local average and variance by Gaussian filtering, as summarized in Alg. 11.9. The width (standard deviation σ) of the Gaussian kernel is set to 0.6 times the radius r of the corresponding disk filter to produce a similar effect as Alg. 11.8. The Gaussian approach has two advantages: First, the Gaussian makes a much superior low-pass filter, compared to the box or disk kernels. Second, the 2D Gaussian is (unlike the circular disk kernel) separable in the x - and y -direction, which permits a very efficient implementation of the 2D filter using only a pair of 1D convolutions (see Ch. 5, Sec. 5.2).

For practical calculation, A_G, B_G can be represented as (floating-point) images, and most modern image-processing environments provide efficient (multi-scale) implementations of Gaussian filters with large-size kernels. In ImageJ, fast Gaussian filtering is implemented by the class `GaussianBlur` with the public methods `blur()`, `blurGaussian()`, and `blurFloat()`, which all use normalized filter kernels by default. Programs 11.2–11.3 show the complete ImageJ implementation of Niblack’s thresholder using Gaussian smoothing kernels.

11.3 Java Implementation

All thresholding methods described in this chapter have been implemented as part of the `imagingbook` library that is available with full source code at the book’s website. The top class in this library¹³ is `Thresholder` with the sub-classes `GlobalThresholder` and `AdaptiveThresholder` for the methods described in Secs. 11.1 and 11.2, respectively. Class `Thresholder` itself is abstract and only defines a set of (non-public) utility methods for histogram analysis.

¹³ Package `imagingbook.pub.threshold`.

```

1: AdaptiveThresholdGauss( $I, r, \kappa, d, bg$ )
Input:  $I$ , intensity image of size  $M \times N$ ;  $r$ , support region radius;  $\kappa$ , variance control parameter;  $d$ , minimum offset;  $bg \in \{\text{dark, bright}\}$ , background type.
Returns a map  $Q$  of local thresholds for the grayscale image  $I$ .
2:  $(M, N) \leftarrow \text{Size}(I)$ 
3: Create maps  $A, B, Q : M \times N \mapsto \mathbb{R}$ 
4: for all image coordinates  $(u, v) \in M \times N$  do
5:    $A(u, v) \leftarrow I(u, v)$ 
6:    $B(u, v) \leftarrow (I(u, v))^2$ 
7:    $H^G \leftarrow \text{MakeGaussianKernel2D}(0.6 \cdot r)$ 
8:    $A \leftarrow A * H^G$                                  $\triangleright$  filter the original image with  $H^G$ 
9:    $B \leftarrow B * H^G$                                  $\triangleright$  filter the squared image with  $H^G$ 
10:  for all image coordinates  $(u, v) \in M \times N$  do
11:     $\mu_G \leftarrow A(u, v)$                             $\triangleright$  Eq. 11.79
12:     $\sigma_G \leftarrow \sqrt{B(u, v) - A^2(u, v)}$            $\triangleright$  Eq. 11.80
13:     $Q(u, v) \leftarrow \begin{cases} \mu_G + (\kappa \cdot \sigma_G + d) & \text{if } bg = \text{dark} \\ \mu_G - (\kappa \cdot \sigma_G + d) & \text{if } bg = \text{bright} \end{cases}$   $\triangleright$  Eq. 11.72
14:  return  $Q$ 
15: MakeGaussianKernel2D( $\sigma$ )
Returns a discrete 2D Gaussian kernel  $H$  with std. deviation  $\sigma$ , sized sufficiently large to avoid truncation effects.
16:  $r \leftarrow \max(1, \lceil 3.5 \cdot \sigma \rceil)$             $\triangleright$  size the kernel sufficiently large
17: Create map  $H : [-r, r]^2 \mapsto \mathbb{R}$ 
18:  $s \leftarrow 0$ 
19: for  $x \leftarrow -r, \dots, r$  do
20:   for  $y \leftarrow -r, \dots, r$  do
21:      $H(x, y) \leftarrow e^{-\frac{x^2+y^2}{2\sigma^2}}$            $\triangleright$  unnormalized 2D Gaussian
22:      $s \leftarrow s + H(x, y)$ 
23:   for  $x \leftarrow -r, \dots, r$  do
24:     for  $y \leftarrow -r, \dots, r$  do
25:        $H(x, y) \leftarrow \frac{1}{s} \cdot H(x, y)$             $\triangleright$  normalize  $H$ 
26:   return  $H$ 

```

11.3 JAVA IMPLEMENTATION

Alg. 11.9

Adaptive thresholding using Gaussian averaging (extended from Alg. 11.8). Parameters are the original image I , the radius r of the Gaussian kernel, variance control k , and minimum offset d . The argument to bg should be `dark` if the image background is darker than the structures of interest, `bright` if the background is brighter than the objects. The procedure `MakeGaussianKernel2D(σ)` creates a discrete, normalized 2D Gaussian kernel with standard deviation σ .

11.3.1 Global Thresholding Methods

The thresholding methods covered in Sec. 11.1 are implemented by the following classes:

- `MeanThreshold`, `MedianThreshold` (Sec. 11.1.2),
- `QuantileThreshold` (Alg. 11.1),
- `IsodataThreshold` (Alg. 11.2–11.3),
- `OtsuThreshold` (Alg. 11.4),
- `MaxEntropyThreshold` (Alg. 11.5), and
- `MinErrorThreshold` (Alg. 11.6).

These are sub-classes of the (abstract) class `GlobalThreshold`. The following example demonstrates the typical use of this method for a given `ByteProcessor` object I :

```

...
GlobalThreshold thr = new IsodataThreshold();
int q = thr.getThreshold(I);

```

11 AUTOMATIC THRESHOLDING

Prog. 11.2

Niblack's thresholder using Gaussian smoothing kernels (ImageJ implementation of Alg. 11.9, part 1).

```
1 package threshold;
2
3 import ij.plugin.filter.GaussianBlur;
4 import ij.plugin.filter.RankFilters;
5 import ij.process.ByteProcessor;
6 import ij.process.FloatProcessor;
7 import imagingbook.pub.threshold.BackgroundMode;
8
9 public abstract class NiblackThresholder extends
10     AdaptiveThresholder {
11
12     // parameters for this thresholder
13     public static class Parameters {
14         public int radius = 15;
15         public double kappa = 0.30;
16         public int dMin = 5;
17         public BackgroundMode bgMode = BackgroundMode.DARK;
18     }
19
20     private final Parameters params; // parameter object
21
22     protected FloatProcessor Imean; // =  $\mu_G(u, v)$ 
23     protected FloatProcessor Isigma; // =  $\sigma_G(u, v)$ 
24
25     public ByteProcessor getThreshold(ByteProcessor I) {
26         int w = I.getWidth();
27         int h = I.getHeight();
28
29         makeMeanAndVariance(I, params);
30         ByteProcessor Q = new ByteProcessor(w, h);
31
32         final double kappa = params.kappa;
33         final int dMin = params.dMin;
34         final boolean darkBg =
35             (params.bgMode == BackgroundMode.DARK);
36
37         for (int v = 0; v < h; v++) {
38             for (int u = 0; u < w; u++) {
39                 double sigma = Isigma.getf(u, v);
40                 double mu = Imean.getf(u, v);
41                 double diff = kappa * sigma + dMin;
42                 int q = (int)
43                     Math.rint((darkBg) ? mu + diff : mu - diff);
44                 if (q < 0) q = 0;
45                 if (q > 255) q = 255;
46                 Q.set(u, v, q);
47             }
48         }
49     }
50
51     // continues in Prog. 11.3
```

```

52 // continued from Prog. 11.2
53
54 public static class Gauss extends NiblackThresholder {
55
56     protected void makeMeanAndVariance(ByteProcessor I,
57         Parameters params) {
58         int width = I.getWidth();
59         int height = I.getHeight();
60
61         Imean = new FloatProcessor(width,height);
62         Isigma = new FloatProcessor(width,height);
63
64         FloatProcessor A = I.convertToFloatProcessor(); // = I
65         FloatProcessor B = I.convertToFloatProcessor(); // = I
66         B.sqr(); // = I2
67
68         GaussianBlur gb = new GaussianBlur();
69         double sigma = params.radius * 0.6;
70         gb.blurFloat(A, sigma, sigma, 0.002); // = A
71         gb.blurFloat(B, sigma, sigma, 0.002); // = B
72
73         for (int v = 0; v < height; v++) {
74             for (int u = 0; u < width; u++) {
75                 float a = A.getf(u, v);
76                 float b = B.getf(u, v);
77                 float sigmaG =
78                     (float) Math.sqrt(b - a*a); // Eq. 11.80
79                 Imean.setf(u, v, a); // = μG(u, v)
80                 Isigma.setf(u, v, sigmaG); // = σG(u, v)
81             }
82         }
83     } // end of inner class NiblackThresholder.Gauss
84 } // end of class NiblackThresholder

```

```

    if (q > 0) I.threshold(q);
    else ...

```

Here `threshold()` is the built-in ImageJ's method defined by class `ImageProcessor`.

11.3.2 Adaptive Thresholding

The techniques described in Sec. 11.2 are implemented by the following classes:

- `BernsenThreshold` (Alg. 11.7),
- `NiblackThresholder` (Alg. 11.8, multiple versions), and
- `SauvolaThreshold` (Eqn. (11.73)).

These are sub-classes of the (abstract) class `AdaptiveThresholder`. The following example demonstrates the typical use of these methods for a given `ByteProcessor` object `I`:

```

...
AdaptiveThresholder thr = new BernsenThresholder();

```

11.3 JAVA IMPLEMENTATION

Prog. 11.3

Niblack's thresholder using Gaussian smoothing kernels (part 2). The floating-point images A_G and B_G correspond to the maps A_G (filtered original image) and B_G (filtered squared image) in Alg. 11.9. An instance of the ImageJ class `GaussianBlur` is created in line 67 and subsequently used to filter both images in lines 69–70. The last argument to the ImageJ method `blurFloat(0.002)` specifies the accuracy of the Gaussian kernel.

```
ByteProcessor Q = thr.getThreshold(I);
thr.threshold(I, Q);
...
```

The 2D threshold surface is represented by the image `Q`; the method `threshold(I, Q)` is defined by class `AdaptiveThresholder`. Alternatively, the same operation can be performed without making `Q` explicit, as demonstrated by the following code segment:

```
...
// Create and set up a parameter object:
Parameters params = new BernsenThresholder.Parameters();
params.radius = 15;
params.cmin = 15;
params.bgMode = BackgroundMode.DARK;

// Create the thresholder:
AdaptiveThresholder thr = new BernsenThresholder(params);

// Perform the threshold operation:
thr.threshold(I);
...
```

This example also shows how to specify a parameter object (`params`) for the instantiation of the thresholder.

11.4 Summary and Further Reading

The intention of this chapter was to give an overview of established methods for automatic image thresholding. A vast body of relevant literature exists, and thus only a fraction of the proposed techniques could be discussed here. For additional approaches and references, several excellent surveys are available, including [86, 178, 204, 231] and [213].

Given the obvious limitations of global techniques, adaptive thresholding methods have received continued interest and are still a focus of ongoing research. Another popular approach is to calculate an adaptive threshold through image decomposition. In this case, the image is partitioned into (possibly overlapping) tiles, an “optimal” threshold is calculated for each tile and the adaptive threshold is obtained by interpolation between adjacent tiles. Another interesting idea, proposed in [260], is to specify a “threshold surface” by sampling the image at specific points that exhibit a high gradient, with the assumption that these points are at transitions between the background and the foreground. From these irregularly spaced point samples, a smooth surface is interpolated that passes through the sample points. Interpolation between these irregularly spaced point samples is done by solving a Laplacian difference equation to obtain a continuous “potential surface”. This is accomplished with the so-called “successive over-relaxation” method, which requires about N scans over an image of size $N \times N$ to converge, so its time complexity is an expensive $\mathcal{O}(N^3)$. A more efficient approach was proposed in [26], which uses a hierarchical, multi-scale algorithm for interpolating the threshold surface. Similarly, a quad-tree representation

was used for this purpose in [49]. Another interesting concept is “kriging” [175], which was originally developed for interpolating 2D geological data [190, Ch. 3, Sec. 3.7.4].

11.5 EXERCISES

In the case of color images, simple thresholding is often applied individually to each color channel and the results are subsequently merged using a suitable logical operation. Transformation to a non-RGB color space (such as HSV or CIELAB) might be helpful for this purpose. For a binarization method aimed specifically at vector-valued images, see [159], for example. Since thresholding can be viewed as a specific form of segmentation, color segmentation methods [50, 53, 85, 216] are also relevant for binarizing color images.

11.5 Exercises

Exercise 11.1. Define a procedure for estimating the minimum and maximum pixel value of an image from its histogram. Threshold the image at the resulting mid-range value (see Eqn. (11.12)). Can anything be said about the size of the resulting partitions?

Exercise 11.2. Define a procedure for estimating the median of an image from its histogram. Threshold the image at the resulting median value (see Eqn. (11.11)) and verify that the foreground and background partitions are of approximately equal size.

Exercise 11.3. The algorithms described in this chapter assume 8-bit grayscale input images (of type `ByteProcessor` in ImageJ). Adopt the current implementations to work with 16-bit integer image (of type `ShortProcessor`). Images of this type may contain pixel values in the range $[0, 2^{16}-1]$ and the `getHistogram()` method returns the histogram as an integer array of length 65536.

Exercise 11.4. Implement simple thresholding for RGB color images by thresholding each (scalar-valued) color channel individually and then merging the results by performing a pixel-wise AND operation. Compare the results to those obtained by thresholding the corresponding grayscale (luminance) images.

Exercise 11.5. Re-implement the Bernsen and/or Niblack thresholder (classes `BernsenThresholder` and `NiblackThresholder`) using integral images (see Ch. 3, Sec. 3.8) for efficiently calculating the required local mean and variance of the input image over a rectangular support region R .

Color Images

Color images are involved in every aspect of our lives, where they play an important role in everyday activities such as television, photography, and printing. Color perception is a fascinating and complicated phenomenon that has occupied the interests of scientists, psychologists, philosophers, and artists for hundreds of years [211, 217]. In this chapter, we focus on those technical aspects of color that are most important for working with digital color images. Our emphasis will be on understanding the various representations of color and correctly utilizing them when programming. Additional color-related issues, such as colorimetric color spaces, color quantization, and color filters, are covered in subsequent chapters.

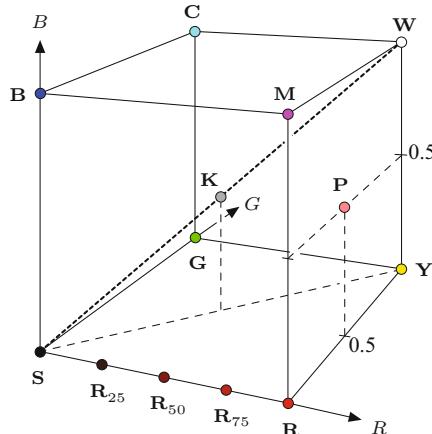
12.1 RGB Color Images

The RGB color schema encodes colors as combinations of the three primary colors: red, green, and blue (R, G, B). This scheme is widely used for transmission, representation, and storage of color images on both analog devices such as television sets and digital devices such as computers, digital cameras, and scanners. For this reason, many image-processing and graphics programs use the RGB schema as their internal representation for color images, and most language libraries, including Java’s imaging APIs, use it as their standard image representation.

RGB is an *additive* color system, which means that all colors start with black and are created by adding the primary colors. You can think of color formation in this system as occurring in a dark room where you can overlay three beams of light—one red, one green, and one blue—on a sheet of white paper. To create different colors, you would modify the intensity of each of these beams independently. The distinct intensity of each primary color beam controls the shade and brightness of the resulting color. The colors gray and white are created by mixing the three primary color beams at the same intensity. A similar operation occurs on the screen of a color television or

Fig. 12.1

Representation of the RGB color space as a 3D unit cube. The primary colors red (R), green (G), and blue (B) form the coordinate system. The “pure” red color (\mathbf{R}), green (\mathbf{G}), blue (\mathbf{B}), cyan (\mathbf{C}), magenta (\mathbf{M}), and yellow (\mathbf{Y}) lie on the vertices of the color cube. All the shades of gray, of which \mathbf{K} is an example, lie on the diagonal between black \mathbf{S} and white \mathbf{W} .



Pt.	Color	RGB values		
		R	G	B
S	Black	0.00	0.00	0.00
R	Red	1.00	0.00	0.00
Y	Yellow	1.00	1.00	0.00
G	Green	0.00	1.00	0.00
C	Cyan	0.00	1.00	1.00
B	Blue	0.00	0.00	1.00
M	Magenta	1.00	0.00	1.00
W	White	1.00	1.00	1.00
K	50% Gray	0.50	0.50	0.50
R₇₅	75% Red	0.75	0.00	0.00
R₅₀	50% Red	0.50	0.00	0.00
R₂₅	25% Red	0.25	0.00	0.00
P	Pink	1.00	0.50	0.50

CRT¹-based computer monitor, where tiny, close-lying dots of red, green, and blue phosphorous are simultaneously excited by a stream of electrons to distinct energy levels (intensities), creating a seemingly continuous color image.

The RGB color space can be visualized as a 3D unit cube in which the three primary colors form the coordinate axis. The RGB values are positive and lie in the range $[0, C_{\max}]$; for most digital images, $C_{\max} = 255$. Every possible color \mathbf{C}_i corresponds to a point within the RGB color cube of the form

$$\mathbf{C}_i = (R_i, G_i, B_i),$$

where $0 \leq R_i, G_i, B_i \leq C_{\max}$. RGB values are often normalized to the interval $[0, 1]$ so that the resulting color space forms a unit cube (Fig. 12.1). The point $\mathbf{S} = (0, 0, 0)$ corresponds to the color black, $\mathbf{W} = (1, 1, 1)$ corresponds to the color white, and all the points lying on the diagonal between \mathbf{S} and \mathbf{W} are shades of gray created from equal color components $R = G = B$.

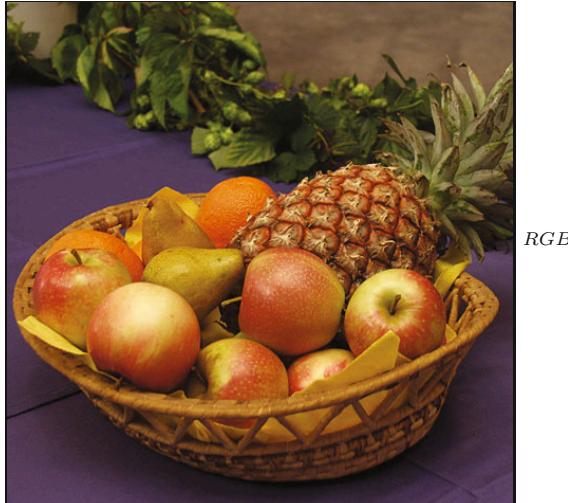
Figure 12.2 shows a color test image and its corresponding RGB color components, displayed here as intensity images. We will refer to this image in a number of examples that follow in this chapter.

RGB is a very simple color system, and as demonstrated in Sec. 12.2, a basic knowledge of it is often sufficient for processing color images or transforming them into other color spaces. At this point, we will not be able to determine what color a particular RGB pixel corresponds to in the real world, or even what the primary colors red, green, and blue truly mean in a physical (i.e., colorimetric) sense. For now we rely on our intuitive understanding of color and will address colorimetry and color spaces later in the context of the CIE color system (see Ch. 14).

12.1.1 Structure of Color Images

Color images are represented in the same way as grayscale images, by using an array of pixels in which different models are used to order the

¹ Cathode ray tube.



RGB



R

G

B

12.1 RGB COLOR IMAGES

Fig. 12.2

A color image and its corresponding RGB channels. The fruits depicted are mainly yellow and red and therefore have high values in the *R* and *G* channels. In these regions, the *B* content is correspondingly lower (represented here by darker gray values) except for the bright highlights on the apple, where the color changes gradually to white. The tabletop in the foreground is purple and therefore displays correspondingly higher values in its *B* channel.

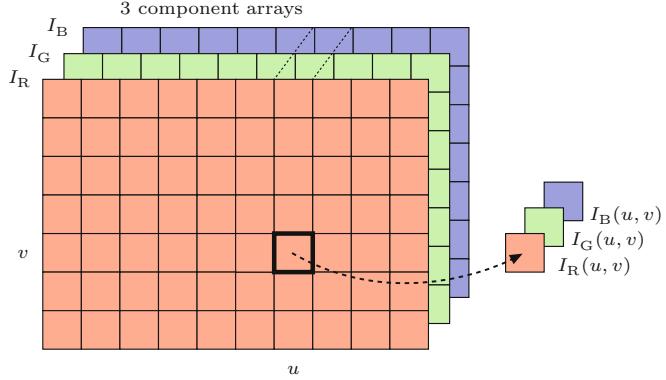
individual color components. In the next sections we will examine the difference between *true color* images, which utilize colors uniformly selected from the entire color space, and so-called *palleted* or *indexed* images, in which only a select set of distinct colors are used. Deciding which type of image to use depends on the requirements of the application. Farbbilder werden üblicherweise, genau wie Grauwertbilder, als Arrays von Pixeln dargestellt, wobei unterschiedliche Modelle für die Anordnung der einzelnen Farbkomponenten verwendet werden. Zunächst ist zu unterscheiden zwischen *Vollfarbenbildern*, die den gesamten Farbraum gleichförmig abdecken können, und so genannten *Paletten-* oder *Indexbildern*, die nur eine beschränkte Zahl unterschiedlicher Farben verwenden. Beide Bildtypen werden in der Praxis häufig eingesetzt.

True color images

A pixel in a true color image can represent any color in its color space, as long as it falls within the (discrete) range of its individual color components. True color images are appropriate when the image contains many colors with subtle differences, as occurs in digital photography and photo-realistic computer graphics. Next we look at two methods of ordering the color components in true color images: *component ordering* and *packed ordering*.

12 COLOR IMAGES

Fig. 12.3
RGB color image in component ordering. The three color components are laid out in separate arrays I_R , I_G , I_B of the same size.



Component ordering

In *component ordering* (also referred to as *planar ordering*) the color components are laid out in separate arrays of identical dimensions. In this case, the color image

$$\mathbf{I}_{\text{comp}} = (I_R, I_G, I_B) \quad (12.1)$$

can be thought of as a vector of related intensity images I_R , I_G , and I_B (Fig. 12.3), and the RGB values of the color image I at position (u, v) are obtained by accessing the three component images in the form

$$\begin{pmatrix} R(u, v) \\ G(u, v) \\ B(u, v) \end{pmatrix} = \begin{pmatrix} I_R(u, v) \\ I_G(u, v) \\ I_B(u, v) \end{pmatrix}. \quad (12.2)$$

Packed ordering

In *packed ordering*, the component values that represent the color of a particular pixel are packed together into a single element of the image array (Fig. 12.4) such that

$$\mathbf{I}_{\text{pack}}(u, v) = (R, G, B). \quad (12.3)$$

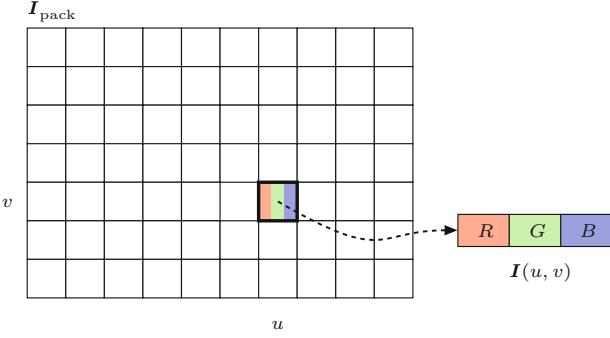
The RGB value of a packed image I at the location (u, v) is obtained by accessing the individual components of the color pixel as

$$\begin{pmatrix} R(u, v) \\ G(u, v) \\ B(u, v) \end{pmatrix} = \begin{pmatrix} \text{Red}(\mathbf{I}_{\text{pack}}(u, v)) \\ \text{Green}(\mathbf{I}_{\text{pack}}(u, v)) \\ \text{Blue}(\mathbf{I}_{\text{pack}}(u, v)) \end{pmatrix}. \quad (12.4)$$

The access functions `Red()`, `Green()`, `Blue()`, will depend on the specific implementation used for encoding the color pixels.

Indexed images

Indexed images permit only a limited number of distinct colors and therefore are used mostly for illustrations and graphics that contain large regions of the same color. Often these types of images are stored in indexed GIF or PNG files for use on the Web. In these indexed



12.1 RGB COLOR IMAGES

Fig. 12.4

RGB-color image using packed ordering. The three color components R , G , and B are placed together in a single array element.

images, the pixel array does not contain color or brightness data but instead consists of integer numbers k that are used to index into a *color table* or “palette”

$$\mathbf{P} = (\mathbf{P}_r, \mathbf{P}_g, \mathbf{P}_b) : [0, Q-1]^3 \mapsto [0, K-1]. \quad (12.5)$$

Here Q denotes the size of the color table, equal to the maximum number of distinct image colors (typically $Q = 2, \dots, 256$). K is the number of distinct component values (typ. $K = 256$). This table contains a specific color vector $\mathbf{P}(q) = (R_q, G_q, B_q)$ for every color index $q = 0, \dots, Q-1$ (see Fig. 12.5). The RGB component values of an indexed image I_{idx} at position (u, v) are obtained as

$$\begin{pmatrix} R(u, v) \\ G(u, v) \\ B(u, v) \end{pmatrix} = \begin{pmatrix} R_q \\ G_q \\ B_q \end{pmatrix} = \begin{pmatrix} \mathbf{P}_r(q) \\ \mathbf{P}_g(q) \\ \mathbf{P}_b(q) \end{pmatrix}, \quad (12.6)$$

with the index $q = I_{\text{idx}}(u, v)$. To allow proper reconstruction, the color table \mathbf{P} must of course be stored and/or transmitted along with the indexed image.

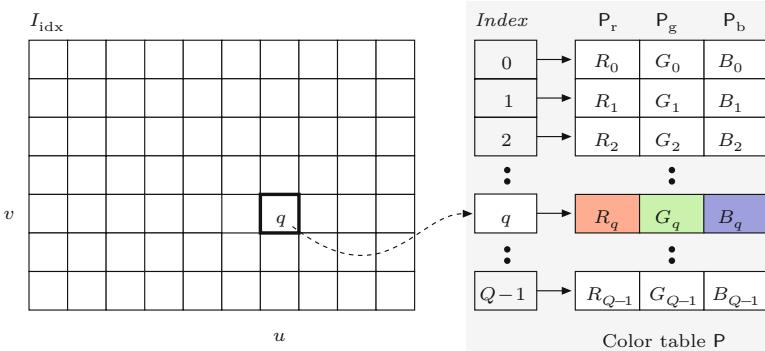


Fig. 12.5

RGB indexed image. The image array I_{idx} itself does not contain any color component values. Instead, each cell contains an index $q \in [0, Q-1]$, into the associated color table (“palette”) \mathbf{P} . The actual color value is specified by the table entry $\mathbf{P}_q = (R_q, G_q, B_q)$.

During the transformation from a true color image to an indexed image (e.g., from a JPEG image to a GIF image), the problem of optimal color reduction, or *color quantization*, arises. Color quantization is the process of determining an optimal color table and then mapping it to the original colors. This process is described in detail in Chapter 13.

12.1.2 Color Images in ImageJ

ImageJ provides two simple types of color images:

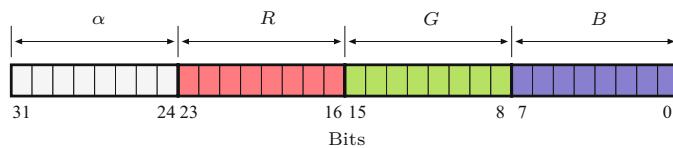
- RGB full-color images (24-bit “RGB color”).
- Indexed images (“8-bit color”).

RGB true color images

RGB color images in ImageJ use a packed order (see Sec. 12.1.1), where each color pixel is represented by a 32-bit `int` value. As Fig. 12.6 illustrates, 8 bits are used to represent each of the RGB components, which limits the range of the individual components to 0–255. The remaining 8 bits are reserved for the transparency,² or *alpha* (α), component. This is also the usual ordering in Java³ for RGB color images.

Fig. 12.6

Structure of a packed RGB color pixel in Java. Within a 32-bit `int`, 8 bits are allocated, in the following order, for each of the color components R , G , B , and the transparency value α (unused in ImageJ).



Accessing RGB pixel values

RGB color images are represented by an array of pixels, the elements of which are standard Java `ints`. To disassemble the packed `int` value into the three color components, you apply the appropriate bitwise shifting and masking operations. In the following example, we assume that the image processor `ip` (of type `ColorProcessor`) contains an RGB color image:

```
int c = ip.getPixel(u,v); // a packed RGB color pixel
int r = (c & 0xff0000) >> 16; // red component
int g = (c & 0x00ff00) >> 8; // green component
int b = (c & 0x0000ff); // blue component
```

In this example, each of the RGB components of the packed pixel `c` are isolated using a bitwise AND operation (`&`) with an appropriate bit mask (following convention, bit masks are given in hexadecimal⁴ notation), and afterwards the extracted bits are shifted right by 16 (for R) or 8 (for G) bit positions (see Fig. 12.7).

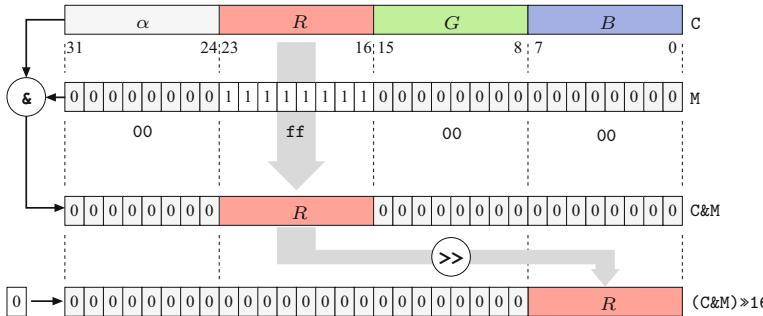
The “assembly” of an RGB pixel from separate R , G , and B values works in the opposite direction using the bitwise OR operator (`|`) and shifting the bits left (`<<`):

```
int r = 169; // red component
int g = 212; // green component
int b = 17; // blue component
int c = ((r & 0xff) << 16) | ((g & 0xff) << 8) | b & 0xff;
ip.putPixel(u, v, c);
```

² The transparency value α (alpha) represents the ability to see through a color pixel onto the background. At this time, the α channel is unused in ImageJ.

³ Java Advanced Window Toolkit (AWT).

⁴ The mask `0xff0000` is of type `int` and represents the 32-bit binary pattern `00000000111111100000000000000000`.



12.1 RGB COLOR IMAGES

Fig. 12.7

Decomposition of a 32-bit RGB color pixel using bit operations. The R component (bits 16–23) of the RGB pixels C (above) is isolated using a bitwise AND operation ($\&$) together with a bit mask $M = 0xffff0000$. All bits except the R component are set to the value 0, while the bit pattern within the R component remains unchanged. This bit pattern is subsequently shifted 16 positions to the right (\gg), so that the R component is moved into the lowest 8 bits and its value lies in the range of 0, …, 255. During the shift operation, zeros are filled in from the left.

```

1 // File Brighten_RGB_1.java
2 import ij.ImagePlus;
3 import ij.plugin.filter.PlugInFilter;
4 import ij.process.ImageProcessor;
5
6 public class Brighten_RGB_1 implements PlugInFilter {
7
8     public int setup(String arg, ImagePlus imp) {
9         return DOES_RGB; // this plugin works on RGB images
10    }
11
12    public void run(ImageProcessor ip) {
13        int[] pixels = (int[]) ip.getPixels();
14
15        for (int i = 0; i < pixels.length; i++) {
16            int c = pixels[i];
17            // split color pixel into rgb-components:
18            int r = (c & 0xffff0000) >> 16;
19            int g = (c & 0x00ff00) >> 8;
20            int b = (c & 0x0000ff);
21            // modify colors:
22            r = r + 10; if (r > 255) r = 255;
23            g = g + 10; if (g > 255) g = 255;
24            b = b + 10; if (b > 255) b = 255;
25            // reassemble color pixel and insert into pixel array:
26            pixels[i]
27                = ((r & 0xff)<<16) | ((g & 0xff)<<8) | b & 0xff;
28        }
29    }
30 }
```

Prog. 12.1

Processing RGB color data with the use of bit operations (ImageJ plugin, version 1). This plugin increases the values of all three color components by 10 units. It demonstrates the use of direct access to the pixel array (line 16), the separation of color components using bit operations (lines 18–20), and the reassembly of color pixels after modification (line 27). The value `DOES_RGB` (defined in the interface `PlugInFilter`) returned by the `setup()` method indicates that this plugin is designed to work on RGB formatted true color images (line 9).

Masking the component values with `0xff` works in this case because, except for the bits in positions 0, …, 7 (values in the range 0–255), all the other bits are already set to zero. A complete example of manipulating an RGB color image using bit operations is presented in Prog. 12.1. Instead of accessing color pixels using ImageJ's access functions, these programs directly access the pixel array for increased efficiency.

The ImageJ class `ColorProcessor` provides an easy to use alternative which returns the separated RGB components (as an `int` array

12 COLOR IMAGES

Prog. 12.2

Working with RGB color images without bit operations (ImageJ plugin, version 2). This plugin increases the values of all three color components by 10 units using the access methods `getPixel(int, int, int[])` and `putPixel(int, int, int[])` from the class `ColorProcessor` (lines 21 and 25, respectively). Execution time is approximately four times higher than that of version 1 (Prog. 12.1) because of the additional method calls.

```
1 // File Brighten_RGB_2.java
2 import ij.ImagePlus;
3 import ij.plugin.filter.PlugInFilter;
4 import ij.process.ColorProcessor;
5 import ij.process.ImageProcessor;
6
7 public class Brighten_RGB_2 implements PlugInFilter {
8     static final int R = 0, G = 1, B = 2; // component indices
9
10    public int setup(String arg, ImagePlus imp) {
11        return DOES_RGB; // this plugin works on RGB images
12    }
13
14    public void run(ImageProcessor ip) {
15        // typecast the image to ColorProcessor (no duplication):
16        ColorProcessor cp = (ColorProcessor) ip;
17        int[] RGB = new int[3];
18
19        for (int v = 0; v < cp.getHeight(); v++) {
20            for (int u = 0; u < cp.getWidth(); u++) {
21                cp.getPixel(u, v, RGB);
22                RGB[R] = Math.min(RGB[R] + 10, 255); // add 10 and
23                RGB[G] = Math.min(RGB[G] + 10, 255); // limit to 255
24                RGB[B] = Math.min(RGB[B] + 10, 255);
25                cp.putPixel(u, v, RGB);
26            }
27        }
28    }
29 }
```

with three elements). In the following example, which demonstrates its use, `ip` is of type `ColorProcessor`:

```
int[] RGB = new int[3];
...
ip.getPixel(u, v, RGB); // modifies RGB
int r = RGB[0];
int g = RGB[1];
int b = RGB[2];
...
ip.putPixel(u, v, RGB);
```

A more detailed and complete example is shown by the simple plugin in Prog. 12.2, which increases the value of all three color components of an RGB image by 10 units. Notice that the plugin limits the resulting component values to 255, because the `putPixel()` method only uses the lowest 8 bits of each component and does not test if the value passed in is out of the permitted 0–255 range. Without this test, arithmetic overflow errors can occur. The price for using this access method, instead of direct array access, is a noticeably longer running time (approximately a factor of 4 when compared to the version in Prog. 12.1).

ImageJ supports the following types of image formats for RGB true color images:

- **TIFF** (uncompressed only): $3 \times 8 RGB. TIFF color images with 16-bit depth are opened as an image stack consisting of three 16-bit intensity images.$
- **BMP, JPEG**: $3 \times 8 RGB.$
- **PNG**: $3 \times 8 RGB.$
- **RAW**: using the ImageJ menu **File > Import > Raw**, RGB images can be opened whose format is not directly supported by ImageJ. It is then possible to select different arrangements of the color components.

Creating RGB color images

The simplest way to create a new RGB image using ImageJ is to use an instance of the class **ColorProcessor**, as the following example demonstrates:

```
int w = 640, h = 480;
ColorProcessor cp = new ColorProcessor(w, h);
(new ImagePlus("My New Color Image", cp)).show();
```

When needed, the color image can be displayed by creating an instance of the class **ImagePlus** and calling its **show()** method. Since **cip** is of type **ColorProcessor**, the resulting **ImagePlus** object **cimg** is also a color image.

Indexed color images

The structure of an indexed image in ImageJ is given in [Fig. 12.5](#), where each element of the index array is 8 bits and therefore can represent a maximum of 256 different colors. When programming, indexed images are similar to grayscale images, as both make use of a color table to determine the actual color of the pixel. Indexed images differ from grayscale images only in that the contents of the color table are not intensity values but RGB values.

Opening and saving indexed images

ImageJ supports the indexed images in GIF, PNG, BMP, and TIFF format with index values of 1–8 bits (i.e., 2–256 distinct colors) and $3 \times 8 color values.$

Processing indexed images

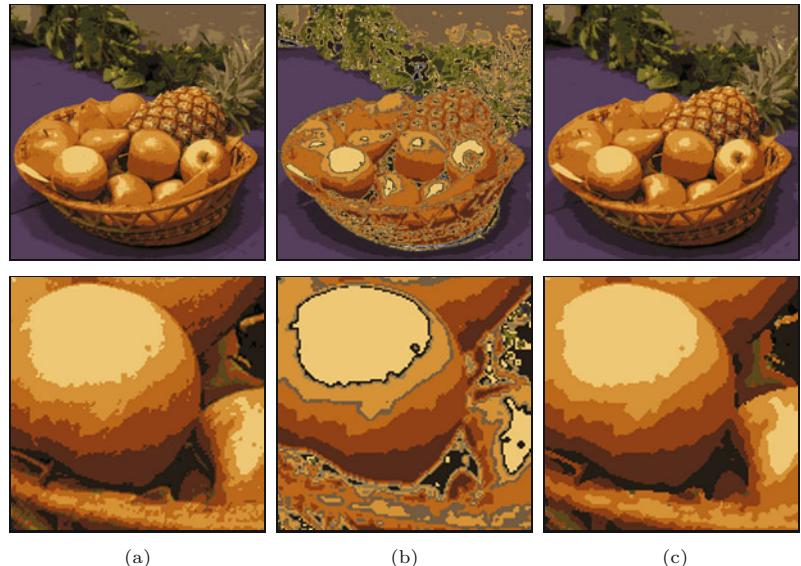
The indexed format is mostly used as a space-saving means of image storage and is not directly useful as a processing format since an index value in the pixel array is arbitrarily related to the actual color, found in the color table, that it represents. When working with indexed images it usually makes no sense to base any numerical interpretations on the pixel values or to apply any filter operations designed for 8-bit intensity images. [Figure 12.8](#) illustrates an example of applying a Gaussian filter and a median filter to the pixels of an indexed image. Since there is no meaningful quantitative relation between the actual colors and the index values, the results are erratic.

