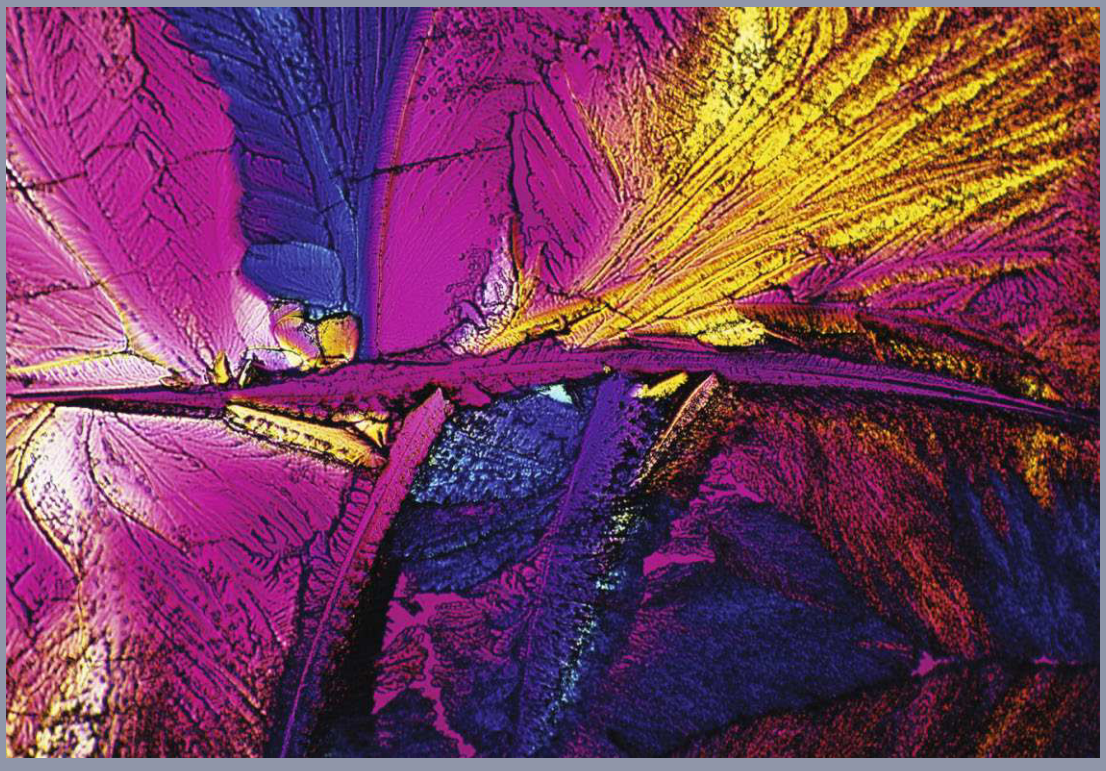
TEXTS IN COMPUTER SCIENCE

Digital Image Processing

An Algorithmic Introduction Using Java



Wilhelm Burger
Mark J. Burge

Springer

Digital Image Processing

First Edition

Wilhelm Burger Mark James Burge

Digital Image Processing

An Algorithmic Introduction using Java

First Edition

Springer

Wilhelm Burger, PhD
Upper Austria University of Applied Sciences
Digital Media Department
11 Softwarepark
Hagenberg 4232
Austria
wilbur@ieee.org

Mark James Burge, PhD
National Science Foundation
4201 Wilson Blvd.
Arlington, VA 22230
USA
mburge@acm.org

ISBN 978-1-84628-379-6 e-ISBN 978-1-84628-968-2

Library of Congress Control Number: 2007938446

Printed on acid-free paper.

9 8 7 6 5 4 3 2 1

springer.com

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 and practitioners in search of critical analysis and modern implementations of the most important techniques. This is the first 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.
- Information technology (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, one of which is that it is the first programming language learned by students in a wide variety of engineering curricula. Practitioners 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 Wayne Rasband of 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 the detailed synthesis of the fundamental concepts and techniques. As both practitioners and developers, we know that the details required for successful understanding and application of classical techniques are often difficult to find, and for this reason we have been very careful to provide the details, many gleaned over years of practical application, necessary to successfully apply these techniques. 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 omitted deliberately, including advanced issues such as object recognition, image understanding, or three-dimensional 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 a comprehensive ImageJ reference and many code examples, programming in and of itself is not our main focus. Instead it serves as just one important element for describing each technique in a precise and immediately testable way.

[1] http://rsb.info.nih.gov/ij/.

Why we wrote this book

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 diverse disciplines. Previous to the explosion of digital media, it was often the case that a computing curriculum would offer only a single course, called "Digital Signal Processing" in engineering or "Digital Image Processing" in computing, and likely only as a graduate elective.

Today the topic has migrated into the early stages of many curricula, where it now serves as a key foundation course. This migration uncovered a problem in that many of the texts relied on as standards in the older graduate-level courses were not appropriate for beginners. The texts were usually too formal for beginners, 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 a need.

The contents of the following chapters can be presented in either a one- or two-semester sequence. Where it was feasible, we have added supporting material in order to make each chapter as independent as possible and provide the instructor with as much flexibility as possible when designing the course. Chapters 13–15 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 road map (on page VIII) provides a sequence of topics for a one- or two-semester syllabus.

One Semester: A one-semester course can be organized around either of two major themes: image *processing* or image *analysis*. While either theme integrates easily into the early semesters of a modern computer science or IT curriculum, image analysis is especially appropriate as an early foundation course in medical informatics.

Two Semesters: When the content can be presented over two semesters, it has been designed so that it can be coherently divided (as described below) into two courses (*fundamentals* and *advanced*) where the themes are grouped according to difficulty.

Supplement to the English edition

This book was translated by the authors from the second German edition (published in 2006) [17], incorporating many enhancements throughout

Road Map for One- and Two-Semester Courses	Image Processing	Image Analysis	Fundamentals	Advanced
1. Crunching Pixels	☒	☒	☒	☐
2. Digital Images	☒	☒	☒	☐
3. ImageJ	☒	☒	☒	☐
4. Histograms	☒	☐	☒	☐
5. Point Operations	☒	☐	☒	☐
6. Filters	☒	☒	☒	☐
7. Edges and Contours	☒	☒	☒	☐
8. Finding Points of Interest	☐	☒	☐	☒
9. Detecting Simple Curves	☐	☒	☐	☒
10. Morphological Filters	☒	☐	☒	☐
11. Regions in Binary Images	☐	☒	☒	☐
12. Color Images	☒	☐	☐	☒
13. Introduction to Spectral Techniques	☐	☒	☐	☒
14. The Discrete Fourier Transform in 2D	☐	☒	☐	☒
15. The Discrete Cosine Transform	☐	☐	☐	☒
16. Geometrical Image Operations	☒	☐	☐	☒
17. Image Comparison	☐	☒	☐	☒
	1 Sem.		2 Sem.	

the text. In addition to the numerous small corrections and improvements that have been made, the presentation of histogram matching in Ch. 5, geometric region properties based on moments in Ch. 11, morphological filters in Ch. 10, and interpolation methods in Ch. 16 have been completely revised. Also, a number of example programs, such as the single-pass region labeling and contour finding algorithm (Sec. 11.2.2), have been rewritten for improved clarity and to take advantage of the new language features in Java 5.

Online resources

Visit the Website for this text

www.imagingbook.com

to download supplementary materials, including the complete Java source code for the examples, the test images used in the examples, and corrections. Additional materials are available for educators, including a complete set of formulas and figures used in the text, in a format suitable for inclusion in presentations. Comments, questions, and corrections are welcome and should be addressed to

imagingbook@gmail.com

Thank you

This book would not have been possible without the understanding and support of our families. Our thanks go to Wayne Rasband (NIH) for developing ImageJ and for his truly outstanding support of the community, to our colleagues Prof. Axel Pinz (TU Graz) and Prof. Vaclav Hlavac (TU Prague) for their comments, and to all the readers of the first two editions who provided valuable input, suggestions for improvement, and encouragement as we translated this edition. The authors greatly appreciate the help of their brave Sony and Apple notebooks that performed an estimated 1.6 quadrillion (10^{15}) CPU cycles to prepare this manuscript, thereby consuming about 560 kWh of electric energy and producing 196 kg of carbon dioxide. Finally, we owe a debt of gratitude to the professionals at Springer-Verlag, Ursula Zimpfer and Jutta Maria Fleschutz, who led the German edition team, and Wayne Wheeler, Catherine Brett, and Jeffrey Taub who were responsible for the English edition.

Hagenberg, Austria / Washington DC, USA
July 2007

Contents

Preface V

1 Crunching Pixels 1
1.1 Programming with Images 2
1.2 Image Analysis and Computer Vision 3

2 Digital Images 5
2.1 Types of Digital Images 5
2.2 Image Acquisition 5
2.2.1 The Pinhole Camera Model 7
2.2.2 The "Thin" Lens 8
2.2.3 Going Digital 8
2.2.4 Image Size and Resolution 10
2.2.5 Image Coordinate System 11
2.2.6 Pixel Values 11
2.3 Image File Formats 13
2.3.1 Raster versus Vector Data 14
2.3.2 Tagged Image File Format (TIFF) 15
2.3.3 Graphics Interchange Format (GIF) 15
2.3.4 Portable Network Graphics (PNG) 16
2.3.5 JPEG 17
2.3.6 Windows Bitmap (BMP) 20
2.3.7 Portable Bitmap Format (PBM) 21
2.3.8 Additional File Formats 21
2.3.9 Bits and Bytes 22
2.4 Exercises 24

3 ImageJ 27
3.1 Image Manipulation and Processing 28
3.2 ImageJ Overview 28
3.2.1 Key Features 29
3.2.2 Interactive Tools 30
3.2.3 ImageJ Plugins 31
3.2.4 A First Example: Inverting an Image 32
3.3 Additional Information on ImageJ and Java 35
3.3.1 Resources for ImageJ 35
3.3.2 Programming with Java 35

3.4 Exercises 36

4 Histograms 37
4.1 What Is a Histogram? 38
4.2 Interpreting Histograms 39
4.2.1 Image Acquisition 40
4.2.2 Image Defects 42
4.3 Computing Histograms 44
4.4 Histograms of Images with More than 8 Bits 46
4.4.1 Binning 46
4.4.2 Example 46
4.4.3 Implementation 47
4.5 Color Image Histograms 47
4.5.1 Intensity Histograms 48
4.5.2 Individual Color Channel Histograms 48
4.5.3 Combined Color Histogram 49
4.6 Cumulative Histogram 50
4.7 Exercises 51

5 Point Operations 53
5.1 Modifying Image Intensity 54
5.1.1 Contrast and Brightness 54
5.1.2 Limiting the Results by Clamping 54
5.1.3 Inverting Images 55
5.1.4 Threshold Operation 55
5.2 Point Operations and Histograms 55
5.3 Automatic Contrast Adjustment 57
5.4 Modified Auto-Contrast 58
5.5 Histogram Equalization 59
5.6 Histogram Specification 62
5.6.1 Frequencies and Probabilities 63
5.6.2 Principle of Histogram Specification 65
5.6.3 Adjusting to a Piecewise Linear Distribution 65
5.6.4 Adjusting to a Given Histogram (Histogram Matching) 67
5.6.5 Examples 68
5.7 Gamma Correction 72
5.7.1 Why Gamma? 73
5.7.2 The Gamma Function 74
5.7.3 Real Gamma Values 74
5.7.4 Applications of Gamma Correction 75
5.7.5 Implementation 76
5.7.6 Modified Gamma Correction 76
5.8 Point Operations in ImageJ 80
5.8.1 Point Operations with Lookup Tables 80
5.8.2 Arithmetic Operations 81
5.8.3 Point Operations Involving Multiple Images 81

5.8.4 Methods for Point Operations on Two Images 82
5.8.5 ImageJ Plugins for Multiple Images 82
5.9 Exercises 83

6 Filters 87
6.1 What Is a Filter? 87
6.2 Linear Filters 89
6.2.1 The Filter Matrix 89
6.2.2 Applying the Filter 90
6.2.3 Computing the Filter Operation 91
6.2.4 Filter Plugin Examples 92
6.2.5 Integer Coefficients 93
6.2.6 Filters of Arbitrary Size 94
6.2.7 Types of Linear Filters 95
6.3 Formal Properties of Linear Filters 98
6.3.1 Linear Convolution 99
6.3.2 Properties of Linear Convolution 100
6.3.3 Separability of Linear Filters 101
6.3.4 Impulse Response of a Filter 103
6.4 Nonlinear Filters 104
6.4.1 Minimum and Maximum Filters 105
6.4.2 Median Filter 106
6.4.3 Weighted Median Filter 107
6.4.4 Other Nonlinear Filters 110
6.5 Implementing Filters 111
6.5.1 Efficiency of Filter Programs 111
6.5.2 Handling Image Borders 111
6.5.3 Debugging Filter Programs 112
6.6 Filter Operations in ImageJ 113
6.6.1 Linear Filters 113
6.6.2 Gaussian Filters 114
6.6.3 Nonlinear Filters 115
6.7 Exercises 115

7 Edges and Contours 117
7.1 What Makes an Edge? 117
7.2 Gradient-Based Edge Detection 118
7.2.1 Partial Derivatives and the Gradient 119
7.2.2 Derivative Filters 119
7.3 Edge Operators 120
7.3.1 Prewitt and Sobel Operators 120
7.3.2 Roberts Operator 123
7.3.3 Compass Operators 123
7.3.4 Edge Operators in ImageJ 125
7.4 Other Edge Operators 125
7.4.1 Edge Detection Based on Second Derivatives 126
7.4.2 Edges at Different Scales 126

7.4.3 Canny Operator 127
7.5 From Edges to Contours 127
7.5.1 Contour Following 127
7.5.2 Edge Maps 129
7.6 Edge Sharpening 130
7.6.1 Edge Sharpening with the Laplace Filter 130
7.6.2 Unsharp Masking 133
7.7 Exercises 137

8 Corner Detection 139
8.1 Points of Interest 139
8.2 Harris Corner Detector 140
8.2.1 Local Structure Matrix 140
8.2.2 Corner Response Function (CRF) 141
8.2.3 Determining Corner Points 142
8.2.4 Example 142
8.3 Implementation 142
8.3.1 Step 1: Computing the Corner Response Function 144
8.3.2 Step 2: Selecting "Good" Corner Points 148
8.3.3 Displaying the Corner Points 152
8.3.4 Summary 152
8.4 Exercises 153

9 Detecting Simple Curves 155
9.1 Salient Structures 155
9.2 Hough Transform 156
9.2.1 Parameter Space 157
9.2.2 Accumulator Array 159
9.2.3 A Better Line Representation 159
9.3 Implementing the Hough Transform 160
9.3.1 Filling the Accumulator Array 161
9.3.2 Analyzing the Accumulator Array 163
9.3.3 Hough Transform Extensions 165
9.4 Hough Transform for Circles and Ellipses 167
9.4.1 Circles and Arcs 167
9.4.2 Ellipses 170
9.5 Exercises 170

10 Morphological Filters 173
10.1 Shrink and Let Grow 174
10.1.1 Neighborhood of Pixels 175
10.2 Basic Morphological Operations 175
10.2.1 The Structuring Element 175
10.2.2 Point Sets 176
10.2.3 Dilation 177
10.2.4 Erosion 178
10.2.5 Properties of Dilation and Erosion 178

10.2.6 Designing Morphological Filters 180
10.2.7 Application Example: Outline 181
10.3 Composite Operations 183
10.3.1 Opening 185
10.3.2 Closing 185
10.3.3 Properties of Opening and Closing 186
10.4 Grayscale Morphology 187
10.4.1 Structuring Elements 187
10.4.2 Dilation and Erosion 187
10.4.3 Grayscale Opening and Closing 188
10.5 Implementing Morphological Filters 189
10.5.1 Binary Images in ImageJ 189
10.5.2 Dilation and Erosion 191
10.5.3 Opening and Closing 193
10.5.4 Outline 194
10.5.5 Morphological Operations in ImageJ 194
10.6 Exercises 196

11 Regions in Binary Images 199
11.1 Finding Image Regions 200
11.1.1 Region Labeling with Flood Filling 200
11.1.2 Sequential Region Labeling 204
11.1.3 Region Labeling—Summary 209
11.2 Region Contours 209
11.2.1 External and Internal Contours 209
11.2.2 Combining Region Labeling and Contour Finding 212
11.2.3 Implementation 213
11.2.4 Example 216
11.3 Representing Image Regions 216
11.3.1 Matrix Representation 216
11.3.2 Run Length Encoding 218
11.3.3 Chain Codes 219
11.4 Properties of Binary Regions 222
11.4.1 Shape Features 222
11.4.2 Geometric Features 223
11.4.3 Statistical Shape Properties 226
11.4.4 Moment-Based Geometrical Properties 228
11.4.5 Projections 233
11.4.6 Topological Properties 234
11.5 Exercises 235

12 Color Images 239
12.1 RGB Color Images 239
12.1.1 Organization of Color Images 241
12.1.2 Color Images in ImageJ 244
12.2 Color Spaces and Color Conversion 253
12.2.1 Conversion to Grayscale 256

12.2.2 Desaturating Color Images 257
12.2.3 HSV/HSB and HLS Color Space 258
12.2.4 TV Color Spaces—YUV, YIQ, and YC_bC_r 267
12.2.5 Color Spaces for Printing—CMY and CMYK 271
12.3 Colorimetric Color Spaces 275
12.3.1 CIE Color Spaces 276
12.3.2 CIE $L^*a^*b^*$ 281
12.3.3 sRGB 283
12.3.4 Adobe RGB 288
12.3.5 Chromatic Adaptation 288
12.3.6 Colorimetric Support in Java 292
12.4 Statistics of Color Images 299
12.4.1 How Many Colors Are in an Image? 299
12.4.2 Color Histograms 299
12.5 Color Quantization 301
12.5.1 Scalar Color Quantization 303
12.5.2 Vector Quantization 305
12.6 Exercises 311

13 Introduction to Spectral Techniques 313
13.1 The Fourier Transform 314
13.1.1 Sine and Cosine Functions 314
13.1.2 Fourier Series of Periodic Functions 317
13.1.3 Fourier Integral 318
13.1.4 Fourier Spectrum and Transformation 319
13.1.5 Fourier Transform Pairs 320
13.1.6 Important Properties of the Fourier Transform . . . 321
13.2 Working with Discrete Signals 325
13.2.1 Sampling 325
13.2.2 Discrete and Periodic Functions 330
13.3 The Discrete Fourier Transform (DFT) 332
13.3.1 Definition of the DFT 332
13.3.2 Discrete Basis Functions 334
13.3.3 Aliasing Again! 334
13.3.4 Units in Signal and Frequency Space 338
13.3.5 Power Spectrum 339
13.4 Implementing the DFT 340
13.4.1 Direct Implementation 340
13.4.2 Fast Fourier Transform (FFT) 342
13.5 Exercises 342

14 The Discrete Fourier Transform in 2D 343
14.1 Definition of the 2D DFT 343
14.1.1 2D Basis Functions 344
14.1.2 Implementing the Two-Dimensional DFT 344
14.2 Visualizing the 2D Fourier Transform 345
14.2.1 Range of Spectral Values 348

14.2.2 Centered Representation 348
14.3 Frequencies and Orientation in 2D 348
14.3.1 Effective Frequency 349
14.3.2 Frequency Limits and Aliasing in 2D 350
14.3.3 Orientation 350
14.3.4 Correcting the Geometry of a 2D Spectrum 351
14.3.5 Effects of Periodicity 352
14.3.6 Windowing 352
14.3.7 Windowing Functions 354
14.4 2D Fourier Transform Examples 359
14.5 Applications of the DFT 359
14.5.1 Linear Filter Operations in Frequency Space 363
14.5.2 Linear Convolution versus Correlation 364
14.5.3 Inverse Filters 364
14.6 Exercises 366

15 The Discrete Cosine Transform (DCT) 367
15.1 One-Dimensional DCT 367
15.1.1 DCT Basis Functions 368
15.1.2 Implementing the One-Dimensional DCT 368
15.2 Two-Dimensional DCT 370
15.2.1 Separability 371
15.2.2 Examples 371
15.3 Other Spectral Transforms 371
15.4 Exercises 373

16 Geometric Operations 375
16.1 2D Mapping Function 376
16.1.1 Simple Mappings 377
16.1.2 Homogeneous Coordinates 377
16.1.3 Affine (Three-Point) Mapping 378
16.1.4 Projective (Four-Point) Mapping 380
16.1.5 Bilinear Mapping 385
16.1.6 Other Nonlinear Image Transformations 386
16.1.7 Local Image Transformations 389
16.2 Resampling the Image 390
16.2.1 Source-to-Target Mapping 390
16.2.2 Target-to-Source Mapping 391
16.3 Interpolation 392
16.3.1 Simple Interpolation Methods 392
16.3.2 Ideal Interpolation 393
16.3.3 Interpolation by Convolution 397
16.3.4 Cubic Interpolation 397
16.3.5 Spline Interpolation 399
16.3.6 Lanczos Interpolation 402
16.3.7 Interpolation in 2D 404
16.3.8 Aliasing 410

16.4 Java Implementation 413
16.4.1 Geometric Transformations 413
16.4.2 Pixel Interpolation 423
16.4.3 Sample Applications 426
16.5 Exercises 427

17 Comparing Images 429
17.1 Template Matching in Intensity Images 430
17.1.1 Distance between Image Patterns 431
17.1.2 Implementation 438
17.1.3 Matching under Rotation and Scaling 439
17.2 Matching Binary Images 441
17.2.1 Direct Comparison 441
17.2.2 The Distance Transform 442
17.2.3 Chamfer Matching 446
17.3 Exercises 447

A Mathematical Notation 451
A.1 Symbols 451
A.2 Set Operators 453
A.3 Complex Numbers 453
A.4 Algorithmic Complexity and $\mathcal{O}$ Notation 454

B Java Notes 457
B.1 Arithmetic 457
B.1.1 Integer Division 457
B.1.2 Modulus Operator 459
B.1.3 Unsigned Bytes 459
B.1.4 Mathematical Functions (Class `Math`) 460
B.1.5 Rounding 461
B.1.6 Inverse Tangent Function 462
B.1.7 `Float` and `Double` (Classes) 462
B.2 Arrays and Collections 462
B.2.1 Creating Arrays 462
B.2.2 Array Size 463
B.2.3 Accessing Array Elements 463
B.2.4 Two-Dimensional Arrays 464
B.2.5 Cloning Arrays 465
B.2.6 Arrays of Objects, Sorting 466
B.2.7 Collections 467

C ImageJ Short Reference 469
C.1 Installation and Setup 469
C.2 ImageJ API 471
C.2.1 Images and Processors 471
C.2.2 Images (Package `ij`) 471
C.2.3 Image Processors (Package `ij.process`) 472

C.2.4 Plugins (Packages `ij.plugin`, `ij.plugin.filter`) 473
C.2.5 GUI Classes (Package `ij.gui`) 474
C.2.6 Window Management (Package `ij`) 475
C.2.7 Utility Classes (Package `ij`) 475
C.2.8 Input-Output (Package `ij.io`) 475
C.3 Creating Images and Image Stacks 476
C.3.1 `ImagePlus` (Class) 476
C.3.2 `ImageStack` (Class) 476
C.3.3 `IJ` (Class) 477
C.3.4 `NewImage` (Class) 477
C.3.5 `ImageProcessor` (Class) 478
C.4 Creating Image Processors 478
C.4.1 `ImagePlus` (Class) 478
C.4.2 `ImageProcessor` (Class) 478
C.4.3 `ByteProcessor` (Class) 479
C.4.4 `ColorProcessor` (Class) 479
C.4.5 `FloatProcessor` (Class) 479
C.4.6 `ShortProcessor` (Class) 480
C.5 Loading and Storing Images 480
C.5.1 `IJ` (Class) 480
C.5.2 `Opener` (Class) 481
C.5.3 `FileSaver` (Class) 482
C.5.4 `FileOpener` (Class) 484
C.6 Image Parameters 485
C.6.1 `ImageProcessor` (Class) 485
C.6.2 `ColorProcessor` (Class) 485
C.7 Accessing Pixels 485
C.7.1 Accessing Pixels by 2D Image Coordinates 485
C.7.2 Accessing Pixels by 1D Indices 487
C.7.3 Accessing Multiple Pixels 488
C.7.4 Accessing All Pixels at Once 489
C.7.5 Specific Access Methods for Color Images 490
C.7.6 Direct Access to Pixel Arrays 490
C.8 Converting Images 492
C.8.1 `ImageProcessor` (Class) 492
C.8.2 `ImagePlus`, `ImageConverter` (Classes) 492
C.9 Histograms and Image Statistics 494
C.9.1 `ImageProcessor` (Class) 494
C.10 Point Operations 494
C.10.1 `ImageProcessor` (Class) 495
C.11 Filters 497
C.11.1 `ImageProcessor` (Class) 497
C.12 Geometric Operations 497
C.12.1 `ImageProcessor` (Class) 497
C.13 Graphic Operations 499
C.13.1 `ImageProcessor` (Class) 499
C.14 Displaying Images and Image Stacks 500

C.14.1 `ImagePlus` (Class) 500
C.14.2 `ImageProcessor` (Class) 502
C.15 Operations on Image Stacks 504
C.15.1 `ImagePlus` (Class) 504
C.15.2 `ImageStack` (Class) 505
C.15.3 Stack Example 506
C.16 Regions of Interest 506
C.16.1 `ImagePlus` (Class) 509
C.16.2 `Roi`, `Line`, `OvalRoi`, `PointRoi`, `PolygonRoi` (Classes) 510
C.16.3 `ImageProcessor` (Class) 511
C.16.4 `ImageStack` (Class) 512
C.16.5 `IJ` (Class) 512
C.17 Image Properties 513
C.17.1 `ImagePlus` (Class) 514
C.18 User Interaction 515
C.18.1 `IJ` (Class) 515
C.18.2 `GenericDialog` (Class) 517
C.19 Plugins 518
C.19.1 `PlugIn` (Interface) 518
C.19.2 `PlugInFilter` (Interface) 519
C.19.3 Executing Plugins: `IJ` (Class) 520
C.20 Window Management 521
C.20.1 `WindowManager` (Class) 521
C.21 Additional Functions 522
C.21.1 `ImagePlus` (Class) 522
C.21.2 `IJ` (Class) 523

D Source Code 525
D.1 Harris Corner Detector 525
D.1.1 `Harris_Corner_Plugin` (Class) 525
D.1.2 File `Corner` (Class) 527
D.1.3 File `HarrisCornerDetector` (Class) 527
D.2 Combined Region Labeling and Contour Tracing 532
D.2.1 `Contour_Tracing_Plugin` (Class) 532
D.2.2 `Contour` (Class) 533
D.2.3 `BinaryRegion` (Class) 535
D.2.4 `ContourTracer` (Class) 536
D.2.5 `ContourOverlay` (Class) 541

References 543

Index 549

1

Crunching Pixels

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. And 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 then 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 organizations. 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 experience 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 object models [31, 37, 103]. 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 non-photorealistic rendering [77, 104]. 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 [105]. 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. 9), segmenting image regions (Ch. 11), and comparing images (Ch. 17). 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, for example, [27, 75, 98]).

Computer vision tackles the problem of engineering artificial visual systems capable of somehow comprehending and interpreting our real, three-dimensional 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, for example, [24, Ch. 13]). The fields still have much in common today, especially in terms of adaptive methods and machine learning. Further literature on computer vision includes [5, 34, 46, 90, 95, 99].

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.

2

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, and instead it focuses on images that are made up of *picture elements*, more commonly known as *pixels*, arranged in a regular rectangular grid.

2.1 Types of Digital Images

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 (Fig. 2.1). Despite all the different sources for these images, they are all, as a rule, ultimately represented as rectangular ordered arrays of image elements.

2.2 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.

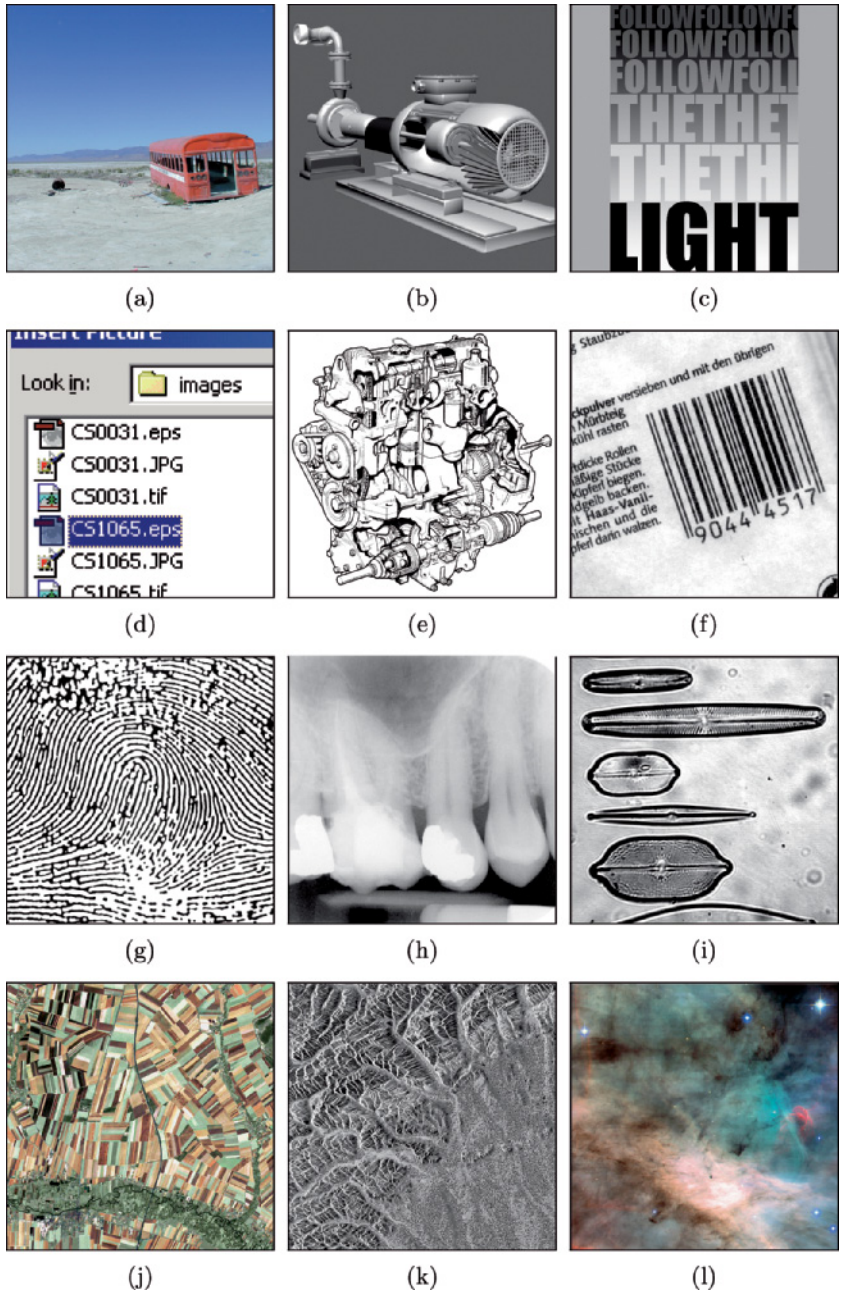


Fig. 2.1
Digital images: natural landscape (a), synthetically generated scene (b), poster graphic (c), computer screenshot (d), black and white illustration (e), barcode (f), fingerprint (g), x-ray (h), microscope slide (i), satellite image (j), radar image (k), astronomical object (l).

2.2.1 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. 2.2).

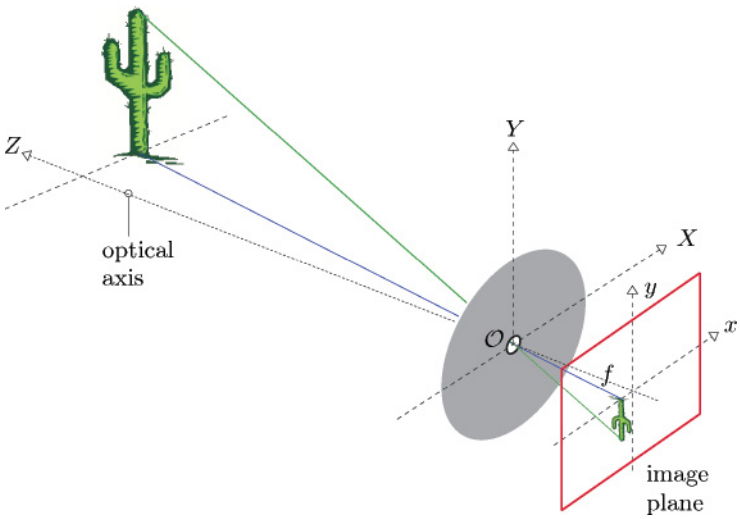


Fig. 2.2
Geometry of the pinhole camera. The pinhole opening serves as the origin ($\mathcal{O}$) of the three-dimensional coordinate system (X, Y, Z) for the objects in the scene. The optical axis, which runs through the opening, is the Z axis of this coordinate system. A separate two-dimensional coordinate 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.

Perspective transformation

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,

$$y = -f\frac{Y}{Z} \qquad \text{just as} \qquad x = -f\frac{X}{Z} \tag{2.1}$$

changes with the scale of the resulting image in proportion to the depth of the box, as well as 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. (2.1) means that the projected image is flipped in the horizontal and vertical directions and rotated by 180°. Equation (2.1) 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.

2.2.2 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. 2.3. 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 as geometrical distortions and the distinct refraction properties of different colors. So while this simple model suffices for our purposes (that is, understanding the mechanics of image acquisition), much more detailed models that incorporate these additional complexities can be found in the literature (see, for example, [59]).

2.2.3 Going Digital

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

1. The continuous light distribution must be spatially sampled.
2. This resulting function must then be sampled in the time domain to create a single image.
3. Finally, the resulting values must be quantized to a finite range of integers so that they are representable within the computer.

[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 Brunoleschi.

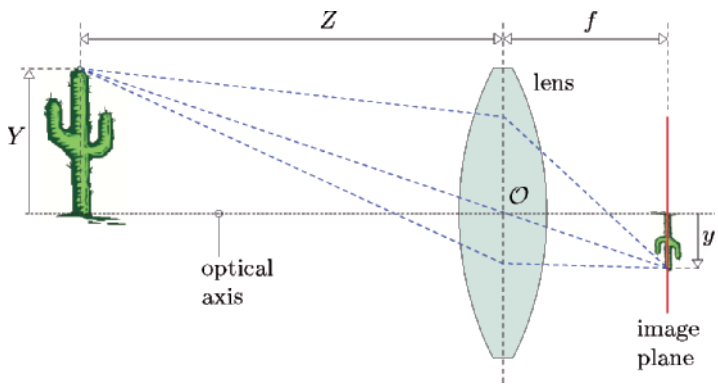


Fig. 2.3
The thin lens model.

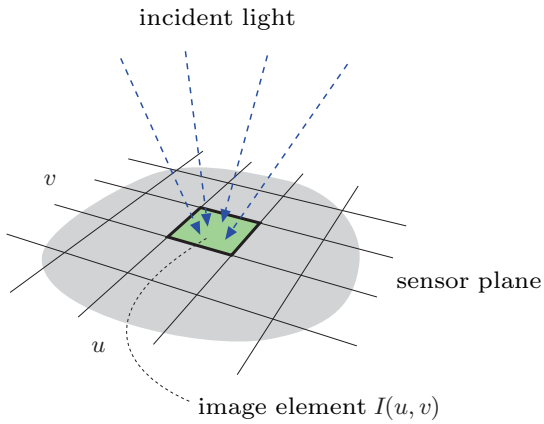


Fig. 2.4
The geometry of the sensor elements is directly responsible for the spatial sampling of the continuous image. In the simplest case, a plane of sensor elements are arranged in an evenly spaced grid, and each element measures the amount of light that falls on it.

Step 1: Spatial sampling

The spatial sampling of an image (that is, 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. 2.4). 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 CCD[2] in a digital camera does this by triggering the charging process and then measuring the amount of electrical charge that built up during the specified amount of time that the CCD was illuminated.

[2] Charge-coupled device.

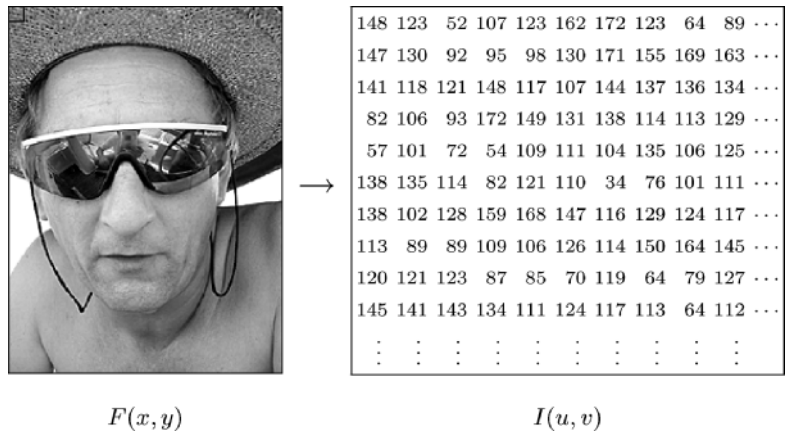


Fig. 2.5
The transformation of a continuous image $F(x, y)$ to a discrete digital image $I(u, v)$ (left), image detail (below).

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 (for example, $256 = 2^8$ or $4096 = 2^{12}$). Occasionally a floating-point scale is 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 two-dimensional, ordered matrix of integers (Fig. 2.5). Stated more formally, a digital image I is a two-dimensional function of integer coordinates $\mathbb{N} \times \mathbb{N}$ that maps a range of image 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 two-dimensional array of numerical data. Before moving on, we need a few more important definitions.

2.2.4 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

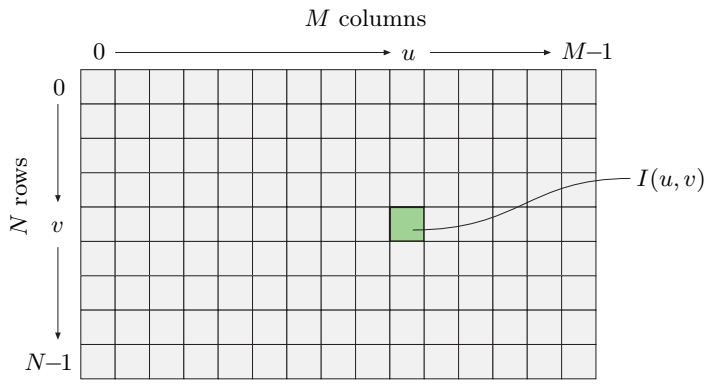


Fig. 2.6
Image coordinates. In digital image processing, it is traditional to use a coordinate system where the origin ($u = 0$, $v = 0$) lies in the upper left corner. The coordinates u, v represent respectively the columns and the rows of the image. For an image with dimensions of $M \times N$, the maximum column index is $u_{\max} = M - 1$ and the maximum row index is $v_{\max} = N - 1$.

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 because the image elements are square.

Except for a few algorithms that deal with geometrical operations, it is generally not necessary to know the spatial resolution of an image. Precise resolution information is, however, important in cases where geometrical elements such as circles need to be drawn on an image or distances within an image need to be measured. For these reasons, most image formats and software systems designed for professional use include very precise information about the resolution.

2.2.5 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 flipped in the vertical direction; that is, the y-coordinate runs from top to bottom and the origin lies in the upper left (Fig. 2.6). While this system has no practical or theoretical advantage, and in fact it makes geometrical transforms more complicated to describe, it is used almost without exception in software systems. The origin of this system lies in television electronics, where the image rows traditionally followed the direction of the electron beam, which moved from the top to the bottom of the screen. We start the numbering of rows and columns at 0 for practical reasons since in Java array indexing begins at 0.

2.2.6 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

Table 2.1 Image element ranges and their associated 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	$-32768...32767$	Whole numbers pos./neg., increased range
1	32	$\pm 3.4 \cdot 10^{38}$	Floating point: medicine, astronomy
1	64	$\pm 1.8 \cdot 10^{308}$	Floating point: internal processing

depends on the bit depth (or just depth) of the image and often the word size of the processor.

The exact bit-level layout of an individual pixel depends on the kind of image; for example, binary, grayscale, or RGB color. Common image types are summarized in Table 2.1.

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 and that cannot be negative, so typically whole integers in the range of $[0 \dots 2^k - 1]$ are used. For example, a typical grayscale image uses $k = 8$ bits (1 byte) per pixel and intensity values in the range of $[0 \dots 255]$, 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 domains. Note that bit depth refers to the number of bits used to represent a single color, not the number of bits needed to represent an entire 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.

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 by many printers.

Color images

Most color images contain encode the primary colors red, green, and blue (RGB), typically making use of 8 bits per 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 $[0 \ldots 255]$. As with intensity images, color images with depths of 30, 36, and 42 bits are often used in professional applications. While even amateur digital cameras now provide the possibility of taking images 36 bits deep, very often digital image-processing software does not fully support images with high bit depths. Finally, while most color images contain three components, in the digital prepress area there are images that use subtractive color models with four or more color components, for example the CMYK (**C**yan-**M**agenta-**Y**ellow-Blac**k**) model (see Ch. 12).

The difference between an *indexed image* or *palette image* and a *true color image* is the number of different colors (fewer for an indexed image) that can be used at one time within the color or gray component of the image. The image values themselves in these cases are only indices (with a maximum of 8 bits) in a table of color values (see Sec. 12.1.1).

Special images

Special images are those for which the standard formats already described are not 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 the detection of curves, and images with floating-point values are often found in applications such as medicine and astronomy, where extended range and precision are necessary. These special formats are almost always application-specific, and it is not possible to use them in general with other image-processing applications.

2.3 Image File Formats

While in this book we almost always consider image data as being already in the form of a two-dimensional array ready to be read 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 (that is, before around 1985), most software developers created a new custom file format for each new application they developed. The result was a chaotic jumble of incompatible file formats that for a long time limited the practical sharing of images between research groups. Today there exist a wide range of standardized file formats, and developers can almost always find at least one existing file 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 longterm. Yet for many projects the selection of the right file format is not always simple, and compromises must be made. The following are a few of the typical criteria that need to be considered when selecting an appropriate file format:

- **Type of image:** These include black and white images, grayscale images, scans from documents, color images, color graphics, and special images such as those using floating-point image data. In many applications, such as satellite imagery, the maximum image size is also an important factor.
- **Storage size and compression:** Are the storage requirements of the file a potential problem, and is the image compression method, especially when considering *lossy compression*, appropriate?
- **Compatibility:** How important is the exchange of image data? And for archives, how important is the long-term machine readability of the data?
- **Application domain:** In which domain will the image data be mainly used? Are they intended for print, Web, film, computer graphics, medicine, or astronomy?

2.3.1 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 standardformat CGM (Computer Graphics Metafile), SVG (Scalable Vector Graphics)[3] as well as proprietary formats such as DXF (Drawing Exchange Format from AutoDesk), AI (Adobe Illustrator), PICT (QuickDraw Graphics Metafilefrom Apple) and WMF/EMF

[3] www.w3.org/TR/SVG/.

(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, though they are usually used for printer output and archival purposes.[4]

2.3.2 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 now Adobe. The format supports grayscale, indexed, and true color images. 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, 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. 2.7), which enables new image types and information blocks to be created by defining new "tags". As an example, in ImageJ an image with floating-point values (`float`) can be saved as a TIFF image without any problems and then read (unfortunately only with ImageJ) again. 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 often encountered when loading TIFF files, so ImageJ can read only a few uncompressed variations of TIFF formats,[5] and bear in mind that currently no Web browser supports TIFF.

2.3.3 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, LZW compression, interlaced image loading, and ability to encode simple animations by storing a number of images in a single file for later sequential display.

[4] 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).

[5] The `ImageIO` plugin offers support for a wider range of TIFF formats (http://ij-plugins.sourceforge.net/plugins/imageio/).

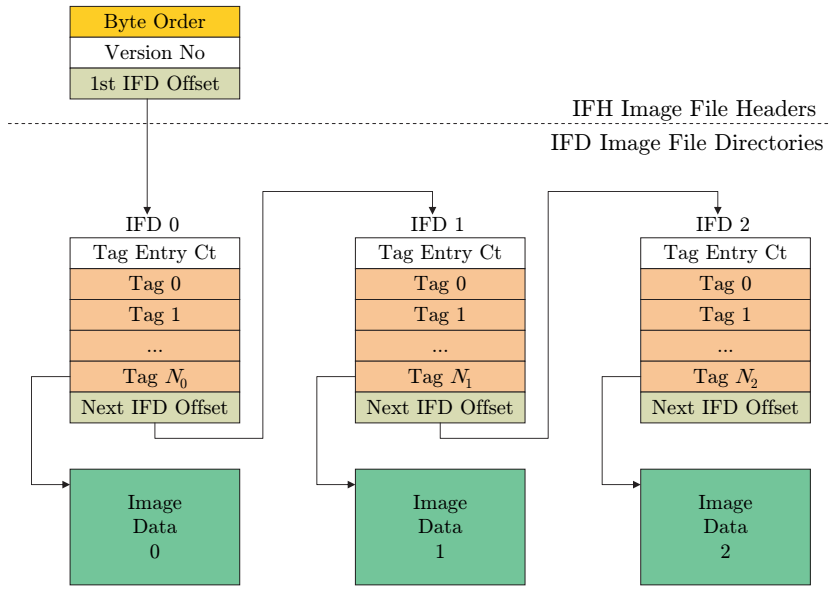


Fig. 2.7 Structure of a typical TIFF file. A TIFF file consists of a header and a linked list of image objects, three in this example. Each image object consists of a list of "tags" with their corresponding entries followed by a pointer to the actual image data.

GIF is essentially an indexed image file format designed for color and gray scale 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 palletes in the range of $2 \ldots 256$, 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 $[0 \ldots 15]$. This means that instead of storing each pixel using one 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 a lossless color quantization (see Sec. 12.5) 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 PNG format, presented in the next section, should be preferred, as it outperforms GIF by almost every metric.

2.3.4 Portable Network Graphics (PNG)

PNG (pronounced "ping") was originally developed as a replacement for the GIF file format when licensing issues[6] arose because of its use of LZW compression. It was designed as a universal image format especially for

[6] Unisys's U.S. LZW Patent No. 4,558,302 expired on June 20, 2003.

use on the Internet, and, as such, PNG supports three different types of images:

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

Additionally, PNG includes an alpha channel for transparency with a maximum width of 16 bits. In comparison, the alpha channel of a GIF image is only a single bit wide. 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 animation. Currently, PNG is the format of choice for representing uncompressed, lossless, true color images for use on the Web.

2.3.5 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)[7] 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 range of 1 bit per pixel (that is, a compression factor of around 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 makes it a relatively straightforward task [71] to create variations on 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 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.

[7] www.jpeg.org.

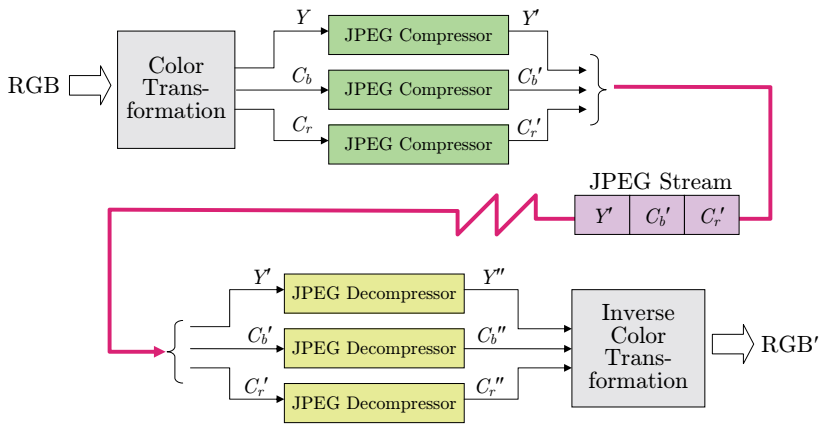


Fig. 2.8 JPEG compression of an RGB image. Using a color space transformation, the color components C_b, C_r are separated from the Y luminance component and subjected to a higher rate of compression. Each of the three components are then run independently through the JPEG compression pipeline and are merged into a single JPEG data stream. Decompression follows the same stages in reverse order.

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. 15). 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:** Next, 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 algorithm combines a number of different compression methods, and implementing even the "baseline" version is nontrivial. So application support for JPEG increased sharply once the Independent JPEG Group (IJG)[8] made a reference implementation of the JPEG algorithm available in 1991. Drawbacks of the JPEG compression algorithm include its performance on images such as line drawings, for which it was not designed, its handling of abrupt transitions within an image, and the limits that 8×8 pixel blocks impose on the compression rates. Figure 2.9 shows the results of compressing a section of a grayscale image using different quality factors (Photoshop $Q_{\text{JPG}} = 10, 5, 1$).

[8] www.ijg.org.

JFIF file format

Despite common usage, JPEG is *not* a file format; it is "only" a method of compressing image data[9] (Fig. 2.8). What is normally referred to as a JPEG file is almost always an instance of a "JPEG File Interchange Format" (JFIF) file as developed by Eric Hamilton and the IJG. The actual JPEG standard only specifies the JPEG codec (compressor and decompressor) and by design leaves the wrapping, or file format, undefined. The 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 codec.

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 and photographic parameters. EXIF was developed by the Japan Electronics and Information Technology Industries Association (JEITA) as a part of the DCF[10] 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"),[11] was designed to overcome some of the better-known weaknesses of the traditional JPEG codec. Among the improvements 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 bit/pixel on RGB color images. Despite these advantages, JPEG-2000 is supported by only a few image-processing applications and Web browsers.[12]

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

[10] Design Rule for Camera File System.

[11] www.jpeg.org/JPEG2000.htm.

[12] At this time, ImageJ does not offer JPEG-2000 support.

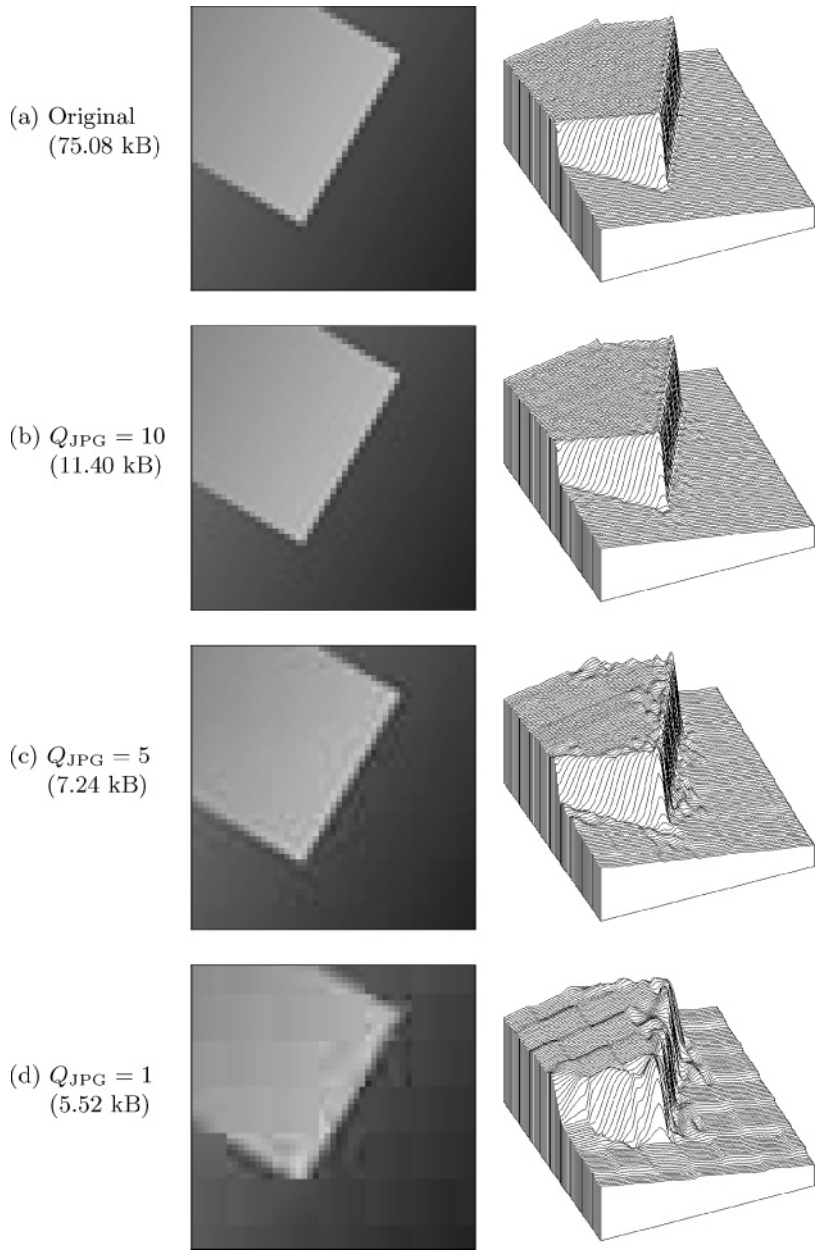


Fig. 2.9
Artifacts arising from JPEG compression. A section of the original image (a) and the results of JPEG compression at different quality factors: $Q_{\text{JPG}} = 10$ (b), $Q_{\text{JPG}} = 5$ (c), and $Q_{\text{JPG}} = 1$ (d). In parentheses are the resulting file sizes for the complete (dimensions 274×274) image.

2.3.6 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

```
P2
# oie.pgm
17 7
255
0 13 13 13 13 13 13 13  0  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  0
```

Fig. 2.10
Example of a PGM file in human-readable format (left) and the resulting image (below).

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.

2.3.7 Portable Bitmap Format (PBM)

The PBM family[13] 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. 2.10. 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 3 gives the image's dimensions, in this case width 17 and height 7, and line 4 defines the maximum pixel value, in this case `255`. The remaining lines give the actual pixel values. The actual image defined by the file is shown on the right.

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 using ImageJ.

2.3.8 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 countless other file formats, such as those encountered in legacy applications or in special

[13] http://netpbm.sourceforge.net.

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) was 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) is a family of ASCII-encoded formats used in X-Windows and is similar to PBM/PGM.

2.3.9 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 then 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 a 32-bit `int` number z with the binary and hexadecimal value[14]

$$z = \underbrace{00010010}_{\substack{12_H \\ \text{(MSB)}}}\,00110100\,01010110\,\underbrace{01111000}_{\substack{78_H \\ \text{(LSB)}}}{}_B = 12345678_H. \qquad (2.2)$$

Then $00010010_B = 12_H$ is the value of the *most significant byte* (MSB) and $01111000_B = 78_H$ 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". For the value z from Eqn. (2.2), that means:

Ordering	Byte Sequence	1	2	3	4
Big Endian	MSB → LSB	12_H	34_H	56_H	78_H
Little Endian	LSB → MSB	78_H	56_H	34_H	12_H

[14] The decimal value of z is 305419896.

Even though correctly ordering the bytes should essentially be the responsibility of the operating and file system, in practice it actually depends on the architecture of the processor.[15] 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.[16] 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, 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 2.2).

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.

Format	*Signature*	
PNG	`0x89504e47`	`□PNG`
JPEG/JFIF	`0xffd8ffe0`	`□□□□`
$\text{TIFF}_{\text{little}}$	`0x49492a00`	`II*□`
TIFF_{big}	`0x4d4d002a`	`MM□*`

Format	*Signature*	
BMP	`0x424d`	`BM`
GIF	`0x4749463839`	`GIF89`
Photoshop	`0x38425053`	`8BPS`
PS/EPS	`0x25215053`	`%!PS`

Table 2.2
Example signatures of image file formats. Most image file formats can be identified by inspecting the first bytes of the file. These bytes, or signatures, are listed in hexadecimal (`0x..`) form and as ASCII text (□ indicating a non-printable character).

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 two bytes of the file. Signatures from a number of popular image formats are given in Table 2.2. 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

[15] At least the ordering of the *bits* within a byte is almost universally uniform.
[16] In Java, this problem does not arise since internally all implementations of the *Java Virtual Machine* use big endian ordering.

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.

2.4 Exercises

Exercise 2.1. Determine the actual physical measurement in millimeters of an image with 1400 rectangular pixels and a resolution of 72 dpi.

Exercise 2.2. 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 dots per inch (dpi).

Exercise 2.3. The image sensor of a certain 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 dots per inch.

Exercise 2.4. Assume the camera geometry described in Exercise 2.3 combined with a lens with focal length $f = 50$ mm. What blurring (in pixels) would be caused by a uniform, 0.1° horizontal turn of the camera during exposure? Recompute this for $f = 300$ mm. Decide if the extent of the blurring also depends on the distance of the object.

Exercise 2.5. Determine the number of bytes necessary to store an uncompressed binary image of size 4000×3000 pixels.

Exercise 2.6. 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 2.7. 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 may 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 3×8 bits per pixel.

Exercise 2.8. Show that the projection of a 3D straight line in a pinhole camera (assuming perspective projection as defined in Eqn. (2.1)) is again a straight line in the resulting 2D image.

Exercise 2.9. Using Fig. 2.10 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.

3

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 [71, 74]. 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 by 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. 2.3), 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.

3.1 Image Manipulation and Processing

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 scipts or adding self-programmed components. For example, Adobe provides a special API[1] for programming Photoshop "plugins" in C++, though this is a nontrivial task and certainly too complex for nonprogrammers.

In contrast to the category of tools above, 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/VisiQuest,[2] IDL,[3] MatLab,[4] and ImageMagick,[5] 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.

3.2 ImageJ Overview

ImageJ, the software that is used for this book, is a combination of both worlds discussed above. 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

[1] www.adobe.com/products/photoshop/.
[2] www.accusoft.com/imaging/visiquest/.
[3] www.rsinc.com/idl/.
[4] www.mathworks.com.
[5] www.imagemagick.org.

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 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 spending much time to learn 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 [79] 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.[6] Installation is simple, with detailed instructions available online, in Werner Bailer's programming tutorial [4], and in Appendix C of this book.

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

While ImageJ is a great tool, it is naturally not perfect, likely because of its roots and history. From a software engineering point of view, its architectural design does not always seem intuitive and one could also wish for stronger orthogonality (i. e., several tasks can be accomplished in a variety of different ways). To give a structured orientation, the short reference in Appendix C is grouped into different task areas and concentrates on the key functionalities. Some specific rarely used functions were deliberately omitted, but they can of course be found in the ImageJ documentation and the (online) source code.

3.2.1 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 [4] for details). In summary, the key features of ImageJ are:

[6] http://rsb.info.nih.gov/ij/.

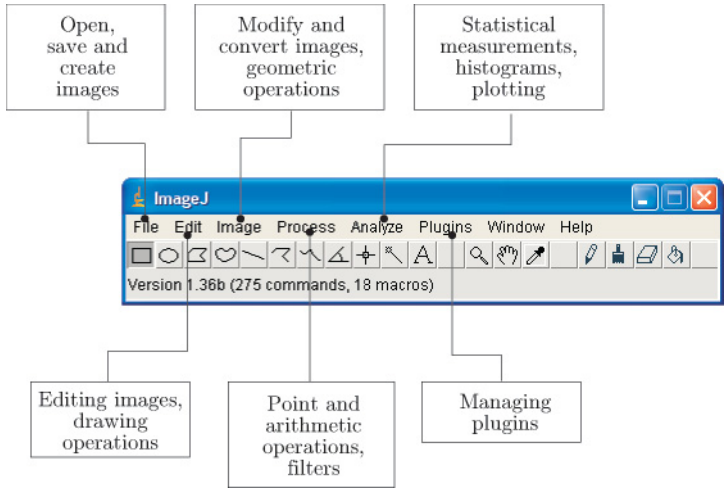


Fig. 3.1 ImageJ main window (under Windows XP).

- 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").
- 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 used in this book, but details can be found in ImageJ's online documentation.[7]

3.2.2 Interactive Tools

When ImageJ starts up, it first opens its main window (Fig. 3.1), which includes the following menu entries:

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

[7] http://rsb.info.nih.gov/ij/developer/macro/macros.html.

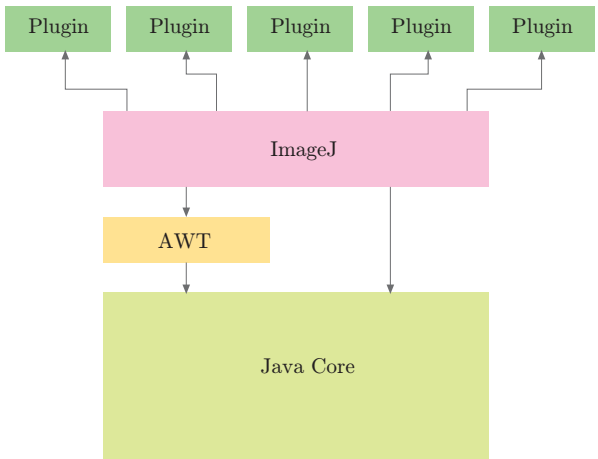


Fig. 3.2
ImageJ software structure (simplified). ImageJ is based on the Java core system and depends in particular upon Java's Advanced Windowing Toolkit (AWT) for the implementation of the user interface and the presentation of image data. Plugins are small Java classes that extend the functionality of the basic ImageJ system.

BMP, as well as the formats DICOM[8] and FITS,[9] 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, the image selected by the user. ImageJ also provides a simple (single-step) "undo" mechanism, which can also revert the results produced by user-defined plugins.

3.2.3 ImageJ Plugins

Plugins are small Java modules for extending the functionality of ImageJ by using a simple standardized interface (Fig. 3.2). Plugins can be created, edited, compiled, invoked, and organized through the Plugin menu in ImageJ's main window (Fig. 3.1). 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.

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

- `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 (`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:

[8] Digital Imaging and Communications in Medicine.
[9] Flexible Image Transport System.

`int setup (String arg, 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 (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 as 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.

3.2.4 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 5. 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 $u = 0 \ldots M-1$ and $v = 0 \ldots N-1$.

The plugin class: `My_Inverter`

We decide to name our first plugin "`My_Inverter`", which is both the name of the Java class and the name of the source file that contains it (Prog. 3.1). The underscore character ("`_`") in the name causes ImageJ to recognize this class as a plugin and 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 (Prog. 3.1, line 8). The parameters `arg` and `im` of the `setup()` method are not used in this case (see also Exercise 3.4).

Program 3.1
ImageJ plugin for inverting 8-bit grayscale images (file `My_Inverter.java`).

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

The `run()` method

As mentioned above, 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` (lines 12–13). 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 (lines 16–17). For reading and writing the pixel values, we use two additional methods of the class `ImageProcessor`:

`int getPixel (int u, int v)`
: Returns the pixel value at position (u, v) 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.

Details on these and other methods can be found in the ImageJ reference in Appendix C.

If we are sure that no coordinates outside the image bounds are ever accessed (as in `My_Inverter` in Prog. 3.1) and the inserted pixel values are guaranteed not to exceed the image processor's range, we can use

the slightly faster methods `get()` and `set()` in place of `getPixel()` and `putPixel()`, respectively (see p. 487). The most efficient way to process the image is to avoid read/write methods altogether and directly access the elements of the corresponding pixel array, as explained in Appendix C.7.6 (see p. 490).

Editing, compiling, and executing the plugin

The source code of our plugin should be stored in a file

`My_Inverter.java`

located within `<ij>/plugins/`[10] or an immediate subdirectory. New plugin files 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,[11] NetBeans,[12] or JBuilder,[13] 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.[14] 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` 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.

[10] `<ij>` denotes ImageJ's installation directory, and `<ij>/plugins/` is the default plugins path, which can be set to any other directory.
[11] www.eclipse.org.
[12] www.netbeans.org.
[13] www.borland.com.
[14] Currently only for Windows; for MacOS and Linux, consult the ImageJ installation manual.

Displaying and "undoing" results

Our first plugin in Prog. 3.1 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.[15] 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.

3.3 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.[16]

3.3.1 Resources for ImageJ

The short reference in Appendix C contains an overview of ImageJ's main capabilities and a short description of its key classes, interfaces, and methods. 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 [4].

3.3.2 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 [3,28,30]. For readers with programming experience who have not worked with Java before, we particularly

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

[16] www.imagingbook.com.

recommend some of the tutorials on Sun's Java Website.[17] Also, in Appendix B of this book, readers will find a small compilation of specific Java topics that cause frequent problems or programming errors.

3.4 Exercises

Exercise 3.1. Install the current version of ImageJ on your computer and make yourself familiar with the built-in functions (open, convert, edit, and save images).

Exercise 3.2. Write a new ImageJ plugin that reflects a grayscale image horizontally (or vertically) using `My_Inverter.java` (Prog. 3.1) as a template. Test your new plugin with appropriate images of different sizes (odd, even, extremely small) and inspect the results carefully.

Exercise 3.3. 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).

Exercise 3.4. Write a new ImageJ plugin that shifts an 8-bit grayscale image horizontally and cyclically 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. 3.1 to keep a reference and to redraw the `ImagePlus` object as follows:

```
1 public class XY_plugin implements PlugInFilter {
2
3   ImagePlus im;    // new instance variable of this plugin object
4
5   public int setup(String arg, ImagePlus im) {
6     if (im == null) {
7       IJ.noImage();  // currently no image is open
8       return DONE;
9     }
10    this.im = im;    // keep a reference to the image im
11    return DOES_8G;
12  }
13
14  public void run(ImageProcessor ip) {
15    ...   // use ip to modify the image
16    im.updateAndDraw();  // use im to redisplay this image
17    ...
18  }
19
20 } // end of class XY_plugin
```

[17] http://java.sun.com/docs/books/tutorial/.

4

Histograms

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. 4.1) 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 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.

Fig. 4.1
Digital camera viewfinder with histogram overlay.

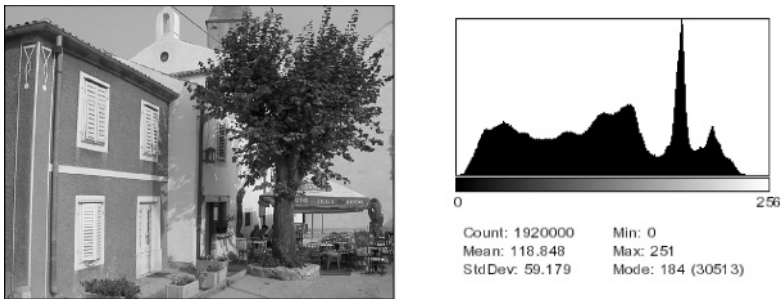


Fig. 4.2
An 8-bit grayscale image and a histogram depicting the frequency distribution of its 256 intensity values.

4.1 What Is a Histogram?

Histograms 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 that shown in Fig. 4.2. A histogram h for a grayscale image I with intensity values in the range $I(u,v) \in [0, K-1]$ would contain exactly K entries, where for a typical 8 bit grayscale image, $K = 2^8 = 256$. Each individual histogram entry is defined as

$$\mathsf{h}(i) = \text{ the } \textit{number} \text{ of pixels in } I \text{ with the intensity value } i$$

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

$$\mathsf{h}(i) = \text{card}\big\{(u,v) \mid I(u,v) = i\big\}.^1 \tag{4.1}$$

Therefore $\mathsf{h}(0)$ is the number of pixels with the value 0, $\mathsf{h}(1)$ the number of pixels with the value 1, and so forth. Finally $\mathsf{h}(255)$ is the number of all white pixels with the maximum intensity value $255 = K-1$. The result of the histogram computation is a one-dimensional vector h of length K. Figure 4.3 gives an example for an image with $K = 16$ possible intensity values.

Since a histogram encodes no information about *where* each of its individual entries originated in the image, histograms contain no information about the spatial arrangement of pixels in the image. This is intentional since the main function of a histogram is to illustrate 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. 4.4).

[1] card{...} denotes the number of elements ("cardinality") in a set (see also p. 451).

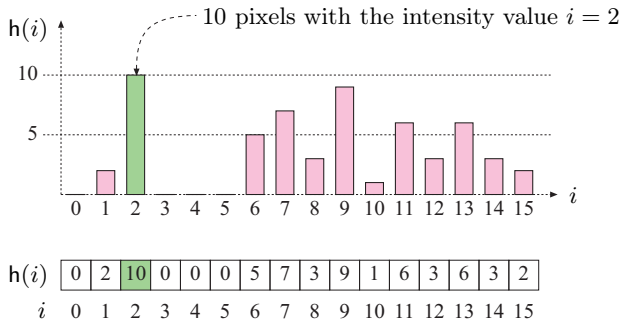


Fig. 4.3
Histogram vector for an image with $K = 16$ possible intensity values. The indices of the vector element $i = 0 \ldots 15$ represent intensity values. The value of 10 at index 2 means that the image contains 10 pixels of intensity value 2.

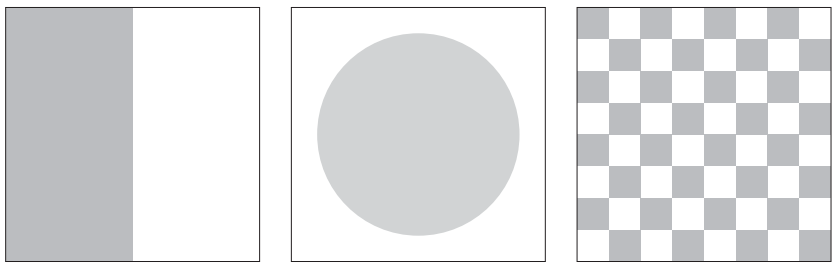


Fig. 4.4
Three very different images with identical histograms.

4.2 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. 4.5) by examining the size and uniformity of the histogram's distribution.

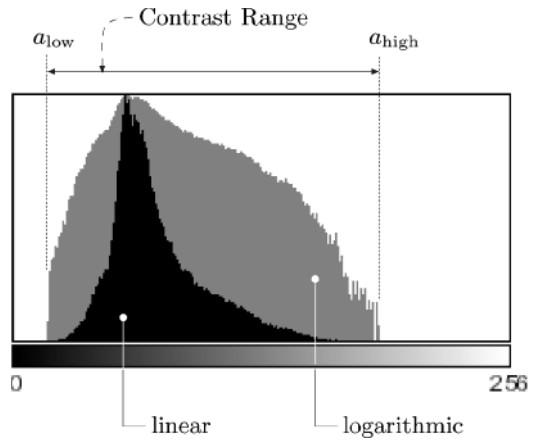
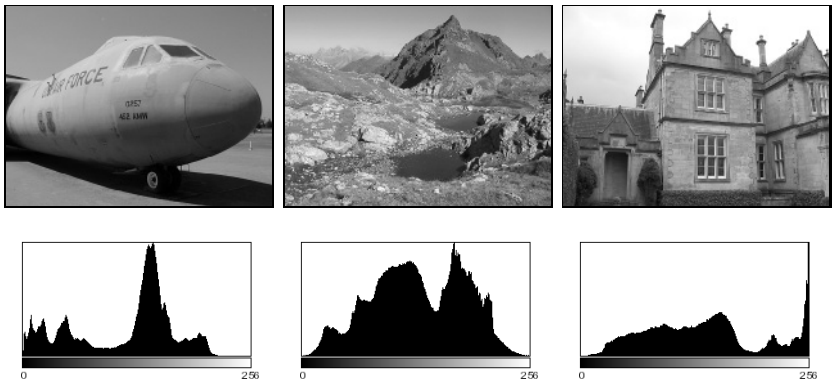


Fig. 4.5
The effective intensity range. The graph depicts how often a pixel value occurs linearly (black bars) and logarithmically (gray bars). The logarithmic form makes even relatively low occurrences, which can be very important in the image, readily apparent.



(a) (b) (c)

Fig. 4.6
Exposure errors are readily apparent in histograms. Underexposed (a), properly exposed (b), and overexposed (c) photographs.

4.2.1 Image Acquisition

Exposure

Histograms make classic exposure problems readily apparent. As an example, a histogram where a large span of the intensity range at one end is largely unused while the other end is crowded with high-value peaks (Fig. 4.6) is representative of an improperly exposed image.

Contrast

Contrast is understood as a combination of the range of intensity values *effectively* used within a given image and 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 $a = a_{\min} \ldots a_{\max} = 0 \ldots K-1$ (black to white). Using this definition, image contrast can be easily read directly from the histogram. Figure 4.7 illustrates how varying the contrast of an image affects its histogram.

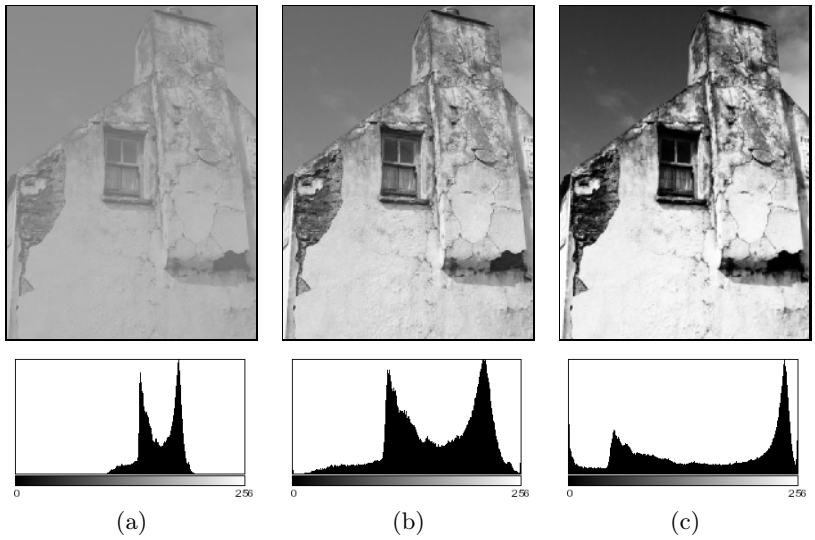
Dynamic range

The dynamic range of an image[2] is understood as the number of *distinct* pixel values in an image. In the ideal case, the dynamic range encompasses all of the usable pixel values K, in which case the value range is completely utilized. When an image has an available range of contrast $a = a_{\text{low}} \ldots a_{\text{high}}$, with

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

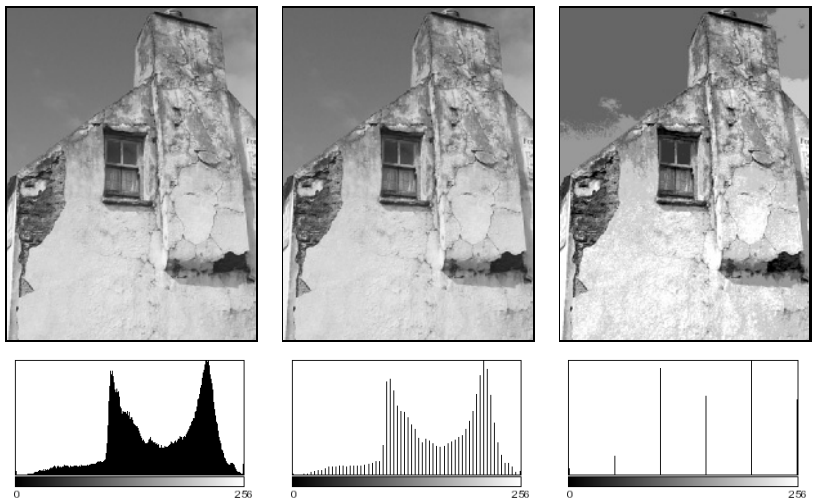
then the maximum possible dynamic range is achieved when all the intensity values lying in this range are utilized (i. e., appear in the image; Fig. 4.8).

[2] The dynamic range of the sensor used to capture an image is typically defined as the ratio of the greatest value to the least value it can generate.



(a) (b) (c)

Fig. 4.7
How changes in contrast affect a histogram: low contrast (a), normal contrast (b), high contrast (c).



(a) (b) (c)

Fig. 4.8
How changes in dynamic range affect a histogram: high dynamic range (a), low dynamic range with 64 intensity values (b), extremely low dynamic range with only 6 intensity values (c).

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 Sec. 16.3). 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 increase the dynamic range at the acquisition stage. This is done for image processing since most

of the output devices, such as monitors and printers, used to display images are unable to produce more than 256 different shades.

4.2.2 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. 4.9 (a)).

Spikes and gaps

As discussed above, 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. 5) causes the histogram lines to separate from each other and, due to

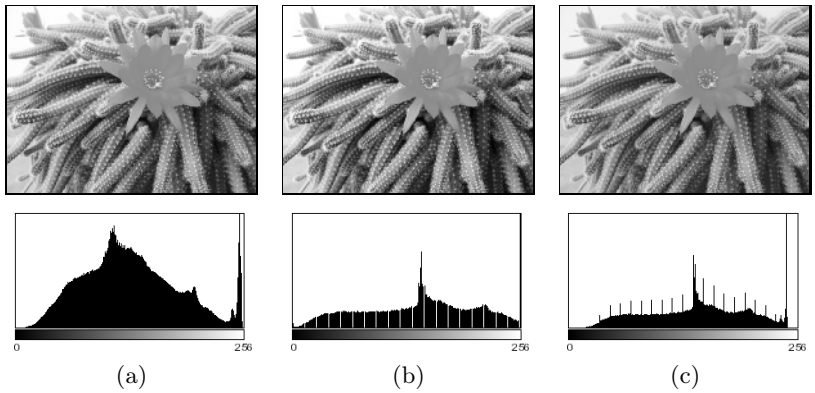


Fig. 4.9
Effect of image capture errors on histograms: saturation of high intensities (a), histogram gaps caused by a slight increase in contrast (b), and histogram spikes resulting from a reduction in contrast (c).

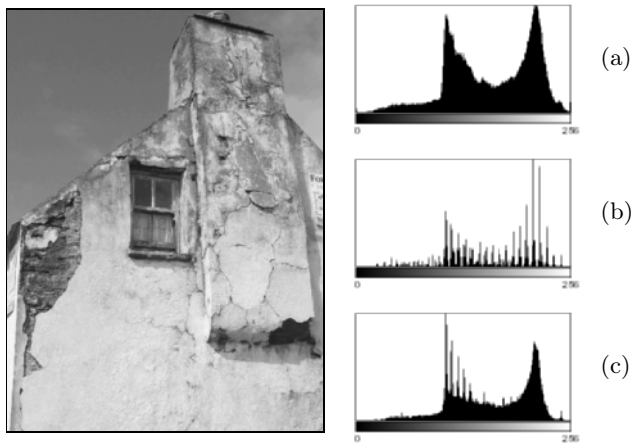


Fig. 4.10
Color quantization effects resulting from GIF conversion. The original image converted to a 256 color GIF image (left). Original histogram (a) and the histogram after GIF conversion (b). When the RGB image is scaled by 50%, some of the lost colors are recreated by interpolation, but the results of the GIF conversion remain clearly visible in the histogram (c).

the discrete values, gaps are created in the histogram (Fig. 4.9 (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. 4.9 (c)).[3]

Impact 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 or colors, resulting in an obvious line structure in the histogram that generally cannot be removed by subsequent processing (Fig. 4.10). 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,

[3] Unfortunately, these types of errors are also caused by the internal contrast "optimization" routines of some image-capture devices, especially low-end scanners.

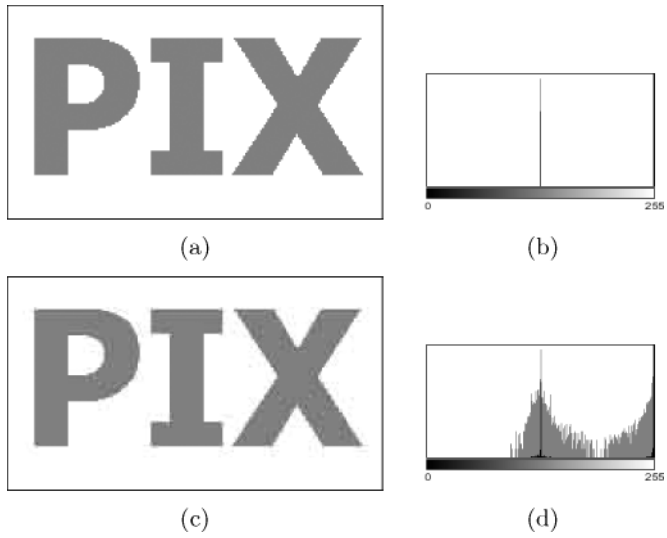


Fig. 4.11 Effect of JPEG compression. The original image (a) contained only two different gray values, as its histogram (b) makes readily apparent. JPEG compression, a poor choice for this type of image, results in numerous additional gray values, which are visible in both the resulting image (c) and its histogram (d). In both histograms, the linear frequency (black bars) and the logarithmic frequency (gray bars) are shown.

even if the image has subsequently been converted to a full-color format such as TIFF or JPEG.

Figure 4.11 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 detailed full-color 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 image[4] that appears dirty, fuzzy, and blurred.

4.3 Computing 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 $I(u, v)$, determining the pixel value p at each location, and incrementing its corresponding counter by one. At the end, each counter will contain the number of pixels in the image that have that 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 implementation would not

[4] 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, for example, Fig. 2.9 on p. 20).

Program 4.1
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 array (line 8) are automatically initialized, in this case to zero.

```
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
21 } // end of class Compute_Histogram
```

use K individual variables to represent the counters but instead would use an array with K entries (`int[]` 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 4.1 contains the complete Java source code for computing a histogram within the `run()` method of an ImageJ plugin.

At the start of Prog. 4.1, the array `H` of type `int[]` is created (line 8) and its elements are automatically initialized[5] to `0`. It makes no difference, at least in terms of the final result, whether the array is traversed in row or column order, as long as all pixels in the image are visited exactly once. In contrast to Prog. 3.1, 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.[6] Once the histogram has been calculated, it is available for further processing steps such as display.

Histogram computation has already been implemented in ImageJ and is available via the method `getHistogram()` for objects of the class

[5] 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`.

[6] In this way, image elements are traversed in exactly the same way that they are laid out in main memory, resulting in more efficient memory access and with it the possibility of increased performance, especially when dealing with larger images (see also Appendix B, p. 462).

`ImageProcessor`. If we use the method provided by ImageJ, the `run()` method of Prog. 4.1 can be simplified to

```
public void run(ImageProcessor ip) {
    int[] H = ip.getHistogram();
    ... // histogram H[] can now be used
}
```

4.4 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 number of values, for instance 16- and 32-bit or floating-point images (see Table 2.1), then th e large number of entries to be displayed makes it no longer practical to represent them directly on the screen without first taking some additional steps. As an example, applying the original histogram algorithm to a 32-bit image would require screen space to display $2^{32} = 4{,}294{,}967{,}296$ columns.

4.4.1 Binning

Since it is not possible to represent each intensity value with its own entry in the histogram, we will instead let a single 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 binning histogram of size B, each bin $\mathsf{h}(j)$ contains the number of image elements having values within the interval $a_j \le a < a_{j+1}$, and therefore (analogous to Eqn. (4.1))

$$\mathsf{h}(j) = \text{card}\,\{(u,v) \mid a_j \le I(u,v) < a_{j+1}\} \quad \text{for } 0 \le j < B. \tag{4.2}$$

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

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

4.4.2 Example

In order to create a typical histogram containing $B = 256$ entries from a 14-bit image, you would divide the available value range from $j = 0 \ldots 2^{14}{-}1$ into 256 equal intervals, each of length $k_B = 2^{14}/256 = 64$, so that $a_0 = 0$, $a_1 = 64$, $a_2 = 128$, ... $a_{255} = 16{,}320$ and $a_{256} = a_B = 2^{14} = 16{,}384 = K$. This results in the following mapping from the pixel values to the histogram bins $\mathsf{h}(0) \ldots \mathsf{h}(255)$:

$$
\begin{array}{lcrcl}
\mathsf{h}(0) & \leftarrow & 0 & \le I(u,v) < & 64 \\
\mathsf{h}(1) & \leftarrow & 64 & \le I(u,v) < & 128 \\
\mathsf{h}(2) & \leftarrow & 128 & \le I(u,v) < & 192 \\
\vdots & & \vdots & \vdots & \vdots \\
\mathsf{h}(j) & \leftarrow & a_j & \le I(u,v) < & a_{j+1} \\
\vdots & & \vdots & \vdots & \vdots \\
\mathsf{h}(255) & \leftarrow & 16320 & \le I(u,v) < & 16384
\end{array}
$$

4.4.3 Implementation

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

$$\frac{I(u,v)}{k_B} = \frac{I(u,v)}{K/B} = I(u,v) \cdot \tfrac{B}{K}. \tag{4.3}$$

As an index to the appropriate histogram bin $\mathsf{h}(j)$, we require a whole number value

$$j = \left\lfloor I(u,v) \cdot \tfrac{B}{K} \right\rfloor, \tag{4.4}$$

where $\lfloor \cdot \rfloor$ denotes the *floor* function.[7] A Java method for computing histograms by "linear binning" is given in Prog. 4.2. Note that all the computations from Eqn. (4.4) 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 floating-point result of such an operation is truncated (i. e., the floating-point value is simply cut off), which is operationally equivalent to the floor function.[8] The binning method can also be applied, in a similar way, to floating-point images.

4.5 Color Image Histograms

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.

[7] $\lfloor x \rfloor$ rounds x down to the next whole number (see Appendix A, p. 451).

[8] For a more detailed discussion, see the section on integer division in Java in Appendix B (p. 457).

Program 4.2
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.

```
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 }
```

4.5.1 Intensity Histograms

The luminance histogram $\mathsf{h}_{\mathrm{Lum}}$ 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. 256).

4.5.2 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 provide a breakdown of the intensity distribution within the individual color channels. When computing component histograms, each color channel is considered as a separate intensity image and its histogram is computed, and displayed, independently of the other channels. Figure 4.12 shows the luminance histogram $\mathsf{h}_{\mathrm{Lum}}$ 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.

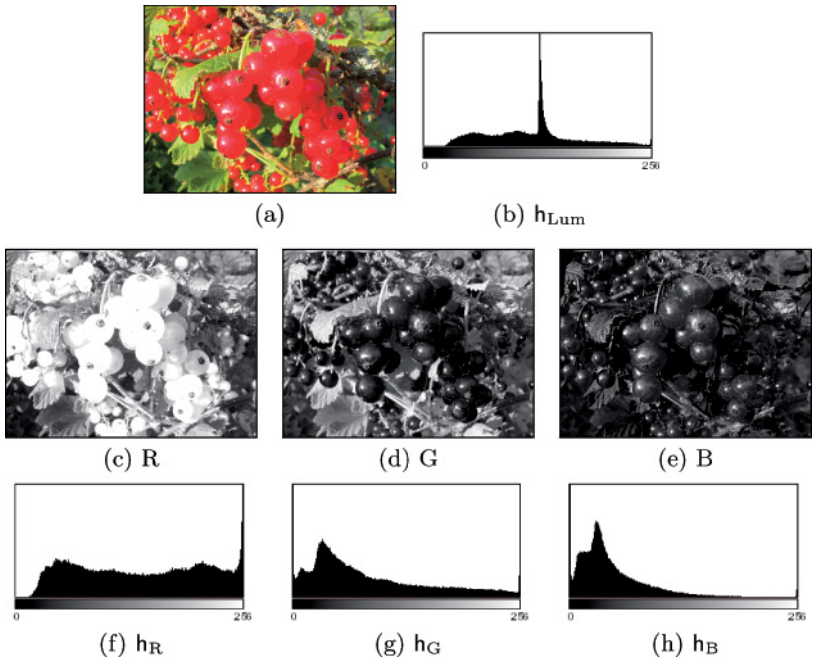


Fig. 4.12
Histograms of an RGB color image: original image (a), luminance histogram h_{Lum} (b), RGB color components as intensity images (c–e), and the associated component histograms h_R, h_G, h_B (f–h). The fact that all three color channels have saturation problems is only apparent in the individual component histograms. The spike in the distribution resulting from this is found in the middle of the luminance histogram (b).

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. 4.12 (b)).

4.5.3 Combined Color Histogram

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 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 make use of 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 these types of histograms 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 detail.

4.6 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. 5.5). The cumulative histogram $\mathsf{H}(i)$ is defined as

$$\mathsf{H}(i) = \sum_{j=0}^{i} \mathsf{h}(j) \quad \text{for } 0 \le i < K. \tag{4.5}$$

A particular value $\mathsf{H}(i)$ is thus the sum of all the values $\mathsf{h}(j)$, with $j \le i$, in the original histogram. Alternatively, we can define it recursively (as implemented in Prog. 5.2 on p. 63):

$$\mathsf{H}(i) = \begin{cases} \mathsf{h}(0) & \text{for } i = 0 \\ \mathsf{H}(i-1) + \mathsf{h}(i) & \text{for } 0 < i < K. \end{cases} \tag{4.6}$$

The cumulative histogram is a monotonically increasing function with a maximum value

$$\mathsf{H}(K-1) = \sum_{j=0}^{K-1} \mathsf{h}(j) = M \cdot N; \tag{4.7}$$

that is, the total number of pixels in an image of width M and height N. Figure 4.13 shows a concrete example of a cumulative histogram.

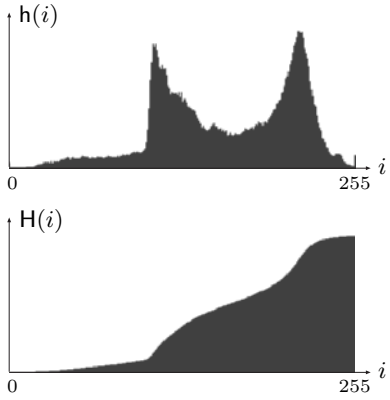


Fig. 4.13
The ordinary histogram $\mathsf{h}(i)$ and its associated cumulative histogram $\mathsf{H}(i)$.

4.7 Exercises

Exercise 4.1. In Prog. 4.2, `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 4.2. Develop an ImageJ plugin that computes the cumulative histogram of an 8-bit grayscale image and displays it as a new image, similar to $\mathsf{H}(i)$ in Fig. 4.13.
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. (4.6). 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 4.3 shows how this plugin could be set up and how a new image is created and displayed.

Exercise 4.3. Develop a technique for nonlinear binning that uses a table of interval limits a_j (Eqn. (4.2)).

Exercise 4.4. 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 4.5. 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.