12 COLOR IMAGES

Fig. 12.8

Improper application of smoothing filters to an indexed color image. Indexed image with 16 colors (a) and results of applying a linear smoothing filter (b) and a 3×3 median filter (c) to the pixel array (that is, the *index* values). The application of a linear filter makes no sense, of course, since no meaningful relation exists between the index values in the pixel array and the actual image intensities.

While the median filter (c) delivers seemingly plausible results in this case, its use is also inadmissible because no meaningful ordering relation exists between the index values.



Note that even the use of the median filter is inadmissible because no ordering relation exists between the index values. Thus, with few exceptions, ImageJ functions do not permit the application of such operations to indexed images. Generally, when processing an indexed image, you first convert it into a true color RGB image and then after processing convert it back into an indexed image.

When an ImageJ plugin is supposed to process indexed images, its `setup()` method should return the `DOES_8C` (“8-bit color”) flag. The plugin in Prog. 12.3 shows how to increase the intensity of the three color components of an indexed image by 10 units (analogously to Progs. 12.1 and 12.2 for RGB images). Notice how in indexed images only the palette is modified and the original pixel data, the index values, remain the same. The color table of `ImageProcessor` is accessible through a `ColorModel`⁵ object, which can be read using the method `getColorModel()` and modified using `setColorModel()`.

The `ColorModel` object for indexed images (as well as 8-bit grayscale images) is a subtype of `IndexColorModel`, which contains three color tables (*maps*) representing the red, green, and blue components as separate `byte` arrays. The size of these tables ($2, \dots, 256$) can be determined by calling the method `getMapSize()`. Note that the elements of the palette should be interpreted as *unsigned* bytes with values ranging from $0, \dots, 255$. Just as with grayscale pixel values, during the conversion to `int` values, these color component values must also be bitwise masked with `0xff` as shown in Prog. 12.3 (lines 30–32).

As a further example, Prog. 12.4 shows how to convert an indexed image to a true color RGB image of type `ColorProcessor`. Conversion in this direction poses no problems because the RGB component values for a particular pixel are simply taken from the corresponding color table entry, as described by Eqn. (12.6). On the other hand,

⁵ Defined in the standard Java class `java.awt.image.ColorModel`.

```

1 // File Brighten_Index_Image.java
2
3 import ij.ImagePlus;
4 import ij.plugin.filter.PlugInFilter;
5 import ij.process.ImageProcessor;
6 import java.awt.image.IndexColorModel;
7
8 public class Brighten_Index_Image implements PlugInFilter {
9
10    public int setup(String arg, ImagePlus imp) {
11        return DOES_8C; // this plugin works on indexed color images
12    }
13
14    public void run(ImageProcessor ip) {
15        IndexColorModel icm =
16            (IndexColorModel) ip.getColorModel();
17        int pixBits = icm.getPixelSize();
18        int nColors = icm.getMapSize();
19
20        //retrieve the current lookup tables (maps) for R, G, B:
21        byte[] pRed = new byte[nColors];
22        byte[] pGrn = new byte[nColors];
23        byte[] pBlu = new byte[nColors];
24        icm.getReds(pRed);
25        icm.getGreens(pGrn);
26        icm.getBlues(pBlu);
27
28        //modify the lookup tables:
29        for (int idx = 0; idx < nColors; idx++){
30            int r = 0xff & pRed[idx]; // mask to treat as unsigned byte
31            int g = 0xff & pGrn[idx];
32            int b = 0xff & pBlu[idx];
33            pRed[idx] = (byte) Math.min(r + 10, 255);
34            pGrn[idx] = (byte) Math.min(g + 10, 255);
35            pBlu[idx] = (byte) Math.min(b + 10, 255);
36        }
37        //create a new color model and apply to the image:
38        IndexColorModel icm2 =
39            new IndexColorModel(pixBits,nColors,pRed,pGrn,pBlu);
40        ip.setColorModel(icm2);
41    }
42 }

```

12.1 RGB COLOR IMAGES

Prog. 12.3

Working with indexed images (ImageJ plugin). This plugin increases the brightness of an image by 10 units by modifying the image's color table (palette). The actual values in the pixel array, which are indices into the palette, are not changed.

conversion in the other direction requires *quantization* of the RGB color space and is as a rule more difficult and involved (see Ch. 13 for details). In practice, most applications make use of existing conversion methods such as those provided by the ImageJ API.

Creating indexed images

In ImageJ, no special method is provided for the creation of indexed images, so in almost all cases they are generated by converting an existing image. The following method demonstrates how to directly create an indexed image if required:

```
ByteProcessor makeIndexColorImage(int w, int h, int nColors) {
```

12 COLOR IMAGES

Prog. 12.4

Converting an indexed image to a true color RGB image (ImageJ plugin).

```
1 // File Index_To_Rgb.java
2
3 import ij.ImagePlus;
4 import ij.plugin.filter.PlugInFilter;
5 import ij.process.ColorProcessor;
6 import ij.process.ImageProcessor;
7 import java.awt.image.IndexColorModel;
8
9 public class Index_To_Rgb implements PlugInFilter {
10     static final int R = 0, G = 1, B = 2;
11     ImagePlus imp;
12
13     public int setup(String arg, ImagePlus imp) {
14         this.imp = imp;
15         return DOES_8C + NO_CHANGES; // does not alter original image
16     }
17
18     public void run(ImageProcessor ip) {
19         int w = ip.getWidth();
20         int h = ip.getHeight();
21
22         // retrieve the lookup tables (maps) for R, G, B:
23         IndexColorModel icm =
24             (IndexColorModel) ip.getColorModel();
25         int nColors = icm.getMapSize();
26         byte[] pRed = new byte[nColors];
27         byte[] pGrn = new byte[nColors];
28         byte[] pBlu = new byte[nColors];
29         icm.getReds(pRed);
30         icm.getGreens(pGrn);
31         icm.getBlues(pBlu);
32
33         // create a new 24-bit RGB image:
34         ColorProcessor cp = new ColorProcessor(w, h);
35         int[] RGB = new int[3];
36         for (int v = 0; v < h; v++) {
37             for (int u = 0; u < w; u++) {
38                 int idx = ip.getPixel(u, v);
39                 RGB[R] = 0xFF & pRed[idx];
40                 RGB[G] = 0xFF & pGrn[idx];
41                 RGB[B] = 0xFF & pBlu[idx];
42                 cp.putPixel(u, v, RGB);
43             }
44         }
45         ImagePlus cwin =
46             new ImagePlus(imp.getShortTitle() + " (RGB)", cp);
47         cwin.show();
48     }
49 }
```

```
byte[] rMap = new byte[nColors]; // red, green, blue color maps
byte[] gMap = new byte[nColors];
byte[] bMap = new byte[nColors];
// color maps need to be filled here
byte[] pixels = new byte[w * h];
```

```
IndexColorModel cm  
    = new IndexColorModel(8, nColors, rMap, gMap, bMap);  
    return new ByteProcessor(w, h, pixels, cm);  
}
```

12.2 COLOR SPACES AND COLOR CONVERSION

The parameter `nColors` defines the number of colors (and thus the size of the palette) and must be a value in the range of $2, \dots, 256$. To use the above template, you would complete it with code that filled the three byte arrays for the RGB components (`rMap`, `gMap`, `bMap`) and the index array (`pixels`) with the appropriate values.

Transparency

Transparency is one of the reasons indexed images are often used for Web graphics. In an indexed image, it is possible to define one of the index values so that it is displayed in a transparent manner and at selected image locations the background beneath the image shows through. In Java this can be controlled when creating the image's color model (`IndexColorModel`). As an example, to make color index 2 in Prog. 12.3 transparent, line 39 would need to be modified as follows:

```
int tidx = 2; // index of transparent color  
IndexColorModel icm2 =  
    new IndexColorModel(pixBits,nColors,pRed,pGrn,pBlu,tidx);  
ip.setColorModel(icm2);
```

At this time, however, ImageJ does not support the transparency property; it is not considered during display, and it is lost when the image is saved.

12.2 Color Spaces and Color Conversion

The RGB color system is well-suited for use in programming, as it is simple to manipulate and maps directly to the typical display hardware. When modifying colors within the RGB space, it is important to remember that the *metric*, or *measured distance* within this color space, does not proportionally correspond to our perception of color (e.g., doubling the value of the red component does not necessarily result in a color which appears to be twice as red). In general, in this space, modifying different color points by the same amount can cause very different changes in color. In addition, brightness changes in the RGB color space are also perceived as nonlinear.

Since changing any component modifies color tone, saturation, and brightness all at once, color selection in RGB space is difficult and quite non-intuitive. Color selection is more intuitive in other color spaces, such as the HSV space (see Sec. 12.2.3), since perceptual color features, such as saturation, are represented individually and can be modified independently. Alternatives to the RGB color space are also used in applications such as the automatic separation of objects from a colored background (the *blue box* technique in television), encoding television signals for transmission, or in printing, and are thus also relevant in digital image processing.

12 COLOR IMAGES

Fig. 12.9

Examples of the color distribution of natural images. Original images: landscape photograph with dominant green and blue components and sun-spot image with rich red and yellow components (a). Distribution of image colors in RGB-space (b).

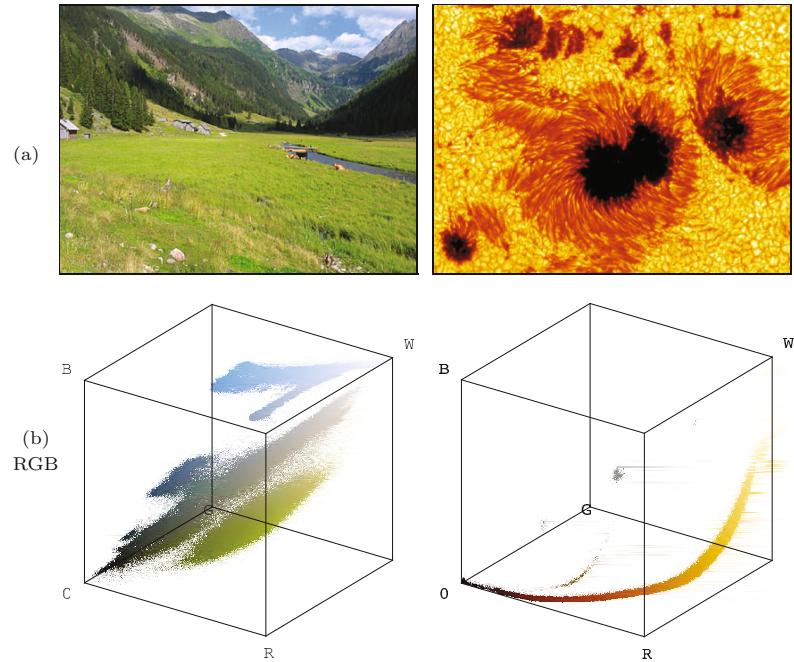


Figure 12.9 shows the distribution of the colors from natural images in the RGB color space. The first half of this section introduces alternative color spaces and the methods of converting between them, and later discusses the choices that need to be made to correctly convert a color image to grayscale. In addition to the classical color systems most widely used in programming, precise reference systems, such as the CIEXYZ color space, gain increasing importance in practical color processing.

12.2.1 Conversion to Grayscale

The conversion of an RGB color image to a grayscale image proceeds by computing the equivalent gray or *luminance* value Y for each RGB pixel. In its simplest form, Y could be computed as the average

$$Y = \text{Avg}(R, G, B) = \frac{R + G + B}{3} \quad (12.7)$$

of the three color components R , G , and B . Since we perceive both red and green as being substantially brighter than blue, the resulting image will appear to be too dark in the red and green areas and too bright in the blue ones. Therefore, a weighted sum of the color components is typically used for calculating the equivalent brightness or *luminance* in the form

$$Y = \text{Lum}(R, G, B) = w_R \cdot R + w_G \cdot G + w_B \cdot B \quad (12.8)$$

The weights most often used were originally developed for encoding analog color television signals (see Sec. 12.2.4) are

$$w_R = 0.299, \quad w_G = 0.587, \quad w_B = 0.114, \quad (12.9)$$

and the weights recommended in ITU-BT.709 [122] for digital color encoding are

$$w_R = 0.2126, \quad w_G = 0.7152, \quad w_B = 0.0722. \quad (12.10)$$

If each color component is assigned the same weight, as in Eqn. (12.7), this is of course just a special case of Eqn. (12.8).

Note that, although these weights were developed for use with TV signals, they are optimized for *linear* RGB component values, that is, signals with no gamma correction. In many practical situations, however, the RGB components are actually *nonlinear*, particularly when we work with sRGB images (see Ch. 14, Sec. 14.4). In this case, the RGB components must first be linearized to obtain the correct luminance values with the aforementioned weights.

In some color systems, instead of a weighted sum of the RGB color components, a nonlinear brightness function, for example the *value* V in HSV (Eqn. (12.14) in Sec. 12.2.3) or the *luminance* L in HLS (Eqn. (12.25)), is used as the intensity value Y .

Hueless (gray) color images

An RGB image is hueless or gray when the RGB components of each pixel $\mathbf{I}(u, v) = (R, G, B)$ are the same; that is, if

$$R = G = B.$$

Therefore, to completely remove the color from an RGB image, simply replace the R , G , and B component of each pixel with the equivalent gray value Y ,

$$\begin{pmatrix} R_{\text{gray}} \\ G_{\text{gray}} \\ B_{\text{gray}} \end{pmatrix} = \begin{pmatrix} Y \\ Y \\ Y \end{pmatrix}, \quad (12.11)$$

by using $Y = \text{Lum}(R, G, B)$ from Eqns. (12.8) and (12.9), for example. The resulting grayscale image should have the same subjective brightness as the original color image.

Grayscale conversion in ImageJ

In ImageJ, the simplest way to convert an RGB color image (of type `ColorProcessor`) into an 8-bit grayscale image is to use the `ImageProcessor`-method

```
convertToByteProcessor(),
```

which returns a new image of type `ByteProcessor`. ImageJ uses the default weights $w_R = w_G = w_B = \frac{1}{3}$ (as in Eqn. (12.7)) for the RGB components, or alternatively $w_R = 0.299$, $w_G = 0.587$, $w_B = 0.114$ (as in Eqn. (12.9)) if the “Weighted RGB Conversions” option is selected in the `Edit > Options > Conversions` dialog. Arbitrary component weights can be specified for subsequent conversion operations through the static `ColorProcessor` method

```
setRGBWeights(double wR, double wG, double wB).
```

Similarly, the static method `getWeightingFactors()` of class `ColorProcessor` can be used to retrieve the current component weights as a 3-element `double`-array. Note that no *linearization* is performed on the color components, which should be considered when working with (nonlinear) sRGB colors (see Ch. 14, Sec. 14.4 for details).

12.2.2 Desaturating RGB Color Images

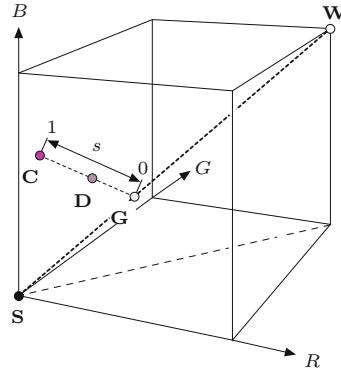
Desaturation is the uniform reduction of the amount of color in an RGB image in a *continuous* manner. It is done by replacing each RGB pixel by a desaturated color obtained by linear interpolation between the pixel's original color and the corresponding (Y, Y, Y) gray point in the RGB space, that is,

$$\begin{pmatrix} R_{\text{desat}} \\ G_{\text{desat}} \\ B_{\text{desat}} \end{pmatrix} = \begin{pmatrix} Y \\ Y \\ Y \end{pmatrix} + s \cdot \begin{pmatrix} R - Y \\ G - Y \\ B - Y \end{pmatrix}, \quad (12.12)$$

again with $Y = \text{Lum}(R, G, B)$ from Eqns. (12.8) and (12.9), where the factor $s \in [0, 1]$ controls the remaining amount of color saturation (Fig. 12.10). A value of $s = 0$ completely eliminates all color, resulting in a true grayscale image, and with $s = 1$ the color values will be unchanged. In Prog. 12.5, continuous desaturation as defined in Eqn. (12.12) is implemented as an ImageJ plugin.

In color spaces where color saturation is represented by an explicit component (such as HSV and HLS, for example), desaturation is of course much easier to accomplish (by simply reducing the saturation value to zero).

Fig. 12.10
Desaturation in RGB space:
original color point $C = (R, G, B)$, its corresponding
gray point $G = (Y, Y, Y)$,
and the desaturated color
point D . Saturation is con-
trolled by the factor s .



12.2.3 HSV/HSB and HLS Color Spaces

In the **HSV** color space, colors are specified by the components *hue*, *saturation*, and *value*. Often, such as in Adobe products and the Java API, the **HSV** space is called **HSB**. While the acronym is different (in this case $B = \text{brightness}$),⁶ it denotes the same color space. The HSV color space is traditionally shown as an upside-down, six-sided pyramid (Fig. 12.11(a)), where the vertical axis represents the V (brightness) value, the horizontal distance from the axis the S (saturation) value, and the angle the H (hue) value. The black point is at the tip of the pyramid and the white point lies in the center of the base. The three primary colors *red*, *green*, and *blue* and the pairwise mixed colors *yellow*, *cyan*, and *magenta* are the corner points of the

⁶ Sometimes the HSV space is also referred to as the “HSI” space, where “I” stands for *intensity*.

```

1 // File Desaturate_Rgb.java
2
3 import ij.ImagePlus;
4 import ij.plugin.filter.PlugInFilter;
5 import ij.process.ImageProcessor;
6
7 public class Desaturate_Rgb implements PlugInFilter {
8     double s = 0.3; // color saturation value
9
10    public int setup(String arg, ImagePlus imp) {
11        return DOES_RGB;
12    }
13
14    public void run(ImageProcessor ip) {
15        //iterate over all pixels:
16        for (int v = 0; v < ip.getHeight(); v++) {
17            for (int u = 0; u < ip.getWidth(); u++) {
18
19                // get int-packed color pixel:
20                int c = ip.get(u, v);
21
22                //extract RGB components from color pixel
23                int r = (c & 0xff0000) >> 16;
24                int g = (c & 0x00ff00) >> 8;
25                int b = (c & 0x0000ff);
26
27                // compute equiv. gray value:
28                double y = 0.299 * r + 0.587 * g + 0.114 * b;
29
30                // linear interpolate (yyy) ↔ (rgb):
31                r = (int) (y + s * (r - y));
32                g = (int) (y + s * (g - y));
33                b = (int) (y + s * (b - y));
34
35                // reassemble the color pixel:
36                c = ((r & 0xff)<<16) | ((g & 0xff)<<8) | b & 0xff;
37                ip.set(u, v, c);
38            }
39        }
40    }
41
42 }

```

12.2 COLOR SPACES AND COLOR CONVERSION

Prog. 12.5

Continuous desaturation of an RGB color image (ImageJ plugin). The amount of color saturation is controlled by the variable `s` defined in line 8 (see Eqn. (12.12)).

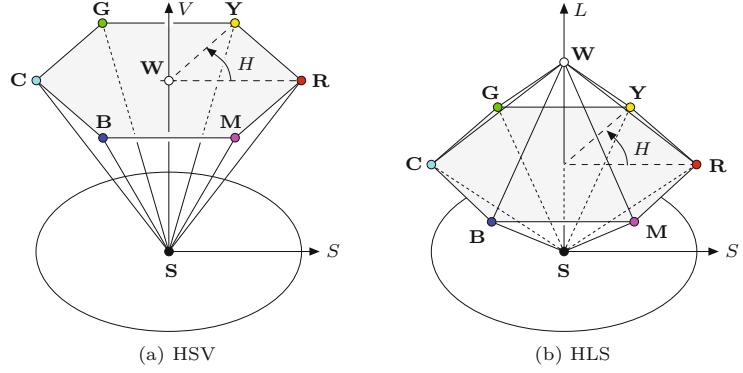
base. While this space is often represented as a pyramid, according to its mathematical definition, the space is actually a *cylinder*, as shown in Fig. 12.12.

The **HLS** color space⁷ (*hue, luminance, saturation*) is very similar to the HSV space, and the *hue* component is in fact completely identical in both spaces. The *luminance* and *saturation* values also correspond to the vertical axis and the radius, respectively, but are defined differently than in HSV space. The common representation of the HLS space is as a double pyramid (Fig. 12.11(b)), with black

⁷ The acronyms HLS and HSL are used interchangeably.

Fig. 12.11

HSV and HLS color space are traditionally visualized as a single or double hexagonal pyramid. The brightness V (or L) is represented by the vertical dimension, the color saturation S by the radius from the pyramid's axis, and the hue h by the angle. In both cases, the primary colors red (**R**), green (**G**), and blue (**B**) and the mixed colors yellow (**Y**), cyan (**C**), and magenta (**M**) lie on a common plane with black (**S**) at the tip. The essential difference between the HSV and HLS color spaces is the location of the white point (**W**).



on the bottom tip and white on the top. The primary colors lie on the corner points of the hexagonal base between the two pyramids. Even though it is often portrayed in this intuitive way, mathematically the HLS space is again a cylinder (see Fig. 12.15).

RGB→HSV conversion

To convert from RGB to the HSV color space, we first find the *saturation* of the RGB color components $R, G, B \in [0, C_{\max}]$, with C_{\max} being the maximum component value (typically 255), as

$$S_{\text{HSV}} = \begin{cases} \frac{C_{\text{rng}}}{C_{\text{high}}} & \text{for } C_{\text{high}} > 0, \\ 0 & \text{otherwise} \end{cases} \quad (12.13)$$

and the brightness (*value*)

$$V_{\text{HSV}} = \frac{C_{\text{high}}}{C_{\max}}, \quad (12.14)$$

with

$$\begin{aligned} C_{\text{low}} &= \min(R, G, B), & C_{\text{high}} &= \max(R, G, B), \\ C_{\text{rng}} &= C_{\text{high}} - C_{\text{low}}. \end{aligned} \quad (12.15)$$

Finally, we need to specify the *hue* value H_{HSV} . When all three RGB color components have the same value ($R = G = B$), then we are dealing with an *achromatic* (gray) pixel. In this particular case $C_{\text{rng}} = 0$ and thus the saturation value $S_{\text{HSV}} = 0$, consequently the hue is undefined. To calculate H_{HSV} when $C_{\text{rng}} > 0$, we first normalize each component using

$$R' = \frac{C_{\text{high}} - R}{C_{\text{rng}}}, \quad G' = \frac{C_{\text{high}} - G}{C_{\text{rng}}}, \quad B' = \frac{C_{\text{high}} - B}{C_{\text{rng}}}. \quad (12.16)$$

Then, depending on which of the three original color components had the maximal value, we compute a preliminary hue H' as

$$H' = \begin{cases} B' - G' & \text{for } R = C_{\text{high}}, \\ R' - B' + 2 & \text{for } G = C_{\text{high}}, \\ G' - R' + 4 & \text{for } B = C_{\text{high}}. \end{cases} \quad (12.17)$$

Since the resulting value for H' lies on the interval $[-1, 5]$, we obtain the final hue value by normalizing to the interval $[0, 1]$ as

$$H_{\text{HSV}} = \frac{1}{6} \cdot \begin{cases} (H' + 6) & \text{for } H' < 0, \\ H' & \text{otherwise.} \end{cases} \quad (12.18)$$

Hence all three components H_{HSV} , S_{HSV} , and V_{HSV} will lie within the interval $[0, 1]$. The hue value H_{HSV} can naturally also be computed in another angle interval, for example, in the 0 to 360° interval using

$$H_{\text{HSV}}^\circ = H_{\text{HSV}} \cdot 360. \quad (12.19)$$

Under this definition, the RGB space unit cube is mapped to a *cylinder* with height and radius of length 1 (Fig. 12.12). In contrast to the traditional representation (Fig. 12.11), all HSB points within the entire cylinder correspond to valid color coordinates in RGB space. The mapping from RGB to the HSV space is nonlinear, as can be noted by examining how the black point stretches completely across the cylinder's base. Figure 12.12 plots the location of some notable color points and compares them with their locations in RGB space (see also Fig. 12.1). Figure 12.13 shows the individual HSV components (in grayscale) of the test image in Fig. 12.2.

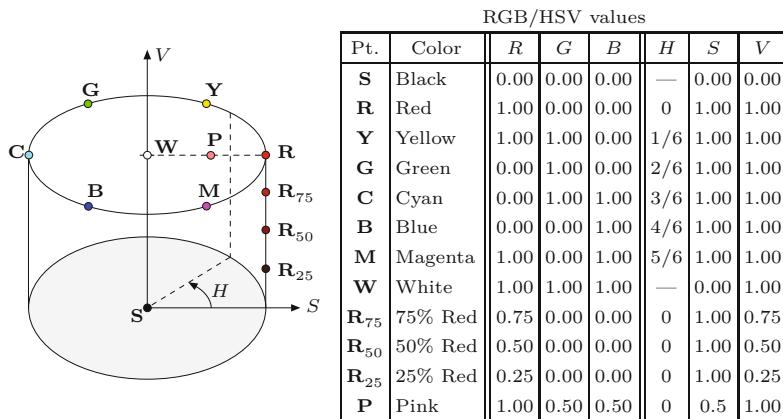


Fig. 12.12

HSV color space. The illustration shows the HSV color space as a cylinder with the coordinates H (hue) as the angle, S (saturation) as the radius, and V (brightness value) as the distance along the vertical axis, which runs between the black point \mathbf{S} and the white point \mathbf{W} . The table lists the (R, G, B) and (H, S, V) values of the color points marked on the graphic. Pure colors (composed of only one or two components) lie on the outer wall of the cylinder ($S = 1$), as exemplified by the gradually saturated reds (\mathbf{R}_{25} , \mathbf{R}_{50} , \mathbf{R}_{75} , \mathbf{R}).

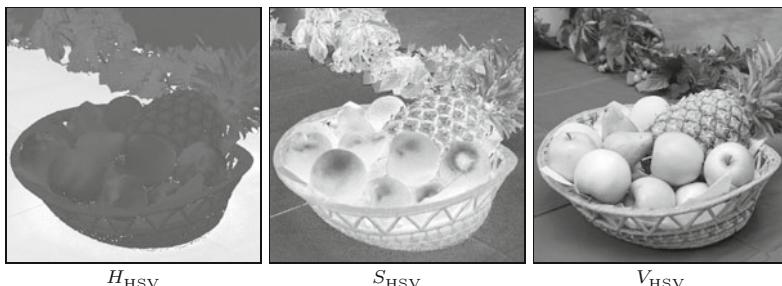


Fig. 12.13

HSV components for the test image in Fig. 12.2. The darker areas in the h_{HSV} component correspond to the red and yellow colors, where the hue angle is near zero.

Java implementation

In Java, the RGB→HSV conversion is implemented in the standard AWT `Color` class by the static method

```
float[] RGBtoHSB (int r, int g, int b, float[] hsv)
```

(HSV and HSB denote the same color space). The method takes three `int` arguments `r`, `g`, `b` (within the range [0, 255]) and returns a `float` array with the resulting H, S, V values in the interval [0, 1]. When an existing `float` array is passed as the argument `hsv`, then the result is placed in it; otherwise (when `hsv = null`) a new array is created. Here is a simple example:

```
import java.awt.Color;
...
float[] hsv = new float[3];
int red = 128, green = 255, blue = 0;
hsv = Color.RGBtoHSB (red, green, blue, hsv);
float h = hsv[0];
float s = hsv[1];
float v = hsv[2];
...
```

A possible implementation of the Java method `RGBtoHSB()` using the definition in Eqns. (12.14)–(12.18) is given in Prog. 12.6.

HSV→RGB conversion

To convert an HSV tuple $(H_{\text{HSV}}, S_{\text{HSV}}, V_{\text{HSV}})$, where H_{HSV} , S_{HSV} , and $V_{\text{HSV}} \in [0, 1]$, into the corresponding (R, G, B) color values, the appropriate color sector

$$H' = (6 \cdot H_{\text{HSV}}) \bmod 6 \quad (12.20)$$

(with $0 \leq H' < 6$) is determined first, followed by computing the intermediate values

$$\begin{aligned} c_1 &= \lfloor H' \rfloor, & x &= (1 - S_{\text{HSV}}) \cdot V_{\text{HSV}}, \\ c_2 &= H' - c_1, & y &= (1 - (S_{\text{HSV}} \cdot c_2)) \cdot V_{\text{HSV}}, \\ & & z &= (1 - (S_{\text{HSV}} \cdot (1 - c_2))) \cdot V_{\text{HSV}}. \end{aligned} \quad (12.21)$$

Depending on the value of c_1 , the normalized RGB values $R', G', B' \in [0, 1]$ are then calculated from $v = V_{\text{HSV}}$, x , y , and z as follows:⁸

$$(R', G', B') \leftarrow \begin{cases} (v, z, x) & \text{for } c_1 = 0, \\ (y, v, x) & \text{for } c_1 = 1, \\ (x, v, z) & \text{for } c_1 = 2, \\ (x, y, v) & \text{for } c_1 = 3, \\ (z, x, v) & \text{for } c_1 = 4, \\ (v, x, y) & \text{for } c_1 = 5. \end{cases} \quad (12.22)$$

Scaling the RGB components back to integer values in the range [0, 255] is carried out as follows:

$$\begin{aligned} R &\leftarrow \min(\text{round}(K \cdot R'), 255), \\ G &\leftarrow \min(\text{round}(K \cdot G'), 255), \\ B &\leftarrow \min(\text{round}(K \cdot B'), 255). \end{aligned} \quad (12.23)$$

⁸ The variables x , y , z used here are not related to the CIEXYZ color space (see Ch. 14, Sec. 14.1).

```

1 float[] RGBtoHSV (int[] RGB) {
2     int R = RGB[0], G = RGB[1], B = RGB[2]; // R, G, B ∈ [0, 255]
3     int cHi = Math.max(R,Math.max(G,B)); // max. comp. value
4     int cLo = Math.min(R,Math.min(G,B)); // min. comp. value
5     int cRng = cHi - cLo; // component range
6     float H = 0, S = 0, V = 0;
7     float cMax = 255.0f;
8
9     // compute value V
10    V = cHi / cMax;
11
12    // compute saturation S
13    if (cHi > 0)
14        S = (float) cRng / cHi;
15
16    // compute hue H
17    if (cRng > 0) { // hue is defined only for color pixels
18        float rr = (float)(cHi - R) / cRng;
19        float gg = (float)(cHi - G) / cRng;
20        float bb = (float)(cHi - B) / cRng;
21        float hh;
22        if (R == cHi) // R is largest component value
23            hh = bb - gg;
24        else if (G == cHi) // G is largest component value
25            hh = rr - bb + 2.0f;
26        else // B is largest component value
27            hh = gg - rr + 4.0f;
28        if (hh < 0)
29            hh = hh + 6;
30        H = hh / 6;
31    }
32    return new float[] {H, S, V};
33 }

```

12.2 COLOR SPACES AND COLOR CONVERSION

Prog. 12.6

RGB→HSV conversion (Java implementation). This Java method for RGB→HSV conversion follows the process given in the text to compute a single color tuple. It takes the same arguments and returns results identical to the standard `Color.RGBtoHSB()` method.

Java implementation

HSV→RGB conversion is implemented in Java's standard AWT `Color` class by the static method

```
int HSBtoRGB (float h, float s, float v),
```

which takes three `float` arguments $h, s, v \in [0, 1]$ and returns the corresponding RGB color as an `int` value with 3×8 bits arranged in the standard Java RGB format (see Fig. 12.6). One possible implementation of this method is shown in Prog. 12.7.

RGB→HLS conversion

In the HLS model, the *hue* value H_{HLS} is computed in the same way as in the HSV model (Eqns. (12.16)–(12.18)), that is,

$$H_{\text{HLS}} = H_{\text{HSV}}. \quad (12.24)$$

The other values, L_{HLS} and S_{HLS} , are calculated as follows (for C_{high} , C_{low} , and C_{rng} , see Eqn. (12.15)):

12 COLOR IMAGES

Prog. 12.7
HSV→RGB conversion
(Java implementation).

```

1   int HSVtoRGB (float[] HSV) {
2       float H = HSV[0], S = HSV[1], V = HSV[2]; // H, S, V ∈ [0, 1]
3       float r = 0, g = 0, b = 0;
4       float hh = (6 * H) % 6;      // h' ← (6 · h) mod 6
5       int c1 = (int) hh;          // c1 ← ⌊h'⌋
6       float c2 = hh - c1;
7       float x = (1 - S) * V;
8       float y = (1 - (S * c2)) * V;
9       float z = (1 - (S * (1 - c2))) * V;
10      switch (c1) {
11          case 0: r = V; g = z; b = x; break;
12          case 1: r = y; g = V; b = x; break;
13          case 2: r = x; g = V; b = z; break;
14          case 3: r = x; g = y; b = V; break;
15          case 4: r = z; g = x; b = V; break;
16          case 5: r = V; g = x; b = y; break;
17      }
18      int R = Math.min((int)(r * 255), 255);
19      int G = Math.min((int)(g * 255), 255);
20      int B = Math.min((int)(b * 255), 255);
21      return new int[] {R, G, B};
22  }

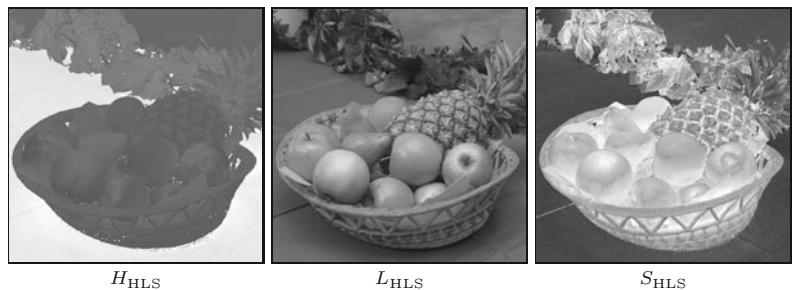
```

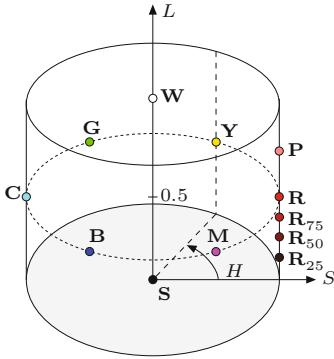
$$L_{\text{HLS}} = \frac{(C_{\text{high}} + C_{\text{low}})/255}{2}, \quad (12.25)$$

$$S_{\text{HLS}} = \begin{cases} 0 & \text{for } L_{\text{HLS}} = 0, \\ 0.5 \cdot \frac{C_{\text{rng}}/255}{L_{\text{HLS}}} & \text{for } 0 < L_{\text{HLS}} \leq 0.5, \\ 0.5 \cdot \frac{C_{\text{rng}}/255}{1-L_{\text{HLS}}} & \text{for } 0.5 < L_{\text{HLS}} < 1, \\ 0 & \text{for } L_{\text{HLS}} = 1. \end{cases} \quad (12.26)$$

Using the aforementioned definitions, the RGB color cube is again mapped to a cylinder with height and radius 1 (see Fig. 12.15). In contrast to the HSV space (Fig. 12.12), the primary colors lie together in the horizontal plane at $L_{\text{HLS}} = 0.5$ and the white point lies outside of this plane at $L_{\text{HLS}} = 1.0$. Using these nonlinear transformations, the black and the white points are mapped to the top and the bottom planes of the cylinder, respectively. All points inside HLS cylinder correspond to valid colors in RGB space. Figure 12.14 shows the individual HLS components of the test image as grayscale images.

Fig. 12.14
HLS color components H_{HLS} (hue), S_{HLS} (saturation), and L_{HLS} (luminance).





RGB/HLS values							
Pt.	Color	R	G	B	H	S	L
S	Black	0.00	0.00	0.00	—	0.00	0.00
R	Red	1.00	0.00	0.00	0	1.00	0.50
Y	Yellow	1.00	1.00	0.00	1/6	1.00	0.50
G	Green	0.00	1.00	0.00	2/6	1.00	0.50
C	Cyan	0.00	1.00	1.00	3/6	1.00	0.50
B	Blue	0.00	0.00	1.00	4/6	1.00	0.50
M	Magenta	1.00	0.00	1.00	5/6	1.00	0.50
W	White	1.00	1.00	1.00	—	0.00	1.00
R₇₅	75% Red	0.75	0.00	0.00	0	1.00	0.375
R₅₀	50% Red	0.50	0.00	0.00	0	1.00	0.250
R₂₅	25% Red	0.25	0.00	0.00	0	1.00	0.125
P	Pink	1.00	0.50	0.50	0/6	1.00	0.75

12.2 COLOR SPACES AND COLOR CONVERSION

Fig. 12.15

HLS color space. The illustration shows the HLS color space visualized as a cylinder with the coordinates H (hue) as the angle, S (saturation) as the radius, and L (lightness) as the distance along the vertical axis, which runs between the black point **S** and the white point **W**. The table lists the (R, G, B) and (H, S, L) values where “pure” colors (created using only one or two color components) lie on the lower half of the outer cylinder wall ($S = 1$), as illustrated by the gradually saturated reds (\mathbf{R}_{25} , \mathbf{R}_{50} , \mathbf{R}_{75} , \mathbf{R}). Mixtures of all three primary colors, where at least one of the components is completely saturated, lie along the upper half of the outer cylinder wall; for example, the point **P** (pink).