Program 4.3

Creating and displaying a new image (ImageJ plugin). First, we create a `ByteProcessor` object (`histIp`, line 15) 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 25) and displayed by applying the `show()` method (line 26). Notice how the title (`String`) is retrieved from the original image inside the `setup()` method (line 5) and used to compose the new image's title (lines 24 and 25). If `histIp` is changed *after* calling `show()`, then the method `updateAndDraw()` could be used to redisplay the associated image again (line 27).

```
1  public class Create_New_Image implements PlugInFilter {
2    String title = null;
3
4    public int setup(String arg, ImagePlus im) {
5      title = im.getTitle();
6      return DOES_8G + NO_CHANGES;
7    }
8
9    public void run(ImageProcessor ip) {
10     int w = 256;
11     int h = 100;
12     int[] hist = ip.getHistogram();
13
14     // create the histogram image:
15     ImageProcessor histIp = new ByteProcessor(w, h);
16     histIp.setValue(255); // white = 255
17     histIp.fill(); // clear this image
18
19     // draw the histogram values as black bars in ip2 here,
20     // for example, using histIp.putpixel(u,v,0)
21     // ...
22
23     // display the histogram image:
24     String hTitle = "Histogram of " + title;
25     ImagePlus histIm = new ImagePlus(hTitle, histIp);
26     histIm.show();
27     // histIm.updateAndDraw();
28   }
29
30 } // end of class Create_New_Image
```

5

Point Operations

Point operations perform a mapping of the pixel values without changing the size, geometry, or local structure of the image. Each new pixel value $a' = I'(u, v)$ 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 are mapped to the new values by a function $f(a)$,

$$\begin{aligned} a' &\leftarrow f(a) \qquad \text{or} \\ I'(u, v) &\leftarrow f\big(I(u, v)\big), \end{aligned} \tag{5.1}$$

for each image position (u, v). If the function $f()$ is independent of the image coordinates (i. e., the same throughout the image), the operation is called "homogeneous". Typical examples of homogeneous point operations include, among others,

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

We will look at some of these techniques in more detail in the following. In contrast, the mapping function $g()$ for a *nonhomogeneous* point operation would also take into account the current image coordinate (u, v); i. e.,

$$\begin{aligned} a' &\leftarrow g(a, u, v) \qquad \text{or} \\ I'(u, v) &\leftarrow g\left(I(u, v), u, v\right). \end{aligned} \tag{5.2}$$

[1] If the result depends on more than one pixel value, the operation is called a "filter", as described in Ch. 6.

Program 5.1
Point operation to increase the contrast by 50% (ImageJ plugin). Note that in line 7 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 7 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 = (int) (ip.get(u, v) * 1.5 + 0.5);
8        if (a > 255)
9          a = 255;    // clamp to maximum value
10       ip.set(u, v, a);
11     }
12   }
13 }
```

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

5.1 Modifying Image Intensity

5.1.1 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

$$f_{\text{contr}}(a) = a \cdot 1.5 \qquad \text{and} \qquad f_{\text{bright}}(a) = a + 10, \tag{5.3}$$

respectively. The first operation is implemented as an ImageJ plugin by the code shown in Prog. 5.1, 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 7 (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.

5.1.2 Limiting the Results by Clamping

When implementing arithmetic operations on pixels, we must keep in mind that the computed results may exceed the maximum range of pixel values for a given image type ($[0 \ldots 255]$ in the case of 8-bit grayscale images). To avoid this, we have included the "clamping" statement

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

in line 9 of Prog. 5.1, which limits any result to the maximum value 255. Similarly one should, in general, also limit the results to the minimum value (0) to avoid negative pixel values (which cannot be represented by an 8-bit image), for example by the statement

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

This second measure is not necessary in Prog. 5.1 because the intermediate results can never be negative in this particular operation.

5.1.3 Inverting Images

Inverting an intensity image is a simple point operation that reverses the ordering of pixel values (by multiplying with -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

$$f_{\text{invert}}(a) = -a + a_{\max} \;=\; a_{\max} - a. \tag{5.4}$$

The inversion of an 8-bit grayscale image with $a_{\max} = 255$ was the task of our first plugin example in Sec. 3.2.4 (Prog. 3.1). 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. Inverting an image mirrors the histogram, as shown in Fig. 5.5 (c).

5.1.4 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 a_{th}. The threshold function $f_{\text{threshold}}(a)$ maps all pixels to one of two fixed intensity values a_0 or a_1; i. e.,

$$f_{\text{threshold}}(a) = \begin{cases} a_0 & \text{for } a < a_{\text{th}} \\ a_1 & \text{for } a \geq a_{\text{th}} \end{cases} \tag{5.5}$$

with $0 < a_{\text{th}} \leq a_{\max}$. 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` $\equiv a_{\text{th}}$, to perform this operation, which can also be invoked through the Image→Adjust→Threshold menu (see Fig. 5.1 for an example). Thresholding affects the histogram by splitting and merging the distribution into two entries at positions a_0 and a_1, as illustrated in Fig. 5.2.

5.2 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,

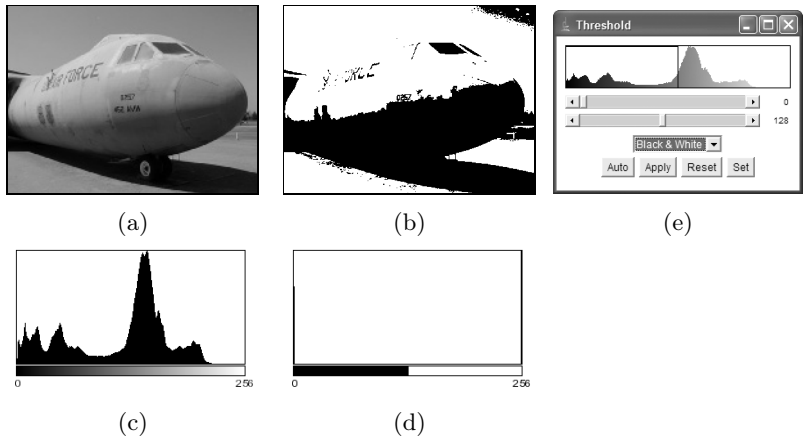


Fig. 5.1
Threshold operation: original image (a) and corresponding histogram (c); result after thresholding with $a_{\text{th}} = 128$, $a_0 = 0$, $a_1 = 255$ (b) and corresponding histogram (d); ImageJ's interactive Threshold menu (e).

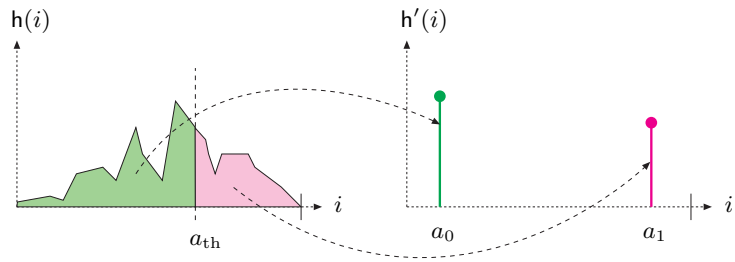


Fig. 5.2
Effects of thresholding upon the histogram. The threshold value is a_{th}. The original distribution (a) is split and merged into two isolated entries at a_0 and a_1 in the resulting histogram (b).

increasing the brightness of an image by a constant value shifts the entire histogram to the right, raising the contrast widens 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. 5.3 shows, every entry (bar) at some position i in the histogram maps to a *set* (of size $\mathsf{h}(i)$) 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 re-

[2] Of course this is only true for ordinary histograms with an entry for every single intensity value. If *binning* is used (see Sec. 4.4.1), each histogram entry maps to pixels within a certain *range* of values.

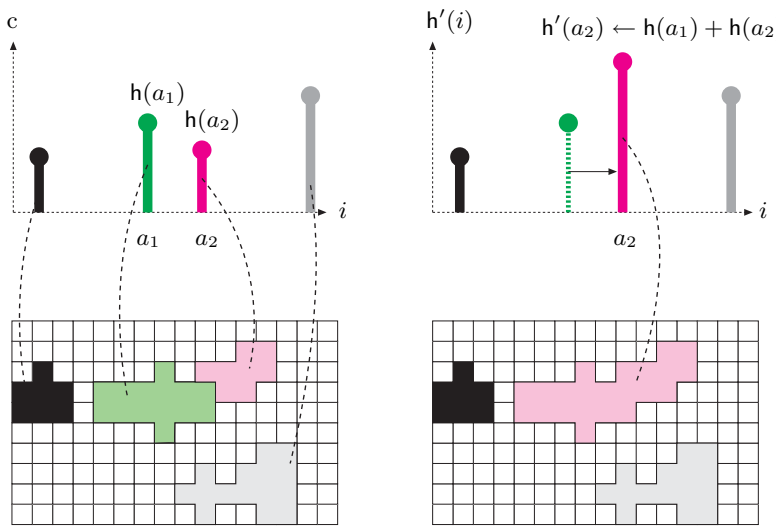


Fig. 5.3
Histogram entries map to *sets* of pixels of the same value. If a histogram line is moved as a result of some point operations, then all pixels in the corresponding set are equally modified (a). If, due to this operation, two histogram lines $\mathsf{h}(a_1)$, $\mathsf{h}(a_2)$ coincide on the same index, the two corresponding pixel sets join and the contained pixels become undiscernable (b).

duction of dynamic range and thus a permanent loss of information in that image.

5.3 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 lowest and highest available intensity values, respectively, and linearly distributing the intermediate values.

Let us assume that a_{low} and a_{high} 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. 5.4), we first map the smallest pixel value a_{low} to zero, subsequently increase the contrast by the factor $(a_{\max} - a_{\min})/(a_{\text{high}} - a_{\text{low}})$, and finally shift to the target range by adding $a_{\min}$. The mapping function for the auto-contrast operation is thus defined as

$$f_{\text{ac}}(a) = a_{\min} + \big(a - a_{\text{low}}\big) \cdot \frac{a_{\max} - a_{\min}}{a_{\text{high}} - a_{\text{low}}}, \tag{5.6}$$

provided that $a_{\text{high}} \neq a_{\text{low}}$; i. e., 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. (5.6) simplifies to

$$f_{\text{ac}}(a) = (a - a_{\text{low}}) \cdot \frac{255}{a_{\text{high}} - a_{\text{low}}}. \tag{5.7}$$

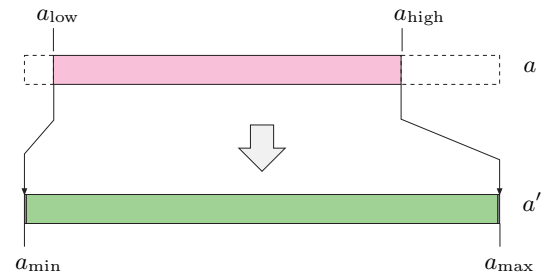
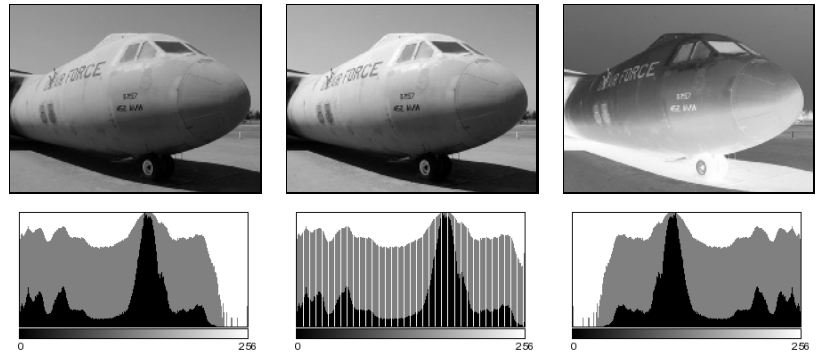


Fig. 5.4
Auto-contrast operation according to Eqn. (5.6). Original pixel values a in the range $[a_{\text{low}}, a_{\text{high}}]$ are mapped linearly to the target range $[a_{\text{min}}, a_{\text{max}}]$.



(a) (b) (c)

Fig. 5.5
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).

The target range $[a_{\text{min}}, a_{\text{max}}]$ 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 5.5 (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.

5.4 Modified Auto-Contrast

In practice, the mapping function in Eqn. (5.6) 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 (s_{low}, s_{high}) of pixels at the upper and lower ends of the target intensity range. To accomplish this, we determine two limiting values $\hat{a}_{\text{low}}$ and $\hat{a}_{\text{high}}$ such that a predefined quantile s_{low} of all pixel values in the image I is less than $\hat{a}_{\text{low}}$ and a quantile s_{high} of the values are greater than $\hat{a}_{\text{high}}$ (Fig. 5.6). The values $\hat{a}_{\text{low}}$, $\hat{a}_{\text{high}}$ depend on the image content and can be easily obtained from the image's cumulative histogram[3] $\mathsf{H}(i)$:

[3] See Sec. 4.6.

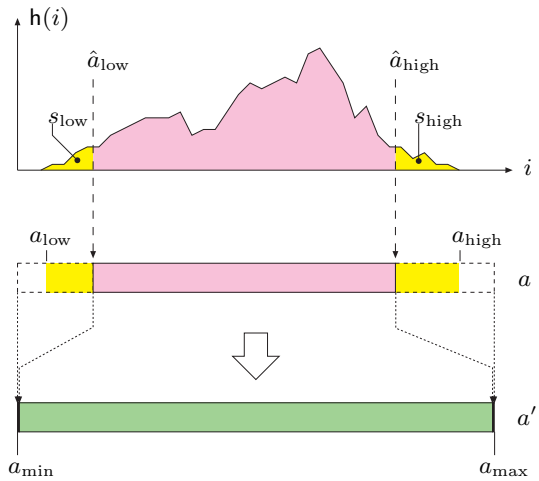


Fig. 5.6
Modified auto-contrast operation (Eqn. (5.10)). Predefined quantiles (s_{low}, s_{high}) of image pixels—shown as dark areas at the left and right ends of the histogram $\mathsf{h}(i)$—are "saturated" (i. e., mapped to the extreme values of the target range). The intermediate values ($a = \hat{a}_{\text{low}} \ldots \hat{a}_{\text{high}}$) are mapped linearly to the interval $[a_{\text{min}}, a_{\text{max}}]$.

$$\hat{a}_{\text{low}} = \min\{\, i \mid \mathsf{H}(i) \geq M \cdot N \cdot s_{\text{low}} \,\}, \tag{5.8}$$

$$\hat{a}_{\text{high}} = \max\{\, i \mid \mathsf{H}(i) \leq M \cdot N \cdot (1 - s_{\text{high}}) \,\}, \tag{5.9}$$

where $0 \leq s_{\text{low}}, s_{\text{high}} \leq 1$, $s_{\text{low}} + s_{\text{high}} \leq 1$, and $M \cdot N$ is the number of pixels in the image. All pixel values *outside* (and including) $\hat{a}_{\text{low}}$ and $\hat{a}_{\text{high}}$ are mapped to the extreme values a_{min} and a_{max}, respectively, and intermediate values are mapped linearly to the interval $[a_{\text{min}}, a_{\text{max}}]$. The mapping function $f_{\text{mac}}()$ for the modified auto-contrast operation can thus be defined as

$$f_{\text{mac}}(a) = \begin{cases} a_{\text{min}} & \text{for } a \leq \hat{a}_{\text{low}} \\ a_{\text{min}} + \big(a - \hat{a}_{\text{low}}\big) \cdot \dfrac{a_{\text{max}} - a_{\text{min}}}{\hat{a}_{\text{high}} - \hat{a}_{\text{low}}} & \text{for } \hat{a}_{\text{low}} < a < \hat{a}_{\text{high}} \\ a_{\text{max}} & \text{for } a \geq \hat{a}_{\text{high}}. \end{cases} \tag{5.10}$$

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. Usually the same value is taken for both upper and lower quantiles (i. e., $s_{\text{low}} = s_{\text{high}} = s$), with $s = 0.005 \ldots 0.015$ ($0.5 \ldots 1.5\,\%$) being common values. For example, the auto-contrast operation in Adobe Photoshop saturates $0.5\,\%$ ($s = 0.005$) 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. 5.7.

5.5 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 them in

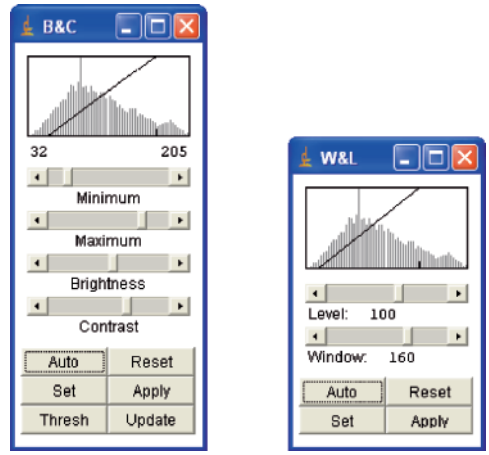


Fig. 5.7
ImageJ's Brightness/Contrast tool (left) and Window/Level tool (right) can be invoked through the Image→Adjust menu. The Auto button displays the result of a modified auto-contrast operation. Apply must be hit to actually modify the image.

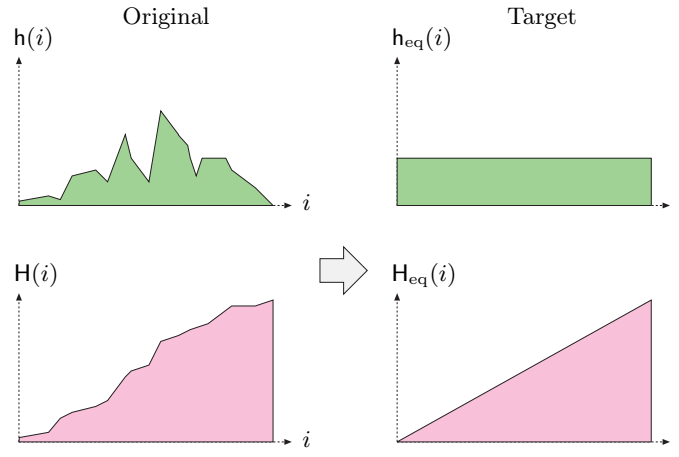


Fig. 5.8
Histogram equalization. The idea is to find and apply a point operation to the image (with original histogram h) such that the histogram h_{eq} of the modified image approximates a *uniform* distribution (top). The cumulative target histogram H_{eq} must thus be approximately wedge-shaped (bottom).

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. 5.8). 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. 4.6) of a uniformly distributed image is a linear ramp (wedge), as shown in Fig. 5.8. So we can reformulate the goal as finding a point op-

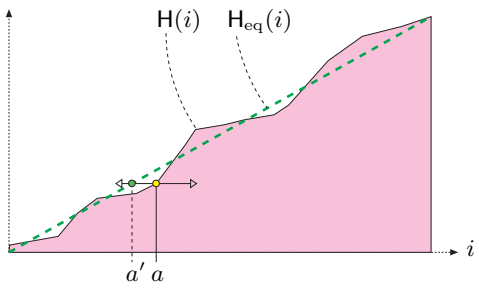


Fig. 5.9
Histogram equalization on the cumulative histogram. A suitable point operation $a' \leftarrow f_{\text{eq}}(a)$ shifts each histogram line from its original position a to a' (left or right) such that the resulting cumulative histogram H_{eq} is approximately linear.

eration that shifts the histogram lines such that the resulting cumulative histogram is approximately linear, as illustrated in Fig. 5.9.

The desired point operation $f_{\text{eq}}()$ is simply obtained from the cumulative histogram H of the original image as[4]

$$f_{\text{eq}}(a) = \left\lfloor \mathsf{H}(a) \cdot \frac{K-1}{MN} \right\rfloor, \tag{5.11}$$

for an image of size $M \times N$ with pixels in the range $[0, K-1]$. The resulting function $f_{\text{eq}}(a)$ in Eqn. (5.11) is monotonically increasing, because $\mathsf{H}(a)$ 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. (5.11). Program 5.2 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. 5.10.

Notice that for "inactive" pixel values i (i. e., pixel values that do not appear in the image, with $\mathsf{h}(i) = 0$), the corresponding entries in the cumulative histogram $\mathsf{H}(i)$ are either zero or identical to the neighboring entry $\mathsf{H}(i-1)$. Consequently a contiguous range of zero values in the histogram $\mathsf{h}(i)$ corresponds to a constant (i. e., flat) range in the cumulative histogram $\mathsf{H}(i)$, 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. 5.2).

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→Enhance Contrast menu (option Equalize). To avoid extreme contrast effects, the histogram equalization in

[4] For a derivation, see, e. g., [38, p. 173].

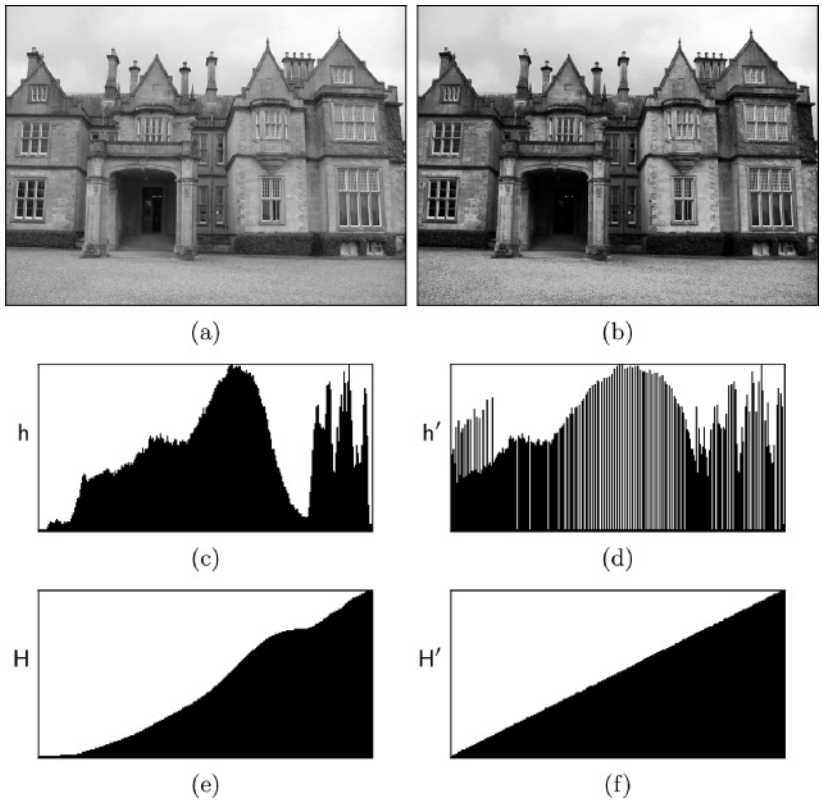


Fig. 5.10 Linear histogram equalization (example). Original image I (a) and modified image I' (b), corresponding histograms h, h' (c, d), and cumulative histograms H, H' (e, f). The resulting cumulative histogram H' (f) approximates a uniformly distributed image. Notice that new peaks are created in the resulting histogram h' (d) by merging original histogram cells, particularly in the lower and upper intensity ranges.

ImageJ by default[5] cumulates the *square root* of the histogram entries using a modified cumulative histogram of the form

$$\tilde{\mathsf{H}}(i) = \sum_{j=0}^{i} \sqrt{\mathsf{H}(i)}. \tag{5.12}$$

5.6 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.

[5] The "classic" (linear) approach is used when simultaneously keeping the Alt key pressed.

```
1  public void run(ImageProcessor ip) {
2      int w = ip.getWidth();
3      int h = ip.getHeight();
4      int M = w * h;   // total number of image pixels
5      int K = 256;     // number of intensity values
6
7      // compute the cumulative histogram:
8      int[] H = ip.getHistogram();
9      for (int j = 1; j < H.length; j++) {
10         H[j] = H[j-1] + H[j];
11     }
12
13     // equalize the image:
14     for (int v = 0; v < h; v++) {
15         for (int u = 0; u < w; u++) {
16             int a = ip.get(u, v);
17             int b = H[a] * (K-1) / M;
18             ip.set(u, v, b);
19         }
20     }
21 }
```

Program 5.2
Linear histogram equalization (ImageJ plugin). First the histogram of the image `ip` is obtained using the standard ImageJ method `ip.getHistogram()` in line 8. In line 10, the cumulative histogram is computed "in place" based on the recursive definition in Eqn. (4.6). The `int` division in line 17 implicitly performs the required floor ($\lfloor\ \rfloor$) operation by truncation.

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.

5.6.1 Frequencies and Probabilities

The value in each histogram cell described 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 $\mathsf{h}(i)$ equals the number of image pixels,

$$\sum_i \mathsf{h}(i) = M \cdot N. \tag{5.13}$$

The associated *normalized* histogram

$$\mathsf{p}(i) = \frac{\mathsf{h}(i)}{MN}, \tag{5.14}$$

for $0 \le i < K$, is usually interpreted as the *probability distribution* or *probability density function* (pdf) of a random process, where $\mathsf{p}(i)$ is

Algorithm 5.1
Computation of the cumulative distribution function (cdf) from a given histogram h of length K. See Prog. 5.3 (p. 70) for the corresponding Java implementation.

```
1:  CDF(h)
      Returns the cumulative distribution function P(i) ∈ [0,1] for a discrete
      histogram h(i), with i = 0,...K−1.
2:    Let K ← Size(h)
3:    Let n ← Σ_{i=0}^{K−1} h(i)
4:    Create table P of size K
5:    Let c ← h(0)
6:    P(0) ← c/n
7:    for i ← 1...(K−1) do
8:      c ← c + h(i)                          ▷ cumulate histogram values
9:      P(i) ← c/n
10:   return P.
```

the probability for the occurrence of the pixel value i. The cumulative probability of i being any possible value is 1, and the distribution p must thus satisfy

$$\sum_i \mathsf{p}(i) = 1. \tag{5.15}$$

The statistical counterpart to the cumulative histogram H (Eqn. (4.5)) is the discrete *distribution function* (also called the *cumulative distribution function* or cdf),

$$\begin{aligned} \mathsf{P}(i) &= \frac{\mathsf{H}(i)}{\mathsf{H}(K-1)} = \frac{\mathsf{H}(i)}{MN} = \sum_{j=0}^{i} \frac{\mathsf{h}(j)}{MN} \\ &= \sum_{j=0}^{i} \mathsf{p}(j) \qquad \text{for } 0 \le i < K. \end{aligned} \tag{5.16}$$

The computation of the cdf from a given histogram h is outlined in Alg. 5.1. The resulting function $\mathsf{P}(i)$ is (like the cumulative histogram) monotonically increasing and, in particular,

$$\mathsf{P}(0) = \mathsf{p}(0) \qquad \text{and} \qquad \mathsf{P}(K-1) = \sum_{i=0}^{K-1} \mathsf{p}(i) = 1. \tag{5.17}$$

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); i. e., each pixel value is the result of a "random experiment" on a single random variable i. The observed frequency distribution given by the histogram $\mathsf{h}(i)$ serves as a (coarse) estimate of the probability distribution $\mathsf{p}(i)$ of this random variable.

[6] Statistical modeling of the image generation process has a long tradition (see, e. g., [60, Ch. 2]).

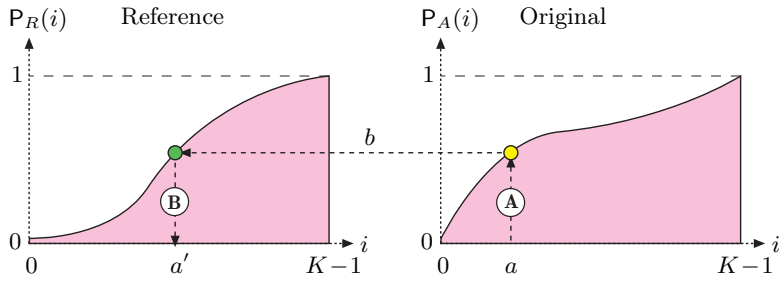


Fig. 5.11
Principle of histogram specification. Given is the reference distribution P_R (left) and the distribution function for the original image P_A (right). The result is the mapping function $f_{\text{hs}}\colon a \to a'$ for a point operation, which replaces each pixel a in the original image I_A by a modified value a'. The process has two main steps: Ⓐ For each pixel value a, determine $b = \mathsf{P}_A(a)$ from the right distribution function. Ⓑ a' is then found by inverting the left distribution function as $a' = \mathsf{P}_R^{-1}(b)$. In summary, the result is $f_{\text{hs}}(a) = a' = \mathsf{P}_R^{-1}\big(\mathsf{P}_A(a)\big)$.

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

$$a' = f_{\text{hs}}(a) \tag{5.18}$$

to convert the original image I_A to a new image $I_{A'}$ by a point operation such that

$$\mathsf{P}_{A'}(i) \approx \mathsf{P}_R(i) \qquad \text{for } 0 \le i < K. \tag{5.19}$$

As illustrated in Fig. 5.11, the desired mapping f_{hs} is found by combining the two distribution functions P_R and P_A (see [38, p. 180] for details). For a given pixel value a in the original image, we get the new pixel value a' as

$$a' = \mathsf{P}_R^{-1}\big(\mathsf{P}_A(a)\big), \tag{5.20}$$

and thus the mapping f_{hs} (Eqn. (5.18)) is obtained as

$$f_{\text{hs}}(a) = a' = \mathsf{P}_R^{-1}\big(\mathsf{P}_A(a)\big) \tag{5.21}$$

for $0 \le a < K$. This of course assumes that $\mathsf{P}_R(i)$ is invertible; i. e., that the function $\mathsf{P}_R^{-1}(b)$ exists for $b \in [0, 1]$.

5.6.3 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. (5.21) without any difficulty. In practice, it is convenient to specify the (synthetic) reference distribution as a *piecewise linear* function $\mathsf{P}_L(i)$; i. e., as a sequence of $N+1$ coordinate pairs

$$\mathcal{L} = [\langle a_0, q_0\rangle, \langle a_1, q_1\rangle, \ldots \langle a_k, q_k\rangle, \ldots \langle a_N, q_N\rangle],$$

each consisting of an intensity value a_k and the corresponding function value q_k (with $0 \le a_k < K$, $a_k < a_{k+1}$, and $0 \le q_k < 1$). The two endpoints $\langle a_0, q_0\rangle$ and $\langle a_N, q_N\rangle$ are fixed at

$$\langle 0, q_0\rangle \qquad \text{and} \qquad \langle K-1, 1\rangle,$$

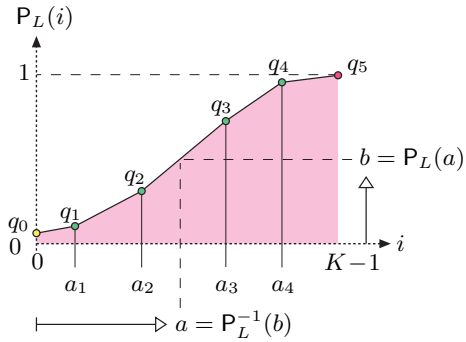


Fig. 5.12
Piecewise linear reference distribution. The function $\mathsf{P}_L(i)$ is specified by $N = 5$ control points $\langle 0, q_0\rangle$, $\langle a_1, q_1\rangle$, $\ldots$ $\langle a_4, q_4\rangle$, with $a_k < a_{k+1}$ and $q_k < q_{k+1}$. The final point q_5 is fixed at $\langle K-1, 1\rangle$.

respectively. To be invertible, the function must also be strictly monotonic; i. e., $q_k < q_{k+1}$ for $0 \le k < N$. Figure 5.12 shows an example for such a function, which is specified by $N = 5$ variable points $(q_0, \ldots q_4)$ and a fixed end point q_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 continuous values of $\mathsf{P}_L(i)$ are obtained by linear interpolation between the control points as

$$\mathsf{P}_L(i) = \begin{cases} q_m + (i - a_m) \cdot \dfrac{(q_{m+1} - q_m)}{(a_{m+1} - a_m)} & \text{for } 0 \le i < K-1 \\ 1 & \text{for } i = K-1, \end{cases} \tag{5.22}$$

where $m = \max\bigl\{j \in [0, N-1] \mid a_j \le i\bigr\}$ is the index of the line segment $\langle a_m, q_m\rangle \rightarrow \langle a_{m+1}, q_{m+1}\rangle$, which overlaps the position i. For instance, in the example in Fig. 5.12, the point a lies within the segment that starts at point $\langle a_2, q_2\rangle$; i. e., $m = 2$.

For the histogram specification according to Eqn. (5.21), we also need the *inverse* distribution function $\mathsf{P}_L^{-1}(b)$ for $b \in [0, 1]$. As we see from the example in Fig. 5.12, the function $\mathsf{P}_L(i)$ is in general not invertible for values $b < \mathsf{P}_L(0)$. We can fix this problem by mapping all values $b < \mathsf{P}_L(0)$ to zero and thus obtain a "semi-inverse" of the reference distribution in Eqn. (5.22) as

$$\mathsf{P}_L^{-1}(b) = \begin{cases} 0 & \text{for } 0 \le b < \mathsf{P}_L(0) \\ a_n + (b - q_n) \cdot \dfrac{(a_{n+1} - a_n)}{(q_{n+1} - q_n)} & \text{for } \mathsf{P}_L(0) \le b < 1 \\ K-1 & \text{for } b \ge 1. \end{cases} \tag{5.23}$$

Here $n = \max\bigl\{j \in \{0, \ldots N-1\} \mid q_j \le b\bigr\}$ is the index of the line segment $\langle a_n, q_n\rangle \rightarrow \langle a_{n+1}, q_{n+1}\rangle$, 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. (5.21), as

$$f_{\text{hs}}(a) = \mathsf{P}_L^{-1}\bigl(\mathsf{P}_A(a)\bigr) \qquad \text{for } 0 \le a < K. \tag{5.24}$$

Algorithm 5.2
Histogram specification using a piecewise linear reference distribution. Given is the histogram h_A of the original image and a piecewise linear reference distribution function, specified as a sequence of N control points $\mathcal{L}_R$. The discrete mapping function f_{hs} for the corresponding point operation is returned.

1: PiecewiseLinearHistogram(h_A, $\mathcal{L}_R$)
   h_A: histogram of the original image.
   $\mathcal{L}_R$: reference distribution function, given as a sequence of $N+1$ control points $\mathcal{L}_R = [\langle a_0, q_0\rangle, \langle a_1, q_1\rangle, \ldots \langle a_N, q_N\rangle]$, with $0 \le a_k < K$ and $0 \le q_k \le 1$.
2: Let $K \leftarrow \text{Size}(\mathsf{h}_A)$
3: Let $\mathsf{P}_A \leftarrow \text{Cdf}(\mathsf{h}_A)$ ▷ cdf for h_A (Alg. 5.1)
4: Create a table $f_{\text{hs}}[\,]$ of size K ▷ mapping function f_{hs}
5: **for** $a \leftarrow 0 \ldots (K-1)$ **do**
6: $b \leftarrow \mathsf{P}_A(a)$
7: **if** $(b \le q_0)$ **then**
8: $a' \leftarrow 0$
9: **else if** $(b \ge 1)$ **then**
10: $a' \leftarrow K-1$
11: **else**
12: $n \leftarrow N-1$
13: **while** $(n \ge 0) \wedge (q_n > b)$ **do** ▷ find line segment in $\mathcal{L}_R$
14: $n \leftarrow n-1$
15: $a' \leftarrow a_n + (b-q_n)\cdot\dfrac{(a_{n+1}-a_n)}{(q_{n+1}-q_n)}$ ▷ see Eqn. (5.23)
16: $f_{\text{hs}}[a] \leftarrow a'$
17: **return** f_{hs}.

The whole process of computing the pixel mapping function for a given image (histogram) and a piecewise linear target distribution is summarized in Alg. 5.2. A real example is shown in Fig. 5.14 (Sec. 5.6.5).

5.6.4 Adjusting to a Given Histogram (Histogram Matching)

If we want to adjust one image to the histogram of another image, the reference distribution function $\mathsf{P}_R(i)$ is not continuous and thus, in general, cannot be inverted (as required by Eqn. (5.21)). For example, if the reference distribution contains zero entries (i. e., pixel values k with probability $\mathsf{p}(k) = 0$), the corresponding cumulative distribution function 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. 5.13. The mapping function f_{hs} is not obtained by inverting but by "filling in" the reference distribution function $\mathsf{P}_R(i)$. For each possible pixel value a, starting with $a = 0$, the corresponding probability $\mathsf{p}_A(a)$ is stacked layer by layer "under" the reference distribution P_R. The thickness of each horizontal bar for a equals the corresponding probability $\mathsf{p}_A(a)$. The bar for a particular intensity value a with thickness $\mathsf{p}_A(a)$ 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.

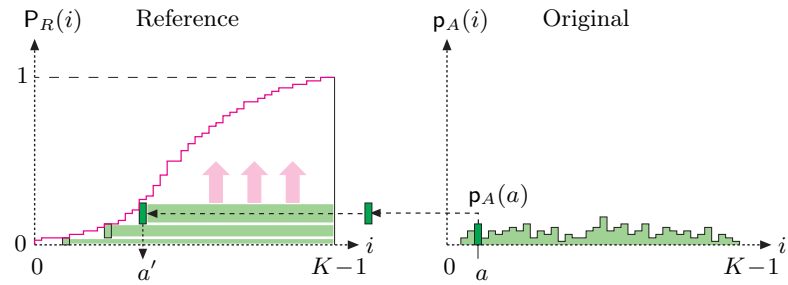


Fig. 5.13
Discrete histogram specification. The reference distribution P_R (left) is "filled" layer by layer from bottom to top and from right to left. For every possible intensity value a (starting from $a = 0$), the associated probability $\mathsf{p}_A(a)$ is added as a horizontal bar to a stack accumulated 'under" the reference distribution P_R. The bar with thickness $\mathsf{p}_A(a)$ is drawn from right to left down to the position a', where the reference distribution P_R is reached. This value a' is the one to which a should be mapped by the function $f_{\text{hs}}(a)$.

Since the sum of all probabilities p_A and the maximum of the distribution function P_R are both 1 (i. e., $\sum_i \mathsf{p}_A(i) = \max_i \mathsf{P}_R(i) = 1$), all horizontal bars will exactly fit underneath the function P_R. One may also notice in Fig. 5.13 that the distribution value resulting at a' is identical to the cumulated probability $\mathsf{P}_A(a)$. Given some intensity value a, it is therefore sufficient to find the minimum value a', where the reference distribution $\mathsf{P}_R(a')$ is greater than or equal to the cumulative probability $\mathsf{P}_A(a)$; i. e.,

$$f_{\text{hs}}(a) = a' = \min\big\{\, j \mid (0 \le j < K) \wedge \big(\mathsf{P}_A(a) \le \mathsf{P}_R(j)\big)\big\}. \qquad (5.25)$$

This results in a very simple method, which is summarized in Alg. 5.3. Due to the use of normalized distribution functions, the *size* of the images involved is not relevant. The corresponding Java implementation in Prog. 5.3, consists of the method `matchHistograms()`, which accepts the original histogram (`Ha`) and the reference histogram (`Hr`) and returns the resulting mapping function (`map`) specifying the required point operation. The following code fragment demonstrates the use of the method `matchHistograms()` from Prog. 5.3 in an ImageJ program:

```
ImageProcessor ipA = ...    // target image I_A (to be modified)
ImageProcessor ipR = ...    // reference image I_R
int[] hA = ipA.getHistogram(); // get the histogram for I_A
int[] hR = ipR.getHistogram(); // get the histogram for I_R
int[] F = matchHistograms(hA, hR); // mapping function f_hs(a)
ipA.applyTable(F);          // apply f_hs() to the target image I_A
```

The original image `ipA` is modified in the last line by applying the mapping function f_{hs} (`F`) with the method `applyTable()` (see also p. 80).

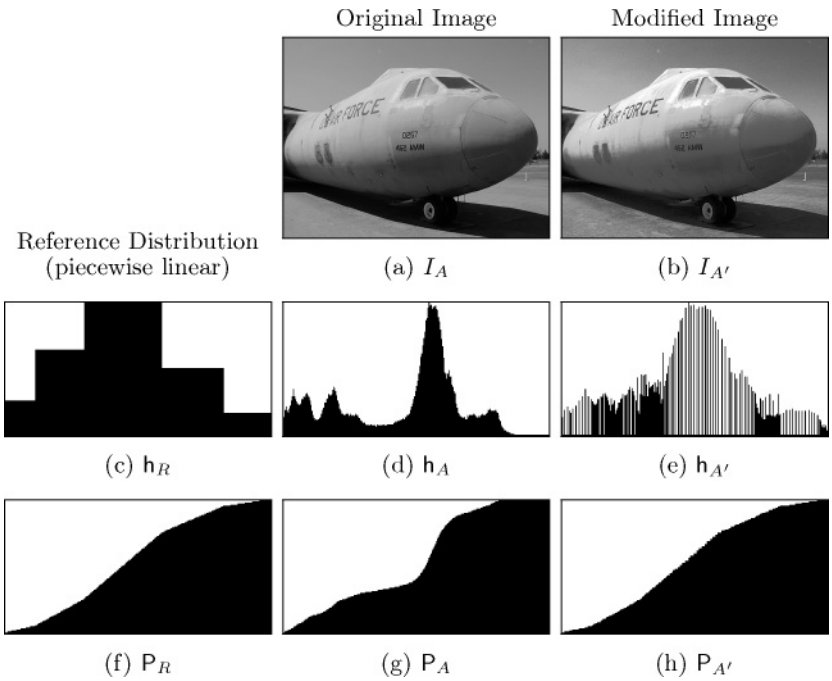
5.6.5 Examples

Adjusting to a piecewise linear reference distribution

The first example in Fig. 5.14 shows the results of histogram specification for a continuous, piecewise linear reference distribution, as described in Sec. 5.6.3. Analogous to Fig. 5.12, the actual distribution function P_R (Fig. 5.14 (f)) is specified as a polygonal line consisting of five control points $\langle a_k, q_k \rangle$ with coordinates

Algorithm 5.3
Histogram matching. Given are two histograms: the histogram h_A of the target image I_A and a reference histogram h_R, both of size K. The result is a discrete mapping function $f_{\text{hs}}()$ that, when applied to the target image, produces a new image with a distribution function similar to the reference histogram.

```
1:  MatchHistograms(h_A, h_R)
      h_A: histogram of the target image
      h_R: reference histogram (of same size as h_A)
2:    Let K ← Size(h_A)
3:    Let P_A ← Cdf(h_A)                          ▷ cdf for h_A (Alg. 5.1)
4:    Let P_R ← Cdf(h_R)                          ▷ cdf for h_R (Alg. 5.1)
5:    Create a table f_hs[ ] of size K            ▷ pixel mapping function f_hs
6:    for a ← 0 ... (K−1) do
7:      j ← K−1
8:      repeat
9:        f_hs[a] ← j
10:       j ← j − 1
11:     while (j ≥ 0) ∧ (P_A(a) ≤ P_R(j))
12:   return f_hs.
```



$k =$	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

Fig. 5.14
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 $\mathsf{h}_{A'}$ (e), and the resulting distribution $\mathsf{P}_{A'}$ (h).

The resulting reference histogram (Fig. 5.14 (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. 5.14 (h)) is quite close to the reference function in Fig. 5.14 (f), while the resulting *histogram* (Fig. 5.14 (e))

Program 5.3 Histogram matching (Java implementation of Alg. 5.3). The method `matchHistograms()` computes the mapping function `F` from the target histogram `Ha` and the reference histogram `Hr` (see Eqn. (5.25)). The method `Cdf()` computes the cumulative distribution function (cdf) for a given histogram (Eqn. (5.16)).