HLS→RGB conversion

When converting from HLS to the RGB space, we assume that H_{HLS} , S_{HLS} , $L_{\text{HLS}} \in [0, 1]$. In the case where $L_{\text{HLS}} = 0$ or $L_{\text{HLS}} = 1$, the result is

$$(R', G', B') = \begin{cases} (0, 0, 0) & \text{for } L_{\text{HLS}} = 0, \\ (1, 1, 1) & \text{for } L_{\text{HLS}} = 1. \end{cases} \quad (12.27)$$

Otherwise, we again determine the appropriate color sector

$$H' = (6 \cdot H_{\text{HLS}}) \bmod 6, \quad (12.28)$$

such that $0 \leq H' < 6$, and from this

$$c_1 = \lfloor H' \rfloor, \quad c_2 = H' - c_1, \quad (12.29)$$

$$d = \begin{cases} S_{\text{HLS}} \cdot L_{\text{HLS}} & \text{for } L_{\text{HLS}} \leq 0.5, \\ S_{\text{HLS}} \cdot (1 - L_{\text{HLS}}) & \text{for } L_{\text{HLS}} > 0.5, \end{cases} \quad (12.30)$$

and the quantities

$$w = L_{\text{HLS}} + d, \quad x = L_{\text{HLS}} - d, \quad (12.31)$$

$$y = w - (w - x) \cdot c_2, \quad z = x + (w - x) \cdot c_2. \quad (12.32)$$

The final mapping to the RGB values is (similar to Eqn. (12.22))

$$(R', G', B') = \begin{cases} (w, z, x) & \text{for } c_1 = 0, \\ (y, w, x) & \text{for } c_1 = 1, \\ (x, w, z) & \text{for } c_1 = 2, \\ (x, y, w) & \text{for } c_1 = 3, \\ (z, x, w) & \text{for } c_1 = 4, \\ (w, x, y) & \text{for } c_1 = 5. \end{cases} \quad (12.33)$$

Finally, scaling the normalized R' , G' , B' ($\in [0, 1]$) color components back to the $[0, 255]$ range is done as in Eqn. (12.23).

```

1   float[] RGBtoHLS (int[] RGB) {
2       int R = RGB[0], G = RGB[1], B = RGB[2]; // R,G,B in [0, 255]
3       float cHi = Math.max(R, Math.max(G, B));
4       float cLo = Math.min(R, Math.min(G, B));
5       float cRng = cHi - cLo;    // component range
6
7       // compute lightness L
8       float L = ((cHi + cLo) / 255f) / 2;
9
10      // compute saturation S
11      float S = 0;
12      if (0 < L && L < 1) {
13          float d = (L <= 0.5f) ? L : (1 - L);
14          S = 0.5f * (cRng / 255f) / d;
15      }
16
17      // compute hue H (same as in HSV)
18      float H = 0;
19      if (cHi > 0 && cRng > 0) {           // this is a color pixel!
20          float r = (float)(cHi - R) / cRng;
21          float g = (float)(cHi - G) / cRng;
22          float b = (float)(cHi - B) / cRng;
23          float h;
24          if (R == cHi)                      // R is largest component
25              h = b - g;
26          else if (G == cHi)                // G is largest component
27              h = r - b + 2.0f;
28          else                            // B is largest component
29              h = g - r + 4.0f;
30          if (h < 0)
31              h = h + 6;
32          H = h / 6;
33      }
34      return new float[] {H, L, S};
35  }

```

Java implementation

Currently there is no method in either the standard Java API or ImageJ for converting color values between RGB and HLS. Program 12.8 gives one possible implementation of the RGB→HLS conversion that follows the definitions in Eqns. (12.24)–(12.26). The HLS→RGB conversion is shown in Prog. 12.9.

HSV and HLS compared

Despite the obvious similarity between the two color spaces, as Fig. 12.16 illustrates, substantial differences in the V/L and S components do exist. The essential difference between the HSV and HLS spaces is the ordering of the colors that lie between the white point **W** and the “pure” colors (**R**, **G**, **B**, **Y**, **C**, **M**), which consist of at most two primary colors, at least one of which is completely saturated.

The difference in how colors are distributed in RGB, HSV, and HLS space is readily apparent in Fig. 12.17. The starting point was a distribution of 1331 ($11 \times 11 \times 11$) color tuples obtained by uniformly

```

1 float[] HLSToRGB (float[] HLS) {
2     float H = HLS[0], L = HLS[1], S = HLS[2]; // H,L,S in [0, 1]
3     float r = 0, g = 0, b = 0;
4     if (L <= 0)           // black
5         r = g = b = 0;
6     else if (L >= 1)      // white
7         r = g = b = 1;
8     else {
9         float hh = (6 * H) % 6;           // = H'
10        int c1 = (int) hh;
11        float c2 = hh - c1;
12        float d = (L <= 0.5f) ? (S * L) : (S * (1 - L));
13        float w = L + d;
14        float x = L - d;
15        float y = w - (w - x) * c2;
16        float z = x + (w - x) * c2;
17        switch (c1) {
18            case 0: r = w; g = z; b = x; break;
19            case 1: r = y; g = w; b = x; break;
20            case 2: r = x; g = w; b = z; break;
21            case 3: r = x; g = y; b = w; break;
22            case 4: r = z; g = x; b = w; break;
23            case 5: r = w; g = x; b = y; break;
24        }
25    } // r, g, b in [0, 1]
26    int R = Math.min(Math.round(r * 255), 255);
27    int G = Math.min(Math.round(g * 255), 255);
28    int B = Math.min(Math.round(b * 255), 255);
29    return new int[] {R, G, B};
30 }

```

12.2 COLOR SPACES AND COLOR CONVERSION

Prog. 12.9

HLS→RGB conversion (Java implementation).

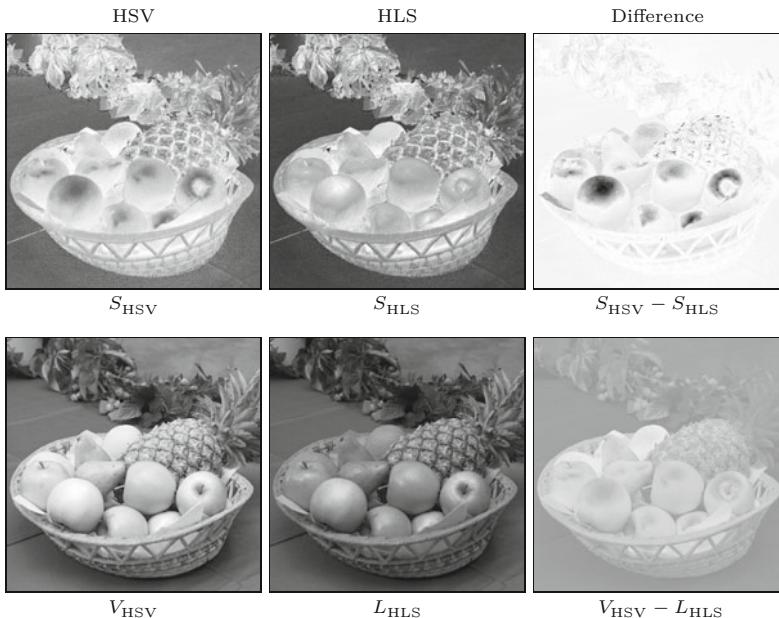
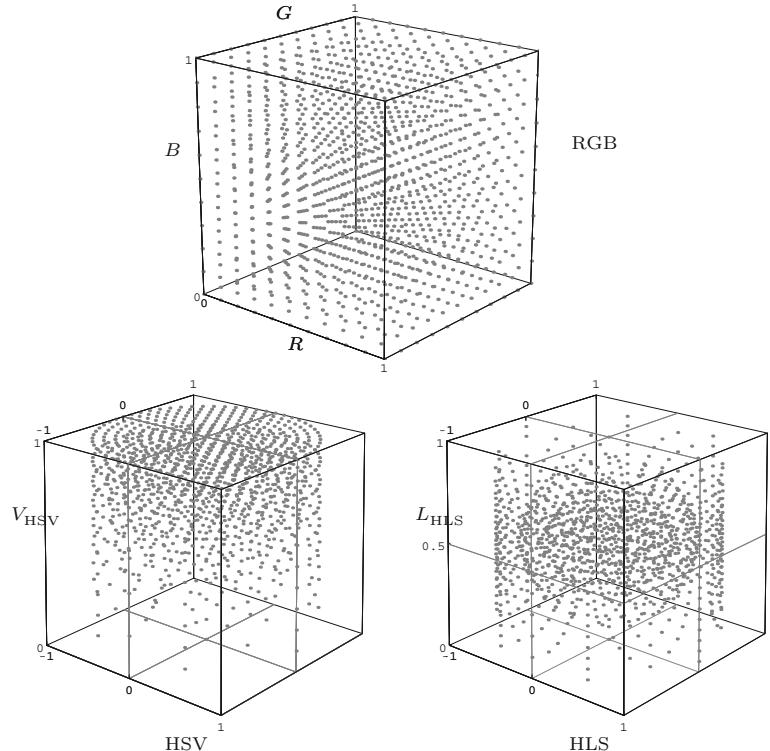


Fig. 12.16
HSV and HLS components compared. *Saturation* (top row) and *intensity* (bottom row). In the color *saturation* difference image $S_{\text{HSV}} - S_{\text{HLS}}$ (top), light areas correspond to positive values and dark areas to negative values. Saturation in the HLS representation, especially in the brightest sections of the image, is notably higher, resulting in negative values in the difference image. For the *intensity* (*value* and *luminance*, respectively) in general, $V_{\text{HSV}} \geq L_{\text{HLS}}$ and therefore the difference $V_{\text{HSV}} - L_{\text{HLS}}$ (bottom) is always positive. The *hue* component \hat{H} (not shown) is identical in both representations.

Fig. 12.17

Distribution of colors in the RGB, HSV, and HLS spaces. The starting point is the uniform distribution of colors in RGB space (top). The corresponding colors in the cylindrical spaces are distributed nonsymmetrically in HSV and symmetrically in HLS.



sampling the RGB space at an interval of 0.1 in each dimension. We can see clearly that in HSV space the maximally saturated colors ($s = 1$) form circular rings with increasing density toward the upper plane of the cylinder. In HLS space, however, the color samples are spread out symmetrically around the center plane and the density is significantly lower, particularly in the region near white. A given coordinate shift in this part of the color space leads to relatively small color changes, which allows the specification of very fine color grades in HLS space, especially for colors located in the upper half of the HLS cylinder.

Both the HSV and HLS color spaces are widely used in practice; for instance, for selecting colors in image editing and graphics design applications. In digital image processing, they are also used for *color keying* (i.e., isolating objects according to their *hue*) on a homogeneously colored background where the brightness is not necessarily constant.

Desaturation in HSV/HLS color space

Desaturation of color images (cf. Sec. 12.2.2) represented in HSV or HLS color space is trivial since color saturation is available as a separate component. In particular, pixels with zero saturation are uncolored or gray. For example, HSV colors can be gradually or fully desaturated by simply multiplying the component S by a fixed saturation factor $s \in [0, 1]$ and keeping H, V unchanged, that is,

$$\begin{pmatrix} H_{\text{desat}} \\ S_{\text{desat}} \\ V_{\text{desat}} \end{pmatrix} = \begin{pmatrix} H \\ s \cdot S \\ V \end{pmatrix}, \quad (12.34)$$

which works analogously with HLS colors. While Eqn. (12.34) applies equally to all colors, it might be interesting to *selectively* modify only colors with certain hues. This is easily accomplished by replacing the fixed saturation factor s by a hue-dependent function $f(H)$ (see also Exercise 12.6).

12.2.4 TV Component Color Spaces—YUV, YIQ, and YC_bC_r

These color spaces are an integral part of the standards surrounding the recording, storage, transmission, and display of television signals. YUV and YIQ are the fundamental color-encoding methods for the analog NTSC and PAL systems, and YC_bC_r is a part of the international standards governing digital television [114]. All of these color spaces have in common the idea of separating the luminance component Y from two chroma components and, instead of directly encoding colors, encoding color differences. In this way, compatibility with legacy black and white systems is maintained while at the same time the bandwidth of the signal can be optimized by using different transmission bandwidths for the brightness and the color components. Since the human visual system is not able to perceive detail in the color components as well as it does in the intensity part of a video signal, the amount of information, and consequently bandwidth, used in the color channel can be reduced to approximately 1/4 of that used for the intensity component. This fact is also used when compressing digital still images and is why, for example, the JPEG codec converts RGB images to YC_bC_r . That is why these color spaces are important in digital image processing, even though raw YIQ or YUV images are rarely encountered in practice.

YUV

YUV is the basis for the color encoding used in analog television in both the North American NTSC and the European PAL systems. The luminance component Y is computed, just as in Eqn. (12.9), from the RGB components as

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \quad (12.35)$$

under the assumption that the RGB values have already been gamma corrected according to the TV encoding standard ($\gamma_{\text{NTSC}} = 2.2$ and $\gamma_{\text{PAL}} = 2.8$, see Ch. 4, Sec. 4.7) for playback. The UV components are computed from a weighted difference between the luminance and the blue or red components as

$$U = 0.492 \cdot (B - Y) \quad \text{und} \quad V = 0.877 \cdot (R - Y), \quad (12.36)$$

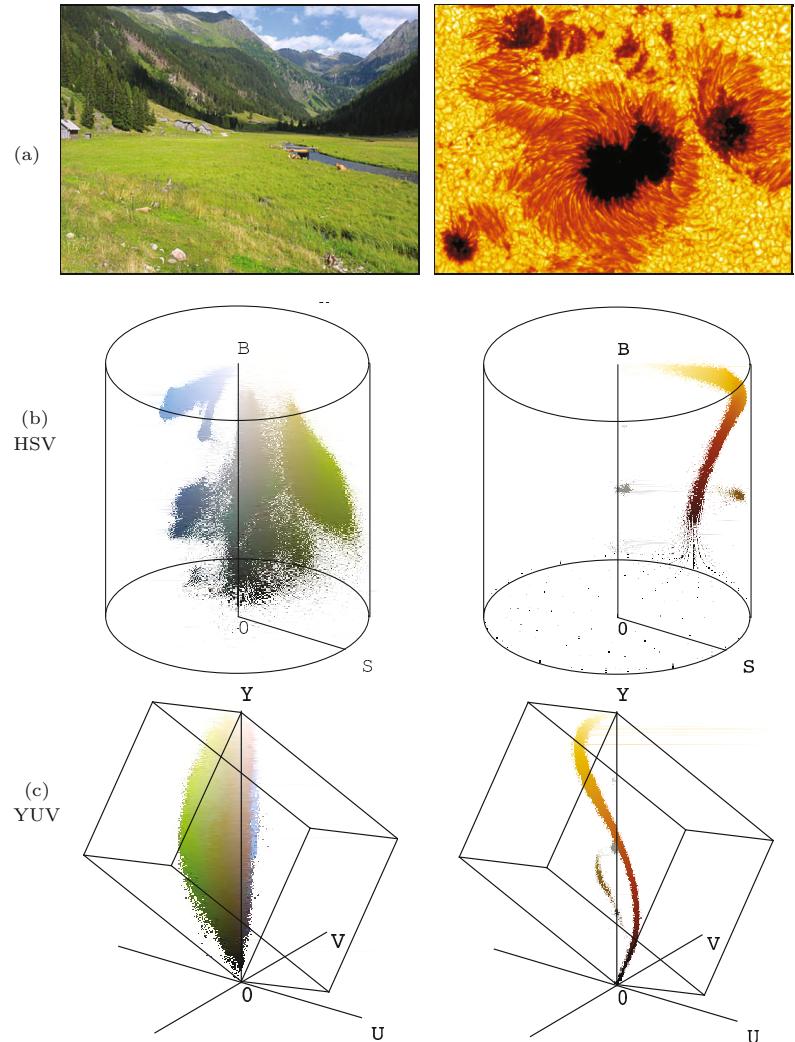
and the entire transformation from RGB to YUV is

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}. \quad (12.37)$$

12 COLOR IMAGES

Fig. 12.18

Examples of the color distribution of natural images in different color spaces. Original images (a); color distribution in HSV- (b), and YUV-space (c). See Fig. 12.9 for the corresponding distributions in RGB color space.



The transformation from YUV back to RGB is found by inverting the matrix in Eqn. (12.37):

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.000 & 1.140 \\ 1.000 & -0.395 & -0.581 \\ 1.000 & 2.032 & 0.000 \end{pmatrix} \cdot \begin{pmatrix} Y \\ U \\ V \end{pmatrix}. \quad (12.38)$$

The color distributions in YUV-space for a set of natural images are shown in [Fig. 12.18](#).

YIQ

The original NTSC system used a variant of YUV called YIQ (I for “in-phase”, Q for “quadrature”), where both the U and V color vectors were rotated and mirrored such that

$$\begin{pmatrix} I \\ Q \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} \cos \beta & \sin \beta \\ -\sin \beta & \cos \beta \end{pmatrix} \cdot \begin{pmatrix} U \\ V \end{pmatrix}, \quad (12.39)$$

where $\beta = 0.576$ (33°). The Y component is the same as in YUV. Although the YIQ has certain advantages with respect to bandwidth requirements it has been completely replaced by YUV [124, p. 240].

YC_bC_r

The YC_bC_r color space is an internationally standardized variant of YUV that is used for both digital television and image compression (e.g., in JPEG). The chroma components C_b, C_r are (similar to U, V) difference values between the luminance and the blue and red components, respectively. In contrast to YUV, the weights of the RGB components for the luminance Y depend explicitly on the coefficients used for the chroma values C_b and C_r [197, p. 16]. For arbitrary weights w_B, w_R , the transformation is defined as

$$Y = w_R \cdot R + (1 - w_B - w_R) \cdot G + w_B \cdot B, \quad (12.40)$$

$$C_b = \frac{0.5}{1 - w_B} \cdot (B - Y), \quad (12.41)$$

$$C_r = \frac{0.5}{1 - w_R} \cdot (R - Y), \quad (12.42)$$

with $w_R = 0.299$ and $w_B = 0.114$ ($w_G = 0.587$)⁹ according to ITU¹⁰ recommendation BT.601 [123]. Analogously, the reverse mapping from YC_bC_r to RGB is

$$R = Y + \frac{(1 - w_R) \cdot C_r}{0.5}, \quad (12.43)$$

$$G = Y - \frac{w_B \cdot (1 - w_B) \cdot C_b + w_R \cdot (1 - w_R) \cdot C_r}{0.5 \cdot (1 - w_B - w_R)}, \quad (12.44)$$

$$B = Y + \frac{(1 - w_B) \cdot C_b}{0.5}. \quad (12.45)$$

In matrix-vector notation this gives the linear transformation

$$\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}, \quad (12.46)$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.000 & 1.403 \\ 1.000 & -0.344 & -0.714 \\ 1.000 & 1.773 & 0.000 \end{pmatrix} \cdot \begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix}. \quad (12.47)$$

Different weights are recommended based on how the color space is used; for example, ITU-BT.709 [122] recommends $w_R = 0.2125$ and $w_B = 0.0721$ to be used in digital HDTV production. The values of U, V, I, Q , and C_b, C_r may be both positive or negative. To encode C_b, C_r values to digital numbers, a suitable offset is typically added to obtain positive-only values, for example, $128 = 2^7$ in case of 8-bit components.

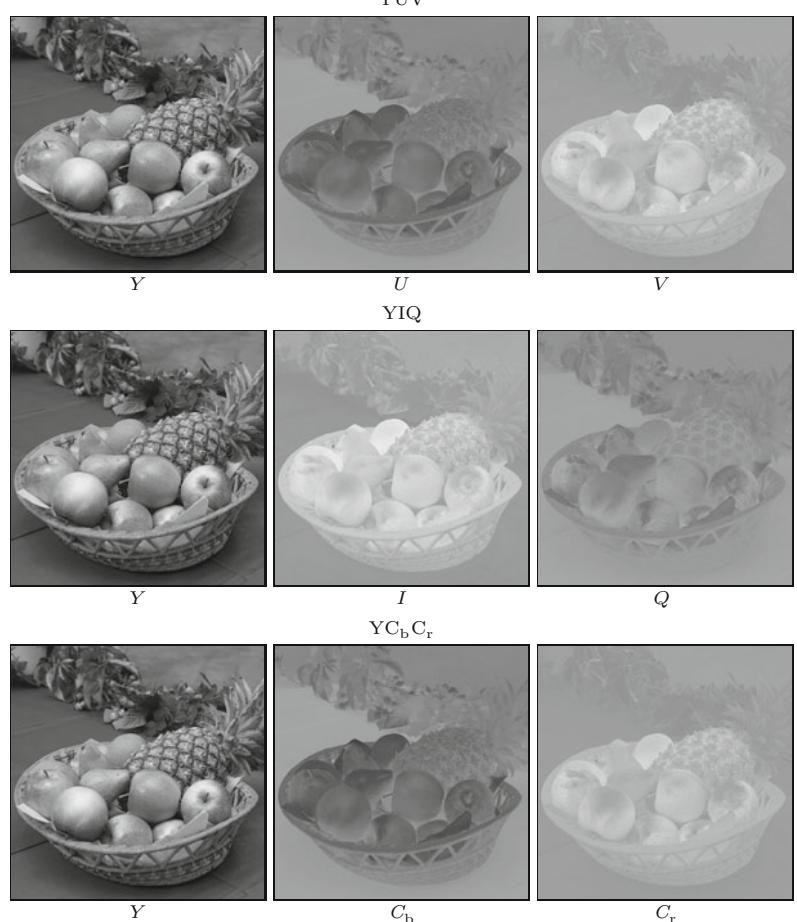
Figure 12.19 shows the three color spaces YUV, YIQ, and YC_bC_r together for comparison. The U, V, I, Q , and C_b, C_r values in the

⁹ $w_R + w_G + w_B = 1$.

¹⁰ International Telecommunication Union (www.itu.int).

12 COLOR IMAGES

Fig. 12.19
Comparing YUV-, YIQ-,
and YC_bC_r values. The
 Y values are identical
in all three color spaces.



right two frames have been offset by 128 so that the negative values are visible. Thus a value of zero is represented as medium gray in these images. The YC_bC_r encoding is practically indistinguishable from YUV in these images since they both use very similar weights for the color components.

12.2.5 Color Spaces for Printing—CMY and CMYK

In contrast to the *additive* RGB color scheme (and its various color models), color printing makes use of a *subtractive* color scheme, where each printed color reduces the intensity of the reflected light at that location. Color printing requires a minimum of three primary colors; traditionally *cyan* (C), *magenta* (M), and *yellow* (Y)¹¹ have been used.

Using subtractive color mixing on a white background, $C = M = Y = 0$ (no ink) results in the color *white* and $C = M = Y = 1$ (complete saturation of all three inks) in the color *black*. A cyan-colored ink will absorb *red* (R) most strongly, magenta absorbs *green*

¹¹ Note that in this case Y stands for *yellow* and is unrelated to the Y luma or luminance component in YUV or YC_bC_r .

(G), and yellow absorbs *blue* (B). The simplest form of the CMY model is defined as

$$C = 1 - R, \quad M = 1 - G, \quad Y = 1 - B. \quad (12.48)$$

In practice, the color produced by fully saturating all three inks is not physically a true black. Therefore, the three primary colors C, M, Y are usually supplemented with a black ink (K) to increase the color range and coverage (gamut). In the simplest case, the amount of black is

$$K = \min(C, M, Y). \quad (12.49)$$

With rising levels of black, however, the intensity of the C, M, Y components can be gradually reduced. Many methods for reducing the primary dyes have been proposed and we look at three of them in the following.

CMY→CMYK conversion (version 1)

In this simple variant the C, M, Y values are reduced linearly with increasing K (Eqn. (12.49)), which yields the modified components as

$$\begin{pmatrix} C_1 \\ M_1 \\ Y_1 \\ K_1 \end{pmatrix} = \begin{pmatrix} C - K \\ M - K \\ Y - K \\ K \end{pmatrix}. \quad (12.50)$$

CMY→CMYK conversion (version 2)

The second variant corrects the color by reducing the C, M, Y components by $s = \frac{1}{1-K}$, resulting in stronger colors in the dark areas of the image:

$$\begin{pmatrix} C_2 \\ M_2 \\ Y_2 \\ K_2 \end{pmatrix} = \begin{pmatrix} (C - K) \cdot s \\ (M - K) \cdot s \\ (Y - K) \cdot s \\ K \end{pmatrix}, \quad \text{with } s = \begin{cases} \frac{1}{1-K} & \text{for } K < 1, \\ 1 & \text{otherwise.} \end{cases} \quad (12.51)$$

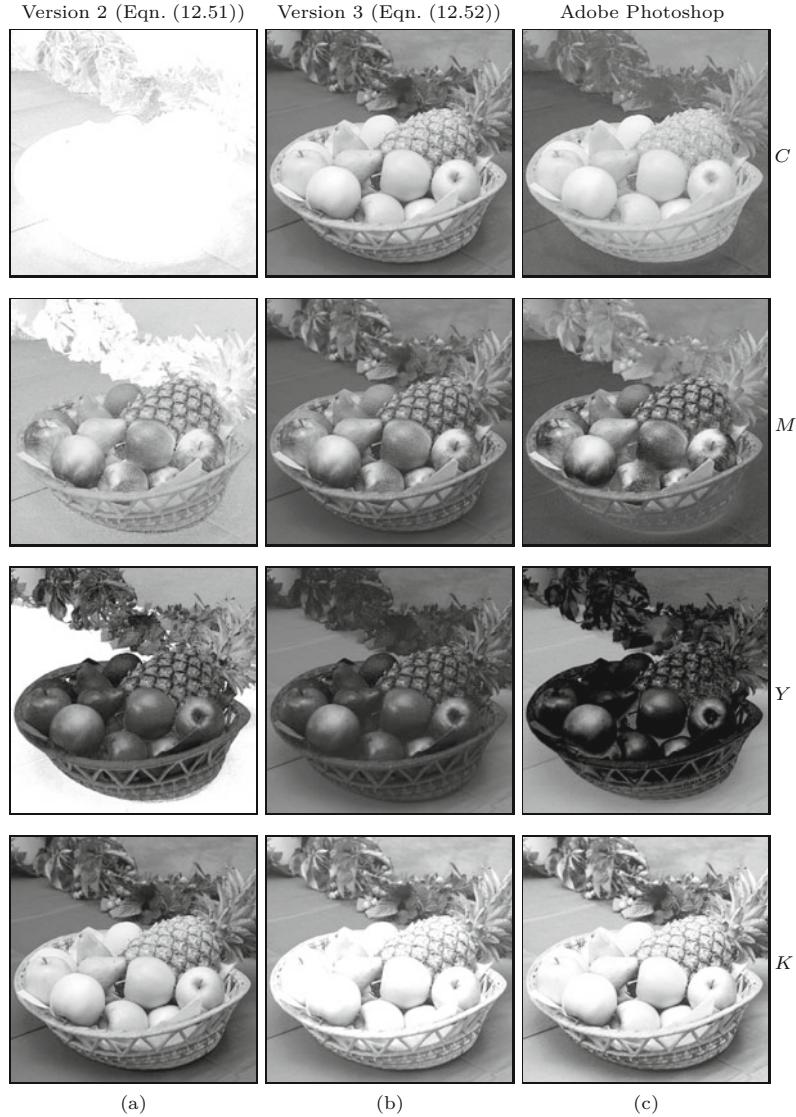
In both versions, the K component (as defined in Eqn. (12.49)) is used directly without modification, and all gray tones (that is, when $R = G = B$) are printed using black ink K , without any contribution from C, M , or Y .

While both of these simple definitions are widely used, neither one produces high quality results. [Figure 12.20\(a\)](#) compares the result from version 2 with that produced with Adobe Photoshop ([Fig. 12.20\(c\)](#)). The difference in the cyan component C is particularly noticeable and also the exceeding amount of black (K) in the brighter areas of the image.

In practice, the required amounts of black K and C, M, Y depend so strongly on the printing process and the type of paper used that print jobs are routinely calibrated individually.

Fig. 12.20

RGB→CMYK conversion comparison. Simple conversion using Eqn. (12.51) (a), applying the *undercolor-removal* and *black-generation* functions of Eqn. (12.52) (b), and results obtained with Adobe Photoshop (c). The color intensities are shown inverted, that is, darker areas represent higher CMYK color values. The simple conversion (a), in comparison with Photoshop's result (c), shows strong deviations in all color components, C and K in particular. The results in (b) are close to Photoshop's and could be further improved by tuning the corresponding function parameters.

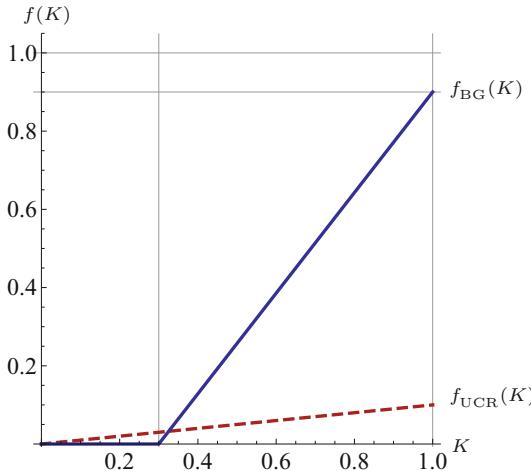


CMY→CMYK conversion (version 3)

In print production, special transfer functions are applied to tune the results. For example, the Adobe PostScript interpreter [135, p. 345] specifies an *undercolor-removal function* $f_{UCR}(K)$ for gradually reducing the CMY components and a separate *black-generation function* $f_{BG}(K)$ for controlling the amount of black. These functions are used in the form

$$\begin{pmatrix} C_3 \\ M_3 \\ Y_3 \\ K_3 \end{pmatrix} = \begin{pmatrix} C - f_{UCR}(K) \\ M - f_{UCR}(K) \\ Y - f_{UCR}(K) \\ f_{BG}(K) \end{pmatrix}, \quad (12.52)$$

where $K = \min(C, M, Y)$, as defined in Eqn. (12.49). The functions f_{UCR} and f_{BG} are usually nonlinear, and the resulting values



12.3 STATISTICS OF COLOR IMAGES

Fig. 12.21

Examples of *undercolor-removal function* f_{UCR} (Eqn. (12.53)) and *black generation function* f_{BG} (Eqn. (12.54)). The parameter settings are $s_K = 0.1$, $K_0 = 0.3$, and $K_{\max} = 0.9$.

C_3, M_3, Y_3, K_3 are scaled (typically by means of *clamping*) to the interval $[0, 1]$. The example shown in Fig. 12.20(b) was produced to approximate the results of Adobe Photoshop using the definitions

$$f_{UCR}(K) = s_K \cdot K, \quad (12.53)$$

$$f_{BG}(K) = \begin{cases} 0 & \text{for } K < K_0, \\ K_{\max} \cdot \frac{K - K_0}{1 - K_0} & \text{for } K \geq K_0, \end{cases} \quad (12.54)$$

where $s_K = 0.1$, $K_0 = 0.3$, and $K_{\max} = 0.9$ (see Fig. 12.21). With this definition, f_{UCR} reduces the CMY components by 10% of the K value (by Eqn. (12.52)), which mostly affects the dark areas of the image with high K values. The effect of the function f_{BG} (Eqn. (12.54)) is that for values of $K < K_0$ (i.e., in the light areas of the image) no black ink is added at all. In the interval $K = K_0, \dots, 1.0$, the black component is increased linearly up to the maximum value K_{\max} . The result in Fig. 12.20(b) is relatively close to the CMYK component values produced by Photoshop¹² in Fig. 12.20(c). It could be further improved by adjusting the function parameters s_K , K_0 , and K_{\max} (Eqn. (12.52)).

Even though the results of this last variant (3) for converting RGB to CMYK are better, it is only a gross approximation and still too imprecise for professional work. As we discuss in Chapter 14, technically correct color conversions need to be based on precise, “colorimetric” grounds.

12.3 Statistics of Color Images

12.3.1 How Many Different Colors are in an Image?

A minor but frequent task in the context of color images is to determine how many different colors are contained in a given image.

¹² Actually Adobe Photoshop does not convert directly from RGB to CMYK. Instead, it first converts to, and then from, the CIELAB color space (see Ch. 14, Sec. 14.1).

One way of doing this would be to create and fill a histogram array with one integer element for each color and subsequently count all histogram cells with values greater than zero. But since a 24-bit RGB color image potentially contains $2^{24} = 16,777,216$ colors, the resulting histogram array (with a size of 64 megabytes) would be larger than the image itself in most cases!

A simple solution to this problem is to *sort* the pixel values in the (1D) pixel array such that all identical colors are placed next to each other. The sorting order is of course completely irrelevant, and the number of contiguous color blocks in the sorted pixel vector corresponds to the number of different colors in the image. This number can be obtained by simply counting the transitions between neighboring color blocks, as shown in Prog. 12.10. Of course, we do not want to sort the original pixel array (which would destroy the image) but a copy of it, which can be obtained with Java's `clone()` method.¹³ Sorting of the 1D array in Prog. 12.10 is accomplished (in line 4) with the generic Java method `Arrays.sort()`, which is implemented very efficiently.

Prog. 12.10

Counting the colors contained in an RGB image. The method `countColors()` first creates a copy of the 1D RGB (int) pixel array (line 3), then sorts that array, and finally counts the transitions between contiguous blocks of identical colors.

```

1  int countColors (ColorProcessor cp) {
2      // duplicate the pixel array and sort it
3      int[] pixels = ((int[]) cp.getPixels()).clone();
4      Arrays.sort(pixels); // requires java.util.Arrays
5
6      int k = 1; // color count (image contains at least 1 color)
7      for (int i = 0; i < pixels.length-1; i++) {
8          if (pixels[i] != pixels[i + 1])
9              k = k + 1;
10     }
11     return k;
12 }
```

12.3.2 Color Histograms

We briefly touched on histograms of color images in Chapter 3, Sec. 3.5, where we only considered the 1D distributions of the image intensity and the individual color channels. For instance, the built-in ImageJ method `getHistogram()`, when applied to an object of type `ColorProcessor`, simply computes the intensity histogram of the corresponding gray values:

```

ColorProcessor cp;
int[] H = cp.getHistogram();
```

As an alternative, one could compute the individual intensity histograms of the three color channels, although (as discussed in Chapter 3, Sec. 3.5.2) these do not provide any information about the actual colors in this image. Similarly, of course, one could compute the distributions of the individual components of any other color space, such as HSV or CIELAB.

¹³ Java arrays implement the `Cloneable` interface.

A *full* histogram of an RGB image is 3D and, as noted earlier, consists of $256 \times 256 \times 256 = 2^{24}$ cells of type `int` (for 8-bit color components). Such a histogram is not only very large¹⁴ but also difficult to visualize.

12.4 EXERCISES

2D color histograms

A useful alternative to the full 3D RGB histogram are 2D histogram projections (Fig. 12.22). Depending on the axis of projection, we obtain 2D histograms with coordinates red-green (h_{RG}), red-blue (h_{RB}), or green-blue (h_{GB}), respectively, with the values

$$\begin{aligned} h_{RG}(r, g) &:= \text{number of pixels with } I(u, v) = (r, g, *), \\ h_{RB}(r, b) &:= \text{number of pixels with } I(u, v) = (r, *, b), \\ h_{GB}(g, b) &:= \text{number of pixels with } I(u, v) = (*, g, b), \end{aligned} \quad (12.55)$$

where $*$ denotes an arbitrary component value. The result is, independent of the original image size, a set of 2D histograms of size 256×256 (for 8-bit RGB components), which can easily be visualized as images. Note that it is not necessary to obtain the full RGB histogram in order to compute the combined 2D histograms (see Prog. 12.11).

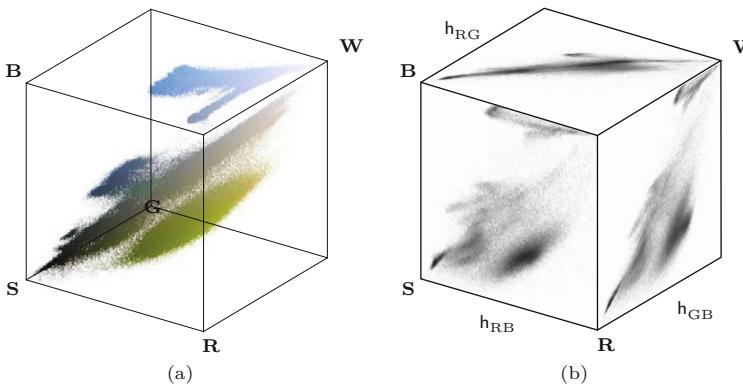


Fig. 12.22
2D RGB histogram projections. 3D RGB cube illustrating an image's color distribution (a). The color points indicate the corresponding pixel colors and not the color frequency. The combined histograms for red-green (h_{RG}), red-blue (h_{RB}), and green-blue (h_{GB}) are 2D projections of the 3D histogram. The corresponding image is shown in Fig. 12.9(a).

As the examples in Fig. 12.23 show, the combined color histograms do, to a certain extent, express the color characteristics of an image. They are therefore useful, for example, to identify the coarse type of the depicted scene or to estimate the similarity between images (see also Exercise 12.8).

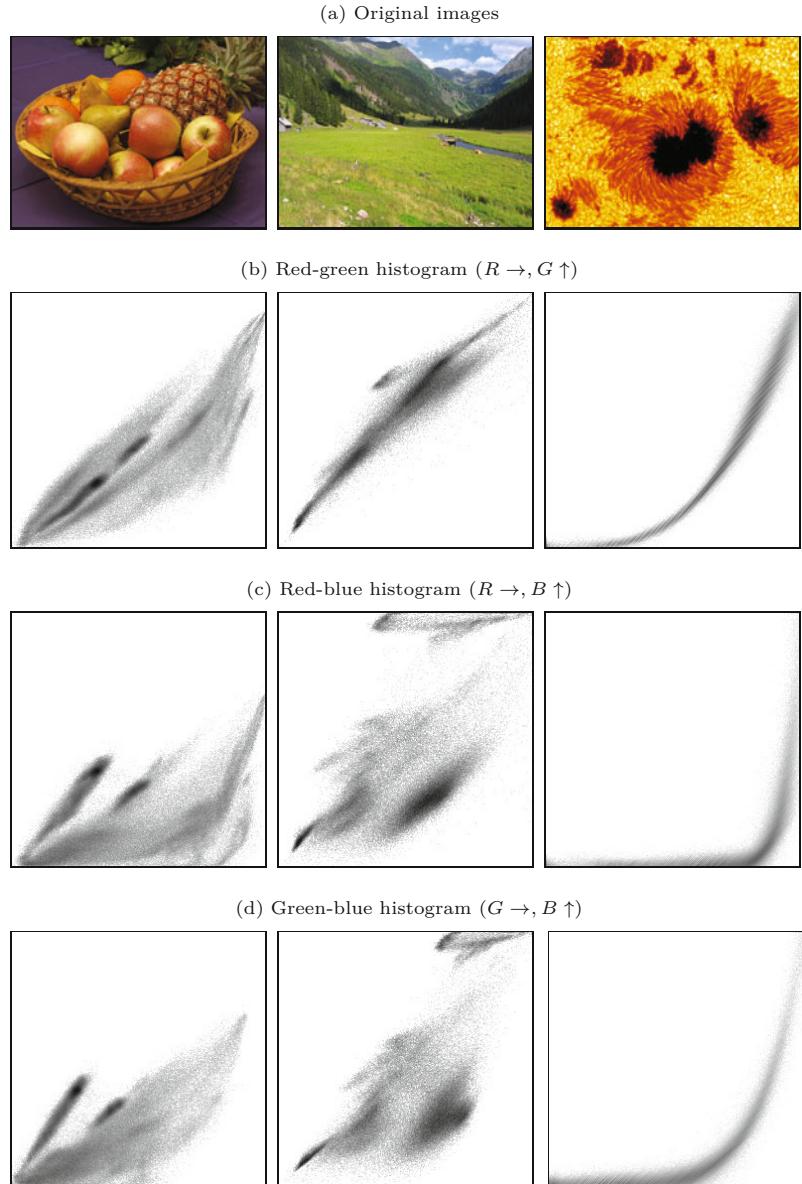
12.4 Exercises

Exercise 12.1. Create an ImageJ plugin that rotates the individual components of an RGB color image; that is, $R \rightarrow G \rightarrow B \rightarrow R$.

Exercise 12.2. Pseudocolors are sometimes used for displaying grayscale images (i.e., for viewing medical images with high dynamic

¹⁴ It may seem a paradox that, although the RGB histogram is usually much larger than the image itself, the histogram is not sufficient in general to reconstruct the original image.

Fig. 12.23
Combined color histogram examples. For better viewing, the images are inverted (dark regions indicate high frequencies) and the gray value corresponds to the logarithm of the histogram entries (scaled to the maximum entries).



range). Create an ImageJ plugin for converting 8-bit grayscale images to an indexed image with 256 colors, simulating the hues of glowing iron (from dark red to yellow and white).

Exercise 12.3. Create an ImageJ plugin that shows the color table of an 8-bit indexed image as a new image with 16×16 rectangular color fields. Mark all unused color table entries in a suitable way. Look at Prog. 12.3 as a starting point.

Exercise 12.4. Show that a “desaturated” RGB pixel produced in the form $(r, g, b) \rightarrow (y, y, y)$, where y is the equivalent luminance value (see Eqn. (12.11)), has the luminance y as well.

```

1 int[][] get2dHistogram
2     (ColorProcessor cp, int c1, int c2) {
3 // c1, c2: component index R = 0, G = 1, B = 2
4
5     int[] RGB = new int[3];
6     int[][] h = new int[256][256]; // 2D histogram h[c1][c2]
7
8     for (int v = 0; v < cp.getHeight(); v++) {
9         for (int u = 0; u < cp.getWidth(); u++) {
10            cp.getPixel(u, v, RGB);
11            int i1 = RGB[c1];
12            int i2 = RGB[c2];
13            // increment the associated histogram cell
14            h[i1][i2]++;
15        }
16    }
17    return h;
18 }
```

12.4 EXERCISES

Prog. 12.11

Java method `get2dHistogram()` for computing a combined 2D color histogram. The color components (histogram axes) are specified by the parameters `c1` and `c2`. The color distribution `H` is returned as a 2D int array. The method is defined in class `ColorStatistics` (Prog. 12.10).

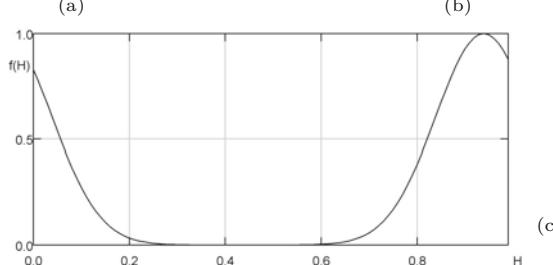
Exercise 12.5. Extend the ImageJ plugin for desaturating color images in Prog. 12.5 such that the image is only modified inside the user-selected region of interest (ROI).

Exercise 12.6. Write an ImageJ plugin that *selectively desaturates* an RGB image, preserving colors with a hue close to a given *reference color* $\mathbf{c}_{\text{ref}} = (R_{\text{ref}}, G_{\text{ref}}, B_{\text{ref}})$, with (HSV) hue H_{ref} (see the example in Fig. 12.24). Transform the image to HSV and modify the colors (cf. Eqn. (12.34)) in the form

$$\begin{pmatrix} H_{\text{desat}} \\ S_{\text{desat}} \\ V_{\text{desat}} \end{pmatrix} = \begin{pmatrix} H \\ f(H) \cdot S \\ V \end{pmatrix}, \quad (12.56)$$



Fig. 12.24
Selective desaturation example. Original image with selected reference color $\mathbf{c}_{\text{ref}} = (250, 92, 150)$ (a), de-saturated image (b). Gaussian saturation function $f(H)$ (see Eqn. (12.58)) with reference hue $H_{\text{ref}} = 0.9388$ and $\sigma = 0.1$ (c).



where $f(H)$ is a smooth saturation function, for example, a Gaussian function of the form

$$f(H) = e^{-\frac{(H-H_{\text{ref}})^2}{2\cdot\sigma^2}} = g_\sigma(H-H_{\text{ref}}), \quad (12.57)$$

with center H_{ref} and variance σ^2 (see Fig. 12.24(c)). Recall that the H component is circular in $[0, 1)$. To obtain a continuous and periodic saturation function we note that $H' = H - H_{\text{ref}}$ is in the range $[-1, 1]$ and reformulate $f(H)$ as

$$f(H) = \begin{cases} g_\sigma(H') & \text{for } -0.5 \leq H' \leq 0.5, \\ g_\sigma(H'+1) & \text{for } H' < -0.5, \\ g_\sigma(H'-1) & \text{for } H' > 0.5. \end{cases} \quad (12.58)$$

Verify the values of the function $f(H)$, check in particular that it is 1 for the reference color! What would be a good (synthetic) color image for validating the saturation function? Use ImageJ's color picker (pipette) tool to specify the reference color c_{ref} interactively.¹⁵

Exercise 12.7. Calculate (analogous to Eqns. (12.46)–(12.47)) the complete transformation matrices for converting from (linear) RGB colors to YC_bC_r for the ITU-BT.709 (HDTV) standard with the coefficients $w_R = 0.2126$, $w_B = 0.0722$ and $w_G = 0.7152$.

Exercise 12.8. Determining the similarity between images of different sizes is a frequent problem (e.g., in the context of image data bases). Color statistics are commonly used for this purpose because they facilitate a coarse classification of images, such as landscape images, portraits, etc. However, 2D color histograms (as described in Sec. 12.3.2) are usually too large and thus cumbersome to use for this purpose. A simple idea could be to split the 2D histograms or even the full RGB histogram into K regions (*bins*) and to combine the corresponding entries into a K -dimensional feature vector, which could be used for a coarse comparison. Develop a concept for such a procedure, and also discuss the possible problems.

Exercise 12.9. Write a program (plugin) that generates a sequence of colors with constant hue and saturation but different brightness (value) in HSV space. Transform these colors to RGB and draw them into a new image. Verify (visually) if the hue really remains constant.

Exercise 12.10. When applying any type of filter in HSV or HLS color space one must keep in mind that the *hue* component H is circular in $[0, 1)$ and thus shows a discontinuity at the $1 \rightarrow 0$ ($360 \rightarrow 0^\circ$) transition. For example, a linear filter would not take into account that $H = 0.0$ and $H = 1.0$ refer to the same hue (red) and thus cannot be applied directly to the H component. One solution is to filter the *cosine* and *sine* values of the H component (which really is an angle) instead, and composing the filtered hue array from the filtered cos / sin values (see Ch. 15, Sec. 15.1.3 for details). Based on this idea, implement a variable-sized linear Gaussian filter (see Ch. 5, Sec. 5.2.7) for the HSV color space.

¹⁵ The current color pick is returned by the ImageJ method `Toolbar.getForegroundColor()`.

Color Quantization

The task of color quantization is to select and assign a limited set of colors for representing a given color image with maximum fidelity. Assume, for example, that a graphic artist has created an illustration with beautiful shades of color, for which he applied 150 different crayons. His editor likes the result but, for some technical reason, instructs the artist to draw the picture again, this time using only 10 different crayons. The artist now faces the problem of color quantization—his task is to select a “palette” of the 10 best suited from his 150 crayons and then choose the most similar color to redraw each stroke of his original picture.

In the general case, the original image I contains a set of m different colors $\mathcal{C} = \{\mathbf{C}_1, \mathbf{C}_2, \dots, \mathbf{C}_m\}$, where m could be only a few or several thousand, but at most 2^{24} for a 3×8 -bit color image. The goal is to replace the original colors by a (usually much smaller) set of colors $\mathcal{C}' = \{\mathbf{C}'_1, \mathbf{C}'_2, \dots, \mathbf{C}'_n\}$, with $n < m$. The difficulty lies in the proper choice of the reduced color palette \mathcal{C}' such that damage to the resulting image is minimized.

In practice, this problem is encountered, for example, when converting from full-color images to images with lower pixel depth or to index (“palette”) images, such as the conversion from 24-bit TIFF to 8-bit GIF images with only 256 (or fewer) colors. Until a few years ago, a similar problem had to be solved for displaying full-color images on computer screens because the available display memory was often limited to only 8 bits. Today, even the cheapest display hardware has at least 24-bit depth and therefore this particular need for (fast) color quantization no longer exists.

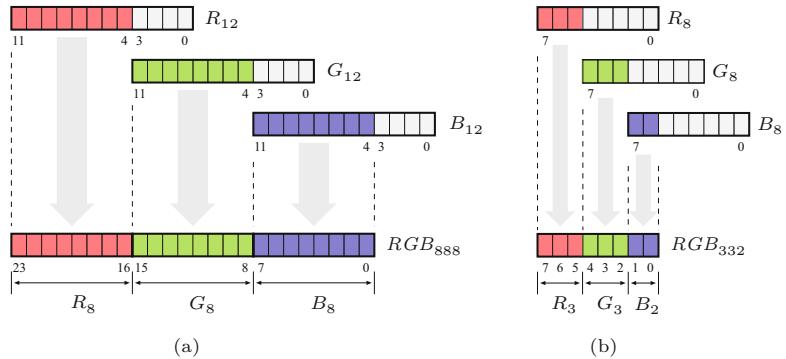
13.1 Scalar Color Quantization

Scalar (or *uniform*) quantization is a simple and fast process that is independent of the image content. Each of the original color components c_i (e.g., R_i, G_i, B_i) in the range $[0, \dots, m-1]$ is independently converted to the new range $[0, \dots, n-1]$, in the simplest case by a

13 COLOR QUANTIZATION

Fig. 13.1

Scalar quantization of color components by truncating lower bits. Quantization of 3×12 -bit to 3×8 -bit colors (a). Quantization of 3×8 -bit to 3:3:2-packed 8-bit colors (b). The Java code segment in Prog. 13.1 shows the corresponding sequence of bit operations.



linear quantization in the form

$$c'_i \leftarrow \left\lfloor c_i \cdot \frac{n}{m} \right\rfloor \quad (13.1)$$

for all color components c_i . A typical example would be the conversion of a color image with 3×12 -bit components ($m = 4096$) to an RGB image with 3×8 -bit components ($n = 256$). In this case, each original component value is multiplied by $n/m = 256/4096 = 1/16 = 2^{-4}$ and subsequently truncated, which is equivalent to an integer division by 16 or simply ignoring the lower 4 bits of the corresponding binary values (see Fig. 13.1(a)). m and n are usually the same for all color components but not always.

An extreme (today rarely used) approach is to quantize 3×8 -color vectors to single-byte (8-bit) colors, where 3 bits are used for red and green and only 2 bits for blue, as shown in Prog. 13.1(b). In this case, $m = 256$ for all color components, $n_{\text{red}} = n_{\text{green}} = 8$, and $m_{\text{blue}} = 4$.

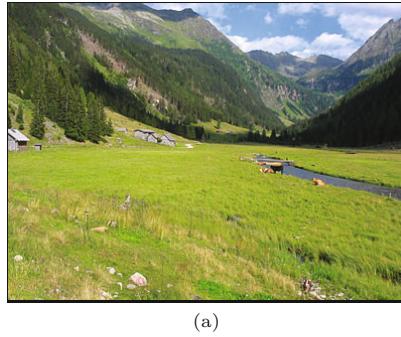
Prog. 13.1
Quantization of a 3×8 -bit RGB color pixel to 8 bits by 3:3:2 packing.

```

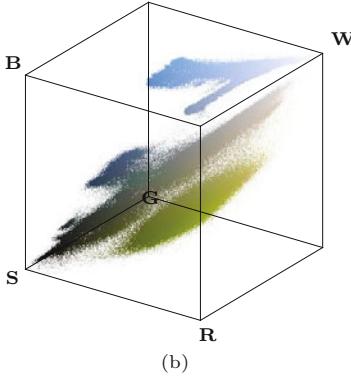
1 ColorProcessor cp = (ColorProcessor) ip;
2 int C = cp.getPixel(u, v);
3 int R = (C & 0x00ff0000) >> 16;
4 int G = (C & 0x0000ff00) >> 8;
5 int B = (C & 0x000000ff);
6 // 3:3:2 uniform color quantization
7 byte RGB =
8     ((byte) (((R & 0xE0) | ((G & 0xE0) >> 3) | ((B & 0xC0) >> 6)));

```

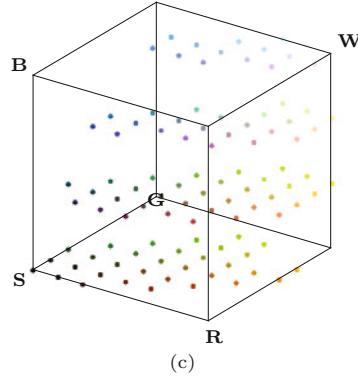
Unlike the techniques described in the following, scalar quantization does not take into account the distribution of colors in the original image. Scalar quantization is an optimal solution only if the image colors are *uniformly* distributed within the RGB cube. However, the typical color distribution in natural images is anything but uniform, with some regions of the color space being densely populated and many colors entirely missing. In this case, scalar quantization is not optimal because the interesting colors may not be sampled with sufficient density while at the same time colors are represented that do not appear in the image at all.



(a)



(b)



(c)

13.2 VECTOR QUANTIZATION

Fig. 13.2

Color distribution after a scalar 3:3:2 quantization. Original color image (a). Distribution of the original 226,321 colors (b) and the remaining $8 \times 8 \times 4 = 256$ colors after 3:3:2 quantization (c) in the RGB color cube.

13.2 Vector Quantization

Vector quantization does not treat the individual color components separately as does scalar quantization, but each color vector $\mathbf{C}_i = (r_i, g_i, b_i)$ or pixel in the image is treated as a single entity. Starting from a set of original color tuples $\mathcal{C} = \{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_m\}$, the task of vector quantization is

- to find a set of n representative color vectors $\mathcal{C}' = \{\mathbf{c}'_1, \mathbf{c}'_2, \dots, \mathbf{c}'_n\}$ and
- to replace each original color \mathbf{C}_i by one of the new color vectors $\mathbf{C}'_j \in \mathcal{C}'$,

where n is usually predetermined ($n < m$) and the resulting deviation from the original image shall be minimal. This is a combinatorial optimization problem in a rather large search space, which usually makes it impossible to determine a global optimum in adequate time. Thus all of the following methods only compute a “local” optimum at best.

13.2.1 Popularity Algorithm

The popularity algorithm¹ [104] selects the n most frequent colors in the image as the representative set of color vectors \mathcal{C}' . Being very easy to implement, this procedure is quite popular. The method described in Sec. 12.3.1, based on sorting the image pixels, can be used to determine the n most frequent image colors. Each original

¹ Sometimes also called the “popularity” algorithm.

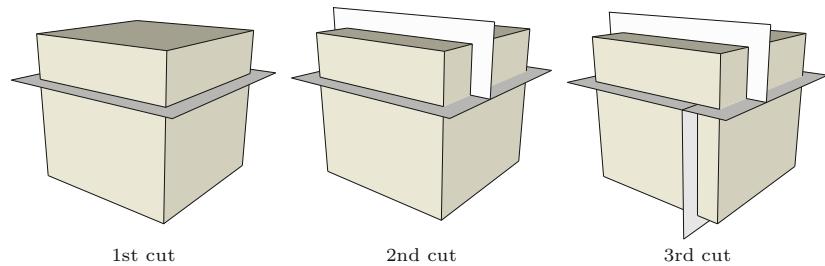
pixel \mathbf{C}_i is then replaced by the closest representative color vector in C' ; that is, the quantized color vector with the smallest distance in the 3D color space.

The algorithm performs sufficiently only as long as the original image colors are not widely scattered through the color space. Some improvement is possible by grouping similar colors into larger cells first (by scalar quantization). However, a less frequent (but possibly important) color may get lost whenever it is not sufficiently similar to any of the n most frequent colors.

13.2.2 Median-Cut Algorithm

The median-cut algorithm [104] is considered a classical method for color quantization that is implemented in many applications (including ImageJ). As in the populosity method, a color histogram is first computed for the original image, traditionally with a reduced number of histogram cells (such as $32 \times 32 \times 32$) for efficiency reasons.² The initial histogram volume is then recursively split into smaller boxes until the desired number of representative colors is reached. In each recursive step, the color box representing the largest number of pixels is selected for splitting. A box is always split across the longest of its three axes at the median point, such that half of the contained pixels remain in each of the resulting subboxes (Fig. 13.3).

Fig. 13.3
Median-cut algorithm. The RGB color space is recursively split into smaller cubes along one of the color axes.



The result of this recursive splitting process is a partitioning of the color space into a set of disjoint boxes, with each box ideally containing the same number of image pixels. In the last step, a representative color vector (e.g., the mean vector of the contained colors) is computed for each color cube, and all the image pixels it contains are replaced by that color.

The advantage of this method is that color regions of high pixel density are split into many smaller cells, thus reducing the overall quantization error. In color regions of low density, however, relatively large cubes and thus large color deviations may occur for individual pixels.

The median-cut method is described in detail in Algorithms 13.1–13.3 and a corresponding Java implementation can be found in the source code section of this book’s website (see Sec. 13.2.5).

² This corresponds to a scalar prequantization on the color components, which leads to additional quantization errors and thus produces suboptimal results. This step seems unnecessary on modern computers and should be avoided.

```

1: MedianCut( $\mathbf{I}, K_{\max}$ )
    $\mathbf{I}$ : color image,  $K_{\max}$ : max. number of quantized colors
   Returns a new quantized image with at most  $K_{\max}$  colors.
2:  $\mathcal{C}_q \leftarrow \text{FindRepresentativeColors}(\mathbf{I}, K_{\max})$ 
3: return QuantizeImage( $\mathbf{I}, \mathcal{C}_q$ ) ▷ see Alg. 13.3


---


4: FindRepresentativeColors( $\mathbf{I}, K_{\max}$ )
   Returns a set of up to  $K_{\max}$  representative colors for the image  $\mathbf{I}$ .
5: Let  $\mathcal{C} = \{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_K\}$  be the set of distinct colors in  $\mathbf{I}$ . Each of
   the  $K$  color elements in  $\mathcal{C}$  is a tuple  $\mathbf{c}_i = \langle \text{red}_i, \text{grn}_i, \text{blu}_i, \text{cnt}_i \rangle$ 
   consisting of the RGB color components ( $\text{red}$ ,  $\text{grn}$ ,  $\text{blu}$ ) and
   the number of pixels ( $\text{cnt}$ ) in  $\mathbf{I}$  with that particular color.
6: if  $|\mathcal{C}| \leq K_{\max}$  then
7:   return  $\mathcal{C}$ 
8: else
   Create a color box  $\mathbf{b}_0$  at level 0 that contains all image colors
    $\mathcal{C}$  and make it the initial element in the set of color boxes  $\mathcal{B}$ :
9:  $\mathbf{b}_0 \leftarrow \text{CreateColorBox}(\mathcal{C}, 0)$  ▷ see Alg. 13.2
10:  $\mathcal{B} \leftarrow \{\mathbf{b}_0\}$  ▷ initial set of color boxes
11:  $k \leftarrow 1$ 
12:  $done \leftarrow \text{false}$ 
13: while  $k < N_{\max}$  and  $\neg done$  do
14:    $\mathbf{b} \leftarrow \text{FindBoxToSplit}(\mathcal{B})$  ▷ see Alg. 13.2
15:   if  $\mathbf{b} \neq \text{nil}$  then
16:      $(\mathbf{b}_1, \mathbf{b}_2) \leftarrow \text{SplitBox}(\mathbf{b})$  ▷ see Alg. 13.2
17:      $\mathcal{B} \leftarrow \mathcal{B} - \{\mathbf{b}\}$  ▷ remove  $\mathbf{b}$  from  $\mathcal{B}$ 
18:      $\mathcal{B} \leftarrow \mathcal{B} \cup \{\mathbf{b}_1, \mathbf{b}_2\}$  ▷ insert  $\mathbf{b}_1, \mathbf{b}_2$  into  $\mathcal{B}$ 
19:      $k \leftarrow k + 1$ 
20:   else ▷ no more boxes to split
21:      $done \leftarrow \text{true}$ 
22: Collect the average colors of all color boxes in  $\mathcal{B}$ :
23:  $\mathcal{C}_q \leftarrow \{\text{AverageColor}(\mathbf{b}_j) \mid \mathbf{b}_j \in \mathcal{B}\}$  ▷ see Alg. 13.3
return  $\mathcal{C}_q$ 

```

13.2 VECTOR QUANTIZATION

Alg. 13.1

Median-cut color quantization (part 1). The input image \mathbf{I} is quantized to up to K_{\max} representative colors and a new, quantized image is returned. The main work is done in procedure `FindRepresentativeColors()`, which iteratively partitions the color space into increasingly smaller boxes. It returns a set of representative colors (\mathcal{C}_q) that are subsequently used by procedure `QuantizeImage()` to quantize the original image \mathbf{I} . Note that (unlike in most common implementations) no prequantization is applied to the original image colors.

13.2.3 Octree Algorithm

Similar to the median-cut algorithm, this method is also based on partitioning the 3D color space into cells of varying size. The octree algorithm [82] utilizes a hierarchical structure, where each cube in color space may contain eight subcubes. This partitioning is represented by a tree structure (octree) with a cube at each node that may again link to up to eight further nodes. Thus each node corresponds to a subrange of the color space that reduces to a single color point at a certain tree depth d (e.g., $d = 8$ for a 3×8 -bit RGB color image).

When an image is processed, the corresponding quantization tree, which is initially empty, is created dynamically by evaluating all pixels in a sequence. Each pixel's color tuple is inserted into the quantization tree, while at the same time the number of nodes is limited to a predefined value K (typically 256). When a new color tuple \mathbf{C}_i is inserted and the tree does not contain this color, one of the following situations can occur:

13 COLOR QUANTIZATION

Alg. 13.2
Median-cut color quantization (part 2).

1: **CreateColorBox**(\mathcal{C}, m)

Creates and returns a new color box containing the colors \mathcal{C} and level m . A color box b is a tuple $\langle \text{colors}, \text{level}, \text{rmin}, \text{rmax}, \text{gmin}, \text{gmax}, \text{bmin}, \text{bmax} \rangle$, where colors is the set of image colors represented by the box, level denotes the split-level, and $\text{rmin}, \dots, \text{bmax}$ describe the color boundaries of the box in RGB space.

Find the RGB extrema of all colors in \mathcal{C} :

```

2:    $r_{\min}, g_{\min}, b_{\min} \leftarrow +\infty$ 
3:    $r_{\max}, g_{\max}, b_{\max} \leftarrow -\infty$ 
4:   for all  $c \in \mathcal{C}$  do
       $r_{\min} \leftarrow \min(r_{\min}, \text{red}(c))$ 
       $r_{\max} \leftarrow \max(r_{\max}, \text{red}(c))$ 
5:    $g_{\min} \leftarrow \min(g_{\min}, \text{grn}(c))$ 
       $g_{\max} \leftarrow \max(g_{\max}, \text{grn}(c))$ 
       $b_{\min} \leftarrow \min(b_{\min}, \text{blu}(c))$ 
       $b_{\max} \leftarrow \max(b_{\max}, \text{blu}(c))$ 
6:    $b \leftarrow \langle \mathcal{C}, m, r_{\min}, r_{\max}, g_{\min}, g_{\max}, b_{\min}, b_{\max} \rangle$ 
7:   return  $b$ 
```

8: **FindBoxToSplit**(\mathcal{B})

Searches the set of boxes \mathcal{B} for a box to split and returns this box, or **nil** if no splittable box can be found.

Find the set of color boxes that can be split (i.e., contain at least 2 different colors):

```

9:    $\mathcal{B}_s \leftarrow \{b \mid b \in \mathcal{B} \wedge |\text{colors}(b)| \geq 2\}$ 
10:  if  $\mathcal{B}_s = \{\}$  then                                 $\triangleright$  no splittable box was found
11:  return nil
12:  else
13:    Select a box  $b_x$  from  $\mathcal{B}_s$ , such that  $\text{level}(b_x)$  is a minimum:
14:     $b_x \leftarrow \underset{b \in \mathcal{B}_s}{\text{argmin}}(\text{level}(b))$ 
15:    return  $b_x$ 
```

15: **SplitBox**(b)

Splits the color box b at the median plane perpendicular to its longest dimension and returns a pair of new color boxes.

```

16:   $m \leftarrow \text{level}(b)$ 
17:   $d \leftarrow \text{FindMaxBoxDimension}(b)$                  $\triangleright$  see Alg. 13.3
18:   $\mathcal{C} \leftarrow \text{colors}(b)$                            $\triangleright$  the set of colors in box  $b$ 
19:  From all colors in  $\mathcal{C}$  determine the median of the color distribution along dimension  $d$  and split  $\mathcal{C}$  into  $\mathcal{C}_1, \mathcal{C}_2$ :
    
$$\mathcal{C}_1 \leftarrow \begin{cases} \{c \in \mathcal{C} \mid \text{red}(c) \leq \underset{c \in \mathcal{C}}{\text{median}}(\text{red}(c))\} & \text{for } d = \text{Red} \\ \{c \in \mathcal{C} \mid \text{grn}(c) \leq \underset{c \in \mathcal{C}}{\text{median}}(\text{grn}(c))\} & \text{for } d = \text{Green} \\ \{c \in \mathcal{C} \mid \text{blu}(c) \leq \underset{c \in \mathcal{C}}{\text{median}}(\text{blu}(c))\} & \text{for } d = \text{Blue} \end{cases}$$

20:   $\mathcal{C}_2 \leftarrow \mathcal{C} \setminus \mathcal{C}_1$ 
21:   $b_1 \leftarrow \text{CreateColorBox}(\mathcal{C}_1, m + 1)$ 
22:   $b_2 \leftarrow \text{CreateColorBox}(\mathcal{C}_2, m + 1)$ 
23:  return  $(b_1, b_2)$ 
```

1. If the number of nodes is less than K and no suitable node for the color \mathbf{c}_i exists already, then a new node is created for \mathbf{C}_i .
2. Otherwise (i.e., if the number of nodes is K), the existing nodes at the maximum tree depth (which represent similar colors) are merged into a common node.

```

1: AverageColor( $b$ )
   Returns the average color  $c_{\text{avg}}$  for the pixels represented by the
   color box  $b$ .
2:  $\mathcal{C} \leftarrow \text{colors}(b)$                                  $\triangleright$  the set of colors in box  $b$ 
3:  $n \leftarrow 0$ 
4:  $\Sigma_r \leftarrow 0, \Sigma_g \leftarrow 0, \Sigma_b \leftarrow 0$ 
5: for all  $c \in \mathcal{C}$  do
6:    $k \leftarrow \text{cnt}(c)$ 
7:    $n \leftarrow n + k$ 
8:    $\Sigma_r \leftarrow \Sigma_r + k \cdot \text{red}(c)$ 
9:    $\Sigma_g \leftarrow \Sigma_g + k \cdot \text{grn}(c)$ 
10:   $\Sigma_b \leftarrow \Sigma_b + k \cdot \text{blu}(c)$ 
11:   $\bar{c} \leftarrow (\Sigma_r/n, \Sigma_g/n, \Sigma_b/n)$ 
12:  return  $\bar{c}$ 

13: FindMaxBoxDimension( $b$ )
   Returns the largest dimension of the color box  $b$  (Red, Green, or
   Blue).
14:  $d_r = \text{rmax}(b) - \text{rmin}(b)$ 
15:  $d_g = \text{gmax}(b) - \text{gmin}(b)$ 
16:  $d_b = \text{bmax}(b) - \text{bmin}(b)$ 
17:  $d_{\text{max}} = \max(d_r, d_g, d_b)$ 
18: if  $d_{\text{max}} = d_r$  then
19:   return Red.
20: else if  $d_{\text{max}} = d_g$  then
21:   return Green
22: else
23:   return Blue

24: QuantizeImage( $I, \mathcal{C}_q$ )
   Returns a new image with color pixels from  $I$  replaced by their
   closest representative colors in  $\mathcal{C}_q$ .
25:  $I' \leftarrow \text{duplicate}(I)$                                  $\triangleright$  create a new image
26: for all image coordinates  $(u, v)$  do
   Find the quantization color in  $\mathcal{C}_q$  that is “closest” to the cur-
   rent pixel color (e.g., using the Euclidean distance in RGB
   space):
27:    $I'(u, v) \leftarrow \underset{c \in \mathcal{C}_q}{\text{argmin}} \|I(u, v) - c\|$ 
28: return  $I'$ 

```

13.2 VECTOR QUANTIZATION

Alg. 13.3

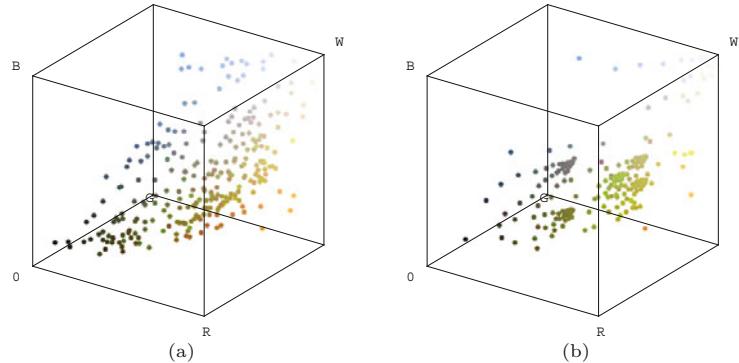
Median-cut color quantization
(part 3).

A key advantage of the iterative octree method is that the number of color nodes remains limited to K in any step and thus the amount of required storage is small. The final replacement of the image pixels by the quantized color vectors can also be performed easily and efficiently with the octree structure because only up to eight comparisons (one at each tree layer) are necessary to locate the best-matching color for each pixel.

Figure 13.4 shows the resulting color distributions in RGB space after applying the median-cut and octree algorithms. In both cases, the original image (Fig. 13.2(a)) is quantized to 256 colors. Notice in particular the dense placement of quantized colors in certain regions of the green hues. For both algorithms and the (scalar) 3:3:2 quan-

Fig. 13.4

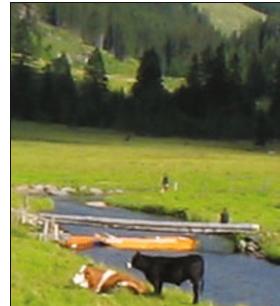
Color distribution after application of the median-cut (a) and octree (b) algorithms. In both cases, the set of 226,321 colors in the original image (Fig. 13.2(a)) was reduced to 256 representative colors.



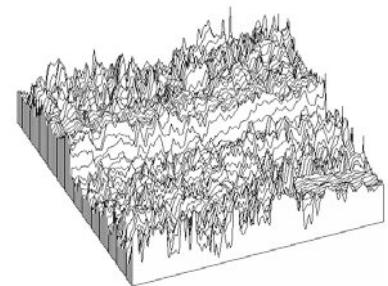
tization, the resulting distances between the original pixels and the quantized colors are shown in Fig. 13.5. The greatest error naturally results from 3:3:2 quantization, because this method does not consider the contents of the image at all. Compared with the median-cut method, the overall error for the octree algorithm is smaller, although the latter creates several large deviations, particularly inside the colored foreground regions and the forest region in the background. In general, however, the octree algorithm does not offer significant advantages in terms of the resulting image quality over the simpler median-cut algorithm.

Fig. 13.5

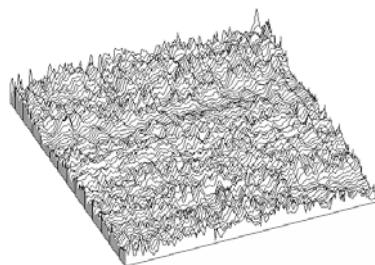
Quantization errors. Original image (a), distance between original and quantized color pixels for scalar 3:3:2 quantization (b), median-cut (c), and octree (d) algorithms.



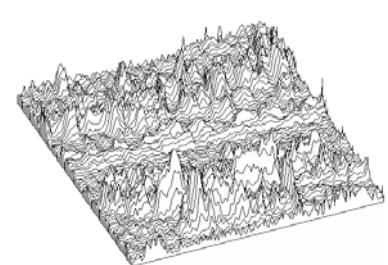
(a) Detail



(b) 3:3:2



(c) Median-cut



(d) Octree

13.2.4 Other Methods for Vector Quantization

A suitable set of representative color vectors can usually be determined without inspecting all pixels in the original image. It is often

sufficient to use only 10% of randomly selected pixels to obtain a high probability that none of the important colors is lost.

13.3 EXERCISES

In addition to the color quantization methods described already, several other procedures and refined algorithms have been proposed. This includes statistical and clustering methods, such as the classical *k-means* algorithm, but also the use of neural networks and genetic algorithms. A good overview can be found in [219].

13.2.5 Java Implementation

The Java implementation³ of the algorithms described in this chapter consists of a common interface `ColorQuantizer` and the concrete classes

- `MedianCutQuantizer`,
- `OctreeQuantizer`.

Program 13.2 shows a complete ImageJ plugin that employs the class `MedianCutQuantizer` for quantizing an RGB full-color image to an indexed image. The choice of data structures for the representation of color sets and the implementation of the associated set operations are essential to achieve good performance. The data structures used in this implementation are illustrated in Fig. 13.6.

Initially, the set of all colors contained in the original image (`ip` of type `ColorProcessor`) is computed by `new ColorHistogram()`. The result is an array `imageColors` of size K . Each cell of `imageColors` refers to a `colorNode` object (c_i) that holds the associated color (`red`, `green`, `blue`) and its frequency (`cnt`) in the image. Each `colorBox` object (corresponding to a color box b in Alg. 13.1) selects a contiguous range of image colors, bounded by the indices `lower` and `upper`. The ranges of elements in `imageColors`, indexed by different `colorBox` objects, never overlap. Each element in `imageColors` is contained in exactly one `colorBox`; that is, the color boxes held in `colorSet` (\mathcal{B} in Alg. 13.1) form a partitioning of `imageColors` (`colorSet` is implemented as a list of `ColorBox` objects). To split a particular `colorBox` along a color dimension $d = \text{Red}, \text{Green}, \text{or Blue}$, the corresponding subrange of elements in `imageColors` is *sorted* with the property `red`, `green`, or `blue`, respectively, as the sorting key. In Java, this is quite easy to implement using the standard `Arrays.sort()` method and a dedicated `Comparator` object for each color dimension. Finally, the method `quantize()` replaces each pixel in `ip` by the closest color in `colorSet`.

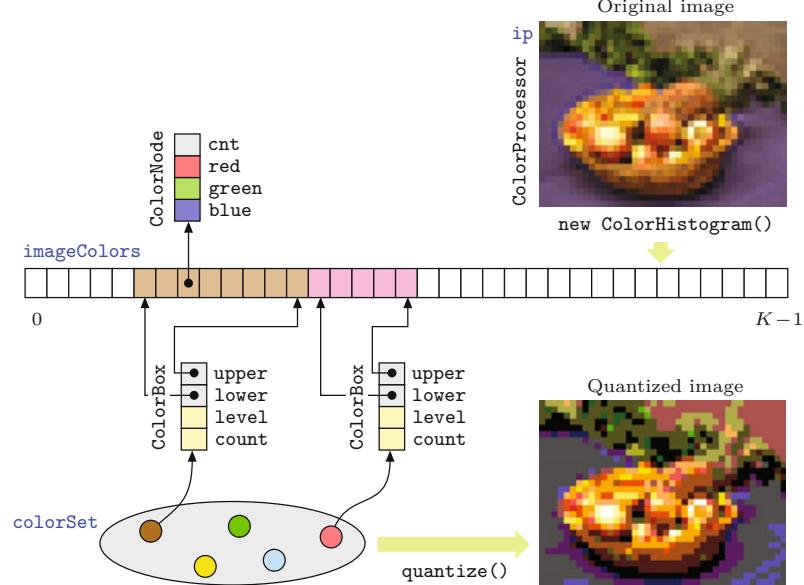
13.3 Exercises

Exercise 13.1. Simplify the 3:3:2 quantization given in Prog. 13.1 such that only a single bit mask/shift step is performed for each color component.

³ Package `imagingbook.pub.color.quantize`.

Fig. 13.6

Data structures used in the implementation of the median-cut quantization algorithm (class `MedianCutQuantizer`).



Exercise 13.2. The median-cut algorithm for color quantization (Sec. 13.2.2) is implemented in the *Independent JPEG Group's⁴ libjpeg* open source software with the following modification: the choice of the cube to be split next depends alternately on (a) the number of contained image pixels and (b) the cube's geometric volume. Consider the possible motives and discuss examples where this approach may offer an improvement over the original algorithm.