```
1  int[] matchHistograms (int[] hA, int[] hR) {
2    // hA ... histogram h_A of target image I_A
3    // hR ... reference histogram h_R
4    // returns the mapping function f_hs() to be applied to image I_A
5
6    int K = hA.length;          // hA, hR must be of length K
7    double[] PA = Cdf(hA);      // get CDF of histogram h_A
8    double[] PR = Cdf(hR);      // get CDF of histogram h_R
9    int[] F = new int[K];       // pixel mapping function 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       F[a] = j;
16       j--;
17     } while (j>=0 && PA[a]<=PR[j]);
18   }
19   return F;
20 }
```

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

shows little similarity with the reference histogram (Fig. 5.14 (c)). However, as discussed earlier, this is all we can expect from a homogeneous point operation.

Adjusting to an arbitrary reference histogram

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. 5.6.4 to compute the required mapping functions. The

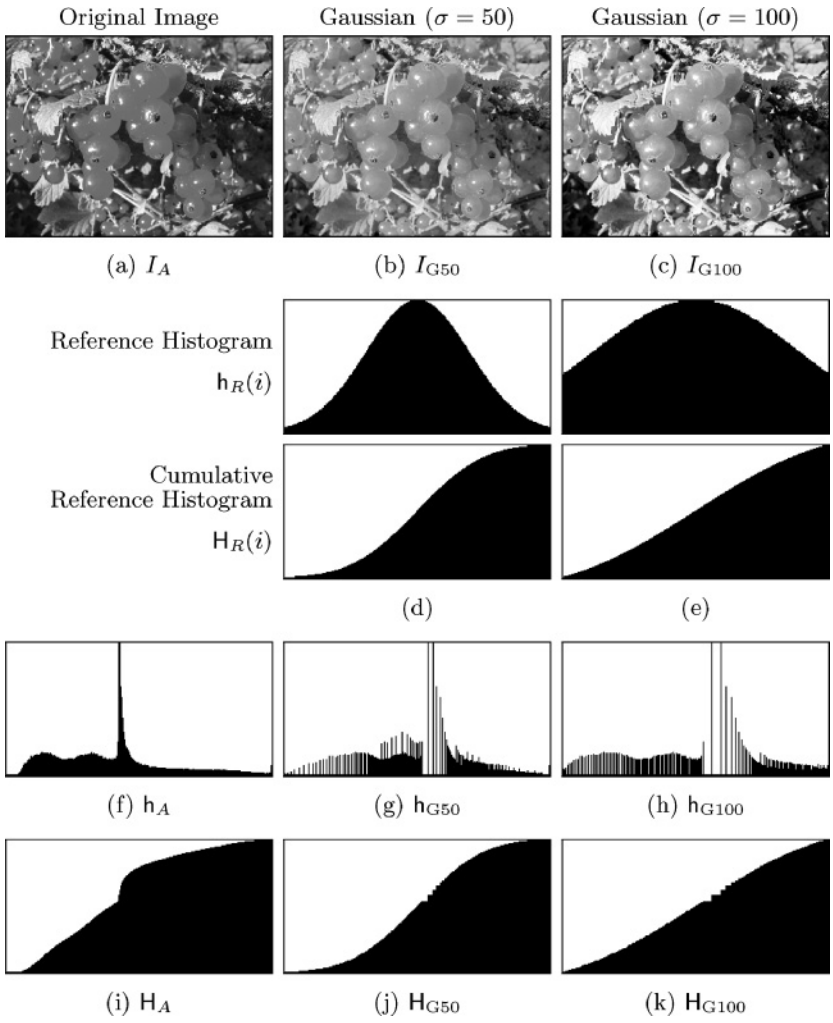


Fig. 5.15
Histogram matching: adjusting to a synthetic histogram. Original image I_A (a), corresponding histogram (f), and cumulative histogram (i). Gaussian-shaped reference histograms with center $\mu = 128$ and $\sigma = 50$ (d) and $\sigma = 100$ (e), respectively. Resulting images after histogram matching, I_{G50} (b) and I_{G100} (c) with the corresponding histograms (g, h) and cumulative histograms (j, k).

examples in Fig. 5.15 demonstrate this technique using synthetic reference histograms whose shape is approximately Gaussian.

The target image (Fig. 5.15 (a)) used here was chosen intentionally for its poor quality, manifested by an extremely unbalanced histogram (Fig. 5.15 (f)). The histograms of the modified images thus naturally show little resemblance to a Gaussian. However, the resulting *cumulative* histograms (Fig. 5.15 (j, k)) match nicely with the integral of the corresponding Gaussians (Fig. 5.15 (d, e)), 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. 5.16 demonstrates the adjustment of two images by matching their intensity histograms. One of the images is selected as the reference image I_R (Fig. 5.16 (b)) and supplies the reference histogram h_R (Fig. 5.16 (e)). The second (target) image I_A (Fig. 5.16 (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. 5.16 (c)) and the reference image give a similar visual impression with regard to tonal range and distribution (assuming that both images show similar content).

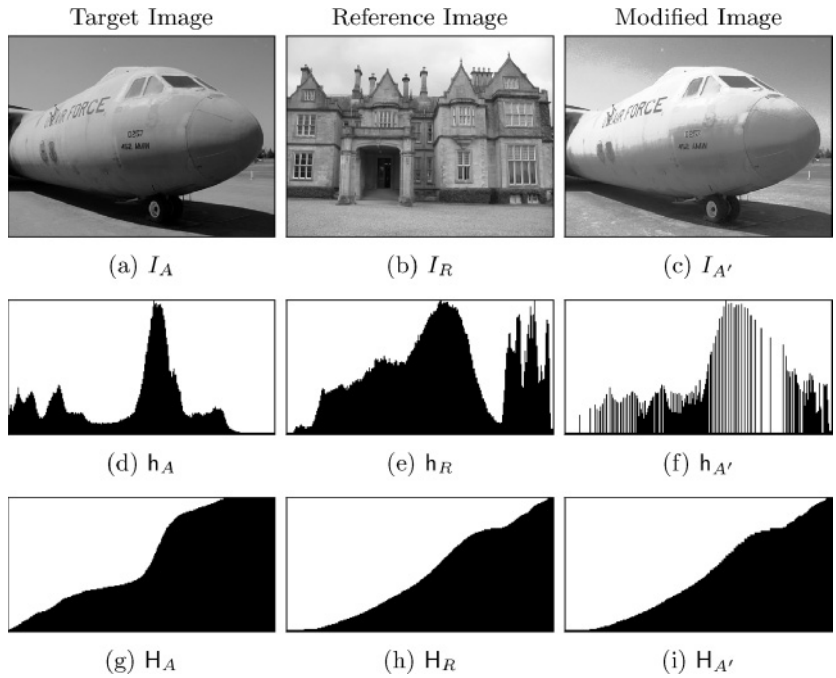


Fig. 5.16
Histogram matching: adjusting to a reference image. The target image I_A (a) is modified by matching its histogram to the reference image I_R (b), resulting in the new image $I_{A'}$ (c). The corresponding histograms h_A, h_R, $\mathsf{h}_{A'}$ (d–f) and cumulative histograms H_A, H_R, $\mathsf{P}_{A'}$ (g–i) are shown. Notice the good agreement between the cumulative histograms of the reference and adjusted images (h, i).

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 5.7).

5.7 Gamma Correction

We have been using the terms "intensity" and "brightness" many times without really bothering with how the numeric pixel values in our im-

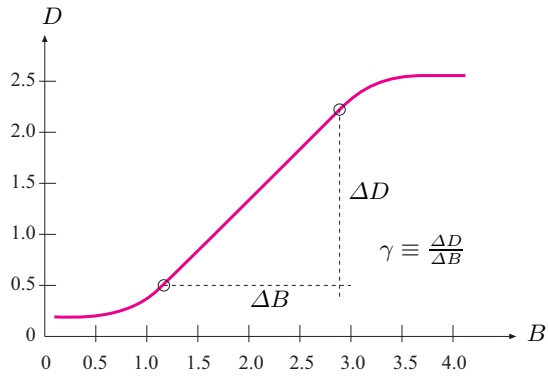


Fig. 5.17
Exposure function of photographic film. With respect to the *logarithmic* light intensity B, the resulting film density D is approximately *linear* over a wide intensity range. The slope ($\Delta D/\Delta B$) of this linear section of the function specifies the "gamma" (γ) value for a particular type of photographic material.

ages 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.

5.7.1 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. 5.17), 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 television broadcasting to describe the nonlinearities of the cathode ray tubes used in TV receivers, i. e., 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.

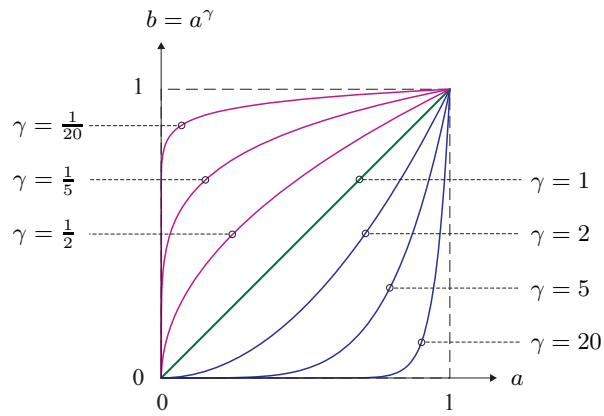


Fig. 5.18
Gamma function $b = f_\gamma(a) = a^\gamma$ for $a \in [0,1]$ for different gamma values.

5.7.2 The Gamma Function

Gamma correction is based on the gamma *function*

$$b = f_\gamma(a) = a^\gamma \quad \text{for } a \in \mathbb{R}\,, \gamma > 0, \tag{5.26}$$

where the parameter γ is called the gamma *value*. If a is limited to the interval $[0,1]$, then—independent of γ—the value of the gamma function 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. 5.18. 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 gamma 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

$$a = f_\gamma^{-1}(b) = b^{1/\gamma}, \tag{5.27}$$

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

$$f_\gamma^{-1}(b) = f_{\bar\gamma}(b), \tag{5.28}$$

with $\bar\gamma = 1/\gamma$. Thus the inverse of the gamma function with parameter γ is another gamma function with parameter $\bar\gamma = 1/\gamma$.

5.7.3 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 $1.8 - -2.8$, 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 EBU[7] standards. The current international standard ITU-R BT.709[8] calls for uniform gamma values of 2.5 in receivers and $1/1.956 \approx 0.51$ for cameras [32, 55]. The ITU 709 standard is based on a slightly modified version of the gamma function (see Sec. 5.7.6).

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 gamma function 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. 12.3.6).

5.7.4 Applications of Gamma Correction

Let us first look at a simple example. 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 energy B as

$$s = B^{\gamma_c}. \tag{5.29}$$

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

$$b = f_{\bar{\gamma}_c}(s) = s^{1/\gamma_c}. \tag{5.30}$$

The resulting signal $b = s^{1/\gamma_c} = (B^{\gamma_c})^{1/\gamma_c} = B^{(\gamma_c \frac{1}{\gamma_c})} = B^1$ is obviously proportional (in theory even identical) to the original light intensity B. Although this example is overly simplistic, it still demonstrates the general rule, which holds for output devices as well:

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

In the above, 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; e. g., in the range $[0, 255]$ for 8-bit images. In general, performing a gamma correction

[7] European Broadcast Union (EBU).
[8] International Telecommunications Union (ITU).

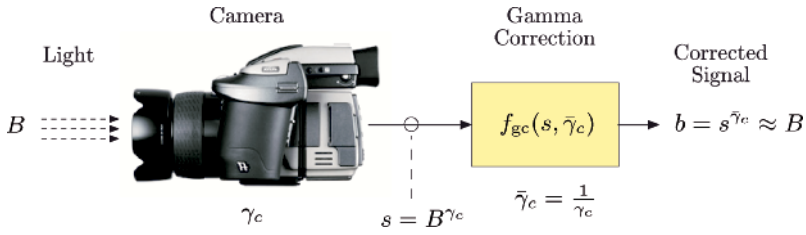


Fig. 5.19 Principle of gamma correction. To compensate the output signal s produced by a camera with nominal gamma value γ_c, a gamma correction is applied with $\bar{\gamma}_c = 1/\gamma_c$. The corrected signal b is proportional to the received light intensity B.

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

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 function to $\hat{a}$: $\hat{b} \leftarrow f_\gamma(\hat{a}) = \hat{a}^\gamma$.
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

$$b \leftarrow f_{gc}(a, \gamma) = \left(\frac{a}{a_{max}}\right)^\gamma \cdot a_{max}. \tag{5.31}$$

Figure 5.20 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.

5.7.5 Implementation

Program 5.4 shows the implementation of gamma correction as an ImageJ plugin for 8-bit grayscale images. The mapping function $f_{gc}(a, \gamma)$ 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. 5.8.1).

5.7.6 Modified Gamma Correction

A subtle problem with the simple gamma function $f_\gamma(a) = a^\gamma$ (Eqn. (5.26)) 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

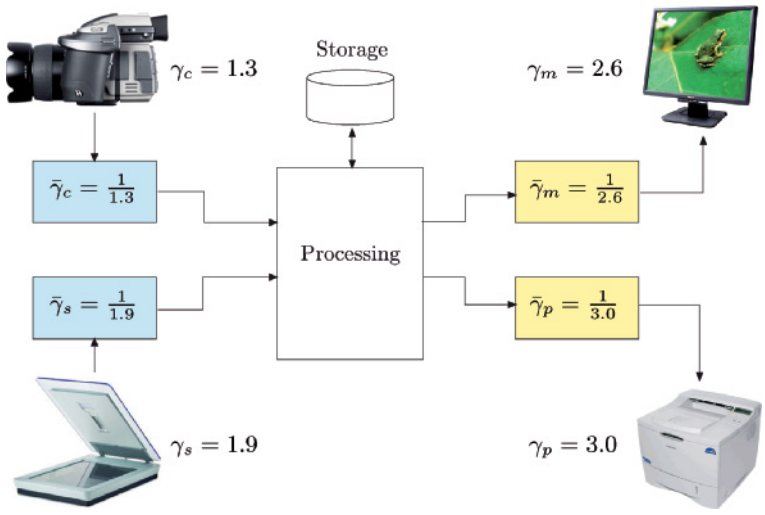


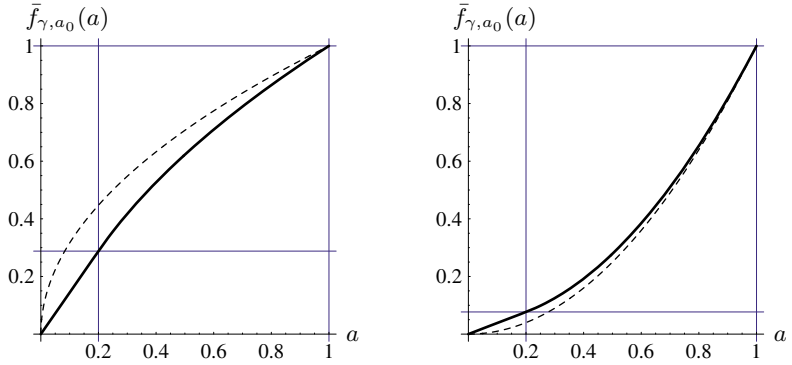
Fig. 5.20
Gamma correction in the digital imaging work flow. Images are processed and stored in a "linear" intensity space, where gamma correction is used to compensate for the transfer characteristic of each input and output device. (The gamma values shown are examples only.)

```
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 a lookup table for the mapping function
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); // gamma 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 ip
19 }
```

Program 5.4
Gamma correction (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).

$$f'_\gamma(0) = \begin{cases} 0 & \text{for } \gamma > 1 \\ 1 & \text{for } \gamma = 1 \\ \infty & \text{for } \gamma < 1. \end{cases} \tag{5.32}$$

The tangent to the function at the origin is thus either 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 gamma function is generally not invertible at the origin.



(a) $\gamma = 0.5$, $a_0 = 0.2$ (b) $\gamma = 2.0$, $a_0 = 0.2$

Fig. 5.21 The modified gamma function $\bar{f}_{\gamma,a_0}()$ consists of a linear segment with fixed slope s between $a = 0$ and $a = a_0$, followed by an ordinary gamma function with parameter γ (Eqn. (5.33)). The dashed lines show the ordinary gamma functions for the same gamma values.

A common solution to this problem is to replace the lower part ($0 \le a \le a_0$) of the gamma function by a linear segment with constant slope and to continue with the ordinary gamma function for $a > a_0$. The resulting modified gamma function $\bar{f}_{(\gamma,a_0)}(a)$ is defined as

$$\bar{f}_{\gamma,a_0}(a) = \begin{cases} s \cdot a & \text{for } 0 \le a \le a_0 \\ (1+d) \cdot a^{\gamma} - d & \text{for } a_0 < a \le 1 \end{cases} \tag{5.33}$$

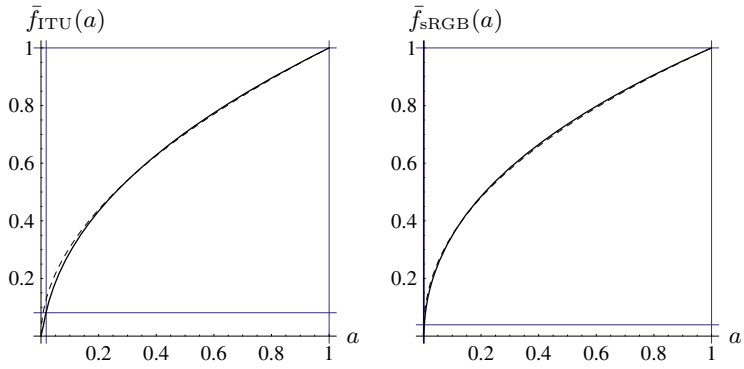
$$\text{with} \quad s = \frac{\gamma}{a_0(\gamma-1) + a_0^{(1-\gamma)}} \qquad \text{and} \qquad d = \frac{1}{a_0^{\gamma}(\gamma-1)+1} - 1. \tag{5.34}$$

The function thus consists of a *linear* section (for $0 \le a \le a_0$) and a *nonlinear* section (for $a_0 < a \le 1$) that connect smoothly at the transition point $a = a_0$. The linear slope s and the parameter d are determined by the condition that the two function segments must have identical values as well as identical first derivatives at $a = a_0$ to give a (C1) continuous function. The function in Eqn. (5.33) is thus fully specified by the two parameters a_0 and γ.

Figure 5.21 shows two examples of the modified gamma function $\bar{f}_{\gamma,a_0}()$ 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 functions $f_{\gamma}(a)$ for the same gamma values (dashed lines), whose slope at the origin is ∞ (Fig. 5.21 (a)) and zero (Fig. 5.21 (b)), respectively.

Gamma correction in common standards

The modified gamma function 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. 5.21, and γ is chosen to obtain a good overall match to the desired correction function. For example, the ITU-BT.709 specification [55] mentioned in Sec. 5.7.3 specifies the parameters



(a) (b)

Fig. 5.22
Gamma functions specified by the ITU-R BT.709 (a) and sRGB (b) standards. The continuous plot shows the modified gamma functions with the nominal gamma value γ and transition point a_0. The dashed lines are the equivalent ordinary gamma functions with effective gamma γ_{eff}.

Standard	Nominal Gamma Value γ	a_0	s	d	Effective Gamma Value γ_{eff}
ITU BT.709	$1/2.222 \approx 0.450$	0.01800	4.5068	0.09915	$1/1.956 \approx 0.511$
sRGB	$1/2.400 \approx 0.417$	0.00304	12.9231	0.05500	$1/2.200 \approx 0.455$

Table 5.1
Gamma correction parameters for the ITU and sRGB standards Eqns. (5.33) and (5.34).

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

with the corresponding slope and offset values $s = 4.50681$ and $d = 0.0991499$, respectively (Eqn. (5.34)). 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 [96] is specified on the same basis (with different parameters; see Sec. 12.3.3).

Figure 5.22 shows the actual correction functions for the ITU and sRGB standards, respectively, each in comparison with the equivalent ordinary gamma function. The ITU function (Fig. 5.22 (a)) with $\gamma = 0.45$ and $a_0 = 0.018$ corresponds to an ordinary gamma function with effective gamma value $\gamma_{\text{eff}} = 0.511$ (dashed line). The curves for sRGB (Fig. 5.22 (b)) differ only by the parameters γ and a_0, as summarized in Table 5.1.

Inverse gamma correction

To invert the modified gamma correction of the form $b = \bar{f}_{\gamma,a_0}(a)$ (Eqn. (5.33)), we need the inverse of the function $\bar{f}_{\gamma,a_0}()$, which is again defined in two parts,

$$\bar{f}^{-1}_{(\gamma,a_0)}(b) = \begin{cases} b/s & \text{for } 0 \le b \le s \cdot a_0 \\ \left(\dfrac{b+d}{1+d}\right)^{1/\gamma} & \text{for } s \cdot a_0 < b \le 1, \end{cases} \tag{5.35}$$

where s and d are the values defined in Eqn. (5.34), and thus

$$a = \bar{f}_{\gamma,a_0}^{-1}\big[\bar{f}_{\gamma,a_0}(a)\big] \qquad \text{for } a \in [0,1]. \tag{5.36}$$

Notice that the value γ is also the same for both functions in Eqn. (5.36). The inverse gamma function is required in particular for transforming between different color spaces if nonlinear (i. e., gamma-corrected) component values are involved (see also Sec. 12.3.2).

5.8 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. 5.4). 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 following subsections.

5.8.1 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,

$$\mathbf{L} : [0, K-1] \xmapsto{f} [0, K-1]. \tag{5.37}$$

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

$$\mathbf{L}[a] \leftarrow f(a) \qquad \text{for } 0 \le a < K. \tag{5.38}$$

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

$$I'(u,v) \leftarrow \mathbf{L}[I(u,v)], \tag{5.39}$$

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

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

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

Table 5.2
ImageJ methods for arithmetic operations applicable to objects of type `ImageProcessor`.

for objects of type `ImageProcessor`, which takes a lookup table ***lut*** (**L**) as a one-dimensional `int` array of size K (see Prog. 5.4 on page 77 for an example). The advantage of this approach is obvious: for an 8-bit image, for example, the mapping function is computed 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).

5.8.2 Arithmetic Operations

ImageJ implements a set of common arithmetic operations as methods for the class `ImageProcessor`, which are summarized in Table 5.2. In the following example, the image is multiplied by a scalar constant (1.5) to increase its contrast:

```
ImageProcessor ip = ... //some image ip
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.

5.8.3 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

$$I'(u,v) \;\leftarrow\; I_1(u,v) + I_2(u,v) \tag{5.40}$$

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

Table 5.3
Transfer mode constants and corresponding arithmetic operations for `ImageProcessor`'s `copyBits()` method. The constants `ADD`, etc., listed in this table are defined by the `Blitter` interface.

`ADD`	$ip1 \leftarrow ip1 + ip2$
`AVERAGE`	$ip1 \leftarrow (ip1 + ip2) \,/\, 2$
`DIFFERENCE`	$ip1 \leftarrow \lvert ip1 - ip2 \rvert$
`DIVIDE`	$ip1 \leftarrow ip1 \,/\, ip2$
`MAX`	$ip1 \leftarrow \max(ip1, ip2)$
`MIN`	$ip1 \leftarrow \min(ip1, ip2)$
`MULTIPLY`	$ip1 \leftarrow ip1 \cdot ip2$
`SUBTRACT`	$ip1 \leftarrow ip1 - ip2$

$$I'(u,v) \;\leftarrow\; f\big(I_1(u,v), I_2(u,v), \ldots I_n(u,v)\big). \tag{5.41}$$

However, most arithmetic operations on multiple images required in practice can be implemented as sequences of successive binary operations on pairs of images.

5.8.4 Methods for Point Operations on Two Images

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

`void 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 code fragment in the following example demonstrates the addition of two images:

```
ImageProcessor ip1 = ... // target image I1
ImageProcessor ip2 = ... // source image I2
// I1(u,v) ← I1(u,v) + I2(u,v)
ip1.copyBits(ip2, 0, 0, Blitter.ADD);  ...
```

By 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 5.3). In addition, `Blitter` defines (bitwise) logical operations, such as `OR` and `AND` (see Appendix C.10.1). 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.

5.8.5 ImageJ Plugins for 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` to the plugin's `run()` method (see also Sec. 3.2.3).

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: Alpha blending

Alpha blending is a simple method for transparently overlaying 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

$$I'(u,v) \leftarrow \alpha \cdot I_{\text{BG}}(u,v) + (1-\alpha) \cdot I_{\text{FG}}(u,v) \tag{5.42}$$

with $0 \le \alpha \le 1$. 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. (5.42)).

Figure 5.23 shows the results of alpha blending for different α values. The Java code for the corresponding implementation (as an ImageJ plugin) is listed in Progs. 5.5 and 5.6. 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 (see also Appendix C.18.2). A similar example that produces a *stack* of images by stepwise alpha blending can be found in Appendix C.15.3.

5.9 Exercises

Exercise 5.1. Implement the auto-contrast operation as defined in Eqns. (5.8)– (5.10) as an ImageJ plugin for an 8-bit grayscale image. Set the quantile s of pixels to be saturated at both ends of the intensity range (0 and 255) to $s_{\text{low}} = s_{\text{high}} = 1\%$.

Exercise 5.2. Modify the histogram equalization plugin in Prog. 5.2 to use a lookup table (Sec. 5.8.1) for computing the point operation.

Exercise 5.3. Implement the histogram equalization as defined in Eqn. (5.11), but use the *modified* cumulative histogram defined in Eqn. (5.12), 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. 5.10.

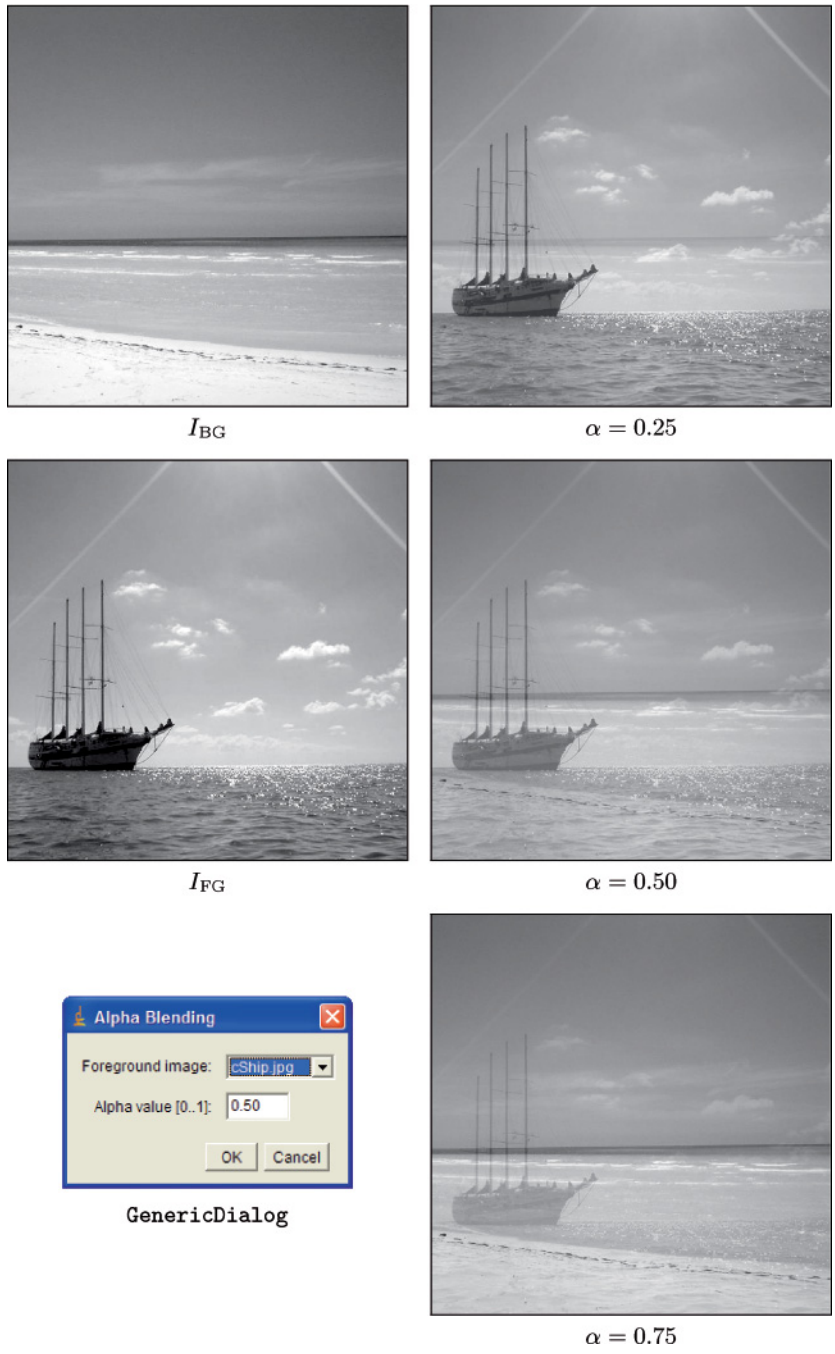


Fig. 5.23 Alpha blending example. Background image (I_{BG}) and foreground image (I_{FG}), `GenericDialog` window (see the implementation in Progs. 5.5 and 5.6), and blended images for transparency values $\alpha = 0.25, 0.50$, and 0.75.

Program 5.5
Alpha blending plugin (part 1). 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 23). The foreground image (`fgIP`, selected in part 2) is first duplicated (line 21) and then multiplied with $(1-\alpha)$ (line 22). Thus the original foreground image is not modified. The final result is obtained by adding the two weighted images (line 24).

```
1  import ij.IJ;
2  import ij.ImagePlus;
3  import ij.WindowManager;
4  import ij.gui.GenericDialog;
5  import ij.plugin.filter.PlugInFilter;
6  import ij.process.*;
7
8  public class Alpha_Blending implements PlugInFilter {
9
10   static double alpha = 0.5; // transparency of foreground image
11   ImagePlus fgIm = null;     // foreground image
12
13   public int setup(String arg, ImagePlus imp) {
14     return DOES_8G;
15   }
16
17   public void run(ImageProcessor bgIp) { // background image
18     if(runDialog()) {
19       ImageProcessor fgIp
20         = fgIm.getProcessor().convertToByte(false);
21       fgIp = fgIp.duplicate();
22       fgIp.multiply(1-alpha);
23       bgIp.multiply(alpha);
24       bgIp.copyBits(fgIp, 0, 0, Blitter.ADD);
25     }
26   }
27
28   // continued ...
```

Exercise 5.4. Show formally that (a) a linear histogram equalization (Eqn. (5.11)) 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 5.5. Show that the linear histogram equalization (Sec. 5.5) is only a special case of histogram specification (Sec. 5.6).

Exercise 5.6. Implement (in Java) a histogram specification using a piecewise linear reference distribution function, as described in Sec. 5.6.3. Define a new object class with all necessary instance variables to represent the distribution function and implement the required functions $\mathsf{P}_L(i)$ (Eqn. (5.22)) and $\mathsf{P}_L^{-1}(b)$ (Eqn. (5.23)) as methods of this class.

Exercise 5.7. Using a histogram specification for adjusting *multiple* images (Sec. 5.6.4), 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).

Program 5.6
Alpha blending plugin (part 2). To select the foreground image, a list of currently open images and image titles is obtained (lines 34–42). Then a dialog object (`GenericDialog`) is created and opened for specifying the foreground image (`fgIm`) and the α value (`alpha`). `fgIm` and `alpha` are variables in the class `AlphaBlend_` (declared in part 1, Prog. 5.5). The `runDialog()` method returns `true` if successful and `false` if no images are open or the dialog was canceled by the user.

```
30  // class Alpha_Blending (continued)
31
32  boolean runDialog() {
33    // get list of open images
34    int[] windowList = WindowManager.getIDList();
35    if (windowList == null){
36      IJ.noImage();
37      return false;
38    }
39
40    // get all image titles
41    String[] windowTitles = new String[windowList.length];
42    for (int i = 0; i < windowList.length; i++) {
43      ImagePlus im = WindowManager.getImage(windowList[i]);
44      if (im == null)
45        windowTitles[i] = "untitled";
46      else
47        windowTitles[i] = im.getShortTitle();
48    }
49
50    // create dialog and show
51    GenericDialog gd = new GenericDialog("Alpha Blending");
52    gd.addChoice("Foreground image:",
53          windowTitles, windowTitles[0]);
54    gd.addNumericField("Alpha value [0..1]:", alpha, 2);
55    gd.showDialog();
56    if (gd.wasCanceled())
57      return false;
58    else {
59      int fgIdx = gd.getNextChoiceIndex();
60      fgIm = WindowManager.getImage(windowList[fgIdx]);
61      alpha = gd.getNextNumber();
62      return true;
63    }
64  }
65
66 } // end of class Alpha_Blending
```

Exercise 5.8. Implement the modified gamma correction (Eqn. (5.33)) as an ImageJ plugin with variable values for γ and a_0 using a lookup table as shown in Prog. 5.4.

Exercise 5.9. Show that the modified gamma function $\bar{f}_{(\gamma,a_0)}(a)$ with the parameters defined in Eqns. (5.33) and (5.34) is C1-continuous (i. e., both the function itself and its first derivative are continuous).

6

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. 6.1). 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 (i. e., the geometry of the image does not change).

6.1 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.

Fig. 6.1
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.

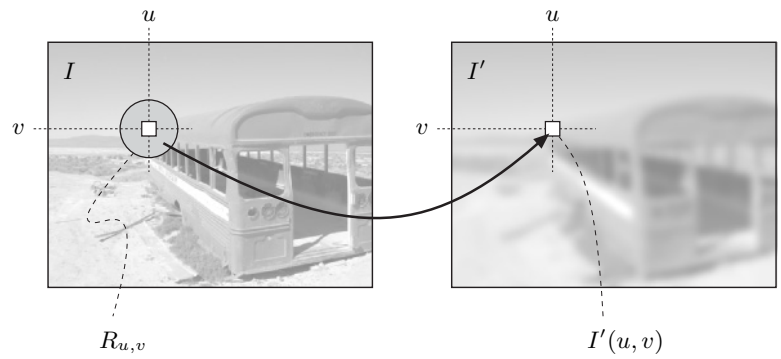


Fig. 6.2 Principal filter operation. Each new pixel value $I'(u,v)$ is computed as a function of the pixels in a corresponding region of source pixels $R_{u,v}$ in the original image I.

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 $p_1, p_2, \ldots p_8$ to compute the arithmetic mean of these nine values,

$$I'(u,v) \leftarrow \frac{p_0 + p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + p_8}{9} \tag{6.1}$$

or, expressed in relative image coordinates,

$$\begin{aligned} I'(u,v) \leftarrow \tfrac{1}{9} \cdot [\, & I(u-1,v-1) + I(u,v-1) + I(u+1,v-1) + \\ & I(u-1,v) + I(u,v) + I(u+1,v) + \\ & I(u-1,v+1) + I(u,v+1) + I(u+1,v+1) \,]\,. \end{aligned} \tag{6.2}$$

Written more compactly, this is equivalent to

$$I'(u,v) \leftarrow \frac{1}{9} \cdot \sum_{j=-1}^{1} \sum_{i=-1}^{1} I(u+i, v+j). \tag{6.3}$$

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. 6.2.

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. (6.2) 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* smoothing 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 (e. g., in a ring-shaped filter region).

It is probably confusing to have so many options—a more systematic method is needed for specifying and applying images in a targeted manner. The traditional and proven classification into *linear* and *nonlinear* filters is based on the mathematical properties of the filter function; i. e., 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.

6.2 Linear Filters

Linear filters are denoted that way because they combine the pixel values in the support region in a linear fashion; i. e., as a weighted summation. The local averaging discussed in the beginning Eqn. (6.3) is a special example, where all nine pixels in the 3×3 support region are added with the same 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.

6.2.1 The Filter Matrix

For any linear filter, the size and shape of the support region, as well as the individual pixel weights, are specified by the "filter matrix" or "filter mask" $H(i,j)$. The size of the matrix 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. (6.3), the filter matrix is

$$H(i,j) = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{6.4}$$

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

In principle, the filter matrix $H(i,j)$ is, just like the image itself, a discrete, two-dimensional, 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. 6.3). The filter function is of infinite extent and considered zero outside the region defined by the matrix H.

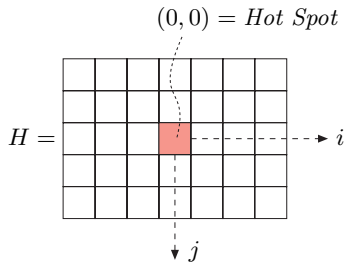


Fig. 6.3
Filter matrix and coordinate system.

6.2.2 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, as illustrated in Fig. 6.4. The following steps are performed at each image position (u, v):

1. The filter matrix 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.
3. Finally, the resulting sum is stored at the current position in the new image $I'(u, v)$.

Described formally, all pixels in the new image $I'(u, v)$ are computed by the expression

$$I'(u, v) \leftarrow \sum_{(i,j)\in R_H} I(u + i, v + j) \cdot H(i, j), \tag{6.5}$$

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

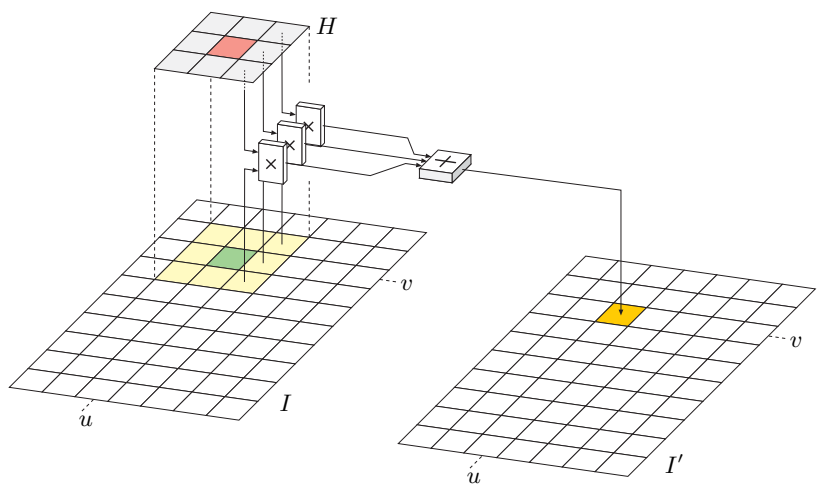


Fig. 6.4
Linear filter. The filter matrix H is placed with its origin at position (u, v) on the image I. Each filter coefficient $H(i, j)$ is multiplied with the corresponding image pixel $I(u+i, v+j)$, the results are added, and the final sum is inserted as the new pixel value $I'(u, v)$.

$$I'(u,v) \leftarrow \sum_{i=-1}^{i=1} \sum_{j=-1}^{j=1} I(u+i, v+j) \cdot H(i,j), \tag{6.6}$$

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. 6.5.2.

6.2.3 Computing the Filter Operation

Now that we understand the principal operation of a filter (Fig. 6.4) 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. 5.1 and 5.2), 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 wanted). Thus the complete filter operation can be implemented in two different ways (Fig. 6.5):

A. The result of the filter computation is initially stored in a new image whose content is eventually copied back to the original image.
B. The original image is first copied to an intermediate image that serves as the source for the filter computation. The results are directly stored in the original image.

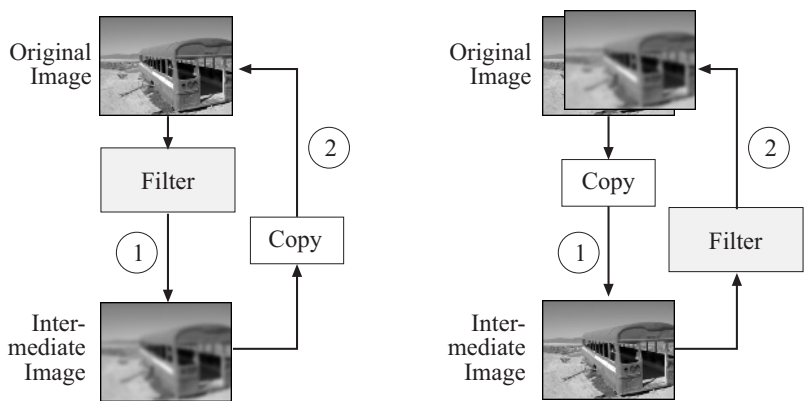


Fig. 6.5
Practical implementation of filter operations.
Version A: The result of the filter is first stored in an intermediate image and subsequently copied back to the original image.
Version B: The original image is first copied to an intermediate image that serves as the source for the filter operation. 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 use version B.

6.2.4 Filter Plugin Examples

Simple 3 × 3 averaging filter ("box" filter)

Program 6.1 shows the ImageJ code for a simple 3 × 3 smoothing filter based on local averaging Eqn. (6.4), which is often called a "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. 5.1.2) 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 two-dimensional 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 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:

$$H(i,j) = \begin{bmatrix} 0.075 & 0.125 & 0.075 \\ 0.125 & \mathbf{0.2} & 0.125 \\ 0.075 & 0.125 & 0.075 \end{bmatrix}. \tag{6.7}$$

Notice that all coefficients in H are positive and sum to 1 (i. e., the matrix is normalized) such that all results are within the given range of pixel values and no clamping is necessary again, and the program structure in Prog. 6.2 is virtually identical to the previous example. The filter matrix (`filter`) is represented by a two-dimensional array[1] 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 (Prog. 6.2, line 20).

[1] See the additional comments in Appendix B.2.4 regarding two-dimensional arrays in Java.

Program 6.1
3×3 averaging "box" filter (ImageJ plugin). 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 22, the result for the current image position (`u`, `v`) is rounded and subsequently stored 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.*;
2  import ij.plugin.filter.PlugInFilter;
3  import ij.process.*;
4
5  public class Filter_Average3x3 implements PlugInFilter {
6      ...
7      public void run(ImageProcessor orig) {
8          int w = orig.getWidth();
9          int h = orig.getHeight();
10         ImageProcessor copy = orig.duplicate();
11
12         for (int v = 1; v <= h-2; v++) {
13             for (int u = 1; u <= w-2; u++) {
14                 //compute filter result for position (u,v)
15                 int sum = 0;
16                 for (int j = -1; j <= 1; j++) {
17                     for (int i = -1; i <= 1; i++) {
18                         int p = copy.getPixel(u+i, v+j);
19                         sum = sum + p;
20                     }
21                 }
22                 int q = (int) Math.round(sum/9.0);
23                 orig.putPixel(u, v, q);
24             }
25         }
26     }
27 } // end of class Filter_Average3x3
```

6.2.5 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,

$$H(i,j) = s \cdot H'(i,j), \tag{6.8}$$

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 as the reciprocal of the sum of the coefficients,

$$s = \frac{1}{\sum_{i,j} H'(i,j)}, \tag{6.9}$$

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. (6.7) could be defined equivalently as

$$H(i,j) = \begin{bmatrix} 0.075 & 0.125 & 0.075 \\ 0.125 & \underline{0.200} & 0.125 \\ 0.075 & 0.125 & 0.075 \end{bmatrix} = \frac{1}{40} \begin{bmatrix} 3 & 5 & 3 \\ 5 & \underline{8} & 5 \\ 3 & 5 & 3 \end{bmatrix} \tag{6.10}$$

Program 6.2 3×3 smoothing filter (ImageJ plugin, `run()` method only). The filter matrix is defined as a two-dimensional array of type `double` (line 5). 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 20. The results are rounded (line 24) and stored in the original image (line 25).

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

with the common scale factor $s = \frac{1}{40} = 0.025$. A similar scaling is used in Prog. 6.3.

In Adobe Photoshop, linear filters can be specified with the "Custom Filter" tool (Fig. 6.6) 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; e. g., 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

$$I'(u,v) \leftarrow \mathsf{Offset} + \frac{1}{\mathsf{Scale}} \sum_{j=-2}^{j=2} \sum_{i=-2}^{i=2} I(u+i, v+j) \cdot H(i,j). \qquad (6.11)$$

6.2.6 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 is

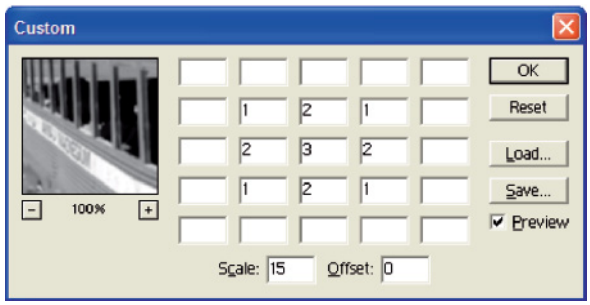


Fig. 6.6
Adobe Photoshop's "Custom Filter" implements linear filters up to a size of 5×5. The filter's coordinate origin ("hot spot") is assumed to be at the center (value set to 3 in this example), and empty cells correspond to zero coefficients. In addition to the (integer) coefficients, common Scale and Offset values can be specified.

centered with an odd number of $(2K + 1)$ rows and $(2L + 1)$ columns $(K, L \geq 0)$. If the image is of size $M \times N$,

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

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

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

as illustrated in Fig. 6.7. Program 6.3 (which is adapted from Prog. 6.2) shows a 7×5 smoothing filter as an example for implementing linear filters of arbitrary size. This example uses integer-valued filter coefficients in combination with a common scale factor `s`, as described above. 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 32–33) as a preventive measure.

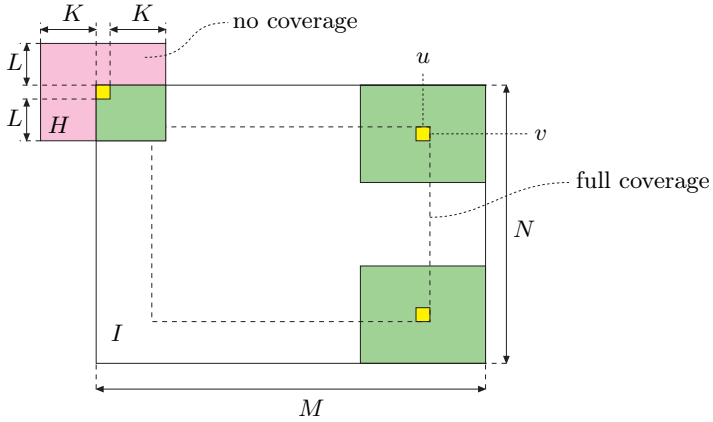


Fig. 6.7
Border geometry. The filter can be applied only at locations (u, v) where the filter matrix H of size $(2K+1) \times (2L+1)$ is fully contained in the image.

6.2.7 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

Program 6.3
ImageJ plugin (`run()` method only) for filters of arbitrary size. 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 31. The border pixels are not modified.