Exercise 13.3. The *signal-to-noise ratio* (SNR) is a common measure for quantifying the loss of image quality introduced by color quantization. It is defined as the ratio between the average *signal energy* P_{signal} and the average *noise energy* P_{noise} . For example, given an original color image \mathbf{I} and the associated quantized image \mathbf{I}' , this ratio could be calculated as

$$\text{SNR}(\mathbf{I}, \mathbf{I}') = \frac{P_{\text{signal}}}{P_{\text{noise}}} = \frac{\sum_{u=0}^{M-1} \sum_{v=0}^{N-1} \|\mathbf{I}(u, v)\|^2}{\sum_{u=0}^{M-1} \sum_{v=0}^{N-1} \|\mathbf{I}(u, v) - \mathbf{I}'(u, v)\|^2}. \quad (13.2)$$

Thus all deviations between the original and the quantized image are considered “noise”. The signal-to-noise ratio is usually specified on a logarithmic scale with the unit *decibel* (dB), that is,

$$\text{SNR}_{\log}(\mathbf{I}, \mathbf{I}') = 10 \cdot \log_{10}(\text{SNR}(\mathbf{I}, \mathbf{I}')) \text{ [dB]}. \quad (13.3)$$

Implement the calculation of the SNR, as defined in Eqns. (13.2)–(13.3), for color images and compare the results for the median-cut and the octree algorithms for the same number of target colors.

```

1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ByteProcessor;
4 import ij.process.ColorProcessor;
5 import ij.process.ImageProcessor;
6 import imagingbook.pub.color.quantize.ColorQuantizer;
7 import imagingbook.pub.color.quantize.MedianCutQuantizer;
8
9 public class Median_Cut_Quantization implements
10    PlugInFilter {
11    static int NCOLORS = 32;
12
13    public int setup(String arg, ImagePlus imp) {
14        return DOES_RGB + NO_CHANGES;
15    }
16
17    public void run(ImageProcessor ip) {
18        ColorProcessor cp = ip.convertToColorProcessor();
19        int w = ip.getWidth();
20        int h = ip.getHeight();
21
22        // create a quantizer:
23        ColorQuantizer q =
24            new MedianCutQuantizer(cp, NCOLORS);
25
26        // quantize cp to an indexed image:
27        ByteProcessor idxIp = q.quantize(cp);
28        (new ImagePlus("Quantized Index Image", idxIp)).show();
29
30        // quantize cp to an RGB image:
31        int[] rgbPix = q.quantize((int[]) cp.getPixels());
32        ImageProcessor rgbiIp =
33            new ColorProcessor(w, h, rgbPix);
34        (new ImagePlus("Quantized RGB Image", rgbiIp)).show();
35    }
36 }
```

13.3 EXERCISES

Prog. 13.2

Color quantization by the median-cut method (ImageJ plugin). This example uses the class `MedianCutQuantizer` to quantize the original full-color RGB image into (a) an indexed color image (of type `ByteProcessor`) and (b) another RGB image (of type `ColorProcessor`). Both images are finally displayed.

Colorimetric Color Spaces

In any application that requires precise, reproducible, and device-independent presentation of colors, the use of calibrated color systems is an absolute necessity. For example, color calibration is routinely used throughout the digital print work flow but also in digital film production, professional photography, image databases, etc. One may have experienced how difficult it is, for example, to render a good photograph on a color laser printer, and even the color reproduction on monitors largely depends on the particular manufacturer and computer system.

All the color spaces described in Chapter 12, Sec. 12.2, somehow relate to the physical properties of some media device, such as the specific colors of the phosphor coatings inside a CRT tube or the colors of the inks used for printing. To make colors appear similar or even identical on different media modalities, we need a representation that is independent of how a particular device reproduces these colors. Color systems that describe colors in a measurable, device-independent fashion are called *colorimetric* or *calibrated*, and the field of *color science* is traditionally concerned with the properties and application of these color systems (see, e.g., [258] or [215] for an overview). While several colorimetric standards exist, we focus on the most widely used CIE systems in the remaining part of this section.

14.1 CIE Color Spaces

The XYZ color system, developed by the CIE (Commission Internationale d'Éclairage)¹ in the 1920s and standardized in 1931, is the foundation of most colorimetric color systems that are in use today [195, p. 22].

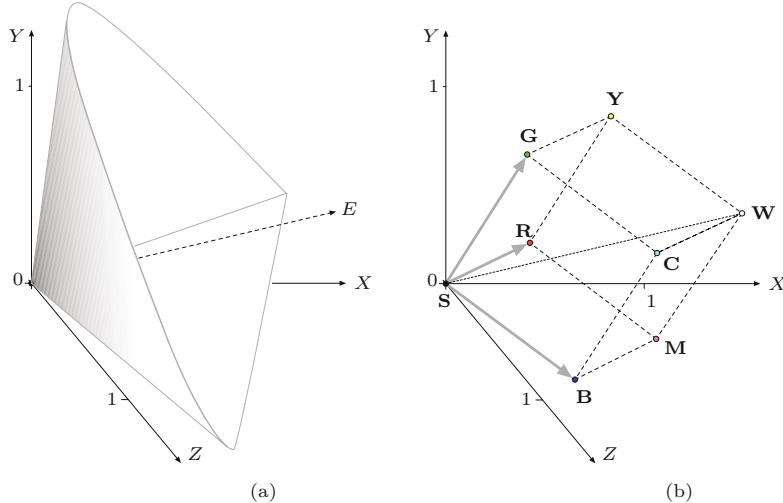
¹ International Commission on Illumination (www.cie.co.at).

14.1.1 CIE XYZ Color Space

The CIE XYZ color scheme was developed after extensive measurements of human visual perception under controlled conditions. It is based on three imaginary primary colors X , Y , Z , which are chosen such that all visible colors can be described as a summation of positive-only components, where the Y component corresponds to the perceived lightness or *luminosity* of a color. All visible colors lie inside a 3D cone-shaped region ([Fig. 14.1\(a\)](#)), which interestingly enough does not include the primary colors themselves.

Fig. 14.1

The XYZ color space is defined by the three imaginary primary colors X , Y , Z , where the Y dimension corresponds to the perceived luminance. All visible colors are contained inside an open, cone-shaped volume that originates at the black point S (a), where E denotes the axis of neutral (gray) colors. The RGB color space maps to the XYZ space as a linearly distorted cube (b). See also [Fig. 14.5\(a\)](#).



Some common color spaces, and the RGB color space in particular, conveniently relate to XYZ space by a *linear* coordinate transformation, as described in Sec. 14.4. Thus, as shown in [Fig. 14.1\(b\)](#), the RGB color space is embedded in the XYZ space as a distorted cube, and therefore straight lines in RGB space map to straight lines in XYZ again. The CIE XYZ scheme is (similar to the RGB color space) *nonlinear* with respect to human visual perception, that is, a particular fixed distance in XYZ is not perceived as a uniform color change throughout the entire color space. The XYZ coordinates of the RGB color cube (based on the primary colors defined by ITU-R BT.709) are listed in [Table 14.1](#).

14.1.2 CIE x, y Chromaticity

As mentioned, the luminance in XYZ color space increases along the Y axis, starting at the black point S located at the coordinate origin ($X = Y = Z = 0$). The color hue is independent of the luminance and thus independent of the Y value. To describe the corresponding “pure” color hues and saturation in a convenient manner, the CIE system also defines the three *chromaticity* values

$$x = \frac{X}{X + Y + Z}, \quad y = \frac{Y}{X + Y + Z}, \quad z = \frac{Z}{X + Y + Z}, \quad (14.1)$$

where (obviously) $x + y + z = 1$ and thus one of the three values (e.g., z) is redundant. Equation (14.1) describes a central projection from

Pt.	Color	R	G	B	X	Y	Z	x	y
S	Black	0.00	0.00	0.00	0.0000	0.0000	0.0000	0.3127	0.3290
R	Red	1.00	0.00	0.00	0.4125	0.2127	0.0193	0.6400	0.3300
Y	Yellow	1.00	1.00	0.00	0.7700	0.9278	0.1385	0.4193	0.5052
G	Green	0.00	1.00	0.00	0.3576	0.7152	0.1192	0.3000	0.6000
C	Cyan	0.00	1.00	1.00	0.5380	0.7873	1.0694	0.2247	0.3288
B	Blue	0.00	0.00	1.00	0.1804	0.0722	0.9502	0.1500	0.0600
M	Magenta	1.00	0.00	1.00	0.5929	0.2848	0.9696	0.3209	0.1542
W	White	1.00	1.00	1.00	0.9505	1.0000	1.0888	0.3127	0.3290

14.1 CIE COLOR SPACES

Table 14.1

Coordinates of the RGB color cube in CIE XYZ space. The X, Y, Z values refer to standard (ITU-R BT. 709) primaries and white point D65 (see Table 14.2), x, y denote the corresponding CIE chromaticity coordinates.

X, Y, Z coordinates onto the 3D plane

$$X + Y + Z = 1, \quad (14.2)$$

with the origin **S** as the projection center (Fig. 14.2). Thus, for an arbitrary XYZ color point $\mathbf{A} = (X_a, Y_a, Z_a)$, the corresponding chromaticity coordinates $\mathbf{a} = (x_a, y_a, z_a)$ are found by intersecting the line $\overline{\mathbf{SA}}$ with the $X + Y + Z = 1$ plane (Fig. 14.2(a)). The final x, y coordinates are the result of projecting these intersection points onto the X/Y -plane (Fig. 14.2(b)) by simply dropping the Z component z_a .

The result is the well-known horseshoe-shaped *CIE x, y chromaticity diagram*, which is shown in Fig. 14.2(c). Any x, y point in this diagram defines the hue and saturation of a particular color, but only the colors inside the horseshoe curve are potentially visible. Obviously an infinite number of X, Y, Z colors (with different luminance values) project to the same x, y, z chromaticity values, and the XYZ color coordinates thus cannot be uniquely reconstructed from given chromaticity values. Additional information is required. For example, it is common to specify the visible colors of the CIE system in the form Yxy , where Y is the original luminance component of the XYZ color. Given a pair of chromaticity values x, y (with $y > 0$) and an arbitrary Y value, the missing X, Z coordinates are obtained (using the definitions in Eqn. (14.1)) as

$$X = x \cdot \frac{Y}{y}, \quad Z = z \cdot \frac{Y}{y} = (1 - x - y) \cdot \frac{Y}{y}. \quad (14.3)$$

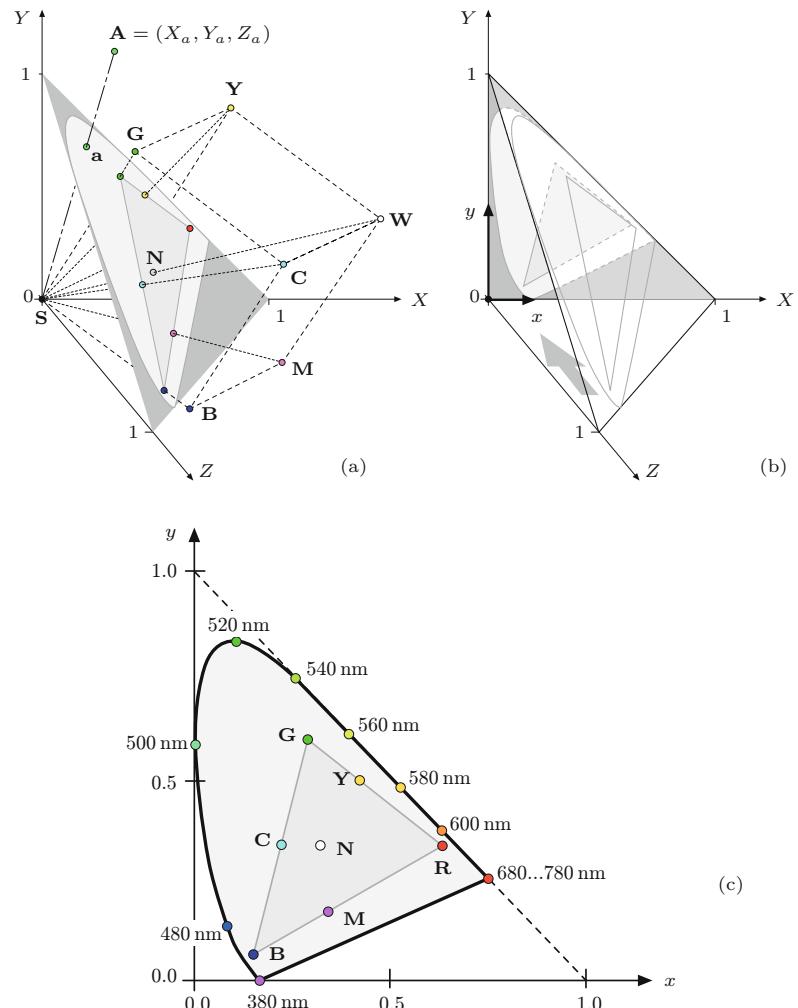
The CIE diagram not only yields an intuitive layout of color hues but exhibits some remarkable formal properties. The xy values along the outer horseshoe boundary correspond to monochromatic (“spectrally pure”), maximally saturated colors with wavelengths ranging from below 400 nm (purple) up to 780 nm (red). Thus the position of any color inside the xy diagram can be specified with respect to any of the primary colors at the boundary, except for the points on the connecting line (“purple line”) between 380 and 780 nm, whose purple hues do not correspond to primary colors but can only be generated by mixing other colors.

The *saturation* of colors falls off continuously toward the “neutral point” (**E**) at the center of the horseshoe, with $x = y = \frac{1}{3}$ (or $X = Y = Z = 1$, respectively) and zero saturation. All other colorless (i.e., gray) values also map to the neutral point, just as any set of colors

14 COLORIMETRIC COLOR SPACES

Fig. 14.2

CIE x, y chromaticity diagram. For an arbitrary XYZ color point $\mathbf{A} = (X_a, Y_a, Z_a)$, the chromaticity values $\mathbf{a} = (x_a, y_a, z_a)$ are obtained by a central projection onto the 3D plane $X + Y + Z = 1$ (a). The corner points of the RGB cube map to a triangle, and its white point \mathbf{W} maps to the (colorless) neutral point \mathbf{E} . The intersection points are then projected onto the X/Y plane (b) by simply dropping the Z component, which produces the familiar CIE chromaticity diagram shown in (c). The CIE diagram contains all visible color tones (hues and saturations) but no luminance information, with wavelengths in the range 380–780 nanometers. A particular color space is specified by at least three primary colors (tristimulus values; e.g., $\mathbf{R}, \mathbf{G}, \mathbf{B}$), which define a triangle (linear hull) containing all representable colors.



with the same hue but different brightness corresponds to a single x, y point. All possible composite colors lie inside the convex hull specified by the coordinates of the primary colors of the CIE diagram and, in particular, complementary colors are located on straight lines that run diagonally through the white point.

14.1.3 Standard Illuminants

A central goal of colorimetry is the quantitative measurement of colors in physical reality, which strongly depends on the color properties of the illumination. The CIE system specifies a number of standard illuminants for a variety of real and hypothetical light sources, each specified by a spectral radiant power distribution and the “correlated color temperature” (expressed in degrees Kelvin) [258, Sec. 3.3.3]. The following daylight (D) illuminants are particularly important for the design of digital color spaces (Table 14.2):

D50 emulates the spectrum of natural (direct) sunlight with an equivalent color temperature of approximately 5000°K. D50 is the recommended illuminant for viewing reflective images, such as paper prints. In practice, D50 lighting is commonly implemented with fluorescent lamps using multiple phosphors to approximate the specified color spectrum.

D65 has a correlated color temperature of approximately 6500°K and is designed to emulate the average (indirect) daylight observed under an overcast sky on the northern hemisphere. D65 is also used as the reference white for emissive devices, such as display screens.

The standard illuminants serve to specify the ambient viewing light but also to define the reference white points in various color spaces in the CIE color system. For example, the sRGB standard (see Sec. 14.4) refers to D65 as the media white point and D50 as the ambient viewing illuminant. In addition, the CIE system also specifies the range of admissible viewing angles (commonly at $\pm 2^\circ$).

	°K	X	Y	Z	x	y
D50	5000	0.96429	1.00000	0.82510	0.3457	0.3585
D65	6500	0.95045	1.00000	1.08905	0.3127	0.3290
N	—	1.00000	1.00000	1.00000	0.3333	0.3333

14.1 CIE COLOR SPACES

Table 14.2
CIE color parameters for the standard illuminants **D50** and **D65**. E denotes the absolute neutral point in CIE XYZ space.

14.1.4 Gamut

The set of all colors that can be handled by a certain media device or can be represented by a particular color space is called “gamut”. This is usually a contiguous region in the 3D CIE XYZ color space or, reduced to the representable color hues and ignoring the luminance component, a convex region in the 2D CIE chromaticity diagram.

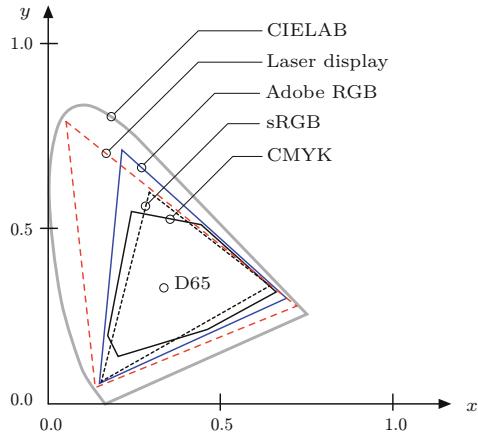
Figure 14.3 illustrates some typical gamut regions inside the CIE diagram. The gamut of an output device mainly depends on the technology employed. For example, ordinary color monitors are typically not capable of displaying all colors of the gamut covered by the corresponding color space (usually sRGB). Conversely, it is also possible that devices would reproduce certain colors that cannot be represented in the utilized color space. Significant deviations exist, for example, between the RGB color space and the gamuts associated with CMYK-based printers. Also, media devices with very large gamuts exist, as demonstrated by the laser display system in Fig. 14.3. Representing such large gamuts and, in particular, transforming between different color representations requires adequately sized color spaces, such as the Adobe-RGB color space or CIELAB (described in Sec. 14.2), which covers the entire visible portion of the CIE diagram.

14.1.5 Variants of the CIE Color Space

The original CIEXYZ color space and the derived xy chromaticity diagram have the disadvantage that color differences are not perceived equally in different regions of the color space. For example,

Fig. 14.3

Gamut regions for different color spaces and output devices inside the CIE diagram.



large color changes are perceived in the *magenta* region for a given shift in XYZ while the change is relatively small in the *green* region for the same coordinate distance. Several variants of the CIE color space have been developed for different purposes, primarily with the goal of creating perceptually uniform color representations without sacrificing the formal qualities of the CIE reference system. Popular CIE-derived color spaces include CIE YUV, YU'V', YC_bC_r, and particularly CIELAB and CIELUV, which are described in the following sections. In addition, CIE-compliant specifications exist for most common color spaces (see Ch. 12, Sec. 12.2), which allow more or less dependable conversions between almost any pair of color spaces.

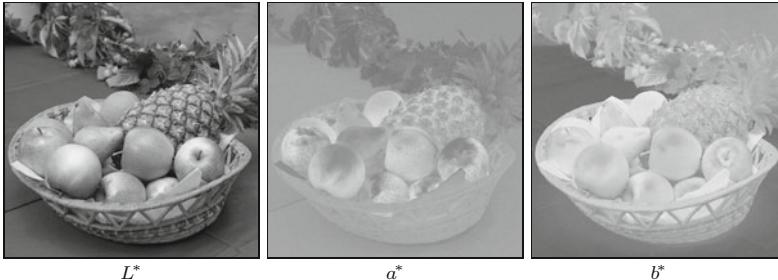
14.2 CIELAB

The CIELAB color model (specified by CIE in 1976) was developed with the goal of linearizing the representation with respect to human color perception and at the same time creating a more intuitive color system. Since then, CIELAB² has become a popular and widely used color model, particularly for high-quality photographic applications. It is used, for example, inside Adobe Photoshop as the standard model for converting between different color spaces. The dimensions in this color space are the luminosity L^* and the two color components a^*, b^* , which specify the color hue and saturation along the *green-red* and *blue-yellow* axes, respectively. All three components are *relative* values and refer to the specified reference white point $\mathbf{C}_{\text{ref}} = (X_{\text{ref}}, Y_{\text{ref}}, Z_{\text{ref}})$. In addition, a nonlinear correction function (similar to the modified gamma correction described in Ch. 4, Sec. 4.7.6) is applied to all three components, as will be detailed further.

14.2.1 CIEXYZ→CIELAB Conversion

Several specifications for converting to and from CIELAB space exist that, however, differ marginally and for very small L values only. The

² Often CIELAB is simply referred to as the “Lab” color space.



14.2 CIELAB

Fig. 14.4

CIELAB components shown as grayscale images. The contrast of the a^* and b^* images has been increased by 40% for better viewing.

current specification for converting between CIEXYZ and CIELAB colors is defined by ISO Standard 13655 [120] as follows:

$$L^* = 116 \cdot Y' - 16, \quad (14.4)$$

$$a^* = 500 \cdot (X' - Y'), \quad (14.5)$$

$$b^* = 200 \cdot (Y' - Z'), \quad (14.6)$$

with

$$X' = f_1\left(\frac{X}{X_{\text{ref}}}\right), \quad Y' = f_1\left(\frac{Y}{Y_{\text{ref}}}\right), \quad Z' = f_1\left(\frac{Z}{Z_{\text{ref}}}\right), \quad (14.7)$$

$$f_1(c) = \begin{cases} c^{1/3} & \text{for } c > \epsilon, \\ \kappa \cdot c + \frac{16}{116} & \text{for } c \leq \epsilon, \end{cases} \quad (14.8)$$

and

$$\epsilon = \left(\frac{6}{29}\right)^3 = \frac{216}{24389} \approx 0.008856, \quad (14.9)$$

$$\kappa = \frac{1}{116} \left(\frac{29}{3}\right)^3 = \frac{841}{108} \approx 7.787. \quad (14.10)$$

For the conversion in Eqn. (14.7), D65 is usually specified as the reference white point $\mathbf{C}_{\text{ref}} = (X_{\text{ref}}, Y_{\text{ref}}, Z_{\text{ref}})$, that is, $X_{\text{ref}} = 0.95047$, $Y_{\text{ref}} = 1.0$ and $Z_{\text{ref}} = 1.08883$ (see Table 14.2). The L^* values are positive and typically in the range $[0, 100]$ (often scaled to $[0, 255]$), but may theoretically be greater. Values for a^* and b^* are in the range $[-127, +127]$. Figure 14.4 shows the separation of a color image into the corresponding CIELAB components. Table 14.3 lists the relation between CIELAB and XYZ coordinates for selected RGB colors. The given $R'G'B'$ values are (nonlinear) sRGB coordinates with D65 as the reference white point.³ Figure 14.5(c) shows the transformation of the RGB color cube into the CIELAB color space.

14.2.2 CIELAB→CIEXYZ Conversion

The reverse transformation from CIELAB space to CIEXYZ coordinates is defined as follows:

$$X = X_{\text{ref}} \cdot f_2\left(L' + \frac{a^*}{500}\right), \quad (14.11)$$

$$Y = Y_{\text{ref}} \cdot f_2(L'), \quad (14.12)$$

$$Z = Z_{\text{ref}} \cdot f_2\left(L' - \frac{b^*}{200}\right), \quad (14.13)$$

³ Note that sRGB colors in Java are specified with respect to white point D50, which explains certain numerical deviations (see Sec. 14.7).

14 COLORIMETRIC COLOR SPACES

Table 14.3

CIELAB coordinates for selected color points in sRGB.

The sRGB components R' , G' , B' are nonlinear (i.e., gamma-corrected), white point is D65 (see Table 14.2).

Pt.	Color	sRGB			CIEXYZ (D65)			CIELAB		
		R'	G'	B'	X_{65}	Y_{65}	Z_{65}	L^*	a^*	b^*
S	Black	0.00	0.00	0.00	0.0000	0.0000	0.0000	0.00	0.00	0.00
R	Red	1.00	0.00	0.00	0.4125	0.2127	0.0193	53.24	80.09	67.20
Y	Yellow	1.00	1.00	0.00	0.7700	0.9278	0.1385	97.14	-21.55	94.48
G	Green	0.00	1.00	0.00	0.3576	0.7152	0.1192	87.74	-86.18	83.18
C	Cyan	0.00	1.00	1.00	0.5380	0.7873	1.0694	91.11	-48.09	-14.13
B	Blue	0.00	0.00	1.00	0.1804	0.0722	0.9502	32.30	79.19	-107.86
M	Magenta	1.00	0.00	1.00	0.5929	0.2848	0.9696	60.32	98.24	-60.83
W	White	1.00	1.00	1.00	0.9505	1.0000	1.0888	100.00	0.00	0.00
K	50% Gray	0.50	0.50	0.50	0.2034	0.2140	0.2330	53.39	0.00	0.00
R₇₅	75% Red	0.75	0.00	0.00	0.2155	0.1111	0.0101	39.77	64.51	54.13
R₅₀	50% Red	0.50	0.00	0.00	0.0883	0.0455	0.0041	25.42	47.91	37.91
R₂₅	25% Red	0.25	0.00	0.00	0.0210	0.0108	0.0010	9.66	29.68	15.24
P	Pink	1.00	0.50	0.50	0.5276	0.3812	0.2482	68.11	48.39	22.83

with

$$L' = \frac{L^* + 16}{116} \quad \text{and} \quad (14.14)$$

$$f_2(c) = \begin{cases} c^3 & \text{for } c^3 > \epsilon, \\ \frac{c - 16/116}{\kappa} & \text{for } c^3 \leq \epsilon, \end{cases} \quad (14.15)$$

and ϵ, κ as defined in Eqns. (14.9–14.10). The complete Java code for the CIELAB→XYZ conversion and the implementation of the associated `ColorSpace` class can be found in Progs. 14.1 and 14.2 (pp. 363–364).

14.3 CIELUV

14.3.1 CIEXYZ→CIELUV Conversion

The CIELUV component values L^* , u^* , v^* are calculated from given X , Y , Z color coordinates as follows:

$$L^* = 116 \cdot Y' - 16, \quad (14.16)$$

$$u^* = 13 \cdot L^* \cdot (u' - u'_{\text{ref}}), \quad (14.17)$$

$$v^* = 13 \cdot L^* \cdot (v' - v'_{\text{ref}}), \quad (14.18)$$

with Y' as defined in Eqn. (14.7) (identical to CIELAB) and

$$\begin{aligned} u' &= f_u(X, Y, Z), & u'_{\text{ref}} &= f_u(X_{\text{ref}}, Y_{\text{ref}}, Z_{\text{ref}}), \\ v' &= f_v(X, Y, Z), & v'_{\text{ref}} &= f_v(X_{\text{ref}}, Y_{\text{ref}}, Z_{\text{ref}}), \end{aligned} \quad (14.19)$$

with the correction functions

$$f_u(X, Y, Z) = \begin{cases} 0 & \text{for } X = 0, \\ \frac{4X}{X+15Y+3Z} & \text{for } X > 0, \end{cases} \quad (14.20)$$

$$f_v(X, Y, Z) = \begin{cases} 0 & \text{for } Y = 0, \\ \frac{9Y}{X+15Y+3Z} & \text{for } Y > 0. \end{cases} \quad (14.21)$$

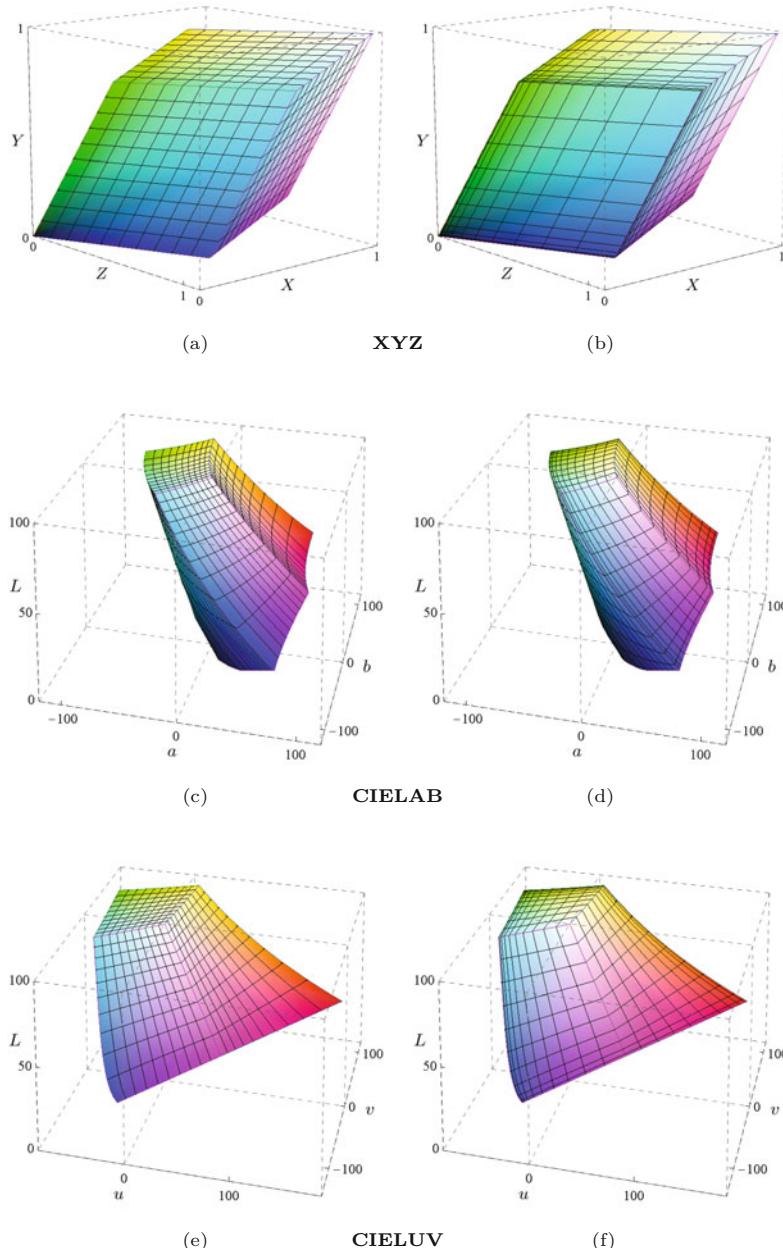


Fig. 14.5 Transformation of the RGB color cube to the XYZ, CIELAB, and CIELUV color space. The left column shows the color cube in *linear* RGB space, the right column in *nonlinear* sRGB space. Both RGB volumes were uniformly subdivided into $10 \times 10 \times 10$ cubes of equal size. In both cases, the transformation to XYZ space (a, b) yields a distorted cube with straight edges and planar faces. Due to the linear transformation from RGB to XYZ, the subdivision of the RGB cube remains uniform (a). However, the nonlinear transformation (due to gamma correction) from sRGB to XYZ makes the tessellation strongly nonuniform in XYZ space (b). Since CIELAB uses gamma correction as well, the transformation of the linear RGB cube in (c) appears much less uniform than the nonlinear sRGB cube in (d), although this appears to be the other way round in CIELUV (e, f). Note that the RGB/s-RGB color cube maps to a *non-convex* volume in both the CIELAB and the CIELUV space.

Note that the checks for zero X, Y in Eqns. (14.20)–(14.21) are not part of the original definitions but are essential in any real implementation to avoid divisions by zero.⁴

⁴ Remember though that floating-point values (`double`, `float`) should never be strictly tested against zero but compared to a sufficiently small (`epsilon`) quantity (see Sec. F.1.8 in the Appendix).

14 COLORIMETRIC COLOR SPACES

Table 14.4

CIELUV coordinates for selected color points in sRGB. Reference white point is D65. The L^* values are identical to CIELAB (see Table 14.3).

Pt.	Color	sRGB			CIEXYZ (D65)			CIELUV		
		R'	G'	B'	X_{65}	Y_{65}	Z_{65}	L^*	u^*	v^*
S	Black	0.00	0.00	0.00	0.0000	0.0000	0.0000	0.00	0.00	0.00
R	Red	1.00	0.00	0.00	0.4125	0.2127	0.0193	53.24	175.01	37.75
Y	Yellow	1.00	1.00	0.00	0.7700	0.9278	0.1385	97.14	7.70	106.78
G	Green	0.00	1.00	0.00	0.3576	0.7152	0.1192	87.74	-83.08	107.39
C	Cyan	0.00	1.00	1.00	0.5380	0.7873	1.0694	91.11	-70.48	-15.20
B	Blue	0.00	0.00	1.00	0.1804	0.0722	0.9502	32.30	-9.40	-130.34
M	Magenta	1.00	0.00	1.00	0.5929	0.2848	0.9696	60.32	84.07	-108.68
W	White	1.00	1.00	1.00	0.9505	1.0000	1.0888	100.00	0.00	0.00
K	50% Gray	0.50	0.50	0.50	0.2034	0.2140	0.2330	53.39	0.00	0.00
\mathbf{R}_{75}	75% Red	0.75	0.00	0.00	0.2155	0.1111	0.0101	39.77	130.73	28.20
\mathbf{R}_{50}	50% Red	0.50	0.00	0.00	0.0883	0.0455	0.0041	25.42	83.56	18.02
\mathbf{R}_{25}	25% Red	0.25	0.00	0.00	0.0210	0.0108	0.0010	9.66	31.74	6.85
P	Pink	1.00	0.50	0.50	0.5276	0.3812	0.2482	68.11	92.15	19.88

14.3.2 CIELUV→CIEXYZ Conversion

The reverse mapping from L^* , u^* , v^* components to X, Y, Z coordinates is defined as follows:

$$Y = Y_{\text{ref}} \cdot f_2\left(\frac{L^* + 16}{116}\right), \quad (14.22)$$

with $f_2()$ as defined in Eqn. (14.15), and

$$X = Y \cdot \frac{9u'}{4v'}, \quad Z = Y \cdot \frac{12 - 3u' - 20v'}{4v'}, \quad (14.23)$$

with

$$(u', v') = \begin{cases} (u'_{\text{ref}}, v'_{\text{ref}}) & \text{for } L^* = 0, \\ (u'_{\text{ref}}, v'_{\text{ref}}) + \frac{1}{13 \cdot L^*} \cdot (u^*, v^*) & \text{for } L^* > 0, \end{cases} \quad (14.24)$$

and $u'_{\text{ref}}, v'_{\text{ref}}$ as in Eqn. (14.19).⁵

14.3.3 Measuring Color Differences

Due to its high uniformity with respect to human color perception, the CIELAB color space is a particularly good choice for determining the difference between colors (the same holds for the CIELUV space) [94, p. 57]. The difference between two color points $\mathbf{c}_1 = (L_1^*, a_1^*, b_1^*)$ and $\mathbf{c}_2 = (L_2^*, a_2^*, b_2^*)$ can be found by simply measuring the *Euclidean distance* in CIELAB or CIELUV space, for example,

$$\text{ColorDist}(\mathbf{c}_1, \mathbf{c}_2) = \|\mathbf{c}_1 - \mathbf{c}_2\| \quad (14.25)$$

$$= \sqrt{(L_1^* - L_2^*)^2 + (a_1^* - a_2^*)^2 + (b_1^* - b_2^*)^2}. \quad (14.26)$$

14.4 Standard RGB (sRGB)

CIE-based color spaces such as CIELAB (and CIELUV) are device-independent and have a gamut sufficiently large to represent virtually

⁵ No explicit check for zero denominators is required in Eqn. (14.23) since v' can be assumed to be greater than zero.

all visible colors in the CIEXYZ system. However, in many computer-based, display-oriented applications, such as computer graphics or multimedia, the direct use of CIE-based color spaces may be too cumbersome or inefficient.

sRGB (“standard RGB” [119]) was developed (jointly by Hewlett-Packard and Microsoft) with the goal of creating a precisely specified color space for these applications, based on standardized mappings with respect to the colorimetric CIEXYZ color space. This includes precise specifications of the three primary colors, the white reference point, ambient lighting conditions, and gamma values. Interestingly, the sRGB color specification is the same as the one specified many years before for the European PAL/SECAM television standards. Compared to CIELAB, sRGB exhibits a relatively small gamut (see Fig. 14.3), which, however, includes most colors that can be reproduced by current computer and video monitors. Although sRGB was not designed as a universal color space, its CIE-based specification at least permits more or less exact conversions to and from other color spaces.

Several standard image formats, including EXIF (JPEG) and PNG are based on sRGB color data, which makes sRGB the de facto standard for digital still cameras, color printers, and other imaging devices at the consumer level [107]. sRGB is used as a relatively dependable archive format for digital images, particularly in less demanding applications that do not require (or allow) explicit color management [225]. Thus, in practice, working with any RGB color data almost always means dealing with sRGB. It is thus no coincidence that sRGB is also the common color scheme in Java and is extensively supported by the Java standard API (see Sec. 14.7 for details).

Table 14.5 lists the key parameters of the sRGB color space (i.e., the XYZ coordinates for the primary colors **R**, **G**, **B** and the white point **W** (D65)), which are defined according to ITU-R BT.709 [122] (see Tables 14.1 and 14.2). Together, these values permit the unambiguous mapping of all other colors in the CIE diagram.

Pt.	<i>R</i>	<i>G</i>	<i>B</i>	X_{65}	Y_{65}	Z_{65}	x_{65}	y_{65}
R	1.0	0.0	0.0	0.412453	0.212671	0.019334	0.6400	0.3300
G	0.0	1.0	0.0	0.357580	0.715160	0.119193	0.3000	0.6000
B	0.0	0.0	1.0	0.180423	0.072169	0.950227	0.1500	0.0600
W	1.0	1.0	1.0	0.950456	1.000000	1.088754	0.3127	0.3290

Table 14.5
sRGB tristimulus values **R**, **G**, **B** with reference to the white point D65 (**W**).

14.4.1 Linear vs. Nonlinear Color Components

sRGB is a *nonlinear* color space with respect to the XYZ coordinate system, and it is important to carefully distinguish between the *linear* and *nonlinear* RGB component values. The nonlinear values (denoted R', G', B') represent the actual color tuples, the data values read from an image file or received from a digital camera. These values are pre-corrected with a fixed Gamma (≈ 2.2) such that they can be easily viewed on a common color monitor without any additional conversion. The corresponding *linear* components (denoted

R, G, B) relate to the CIEXYZ color space by a linear mapping and can thus be computed from X, Y, Z coordinates and vice versa by simple matrix multiplication, that is,

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = M_{\text{RGB}} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = M_{\text{RGB}}^{-1} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}, \quad (14.27)$$

with

$$M_{\text{RGB}} = \begin{pmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix}, \quad (14.28)$$

$$M_{\text{RGB}}^{-1} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix}. \quad (14.29)$$

Notice that the three column vectors of M_{RGB}^{-1} (Eqn. (14.29)) are the coordinates of the primary colors \mathbf{R} , \mathbf{G} , \mathbf{B} (tristimulus values) in XYZ space (cf. [Table 14.5](#)) and thus

$$\mathbf{R} = M_{\text{RGB}}^{-1} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{G} = M_{\text{RGB}}^{-1} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \mathbf{B} = M_{\text{RGB}}^{-1} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}. \quad (14.30)$$

14.4.2 CIEXYZ→sRGB Conversion

To transform a given XYZ color to sRGB ([Fig. 14.6](#)), we first compute the *linear* R, G, B values by multiplying the (X, Y, Z) coordinate vector with the matrix M_{RGB} (Eqn. (14.28)),

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = M_{\text{RGB}} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}. \quad (14.31)$$

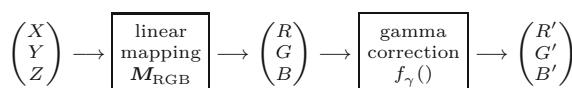
Subsequently, a modified gamma correction (see Ch. 4, Sec. 4.7.6) with $\gamma = 2.4$ (which corresponds to an effective gamma value of ca. 2.2) is applied to the linear R, G, B values,

$$R' = f_1(R), \quad G' = f_1(G), \quad B' = f_1(B), \quad (14.32)$$

with

$$f_1(c) = \begin{cases} 12.92 \cdot c & \text{for } c \leq 0.0031308, \\ 1.055 \cdot c^{1/2.4} - 0.055 & \text{for } c > 0.0031308. \end{cases} \quad (14.33)$$

Fig. 14.6
Color transformation
from CIEXYZ to sRGB.



The resulting sRGB components R', G', B' are limited to the interval $[0, 1]$ (see [Table 14.6](#)). To obtain discrete numbers, the R', G', B' values are finally scaled linearly to the 8-bit integer range $[0, 255]$.

Pt.	Color	sRGB (nonlinear)			RGB (linear)			CIEXYZ		
		R'	G'	B'	R	G	B	X ₆₅	Y ₆₅	Z ₆₅
S	Black	0.00	0.00	0.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
R	Red	1.00	0.00	0.00	1.0000	0.0000	0.0000	0.4125	0.2127	0.0193
Y	Yellow	1.00	1.00	0.00	1.0000	1.0000	0.0000	0.7700	0.9278	0.1385
G	Green	0.00	1.00	0.00	0.0000	1.0000	0.0000	0.3576	0.7152	0.1192
C	Cyan	0.00	1.00	1.00	0.0000	1.0000	1.0000	0.5380	0.7873	1.0694
B	Blue	0.00	0.00	1.00	0.0000	0.0000	1.0000	0.1804	0.0722	0.9502
M	Magenta	1.00	0.00	1.00	1.0000	0.0000	1.0000	0.5929	0.2848	0.9696
W	White	1.00	1.00	1.00	1.0000	1.0000	1.0000	0.9505	1.0000	1.0888
K	50% Gray	0.50	0.50	0.50	0.2140	0.2140	0.2140	0.2034	0.2140	0.2330
R₇₅	75% Red	0.75	0.00	0.00	0.5225	0.0000	0.0000	0.2155	0.1111	0.0101
R₅₀	50% Red	0.50	0.00	0.00	0.2140	0.0000	0.0000	0.0883	0.0455	0.0041
R₂₅	25% Red	0.25	0.00	0.00	0.0509	0.0000	0.0000	0.0210	0.0108	0.0010
P	Pink	1.00	0.50	0.50	1.0000	0.2140	0.2140	0.5276	0.3812	0.2482

14.4.3 sRGB→CIEXYZ Conversion

To calculate the reverse transformation from sRGB to XYZ, the given (nonlinear) $R'G'B'$ values (in the range $[0, 1]$) are first linearized by inverting the gamma correction (Eqn. (14.33)), that is,

$$R = f_2(R'), \quad G = f_2(G'), \quad B = f_2(B'), \quad (14.34)$$

with

$$f_2(c') = \begin{cases} \frac{c'}{12.92} & \text{for } c' \leq 0.04045, \\ \left(\frac{c'+0.055}{1.055}\right)^{2.4} & \text{for } c' > 0.04045. \end{cases} \quad (14.35)$$

Subsequently, the linearized (R, G, B) vector is transformed to XYZ coordinates by multiplication with the inverse of the matrix \mathbf{M}_{RGB} (Eqn. (14.29)),

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \mathbf{M}_{\text{RGB}}^{-1} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}. \quad (14.36)$$

14.4.4 Calculations with Nonlinear sRGB Values

Due to the wide use of sRGB in digital photography, graphics, multimedia, Internet imaging, etc., there is a probability that a given image is encoded in sRGB colors. If, for example, a JPEG image is opened with ImageJ or Java, the pixel values in the resulting data array are media-oriented (i.e., nonlinear R', G', B' components of the sRGB color space). Unfortunately, this fact is often overlooked by programmers, with the consequence that colors are incorrectly manipulated and reproduced.

As a general rule, any arithmetic operation on color values should always be performed on the *linearized* R, G, B components, which are obtained from the nonlinear R', G', B' values through the inverse gamma function f_γ^{-1} (Eqn. (14.35)) and converted back again with f_γ (Eqn. (14.33)).

Example: color to grayscale conversion

The principle of converting RGB colors to grayscale values by computing a weighted sum of the color components was described already

14.4 STANDARD RGB (sRGB)

Table 14.6

CIEXYZ coordinates for selected sRGB colors. The table lists the *nonlinear* R', G' , and B' components, the *linearized* R, G , and B values, and the corresponding X, Y , and Z coordinates (for white point D65). The linear and nonlinear RGB values are identical for the extremal points of the RGB color cube **S**, ..., **W** (top rows) because the gamma correction does not affect 0 and 1 component values. However, *intermediate* colors (**K**, ..., **P**, shaded rows) may exhibit large differences between the nonlinear and linear components (e.g., compare the R' and R values for **R₂₅**).

in Chapter 12, Sec. 12.2.1, where we had simply ignored the issue of possible nonlinearities. As one may have guessed, however, the variables R , G , B , and Y in Eqn. (12.10) on p. 305,

$$Y = 0.2125 \cdot R + 0.7154 \cdot G + 0.072 \cdot B \quad (14.37)$$

implicitly refer to *linear* color and gray values, respectively, and not the raw sRGB values! Based on Eqn. (14.37), the *correct* grayscale conversion from raw (nonlinear) sRGB components R' , G' , B' is

$$Y' = f_1(0.2125 \cdot f_2(R') + 0.7154 \cdot f_2(G') + 0.0721 \cdot f_2(B')), \quad (14.38)$$

with $f_\gamma()$ and $f_\gamma^{-1}()$ as defined in Eqns. (14.33) and (14.35). The result (Y') is again a nonlinear, sRGB-compatible gray value; that is, the sRGB color tuple (Y', Y', Y') should have the same perceived luminance as the original color (R', G', B') .

Note that setting the components of an sRGB color pixel to three arbitrary but identical values Y' ,

$$(R', G', B') \leftarrow (Y', Y', Y')$$

always creates a gray (colorless) pixel, despite the nonlinearities of the sRGB space. This is due to the fact that the gamma correction (Eqns. (14.33) and (14.35)) applies evenly to all three color components and thus any three identical values map to a (linearized) color on the straight gray line between the black point **S** and the white point **W** in XYZ space (cf. Fig. 14.1(b)).

For many applications, however, the following *approximation* to the exact grayscale conversion in Eqn. (14.38) is sufficient. It works without converting the sRGB values (i.e., directly on the nonlinear R' , G' , B' components) by computing a linear combination

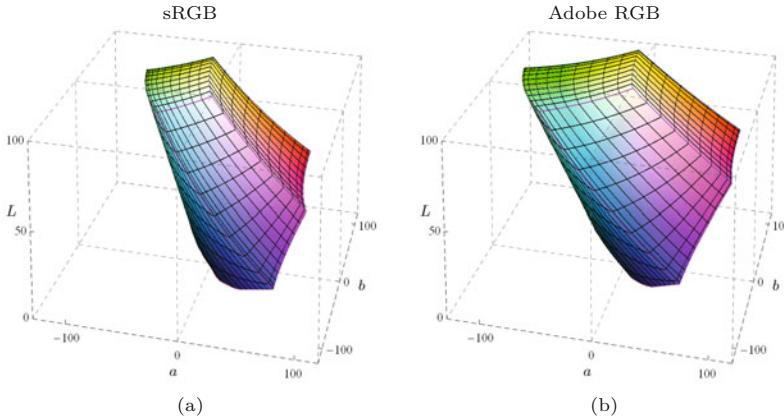
$$Y' \approx w'_R \cdot R' + w'_G \cdot G' + w'_B \cdot B', \quad (14.39)$$

with a slightly different set of weights; for example, $w'_R = 0.309$, $w'_G = 0.609$, $w'_B = 0.082$, as proposed in [188]. The resulting quantity from Eqn. (14.39) is sometimes called *luma* (compared to *luminance* in Eqn. (14.37)).

14.5 Adobe RGB

A distinct weakness of sRGB is its relatively small gamut, which is limited to the range of colors reproducible by ordinary color monitors. This causes problems, for example, in printing, where larger gamuts are needed, particularly in the green regions. The “Adobe RGB (1998)” [1] color space, developed by Adobe as their own standard, is based on the same general concept as sRGB but exhibits a significantly larger gamut (Fig. 14.3), which extends its use particularly to print applications. Figure 14.7 shows the noted difference between the sRGB and Adobe RGB gamuts in 3D CIEXYZ color space.

The neutral point of Adobe RGB corresponds to the D65 standard (with $x = 0.3127$, $y = 0.3290$), and the gamma value is 2.199



14.6 CHROMATIC ADAPTATION

Fig. 14.7
Gamuts of sRGB and Adobe RGB shown in CIELAB color space. The volume of the sRGB gamut (a) is significantly smaller than the Adobe RGB gamut (b), particularly in the green color region. The tesselation corresponds to a uniform subdivision of the original RGB cubes (in the respective color spaces).

(compared with 2.4 for sRGB) for the forward correction and $\frac{1}{2,199}$ for the inverse correction, respectively. The associated file specification provides for a number of different codings (8- to 16-bit integer and 32-bit floating point) for the color components. Adobe RGB is frequently used in professional photography as an alternative to the CIELAB color space and for picture archive applications.

14.6 Chromatic Adaptation

The human eye has the capability to interpret colors as being constant under varying viewing conditions and illumination in particular. A white sheet of paper appears white to us in bright daylight as well as under fluorescent lighting, although the spectral composition of the light that enters the eye is completely different in both situations. The CIE color system takes into account the color temperature of the ambient lighting because the exact interpretation of XYZ color values also requires knowledge of the corresponding reference white point. For example, a color value (X, Y, Z) specified with respect to the D50 reference white point is generally perceived differently when reproduced by a D65-based media device, although the absolute (i.e., measured) color is the same. Thus the actual meaning of XYZ values cannot be known without knowing the corresponding white point. This is known as *relative colorimetry*.

If colors are specified with respect to *different* white points, for example $\mathbf{W}_1 = (X_{W1}, Y_{W1}, Z_{W1})$ and $\mathbf{W}_2 = (X_{W2}, Y_{W2}, Z_{W2})$, they can be related by first applying a so-called *chromatic adaptation transformation* (CAT) [114, Ch. 34] in XYZ color space. This transformation determines, for given color coordinates (X_1, Y_1, Z_1) and the associated white point \mathbf{W}_1 , the new color coordinates (X_2, Y_2, Z_2) relative to another white point \mathbf{W}_2 .

14.6.1 XYZ Scaling

The simplest chromatic adaptation method is XYZ scaling, where the individual color coordinates are individually multiplied by the ratios of the corresponding white point coordinates, that is,

$$X_2 = X_1 \cdot \frac{\hat{X}_2}{\hat{X}_1}, \quad Y_2 = Y_1 \cdot \frac{\hat{Y}_2}{\hat{Y}_1}, \quad Z_2 = Z_1 \cdot \frac{\hat{Z}_2}{\hat{Z}_1}. \quad (14.40)$$

For example, for converting colors (X_{65}, Y_{65}, Z_{65}) related to the white point $\mathbf{D65} = (\hat{X}_{65}, \hat{Y}_{65}, \hat{Z}_{65})$ to the corresponding colors for white point $\mathbf{D50} = (\hat{X}_{50}, \hat{Y}_{50}, \hat{Z}_{50})$,⁶ the concrete scaling is

$$\begin{aligned} X_{50} &= X_{65} \cdot \frac{\hat{X}_{50}}{\hat{X}_{65}} = X_{65} \cdot \frac{0.964296}{0.950456} = X_{65} \cdot 1.01456, \\ Y_{50} &= Y_{65} \cdot \frac{\hat{Y}_{50}}{\hat{Y}_{65}} = Y_{65} \cdot \frac{1.000000}{1.000000} = Y_{65}, \\ Z_{50} &= Z_{65} \cdot \frac{\hat{Z}_{50}}{\hat{Z}_{65}} = Z_{65} \cdot \frac{0.825105}{0.1088754} = Z_{65} \cdot 0.757843. \end{aligned} \quad (14.41)$$

This form of scaling the color coordinates in XYZ space is usually not considered a good color adaptation model and is not recommended for high-quality applications.

14.6.2 Bradford Adaptation

The most common chromatic adaptation models are based on scaling the color coordinates not directly in XYZ but in a “virtual” $R^*G^*B^*$ color space obtained from the XYZ values by a linear transformation

$$\begin{pmatrix} R^* \\ G^* \\ B^* \end{pmatrix} = \mathbf{M}_{\text{CAT}} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}, \quad (14.42)$$

where \mathbf{M}_{CAT} is a 3×3 transformation matrix (defined in Eqn. (14.45)). After appropriate scaling, the $R^*G^*B^*$ coordinates are transformed back to XYZ, so the complete adaptation transform from color coordinates X_1, Y_1, Z_1 (w.r.t. white point $\mathbf{W}_1 = (X_{W1}, Y_{W1}, Z_{W1})$) to the new color coordinates X_2, Y_2, Z_2 (w.r.t. white point $\mathbf{W}_2 = (X_{W2}, Y_{W2}, Z_{W2})$) takes the form

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \mathbf{M}_{\text{CAT}}^{-1} \cdot \begin{pmatrix} \frac{R_{W2}^*}{R_{W1}^*} & 0 & 0 \\ 0 & \frac{G_{W2}^*}{G_{W1}^*} & 0 \\ 0 & 0 & \frac{B_{W2}^*}{B_{W1}^*} \end{pmatrix} \cdot \mathbf{M}_{\text{CAT}} \cdot \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix}, \quad (14.43)$$

where the diagonal elements $\frac{R_{W2}^*}{R_{W1}^*}, \frac{G_{W2}^*}{G_{W1}^*}, \frac{B_{W2}^*}{B_{W1}^*}$ are the (constant) ratios of the $R^*G^*B^*$ values of the white points $\mathbf{W}_2, \mathbf{W}_1$, respectively; that is,

$$\begin{pmatrix} R_{W1}^* \\ G_{W1}^* \\ B_{W1}^* \end{pmatrix} = \mathbf{M}_{\text{CAT}} \cdot \begin{pmatrix} X_{W1} \\ Y_{W1} \\ Z_{W1} \end{pmatrix}, \quad \begin{pmatrix} R_{W2}^* \\ G_{W2}^* \\ B_{W2}^* \end{pmatrix} = \mathbf{M}_{\text{CAT}} \cdot \begin{pmatrix} X_{W2} \\ Y_{W2} \\ Z_{W2} \end{pmatrix}. \quad (14.44)$$

The “Bradford” model [114, p. 590] specifies for Eqn. (14.43) the particular transformation matrix

$$\mathbf{M}_{\text{CAT}} = \begin{pmatrix} 0.8951 & 0.2664 & -0.1614 \\ -0.7502 & 1.7135 & 0.0367 \\ 0.0389 & -0.0685 & 1.0296 \end{pmatrix}. \quad (14.45)$$

⁶ See Table 14.2.

14.6 CHROMATIC ADAPTATION

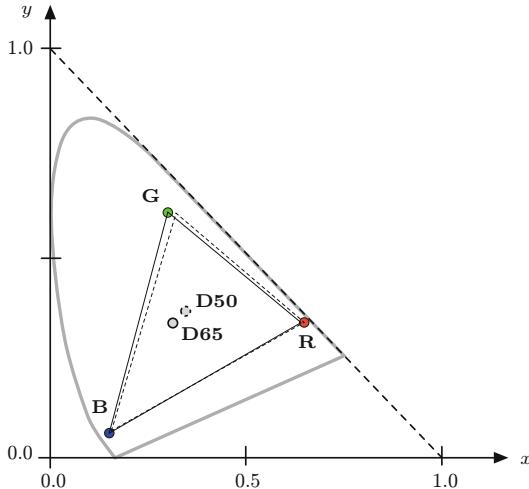


Fig. 14.8
Bradford chromatic adaptation from white point D65 to D50. The solid triangle represents the original RGB gamut for white point D65, with the primaries (R, G, B) located at the corner points. The dashed triangle is the corresponding gamut after chromatic adaptation to white point D50.

Inserting M_{CAT} matrix in Eqn. (14.43) gives the complete chromatic adaptation. For example, the resulting transformation for converting from D65-based to D50-based colors (i.e., $\mathbf{W}_1 = \mathbf{D65}$, $\mathbf{W}_2 = \mathbf{D50}$, as listed in Table 14.2) is

$$\begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} = M_{50|65} \cdot \begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix} \\ = \begin{pmatrix} 1.047884 & 0.022928 & -0.050149 \\ 0.029603 & 0.990437 & -0.017059 \\ -0.009235 & 0.015042 & 0.752085 \end{pmatrix} \cdot \begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix}, \quad (14.46)$$

and conversely from D50-based to D65-based colors (i.e., $\mathbf{W}_1 = \mathbf{D50}$, $\mathbf{W}_2 = \mathbf{D65}$),

$$\begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix} = M_{65|50} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} = M_{50|65}^{-1} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} \\ = \begin{pmatrix} 0.955513 & -0.023079 & 0.063190 \\ -0.028348 & 1.009992 & 0.021019 \\ 0.012300 & -0.020484 & 1.329993 \end{pmatrix} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix}. \quad (14.47)$$

Figure 14.8 illustrates the effects of adaptation from the D65 white point to D50 in the CIE x, y chromaticity diagram. A short list of corresponding color coordinates is given in Table 14.7.

The Bradford model is a widely used chromatic adaptation scheme but several similar procedures have been proposed (see also Exercise 14.1). Generally speaking, chromatic adaptation and related problems have a long history in color engineering and are still active fields of scientific research [258, Ch. 5, Sec. 5.12].

14 COLORIMETRIC COLOR SPACES

Table 14.7

Bradford chromatic adaptation from white point D65 to D50 for selected sRGB colors. The XYZ coordinates X_{65} , Y_{65} , Z_{65} relate to the original white point D65 (\mathbf{W}_1). X_{50} , Y_{50} , Z_{50} are the corresponding coordinates for the new white point D50 (\mathbf{W}_2), obtained with the Bradford adaptation according to Eqn. (14.46).

Pt.	Color	sRGB			XYZ (D65)			XYZ (D50)		
		R'	G'	B'	X_{65}	Y_{65}	Z_{65}	X_{50}	Y_{50}	Z_{50}
S	Black	0.00	0.0	0.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
R	Red	1.00	0.0	0.0	0.4125	0.2127	0.0193	0.4361	0.2225	0.0139
Y	Yellow	1.00	1.0	0.0	0.7700	0.9278	0.1385	0.8212	0.9394	0.1110
G	Green	0.00	1.0	0.0	0.3576	0.7152	0.1192	0.3851	0.7169	0.0971
C	Cyan	0.00	1.0	1.0	0.5380	0.7873	1.0694	0.5282	0.7775	0.8112
B	Blue	0.00	0.0	1.0	0.1804	0.0722	0.9502	0.1431	0.0606	0.7141
M	Magenta	1.00	0.0	1.0	0.5929	0.2848	0.9696	0.5792	0.2831	0.7280
W	White	1.00	1.0	1.0	0.9505	1.0000	1.0888	0.9643	1.0000	0.8251
K	50% Gray	0.50	0.5	0.5	0.2034	0.2140	0.2330	0.2064	0.2140	0.1766
R₇₅	75% Red	0.75	0.0	0.0	0.2155	0.1111	0.0101	0.2279	0.1163	0.0073
R₅₀	50% Red	0.50	0.0	0.0	0.0883	0.0455	0.0041	0.0933	0.0476	0.0030
R₂₅	25% Red	0.25	0.0	0.0	0.0210	0.0108	0.0010	0.0222	0.0113	0.0007
P	Pink	1.00	0.5	0.5	0.5276	0.3812	0.2482	0.5492	0.3889	0.1876

14.7 Colorimetric Support in Java

sRGB is the standard color space in Java; that is, the components of color objects and RGB color images are gamma-corrected, *nonlinear* R' , G' , B' values (see Fig. 14.6). The nonlinear R' , G' , B' values are related to the linear R , G , B values by a modified gamma correction, as specified by the sRGB standard (Eqns. (14.33) and (14.35)).

14.7.1 Profile Connection Space (PCS)

The Java API (AWT) provides classes for representing color objects and color spaces, together with a rich set of corresponding methods. Java's color system is designed after the ICC⁷ "color management architecture", which uses a CIEXYZ-based device-independent color space called the "profile connection space" (PCS) [118, 121]. The PCS color space is used as the intermediate reference for converting colors between different color spaces. The ICC standard defines device profiles (see Sec. 14.7.4) that specify the transforms to convert between a device's color space and the PCS. The advantage of this approach is that for any given device only a single color transformation (profile) must be specified to convert between device-specific colors and the unified, colorimetric profile connection space. Every `ColorSpace` class (or subclass) provides the methods `fromCIEXYZ()` and `toCIEXYZ()` to convert device color values to XYZ coordinates in the standardized PCS. Figure 14.9 illustrates the principal application of `ColorSpace` objects for converting colors between different color spaces in Java using the XYZ space as a common "hub".

Different to the sRGB specification, the ICC specifies **D50** (and *not* D65) as the illuminant white point for its default PCS color space (see Table 14.2). The reason is that the ICC standard was developed primarily for color management in photography, graphics, and printing, where D50 is normally used as the reflective media white point. The Java methods `fromCIEXYZ()` and `toCIEXYZ()` thus take and return X , Y , Z color coordinates that are relative to the D50 white point. The resulting coordinates for the primary colors (listed in Table 14.8) are different from the ones given for white point D65 (see Table 14.5)! This is a frequent cause of confusion since the sRGB

⁷ International Color Consortium (ICC, www.color.org).

14.7 COLORIMETRIC SUPPORT IN JAVA

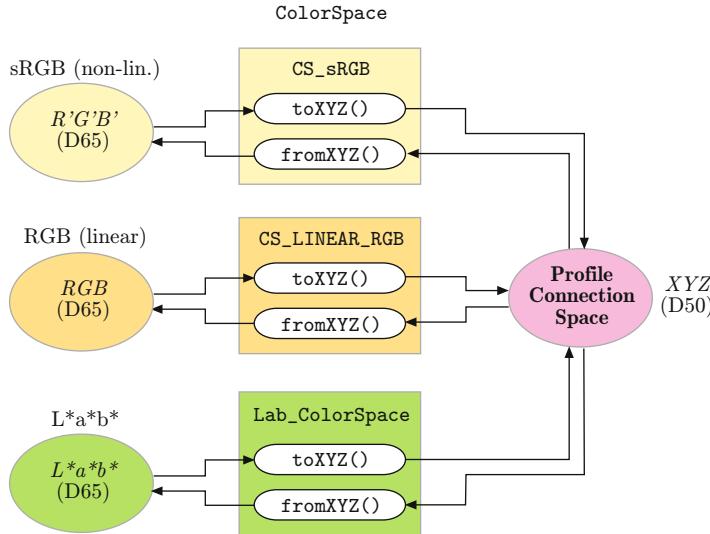


Fig. 14.9

XYZ-based color conversion in Java. `ColorSpace` objects implement the methods `fromCIEXYZ()` and `toCIEXYZ()` to convert color vectors from and to the CIEXYZ color space, respectively. Colorimetric transformations between color spaces can be accomplished as a two-step process via the XYZ space. For example, to convert from sRGB to CIELAB, the sRGB color is first converted to XYZ and subsequently from XYZ to CIELAB. Notice that Java's standard XYZ color space is based on the D50 white point, while most common color spaces refer to D65.

Pt.	R	G	B	X_{50}	Y_{50}	Z_{50}	x_{50}	y_{50}
R	1.0	0.0	0.0	0.436108	0.222517	0.013931	0.6484	0.3309
G	0.0	1.0	0.0	0.385120	0.716873	0.097099	0.3212	0.5978
B	0.0	0.0	1.0	0.143064	0.060610	0.714075	0.1559	0.0660
W	1.0	1.0	1.0	0.964296	1.000000	0.825106	0.3457	0.3585

component values are D65-based (as specified by the sRGB standard) but Java's XYZ values are relative to the D50.

Chromatic adaptation (see Sec. 14.6) is used to convert between XYZ color coordinates that are measured with respect to different white points. The ICC specification [118] recommends a linear chromatic adaptation based on the Bradford model to convert between the D65-related XYZ coordinates (X_{65}, Y_{65}, Z_{65}) and D50-related values (X_{50}, Y_{50}, Z_{50}). This is also implemented by the Java API.

The complete mapping between the linearized sRGB color values (R, G, B) and the D50-based (X_{50}, Y_{50}, Z_{50}) coordinates can be expressed as a linear transformation composed of the $\text{RGB} \rightarrow \text{XYZ}_{65}$ transformation by matrix M_{RGB} (Eqns. (14.28) and (14.29)) and the chromatic adaptation transformation $\text{XYZ}_{65} \rightarrow \text{XYZ}_{50}$ defined by the matrix $M_{50|65}$ (Eqn. (14.46)),

$$\begin{aligned} \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} &= M_{50|65} \cdot M_{\text{RGB}}^{-1} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \left(M_{\text{RGB}} \cdot M_{65|50} \right)^{-1} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \\ &= \begin{pmatrix} 0.436131 & 0.385147 & 0.143033 \\ 0.222527 & 0.716878 & 0.060600 \\ 0.013926 & 0.097080 & 0.713871 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}, \quad (14.48) \end{aligned}$$

and, in the reverse direction,

Table 14.8

Color coordinates for sRGB primaries and the white point in Java's default XYZ color space. Color coordinates for sRGB primaries and the white point in Java's default XYZ color space. The white point **W** is equal to D50.

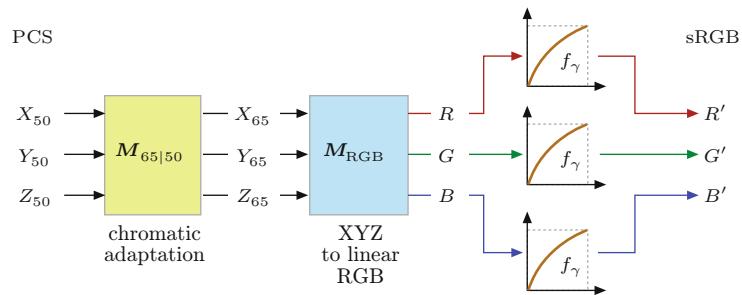
$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = M_{\text{RGB}} \cdot M_{65|50} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix}$$

$$= \begin{pmatrix} 3.133660 & -1.617140 & -0.490588 \\ -0.978808 & 1.916280 & 0.033444 \\ 0.071979 & -0.229051 & 1.405840 \end{pmatrix} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix}. \quad (14.49)$$

Equations (14.48) and (14.49) are the transformations implemented by the methods `toCIEXYZ()` and `fromCIEXYZ()`, respectively, for Java's default sRGB `ColorSpace` class. Of course, these methods must also perform the necessary gamma correction between the linear R, G, B components and the actual (nonlinear) sRGB values R', G', B' . Figure 14.10 illustrates the complete transformation from D50-based PCS coordinates to nonlinear sRGB values.

Fig. 14.10

Transformation from D50-based CIEXYZ coordinates (X_{50}, Y_{50}, Z_{50}) in Java's *Profile Connection Space* (PCS) to nonlinear sRGB values (R', G', B') . The first step ist chromatic adaptation from D50 to D65 (by $M_{65|50}$), followed by mapping the CIE-XYZ coordinates to linear RGB values (by M_{RGB}). Finally, gamma correction is applied individually to all three color components.



14.7.2 Color-Related Java Classes

The Java standard API offers extensive support for working with colors and color images. The most important classes contained in the Java AWT package are:

- `Color`: defines individual color objects.
- `ColorSpace`: specifies entire color spaces.
- `ColorModel`: describes the structure of color images; e.g., full-color images or indexed-color images (see Prog. 12.3 on p. 301).

Class Color (java.awt.Color)

An object of class `Color` describes a particular color in the associated color space, which defines the number and type of the color components. `Color` objects are primarily used for graphic operations, such as to specify the color for drawing or filling graphic objects. Unless the color space is not explicitly specified, new `Color` objects are created as sRGB colors. The arguments passed to the `Color` constructor methods may be either `float` components in the range [0, 1] or integers in the range [0, 255], as demonstrated by the following example:

```
Color pink = new Color(1.0f, 0.5f, 0.5f);
Color blue = new Color(0, 0, 255);
```

Note that in both cases the arguments are interpreted as *nonlinear* sRGB values (R', G', B') . Other constructor methods exist for class

`Color` that also accept alpha (transparency) values. In addition, the `Color` class offers two useful static methods, `RGBtoHSB()` and `HSBtoRGB()`, for converting between sRGB and HSV⁸ colors (see Ch. 12, Sec. 12.2.3).

Class `ColorSpace` (`java.awt.color.ColorSpace`)

An object of type `ColorSpace` represents an entire color space, such as sRGB or CMYK. Every subclass of `ColorSpace` (which itself is an abstract class) provides methods for converting its native colors to the CIEXYZ and sRGB color space and vice versa, such that conversions between arbitrary color spaces can easily be performed (through Java's XYZ-based profile connection space). In the following example, we first create an instance of the default sRGB color space by invoking the static method `ColorSpace.getInstance()` and subsequently convert an sRGB color object (R', G', B') to the corresponding (X, Y, Z) coordinates in Java's (D50-based) profile connection space:

```
// create an sRGB color space object:  
ColorSpace sRGBcsp  
    = ColorSpace.getInstance(ColorSpace.CS_sRGB);  
float[] pink_RGB = new float[] {1.0f, 0.5f, 0.5f};  
// convert from sRGB to XYZ:  
float[] pink_XYZ = sRGBcsp.toCIEXYZ(pink_RGB);
```

Notice that color vectors are represented as `float[]` arrays for color conversions with `ColorSpace` objects. If required, the method `getComponents()` can be used to convert `Color` objects to `float[]` arrays. In summary, the types of color spaces that can be created with the `ColorSpace.getInstance()` method include:

- `CS_sRGB`: the standard (D65-based) RGB color space with *non-linear* R', G', B' components, as specified in [119].
- `CS_LINEAR_RGB`: color space with *linear* R, G, B components (i.e., no gamma correction applied).
- `CS_GRAY`: single-component color space with linear grayscale values.
- `CS_PYCC`: Kodak's Photo YCC color space.
- `CS_CIEXYZ`: the default XYZ profile connection space (based on the D50 white point).

Other color spaces can be implemented by creating additional implementations (subclasses) of `ColorSpace`, as demonstrated for CIELAB in the example in Sec. 14.7.3.

14.7.3 Implementation of the CIELAB Color Space (Example)

In the following, we show a complete implementation of the CIELAB color space, which is not available in the current Java API, based on the specification given in Sec. 14.2. For this purpose, we define a

⁸ The HSV color space is referred to as "HSB" (hue, saturation, *brightness*) in the Java API.

subclass of `ColorSpace` (defined in the package `java.awt.color`) named `Lab_ColorSpace`, which implements the required methods `toCIEXYZ()`, `fromCIEXYZ()` for converting to and from Java's default profile connection space, respectively, and `toRGB()`, `fromRGB()` for converting between CIELAB and sRGB (Progs. 14.1 and 14.2). These conversions are performed in two steps via XYZ coordinates, where care must be taken regarding the right choice of the associated white point (CIELAB is based on D65 and Java XYZ on D50). The following examples demonstrate the principal use of the new `Lab_ColorSpace` class:⁹

```
ColorSpace labCs = new LabColorSpace();
float[] cyan_sRGB = {0.0f, 1.0f, 1.0f};
float[] cyan_LAB = labCs.fromRGB(cyan_sRGB) // sRGB→LAB
float[] cyan_XYZ = labCs.toXYZ(cyan_LAB);    // LAB→XYZ (D50)
```

14.7.4 ICC Profiles

Even with the most precise specification, a standard color space may not be sufficient to accurately describe the transfer characteristics of some input or output device. ICC¹⁰ profiles are standardized descriptions of individual device transfer properties that warrant that an image or graphics can be reproduced accurately on different media. The contents and the format of ICC profile files is specified in [118], which is identical to ISO standard 15076 [121]. Profiles are thus a key element in the process of digital color management [246].

The Java graphics API supports the use of ICC profiles mainly through the classes `ICC_ColorSpace` and `ICC_Profile`, which allow application designers to create various standard profiles and read ICC profiles from data files.

Assume, for example, that an image was recorded with a calibrated scanner and shall be displayed accurately on a monitor. For this purpose, we need the ICC profiles for the scanner and the monitor, which are often supplied by the manufacturers as `.icc` data files.¹¹ For standard color spaces, the associated ICC profiles are often available as part of the computer installation, such as `CIERGB.icc` or `NTSC1953.icc`. With these profiles, a color space object can be specified that converts the image data produced by the scanner into corresponding CIEXYZ or sRGB values, as illustrated by the following example:

```
// load the scanner's ICC profile and create a corresponding color space:
ICC_ColorSpace scannerCs = new
ICC_ColorSpace(ICC_ProfileRGB.getInstance("scanner.icc"));

// specify a device-specific color:
float[] deviceColor = {0.77f, 0.13f, 0.89f};
```

⁹ Classes `LabColorSpace`, `LuvColorSpace` (analogous implementation of the CIELUV color space) and associated auxiliary classes are found in package `imagingbook.pub.colorimage`.

¹⁰ International Color Consortium ICC (www.color.org).

¹¹ ICC profile files may also come with extensions `.icm` or `.pf` (as in the Java distribution).

```

1 package imagingbook.pub.color.image;
2
3 import static imagingbook.pub.color.image.Illuminant.D50;
4 import static imagingbook.pub.color.image.Illuminant.D65;
5
6 import java.awt.color.ColorSpace;
7
8 public class LabColorSpace extends ColorSpace {
9
10 // D65 reference white point and chromatic adaptation objects:
11 static final double Xref = D65.X; // 0.950456
12 static final double Yref = D65.Y; // 1.000000
13 static final double Zref = D65.Z; // 1.088754
14
15 static final ChromaticAdaptation catD65toD50 =
16     new BradfordAdaptation(D65, D50);
17 static final ChromaticAdaptation catD50toD65 =
18     new BradfordAdaptation(D50, D65);
19
20 // the only constructor:
21 public LabColorSpace() {
22     super(TYPE_Lab,3);
23 }
24
25 // XYZ (Profile Connection Space, D50) → CIELab conversion:
26 public float[] fromCIEXYZ(float[] XYZ50) {
27     float[] XYZ65 = catD50toD65.apply(XYZ50);
28     return fromCIEXYZ65(XYZ65);
29 }
30
31 // XYZ (D65) → CIELab conversion (Eqn. (14.6)–14.10):
32 public float[] fromCIEXYZ65(float[] XYZ65) {
33     double xx = f1(XYZ65[0] / Xref);
34     double yy = f1(XYZ65[1] / Yref);
35     double zz = f1(XYZ65[2] / Zref);
36     float L = (float)(116.0 * yy - 16.0);
37     float a = (float)(500.0 * (xx - yy));
38     float b = (float)(200.0 * (yy - zz));
39     return new float[] {L, a, b};
40 }
41 // CIELab→XYZ (Profile Connection Space, D50) conversion:
42 public float[] toCIEXYZ(float[] Lab) {
43     float[] XYZ65 = toCIEXYZ65(Lab);
44     return catD65toD50.apply(XYZ65);
45 }
46
47 // CIELab→XYZ (D65) conversion (Eqn. (14.13)–14.15):
48 public float[] toCIEXYZ65(float[] Lab) {
49     double ll = (Lab[0] + 16.0) / 116.0;
50     float Y65 = (float) (Yref * f2(ll));
51     float X65 = (float) (Xref * f2(ll + Lab[1] / 500.0));
52     float Z65 = (float) (Zref * f2(ll - Lab[2] / 200.0));
53     return new float[] {X65, Y65, Z65};
54 }

```

14.7 COLORIMETRIC SUPPORT IN JAVA

Prog. 14.1

Java implementation of the CIELAB color space as a sub-class of `ColorSpace` (part 1). The conversion from D50-based profile connection space XYZ coordinates to CIELAB (Eqn. (14.6)) and back is implemented by the required methods `fromCIEXYZ()` and `toCIEXYZ()`, respectively. The auxiliary methods `fromCIEXYZ65()` and `toCIEXYZ65()` are used for converting D65-based XYZ coordinates (see Eqn. (14.6)). Chromatic adaptation from D50 to D65 is performed by the objects `catD65toD50` and `catD50toD65` of type `ChromaticAdaptation`. The gamma correction functions f_1 (Eqn. (14.8)) and f_2 (Eqn. (14.15)) are implemented by the methods `f1()` and `f2()`, respectively (see Prog. 14.2).