```
1   public void run(ImageProcessor orig) {
2       int M = orig.getWidth();
3       int N = orig.getHeight();
4
5       // filter matrix of size (2K + 1) × (2L + 1)
6       int[][] filter = {
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      int K = filter[0].length/2;
16      int L = filter.length/2;
17
18      ImageProcessor copy = orig.duplicate();
19
20      for (int v = L; v <= N-L-1; v++) {
21          for (int u = K; u <= M-K-1; u++) {
22              // compute filter result for position (u, v)
23              int sum = 0;
24              for (int j = -L; j <= L; j++) {
25                  for (int i = -K; i <= K; i++) {
26                      int p = copy.getPixel(u+i, v+j);
27                      int c = filter[j+L][i+K];
28                      sum = sum + c * p;
29                  }
30              }
31              int q = (int) Math.round(s * sum);
32              if (q < 0)  q = 0;
33              if (q > 255) q = 255;
34              orig.putPixel(u, v, q);
35          }
36      }
37  }
```

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 of key importance in practice: smoothing filters and difference filters (Fig. 6.8).

Smoothing filters

Every filter we have discussed so far caused 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.

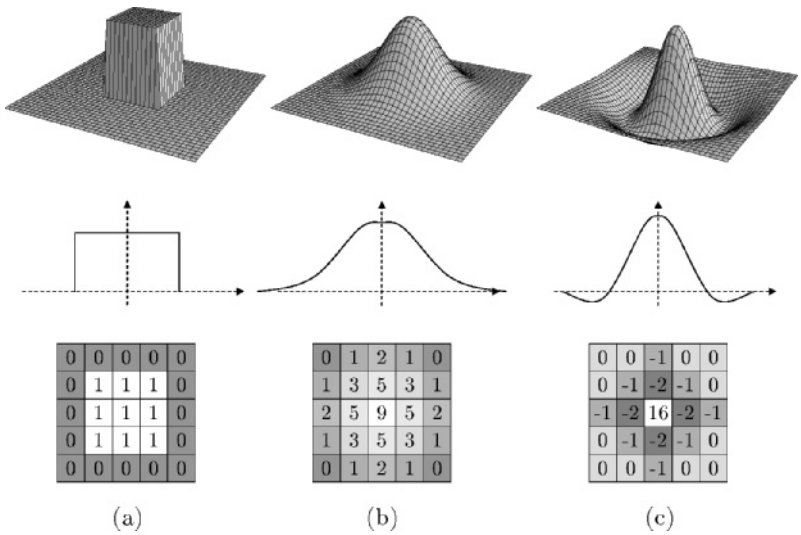


Fig. 6.8
Typical examples of linear filters, illustrated as 3D plots (top), profiles (center), and approximations by discrete filter matrices (bottom). The "box" filter (a) and the Gauss filter (b) are both *smoothing filters* with all-positive coefficients. The "Laplace" or "Mexican hat" filter (c) 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.

Box filter

This simplest of all smoothing filters, whose 3D shape resembles a box (Fig. 6.8 (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 (i. e., effectively attenuating all signal components above a given cutoff frequency).[2] 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 square-shaped box filter.

Gaussian filter

The filter matrix (Fig. 6.8 (b)) of this smoothing filter corresponds to a discrete, two-dimensional Gaussian function,

$$G_\sigma(r) = e^{-\frac{r^2}{2\sigma^2}} \qquad \text{or} \qquad G_\sigma(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}}\,, \tag{6.12}$$

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

[2] More details on the image vs. frequency space and related concepts are covered in Chs. 13 and 14.

value 9 in the matrix shown in Fig. 6.8 (b)), and the remaining coefficients drop off smoothly with increasing distance from the center. The Gaussian filter is isotropic if the 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 two-dimensional Gaussian filter is separable into a pair of one-dimensional filters (see Sec. 6.3.3), which counterweights its otherwise slightly higher computational costs.

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,

$$\begin{aligned} I'(u,v) = &\sum_{(i,j)\in R_H^+} I(u+i, v+j) \cdot |H(i,j)| \\ &- \sum_{(i,j)\in R_H^-} I(u+i, v+j) \cdot |H(i,j)|\,, \end{aligned} \tag{6.13}$$

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. 6.8 (c) computes the difference between the center pixel (with weight 16) and the weighted sum of 12 surrounding pixels (with weights -1 and -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 enhancing edges (Sec. 7.2) and image sharpening (Sec. 7.6).

6.3 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 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 yet mentioned the term "convolution" in this context. We make up for this in the following subsection.

6.3.1 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, two-dimensional functions I and H, the convolution operation is defined as

$$I'(u,v) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(u-i, v-j) \cdot H(i,j), \tag{6.14}$$

or

$$I' = I * H \tag{6.15}$$

for short, where $*$ denotes the convolution operator. This almost looks the same as Eqn. (6.5), 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. (6.14) by replacing the summation variables i, j to

$$\begin{aligned} I'(u,v) &= \sum_{(i,j)\in R_H} I(u-i, v-j) \cdot H(i,j) \\ &= \sum_{(i,j)\in R_H} I(u+i, v+j) \cdot H(-i,-j) \\ &= \sum_{(i,j)\in R_H} I(u+i, v+j) \cdot H^*(i,j). \end{aligned} \tag{6.16}$$

The result is identical to the linear filter in Eqn. (6.5), with the filter function $H^*(i,j) = H(-i,-j)$ being the horizontally and vertically *reflected* (i. e., rotated by 180°) function H. To be precise, the operation in Eqn. (6.5) actually defines the linear *correlation*, which is merely a convolution with a reflected filter matrix.[4]

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. 6.9.

[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 one-dimensional case. Linear correlation is typically used for comparing images or subpatterns (see Ch. 17).

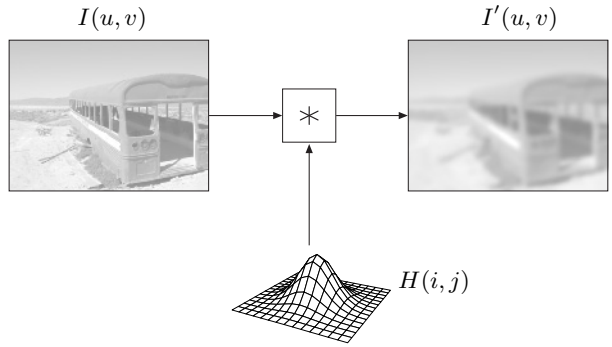


Fig. 6.9
Convolution as a "black box" operation. The original image I is subject to a linear convolution ($*$) with the convolution kernel H, producing the resulting I'.

6.3.2 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. 13), 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*; i. e.,

$$I * H = H * I. \tag{6.17}$$

Thus the result is the same if the image and filter kernel were 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 exchangeable and may assume either role.

Linearity

Linear filters are called that 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,

$$(s \cdot I) * H \;=\; I * (s \cdot H) \;=\; s \cdot (I * H)\,. \tag{6.18}$$

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:

$$(I_1 + I_2) * H \quad = \quad (I_1 * H) + (I_2 * H). \tag{6.19}$$

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,

$$(b + I) * H \quad \neq \quad b + (I * H), \tag{6.20}$$

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:

$$A * (B * C) = (A * B) * C. \tag{6.21}$$

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

6.3.3 Separability of Linear Filters

If a convolution kernel H can be expressed as the convolution of multiple kernels itself,

$$H = H_1 * H_2 * \ldots * H_n,$$

then (as a consequence of Eqn. (6.21)) the filter operation $I * H$ may be performed as a sequence of convolutions with the constituting kernels,

$$\begin{aligned} I * H &= I * (H_1 * H_2 * \ldots * H_n) \\ &= \big(\ldots((I * H_1) * H_2) * \ldots * H_n\big). \end{aligned} \tag{6.22}$$

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

x/y-separability

The possibility of separating a two-dimensional kernel H into a pair of one-dimensional 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 one-dimensional kernels H_x and H_y with

$$H_x = \begin{bmatrix} 1 & 1 & \mathbf{1} & 1 & 1 \end{bmatrix} \qquad \text{and} \qquad H_y = \begin{bmatrix} 1 \\ \mathbf{1} \\ 1 \end{bmatrix}, \tag{6.23}$$

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

$$I' \leftarrow (I * H_x) * H_y = I * \underbrace{(H_x * H_y)}_{H_{xy}}, \tag{6.24}$$

then according to Eqn. (6.22) this is equivalent to applying the composite filter

$$H_{xy} = H_x * H_y = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & \mathbf{1} & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}. \tag{6.25}$$

Thus this two-dimensional 5×3 box filter H_{xy} can be constructed from two one-dimensional 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 case above, the required amount of processing is $5 \cdot 3 = 15$ steps per image pixel for the 2D filter H_{xy} 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. 6.5.1).

Separable Gaussian filters

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

$$H_{x,y}(i, j) = (H_x \otimes H_y)\,(i, j) = H_x(i) \cdot H_y(j), \tag{6.26}$$

because in this case the resulting function also corresponds to the convolution product $H_{x,y} = H_x * H_y$. A prominent example is the widely employed two-dimensional Gaussian function $G_\sigma(x, y)$ Eqn. (6.12), which can be expressed as the product

$$G_\sigma(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}} = e^{-\frac{x^2}{2\sigma^2}} \cdot e^{-\frac{y^2}{2\sigma^2}} = g_\sigma(x) \cdot g_\sigma(y). \tag{6.27}$$

Thus a two-dimensional Gaussian filter $H^{G,\sigma}$ can be implemented by a pair of one-dimensional Gaussian filters $H_x^{G,\sigma}$, $H_y^{G,\sigma}$ as

$$I' \leftarrow I * H^{G,\sigma} = I * H_x^{G,\sigma} * H_y^{G,\sigma}.$$

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

```
1  float[] makeGaussKernel1d(double sigma) {
2
3    // create the kernel
4    int center = (int) (3.0*sigma);
5    float[] kernel = new float[2*center+1]; // odd size
6
7    // fill the kernel
8    double sigma2 = sigma * sigma;          // σ²
9    for (int i=0; i<kernel.length; i++) {
10     double r = center - i;
11     kernel[i] = (float) Math.exp(-0.5 * (r*r) / sigma2);
12   }
13
14   return kernel;
15 }
```

Program 6.4
Dynamic creation of one-dimensional 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.

run about 50 times faster than the full 2D filter. The Java method `makeGaussKernel1d()` in Prog. 6.4 shows how to dynamically create a one-dimensional 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. 7.1 in Sec. 7.6.2).

6.3.4 Impulse Response of a Filter

Linear convolution is a binary operation involving two functions and also has a "neutral element", which of course is a function, too. The *impulse* or *Dirac* function $\delta()$ is neutral under convolution; i. e.,

$$I * \delta = I. \tag{6.28}$$

In the discrete, two-dimensional case, the impulse function is defined as

$$\delta(u, v) = \begin{cases} 1 & \text{for } u = v = 0 \\ 0 & \text{otherwise.} \end{cases} \tag{6.29}$$

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. 6.10).

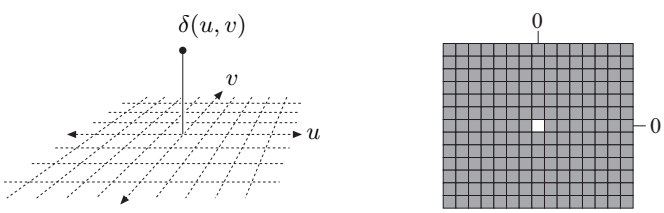


Fig. 6.10
Discrete, two-dimensional *impulse* or *Dirac* function $\delta(u, v)$.

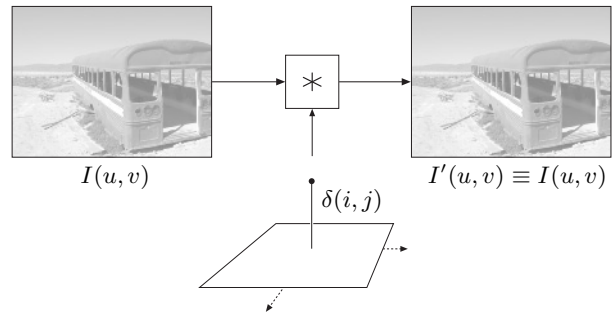


Fig. 6.11
Convolving the image I with the impulse δ returns the original unmodified image.

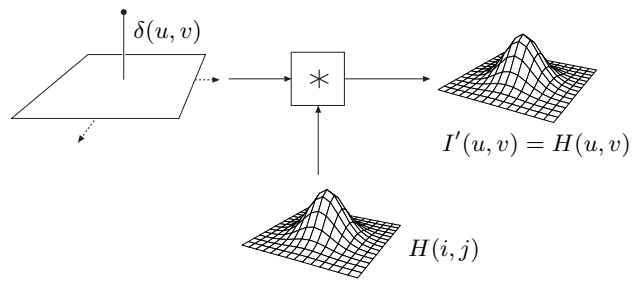


Fig. 6.12
The linear filter H with the impulse δ as the input yields the filter H as the result.

When the Dirac function is used as the filter kernel in a linear convolution as in Eqn. (6.28), the result obtained is the original unmodified image (Fig. 6.11). 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. (6.17) it is evident that

$$H * \delta = \delta * H = H \tag{6.30}$$

and thus the result of this filter operation is the filter H itself (Fig. 6.12)! 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 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.

6.4 Nonlinear Filters

Linear filters have an important disadvantage when used for smoothing or removing noise: all image structures, including points, edges, and lines, are also blurred, and the quality of the whole image is evenly

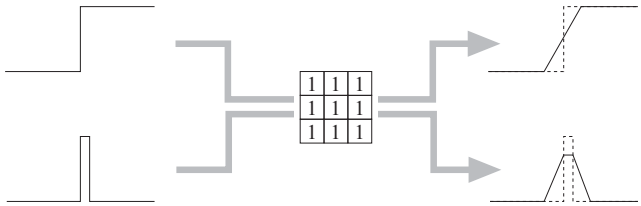


Fig. 6.13
Any image structure is blurred by a linear smoothing filter. Important image structures such as step edges (top) or thin lines (bottom) are widened, and the local contrast is reduced.

reduced (Fig. 6.13). This effect cannot be avoided, and thus the use of linear filters for these kinds of tasks (noise removal in particular) is limited. In the following, we investigate certain nonlinear filters to see if they can offer any better solution to this problem.

6.4.1 Minimum and Maximum Filters

Like all other filters, nonlinear filters compute the result at some 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

$$I'(u,v) \leftarrow \min\{I(u+i, v+j) \mid (i,j) \in R\}, \tag{6.31}$$

$$I'(u,v) \leftarrow \max\{I(u+i, v+j) \mid (i,j) \in R\}, \tag{6.32}$$

where R denotes the filter region (set of filter coordinates), usually a square of size 3×3 pixels. Figure 6.14 illustrates the effects of a one-dimensional minimum filter on various local signal structures.

Figure 6.15 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

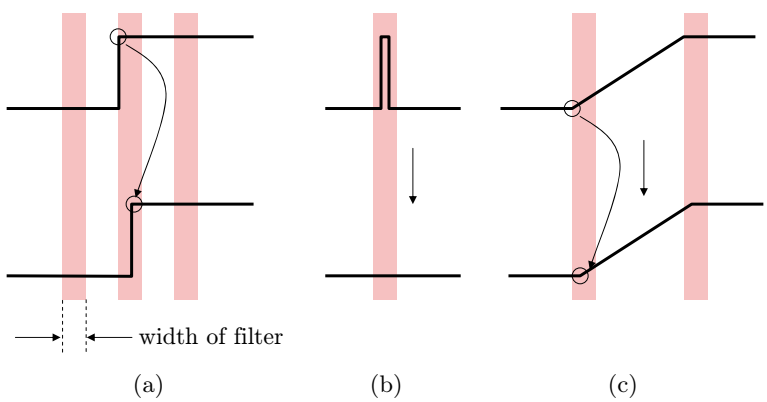


Fig. 6.14
Effects of a one-dimensional minimum filter on various local signal structures. Original signal (top) and result after filtering (bottom), where the colored bars indicate the extent of the filter. The step edge (a) and the linear ramp (c) are shifted to the right by half the filter width, and the narrow pulse (b) is completely removed.

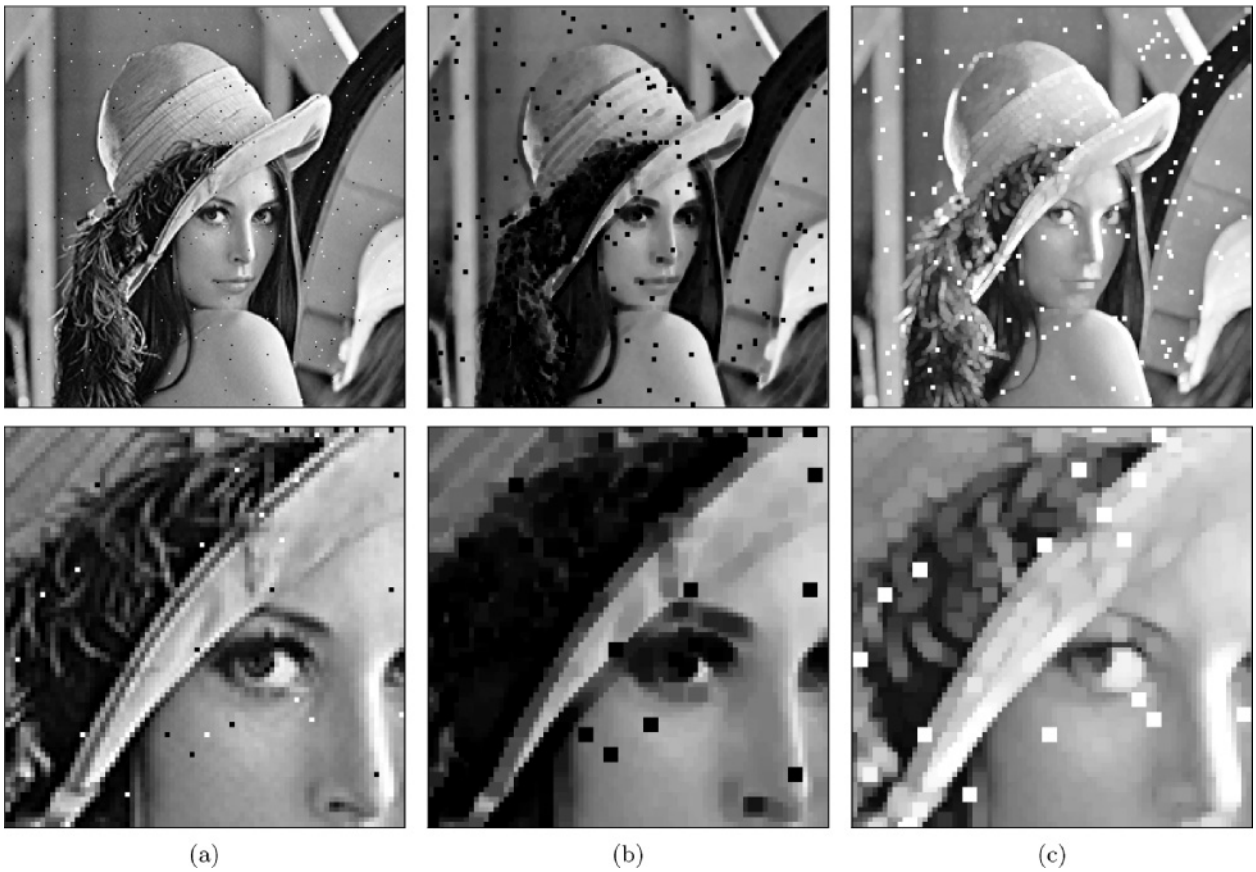


Fig. 6.15. Minimum and maximum filters applied to a grayscale image. The original image is corrupted with "salt-and-pepper" noise (a). The 3×3 pixel *minimum* filter eliminates the bright dots and widens all dark image structures (b). The *maximum* filter shows the exact opposite effects (c).

single white pixel within the 3×3 filter region is replaced 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.

6.4.2 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

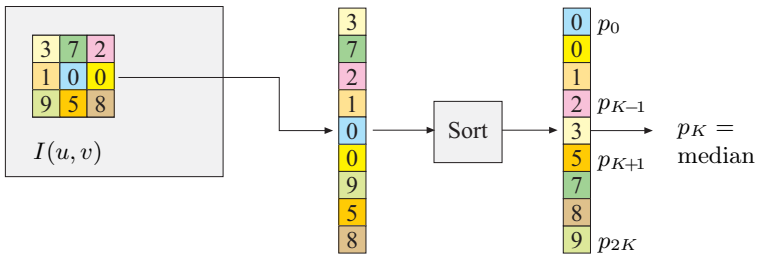


Fig. 6.16
Computation of a 3×3 pixel median filter. The nine pixel values extracted from the 3×3 image region are arranged as a vector that is sorted, and the resulting center value is taken as the median.

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.

The median filter replaces every image pixel by the *median* of the pixels in the corresponding filter region R,

$$I'(u,v) \leftarrow \text{median}\{I(u+i, v+j) \mid (i,j) \in R\}. \tag{6.33}$$

The median of $2K+1$ pixel values p_i is defined as

$$\text{median}(p_0, p_1, \ldots, p_K, \ldots, p_{2K}) \triangleq p_K; \tag{6.34}$$

i. e., the center value p_K if the sequence $(p_0, \ldots, p_{2K})$ is *sorted* ($p_i \leq p_{i+1}$). Figure 6.16 demonstrates the computation of the median filter or a filter region of size 3×3 pixels.

Equation (6.34) 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 image before. If, however, the number of elements is *even* ($2K$ for some $K > 0$), then the median of the sorted sequence $(p_0, \ldots, p_{2K-1})$ is defined as the arithmetic mean of the two middle values,

$$\text{median}(p_0, \ldots, p_{K-1}, p_K, \ldots, p_{2K-1}) \triangleq (p_{K-1} + p_K)/2. \tag{6.35}$$

Because of the interpolation above, new pixel values are generally introduced by the median filter if the region is of even size.

Figure 6.17 illustrates the effects of a 3×3 pixel median filter on selected two-dimensional image structures. In particular, very small structures (smaller than half the filter size) are eliminated, but all other structures remain largely unchanged. Finally, Fig. 6.18 compares the results of median filtering with a linear-smoothing filter. A sample Java implementation of the median filter, whose principal structure is identical to the 3×3 pixel linear filter in Prog. 6.2, is shown in Prog. 6.5.

6.4.3 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

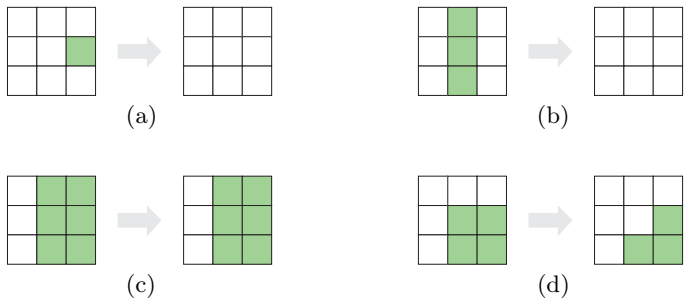


Fig. 6.17
Effects of a 3×3 pixel median filter on two-dimensional image structures. Isolated dots are eliminated (a), as are thin lines (b). The step edge remains unchanged (c), while a corner is rounded off (d).

Fig. 6.18. Linear smoothing filter vs. median filter. The original image is corrupted with "salt-and-pepper" noise (a). The linear 3×3 pixel box filter (b) 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 (c) 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.

Program 6.5
A 3×3 median filter (ImageJ plugin). An array `P` of type `int` is defined (line 15) to hold the 9 pixels for each filter position (u, v). This vector is sorted by using the Java utility method `Arrays.sort()` in line 28. The center element of the sorted vector (`P[K]`) is taken as the median value and stored in the original image (line 29).

```
1  import ij.*;
2  import ij.plugin.filter.PlugInFilter;
3  import ij.process.*;
4  import java.util.Arrays;
5
6  public class Filter_Median3x3 implements PlugInFilter {
7      final int K = 4; // filter size
8
9      public void run(ImageProcessor orig) {
10         int w = orig.getWidth();
11         int h = orig.getHeight();
12         ImageProcessor copy = orig.duplicate();
13
14         // vector to hold pixels from 3×3 neighborhood
15         int[] P = new int[2*K+1];
16
17         for (int v = 1; v <= h-2; v++) {
18             for (int u = 1; u <= w-2; u++) {
19                 // fill the pixel vector P for filter position u, v
20                 int k = 0;
21                 for (int j = -1; j <= 1; j++) {
22                     for (int i = -1; i <= 1; i++) {
23                         P[k] = copy.getPixel(u+i, v+j);
24                         k++;
25                     }
26                 }
27                 // sort pixel vector and take the center element
28                 Arrays.sort(P);
29                 orig.putPixel(u, v, P[K]);
30             }
31         }
32     }
33
34 } // end of class Filter_Median3x3
```

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 a regular median filter, each pixel in the filter region has the same influence, regardless of its distance from the center.

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* $W(i, j) \in \mathbb{N}$. To compute the result of the filter, each pixel value $I(u + i, v + j)$ involved is inserted $W(i, j)$ times into the extended pixel vector

$$Q = (p_0, \ldots, p_{L-1}) \quad \text{of length} \quad L = \sum_{(i,j) \in R} W(i, j).$$

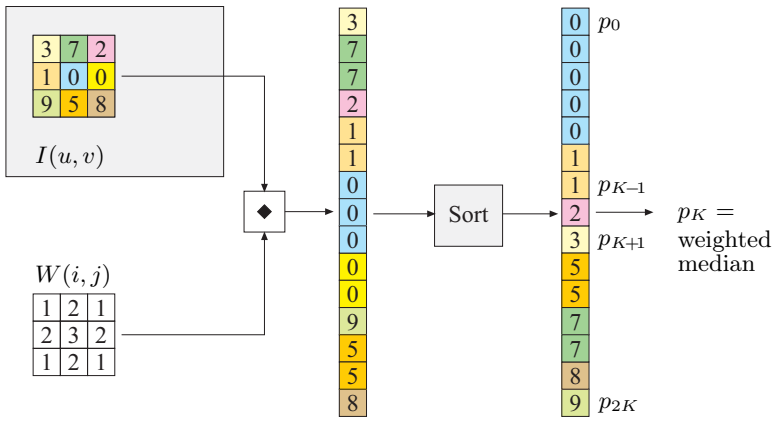


Fig. 6.19 Weighted median example. Each pixel value is inserted into the extended pixel vector multiple times, as specified by the weight matrix W. For example, the value 0 from the center pixel is inserted three times (since $W(0,0) = 3$) and the pixel value 7 twice. The pixel vector is sorted and the center value (2) is taken as the median.

This vector is then sorted, and the resulting center value is taken as the median, as in the standard median filter. Figure 6.19 illustrates the computation of the weighted median filter using the 3×3 weight matrix

$$W(i,j) = \begin{bmatrix} 1 & 2 & 1 \\ 2 & \mathbf{3} & 2 \\ 1 & 2 & 1 \end{bmatrix}, \tag{6.36}$$

which requires an extended pixel vector of length $L = 15$, equal to the sum of the weights in W.

Of course this method may also be used to implement conventional median filters of nonrectangular shape; for example, a *cross-shaped* median filter with the weight matrix

$$W^{+}(i,j) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{6.37}$$

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" and dictate the resulting value, thus inhibiting any filter effect.

6.4.4 Other Nonlinear Filters

Median and weighted median filters are two examples of nonlinear filters that are easy to describe and frequently used. Since "nonlinear" 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. 10. Other types of nonlinear filters, such as the corner detector described in Ch. 8, 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. (6.19) for linear convolution. Similarly, not much (if anything) can be stated in general about the effects of nonlinear filters in frequency space.

6.5 Implementing Filters

6.5.1 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

$$2K \cdot 2L \cdot M \cdot N = 4KLMN$$

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[5] of direct filtering is $\mathcal{O}(N^4)$. As described in Sec. 6.3.3, substantial savings are possible when large, two-dimensional filters can be decomposed (separated) into smaller, possibly one-dimensional filters.

The programming examples in this chapter are deliberately designed to be simple and easy to understand, and none of the solutions shown are 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 (particularly in Java).

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 access 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, as described in detail in Appendix C (p. 485).

6.5.2 Handling Image Borders

As mentioned briefly in Sec. 6.2.2, the image borders require special attention in most filter implementations. We have argued that theo-

[5] See Appendix A (p. 454) for a short description of the $\mathcal{O}()$ notation.

retically 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: Extend the image by "padding" additional pixels around it (Fig. 6.20) and filter the border regions, too, assuming that:

A. The pixels outside the image have a *constant value* (e. g., "black" or "gray"; Fig. 6.20 (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. 6.20 (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. 6.20 (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. 6.20 (d)). This may look weird at first, and also the results are generally not satisfactory. However, in discrete spectral analysis, the image is implicitly treated as a periodic function, too.[6] 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.

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.

6.5.3 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

[6] This comment refers to topics covered in Ch. 13.

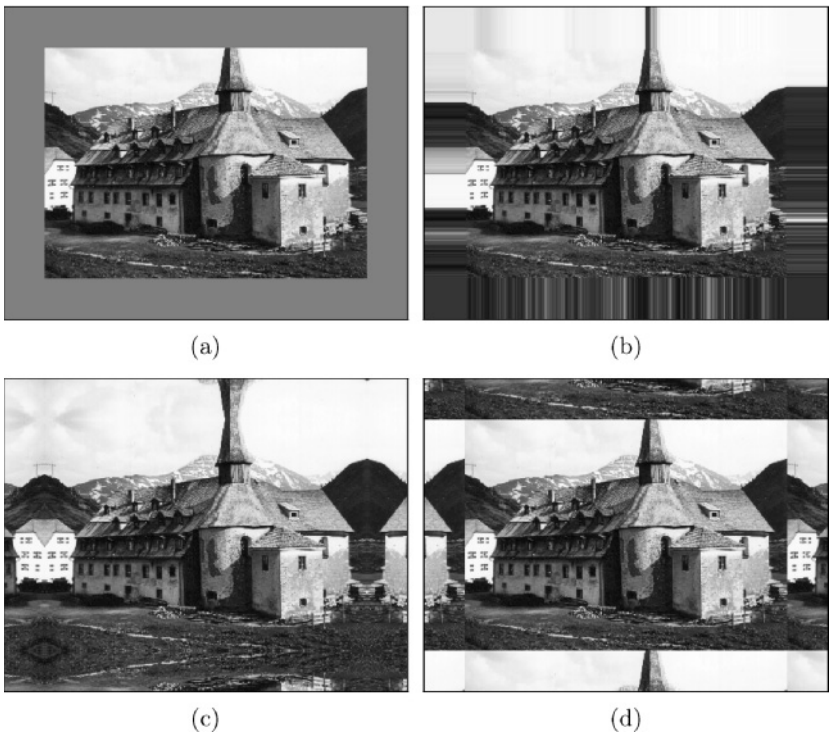


Fig. 6.20
Methods for extending the image to facilitate filtering along the borders. The assumption is that the (nonexisting) pixels outside the original image are either set to some constant value (a), take on the value of the closest border pixel (b), are mirrored at the image boundaries (c), or repeat periodically along the coordinate axes (d).

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. 6.3.4) before processing any real images.

6.6 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.

6.6.1 Linear Filters

Filters based on linear convolution are implemented by the ImageJ plugin class `ij.plugin.filter.Convolver`, which offers several methods in addition to the usual `run()` method. Its use is easily illustrated with the following example that convolves an 8-bit grayscale image with the filter kernel from Eqn. (6.7):

$$H(i,j) = \begin{bmatrix} 0.075 & 0.125 & 0.075 \\ 0.125 & \mathbf{0.2} & 0.125 \\ 0.075 & 0.125 & 0.075 \end{bmatrix}.$$

In the `run()` method below, we first define the filter matrix `H` as a *one*-dimensional `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:

```
1  import ij.plugin.filter.Convolver;
2  ...
3  public void run(ImageProcessor I) {
4    float[] H = {              // filter array is one-dimensional!
5        0.075f, 0.125f, 0.075f,
6        0.125f, 0.200f, 0.125f,
7        0.075f, 0.125f, 0.075f };
8    Convolver cv = new Convolver();
9    cv.setNormalize(false);   // do not use filter normalization
10   cv.convolve(I, H, 3, 3); // apply the filter H to I
11 }
```

The invocation of the method `convolve()` in line 10 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 one-dimensional array. The image `I` is destructively modified by this operation.

In this case, one could have also used the nonnormalized, integer-valued filter matrix given in Eqn. (6.10) because `convolve()` normalizes the given filter automatically (after `cv.setNormalize(true)`).

6.6.2 Gaussian Filters

The ImageJ class `ij.plugin.filter.GaussianBlur` implements a simple Gaussian filter with arbitrary radius (σ). The filter uses separable one-dimensional Gaussians as described in Sec. 6.3.3. Here is an example showing its application with the radius $\sigma = 2.5$:

```
1  import ij.plugin.filter.GaussianBlur;
2  ...
3  public void run(ImageProcessor ip) {
4    GaussianBlur gb = new GaussianBlur();
5    double radius = 2.5;
6    gb.blur(ip, radius);
7  }
```

An alternative implementation of separable Gaussian filters can be found in Prog. 7.1 (see p. 137), which uses the method `makeGaussKernel1d()` defined in Prog. 6.4 (page 103) for dynamically computing the required 1D filter kernels.

6.6.3 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:

```
1  import ij.plugin.filter.RankFilters;
2  ...
3  public void run(ImageProcessor ip) {
4    RankFilters rf = new RankFilters();
5    double radius = 3.5;
6    rf.rank(ip, radius, RankFilters.MIN);   // minimum filter
7    rf.rank(ip, radius, RankFilters.MAX);   // maximum filter
8    rf.rank(ip, radius, RankFilters.MEDIAN);   // median filter
9  }
```

6.7 Exercises

Exercise 6.1. Explain why the "custom filter" in Adobe Photoshop (Fig. 6.6) is not strictly a linear filter.

Exercise 6.2. Determine the possible maximum and minimum results (pixel values) for a linear filter with

$$H(i,j) = \begin{bmatrix} -1 & -2 & 0 \\ -2 & \mathbf{0} & 2 \\ 0 & 2 & 1 \end{bmatrix}$$

when applied to an 8-bit grayscale image (with pixel values in the range $[0, 255]$). Assume that no clamping of the results occurs.

Exercise 6.3. Modify the ImageJ plugin shown in Prog. 6.3 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. 6.5.2.

Exercise 6.4. Show that a standard box filter is not isotropic (i. e., does not smooth the image identically in all directions).

Exercise 6.5. Explain why the clamping of results to a limited range of pixel values may violate the linearity property (Sec. 6.3.2) of linear filters.

Exercise 6.6. 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 6.7. Implement a weighted median filter (Sec. 6.4.3) as an ImageJ plugin, specifying the weights as a constant, two-dimensional `int` array. Test the filter on suitable images and compare the results with those from a standard median filter. Explain why, for example, the weight matrix

$$W(i,j) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & \mathbf{5} & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

does *not* make sense.

Exercise 6.8. Verify the properties of the *impulse* function with respect to linear filters Eqn. (6.30). Create a black image with a white pixel at its center and use this image as the two-dimensional impulse. See if linear filters really deliver the filter matrix H as their impulse response.

Exercise 6.9. Describe the effect of a linear filter with the following filter matrix:

$$H(i,j) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \mathbf{0} & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

Exercise 6.10. Design a linear filter (matrix) that creates a horizontal blur over a length of 7 pixels, thus simulating the effect of camera movement during exposure.

Exercise 6.11. 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. 6.3.3).

Exercise 6.12. The "*Laplacian of Gaussian*" (LoG) filter (Fig. 6.8) is based on the sum of the second derivatives of the two-dimensional Gaussian. It is defined as

$$\mathrm{LoG}_\sigma(x,y) = -\Big(\frac{x^2+y^2-\sigma^2}{\sigma^4}\Big) \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}.$$

Implement the LoG filter as an ImageJ plugin of variable width (σ), analogous to Exercise 6.11. Find out if the LoG function is x/y-separable.

7

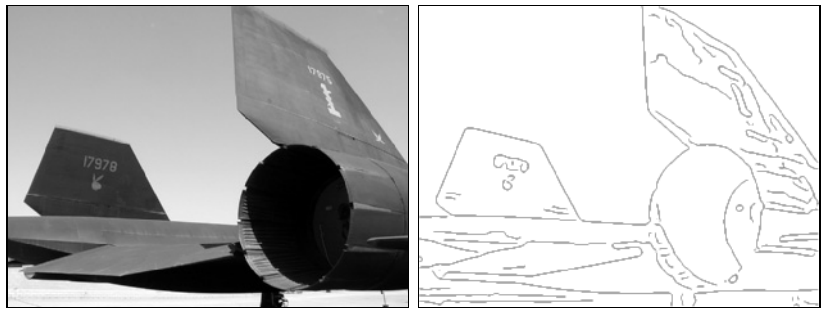
Edges and Contours

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.

7.1 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. 7.1 shows. But how do edges arise, and how can they be technically localized in an image?

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.



(a) (b)

Fig. 7.1
Edges play an important role in human vision. Original image (a) and edge image (b).

7.2 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. 7.2 (a)). In this case, the intensity profile along one image line would look like the one-dimensional function $f(x)$, as shown in Fig. 7.2 (b). Computing the first derivative of $f(x)$ from left to right as

$$f'(x) = \frac{df}{dx}(x) \tag{7.1}$$

results in a positive swing at those positions where the intensity rises and a negative swing where the value of the function drops (Fig. 7.2 (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 7.3 gives the basic idea, again for the one-dimensional 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 way 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 $f(u+1)$,

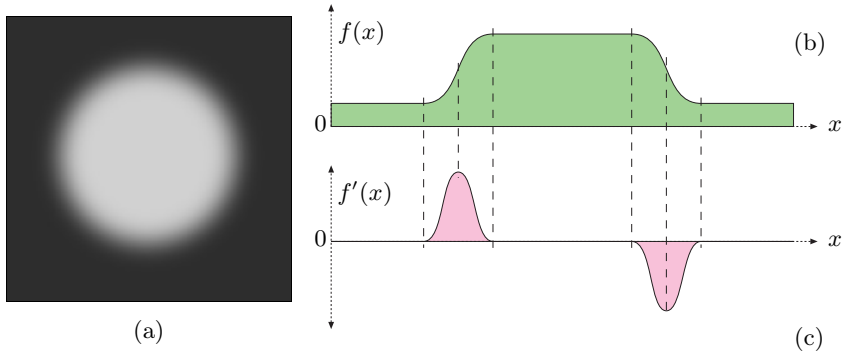


Fig. 7.2
Sample image and first derivative in one dimension: original image (a), horizontal intensity profile $f(x)$ along the center image line (b), and first derivative $f'(x)$ (c).

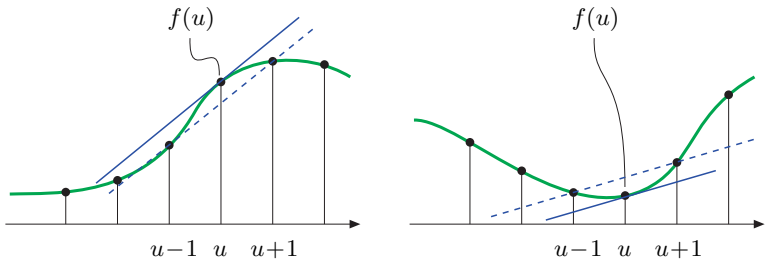


Fig. 7.3
Estimating the first derivative of a discrete function. The slope of the straight (dashed) line between the neighboring function values $f(u-1)$ and $f(u+1)$ is taken as the estimate for the slope of the tangent (i. e., the first derivative) at $f(u)$.

$$\frac{df}{du}(u) \approx \frac{f(u+1) - f(u-1)}{2} = 0.5 \cdot \big(f(u+1) - f(u-1)\big). \tag{7.2}$$

The same estimation can be applied of course in the vertical direction; i. e., along the image columns.

7.2.1 Partial Derivatives and the Gradient

A derivative of a multidimensional function taken along one of its coordinate axes is called a *partial derivative*; for example,

$$\frac{\partial I}{\partial u}(u,v) \qquad \text{and} \qquad \frac{\partial I}{\partial v}(u,v) \tag{7.3}$$

are the partial derivatives of the image function $I(u,v)$ along the u and v axes, respectively.[1] The function

$$\nabla I(u,v) = \begin{bmatrix} \frac{\partial I}{\partial u}(u,v) \\ \frac{\partial I}{\partial v}(u,v) \end{bmatrix} \tag{7.4}$$

is called the *gradient vector* (or "gradient" for short) of the function I at position (u,v). The *magnitude* of the gradient,

$$|\nabla I|(u,v) = \sqrt{\Big(\frac{\partial I}{\partial u}(u,v)\Big)^2 + \Big(\frac{\partial I}{\partial v}(u,v)\Big)^2}, \tag{7.5}$$

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.

7.2.2 Derivative Filters

The components of the gradient function (Eqn. (7.4)) are simply the first derivatives of the image lines (Eqn. (7.1)) and columns along the horizontal and vertical axes, respectively. The approximation of the first

[1] ∂ denotes the *partial derivative* or "del" operator.

horizontal derivatives (Eqn. (7.2)) can be easily implemented by a linear filter (see Sec. 6.2) with the coefficient matrix

$$H_x^D = \begin{bmatrix} -0.5 & \mathbf{0} & 0.5 \end{bmatrix} = 0.5 \cdot \begin{bmatrix} -1 & \mathbf{0} & 1 \end{bmatrix}, \tag{7.6}$$

where the coefficients -0.5 and $+0.5$ correspond 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. Similarly, the vertical component of the gradient can be computed with the linear filter

$$H_y^D = \begin{bmatrix} -0.5 \\ \mathbf{0} \\ 0.5 \end{bmatrix} = 0.5 \cdot \begin{bmatrix} -1 \\ \mathbf{0} \\ 1 \end{bmatrix}. \tag{7.7}$$

Figure 7.4 shows the results of applying the gradient filters in Eqn. (7.6) and Eqn. (7.7) 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 gray in Fig. 7.4 (b, c)).

7.3 Edge Operators

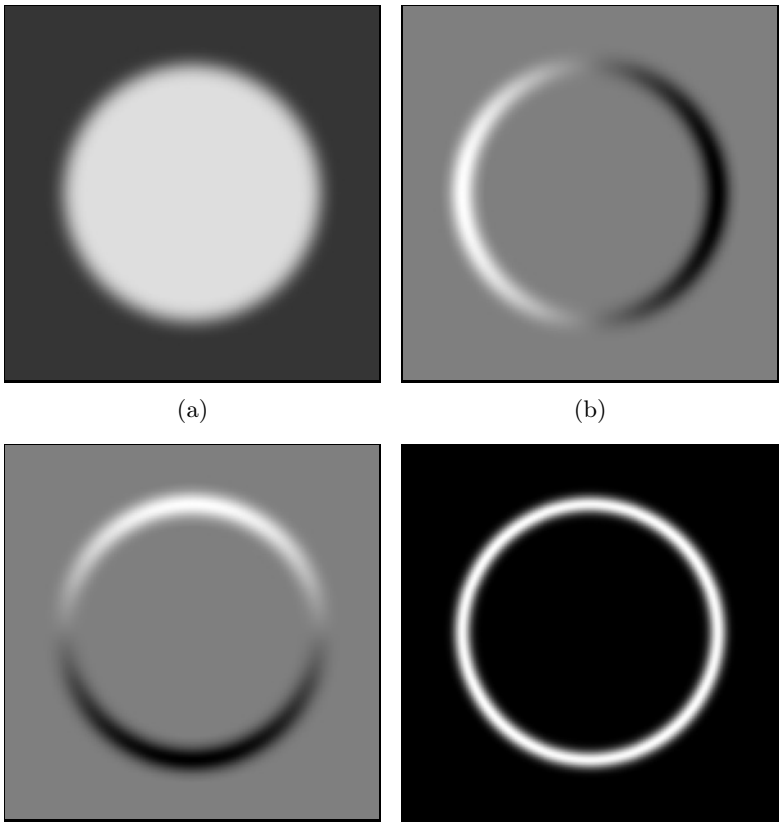
Approximating the local gradient of the image function is the basis of many classical edge-detection operators. Practically, they only differ in the type of filter used for estimating the gradient components and the way these components are combined. In many situations, one is not only interested in 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 historical perspective as well.

7.3.1 Prewitt and Sobel Operators

The edge operators by Prewitt and Sobel [26] are two classic methods that differ only marginally in the filters they use.

Gradient filters

The Prewitt and Sobel 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. (7.6) and (7.7)). The Prewitt operator uses the filters



(a) (b)

(c) (d)

Fig. 7.4
Partial derivatives of a two-dimensional 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.

$$H_x^P = \begin{bmatrix} -1 & 0 & 1 \\ -1 & \mathbf{0} & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad H_y^P = \begin{bmatrix} -1 & -1 & -1 \\ 0 & \mathbf{0} & 0 \\ 1 & 1 & 1 \end{bmatrix}, \tag{7.8}$$

which compute the average gradient components across three neighboring lines or columns, respectively. When the filters are written in separated form,

$$H_x^P = \begin{bmatrix} 1 \\ \mathbf{1} \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & \mathbf{0} & 1 \end{bmatrix} \quad \text{and} \quad H_y^P = \begin{bmatrix} 1 & \mathbf{1} & 1 \end{bmatrix} * \begin{bmatrix} -1 \\ \mathbf{0} \\ 1 \end{bmatrix}, \tag{7.9}$$

it becomes obvious that H_x^P performs a simple (box) smoothing over three lines before computing the x gradient (Eqn. (7.6)), and analogously H_y^P smooths over three columns before computing the y gradient (Eqn. (7.7)).[2] Because of the commutativity of linear convolution, this could equally be described the other way around, with smoothing to be applied *after* the computation of the gradients.

[2] In Eqn. (7.9), $*$ is the linear convolution operator (see Sec. 6.3.1).

The filters for the Sobel operator are almost identical; however, the smoothing part assigns higher weight to the current center line and column, respectively:

$$H_x^S = \begin{bmatrix} -1 & 0 & 1 \\ -2 & \mathbf{0} & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad H_y^S = \begin{bmatrix} -1 & -2 & -1 \\ 0 & \mathbf{0} & 0 \\ 1 & 2 & 1 \end{bmatrix}. \tag{7.10}$$

The estimates for the local gradient components are obtained from the filter results by appropriate scaling:

$$\nabla I(u,v) \approx \frac{1}{6} \cdot \begin{bmatrix} \big(I * H_x^P\big)(u,v) \\ \big(I * H_y^P\big)(u,v) \end{bmatrix} \tag{7.11}$$

for the *Prewitt* operator and

$$\nabla I(u,v) \approx \frac{1}{8} \cdot \begin{bmatrix} \big(I * H_x^S\big)(u,v) \\ \big(I * H_y^S\big)(u,v) \end{bmatrix} \tag{7.12}$$

for the *Sobel* operator.

Edge strength and orientation

In the following, we denote the scaled filter results (obtained with either the Prewitt or Sobel operator) as

$$D_x(u,v) = H_x * I \quad \text{and} \quad D_y(u,v) = H_y * I.$$

In both cases, the local edge strength $E(u,v)$ is defined as the gradient magnitude

$$E(u,v) = \sqrt{\big(D_x(u,v)\big)^2 + \big(D_y(u,v)\big)^2}, \tag{7.13}$$

and the local edge orientation angle $\Phi(u,v)$ is[3]

$$\Phi(u,v) = \tan^{-1}\Big(\frac{D_y(u,v)}{D_x(u,v)}\Big) = \mathrm{ArcTan}\big(D_x(u,v), D_y(u,v)\big). \tag{7.14}$$

The whole process of extracting the edge magnitude and orientation is summarized in Fig. 7.5. 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.

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 [59, p. 353] to minimize the orientation errors:

[3] See the hints in Appendix B.1.6 for computing the inverse tangent $\tan^{-1}(y/x)$ with the $\mathrm{ArcTan}(x,y)$ function.

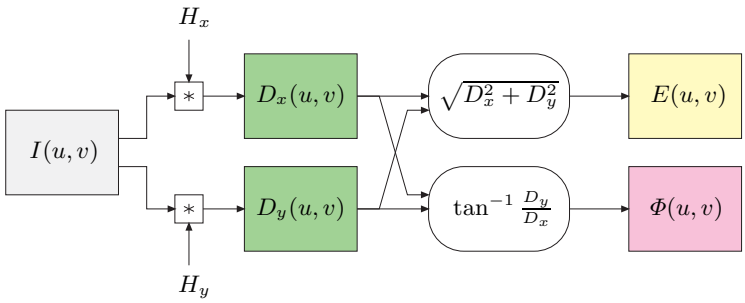


Fig. 7.5
Typical process of gradient-based edge extraction. The two linear gradient filters H_x and H_y produce two gradient images, D_x and D_y, respectively. They are used to compute the edge strength E and orientation Φ for each image position (u, v).

$$H_x^{S'} = \frac{1}{32}\begin{bmatrix} -3 & 0 & 3 \\ -10 & \mathbf{0} & 10 \\ -3 & 0 & 3 \end{bmatrix} \quad \text{and} \quad H_y^{S'} = \frac{1}{32}\begin{bmatrix} -3 & -10 & -3 \\ 0 & \mathbf{0} & 0 \\ 3 & 10 & 3 \end{bmatrix}. \tag{7.15}$$

These edge operators are frequently used because of their good results (see also Fig. 7.10) and simple implementation. The Sobel operator, in particular, is available in many image-processing tools and software packages (including ImageJ).

7.3.2 Roberts Operator

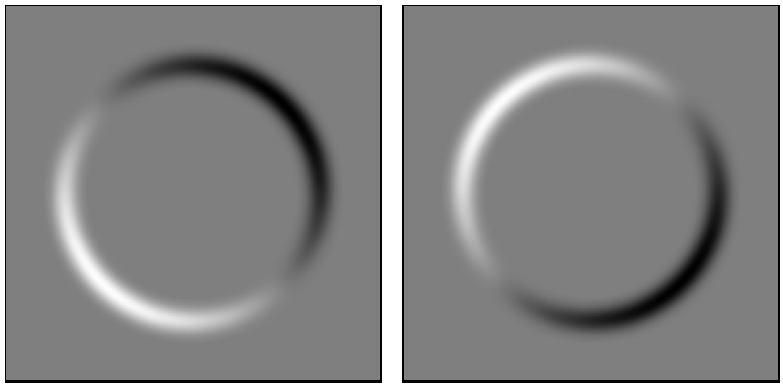
As one of the simplest and oldest edge finders, the Roberts operator [83] today is mainly of historical interest. It employs two extremely small filters of size 2×2 for estimating the directional gradient along the image diagonals:

$$H_1^R = \begin{bmatrix} 0 & \mathbf{1} \\ -1 & 0 \end{bmatrix} \quad \text{and} \quad H_2^R = \begin{bmatrix} -1 & 0 \\ 0 & \mathbf{1} \end{bmatrix}. \tag{7.16}$$

These filters naturally respond to diagonal edges but are not highly selective to orientation; i. e., both filters show strong results over a relatively wide range of angles (Fig. 7.6). The local edge strength is computed by measuring the length of the resulting 2D vector, similar to the gradient computation but with its components rotated 45° (Fig. 7.7).

7.3.3 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 Sobel operators) but a larger set of filters with narrowly spaced orientations instead. A classic example is the edge



$D_1 = I * H_1^R$ $D_2 = I * H_2^R$

Fig. 7.6 Diagonal gradient components produced by the two Roberts filters.

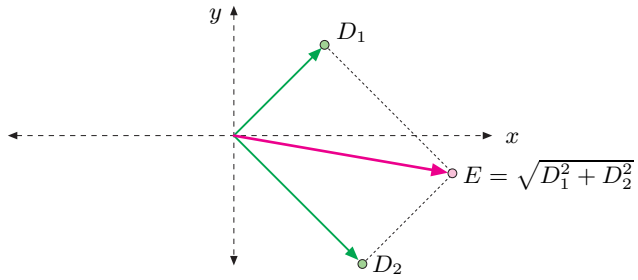


Fig. 7.7 Definition of edge strength for the Roberts operator. The edge strength $E(u,v)$ corresponds to the length of the vector obtained by adding the two orthogonal gradient components (filter results) $D_1(u,v)$ and $D_2(u,v)$.

operator by *Kirsch* [63], which employs the following eight filters with orientations spaced at 45°:

$$H_0^K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad H_4^K = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \tag{7.17}$$

$$H_1^K = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} \qquad H_5^K = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}, \tag{7.18}$$

$$H_2^K = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad H_6^K = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, \tag{7.19}$$

$$H_3^K = \begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix} \qquad H_7^K = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}. \tag{7.20}$$

Only the results of four of the eight filters $H_0, H_1, \ldots H_7$ above must actually be computed since the four others are identical except for the reversed sign. For example, from the fact that $H_4^K = -H_0^K$ and the convolution being linear (Eqn. (6.18)), it follows that

$$I * H_4^K \;=\; I * -H_0^K \;=\; -(I * H_0^K); \tag{7.21}$$

i. e., the result for filter H_4^K is simply the negative result for filter H_0^K. The directional outputs $D_0, D_1, \ldots D_7$ for the eight Kirsch filters can thus be computed as follows:

$$\begin{array}{llll} D_0 \leftarrow I * H_0^K & D_1 \leftarrow I * H_1^K & D_2 \leftarrow I * H_2^K & D_3 \leftarrow I * H_3^K \\ D_4 \leftarrow -D_0 & D_5 \leftarrow -D_1 & D_6 \leftarrow -D_2 & D_7 \leftarrow -D_3. \end{array} \tag{7.22}$$

The edge strength E^K at position (u, v) is defined as the maximum of the eight filter outputs; i. e.,

$$\begin{aligned} E^K(u,v) &\triangleq \max\big(D_0(u,v), D_1(u,v), \ldots D_7(u,v)\big) \\ &= \max\big(|D_0(u,v)|, |D_1(u,v)|, |D_2(u,v)|, |D_3(u,v)|\big) \end{aligned} \tag{7.23}$$

and the strongest-responding filter also determines the local edge orientation as

$$\Phi^K(u,v) \triangleq \frac{\pi}{4} \qquad \text{with } \; j = \operatorname*{argmax}_{0 \le i \le 7} D_i(u,v). \tag{7.24}$$

In practice, however, 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).

7.3.4 Edge Operators in ImageJ

The current version of ImageJ implements the Sobel operator (as described in Eqn. (7.10)) 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`.

7.4 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.

7.4.1 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. 7.8 for the one-dimensional case. Since second derivatives generally tend to amplify image noise, some sort of presmoothing is usually applied with suitable low-pass filters.

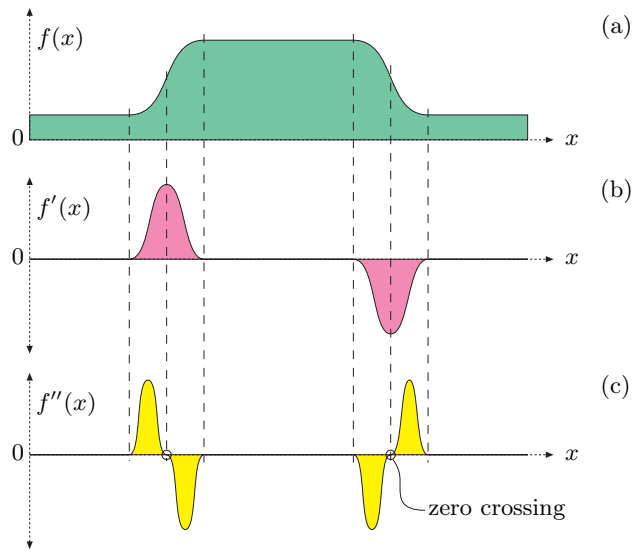


Fig. 7.8
Principle of edge detection with the second derivative: original function (a), first derivative (b), and second derivative (c). Edge points are located where the second derivative crosses through zero and the first derivative has a high magnitude.

A popular example is the "Laplacian-of-Gaussian" (LoG) operator [69], which combines gaussian smoothing and computing the second derivatives (see the *Laplace Filter* in Sec. 7.6.1) into a single linear filter. The example in Fig. 7.10 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 [85, Ch. 4] and [73].

7.4.2 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:

- 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 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 [18, 65]. 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.

7.4.3 Canny Operator

A popular example for such a method is the edge operator by Canny [19], which employs a set of relatively large, oriented filters at multiple image resolutions and merges the individual results into a common edge map. 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. At its core, the Canny "filter" is a gradient method (based on first derivatives; see Sec. 7.2), but it uses the zero crossings of second derivatives for precise edge localization. Frequently, however, only a single-scale implementation of the algorithm with an adjustable filter radius (smoothing parameter σ) is used, which is nevertheless superior to most of the simple edge operators (see Figs. 7.9 and 7.10). Thus, even in its basic (single-scale) form, the Canny operator is often preferred over other edge detection methods. A more detailed description of the algorithm and a Java implementation can be found, for example, in [29, Ch. 7].

7.5 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?

7.5.1 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

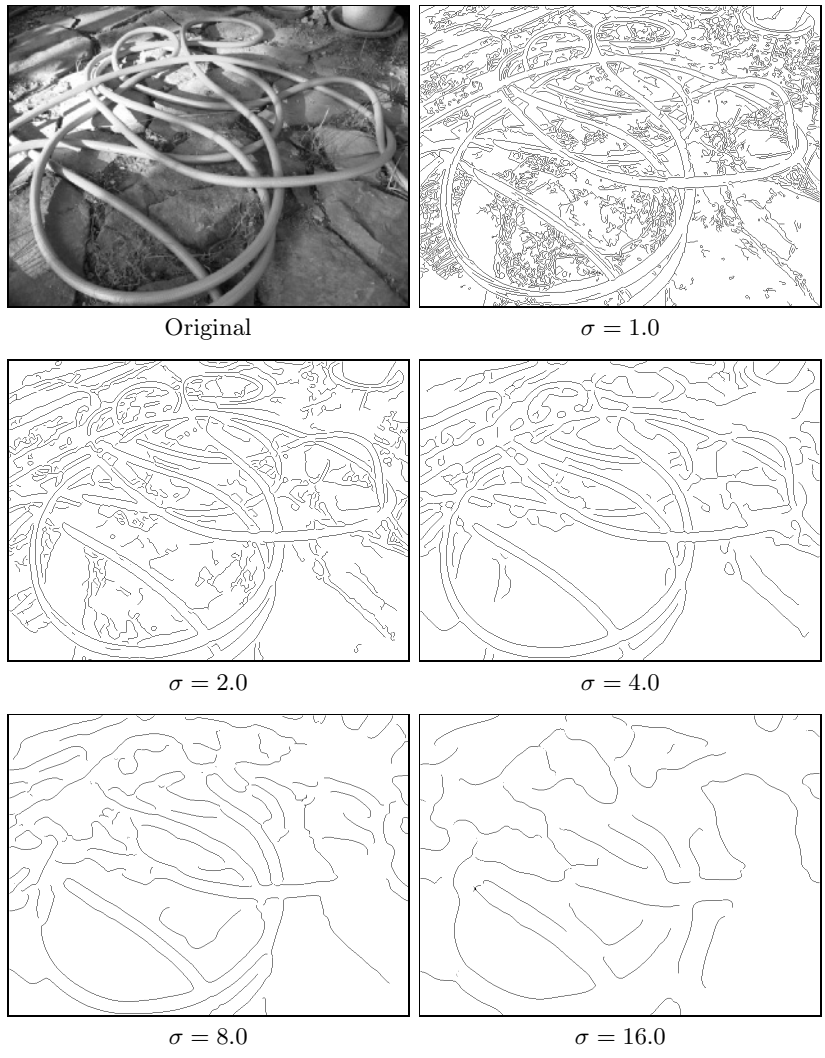


Fig. 7.9
Canny edge operator. Resulting edge maps for different settings of the smoothing (scale) parameter σ.

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.
- 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 binary images is much simpler, of course (see Ch. 11).

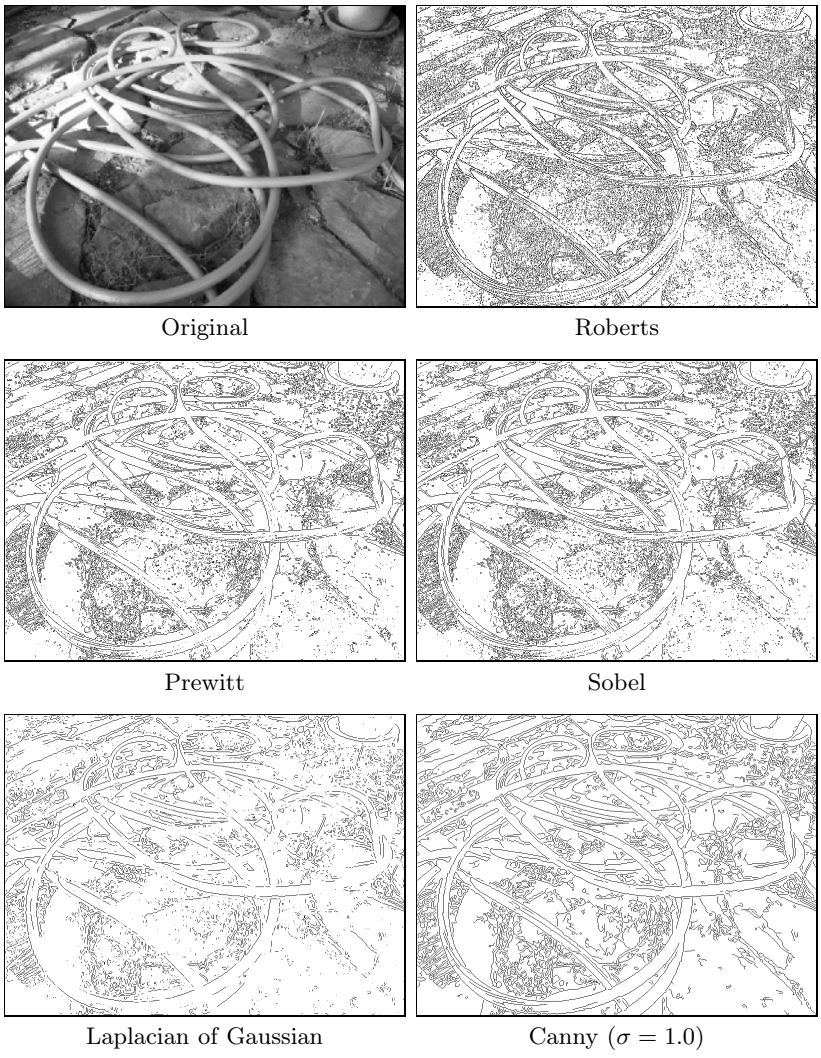


Fig. 7.10
Comparison of various edge operators. Important criteria for the quality of edge results are the amount of "clutter" (irrelevant edge elements) and the connectedness of dominant edges. The Roberts operator responds to very small edge structures because of the small size of its filters. The 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.

7.5.2 Edge Maps

In many situations, the next step after edge enhancement (by some edge operator) is the selection of edge points, a binary decision 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. 9), can cope well with incomplete edge maps.

7.6 Edge Sharpening

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 precompensate for a subsequent loss of sharpness in the course of printing or displaying an image. The 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.

7.6.1 Edge Sharpening with the Laplace Filter

A common method for localizing rapid intensity changes are filters based on the second derivatives of the image function. Figure 7.11 illustrates this idea on a one-dimensional, 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 $f(x)$,

$$\check{f}(x) = f(x) - w \cdot f''(x). \tag{7.25}$$

Depending upon the weight factor $w \geq 0$, the expression in Eqn. (7.25) causes the intensity function to overshoot at both sides of an edge, thus exaggerating edges and increasing the perceived sharpness.

Laplace operator

Sharpening of a two-dimensional function can be accomplished with the second derivatives in the horizontal and vertical directions combined by the so-called Laplace operator. The Laplace operator ∇^2 of a two-dimensional function $f(x, y)$ is defined as the sum of the second partial derivatives along the x and y directions:

$$\big(\nabla^2 f\big)(x, y) = \frac{\partial^2 f}{\partial^2 x}(x, y) + \frac{\partial^2 f}{\partial^2 y}(x, y). \tag{7.26}$$

Similar to the first derivatives (see Sec. 7.2.2), the second derivatives of a discrete image function can also be estimated with a set of simple linear

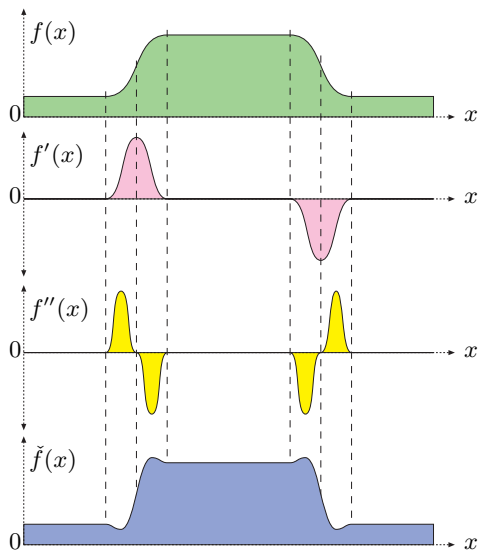


Fig. 7.11
Edge sharpening with the second derivative. The original intensity function $f(x)$, first derivative $f'(x)$, second derivative $f''(x)$, and sharpened intensity function $\check{f}(x) = f(x) - w \cdot f''(x)$ are shown.

filters. Again, several versions, have been proposed. For example, the two one-dimensional filters

$$\frac{\partial^2 f}{\partial^2 x} \equiv H_x^L = \begin{bmatrix} 1 & -\mathbf{2} & 1 \end{bmatrix} \quad \text{and} \quad \frac{\partial^2 f}{\partial^2 y} \equiv H_y^L = \begin{bmatrix} 1 \\ -\mathbf{2} \\ 1 \end{bmatrix} \tag{7.27}$$

for estimating the second derivatives along the x and y directions, respectively, combine to make the two-dimensional Laplace filter

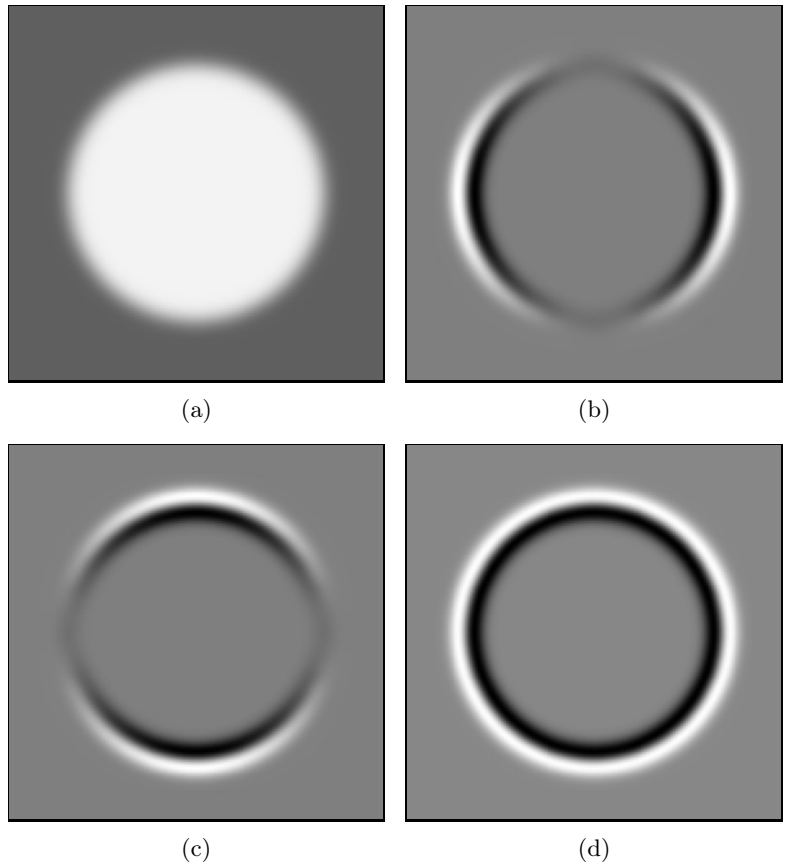
$$H^L = H_x^L + H_y^L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -\mathbf{4} & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{7.28}$$

Figure 7.12 shows an example of applying the Laplace 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. (7.28) is not a *separable* filter in the usual sense (as described in Sec. 6.3.3) but, because of the linearity property of convolution (Eqns. (6.17) and (6.19)), it can be expressed (and computed) as the *sum* of two one-dimensional filters,

$$I * H^L = I * (H_x^L + H_y^L) = (I * H_x^L) + (I * H_y^L).$$

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. 7.12). Other common variants of 3×3 pixel Laplace filters are



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

Fig. 7.12
Results of Laplace filter H^L: synthetic test image I (a), second partial derivative $\partial^2 I/\partial^2 u$ in the horizontal direction (b), second partial derivative $\partial^2 I/\partial^2 v$ in the vertical direction (c), and Laplace filter $\nabla^2 I(u, v)$ (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.

$$H_8^L = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \mathbf{-8} & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{and} \quad H_{12}^L = \begin{bmatrix} 1 & 2 & 1 \\ 2 & \mathbf{-12} & 2 \\ 1 & 2 & 1 \end{bmatrix}. \tag{7.29}$$

Sharpening

To perform the actual sharpening, as described by Eqn. (7.25) for the one-dimensional case, we first apply a Laplace filter to the image I and then subtract a fraction of the result from the original image,

$$\check{I} \leftarrow I - w \cdot (H^L * I). \tag{7.30}$$

The factor w specifies the proportion of the Laplace component and thus the sharpening strength. The proper choice of w also depends on the specific Laplace filter used in Eqn. (7.30) since none of the filters above is normalized.

Figure 7.12 shows the result of applying a Laplace filter (with the kernel given in Eqn. (7.28)) to a synthetic test image where the pairs of positive/negative peaks at both sides of each edge are clearly visible.

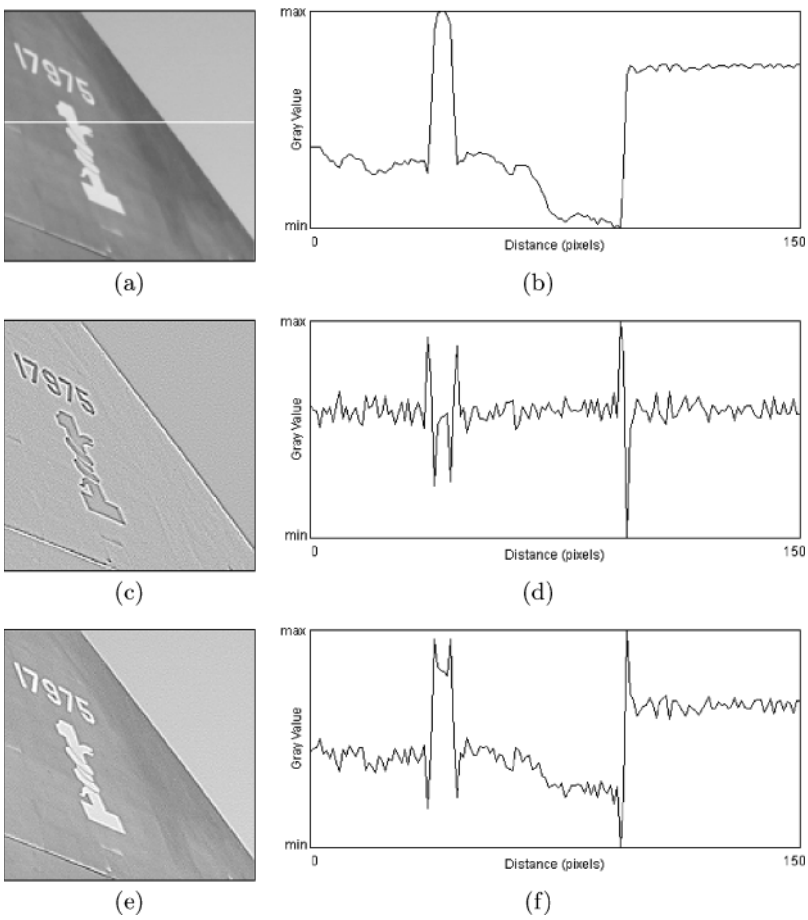


Fig. 7.13
Edge sharpening with the Laplace filter: original image with a horizontal profile taken from the marked line (a, b), result of Laplace filter H^L (c, d), and sharpened image (e, f).

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. (7.28)) and $w = 1.0$ is shown in Fig. 7.13.

As we can expect from second-order derivatives, the Laplace 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. 7.4.1).

7.6.2 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 M is generated by subtracting a smoothed version of the image I from the original,

$$M \leftarrow I - (I * \tilde{H}) = I - \tilde{I}, \tag{7.31}$$

   where the kernel $\tilde{H}$ of the smoothing filter is assumed to be normalized (see Sec. 6.2.5).
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,

$$\check{I} \leftarrow I + a \cdot M, \tag{7.32}$$

   and thus (substituting from Eqn. (7.31))

$$\check{I} \leftarrow I + a \cdot (I - \tilde{I}) = (1 + a) \cdot I - a \cdot \tilde{I}. \tag{7.33}$$

Smoothing filter

In principle, any smoothing filter could be used for the kernel $\tilde{H}$ in Eqn. (7.31), but Gaussian filters $H^{G,\sigma}$ with variable radius σ are most common (see also Sec. 6.2.7). 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 7.14 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 a 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),

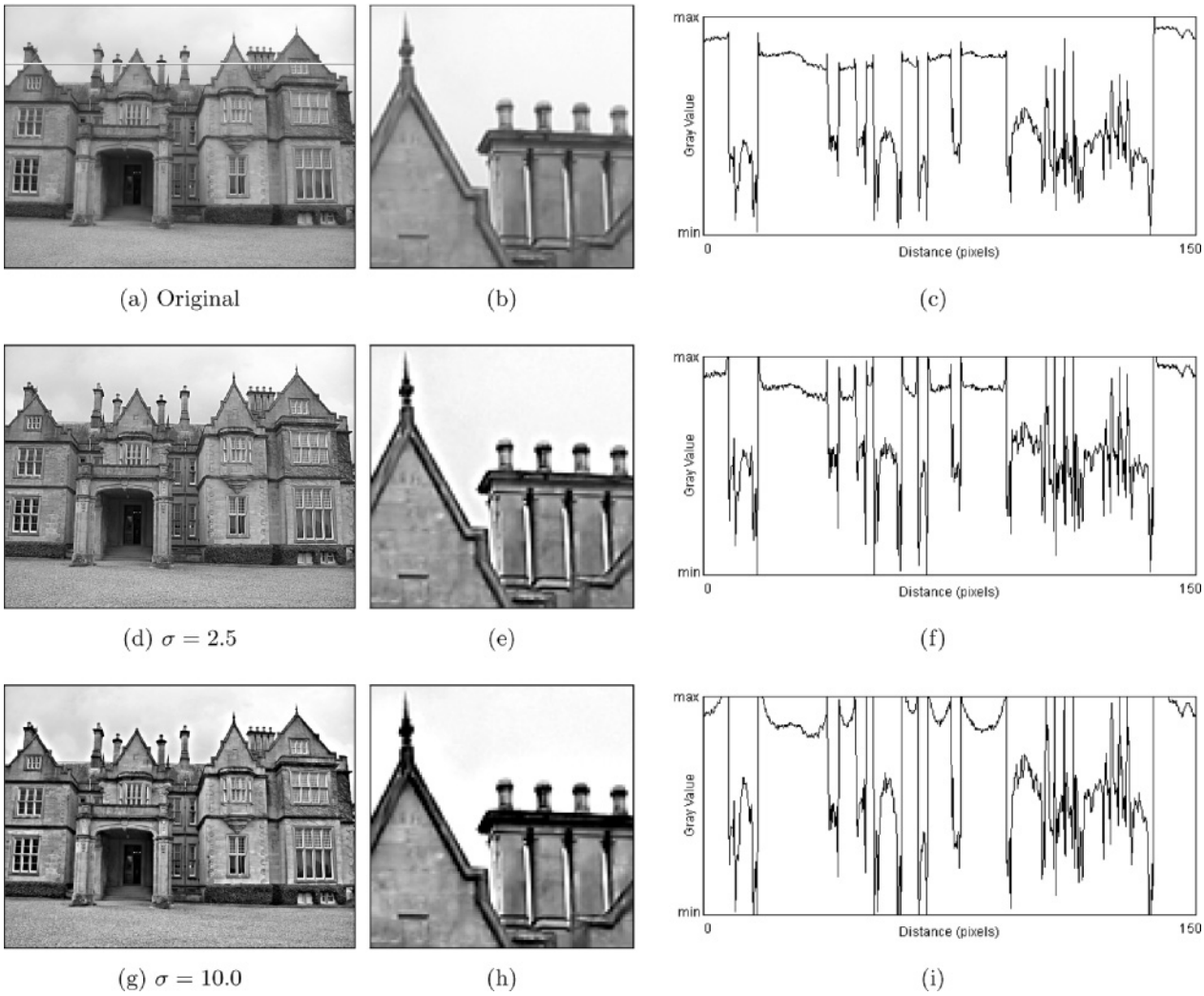


Fig. 7.14. USM filter: original image (a), detail (b), and intensity profile of marked image line (c); results of USM filtering with Gaussian smoothing radius $\sigma = 2.5$ (d–f) and 10.0 (g–i). The value of the sharpening factor a is 1.0 (100%).

expressed for example by the gradient magnitude $|\nabla I|$ (Eqn. (7.5)), is greater than that threshold. Otherwise, that pixel remains unmodified:

$$\check{I}(u,v) \leftarrow \begin{cases} I(u,v) + a \cdot M(u,v) & \text{for } |\nabla I|(u,v) \geq t_c \\ I(u,v) & \text{otherwise.} \end{cases} \tag{7.34}$$

Different from the original USM filter (Eqn. (7.32)), 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 `ij.plugin.filter.UnsharpMask`, which can be invoked through the menu

Process→Filter→Unsharp Mask...

ImageJ's `UnsharpMask` implementation uses the class `GaussianBlur` for the required smoothing operation, whose filter kernels tend to be too small (as argued in Sec. 6.6.2). The alternative implementation shown in Prog. 7.1 follows the definition in Eqn. (7.33) and uses sufficiently large filter kernels that are created with the method `makeGaussKernel1d()`, as defined in Prog. 6.4.

Laplace versus USM filter

A closer look at these two methods reveals that sharpening with the Laplace filter (Sec. 7.6.1) can be viewed as a special case of the USM filter. If the Laplace filter in Eqn. (7.28) is decomposed as

$$
\begin{aligned}
H^L &= \begin{bmatrix} 0 & 1 & 0 \\ 1 & -\mathbf{4} & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 \\ 0 & 1 & 0 \end{bmatrix} - 5 \begin{bmatrix} 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
&= 5 \left(\frac{1}{5} \begin{bmatrix} 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 \\ 0 & 1 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 \end{bmatrix} \right) = 5 \left(\tilde{H} - \delta \right), \qquad (7.35)
\end{aligned}
$$

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. (7.30) can therefore (by a little manipulation) be expressed as

$$
\begin{aligned}
\check{I}_L &\leftarrow I - w \cdot (H^L * I) = I - w \cdot \big(5(\tilde{H}^L - \delta) * I\big) \\
&= I - 5w \cdot (\tilde{H}^L * I - I) = I + 5w \cdot (I - \tilde{H}^L * I) \\
&= I + 5w \cdot M^L; \qquad (7.36)
\end{aligned}
$$

i. e., in the form of a USM filter $\check{I} \leftarrow I + a \cdot M$ (Eqn. (7.32)). 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

```
1  public void unsharpMask(ImageProcessor ip,
2       double sigma, double a) {
3    ImageProcessor I = ip.convertToFloat(); // I
4
5    // create a blurred version of the image
6    ImageProcessor J = I.duplicate();       // Ĩ
7    float[] H = GaussKernel1d.create(sigma);  // see Prog. 6.4
8    Convolver cv = new Convolver();
9    cv.setNormalize(true);
10   cv.convolve(J, H, 1, H.length);
11   cv.convolve(J, H, H.length, 1);
12
13   I.multiply(1+a);     // I ← (1 + a) · I
14   J.multiply(a);       // Ĩ ← a · Ĩ
15   I.copyBits(J,0,0,Blitter.SUBTRACT); // Ĩ ← (1 + a) · I − a · Ĩ
16
17   //copy result back into original byte image
18   ip.insert(I.convertToByte(false), 0, 0);
19 }
```

$$\tilde{H} = \tilde{H}^L = \frac{1}{5}\begin{bmatrix} 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

and the sharpening factor $a = 5w$.

7.7 Exercises

Exercise 7.1. Compute manually the gradient and the Laplacian for the following image function by using the approximations in Eqn. (7.2) and Eqn. (7.28), respectively:

$$I(u,v) = \begin{bmatrix} 14 & 10 & 19 & 16 & 14 & 12 \\ 18 & 9 & 11 & 12 & 10 & 19 \\ 9 & 14 & 15 & 26 & 13 & 6 \\ 21 & 27 & 17 & 17 & 19 & 16 \\ 11 & 18 & 18 & 19 & 16 & 14 \\ 16 & 10 & 13 & 7 & 22 & 21 \end{bmatrix}.$$

Exercise 7.2. Implement the Sobel edge operator as defined in Eqn. (7.10) (and illustrated in Fig. 7.5) 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 the edge orientation.

Exercise 7.3. Express the Sobel operator in x/y-separable form analogous to Eqn. (7.9) (Prewitt operator).

Program 7.1
Unsharp masking (Java implementation). First the original image is converted to a `FloatProcessor` object `I` (I) in line 3, which is duplicated to hold the blurred image `J` ($\tilde{I}$) in line 6. The method `makeGaussKernel1d()`, defined in Prog. 6.4, is used to create the 1D Gaussian filter kernel applied in the horizontal and vertical directions (lines 10–11). The remaining computations follow Eqn. (7.33).

Exercise 7.4. Implement the Kirsch operator analogous to Exercise 7.2 and in particular compare the estimates for the edge orientation produced by the two methods.

Exercise 7.5. Devise and implement a compass edge operator with more than 8 (16?) differently oriented filters.

Exercise 7.6. 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?

8

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. Corner points are important not only in human vision, where they alert us to boundaries, but also in machine vision, where they belong to the small set of features referred to as "robust". Robust features are those that, for the most part, do not arise accidentally in 3D scenes and furthermore can be relatively consistently and accurately located under a wide range of viewing angles and lighting conditions.

8.1 Points of Interest

Despite being easily recognized by our visual system, accurately and precisely detecting corners automatically is nontrivial. 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, and finally it should be possible to implement the detector efficiently enough so that it can be utilized in real-time applications such as video tracking.

While a number of methods for finding corners, and related interest points, have been proposed, 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 an area 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 (e. g., [33, 42, 64, 67]) to find corners. In the next section, we describe in detail the Harris detector, also known as the "Plessey feature point detector" [42], since it turns out that even though more efficient detectors are known (see, for example, [87,94]), the Harris detector, and other detectors based on it, are the most widely used in practice.

8.2 Harris Corner Detector

This operator, developed by Harris and Stephens [42], 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, most of these detectors take advantage of the heuristics that locations along edges, where the gradient is strong in only one direction, should not be considered as corners, and since corners can exist in any orientation, the detector should be isotropic (i. e., orientation independent).

8.2.1 Local Structure Matrix

Computations based on the first partial derivative of the image function $I(u, v)$ in the horizontal and vertical directions are the foundation of the Harris detector:

$$I_x(u, v) = \frac{\partial I}{\partial x}(u, v) \quad \text{and} \quad I_y(u, v) = \frac{\partial I}{\partial y}(u, v). \tag{8.1}$$

For each location in the image (u, v), we first compute the three values $A(u, v)$, $B(u, v)$, and $C(u, v)$,

$$A(u, v) = I_x^2(u, v), \tag{8.2}$$
$$B(u, v) = I_y^2(u, v), \tag{8.3}$$
$$C(u, v) = I_x(u, v) \cdot I_y(u, v), \tag{8.4}$$

which will be interpreted as elements of the *local structural matrix* $M(u, v)$:[1]

$$\boldsymbol{M} = \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix} = \begin{pmatrix} A & C \\ C & B \end{pmatrix}. \tag{8.5}$$

Next, each of the three functions $A(u, v)$, $B(u, v)$, $C(u, v)$ is individually smoothed by convolution with a linear Gaussian filter $H^{G,\sigma}$ (see Sec. 6.2.7),

[1] For improved legibility, the notation used in the following functions has been simplified by omitting the coordinates (u, v); e. g., $I_x \equiv I_x(u, v)$ or $A \equiv A(u, v)$.

$$\bar{M} = \begin{pmatrix} A*H^{G,\sigma} & C*H^{G,\sigma} \\ C*H^{G,\sigma} & B*H^{G,\sigma} \end{pmatrix} = \begin{pmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{pmatrix}. \tag{8.6}$$

Since the matrix $\bar{M}$ is symmetric, it can be diagonalized,

$$\bar{M}' = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}, \tag{8.7}$$

where λ_1 and λ_2 are the *eigenvalues* of the matrix $\bar{M}$, defined as[2]

$$\begin{aligned} \lambda_{1,2} &= \frac{\text{trace}(\bar{M})}{2} \pm \sqrt{\Big(\frac{\text{trace}(\bar{M})}{2}\Big)^2 - \det(\bar{M})} \\ &= \frac{1}{2}\left(\bar{A} + \bar{B} \pm \sqrt{\bar{A}^2 - 2\bar{A}\bar{B} + \bar{B}^2 + 4\bar{C}^2}\right). \end{aligned} \tag{8.8}$$

These eigenvalues, which are positive and real, contain essential information about the local image structure. Within an image region that is uniform (that is, appears flat), $\bar{M} = 0$ and therefore $\lambda_1 = \lambda_2 = 0$. On the other hand, given an ideal ramp, $\lambda_1 > 0$ and $\lambda_2 = 0$, independent of the orientation of the edge. The eigenvalues thus encode an edge's *strength*, and their associated *eigenvectors* represent the edge's *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 the eigenvalues of both of these must be significant. Since $\bar{A}, \bar{B} \geq 0$, we can assume that $\text{trace}(\bar{M}) > 0$ and thus $|\lambda_1| \geq |\lambda_2|$. Therefore only the smaller of the two eigenvalues, $\lambda_2 = \text{trace}(\bar{M})/2 - \sqrt{\ldots}$, is relevant when determining a corner.

8.2.2 Corner Response Function (CRF)

As we can see from Eqn. (8.8), the difference between the two eigenvalues is

$$\lambda_1 - \lambda_2 = 2 \cdot \sqrt{\frac{1}{4} \cdot \big(\text{trace}(\bar{M})\big)^2 - \det(\bar{M})},$$

where in every case $\big(0.25 \cdot \text{trace}(\bar{M})^2\big) > \det(\bar{M})$ holds. At a corner, this expression should be as small as possible, and therefore the Harris detector defines the function

$$\begin{aligned} Q(u,v) &= \det(\bar{M}) - \alpha \cdot \big(\text{trace}(\bar{M})\big)^2 \\ &= (\bar{A}\bar{B} - \bar{C}^2) - \alpha \cdot (\bar{A} + \bar{B})^2 \end{aligned} \tag{8.9}$$

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.

[2] Where $\det(\bar{M})$ is the *determinant* and $\text{trace}(\bar{M})$ the *trace* of the matrix $\bar{M}$ (see, for example, [15, pp. 259, 252]).

8.2.3 Determining Corner Points

An image location (u, v) is selected as a candidate for a corner point when

$$Q(u, v) > t_H,$$

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 $\boldsymbol{c}_i = \langle u_i, v_i, q_i \rangle$ are inserted into the set

$$\mathit{Corners} = [\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots \boldsymbol{c}_N],$$

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. (8.9). To suppress the false corners that tend to arise in densely packed groups around true corners, all except the strongest corner in a specified area are eliminated. To accomplish this, the list *Corners* is traversed from the front to the back, and the weaker corners toward the end of the list, which lie in the neighborhood of a stronger corner, are deleted.

The complete algorithm for the Harris detector is summarized again in Alg. 8.1, and the associated parameters are explained in Fig. 8.1.

8.2.4 Example

Figure 8.2 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 $M(u, v) = \left(\begin{smallmatrix} A & C \\ C & B \end{smallmatrix}\right)$, and the values of the *corner response function* $Q(u, v)$ for each image position (u, v). This example utilizes the standard settings as given in Fig. 8.1.

The second example (Fig. 8.3) 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.

8.3 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 sections you may wish to refer to the complete source code for the class `HarrisCornerDetector`, which can be found in Appendix D (pp. 525–532).

1: HarrisCorners(I)

Returns a list of the strongest corners found in the image I.

2: Step 1—compute the corner response function:

3: Prefilter (smooth) the original image: $I' \leftarrow I * H_p$

4: Compute the horizontal and vertical image derivatives:
$I_x \leftarrow I' * H_{dx}$
$I_y \leftarrow I' * H_{dy}$

5: Compute the local structure matrix $M(u,v) = \left(\begin{smallmatrix} A & C \\ C & B \end{smallmatrix}\right)$:
$A(u,v) \leftarrow I_x^2(u,v)$
$B(u,v) \leftarrow I_y^2(u,v)$
$C(u,v) \leftarrow I_x(u,v) \cdot I_y(u,v)$

6: Blur each component of the structure matrix: $\bar{M} = \left(\begin{smallmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{smallmatrix}\right)$:
$\bar{A} \leftarrow A * H_b$
$\bar{B} \leftarrow B * H_b$
$\bar{C} \leftarrow C * H_b$

7: Compute the corner response function:
$Q \leftarrow (\bar{A} \cdot \bar{B} - \bar{C}^2) - \alpha \cdot (\bar{A} + \bar{B})^2$

8: Step 2—collect corner points:

9: Create an empty list:
Corners $\leftarrow$ [ ]

10: **for** all image coordinates (u,v) **do**

11: **if** $Q(u,v) > t_H$ **and** IsLocalMax(Q,u,v) **then**

12: Create a new corner:
$\boldsymbol{c}_i \leftarrow \langle u_i, v_i, q_i \rangle = \langle u, v, Q(u,v) \rangle$

13: Add $\boldsymbol{c}_i$ to *Corners*

14: **Sort** *Corners* by q_i in *descending* order (strongest corners first)

15: *GoodCorners* $\leftarrow$ CleanUpNeighbors(*Corners*)

16: **return** *GoodCorners*.

17: IsLocalMax(Q,u,v) ▷ determine if $Q(u,v)$ is a local maximum

18: Let $q_c \leftarrow Q(u,v)$ (center pixel)

19: Let $\mathcal{N} \leftarrow$ *Neighbors*(Q,u,v) ▷ values of all neighboring pixels

20: **if** $q_c \geq q_i$ for all $q_i \in \mathcal{N}$ **then**

21: **return** *true*

22: **else**

23: **return** *false*.

24: CleanUpNeighbors(*Corners*) ▷ *Corners* is sorted by descending q

25: Create an empty list:
GoodCorners $\leftarrow$ [ ]

26: **while** *Corners* is not empty **do**

27: $\boldsymbol{c}_i \leftarrow$ RemoveFirst(*Corners*)

28: Add $\boldsymbol{c}_i$ to *GoodCorners*

29: **for** all $\boldsymbol{c}_j$ in *Corners* **do**

30: **if** *Dist*$(\boldsymbol{c}_i, \boldsymbol{c}_j) < d_{\min}$ **then**

31: Delete $\boldsymbol{c}_j$ from *Corners*

32: **return** *GoodCorners*.

Algorithm 8.1
Harris corner detector. This algorithm takes an intensity image I and creates a sorted list of detected corner points. Details for the parameters H_p, H_{dx}, H_{dy}, H_b, α, t_H, and $d_{\min}$, can be found in Fig. 8.1.

Fig. 8.1
Harris corner detector—actual parameter values. The line numbers refer to Alg. 8.1.

Prefilter (line 3): Smoothing with a small xy-separable filter $H_p = H_{px} * H_{py}$, where

$$H_{px} = \frac{1}{9}\begin{bmatrix}2 & \mathbf{5} & 2\end{bmatrix} \quad \text{and} \quad H_{py} = H_{px}^T = \frac{1}{9}\begin{bmatrix}2\\ \mathbf{5}\\ 2\end{bmatrix}.$$

Gradient filter (line 4): Computing the first partial derivative in the x and y directions with

$$H_{dx} = \begin{bmatrix}-0.453014 & \mathbf{0} & 0.453014\end{bmatrix} \quad \text{and} \quad H_{dy} = H_{dx}^T = \begin{bmatrix}-0.453014\\ \mathbf{0}\\ 0.453014\end{bmatrix}.$$

Blurfilter (line 6): Smoothing the individual components of the structure matrix M with separable Gaussian filters $H_b = H_{bx} * H_{by}$ with

$$H_{bx} = \frac{1}{64}\begin{bmatrix}1 & 6 & 15 & \mathbf{20} & 15 & 6 & 1\end{bmatrix}, \qquad H_{by} = H_{bx}^T = \frac{1}{64}\begin{bmatrix}1\\ 6\\ 15\\ \mathbf{20}\\ 15\\ 6\\ 1\end{bmatrix}.$$

Steering parameter (line 7): $\alpha = 0.04$ to 0.06 (default 0.05)
Response threshold (line 13): $t_H = 10{,}000$ to $1{,}000{,}000$ (default 25,000)
Neighborhood radius (line 31): $d_{\min} = 10$ pixels

8.3.1 Step 1: Computing the Corner Response Function

In order to handle the range of the positive and negative values generated in this step by the filter process, we will need to use floating-point images to store the intermediate results. In order to correctly, and precisely, represent these small values, it is necessary to store them as floating-point values. The kernel of this filter, as well as the presmoothing H_p, the gradients H_{dx}, H_{dy}, and the smoothing filter for the structure matrix H_b are stored as one-dimensional `float` arrays:

```
1 float[] pfilt = {0.223755f,0.552490f,0.223755f}; // Hp
2 float[] dfilt = {0.453014f,0.0f,-0.453014f}; // Hdx, Hdy
3 float[] bfilt = {0.01563f,0.09375f,0.234375f,0.3125f,
4                  0.234375f,0.09375f,0.01563f}; // Hb
```

From the original 8-bit image (of type `ByteProcessor`), we first create two copies, `Ix` and `Iy`, of type `FloatProcessor`:

```
5 FloatProcessor Ix = (FloatProcessor) ip.convertToFloat();
6 FloatProcessor Iy = (FloatProcessor) ip.convertToFloat();
```

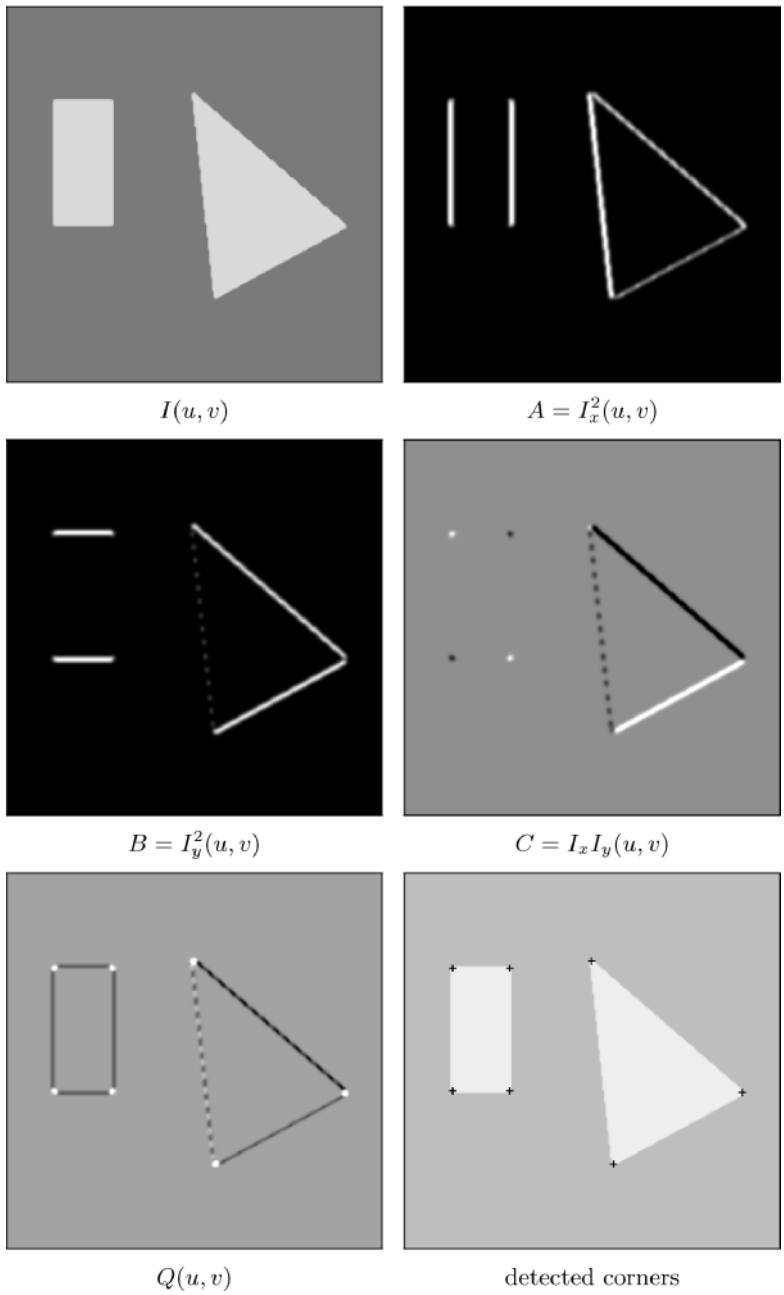


Fig. 8.2
Harris corner detector—Example 1. Starting with the original image $I(u,v)$, the first derivative is computed, and then from it the components of the structure matrix $A = I_x^2$, $B = I_y^2$, $C = I_x I_y$. A and B represent, respectively, the strength of the horizontal and vertical edges. In C, 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, exhibits noticeable positive spikes at the corner positions. Final corners are determined by thresholding and then finding the local maxima of the function Q.

The first processing step is a presmoothing with the filter H_p (Alg. 8.1, line 3). Subsequently the gradient filters H_{dx} and H_{dy} are used to compute the horizontal and vertical derivatives (Alg. 8.1, line 4). Since one-dimensional filters of the same direction are applied in each step, pres-

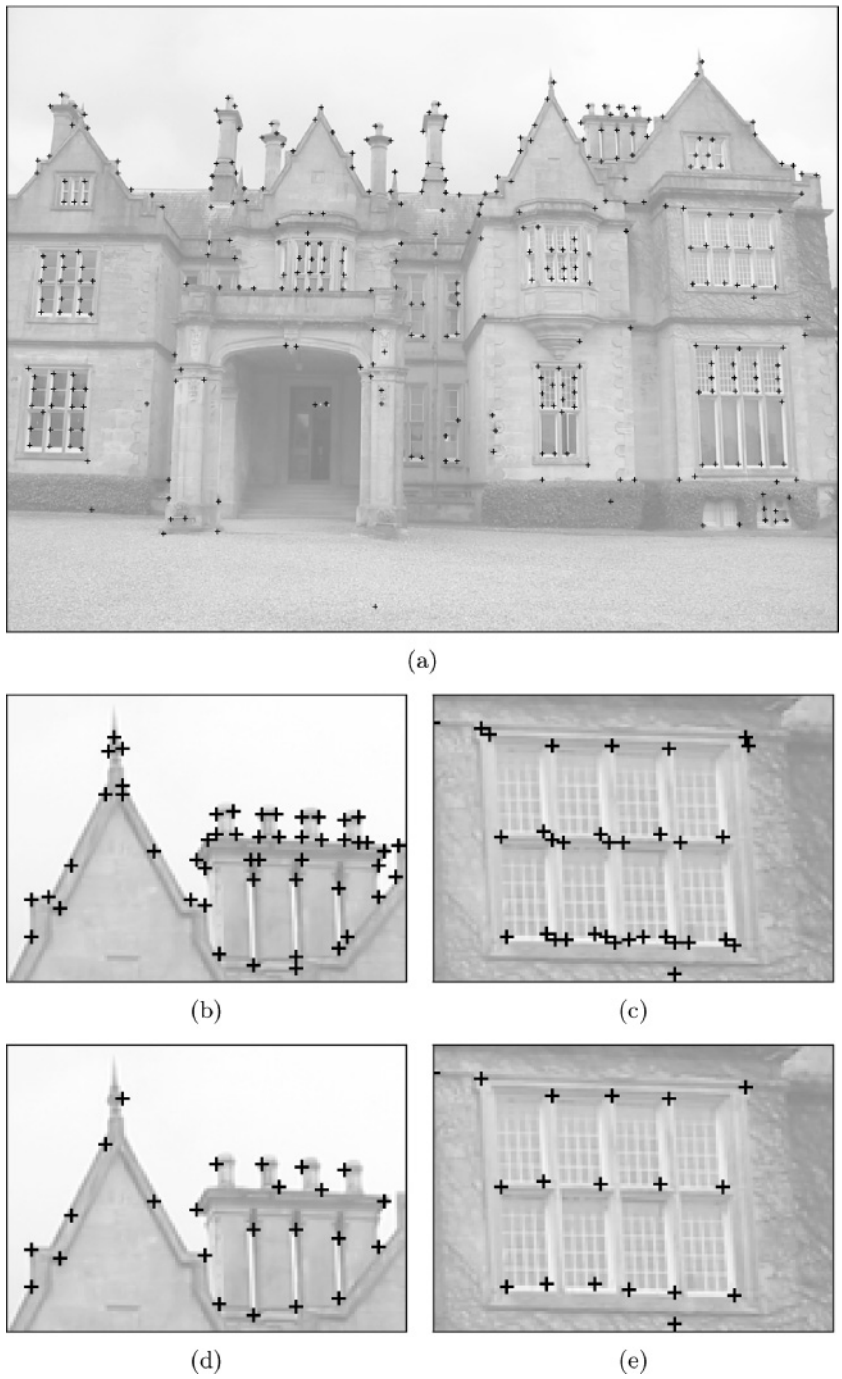


Fig. 8.3
Harris corner detector—Example 2. A complete result with the final corner points marked (a). After selecting the strongest corner points within a 10-pixel radius, only 335 of the original 615 candidate corners remain. Details *before* (b, c) and *after* selection (d, e).

moothing and gradient computation can be combined in a single step:

```
7 Ix = convolve1h(convolve1h(Ix,pfilt),dfilt);
8 Iy = convolve1v(convolve1v(Iy,pfilt),dfilt);
```

The methods `convolve1h(`I, h`)` and `convolve1v(`I, h`)` above perform one-dimensional filter operations h on the image I in the horizontal and vertical directions, respectively (see "filter methods" below). Now the components A, B, C of the structure matrix are computed and then smoothed using the separable 2D filter H_b (`bfilt`):

```
9  A = sqr ((FloatProcessor) Ix.duplicate());
10 B = sqr ((FloatProcessor) Iy.duplicate());
11 C = mult((FloatProcessor) Ix.duplicate(),Iy);
12
13 A = convolve2(A,bfilt);    // convolve with Hb
14 B = convolve2(B,bfilt);
15 C = convolve2(C,bfilt);
```

The variables `A`, `B`, `C` of type `FloatProcessor` are declared in the class `HarrisCornerDetector`. The method `convolve2(`I, h`)` performs a separable 2D convolution of the image I using the 1D filter kernel h. `mult()` and `sqr()` are auxiliary methods for multiplying two images and squaring an image, respectively (see Appendix D, p. 531).

Finally, the corner response function (Alg. 8.1, line 7) is computed using the method `makeCrf()`, and a new image of type `FloatProcessor` is created:

```
16 void makeCrf() { // defined in class HarrisCornerDetector
17   int w = ipOrig.getWidth();
18   int h = ipOrig.getHeight();
19   Q = new FloatProcessor(w,h);
20   float[] Apix = (float[]) A.getPixels();
21   float[] Bpix = (float[]) B.getPixels();
22   float[] Cpix = (float[]) C.getPixels();
23   float[] Qpix = (float[]) Q.getPixels();
24   for (int v=0; v<h; v++) {
25     for (int u=0; u<w; u++) {
26       int i = v*w+u;
27       float a = Apix[i], b = Bpix[i], c = Cpix[i];
28       float det = a*b-c*c;          // det(M̄)
29       float trace = a+b;            // trace(M̄)
30       Qpix[i] = det - alpha * (trace * trace);
31     }
32   }
33 }
```

Filter methods

The filter methods above use the ImageJ class `Convolver` (defined in package `ij.plugin.filter`) to perform the actual filter operation.

These static methods are defined in class `HarrisCornerDetector` (see App. D, p. 527) as follows:

```
34 static FloatProcessor convolve1h(FloatProcessor I,float[] h) {
35   Convolver conv = new Convolver();
36   conv.setNormalize(false);
37   conv.convolve(I, h, 1, h.length);
38   return I; }
39
40 static FloatProcessor convolve1v(FloatProcessor I,float[] h) {
41   Convolver conv = new Convolver();
42   conv.setNormalize(false);
43   conv.convolve(I, h, h.length, 1);
44   return I; }
45
46 static FloatProcessor convolve2(FloatProcessor I,float[] h) {
47   convolve1h(I,h);
48   convolve1v(I,h);
49   return I; }
```

8.3.2 Step 2: Selecting "Good" Corner Points

The result of the first stage of Alg. 8.1 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.

`Corner` class

Next we define a new class for representing single corner points $\boldsymbol{c}_i = \langle u_i, v_i, q_i \rangle$ and a constructor for creating new objects of the class `Corner` that uses the position (u_i, v_i) and the corner strength q_i:

```
50 public class Corner implements Comparable {
51   int u;   // x position
52   int v;   // y position
53   float q; // corner strength
54
55   Corner (int u, int v, float q) { //constructor method
56     this.u = u;
57     this.v = v;
58     this.q = q;
59   }
60 }
```

The class `Corner` implements the Java `Comparable` Interface, so that `Corner` objects can be compared with each other and thereby sorted into an ordered sequence.

Choosing a container

In Alg. 8.1, we made use of the mathematical notation for *lists* and *sets* to organize and manipulate the large collections of potential corner points generated at various stages. While these generic concepts are well-suited for describing the abstract algorithm, in order to implement it, we need to replace them with real Java constructs.

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. Since this would be a very inefficient use of memory, we will instead make use of the `Vector` class, which is one of the dynamic data structures conveniently included in Java's *Collections Framework* (package `java.util`; also see App. B.2.7).

A `Vector` is similar in use to an array but can automatically increase its capacity as needed.[3] Just as in an array, individual elements in a `Vector` can be accessed through their index, but since the class `Vector` implements the Java `List` interface, a suite of additional access methods are available. Consequently, we also use the generic `List` type to declare variables and return values wherever possible, such that the actual implementation (as `Vector` in this case) is only specified once where list objects are created.

The `collectCorners()` method

The method `collectCorners()` below 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.

First (in line 62), the variable `cornerList` (of the generic type `List`) is assigned a new `Vector` object with an initial capacity of 1000 objects.

[3] While a `Vector` will increase its capacity as needed, there is an underlying expense to consider. When instantiated using the default constructor, storage is allocated for n potential elements. Once n elements have been stored, there is no more space remaining, so the object dynamically creates more by first allocating space for roughly $2n$ elements and then copying the original n elements into this new space. Since this allocate-and-copy operation is expensive, if you have an expectation of the maximum number of elements that will be stored in the `Vector`, you should specify it using the convenience constructor `Vector(n)`, as demonstrated in line 62. Alternatively, we could have made use of the class `ArrayList`, which differs only slightly from the class `Vector`.

Then the image `Q` is traversed, and when a potential corner point is located, a new `Corner` object is instantiated and stored in the `cornerList` (line 72):

```
61 List<Corner> collectCorners(FloatProcessor Q, int border) {
62   List<Corner> cornerList = new Vector<Corner>(1000);
63   int w = Q.getWidth();
64   int h = Q.getHeight();
65   float[] Qpix = (float[]) Q.getPixels();
66   // traverse the Q-image and check for corners:
67   for (int v = border; v < h-border; v++){
68     for (int u = border; u < w-border; u++) {
69       float q = Qpix[v*w+u];
70       if (q > threshold && isLocalMax(crf,u,v)) {
71         Corner c = new Corner(u,v,q);
72         cornerList.add(c);
73       }
74     }
75   }
76   Collections.sort(cornerList);
77   return cornerList;
78 }
```

The Boolean method `isLocalMax(`Q, u, v`)` (defined in the class `HarrisCornerDetector`) determines if the 2D function Q at the position (u, v) is a local maximum (see the definition in App. D, p. 531). Finally, at line 76, the corner points in `cornerList` are sorted according to their strength by calling the method `sort()` (a static method defined in class `java.util.Collections`).

In order to sort the points in this way, a class—in this case `Corner`—must implement the Java `Comparable` interface and provide a suitable `compareTo()` method. Since we want to sort the corner points in descending order according to their q values, we define the `compareTo()` method of the class `Corner` as follows:

```
79 public int compareTo (Object obj) { // in class Corner
80   Corner c2 = (Corner) obj;
81   if (this.q > c2.q) return -1;
82   if (this.q < c2.q) return 1;
83   else return 0;
84 }
```

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. 8.1, lines 24–32). This process is outlined in Fig. 8.4 and implemented in the method `cleanupCorners()` below. The `Vector corners`, which was already sorted according to q, is now converted into an ordinary array (line 89) and then iterated through from beginning to end:

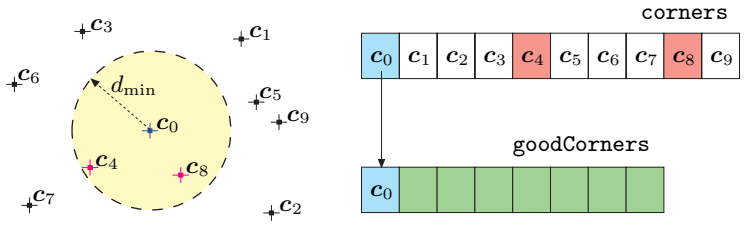


Fig. 8.4
Selecting the strongest corners within a given spatial distance. The original list of corners (`corners`) is sorted by "corner strength" in descending order; i. e., $\boldsymbol{c}_0$ is the strongest corner. First, corner $\boldsymbol{c}_0$ is added to a new list `goodCorners`, while the weaker corners $\boldsymbol{c}_4$ and $\boldsymbol{c}_8$ (which are both within distance $d_{\min}$ from $\boldsymbol{c}_0$) are removed from the original `corners` list. The following corners $\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots$ are treated similarly until no more elements remain in `corners`. None of the corners in the resulting list `goodCorners` is closer to another corner than $d_{\min}$.

```
85 List<Corner> cleanupCorners(List<Corner> corners) {
86   // corners is assumed to be sorted by descending q
87   double dmin2 = dmin*dmin; // d²min (dmin is an object variable)
88   Corner[] cornerArray = new Corner[corners.size()];
89   cornerArray = corners.toArray(cornerArray);
90
91   List<Corner> goodCorners =
92                  new Vector<Corner>(corners.size());
93
94   for (int i = 0; i < cornerArray.length; i++){
95     if (cornerArray[i] != null) {
96       // select the next "good" corner c1
97       Corner c1 = cornerArray[i];
98       goodCorners.add(c1);
99       // remove all remaining corners too close to c1
100      for (int j = i+1; j < cornerArray.length; j++) {
101        if (cornerArray[j] != null) {
102          Corner c2 = cornerArray[j];
103          if (c1.dist2(c2) < dmin2) //compare squared distances
104            cornerArray[j] = null;  //remove corner c2
105        }
106      }
107    }
108  }
109  return goodCorners;
110 }
```

At this point, weak corner points within the neighborhood of a stronger corner point, where the neighborhood is defined by the $d_{\min}$ radius, are deleted (line 104), and only those corner points that remain (that is, the strongest ones) are copied into the new list `goodCorners` (which is also implemented as a `Vector`).

The method call `c1.dist2(c2)` in line 103 computes the *squared* Euclidean distance $d^2(\boldsymbol{c}_1, \boldsymbol{c}_2) = (u_1 - u_2)^2 + (v_1 - v_2)^2$ between the corner points $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$. 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.

8.3.3 Displaying the Corner Points

In order to visualize the locations of the corner points finally selected, we now place markers at the corresponding positions in the original image. The method `showCornerPoints()` below (defined in the class `HarrisCornerDetector`) first creates a copy of the original image `ip` and increases, with the help of a lookup table, the overall brightness of the intensity range 128 to 255, and at the same time reduces the contrast by half (lines 114–118). Then the list `corners` is iterated through, and each `Corner` object "draws itself" onto the display image `ipResult` by calling its `draw()` method (line 121):

```
111 ImageProcessor showCornerPoints(ImageProcessor ip) {
112   ByteProcessor ipResult = (ByteProcessor) ip.duplicate();
113   // change background image contrast and brightness
114   int[] lookupTable = new int[256];
115   for (int i=0; i<256; i++){
116     lookupTable[i] = 128 + (i/2);
117   }
118   ipResult.applyTable(lookupTable);
119   // draw all corners:
120   for (Corner c: corners) {
121     c.draw(ipResult);
122   }
123   return ipResult;
124 }
```

The `draw()` method is defined in the class `Corner` and simply draws a fixed-size cross at the position of the corner point (u, v):

```
125 void draw(ByteProcessor ip){ // defined in class Corner
126   //draw this corner as a black cross
127   int paintvalue = 0;       // set draw value to black
128   int size = 2;             // set size of cross marker
129   ip.setValue(paintvalue);
130   ip.drawLine(u-size,v,u+size,v);
131   ip.drawLine(u,v-size,u,v+size);
132 }
```

8.3.4 Summary

Most of the implementation steps we have just described are initiated through calls from the method `findCorners()`:

```
133 void findCorners(){           // defined in class Corner
134   makeDerivatives();
135   makeCrf();     // compute corner response function (CRF)
136   corners = collectCorners(border);
137   corners = cleanupCorners(corners);
138 }
```

Since we have broken up the processing steps up into small meaningful methods, the actual `run()` method of the plugin `Find_Corners` is reduced to just a few lines. This method simply creates a new object of the class `HarrisCornerDetector`, calls its `findCorners()` method, and finally displays the results in a new window:

```
139 public void run(ImageProcessor ip) {
140   HarrisCornerDetector hcd = new HarrisCornerDetector(ip);
141   hcd.findCorners();
142   ImageProcessor result = hcd.showCornerPoints(ip);
143   ImagePlus win = new ImagePlus("Corners",result);
144   win.show();
145 }
```

As previously mentioned, the complete source code for this section can be found in App. D (pp. 525–532). Again, when writing this code, we focused 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.

8.4 Exercises

Exercise 8.1. Adapt the `draw()` method in the class `Corner` (p. 148) 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.

Exercise 8.2. 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 8.3. 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 8.4. 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.

9

Detecting Simple Curves

Chapter 7 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 (for example, 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 (false negatives) and so are missing from the map.[1] In general, edge maps contain many irrelevant structures, while at the same time many important structures are completely missing. The theme of this chapter is how, given a binary edge map, one can find relevant and possibly significant structures based on their forms.

9.1 Salient 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

[1] Typically thresholding is performed at a level that decreases false negatives at the expense of introducing false positives, the reasoning being that it is much simpler to remove false positives during higher-level processing than it is to, in essence, rediscover the missing edges eliminated during low-level processing.

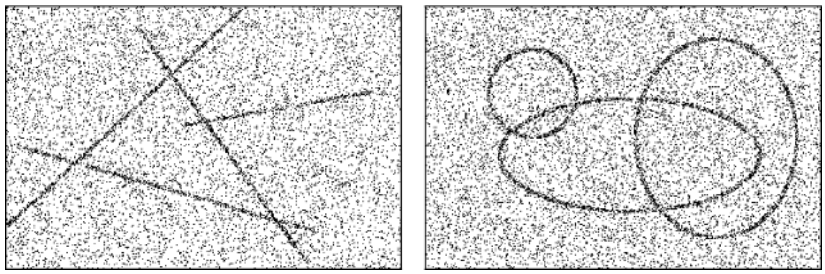


Fig. 9.1
The human visual system is capable of instantly recognizing prominent image structures even under difficult conditions.

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 [60] (see Sec. 11.2).

A completely different approach is to search for globally apparent structures that inherently express certain common shape features. As an example, Fig. 9.1 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 recognizes four lines or three circles in Fig. 9.1 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.

9.2 Hough Transform

The method from Paul Hough—originally published as a US Patent [47] 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 [27, 50]. 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. 9.2).

The Hough transform is perhaps most often used for detecting line segments in edge maps. A line segment in 2D can be described with two real-valued parameters using the classic slope-intercept form

$$y = kx + d, \tag{9.1}$$

where k is the slope and d the intercept—that is, the height at which the line would intercept the y axis (Fig. 9.3). A line segment that passes

Fig. 9.2
Simple geometrical forms such as sections of lines, circles, and ellipses are often found in man-made objects.

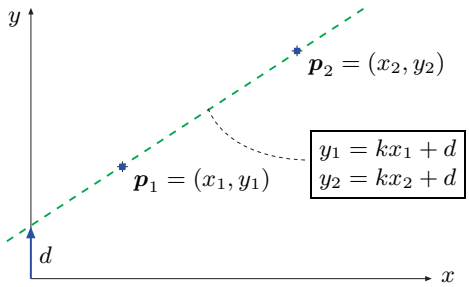


Fig. 9.3
Two points, $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$, lie on the same line when $y_1 = kx_1 + d$ and $y_2 = kx_2 + d$ for a particular pair of parameters k and d.

through two given edge points $\boldsymbol{p}_1 = (x_1, y_1)$ and $\boldsymbol{p}_2 = (x_2, y_2)$ must satisfy the conditions

$$y_1 = kx_1 + d \qquad \text{and} \qquad y_2 = kx_2 + d \tag{9.2}$$

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 makes this approach possible, generating such a large number of lines is very computationally expensive.

9.2.1 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 that runs through a point $\boldsymbol{p}_0 = (x_0, y_0)$ must satisfy the condition

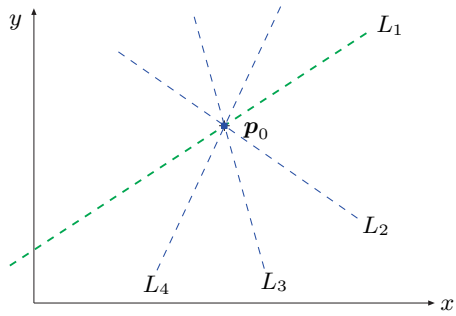


Fig. 9.4
Set of lines passing through an image point. For all possible lines L_j passing through the point $\boldsymbol{p}_0 = (x_0, y_0)$, the equation $y_0 = k_j x_0 + d_j$ holds for appropriate values of the parameters k_j, d_j.

$$L_j : y_0 = k_j x_0 + d_j \tag{9.3}$$

for all appropriate values of k_j, d_j. The possible solutions for k_j, d_j in Eqn. (9.3) correspond to the infinite set of lines passing through the given point $\boldsymbol{p}_0$ (Fig. 9.4). For a given k_j, the solution for d_j in Eqn. (9.3) is

$$d_j = -x_0 k_j + y_0, \tag{9.4}$$

which is another equation for a line, where now k_j, d_j are the *variables* and x_0, y_0 the constant *parameters* in the equation. The solution set $\{(k_j, d_j)\}$ of Eqn. (9.4) describes the parameters of all possible lines L_j passing through the image point $\boldsymbol{p}_0 = (x_0, y_0)$. For an *arbitrary* image point $\boldsymbol{p}_i = (x_i, y_i)$, Eqn. (9.4) describes the line

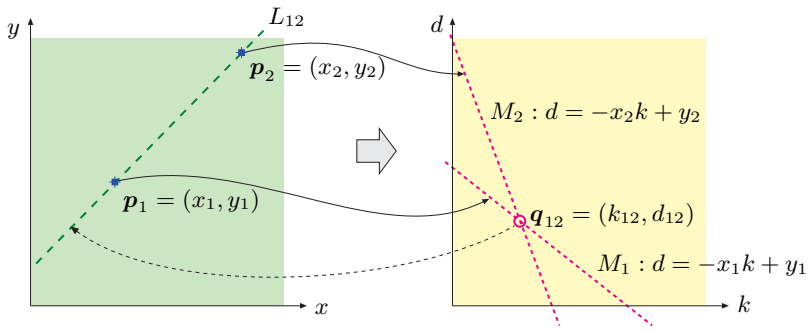
$$M_i : d = -x_i k + y_i \tag{9.5}$$

with the parameters $-x_i, y_i$ in the so-called *parameter* or *Hough* space spanning the coordinates k, d.

The relationship between image space and parameter space can be summarized as follows:

Image Space (x, y)		*Parameter Space* (k, d)	
Point	$\boldsymbol{p}_i = (x_i, y_i)$	$M_i : d = -x_i k + y_i$	Line
Line	$L_j : y = k_j x + d_j$	$\boldsymbol{q}_j = (k_j, d_j)$	Point

Each image point $\boldsymbol{p}_i$ 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. 9.5 illustrates how the lines M_1 and M_2 intersect at the position $\boldsymbol{q}_{12} = (k_{12}, d_{12})$ 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 $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$. 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 other words, when many lines intersect at a single point in the parameter space, then there are many points in the image space that lie on the line corresponding to that parameter space point.



(a) x/y image space (b) k/d parameter space

Fig. 9.5
Relationship between image space and parameter space. The parameter values for all possible lines passing through the image point $\boldsymbol{p}_i = (x_i, y_i)$ in image space (a) lie on a single line M_i in parameter space (b). This means that each point $\boldsymbol{q}_j = (k_j, d_j)$ 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 $\boldsymbol{q}_{12} = (k_{12}, d_{12})$ in parameter space indicates that a line L_{12} through the two points k_{12} and d_{12} exists in the image space.

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*.

9.2.2 Accumulator Array

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 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 an 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 through are incremented so that ultimately each cell accumulates the total number of lines that intersect at that cell (Fig. 9.6).

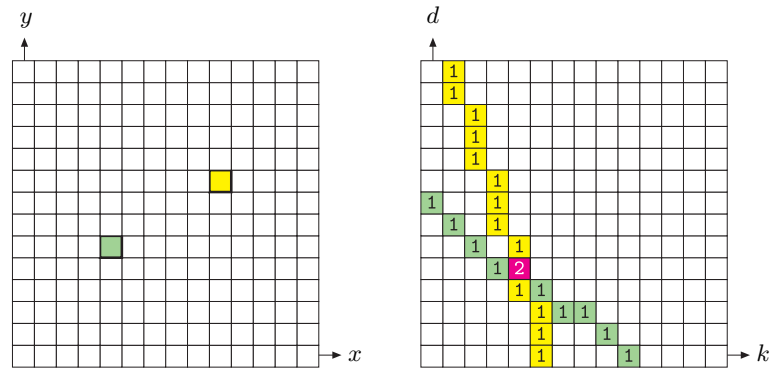
9.2.3 A Better Line Representation

The line representation in Eqn. (9.1) is not used in practice because for vertical lines $k = \infty$. A more practical representation is the so-called *Hessian normal form* (HNF, [15, p. 195]) for representing lines,

$$x \cdot \cos(\theta) + y \cdot \sin(\theta) = r, \tag{9.6}$$

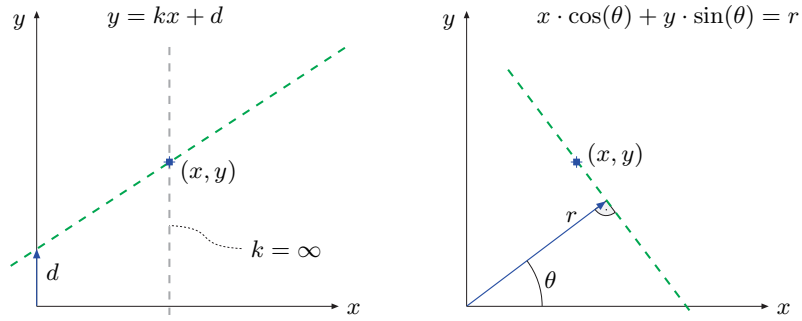
which does not exhibit such singularities and also provides a natural linear quantization for its parameters, the angle θ and the radius r (Fig. 9.7). With the HNF representation, the parameter space is defined by the coordinates θ, r, and the mapping from an image space point $\boldsymbol{p}_i = (x_i, y_i)$ to a parameter space point is defined as

$$r_{x_i,y_i}(\theta) = x_i \cdot \cos(\theta) + y_i \cdot \sin(\theta) \tag{9.7}$$



(a) Image Space (b) Accumulator Array

Fig. 9.6
Main idea of the Hough transform. The accumulator array 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).



(a) (b)

Fig. 9.7
Representation of lines in 2D. In the normal 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.

for angles in the range $0 \leq \theta < \pi$ (Fig. 9.8). If we use the center of the image as the reference point for the x/y image space, then it is possible to limit the range of the radius to half the diagonal of the image,

$$-r_{\max} \leq r_{x,y}(\theta) \leq r_{\max}, \quad \text{where} \quad r_{\max} = \tfrac{1}{2}\sqrt{M^2 + N^2}\,, \tag{9.8}$$

for an image of width M and height N.

9.3 Implementing the Hough Transform

The fundamental Hough algorithm using the HNF line representation (Eqn. (9.6)) is given in Alg. 9.1. Starting with a binary image $I(u, v)$ where the edge pixels have been assigned a value of 1, the first stage creates a two-dimensional accumulator array and then iterates over the image to fill it. In the second stage, the accumulator array is searched (FindMaxLines()) for maximum values, and a vector containing the parameters for the K strongest lines

$$\mathit{MaxLines} = \big(\langle\theta_1, r_1\rangle, \langle\theta_2, r_2\rangle, \ldots \langle\theta_K, r_K\rangle\big)$$

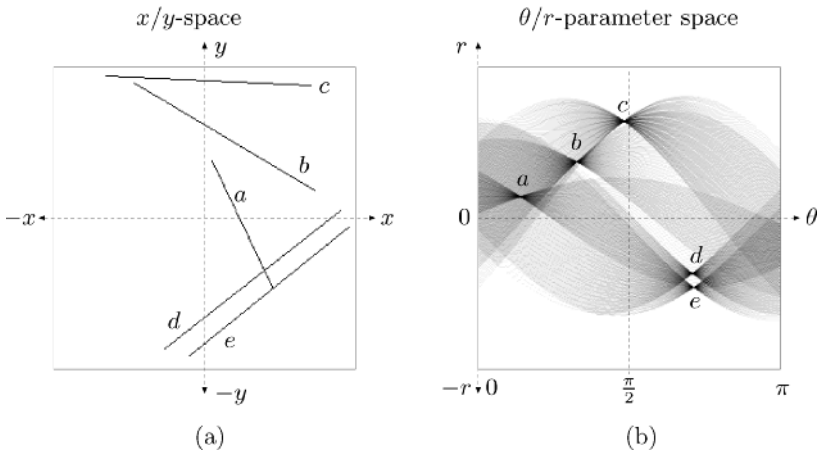


Fig. 9.8
Image space and parameter space using the HNF representation.

1: HoughLines(I)
Returns the list of parameters $\langle \theta_i, r_i \rangle$ corresponding to the strongest lines found in the binary image I.
2: Set up a two-dimensional array $Acc[\theta, r]$ of counters, initialize to 0.
3: Let (u_c, v_c) be the center coordinates of the image I
4: **for** all image coordinates (u, v) **do**
5: **if** $I(u, v)$ is an edge point **then**
Get coordinate relative to the image center (u_c, v_c):
6: $(x, y) \leftarrow (u - u_c, v - v_c)$
7: **for** $\theta_i = 0 \ldots \pi$ **do**
8: $r_i = x \cdot \cos(\theta_i) + y \cdot \sin(\theta_i)$
9: Increment $Acc[\theta_i, r_i]$
Return the list of parameter pairs $\langle \theta_j, r_j \rangle$ for K strongest lines:
10: *MaxLines* ← FindMaxLines(Acc, K)
11: **return** *MaxLines*.

Algorithm 9.1
Simple Hough algorithm for localizing lines. It returns a list containing the parameters $\langle \theta, r \rangle$ of the K strongest lines in a binary edge map.

is computed. The next sections explain these two stages in detail.

9.3.1 Filling the Accumulator Array

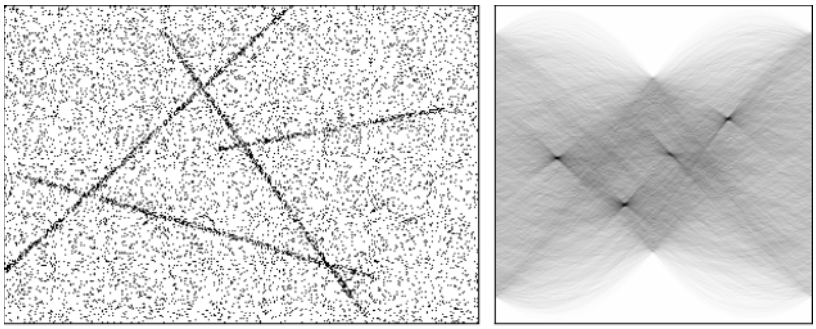
A direct implementation of the first phase of Alg. 9.1 is given in the Java class `LinearHT` Prog. 9.1. The accumulator array (`houghArray`) is defined as a two-dimensional `int` Array. The HT is computed from the original image `ip` by creating a new instance of the class `LinearHT`, e. g.,

```
LinearHT ht = new LinearHT(ip, 256, 256);
```

The binary image is passed as an `ImageProcessor` (`ip`), wherein any value greater than 0 is interpreted as an edge pixel. The other two parameters, `nAng` (256) and `nRad` (256), specify the number of discrete steps to use for the angle (N_θ steps for $\theta_i = 0$ to π) and the radius (N_r steps for $r_i = -r_{\max}$ to $r_{\max}$). The resulting increments for the angle and radius are thus

Program 9.1
Hough transform for localizing straight lines (Java implementation).

```
1  class LinearHT {
2    ImageProcessor ip; // reference to the original image I
3    int xCtr, yCtr;    // x/y-coordinates of image center (u_c, v_c)
4    int nAng;          // N_θ steps for the angle (θ = 0 ... π)
5    int nRad;          // N_r steps for the radius (r = −r_max ... r_max)
6    int cRad;          // center of radius axis (r = 0)
7    double dAng;       // increment of angle Δ_θ
8    double dRad;       // increment of radius Δ_r
9    int[][] houghArray; // Hough accumulator Acc[θ_i, r_i]
10
11   //constructor method:
12   LinearHT(ImageProcessor ip, int nAng, int nRad) {
13     this.ip = ip;
14     this.xCtr = ip.getWidth()/2;
15     this.yCtr = ip.getHeight()/2;
16     this.nAng = nAng;
17     this.dAng = Math.PI / nAng;
18     this.nRad = nRad;
19     this.cRad = nRad / 2;
20     double rMax = Math.sqrt(xCtr * xCtr + yCtr * yCtr);
21     this.dRad = (2.0 * rMax) / nRad;
22     this.houghArray = new int[nAng][nRad];
23     fillHoughAccumulator();
24   }
25
26   void fillHoughAccumulator() {
27     int h = ip.getHeight();
28     int w = ip.getWidth();
29     for (int v = 0; v < h; v++) {
30       for (int u = 0; u < w; u++) {
31         if (ip.get(u, v) > 0) {
32           doPixel(u, v);
33         }
34       }
35     }
36   }
37
38   void doPixel(int u, int v) {
39     int x = u - xCtr, y = v - yCtr;
40     for (int i = 0; i < nAng; i++) {
41       double theta = dAng * i;
42       int r = cRad + (int) Math.rint
43          ((x*Math.cos(theta) + y*Math.sin(theta)) / dRad);
44       if (r >= 0 && r < nRad) {
45         houghArray[i][r]++;
46       }
47     }
48   }
49
50 } // end of class LinearHT
```



(a) (b)

Fig. 9.9
Hough transform for lines. The dimensions of the original image (a) are 360 × 240 pixels, so the maximal radius (measured from the image center (u_c, v_c)) is $r_{\max} \approx 216$. For the parameter space (b), a step size of 256 is used for both the angle $\theta = 0 \ldots \pi$ (horizontal axis) and the radius $r = -r_{\max} \ldots r_{\max}$ (vertical axis). The four darkest spots in (b) mark the maximum values in the accumulator array, and their parameters correspond to the four lines in the original image. In (b), intensities have been inverted to improve legibility.

$$\Delta_\theta = \frac{\pi}{N_\theta} \qquad \text{and} \qquad \Delta_r = \frac{2 \cdot r_{\max}}{N_r}$$

(see lines 17 and 21 in Prog. 9.1, respectively). The output of this program for a very noisy edge image is given in Fig. 9.9.

9.3.2 Analyzing the Accumulator Array

The second phase is localizing the maximum values in the accumulator array $Acc[\theta, r]$. As can readily be seen in Fig. 9.9 (b), even in the case where the lines in the image are geometrically "straight", the parameter space curves associated with them do not intercept 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 due to the discrete coordinate grid used in the accumulator array. Since the maximum points are really maximum areas in the accumulator array, simply traversing the array and returning its K largest values is not sufficient. Since this is a critical step in the algorithm, we will examine two different approaches (Fig. 9.10) in the following.

Approach A: Thresholding

First the accumulator is thresholded to the value of t_a by setting all accumulator values $Acc[\theta, r] < t_a$ to 0. The resulting scattering of points, or point clouds, are first coalesced into regions (Fig. 9.10 (b)) using a technique such as a morphological *closing* operation (see Sec. 10.3.2). Next the remaining regions must be localized, for instance using the region-finding technique from Sec. 11.1, and then each region's centroid (see Sec. 11.4.3) 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.

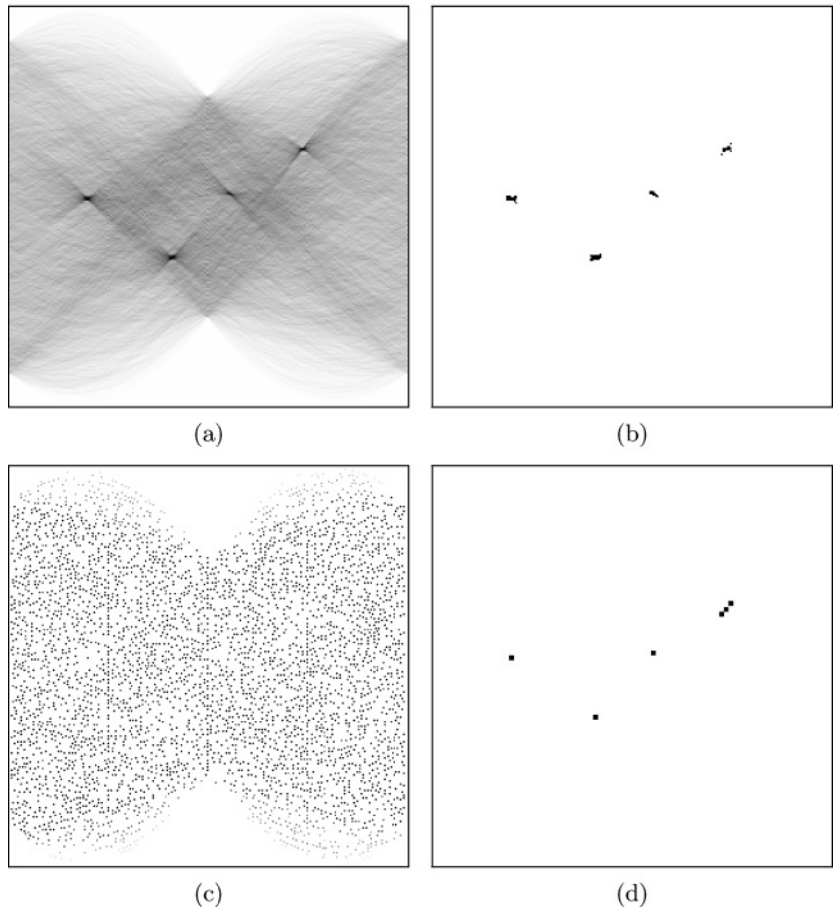


Fig. 9.10
Determining the local maximum values in the accumulator array. Original distribution of the values in the Hough accumulator (a). **Variant A:** *Threshold operation* using 50% of the maximum value (b). The remaining regions represent the four dominant lines in the image, and the coordinates of their centroids are a good approximation to the line parameters. **Variant B:** Using *nonmaximum suppression* results in a large number of local maxima (c) that must then be reduced using a threshold operation (d).

Approach B: Nonmaximum suppression

In this method, local maxima in the accumulator array are found by suppressing nonmaximal values.[2] This is carried out by determining for every cell in $Acc[\theta, r]$ 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. 9.10 (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. 9.9 (a) is shown in Fig. 9.10 (d).

[2] Nonmaximum suppression is also used in Sec. 8.2.3 for isolating corner points.

9.3.3 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 algorithm.

Modified accumulator updating

The purpose of the accumulator array is to find the intersections of two-dimensional curves. Due to the discrete nature of the image and accumulator coordinates, rounding errors usually cause the parameter curves for multiple image points on the same line not to intersect in a single accumulator cell. A common remedy is, for a given angle θ_i (Alg. 9.1), to increment not only the corresponding accumulator cell $Acc[\theta_i, r_i]$ but also the *neighboring* cells $Acc[\theta_i, r_{i-1}]$ and $Acc[\theta_i, r_{i+1}]$. This makes the Hough transform more tolerant against inaccurate point coordinates.

Bias problem

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. 9.11).

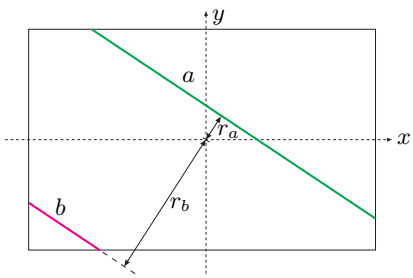


Fig. 9.11
Bias problem. When an image represents only a finite section of an object, then those lines nearer the center (smaller r values) will have higher values than those farther away (larger r values). As an example, the maximum value of the accumulator for line a will be higher than that of line b.

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 this inherent bias is to compute for each accumulator entry $Acc[\theta, r]$ the maximum number of image points $MaxHits[\theta, r]$ possible for a line with the parameters θ, r and then normalize

$$Acc'[\theta, r] \leftarrow \frac{Acc[\theta, r]}{MaxHits[\theta, r]} \tag{9.9}$$

for $MaxHits[\theta, r] > 0$. The normalization term $MaxHits[\theta, r]$ can be determined, for example, by computing the Hough transform of an image with the same dimensions in which all pixels are activated or 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 line and the 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 two additional coordinate pairs $(x_{\text{start}}, y_{\text{start}})$, $(x_{\text{end}}, y_{\text{end}})$, i. e.,

$$Acc[\theta, r] = \langle count, x_{\text{start}}, y_{\text{start}}, x_{\text{end}}, y_{\text{end}} \rangle.$$

Now the coordinates for the endpoints of every line can be stored while filling in the accumulator array so that by the end of the process each cell contains the two endpoints that lie farthest from each other on the line it represents. When finding the maximum values in the second stage, care should be taken so that the merged cell values contain the correct endpoints.

Line intersections

It may be useful in certain applications not to find the lines themselves but their intersections, e. g., 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 (i. e., as pairs $L_i = \langle \theta_i, r_i \rangle$). To compute the point of intersection $\boldsymbol{x}_0 = (x_0, y_0)^T$ for two lines

$$L_1 = \langle \theta_1, r_1 \rangle \qquad \text{and} \qquad L_2 = \langle \theta_2, r_2 \rangle,$$

we need to solve the system of linear equations

$$x_0 \cdot \cos(\theta_1) + y_0 \cdot \sin(\theta_1) = r_1, \tag{9.10}$$
$$x_0 \cdot \cos(\theta_2) + y_0 \cdot \sin(\theta_2) = r_2, \tag{9.11}$$

for the unknowns x_0, y_0. The solution is

$$\begin{aligned} \boldsymbol{x}_0 &= \frac{1}{\cos(\theta_1)\sin(\theta_2) - \cos(\theta_2)\sin(\theta_1)} \cdot \begin{bmatrix} r_1 \sin(\theta_2) - r_2 \sin(\theta_1) \\ r_2 \cos(\theta_1) - r_1 \cos(\theta_2) \end{bmatrix} \\ &= \frac{1}{\sin(\theta_2 - \theta_1)} \cdot \begin{bmatrix} r_1 \sin(\theta_2) - r_2 \sin(\theta_1) \\ r_2 \cos(\theta_1) - r_1 \cos(\theta_2) \end{bmatrix} \end{aligned} \tag{9.12}$$

for $\sin(\theta_2 - \theta_1) \neq 0$. Obviously $\boldsymbol{x}_0$ is undefined (no intersection point exists) if the lines L_1, L_2 are parallel to each other (i. e., if $\theta_1 \equiv \theta_2$).

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. 7.3), 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:

$$Acc[\theta, r] \leftarrow Acc[\theta, r] + E(u, v).$$

In this way, strong edge points will contribute more to the accumulated value than weak points.

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 required 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, for example, [58, p. 483]).

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 every $\frac{\pi}{256} \approx 0.7°$. While increasing the number of accumulators 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.

9.4 Hough Transform for Circles and Ellipses

9.4.1 Circles and Arcs

Since lines in 2D have two degrees of freedom, they could be completely specified using two real value parameters. In a similar fashion, representing a circle in 2D requires *three* parameters, for example

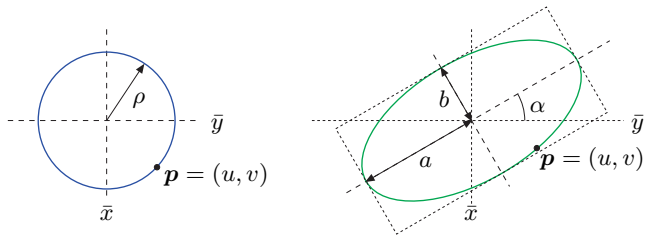


Fig. 9.12
Representation of circles and ellipses in 2D.

$$Circle = \langle \bar{x}, \bar{y}, \rho \rangle,$$

where $\bar{x}$, $\bar{y}$ are the coordinates of the center and ρ is the radius of the circle (Fig. 9.12). A point $\boldsymbol{p} = (u, v)$ lies on a circle when the relation

$$(u - \bar{x})^2 + (v - \bar{y})^2 = \rho^2 \tag{9.13}$$

holds. Therefore the Hough transform requires a three-dimensional parameter space $Acc[\bar{x}, \bar{y}, \rho]$ 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 $(\bar{x}, \bar{y}, \rho)$ that satisfies Eqn. (9.13) for a given image point $\boldsymbol{p} = (u, v)$? One solution is to apply a "brute force" method such as described in Alg. 9.2 that exhaustively tests each cell in the parameter space to see if the relation in Eqn. (9.13) holds.

Algorithm 9.2
Exhaustive Hough algorithm for localizing circles.