14 COLORIMETRIC COLOR SPACES

Prog. 14.2

Java implementation of the CIELAB color space as a subclass of *ColorSpace* (part 2). The methods *fromRGB()* and *toRGB()* perform direct conversion between CIELAB and sRGB via D65-based XYZ coordinates, i.e., without conversion to Java's *Profile Connection Space*. Gamma correction (for mapping between linear RGB and sRGB component values) is implemented by the methods *gammaFwd()* and *gammaInv()* in class *sRgbUtil* (not shown). The methods *f1()* and *f2()* implement the forward and inverse gamma correction of CIELAB components (see Eqns. (14.6) and (14.13)).

```
55 // sRGB→CIELab conversion:  
56 public float[] fromRGB(float[] srgb) {  
57     // get linear rgb components:  
58     double r = sRgbUtil.gammaInv(srgb[0]);  
59     double g = sRgbUtil.gammaInv(srgb[1]);  
60     double b = sRgbUtil.gammaInv(srgb[2]);  
61     // convert to XYZ (D65-based, Eqn. (14.29)):  
62     float X =  
63         (float) (0.412453 * r + 0.357580 * g + 0.180423 * b);  
64     float Y =  
65         (float) (0.212671 * r + 0.715160 * g + 0.072169 * b);  
66     float Z =  
67         (float) (0.019334 * r + 0.119193 * g + 0.950227 * b);  
68     float[] XYZ65 = new float[] {X, Y, Z};  
69     return fromCIEXYZ65(XYZ65);  
70 }  
71  
72 // CIELab→sRGB conversion:  
73 public float[] toRGB(float[] Lab) {  
74     float[] XYZ65 = toCIEXYZ65(Lab);  
75     double X = XYZ65[0];  
76     double Y = XYZ65[1];  
77     double Z = XYZ65[2];  
78     // XYZ→RGB (linear components, Eqn. (14.28)):  
79     double r = ( 3.240479*X + -1.537150*Y + -0.498535*Z);  
80     double g = (-0.969256*X + 1.875992*Y + 0.041556*Z);  
81     double b = ( 0.055648*X + -0.204043*Y + 1.057311*Z);  
82     // RGB→sRGB (nonlinear components):  
83     float rr = (float) sRgbUtil.gammaFwd(r);  
84     float gg = (float) sRgbUtil.gammaFwd(g);  
85     float bb = (float) sRgbUtil.gammaFwd(b);  
86     return new float[] {rr, gg, bb};  
87 }  
88  
89 static final double epsilon = 216.0 / 24389; // Eqn. (14.9)  
90 static final double kappa = 841.0 / 108; // Eqn. (14.10)  
91  
92 // Gamma correction for L* (forward, Eqn. (14.8)):  
93 double f1 (double c) {  
94     if (c > epsilon) // 0.008856  
95         return Math.cbrt(c);  
96     else  
97         return (kappa * c) + (16.0 / 116);  
98 }  
99  
100 // Gamma correction for L* (inverse, Eqn. (14.15)):  
101 double f2 (double c) {  
102     double c3 = c * c * c;  
103     if (c3 > epsilon)  
104         return c3;  
105     else  
106         return (c - 16.0 / 116) / kappa;  
107 }  
108  
109 } // end of class LabColorSpace
```

```
// convert to sRGB:  
float[] RGBColor = scannerCs.toRGB(deviceColor);  
  
// convert to (D50-based) XYZ:  
float[] XYZColor = scannerCs.toCIEXYZ(deviceColor);
```

Similarly, we can calculate the accurate color values to be sent to the monitor by creating a suitable color space object from this device's ICC profile.

14.8 Exercises

Exercise 14.1. For chromatic adaptation (defined in Eqn. (14.43)), transformation matrices other than the Bradford model (Eqn. (14.45)) have been proposed; for example, [225],

$$\mathbf{M}_{\text{CAT}}^{(2)} = \begin{pmatrix} 1.2694 & -0.0988 & -0.1706 \\ -0.8364 & 1.8006 & 0.0357 \\ 0.0297 & -0.0315 & 1.0018 \end{pmatrix} \quad \text{or} \quad (14.50)$$

$$\mathbf{M}_{\text{CAT}}^{(3)} = \begin{pmatrix} 0.7982 & 0.3389 & -0.1371 \\ -0.5918 & 1.5512 & 0.0406 \\ 0.0008 & -0.0239 & 0.9753 \end{pmatrix}. \quad (14.51)$$

Derive the complete chromatic adaptation transformations $\mathbf{M}_{50|65}$ and $\mathbf{M}_{65|50}$ for converting between D65 and D50 colors, analogous to Eqns. (14.46) and (14.47), for each of the above transformation matrices.

Exercise 14.2. Implement the conversion of an sRGB color image to a colorless (grayscale) sRGB image using the three methods in Eqn. (14.37) (incorrectly applying standard weights to nonlinear $R'G'B'$ components), Eqn. (14.38) (exact computation), and Eqn. (14.39) (approximation using nonlinear components and modified weights). Compare the results by computing difference images, and also determine the total errors.

Exercise 14.3. Write a program to evaluate the errors that are introduced by using *nonlinear* instead of linear color components for grayscale conversion. To do this, compute the difference between the Y values obtained with the linear variant (Eqn. (14.38)) and the nonlinear variant (Eqn. (14.39) with $w'_R = 0.309$, $w'_G = 0.609$, $w'_B = 0.082$) for all possible 2^{24} RGB colors. Let your program return the maximum gray value difference and the sum of the absolute differences for all colors.

Exercise 14.4. Determine the virtual primaries \mathbf{R}^* , \mathbf{G}^* , \mathbf{B}^* obtained by Bradford adaptation (Eqn. (14.42)), with \mathbf{M}_{CAT} as defined in Eqn. (14.45). What are the resulting coordinates in the xy chromaticity diagram? Are the primaries inside the visible color range?

Filters for Color Images

Color images are everywhere and filtering them is such a common task that it does not seem to require much attention at all. In this chapter, we describe how classical linear and nonlinear filters, which we covered before in the context of grayscale images (see Ch. 5), can be either used directly or adapted for the processing of color images. Often color images are treated as stacks of intensity images and existing monochromatic filters are simply applied independently to the individual color channels. While this is straightforward and performs satisfactorily in many situations, it does not take into account the vector-valued nature of color pixels as samples taken in a specific, multi-dimensional color space. As we show in this chapter, the outcome of filter operations depends strongly on the working color space and the variations between different color spaces may be substantial. Although this may not be apparent in many situations, it should be of concern if high-quality color imaging is an issue.

15.1 Linear Filters

Linear filters are important in many applications, such as smoothing, noise removal, interpolation for geometric transformations, decimation in scale-space transformations, image compression, reconstruction and edge enhancement. The general properties of linear filters and their use on scalar-valued grayscale images are detailed in Chapter 5, Sec. 5.2. For color images, it is common practice to apply these monochromatic filters separately to each color channel, thereby treating the image as a stack of scalar-valued images. As we describe in the following section, this approach is simple as well as efficient, since existing implementations for grayscale images can be reused without any modification. However, the outcome depends strongly on the choice of the color space in which the filter operation is performed. For example, it makes a great difference if the channels of an RGB image contain linear or nonlinear component values. This issue is discussed in more detail in Sec. 15.1.2.

15.1.1 Monochromatic Application of Linear Filters

Given a discrete *scalar* (grayscale) image with elements $I(u, v) \in \mathbb{R}$, the application of a linear filter can be expressed as a linear 2D convolution¹

$$\bar{I}(u, v) = (I * H)(u, v) = \sum_{(i,j) \in \mathcal{R}_H} I(u-i, v-j) \cdot H(i, j), \quad (15.1)$$

where H denotes a discrete filter kernel defined over the (usually rectangular) region \mathcal{R}_H , with $H(i, j) \in \mathbb{R}$. For a *vector*-valued image \mathbf{I} with K components, the individual picture elements are vectors, that is,

$$\mathbf{I}(u, v) = \begin{pmatrix} I_1(u, v) \\ I_2(u, v) \\ \vdots \\ I_K(u, v) \end{pmatrix}, \quad (15.2)$$

with $\mathbf{I}(u, v) \in \mathbb{R}^K$ or $I_k(u, v) \in \mathbb{R}$, respectively. In this case, the linear filter operation can be generalized to

$$\bar{\mathbf{I}}(u, v) = (\mathbf{I} * H)(u, v) = \sum_{(i,j) \in \mathcal{R}_H} \mathbf{I}(u-i, v-j) \cdot H(i, j), \quad (15.3)$$

with the same scalar-valued filter kernel H as in Eqn. (15.1). Thus the k th element of the resulting pixels,

$$\bar{I}_k(u, v) = \sum_{(i,j) \in \mathcal{R}_H} I_k(u-i, v-j) \cdot H(i, j) = (I_k * H)(u, v), \quad (15.4)$$

is simply the result of scalar convolution (Eqn. (15.1)) applied to the corresponding component plane I_k . In the case of an RGB color image (with $K = 3$ components), the filter kernel H is applied separately to the scalar-valued R , G , and B planes (I_R, I_G, I_B), that is,

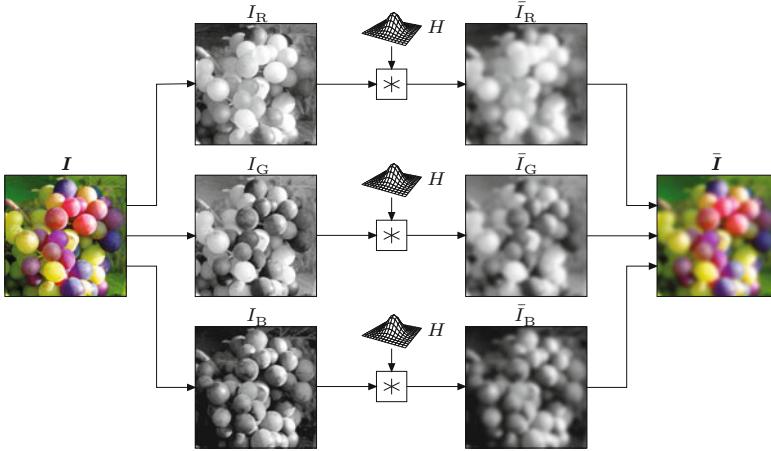
$$\bar{\mathbf{I}}(u, v) = \begin{pmatrix} \bar{I}_R(u, v) \\ \bar{I}_G(u, v) \\ \bar{I}_B(u, v) \end{pmatrix} = \begin{pmatrix} (I_R * H)(u, v) \\ (I_G * H)(u, v) \\ (I_B * H)(u, v) \end{pmatrix}. \quad (15.5)$$

Figure 15.1 illustrates how linear filters for color images are typically implemented by individually filtering the three scalar-valued color components.

Linear smoothing filters

Smoothing filters are a particular class of linear filters that are found in many applications and characterized by positive-only filter coefficients. Let $C_{u,v} = (c_1, \dots, c_n)$ denote the vector of color pixels $c_m \in \mathbb{R}^K$ contained in the spatial support region of the kernel H , placed at position (u, v) in the original image \mathbf{I} , where n is the size of H . With arbitrary kernel coefficients $H(i, j) \in \mathbb{R}$, the resulting

¹ See also Chapter 5, Sec. 5.3.1.

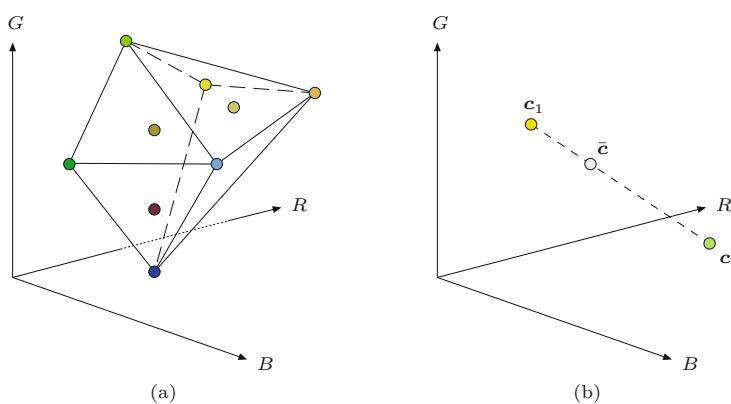


color pixel $\bar{I}(u, v) = \bar{c}$ in the filtered image is a *linear combination* of the original colors in $C_{u,v}$, that is,

$$\bar{c} = w_1 \cdot c_1 + w_2 \cdot c_2 + \cdots + w_n \cdot c_n = \sum_{i=1}^n w_i \cdot c_i, \quad (15.6)$$

where w_m is the coefficient in H that corresponds to pixel c_m . If the kernel is *normalized* (i.e., $\sum H(i, j) = \sum \alpha_m = 1$), the result is an *affine combination* of the original colors. In case of a typical smoothing filter, with H normalized and all coefficients $H(i, j)$ being *positive*, any resulting color \bar{c} is a *convex combination* of the original color vectors c_1, \dots, c_n .

Geometrically this means that the mixed color \bar{c} is contained within the *convex hull* of the contributing colors c_1, \dots, c_n , as illustrated in Fig. 15.2. In the special case that only *two* original colors c_1, c_2 are involved, the result \bar{c} is located on the straight line segment connecting c_1 and c_2 (Fig. 15.2(b)).²



15.1 LINEAR FILTERS

Fig. 15.1

Monochromatic application of a linear filter. The filter, specified by the kernel H , is applied separately to each of the scalar-valued color channels I_R, I_G, I_B . Combining the filtered component channels $\bar{I}_R, \bar{I}_G, \bar{I}_B$ produces the filtered color image \bar{I} .

Fig. 15.2

Convex linear color mixtures. The result of the convex combination (mixture) of n color vectors $\mathcal{C} = \{c_1, \dots, c_n\}$ is confined to the convex hull of \mathcal{C} (a). In the special case of only two initial colors c_1 and c_2 , any mixed color \bar{c} is located on the straight line segment connecting c_1 and c_2 (b).

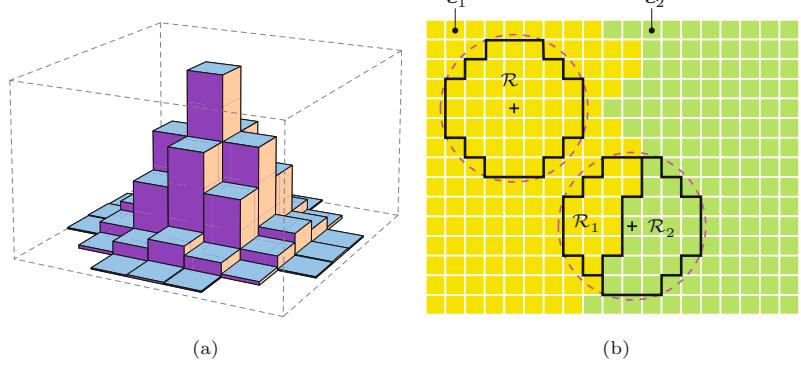
² The convex hull of *two* points c_1, c_2 consists of the straight line segment between them.

Fig. 15.3

Linear smoothing filter at a color edge. Discrete filter kernel with positive-only elements and support region \mathcal{R} (a). Filter kernel positioned over a region of constant color c_1 and over a color step edge c_1/c_2 , respectively (b). If the (normalized) filter kernel of extent

\mathcal{R} is completely embedded in a region of constant color (c_1), the result of filtering is exactly that same color. At a step edge between two colors c_1, c_2 , one part of the kernel (\mathcal{R}_1) covers pixels of color c_1 and the remaining part (\mathcal{R}_2)

covers pixels of color c_2 . In this case, the result is a *linear mixture* of the colors c_1, c_2 , as illustrated in Fig. 15.2(b).



Response to a color step edge

Assume, as a special case, that the original RGB image \mathbf{I} contains a *step edge* separating two regions of constant colors c_1 and c_2 , respectively, as illustrated in Fig. 15.3(b). If the normalized smoothing kernel H is placed at some position (u, v) , where it is fully supported by pixels of identical color c_1 , the (trivial) response of the filter is

$$\bar{\mathbf{I}}(u, v) = \sum_{(i,j) \in \mathcal{R}_H} c_1 \cdot H(i, j) = c_1 \cdot \sum_{(i,j) \in \mathcal{R}_H} H(i, j) = c_1 \cdot 1 = c_1. \quad (15.7)$$

Thus the result at this position is the original color c_1 . Alternatively, if the filter kernel is placed at some position *on* a color edge (between two colors c_1, c_2 , see again Fig. 15.3(b)), a subset of its coefficients (\mathcal{R}_1) is supported by pixels of color c_1 , while the other coefficients (\mathcal{R}_2) overlap with pixels of color c_2 . Since $\mathcal{R}_1 \cup \mathcal{R}_2 = \mathcal{R}$ and the kernel is normalized, the resulting color is

$$\bar{c} = \sum_{(i,j) \in \mathcal{R}_1} c_1 \cdot H(i, j) + \sum_{(i,j) \in \mathcal{R}_2} c_2 \cdot H(i, j) \quad (15.8)$$

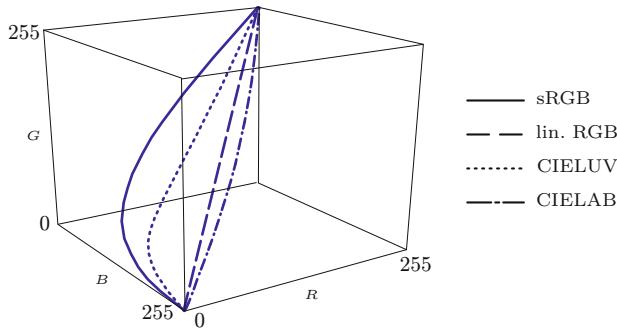
$$= c_1 \cdot \underbrace{\sum_{(i,j) \in \mathcal{R}_1} H(i, j)}_{1-s} + c_2 \cdot \underbrace{\sum_{(i,j) \in \mathcal{R}_2} H(i, j)}_s \quad (15.9)$$

$$= c_1 \cdot (1-s) + c_2 \cdot s = c_1 + s \cdot (c_2 - c_1), \quad (15.10)$$

for some $s \in [0, 1]$. As we see, the resulting color coordinate \bar{c} lies on the straight line segment connecting the original colors c_1 and c_2 in the respective color space. Thus, at a step edge between two colors c_1, c_2 , the intermediate colors produced by a (normalized) smoothing filter are located on the straight line between the two original color coordinates. Note that this relationship between linear filtering and linear color mixtures is independent of the particular color space in which the operation is performed.

15.1.2 Color Space Considerations

Since a linear filter always yields a convex linear mixture of the involved colors it should make a difference in which color space the



15.1 LINEAR FILTERS

Fig. 15.4

Intermediate colors produced by linear interpolation between *yellow* and *blue*, performed in different color spaces. The 3D plot shows the resulting colors in linear RGB space.

filter operation is performed. For example, Fig. 15.4 shows the intermediate colors produced by a smoothing filter being applied to the same blue/yellow step edge but in different color spaces: sRGB, linear RGB, CIELUV, and CIELAB. As we see, the differences between the various color spaces are substantial. To obtain dependable and standardized results it might be reasonable to first transform the input image to a particular operating color space, perform the required filter operation, and finally transform the result back to the original color space, as illustrated in Fig. 15.5.

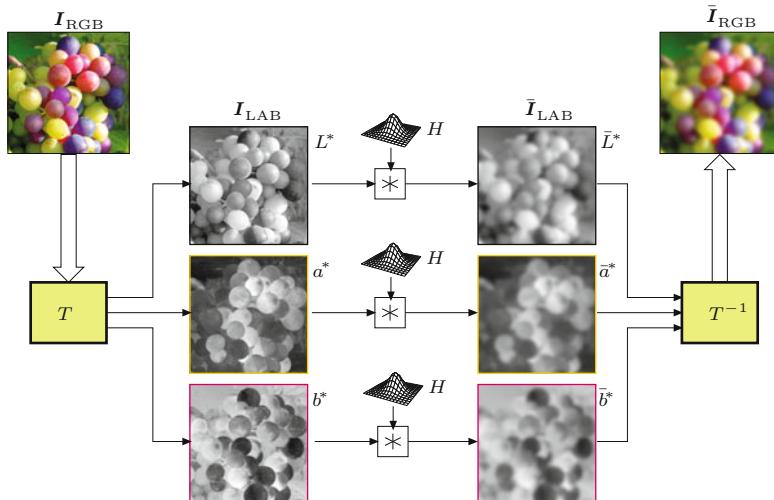


Fig. 15.5

Linear filter operation performed in a “foreign” color space. The original RGB image I_{RGB} is first transformed to CIELAB (by T), where the linear filter is applied separately to the three channels L^* , a^* , b^* . The filtered RGB image \bar{I}_{RGB} is obtained by transforming back from CIELAB to RGB (by T^{-1}).

Obviously, a linear filter implies certain “metric” properties of the underlying color space. If we assume that a certain color space S_A has this property, then this is also true for any color space S_B that is related to S_A by a linear transformation, such as CIEXYZ and linear RGB space (see Ch. 14, Sec. 14.4.1). However, many color spaces used in practice (sRGB in particular) are related to these reference color spaces by highly nonlinear mappings, and thus significant deviations can be expected.

Preservation of brightness (luminance)

Apart from the intermediate colors produced by interpolation, another important (and easily measurable) aspect is the resulting change of *brightness* or *luminance* across the filter region. In par-

ticular it should generally hold that the luminance of the filtered color image is identical to the result of filtering only the (scalar) luminance channel of the original image with the same kernel H . Thus, if $\text{Lum}(\mathbf{I})$ denotes the luminance of the original color image and $\text{Lum}(\mathbf{I} * H)$ is the luminance of the filtered image, it should hold that

$$\text{Lum}(\mathbf{I} * H) = \text{Lum}(\mathbf{I}) * H. \quad (15.11)$$

This is only possible if $\text{Lum}(\cdot)$ is linearly related to the components of the associated color space, which is mostly not the case. From Eqn. (15.11) we also see that, when filtering a step edge with colors \mathbf{c}_1 and \mathbf{c}_2 , the resulting brightness should also change *monotonically* from $\text{Lum}(\mathbf{c}_1)$ to $\text{Lum}(\mathbf{c}_2)$ and, in particular, none of the intermediate brightness values should fall outside this range.

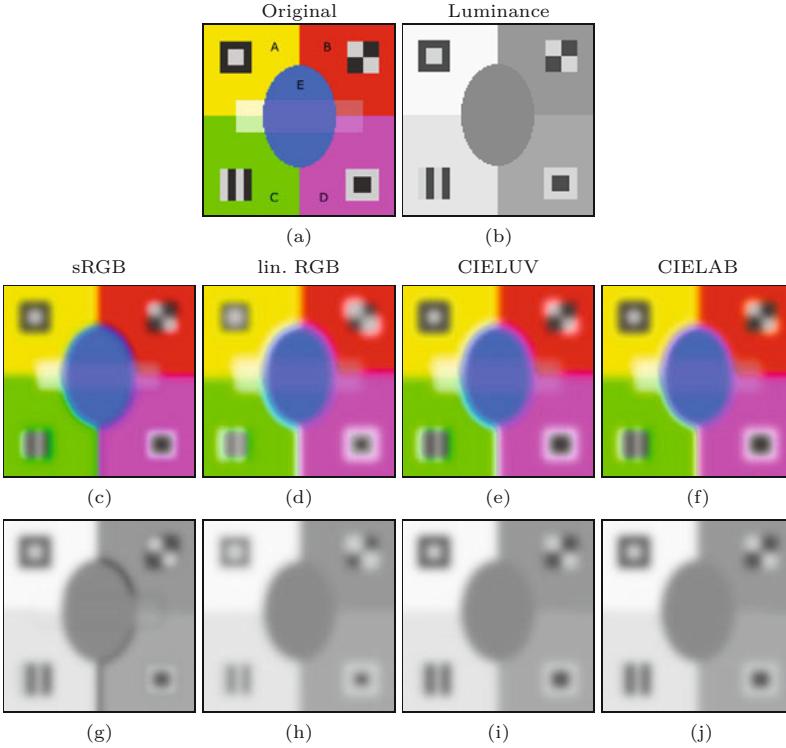
Figure 15.6 shows the results of filtering a synthetic test image with a normalized Gaussian kernel (of radius $\sigma = 3$) in different color spaces. Differences are most notable at the *red–blue* and *green–magenta* transitions, with particularly large deviations in the sRGB space. The corresponding luminance values Y (calculated from linear RGB components as in Eqn. (12.35)) are shown in Fig. 15.6(g–j). Again conspicuous is the result for sRGB (Fig. 15.6(c,g)), which exhibits transitions at the *red–blue*, *magenta–blue*, and *magenta–green* edges, where the resulting brightness drops below the original brightness of both contributing colors. Thus Eqn. (15.11) is not satisfied in this case. On the other hand, filtering in linear RGB space has the tendency to produce too high brightness values, as can be seen at the *black–white* markers in Fig. 15.6(d,h).

Out-of-gamut colors

If we apply a linear filter in RGB or sRGB space, the resulting intermediate colors are always valid RGB colors again and contained in the original RGB gamut volume. However, transformed to CIELUV or CIELAB, the set of possible RGB or sRGB colors forms a non-convex shape (see Ch. 14, Fig. 14.5), such that linearly interpolated colors may fall outside the RGB gamut volume. Particularly critical (in both CIELUV and CIELAB) are the *red–white*, *red–yellow*, and *red–magenta* transitions, as well as *yellow–green* in CIELAB, where the resulting distances from the gamut surface can be quite large (see Fig. 15.7). During back-transformation to the original color space, such “out-of-gamut” colors must receive special treatment, since simple clipping of the affected components may cause unacceptable color distortions [167].

Implications and further reading

Applying a linear filter to the individual component channels of a color image presumes a certain “linearity” of the underlying color space. Smoothing filters implicitly perform additive linear mixing and interpolation. Despite common practice (and demonstrated by the results), there is no justification for performing a linear filter operation directly on gamma-mapped sRGB components. However, contrary to expectation, filtering in linear RGB does not yield better



15.1 LINEAR FILTERS

Fig. 15.6

Gaussian smoothing performed in different color spaces. Synthetic color image (a) and corresponding luminance image (b). The test image contains a horizontal bar with reduced color saturation but the same luminance as its surround, i.e., it is invisible in the luminance image. Gaussian filter applied in different color spaces: sRGB (c), linear RGB (d), CIELUV (e), and CIELAB (f). The bottom row (g–j) shows the corresponding luminance (Y) images. Note the dark bands in the sRGB result (b), particularly along the color boundaries between regions B–E, C–D, and D–E, which stand out clearly in the corresponding luminance image (g). Filtering in linear RGB space (d, h) gives good results between highly saturated colors, but subjectively too high luminance in unsaturated regions, which is apparent around the gray markers. Results with CIELUV (e, i) and CIELAB color spaces (f, j) appear most consistent as far as the preservation of luminance is concerned.

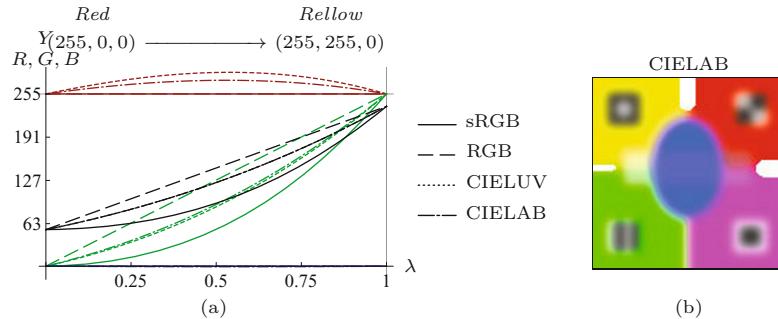


Fig. 15.7

Out-of-gamut colors produced by linear interpolation between *red* and *yellow* in “foreign” color spaces. The graphs in (a) show the (linear) R , G , B component values and the luminance Y (gray curves) resulting from a linear filter between *red* and *yellow* performed in different color spaces. The graphs show that the red component runs significantly outside the RGB gamut for both CIELUV and CIELAB. In (b) all pixels with any component outside the RGB gamut by more than 1% are marked white (for filtering in CIELAB).

overall results either. In summary, both nonlinear sRGB and linear RGB color spaces are unsuitable for linear filtering if perceptually accurate results are desired. Perceptually uniform color spaces, such as CIELUV and CIELAB, are good choices for linear filtering because of their metric properties, with CIELUV being perhaps slightly superior when it comes to interpolation over large color distances. When using CIELUV or CIELAB as intermediate color spaces for filtering RGB images, one must consider that out-of-gamut colors may be produced that must be handled properly. Thus none of the existing standard color spaces is universally suited or even “ideal” with respect to linear filtering.

The proper choice of the working color space is relevant not only to smoothing filters, but also to other types of filters, such as linear interpolation filters for geometric image transformations, decimation filters used in multi-scale techniques, and also nonlinear filters that

involve averaging colors or calculation of color distances, such as the vector median filter (see Sec. 15.2.2). While complex color space transformations in the context of filtering (e.g., sRGB \leftrightarrow CIELUV) are usually avoided for performance reasons, they should certainly be considered when high-quality results are important.

Although the issues related to color mixtures and interpolation have been investigated for some time (see, e.g., [149, 258]), their relevance to image filtering has not received much attention in the literature. Most image processing tools (including commercial software) apply linear filters directly to color images, without proper linearization or color space conversion. Lindbloom [149] was among the first to describe the problem of accurate color reproduction, particularly in the context of computer graphics and photo-realistic imagery. He also emphasized the relevance of perceptual uniformity for color processing and recommended the use of CIELUV as a suitable (albeit not perfect) processing space. Tomasi and Manduchi [229] suggested the use of the Euclidean distance in CIELAB space as “most natural” for bilateral filtering applied to color images (see also Ch. 17, Sec. 17.2) and similar arguments are put forth in [109]. De Weijer [239] notes that the additional chromaticities introduced by linear smoothing are “visually unacceptable” and argues for the use of nonlinear operators as an alternative. Lukac et al. [156] mention “certain inaccuracies” and color artifacts related to the application of scalar filters and discuss the issue of choosing a proper distance metric for vector-based filters. The practical use of alternative color spaces for image filtering is described in [141, Ch. 5].

15.1.3 Linear Filtering with Circular Values

If any of the color components is a *circular* quantity, such as the hue component in the HSV and HLS color spaces (see Ch. 12, Sec. 12.2.3), linear filters cannot be applied directly without additional provisions. As described in the previous section, a linear filter effectively calculates a weighted average over the values inside the filter region. Since the hue component represents a revolving angle and exhibits a discontinuity at the $1 \rightarrow 0$ (i.e., $360^\circ \rightarrow 0^\circ$) transition, simply averaging this quantity is not admissible (see Fig. 15.8).

However, correct interpolation of angular data is possible by utilizing the corresponding cosine and sine values, without any special treatment of discontinuities [69]. Given two angles α_1, α_2 , the average angle α_{12} can be calculated as³

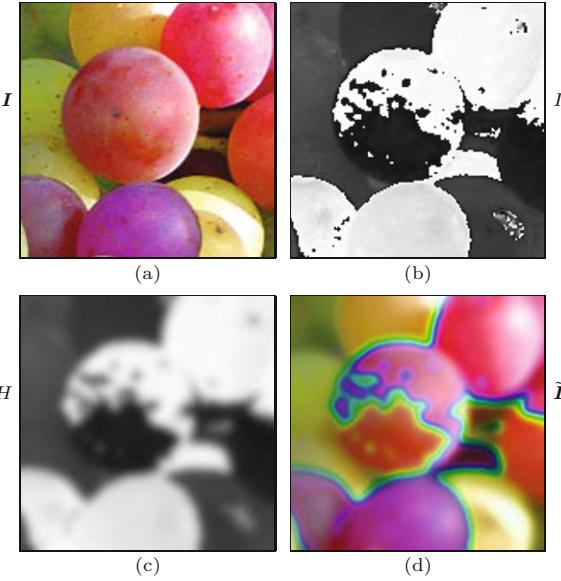
$$\alpha_{12} = \tan^{-1}\left(\frac{\sin(\alpha_1) + \sin(\alpha_2)}{\cos(\alpha_1) + \cos(\alpha_2)}\right) \quad (15.12)$$

$$= \text{ArcTan}(\cos(\alpha_1) + \cos(\alpha_2), \sin(\alpha_1) + \sin(\alpha_2)) \quad (15.13)$$

and, in general, multiple angular values $\alpha_1, \dots, \alpha_n$ can be correctly averaged in the form

$$\bar{\alpha} = \text{ArcTan}\left(\sum_{i=1}^n \cos(\alpha_i), \sum_{i=1}^n \sin(\alpha_i)\right). \quad (15.14)$$

³ See Sec. A.1 in the Appendix for the definition of the ArcTan() function.



15.1 LINEAR FILTERS

Fig. 15.8
Naive linear filtering in HSV color space. Original RGB color image (a) and the associated HSV hue component I_h (b), with values in the range $[0, 1]$. Hue component after direct application of a Gaussian blur filter H with $\sigma = 3.0$ (c). Reconstructed RGB image \bar{I} after filtering all components in HSV space (d). Note the false colors introduced around the $0 \rightarrow 1$ discontinuity (near red) of the hue component.

Also, the calculation of a *weighted* average is possible in the same way, that is,

$$\bar{\alpha} = \text{ArcTan}\left(\sum_{i=1}^n w_i \cdot \cos(\alpha_i), \sum_{i=1}^n w_i \cdot \sin(\alpha_i)\right), \quad (15.15)$$

without any additional provisions, even the weights w_i need not be normalized. This approach can be used for linearly filtering circular data in general.

Filtering the hue component in HSV color space

To apply a linear filter H to the circular hue component I_h (with original values in $[0, 1]$) of a HSV or HLS image (see Ch. 12, Sec. 12.2.3), we first calculate the corresponding cosine and sine parts I_h^{\sin} and I_h^{\cos} by

$$\begin{aligned} I_h^{\sin}(u, v) &= \sin(2\pi \cdot I_h(u, v)), \\ I_h^{\cos}(u, v) &= \cos(2\pi \cdot I_h(u, v)), \end{aligned} \quad (15.16)$$

with resulting values in the range $[-1, 1]$. These are then filtered individually, that is,

$$\begin{aligned} \bar{I}_h^{\sin} &= I_h^{\sin} * H, \\ \bar{I}_h^{\cos} &= I_h^{\cos} * H. \end{aligned} \quad (15.17)$$

Finally, the filtered hue component \bar{I}_h is obtained in the form

$$\bar{I}_h(u, v) = \frac{1}{2\pi} \cdot [\text{ArcTan}(\bar{I}_h^{\cos}(u, v), \bar{I}_h^{\sin}(u, v)) \bmod 2\pi], \quad (15.18)$$

with values again in the range $[0, 1]$.

Fig. 15.9 demonstrates the correct application of a Gaussian smoothing filter to the hue component of an HSV color image by

15 FILTERS FOR COLOR IMAGES

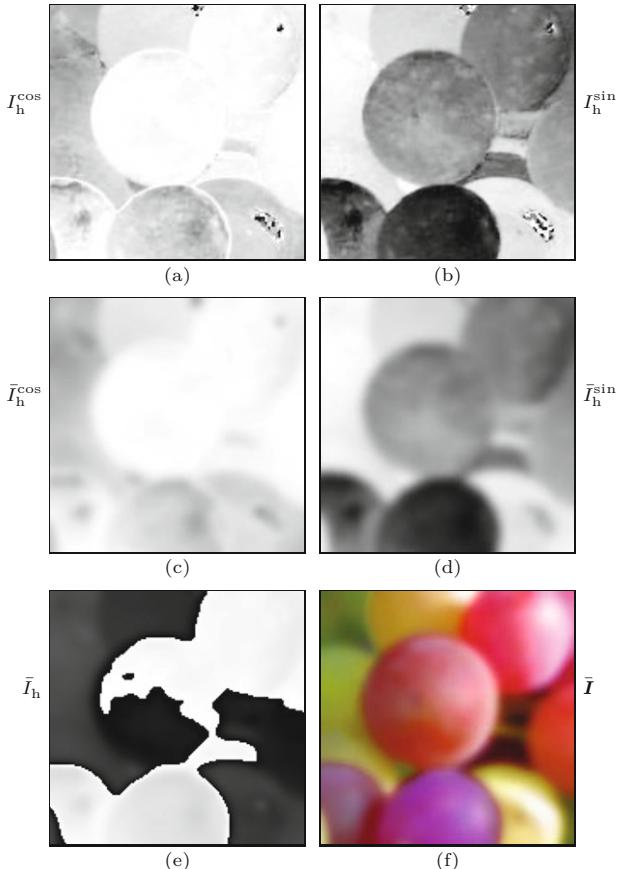
Fig. 15.9

Correct filtering of the HSV hue component by separation into cosine and sine parts (see Fig. 15.8(a) for the original image).

Cosine and sine parts I_h^{\sin} , I_h^{\cos} of the hue component before (a, b) and after the application of a Gaussian blur filter with $\sigma = 3.0$ (c, d). Smoothed hue component \bar{I}_h after merging the filtered cosine and sine parts \bar{I}_h^{\sin} , \bar{I}_h^{\cos} (e). Reconstructed RGB image \bar{I} after filtering all HSV components (f). It is apparent that the hard 0/1 hue transitions in (e) are in fact only gradual color changes around the red hues.

The other HSV components (S, V , which are non-circular) were filtered in the usual way.

The reconstructed RGB image (f) shows no false colors and all hues correctly filtered.



separation into cosine and sine parts. The other two HSV components (S, V) are non-circular and were filtered as usual. In contrast to the result in Fig. 15.8(d), no false colors are produced at the $0 \rightarrow 1$ boundary. In this context it is helpful to look at the *distribution* of the hue values, which are clustered around 0/1 in the sample image (see Fig. 15.10(a)). In Fig. 15.10(b) we can clearly see how naive filtering of the hue component produces new (false) colors in the middle of the histogram. This does not occur when the hue component is filtered correctly (see Fig. 15.10(c)).

Saturation-weighted filtering

The method just described does not take into account that in HSV (and HLS) the hue and saturation components are closely related. In particular, the hue angle may be very inaccurate (or even indeterminate) if the associated saturation value goes to zero. For example, the test image in Fig. 15.8(a) contains a bright patch in the lower right-hand corner, where the *saturation* is low and the *hue* value is quite unstable, as seen in Fig. 15.9(a, b). However, the circular filter defined in Eqns. (15.16)–(15.18) takes all color samples as equally significant.

A simple solution is to use the saturation value $I_s(u, v)$ as a weight factor for the associated pixel [98], by modifying Eqn. (15.16) to

1: **HsvLinearFilter**(I_{hsv} , H)

Input: $I_{\text{hsv}} = (I_h, I_s, I_v)$, a HSV color image of size $M \times N$, with all components in $[0, 1]$; H , a 2D filter kernel. Returns a new (filtered) HSV color image of size $M \times N$.

2: $(M, N) \leftarrow \text{Size}(I_{\text{hsv}})$

3: Create 2D maps $I_h^{\sin}, I_h^{\cos}, \bar{I}_h : M \times N \mapsto \mathbb{R}$

Split the hue channel into sine/cosine parts:

4: **for all** $(u, v) \in M \times N$ **do**

$$5: \quad \theta \leftarrow 2\pi \cdot I_h(u, v) \quad \triangleright \text{hue angle } \theta \in [0, 2\pi]$$

$$6: \quad s \leftarrow I_s(u, v) \quad \triangleright \text{saturation } s \in [0, 1]$$

$$7: \quad I_h^{\sin}(u, v) \leftarrow s \cdot \sin(\theta) \quad \triangleright I_h^{\sin}(u, v) \in [-1, 1]$$

$$8: \quad I_h^{\cos}(u, v) \leftarrow s \cdot \cos(\theta) \quad \triangleright I_h^{\cos}(u, v) \in [-1, 1]$$

Filter all components with the same kernel:

$$9: \quad \bar{I}_h^{\sin} \leftarrow I_h^{\sin} * H$$

$$10: \quad \bar{I}_h^{\cos} \leftarrow I_h^{\cos} * H$$

$$11: \quad \bar{I}_s \leftarrow I_s * H$$

$$12: \quad \bar{I}_v \leftarrow I_v * H$$

Reassemble the filtered hue channel:

13: **for all** $(u, v) \in M \times N$ **do**

$$14: \quad \theta \leftarrow \text{ArcTan}(\bar{I}_h^{\cos}(u, v), \bar{I}_h^{\sin}(u, v)) \quad \triangleright \theta \in [-\pi, \pi]$$

$$15: \quad \bar{I}_h(u, v) \leftarrow \frac{1}{2\pi} \cdot (\theta \bmod 2\pi) \quad \triangleright \bar{I}_h(u, v) \in [0, 1]$$

$$16: \quad \bar{I}_{\text{hsv}} \leftarrow (\bar{I}_h, \bar{I}_s, \bar{I}_v)$$

17: **return** \bar{I}_{hsv}

15.1 LINEAR FILTERS

Alg. 15.1

Linear filtering in HSV color space. All component values of the original HSV image are in the range $[0, 1]$. The algorithm considers the circular nature of the hue component and uses the saturation component (in line 6) as a weight factor, as defined in Eqn. (15.19). The same filter kernel H is applied to all three color components (lines 9–12).

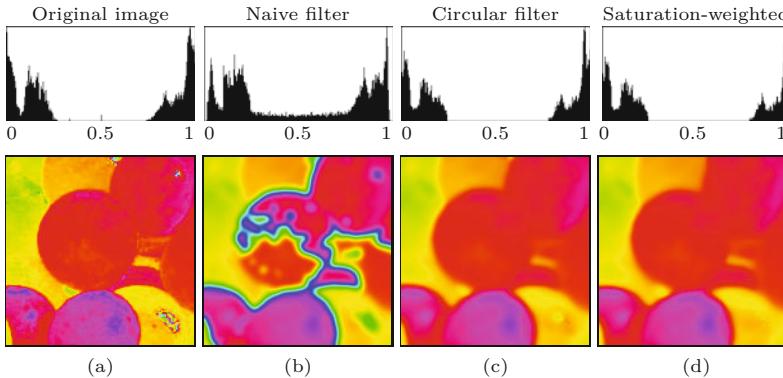


Fig. 15.10

Histogram of the HSV hue component before and after linear filtering. Original distribution of hue values I_h (a), showing that colors are clustered around the $0/1$ discontinuity (red). Result after naive filtering the hue component (b), after filtering separated cosine and sine parts (c), and after addition weighting with saturation values (d). The bottom row shows the isolated hue component (color angle) by the corresponding colors (saturation and value set to 100%). Note the noisy spot in the lower right-hand corner of (a), where color saturation is low and hue angles are very unstable.

$$\begin{aligned} I_h^{\sin}(u, v) &= I_s(u, v) \cdot \sin(2\pi \cdot I_h(u, v)), \\ I_h^{\cos}(u, v) &= I_s(u, v) \cdot \cos(2\pi \cdot I_h(u, v)). \end{aligned} \quad (15.19)$$

All other steps in Eqns. (15.17)–(15.18) remain unchanged. The complete process is summarized in Alg. 15.1. The result in Fig. 15.10(d) shows that, particularly in regions of low color saturation, more stable hue values can be expected. Note that no normalization of the weights is required because the calculation of the hue angles (with the ArcTan() function in Eqn. (15.18)) only considers the ratio of the resulting sine and cosine parts.

15.2 Nonlinear Color Filters

In many practical image processing applications, linear filters are of limited use and nonlinear filters, such as the median filter, are applied instead.⁴ In particular, for effective noise removal, nonlinear filters are usually the better choice. However, as with linear filters, the techniques originally developed for scalar (grayscale) images do not transfer seamlessly to vector-based color data. One reason is that, unlike in scalar data, no natural ordering relation exists for multi-dimensional data. As a consequence, nonlinear filters of the scalar type are often applied separately to the individual color channels, and again one must be cautious about the intermediate colors being introduced by these types of filters.

In the remainder of this section we describe the application of the classic (scalar) median filter to color images, a vector-based version of the median filter, and edge-preserving smoothing filters designed for color images. Additional filters for color images are presented in Chapter 17.

15.2.1 Scalar Median Filter

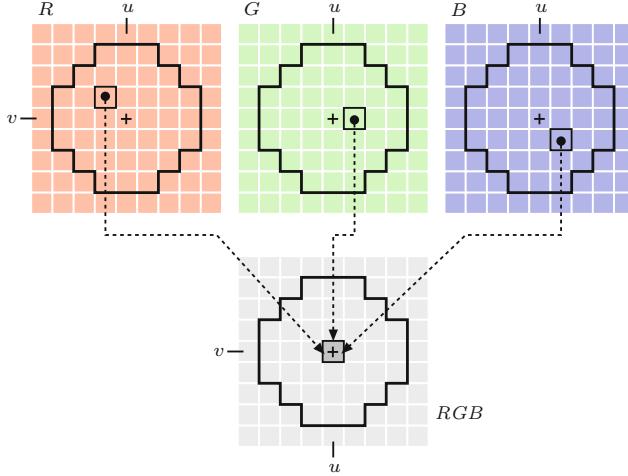
Applying a median filter with support region \mathcal{R} (e.g., a disk-shaped region) at some image position (u, v) means to select one pixel value that is the most representative of the pixels in \mathcal{R} to replace the current center pixel (*hot spot*). In case of a median filter, the statistical *median* of the pixels in \mathcal{R} is taken as that representative. Since we always select the value of one of the existing image pixels, the median filter does not introduce any new pixel values that were not contained in the original image.

If a median filter is applied independently to the components of a color image, each channel is treated as a scalar image, like a single grayscale image. In this case, with the support region \mathcal{R} centered at some point (u, v) , the median for each color channel will typically originate from a *different* spatial position in \mathcal{R} , as illustrated in Fig. 15.11. Thus the components of the resulting color vector are generally collected from more than one pixel in \mathcal{R} , therefore the color placed in the filtered image may not match any of the original colors and new colors may be generated that were not contained in the original image. Despite its obvious deficiencies, the scalar (monochromatic) median filter is used in many popular image processing environments (including Photoshop and ImageJ) as the standard median filter for color images.

15.2.2 Vector Median Filter

The scalar median filter is based on the concept of *rank ordering*, that is, it assumes that the underlying data can be ordered and sorted. However, no such natural ordering exists for data elements that are vectors. Although vectors can be sorted in many different ways, for example by length or lexicographically along their dimensions, it is

⁴ See also Chapter 5, Sec. 5.4.



15.2 NONLINEAR COLOR FILTERS

Fig. 15.11

Scalar median filter applied separately to color channels. With the filter region \mathcal{R} centered at some point (u, v) , the median pixel value is generally found at different locations in the R, G, B channels of the original image. The components of the resulting RGB color vector are collected from spatially separated pixels. It thus may not match any of the colors in the original image.

usually impossible to define a useful greater-than relation between any pair of vectors.

One can show, however, that the median of a sequence of n scalar values $P = (p_1, \dots, p_n)$ can also be defined as the value p_m selected from P , such that

$$\sum_{i=1}^n |p_m - p_i| \leq \sum_{i=1}^n |p_j - p_i|, \quad (15.20)$$

holds for any $p_j \in P$. In other words, the median value $p_m = \text{median}(P)$ is the one for which the sum of the differences to *all other* elements in the sequence P is the smallest.

With this definition, the concept of the median can be easily extended from the scalar situation to the case of multi-dimensional data. Given a sequence of vector-valued samples $\mathbf{P} = (\mathbf{p}_1, \dots, \mathbf{p}_n)$, with $\mathbf{p}_i \in \mathbb{R}^K$, we define the median element \mathbf{p}_m to satisfy

$$\sum_{i=1}^n \|\mathbf{p}_m - \mathbf{p}_i\| \leq \sum_{i=1}^n \|\mathbf{p}_j - \mathbf{p}_i\|, \quad (15.21)$$

for every possible $\mathbf{p}_j \in \mathbf{P}$. This is analogous to Eqn. (15.20), with the exception that the scalar difference $|\cdot|$ has been replaced by the vector norm $\|\cdot\|$ for measuring the distance between two points in the K -dimensional space.⁵ We call

$$D_L(\mathbf{p}, \mathbf{P}) = \sum_{\mathbf{p}_i \in \mathbf{P}} \|\mathbf{p} - \mathbf{p}_i\|_L \quad (15.22)$$

the “aggregate distance” of the sample vector \mathbf{p} with respect to all samples \mathbf{p}_i in \mathbf{P} under the distance norm L . Common choices for the distance norm are the L_1 , L_2 and L_∞ norms, that is,

⁵ K denotes the dimensionality of the samples in \mathbf{p}_i , for example, $K = 3$ for RGB color samples.

Fig. 15.12

Noisy test image (a) with enlarged details (b, c), used in the following examples.



$$L_1: \quad \|p - q\|_1 = \sum_{k=1}^K |p_k - q_k|, \quad (15.23)$$

$$L_2: \quad \|p - q\|_2 = \left[\sum_{k=1}^K |p_k - q_k|^2 \right]^{1/2}, \quad (15.24)$$

$$L_\infty: \quad \|p - q\|_\infty = \max_{1 \leq k \leq K} |p_k - q_k|. \quad (15.25)$$

The vector median of the sequence \mathbf{P} can thus be defined as

$$\text{median}(\mathbf{P}) = \underset{\mathbf{p} \in \mathbf{P}}{\operatorname{argmin}} D_L(\mathbf{p}, \mathbf{P}), \quad (15.26)$$

that is, the sample \mathbf{p} with the smallest aggregate distance to all other elements in \mathbf{P} .

A straight forward implementation of the vector median filter for RGB images is given in Alg. 15.2. The calculation of the aggregate distance $D_L(\mathbf{p}, \mathbf{P})$ is performed by the function `AggregateDistance` (\mathbf{p}, \mathbf{P}). At any position (u, v) , the center pixel is replaced by the neighborhood pixel with the smallest aggregate distance D_{\min} , but only if it is smaller than the center pixel's aggregate distance D_{ctr} (line 15). Otherwise, the center pixel is left unchanged (line 17). This is to prevent that the center pixel is unnecessarily changed to another color, which incidentally has the same aggregate distance.

The optimal choice of the norm L for calculating the distances between color vectors in Eqn. (15.22) depends on the assumed noise distribution of the underlying signal [10]. The effects of using different norms (L_1 , L_2 , L_∞) are shown in Fig. 15.13 (see Fig. 15.12 for the original images). Although the results for these norms show numerical differences, they are hardly noticeable in real images (particularly in print). Unless otherwise noted, the L_1 norm is used in all subsequent examples.

Results of the scalar median filter and the vector median filter are compared in Fig. 15.14. Note how new colors are introduced by the scalar filter at certain locations (Fig. 15.14(a,c)), as illustrated in Fig. 15.11. In contrast, the vector median filter (Fig. 15.14(b,d)) can only produce colors that already exist in the original image. Figure

```

1: VectorMedianFilter( $\mathbf{I}, r$ )
   Input:  $\mathbf{I} = (I_R, I_G, I_B)$ , a color image of size  $M \times N$ ;
           $r$ , filter radius ( $r \geq 1$ ).
   Returns a new (filtered) color image of size  $M \times N$ .
2:  $(M, N) \leftarrow \text{Size}(\mathbf{I})$ 
3:  $\mathbf{I}' \leftarrow \text{Duplicate}(\mathbf{I})$ 
4: for all image coordinates  $(u, v) \in M \times N$  do
5:    $\mathbf{p}_{\text{ctr}} \leftarrow \mathbf{I}(u, v)$             $\triangleright$  center pixel of support region
6:    $\mathbf{P} \leftarrow \text{GetSupportRegion}(\mathbf{I}, u, v, r)$ 
7:    $d_{\text{ctr}} \leftarrow \text{AggregateDistance}(\mathbf{p}_{\text{ctr}}, \mathbf{P})$ 
8:    $d_{\min} \leftarrow \infty$ 
9:   for all  $\mathbf{p} \in \mathbf{P}$  do
10:     $d \leftarrow \text{AggregateDistance}(\mathbf{p}, \mathbf{P})$ 
11:    if  $d < d_{\min}$  then
12:       $\mathbf{p}_{\min} \leftarrow \mathbf{p}$ 
13:       $d_{\min} \leftarrow d$ 
14:    if  $d_{\min} < d_{\text{ctr}}$  then
15:       $\mathbf{I}'(u, v) \leftarrow \mathbf{p}_{\min}$             $\triangleright$  modify this pixel
16:    else
17:       $\mathbf{I}'(u, v) \leftarrow \mathbf{I}(u, v)$             $\triangleright$  keep the original pixel value
18:   return  $\mathbf{I}'$ 

19: GetSupportRegion( $\mathbf{I}, u, v, r$ )
   Returns a vector of  $n$  pixel values  $\mathbf{P} = (\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n)$  from
   image  $\mathbf{I}$  that are inside a disk of radius  $r$ , centered at position
    $(u, v)$ .
20:  $\mathbf{P} \leftarrow ()$ 
21: for  $i \leftarrow \lfloor u - r \rfloor, \dots, \lceil u + r \rceil$  do
22:   for  $j \leftarrow \lfloor v - r \rfloor, \dots, \lceil v + r \rceil$  do
23:     if  $(u - i)^2 + (v - j)^2 \leq r^2$  then
24:        $\mathbf{p} \leftarrow \mathbf{I}(i, j)$ 
25:        $\mathbf{P} \leftarrow \mathbf{P} \cup (\mathbf{p})$ 
26:   return  $\mathbf{P}$             $\triangleright \mathbf{P} = (\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n)$ 

27: AggregateDistance( $\mathbf{p}, \mathbf{P}$ )
   Returns the aggregate distance  $D_L(\mathbf{p}, \mathbf{P})$  of the sample vector  $\mathbf{p}$ 
   over all elements  $\mathbf{p}_i \in \mathbf{P}$  (see Eq. 15.22).
28:  $d \leftarrow 0$ 
29: for all  $\mathbf{q} \in \mathbf{P}$  do
30:    $d \leftarrow d + \|\mathbf{p} - \mathbf{q}\|_L$             $\triangleright$  choose any distance norm L
31: return  $d$ 

```

15.2 NONLINEAR COLOR FILTERS

Alg. 15.2

Vector median filter for color images.

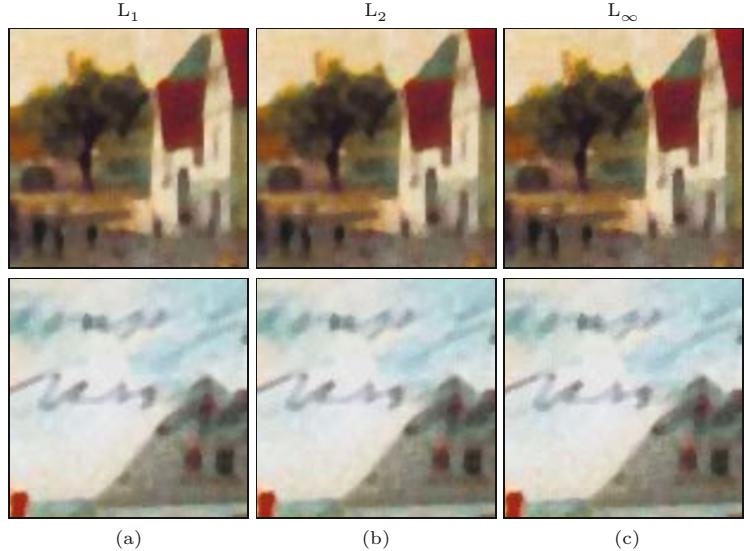
15.15 shows the results of applying the vector median filter to real color images while varying the filter radius.

Since the vector median filter relies on measuring the distance between pairs of colors, the considerations in Sec. 15.1.2 regarding the metric properties of the color space do apply here as well. It is thus not uncommon to perform this filter operation in a perceptual uniform color space, such as CIELUV or CIELAB, rather than in RGB [132, 240, 254].

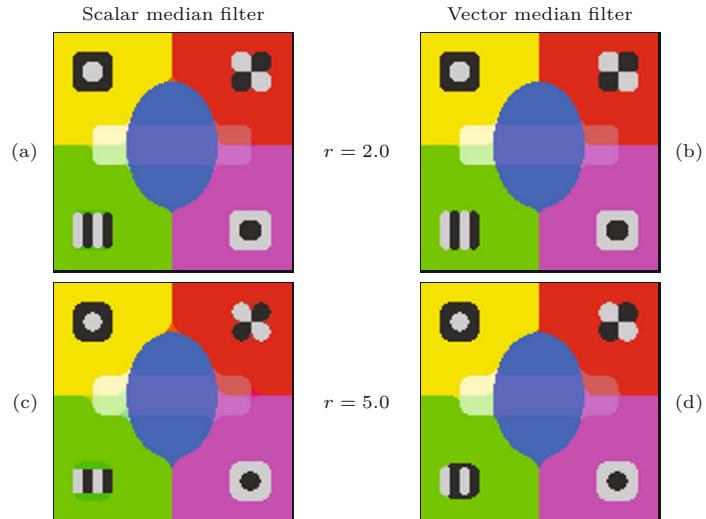
The vector median filter is computationally expensive. Calculating the aggregate distance for all sample vectors \mathbf{p}_i in \mathbf{P} requires $\mathcal{O}(n^2)$ steps, for a support region of size n . Finding the candidate neighborhood pixel with the minimum aggregate distance in \mathbf{P} can

Fig. 15.13

Results of vector median filtering using different color distance norms: L_1 norm (a), L_2 norm (b), L_∞ norm (c). Filter radius $r = 2.0$.


Fig. 15.14

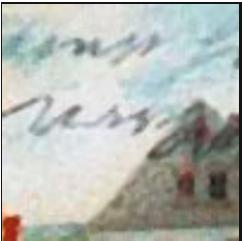
Scalar median vs. vector median filter applied to a color test image, with filter radius $r = 2.0$ (a, b) and $r = 5.0$ (c, d). Note how the scalar median filter (a, c) introduces new colors that are not contained in the original image.



be done in $\mathcal{O}(n)$. Since n is proportional to the square of the filter radius r , the number of steps required for calculating a single image pixel is roughly $\mathcal{O}(r^4)$. While faster implementations have been proposed [10, 18, 221], calculating the vector median filter remains computationally demanding.

15.2.3 Sharpening Vector Median Filter

Although the vector median filter is a good solution for suppressing impulse noise and additive Gaussian noise in color images, it does tend to blur or even eliminate relevant structures, such as lines and edges. The *sharpening* vector median filter, proposed in [155], aims at improving the edge preservation properties of the standard vector median filter described earlier. The key idea is not to calculate the aggregate distances against *all* other samples in the neighborhood but only against the *most similar* ones. The rationale is that



15.2 NONLINEAR COLOR FILTERS

Fig. 15.15

Vector median filter with varying radii applied to a real color image (L_1 norm).

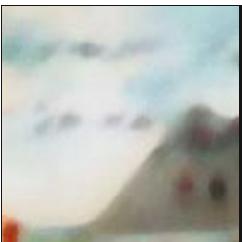
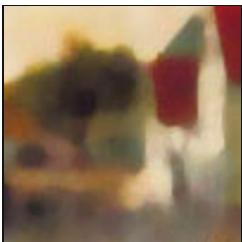
(a) $r = 1.0$



(b) $r = 2.0$



(c) $r = 3.0$



(d) $r = 5.0$

the samples deviating strongly from their neighbors tend to be *outliers* (e.g., caused by nearby edges) and should be excluded from the median calculation to avoid blurring of structural details.

The operation of the sharpening vector median filter is summarized in Alg. 15.3. For calculating the aggregate distance $D_L(\mathbf{p}, \mathbf{P})$ of a given sample vector \mathbf{p} (see Eqn. (15.22)), not all samples in \mathbf{P} are considered, but only those a samples that are *closest* to \mathbf{p} in the 3D color space (a being a fixed fraction of the support region size). The subsequent minimization is performed over what is called the “trimmed aggregate distance”. Thus, only a fixed number (a) of neighborhood pixels is included in the calculation of the aggregate distances. As a consequence, the sharpening vector median filter provides good noise removal while at the same time leaving edge structures intact.

15 FILTERS FOR COLOR IMAGES

Alg. 15.3

Sharpening vector median filter for RGB color images (extension of Alg. 15.2).

The *sharpening parameter* $s \in [0, 1]$ controls the number of most-similar neighborhood pixels included in the median calculation. For $s = 0$, all pixels in the given support region are included and no sharpening occurs; setting $s = 1$ leads to maximum sharpening. The *threshold parameter* t controls how much smaller the aggregate distance of any neighborhood pixel must be to replace the current center pixel.

```

1: SharpeningVectorMedianFilter( $I, r, s, t$ )
   Input:  $I$ , a color image of size  $M \times N$ ,  $I(u, v) \in \mathbb{R}^3$ ;  $r$ , filter radius ( $r \geq 1$ );  $s$ , sharpening parameter ( $0 \leq s \leq 1$ );  $t$ , threshold ( $t \geq 0$ ). Returns a new (filtered) color image of size  $M \times N$ .
2:  $(M, N) \leftarrow \text{Size}(I)$ 
3:  $I' \leftarrow \text{Duplicate}(I)$ 
4: for all image coordinates  $(u, v) \in M \times N$  do
5:    $P \leftarrow \text{GetSupportRegion}(I, u, v, r)$                                  $\triangleright$  see Alg. 15.2
6:    $n \leftarrow |P|$                                                         $\triangleright$  size of  $P$ 
7:    $a \leftarrow \text{round}(n - s \cdot (n - 2))$                                 $\triangleright a = 2, \dots, n$ 
8:    $d_{\text{ctr}} \leftarrow \text{TrimmedAggregateDistance}(I(u, v), P, a)$ 
9:    $d_{\min} \leftarrow \infty$ 
10:  for all  $p \in P$  do
11:     $d \leftarrow \text{TrimmedAggregateDistance}(p, P, a)$ 
12:    if  $d < d_{\min}$  then
13:       $p_{\min} \leftarrow p$ 
14:       $d_{\min} \leftarrow d$ 
15:    if  $(d_{\text{ctr}} - d_{\min}) > t \cdot a$  then
16:       $I'(u, v) \leftarrow p_{\min}$                                           $\triangleright$  replace the center pixel
17:    else
18:       $I'(u, v) \leftarrow I(u, v)$                                           $\triangleright$  keep the original center pixel
19:  return  $I'$ 
20: TrimmedAggregateDistance( $p, P, a$ )
   Returns the aggregate distance from  $p$  to the  $a$  most similar elements in  $P = (p_1, p_2, \dots, p_n)$ .
21:  $n \leftarrow |P|$                                                         $\triangleright$  size of  $P$ 
22: Create map  $D : [1, n] \mapsto \mathbb{R}$ 
23: for  $i \leftarrow 1, \dots, n$  do
24:    $D(i) \leftarrow \|p - P(i)\|_L$                                       $\triangleright$  choose any distance norm L
25:    $D' \leftarrow \text{Sort}(D)$                                           $\triangleright D'(1) \leq D'(2) \leq \dots \leq D'(n)$ 
26:    $d \leftarrow 0$ 
27:   for  $i \leftarrow 2, \dots, a$  do                                          $\triangleright D'(1) = 0$ , thus skipped
28:      $d \leftarrow d + D'(i)$ 
29: return  $d$ 

```

Typically, the aggregate distance of p to the a closest neighborhood samples is found by first calculating the distances between p and all other samples in P , then sorting the result, and finally adding up only the a initial elements of the sorted distances (see procedure $\text{TrimmedAggregateDistance}(p, P, a)$ in Alg. 15.3). Thus the sharpening median filter requires an additional sorting step over $n \propto r^2$ elements at each pixel, which again adds to its time complexity.

The parameter s in Alg. 15.3 specifies the fraction of region pixels included in the calculation of the median and thus controls the amount of sharpening. The number of incorporated pixels a is determined as $a = \text{round}(n - s \cdot (n - 2))$ (see Alg. 15.3, line 7), so that $a = n, \dots, 2$ for $s \in [0, 1]$. With $s = 0$, all $a = |P| = n$ pixels in the filter region are included in the median calculation and the filter behaves like the ordinary vector-median filter described in Alg. 15.2. At maximum sharpening (i.e., with $s = 1$) the calculation of the aggregate distance includes only the single most similar color pixel in the neighborhood P .

The calculation of the “trimmed aggregate distance” is shown in Alg. 15.3 (lines 20–29). The function `TrimmedAggregateDistance` ($\mathbf{p}, \mathbf{P}, a$) calculates the aggregate distance for a given vector (color sample) \mathbf{p} over the a closest samples in the support region \mathbf{P} . Initially (in line 24), the n distances $D(i)$ between \mathbf{p} and all elements in \mathbf{P} are calculated, with $D(i) = \|\mathbf{p} - \mathbf{P}(i)\|_L$ (see Eqns. (15.23)–(15.25)). These are subsequently sorted by increasing value (line 25) and the sum of the a smallest values $D'(1), \dots, D'(a)$ (line 28) is returned.⁶

The effects of varying the sharpen parameter s are shown in Fig. 15.16, with a fixed filter radius $r = 2.0$ and threshold $t = 0$. For $s = 0.0$ (Fig. 15.16(a)), the result is the same as that of the ordinary vector median filter (see Fig. 15.15(b)).

The value of the current center pixel is only replaced by a neighboring pixel value if the corresponding minimal (trimmed) aggregate distance d_{\min} is significantly smaller than the center pixel’s aggregate distance d_{ctr} . In Alg. 15.3, this is controlled by the threshold t . The center pixel is replaced only if the condition

$$(d_{\text{ctr}} - d_{\min}) > t \cdot a \quad (15.27)$$

holds; otherwise it remains unmodified. Note that the distance limit is proportional to a and thus t really specifies the minimum “average” pixel distance; it is independent of the filter radius r and the sharpening parameter s .

Results for typical values of t (in the range $0, \dots, 10$) are shown in Figs. 15.17–15.18. To illustrate the effect, the images in Fig. 15.18 only display those pixels that were *not* replaced by the filter, while all modified pixels are set to black. As one would expect, increasing the threshold t leads to fewer pixels being modified. Of course, the same thresholding scheme may also be used with the ordinary vector median filter (see Exercise 15.2).

15.3 Java Implementation

Implementations of the scalar and vector median filter as well as the sharpening vector median filter are available with full Java source code at the book’s website.⁷ The corresponding classes

- `ScalarMedianFilter`,
- `VectorMedianFilter`, and
- `VectorMedianFilterSharpen`

are based on the common super-class `GenericFilter`, which provides the abstract methods

```
void applyTo (ImageProcessor ip),
```

which greatly simplifies the use of these filters. The code segment in Prog. 15.1 demonstrates the use of the class `VectorMedianFilter` (with radius 3.0 and L_1 -norm) for RGB color images in an ImageJ plugin. For the specific filters described in this chapter, the following constructors are provided:

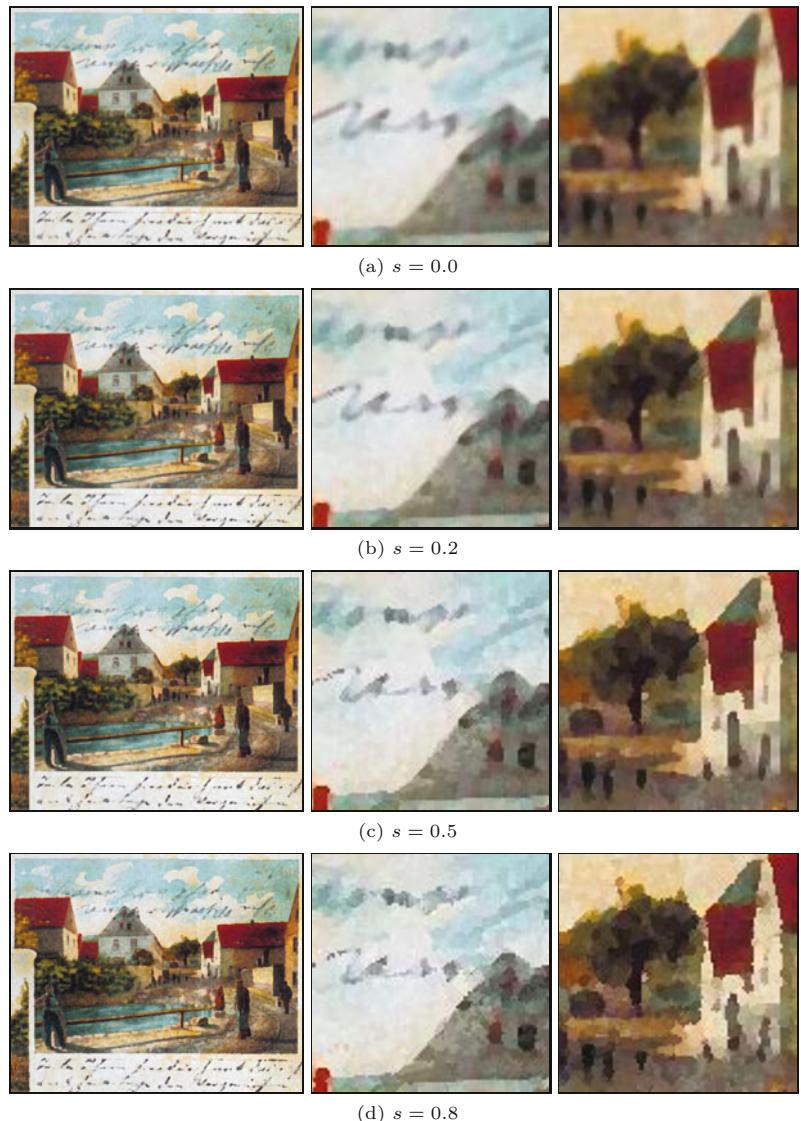
⁶ $D'(1)$ is zero because it is the distance between \mathbf{p} and itself.

⁷ Package `imagingbook.pub.color.filters`.

15 FILTERS FOR COLOR IMAGES

Fig. 15.16

Sharpening vector median filter with different sharpness values s . The filter radius is $r = 2.0$ and the corresponding filter mask contains $n = 21$ pixels. At each pixel, only the $a = 21, 17, 12, 6$ closest color samples (for sharpness $s = 0.0, 0.2, 0.5, 0.8$, respectively) are considered when calculating the local vector median.



ScalarMedianFilter (Parameters params)

Creates a scalar median filter, as described in Sec. 15.2.1, with parameter **radius** = 3.0 (default).

VectorMedianFilter (Parameters params)

Creates a vector median filter, as described in Sec. 15.2.2, with parameters **radius** = 3.0 (default), **distanceNorm** = **NormType.L1** (default), **L2**, **Lmax**.

VectorMedianFilterSharpen (Parameters params)

Creates a sharpening vector median filter (see Sec. 15.2.3) with parameters **radius** = 3.0 (default), **distanceNorm** = **NormType.L1** (default), **L2**, **Lmax**, sharpening factor **sharpen** = 0.5 (default), **threshold** = 0.0 (default).

The listed default values pertain to the parameterless constructors that are also available. See the online API documentation or the



15.4 FURTHER READING

Fig. 15.17

Sharpening vector median filter with different threshold values $t = 0, 2, 5, 10$. The filter radius and sharpening factor are fixed at $r = 2.0$ and $s = 0.0$, respectively.

(a) $t = 0$



(b) $t = 2$



(c) $t = 5$



(d) $t = 10$

source code for additional details. Note that the created filter objects are generic and can be applied to both grayscale and color images without any modification.

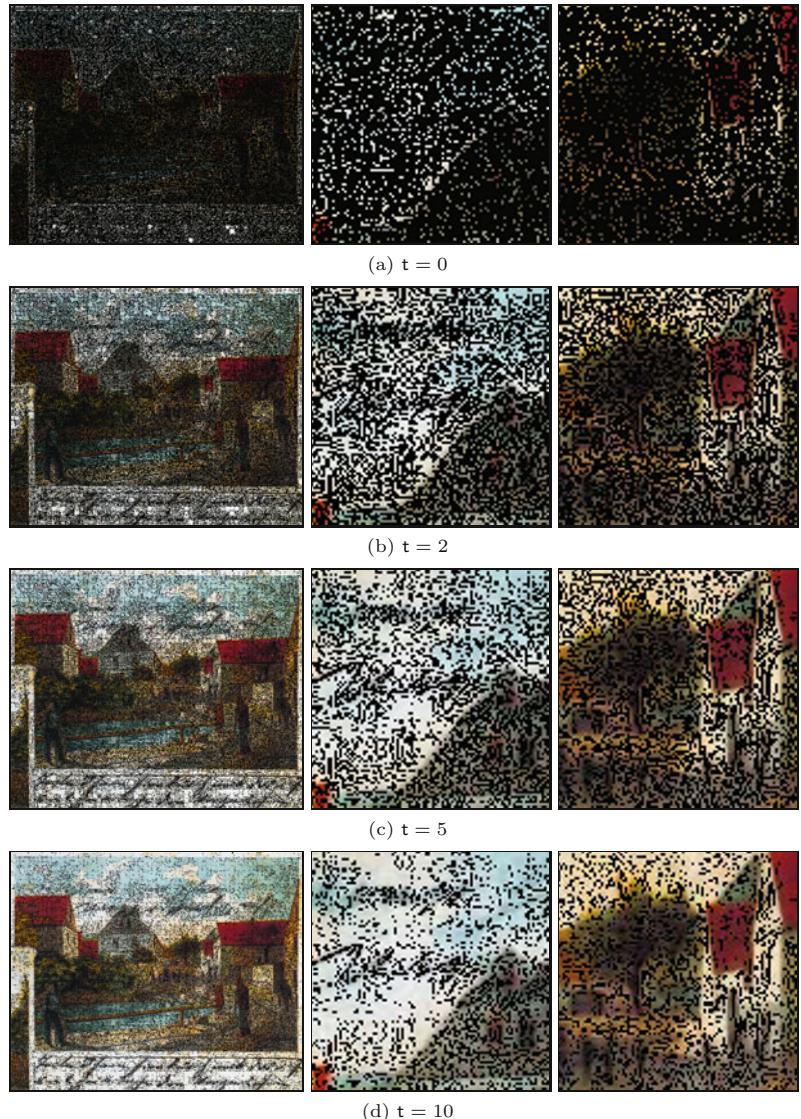
15.4 Further Reading

A good overview of different linear and nonlinear filtering techniques for color images can be found in [141]. In [186, Ch. 2], the authors give a concise treatment of color image filtering, including statistical noise models, vector ordering schemes, and different color similarity measures. Several variants of weighted median filters for color images and multi-channel data in general are described in [6, Ch. 2, Sec. 2.4]. A very readable and up-to-date survey of important color issues in computer vision, such as color constancy, photometric invariance, and

15 FILTERS FOR COLOR IMAGES

Fig. 15.18

Sharpening vector median filter with different threshold values $t = 0, 2, 5, 10$ (also see Fig. 15.17). Only the *unmodified* pixels are shown in color, while all modified pixels are set to *black*. The filter radius and sharpening factor are fixed at $r = 2.0$ and $s = 0.0$, respectively.



color feature extraction, can be found in [83]. A vector median filter operating in HSV color space is proposed in [240]. In addition to the techniques discussed in this chapter, most of the filters described in Chapter 17 can either be applied directly to color images or easily modified for this purpose.

15.5 Exercises

Exercise 15.1. Verify Eqn. (15.20) by showing (formally or experimentally) that the usual calculation of the scalar median (by sorting a sequence and selecting the center value) indeed gives the value with the smallest sum of differences from all other values in the same sequence. Is the result independent of the type of distance norm used?

```

1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4 import imagingbook.lib.math.VectorNorm.NormType;
5 import imagingbook.lib.util.Enums;
6 import imagingbook.pub.colorfilters.VectorMedianFilter;
7 import imagingbook.pub.colorfilters.VectorMedianFilter.*;
8
9 public class MedianFilter_Color_Vector implements
10    PlugInFilter
11 {
12     public int setup(String arg, ImagePlus imp) {
13         return DOES_RGB;
14     }
15     public void run(ImageProcessor ip) {
16         Parameters params =
17             new VectorMedianFilter.Parameters();
18         params.distanceNorm = NormType.L1;
19         params.radius = 3.0;
20
21         VectorMedianFilter filter =
22             new VectorMedianFilter(params);
23
24         filter.applyTo(ip);
25     }
26 }
```

15.5 EXERCISES

Prog. 15.1

Color median filter using class `VectorMedianFilter`. In line 17, a suitable parameter object (with default values) is created, then modified and passed to the constructor of the filter (in line 22). The filter itself is applied to the input image, which is destructively modified (in line 24).

Exercise 15.2. Modify the ordinary vector median filter described in Alg. 15.2 to incorporate a threshold t for deciding whether to modify the current center pixel or not, analogous to the approach taken in the sharpening vector median filter in Alg. 15.3.

Exercise 15.3. Implement a dedicated median filter (analogous to Alg. 15.1) for the HSV color space. The filter should process the color components independently but consider the circular nature of the hue component, as discussed in Sec. 15.1.3. Compare the results to the vector-median filter in Sec. 15.2.2.