```
1: HoughCircles(I)
     Returns the list of parameters ⟨x̄_i, ȳ_i, ρ_i⟩ corresponding to the
     strongest circles found in the binary image I.
2:   Set up a three-dimensional array Acc[x̄, ȳ, ρ] and initialize to 0
3:   for all image coordinates (u, v) do
4:     if I(u, v) is an edge point then
5:       for all (x̄_i, ȳ_i, ρ_i) in the accumulator space do
6:         if (u−x̄_i)² + (v−ȳ_i)² = ρ_i² then
7:           Increment Acc[x̄_i, ȳ_i, ρ_i]
8:   MaxCircles ← FindMaxCircles(Acc)  ▷ a list of tuples ⟨x̄_j, ȳ_j, ρ_j⟩
9:   return MaxCircles.
```

If we examine Fig. 9.13, 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 three-dimensional parameter space for each image point $\boldsymbol{p} = (u, v)$. Instead we need only increase the cell values along the edge of the appropriate circle on each ρ 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 [13] is particularly well-suited.

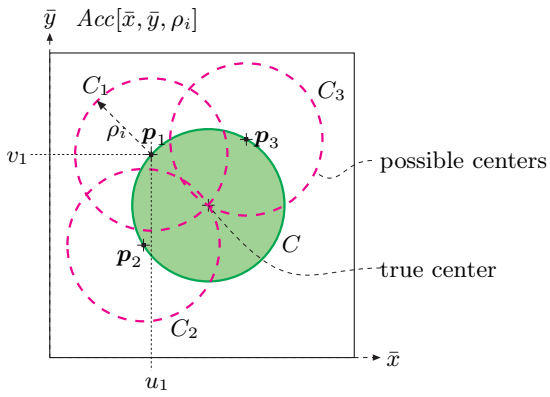
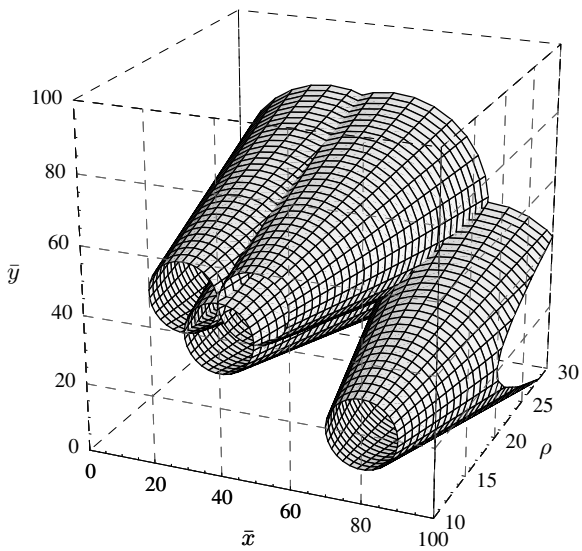


Fig. 9.13
Hough transform for circles. The illustration depicts a slice of the three-dimensional accumulator array $Acc[\bar{x},\bar{y},\rho]$ at a given circle radius $\rho = \rho_i$. The center points of all the circles running through a given image point $\boldsymbol{p}_1 = (u_1, v_1)$ form a circle C_1 with a radius of ρ_i centered around $\boldsymbol{p}_1$, just as the center points of the circles that pass through $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$ 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 ρ_i are traversed and their values in the accumulator array incremented. The cell in the accumulator array contains a value of three where the circles intersect at the true center of the image circle C.

Figure 9.14 shows the spatial structure of the three-dimensional parameter space for circles. For a given image point $\boldsymbol{p}_k = (u_k, v_k)$, at each plane along the ρ axis (for $\rho_i = \rho_{\min} \ldots \rho_{\max}$), a circle centered at (u_k, v_k) with the radius ρ_i is traversed, ultimately creating a three-dimensional 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. 9.3.3) 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.



3D parameter space:
$\bar{x}, \bar{y} = 0 \ldots 100$
$\rho = 10 \ldots 30$

Image points $\boldsymbol{p}_k$:
$\boldsymbol{p}_1 = (30, 50)$
$\boldsymbol{p}_2 = (50, 50)$
$\boldsymbol{p}_3 = (40, 40)$
$\boldsymbol{p}_4 = (80, 20)$

Fig. 9.14
Three-dimensional parameter space for circles. For each image point $\boldsymbol{p}_k = (u_k, v_k)$, the cells lying along a cone in the three-dimensional accumulator array $Acc[\bar{x},\bar{y},\rho]$ are incremented.

9.4.2 Ellipses

In a perspective image, most circular objects originating in our real, three-dimensional 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 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,

$$\mathit{Ellipse} = \langle \bar{x}, \bar{y}, r_a, r_b, \alpha \rangle,$$

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. 9.12).[3] In order to find ellipses of any size, position, and orientation using the Hough transform, a five-dimensional 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 ever dimension results in 2^{35} accumulator cells, and implementing these using 4-byte `int` values thus requires 2^{37} bytes (128 gigabytes) of memory.

An interesting alternative in this case is the *generalized Hough transform*, which in principle can be used for detecting any arbitrary two-dimensional shape [5, 50]. 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 four-dimensional space, which is smaller than that of the Hough method for ellipses described above.

9.5 Exercises

Exercise 9.1. Implement a version of the Hough transform for straight lines that incorporates the modified accumulator update, as suggested in Sec. 9.3.3. Analyze the extent to which the method improves the robustness with respect to inaccurate or noisy point positions.

Exercise 9.2. Implement a version of the Hough transform for finding lines that takes into account line endpoints as described in Sec. 9.3.3.

Exercise 9.3. Implement a *hierarchical* Hough transform for straight lines (see p. 167) capable of accurately determining line parameters.

[3] See Eqn. (11.33) on p. 232 for a parametric equation of this ellipse.

Exercise 9.4. Implement the Hough transform for finding circles and circular arcs with varying radii. Make use of a fast algorithm for generating circles, such as described in Sec. 9.4, in the accumulator array.

10

Morphological Filters

In the discussion of the median filter in Ch. 6 (Sec. 6.4.2), we noticed that this type of filter can somehow alter two-dimensional image structures. Figure 10.1 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.

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. 5.1.4) 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

Fig. 10.1
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).

are shown black and background pixels are shown 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 even color images, though these operations differ significantly from their binary counterparts.

10.1 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 for one 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. 10.2):

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 its outer border

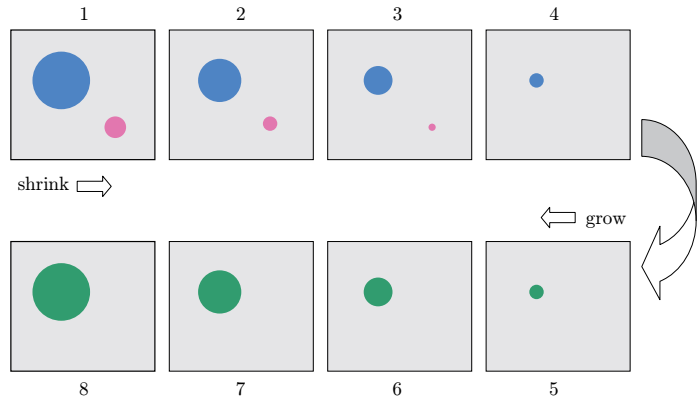


Fig. 10.2
Removing small image structures by stepwise shrinking and subsequent growing.

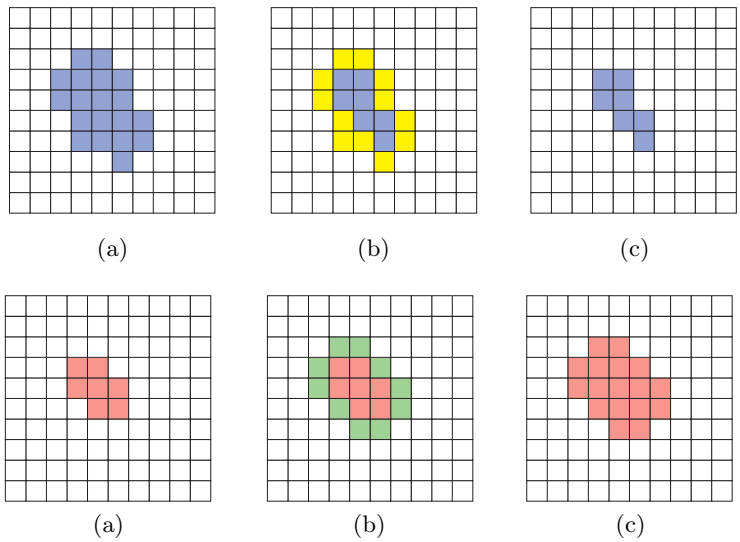


Fig. 10.3
"Shrinking" a foreground region by removing a layer of border pixels: original image (a), identified border pixels that are in direct contact with the background (b), and result after shrinking (c).

Fig. 10.4
"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).

against the background (Fig. 10.3). The other way around, "growing", adds a layer of pixels around the border of a foreground region (Fig. 10.4).

10.1.1 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. 10.5):

- **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.

10.2 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 above. They go well beyond removing or attaching single pixel layers and—in combination—can perform much more complex operations.

10.2.1 The Structuring Element

Similar to the coefficient matrix of a linear filter (see Sec. 6.2), the properties of a morphological filter are specified by elements in a matrix called

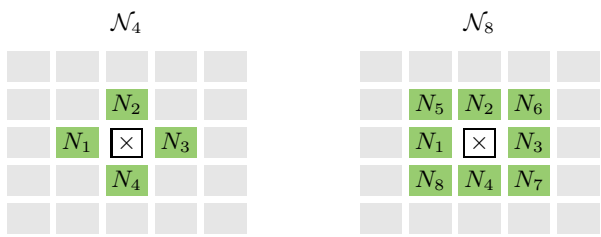


Fig. 10.5
Definitions of "neighborhood" on a rectangular pixel grid: *4-neighborhood* $\mathcal{N}_4 = \{N_1, \ldots N_4\}$ (left) and *8-neighborhood* $\mathcal{N}_8 = \mathcal{N}_4 \cup \{N_5, \ldots N_8\}$ (right).

a "structuring element". In binary morphology, the structuring element (like the image) 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. 10.6). Notice that the hot spot is not necessarily located at the center of the structuring element, nor must its value be 1.

Fig. 10.6
Binary structuring element (example). 1–elements are marked with •; 0–cells are empty.

10.2.2 Point Sets

For the formal specification of morphological operations, it is helpful to describe binary images as *sets* of two-dimensional coordinate points. For a binary image $I(u, v) \in \{0, 1\}$, the corresponding point set $\mathcal{Q}_I$ consists of the coordinate pairs $\boldsymbol{p} = (u, v)$ of all foreground pixels,

$$\mathcal{Q}_I = \{\boldsymbol{p} \mid I(\boldsymbol{p}) = 1\}. \tag{10.1}$$

Of course, as shown in Fig. 10.7, not only a binary image I but also a structuring element H can be described as a point set.

Given a description as point sets, fundamental operations on binary images can also be expressed as simple set operations. For example, *inverting* a binary image $I \to \bar{I}$ (i. e., exchanging foreground and background) is equivalent to building the *complementary* set

$$\mathcal{Q}_{\bar{I}} = \bar{\mathcal{Q}}_I = \{\boldsymbol{p} \in \mathbb{Z}^2 \mid \boldsymbol{p} \notin \mathcal{Q}_I\}. \tag{10.2}$$

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,

$$\mathcal{Q}_{I_1 \vee I_2} = \mathcal{Q}_{I_1} \cup \mathcal{Q}_{I_2}. \tag{10.3}$$

$I \equiv \mathcal{Q}_I = \{(1,1),(2,1),(2,2)\}$ $\quad$ $H \equiv \mathcal{Q}_H = \{(0,0),(1,0)\}$

Fig. 10.7
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).

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. (10.2) or $I_1 \cup I_2$ instead of $\mathcal{Q}_{I_1} \cup \mathcal{Q}_{I_2}$ in Eqn. (10.3). The meaning should always be clear in the given context.

Thus, *translating* (shifting) the binary image I by some coordinate vector $\boldsymbol{d}$ creates a new image with the content $I_{\boldsymbol{d}}(\boldsymbol{p}+\boldsymbol{d}) = I(\boldsymbol{p})$, which corresponds to all coordinates in the point set $\mathcal{Q}_I$ being shifted by $\boldsymbol{d}$; i. e.,

$$I_{\boldsymbol{d}} \equiv \big\{(\boldsymbol{p}+\boldsymbol{d}) \mid \boldsymbol{p} \in I\big\}. \tag{10.4}$$

In some cases, it is necessary to *reflect* (mirror) a binary image or point set about its origin, which we denote as

$$H^* \equiv \{-\boldsymbol{p} \mid \boldsymbol{p} \in H\}. \tag{10.5}$$

10.2.3 Dilation

A *dilation* is the morphological operation that corresponds to our intuitive concept of "growing" as discussed above. As a set operation, it is defined as

$$I \oplus H \equiv \big\{(\boldsymbol{p}+\boldsymbol{q}) \mid \text{for some } \boldsymbol{p} \in I \text{ and } \boldsymbol{q} \in H\big\}. \tag{10.6}$$

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. 10.8.

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,[1] this is

$$I \oplus H \equiv \bigcup_{\boldsymbol{p} \in I} H_{\boldsymbol{p}} = \bigcup_{\boldsymbol{q} \in H} I_{\boldsymbol{q}}, \tag{10.7}$$

with $H_{\boldsymbol{p}}, I_{\boldsymbol{q}}$ denoting the sets H, I shifted by $\boldsymbol{p}$ and $\boldsymbol{q}$, respectively Eqn. (10.4).

[1] Also see Sec. 1.2.

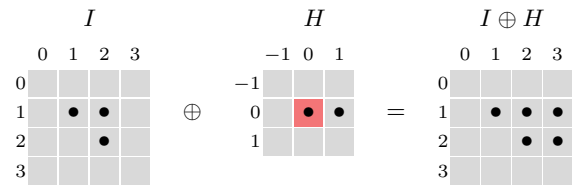


Fig. 10.8
Dilation example. The binary image I is subject to dilation with the structuring element H. In the result $I \oplus H$, the structuring element H is replicated at every foreground pixel of the original image I.

$$I \equiv \{(1,1),(2,1),(2,2)\},\ H \equiv \{(\mathbf{0},\mathbf{0}),(\mathbf{1},\mathbf{0})\}$$

$$\begin{aligned} I \oplus H \ \equiv\ \{\ &(1,1)+(\mathbf{0},\mathbf{0}),\ (1,1)+(\mathbf{1},\mathbf{0}),\\ &(2,1)+(\mathbf{0},\mathbf{0}),\ (2,1)+(\mathbf{1},\mathbf{0}),\\ &(2,2)+(\mathbf{0},\mathbf{0}),\ (2,2)+(\mathbf{1},\mathbf{0})\ \} \end{aligned}$$

10.2.4 Erosion

The quasi-inverse of dilation is the *erosion* operation, again defined in set notation as

$$I \ominus H \equiv \big\{\boldsymbol{p} \in \mathbb{Z}^2 \mid (\boldsymbol{p}+\boldsymbol{q}) \in I,\ \text{for every } \boldsymbol{q} \in H\big\}. \tag{10.8}$$

This definition may appear quite cryptic but is simply explained as follows. A position $\boldsymbol{p}$ is contained in the result $I \ominus H$ if (and only if) the structuring element H—when placed at this position $\boldsymbol{p}$—is *fully contained* in the foreground pixels of the original image; i. e., if $H_{\boldsymbol{p}}$ is a subset of I. Equivalent to Eqn. (10.8), we could thus define erosion as

$$I \ominus H \equiv \{\boldsymbol{p} \in \mathbb{Z}^2 \mid H_{\boldsymbol{p}} \subseteq I\}. \tag{10.9}$$

Figure 10.9 shows a simple example for binary erosion.

10.2.5 Properties of Dilation and Erosion

The dilation operation is *commutative*,

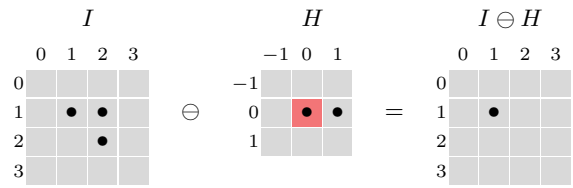
$$I \oplus H = H \oplus I\,, \tag{10.10}$$

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*,

$$(I_1 \oplus I_2) \oplus I_3 = I_1 \oplus (I_2 \oplus I_3), \tag{10.11}$$

and therefore the ordering of multiple dilations is not relevant. This also means—analogous to linear filters (cf. Eqn. (6.21))—that a dilation with a large structuring element of the form $H_{\text{big}} = H_1 \oplus H_2 \oplus \ldots \oplus H_K$ can be efficiently implemented as a sequence of multiple dilations with smaller structuring elements by

$$I \oplus H_{\text{big}} \ =\ (\ldots((I \oplus H_1) \oplus H_2) \oplus \ldots \oplus H_K). \tag{10.12}$$



$$I \equiv \{(1,1),(2,1),(2,2)\},\ H \equiv \{(\mathbf{0},\mathbf{0}),(\mathbf{1},\mathbf{0})\}$$

$$I \ominus H \equiv \{\,(1,1)\,\} \text{ because}$$
$$(1,1)+(\mathbf{0},\mathbf{0}) = (1,1) \in I \quad \textbf{and} \quad (1,1)+(\mathbf{1},\mathbf{0}) = (2,1) \in I$$

Fig. 10.9
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 $\boldsymbol{p} = (1,1)$. Thus the resulting point set contains only the single coordinate $(1,1)$.

There is also a *neutral!element* δ for dilation, similar to the Dirac function for the linear convolution (see Sec. 6.3.4),

$$I \oplus \delta \;=\; \delta \oplus I \;=\; I, \qquad \text{with } \delta \equiv \big\{(0,0)\big\}. \tag{10.13}$$

The *erosion* operation is, in contrast to dilation (but similar to arithmetic subtraction), *not* commutative; i. e.,

$$I \ominus H \neq H \ominus I \tag{10.14}$$

in general. However, if erosion and dilation are combined, then—again in analogy with arithmetic subtraction and addition—the following chain rule holds:

$$(I_1 \ominus I_2) \ominus I_3 \;=\; I_1 \ominus (I_2 \oplus I_3). \tag{10.15}$$

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,

$$I \oplus H \;\equiv\; \overline{(\bar{I} \ominus H^*)}, \tag{10.16}$$

where H^* denotes the *reflection* of H (Eqn. (10.5)). This works similarly the other way, too, namely

$$I \ominus H \;\equiv\; \overline{(\bar{I} \oplus H^*)}, \tag{10.17}$$

effectively eroding the foreground by dilating the background with the mirrored structuring element, as illustrated by the example in Fig. 10.10 (see [38, pp. 521–524] for a proof).

Equation (10.17) 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 10.1 gives a simple algorithmic description of dilation and erosion based on the relationships above. The corresponding Java implementation is shown later, in Sec. 10.5.2.

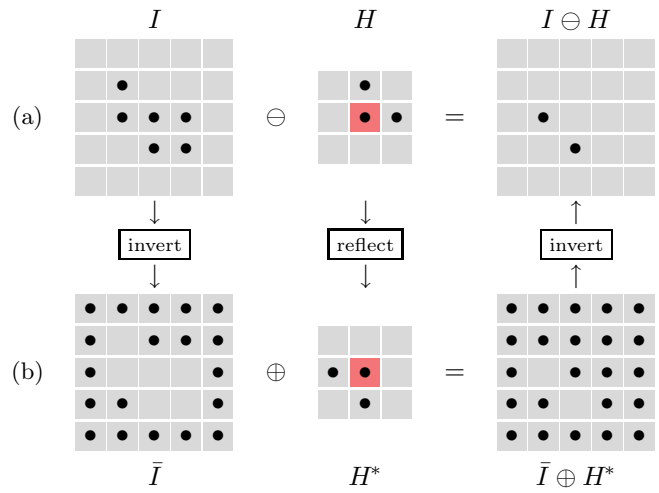


Fig. 10.10
Implementing erosion via dilation. The binary erosion of the foreground $I \ominus H$ (a) can be implemented by dilating the inverted (background) image $\bar{I}$ with the reflected structuring element H^* and subsequently inverting the result again (b).

Algorithm 10.1
Binary dilation and erosion. Procedure DILATE() implements the binary dilation as suggested by Eqn. (10.7). 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. (10.17).

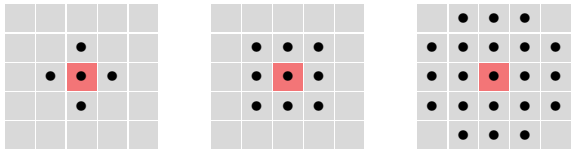
```
1:  DILATE (I, H)
        I: binary image of size w × h
        H: binary structuring element defined over region R_H
        Returns the dilated image I' = I ⊕ H
2:      I' ← new binary image of size w × h
3:      I'(u, v) ← 0, for all (u, v)                          ▷ I' ← ∅
4:      for all (i, j) ∈ R_H do                               ▷ (i, j) = q
5:          if H(i, j) = 1 then                               ▷ q ∈ H
6:              MERGE THE SHIFTED I_q WITH I':                ▷ I' ← I' ∪ I_q
7:              for u ← 0 ... (w−1) do
8:                  for v ← 0 ... (h−1) do                    ▷ (u, v) = p
9:                      if I(u, v) = 1 then                   ▷ p ∈ I
10:                         I'(u+i, v+j) ← 1                  ▷ I' ← I' ∪ (p+q)
11:     return I'.

12: ERODE (I, H)
13:     Ī ← INVERT(I)                                         ▷ Ī ← ¬I
14:     H* ← REFLECT(H)
15:     return INVERT(DILATE(Ī, H*)).                         ▷ I ⊕ H = ¬(Ī ⊕ H*)
```

10.2.6 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. 10.11.

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



(a) (b) (c)

Fig. 10.11
Typical small structuring elements: 4-neighborhood (a), 8-neighborhood (b), and "small disk" (c).

layers of the same thickness. Figure 10.13 shows the results of dilation and erosion with disk-shaped structuring elements of different diameters applied to the original image in Fig. 10.12. Dilation and erosion results for various other structuring elements are shown in Fig. 10.14.

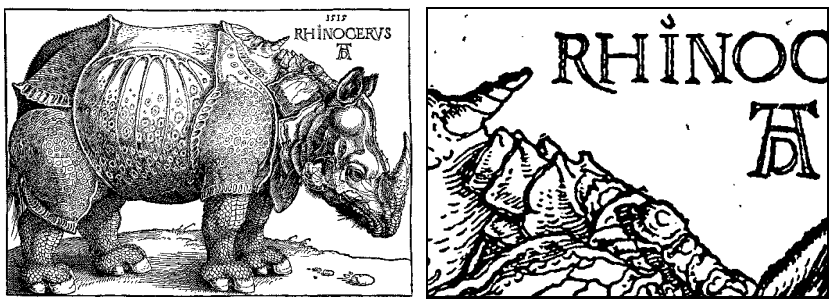


Fig. 10.12
Original binary image and the section used in the following examples (illustration by Albrecht Dürer, 1515).

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. 6.3.3), it is generally not possible to compose an isotropic 2D structuring element H° from one-dimensional structuring elements H_x and H_y since the dilation $H_x \oplus H_y$ always results in a rectangular (i. e., nonisotropic) structure. A remedy for approximating large disk-shaped filters is to alternately apply smaller disk-shaped operators of different shapes, as illustrated in Fig. 10.15. The resulting filter is generally not fully isotropic but can be implemented efficiently as a sequence of small filters.

10.2.7 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,

$$I' = I \ominus H_n,$$

using the 4- or 8-neighborhood (Fig. 10.11) as the structuring element H_n. To extract the actual boundary pixels B, we take the intersection of the original image I and the inverted result $\bar{I}'$; that is,

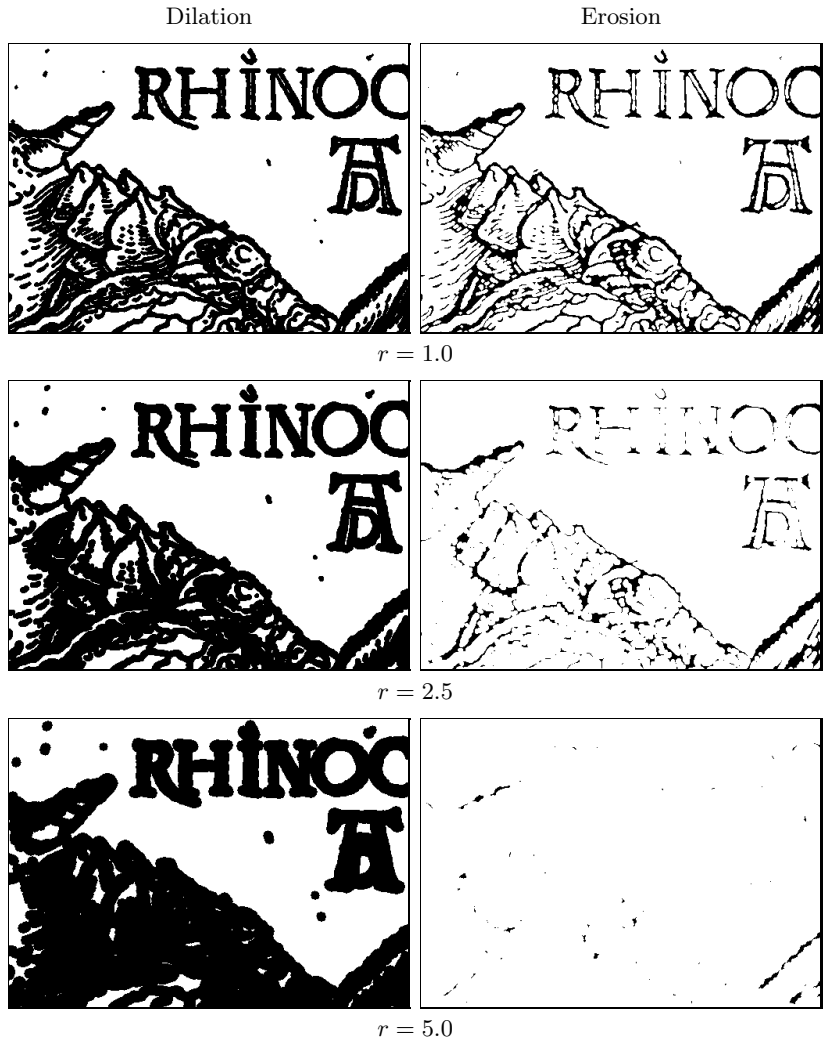


Fig. 10.13
Results of binary dilation and erosion with disk-shaped structuring elements. The radius of the disk (r) is 1.0 (top), 2.5 (center), or 5.0 (bottom).

$$B = I \cap \overline{I'} = I \cap \overline{(I \ominus H_n)}. \tag{10.18}$$

Notice that using the 4-neighborhood as the structuring element H_n produces "8-connected" contours and vice versa [58, p. 504].

The process of boundary extraction is illustrated on a simple example in Fig. 10.16. As can be observed in this figure, the result B 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

$$B(u,v) = \mathrm{XOR}\big(I(u,v), I'(u,v)\big) \quad \text{for all } (u,v). \tag{10.19}$$

Figure 10.17 shows a more complex example for extracting the boundary pixels from a real image.

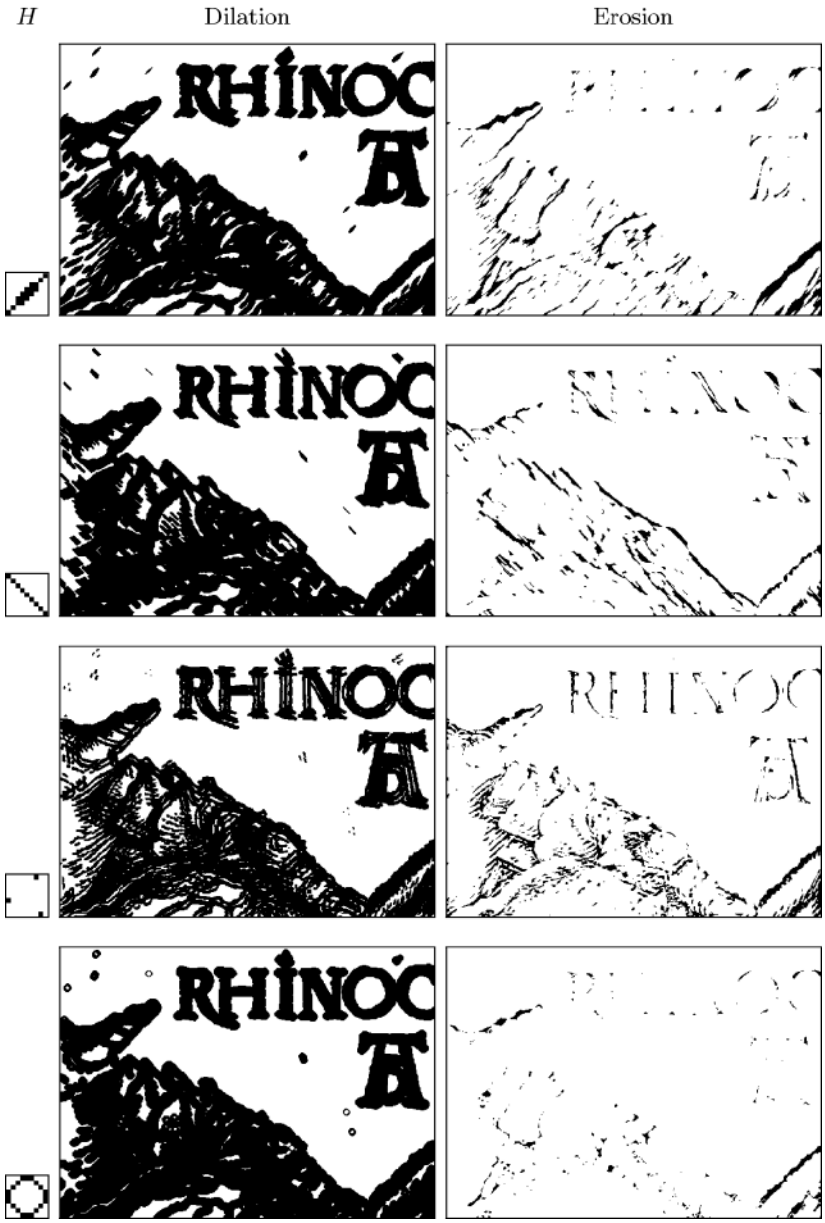


Fig. 10.14
Examples of binary dilation and erosion with various free-form structuring elements. The structuring elements H are shown in the left column (enlarged). Notice that the dilation expands every isolated foreground point to the shape of the structuring element, analogous to the *impulse response* of a linear filter. Under erosion, only those elements where the structuring element is fully contained in the original image survive.

10.3 Composite 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.

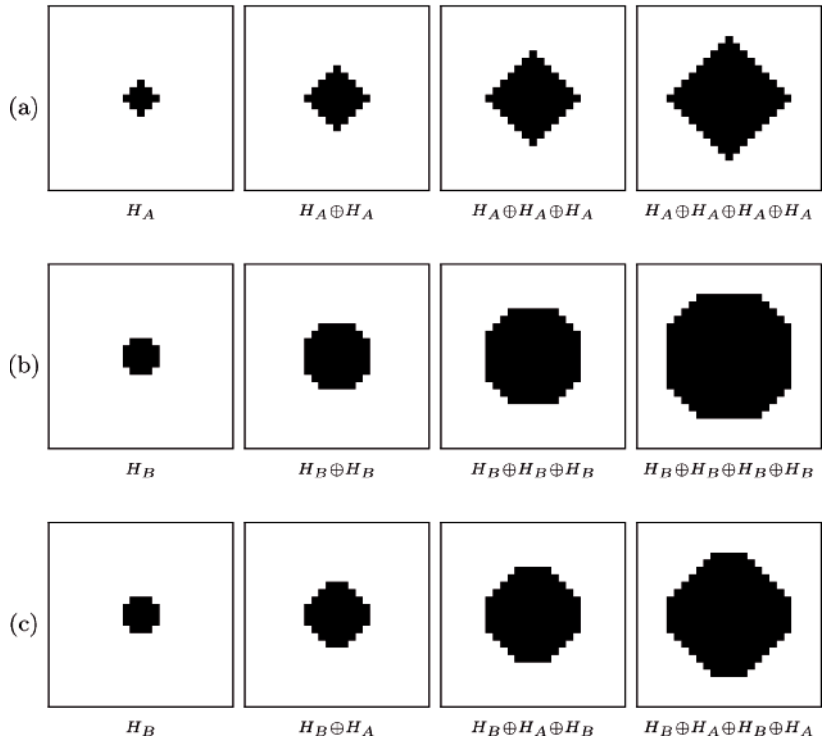


Fig. 10.15
Composition of large filters by repeated application of smaller filters: repeated application of the structuring element H_A (a) and structuring element H_B (b); alternating application of H_B and H_A (c).

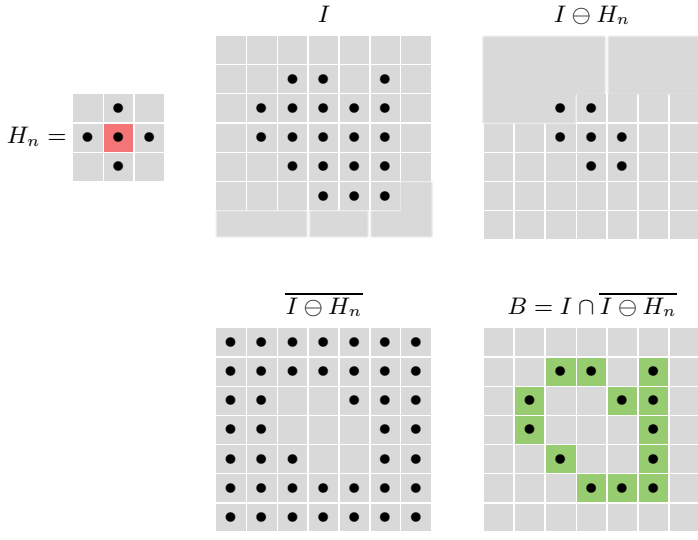
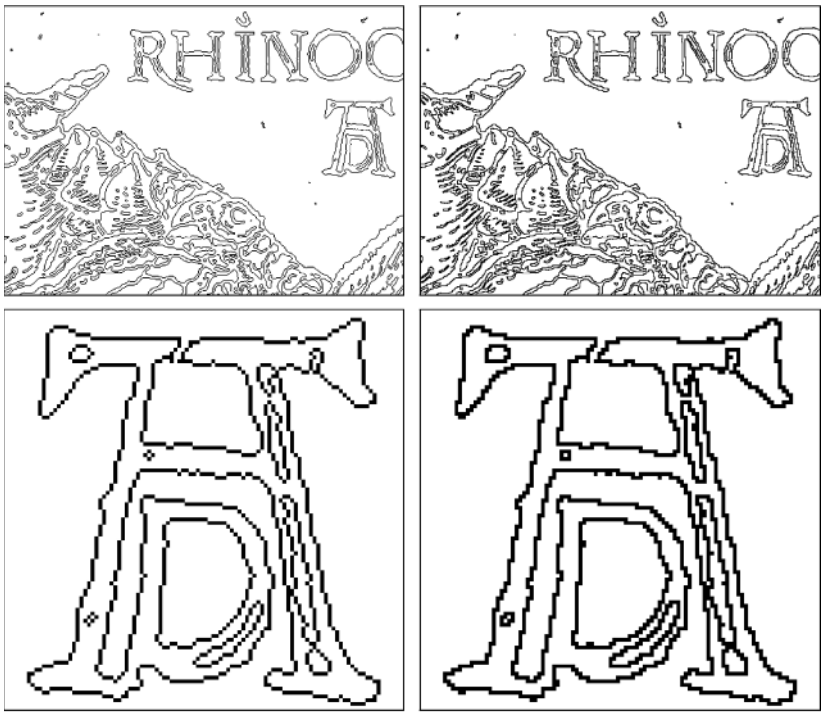


Fig. 10.16
Outline example using a 4-neighborhood structuring element H_n. The image I is first eroded ($I \ominus H_n$) and subsequently inverted ($\overline{I \ominus H_n}$). The boundary pixels are finally obtained as the intersection $I \cap \overline{I \ominus H_n}$.



(a) (b)

Fig. 10.17
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).

10.3.1 Opening

A binary opening $I \circ H$ denotes an erosion followed by a dilation with the *same* structuring element H,

$$I \circ H = (I \ominus H) \oplus H. \tag{10.20}$$

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. 10.18). This process of shrinking and subsequent growing corresponds to the idea for eliminating small structures that we had initially sketched in Sec. 10.1.

10.3.2 Closing

When the sequence of erosion and dilation is reversed, the resulting operation is called a closing and denoted $I \bullet H$,

$$I \bullet H = (I \oplus H) \ominus H. \tag{10.21}$$

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. 10.18.

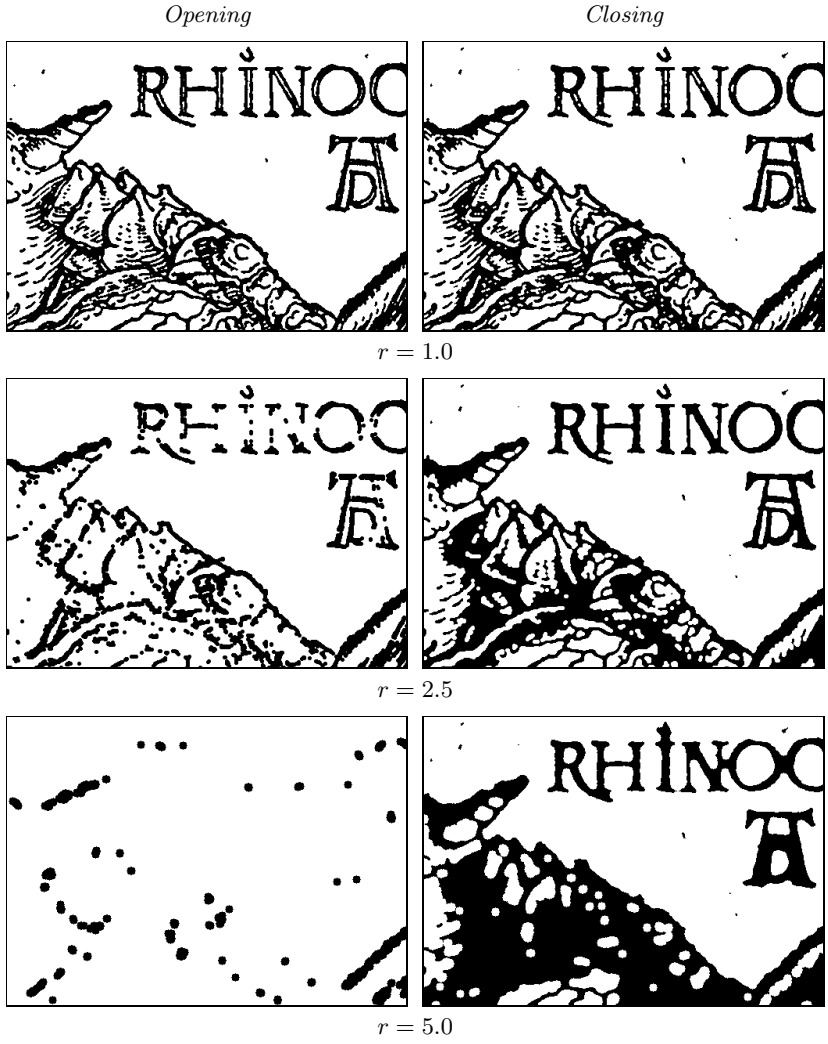


Fig. 10.18
Binary opening and closing with disk-shaped structuring elements. The radius r of the structuring element H is 1.0 (top), 2.5 (center), or 5.0 (bottom).

10.3.3 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; i. e.,

$$\begin{aligned} I \circ H &= (I \circ H) \circ H = ((I \circ H) \circ H) \circ H = \dots , \\ I \bullet H &= (I \bullet H) \bullet H = ((I \bullet H) \bullet H) \bullet H = \dots . \end{aligned} \tag{10.22}$$

Also, opening and closing are "duals" in the sense that opening the foreground is equivalent to closing the background and vice versa; i. e.,

$$I \circ H = \overline{(\bar{I} \bullet H)} \quad \text{and} \quad I \bullet H = \overline{(\bar{I} \circ H)}. \tag{10.23}$$

10.4 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).[2] In the case of color images, the grayscale operations are usually applied individually to each color channel.

10.4.1 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,

$$H(i,j) \in \mathbb{R}\,, \qquad \text{for } (i,j) \in \mathbb{Z}^2.$$

The values in H may be negative or zero. Notice, however, that in contrast to linear convolution (Sec. 6.3.1), zero elements in grayscale morphology generally *do* contribute to the result.[3] 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

$$\begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 1 & 2 & 1 \\ \hline 0 & 1 & 0 \\ \hline \end{array} \quad \neq \quad \begin{array}{|c|c|c|}\hline & 1 & \\ \hline 1 & 2 & 1 \\ \hline & 1 & \\ \hline \end{array} \quad . \tag{10.24}$$

10.4.2 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,

$$(I \oplus H)(u,v) = \max_{(i,j)\in H} \big\{I(u+i, v+j) + H(i,j)\big\}. \tag{10.25}$$

Similarly, the result of grayscale *erosion* is the minimum of the differences,

$$(I \ominus H)(u,v) = \min_{(i,j)\in H} \big\{I(u+i, v+j) - H(i,j)\big\}. \tag{10.26}$$

Figures 10.19 and 10.20 demonstrate the basic process of grayscale dilation and erosion, respectively, on a simple example. In general, either operation may produce *negative* results that must be considered if the

[2] ImageJ provides a single implementation of morphological operations that handles both binary and grayscale images (see Sec. 10.5.5).

[3] While a zero coefficient in a linear convolution matrix simply means that the corresponding image pixel is ignored.

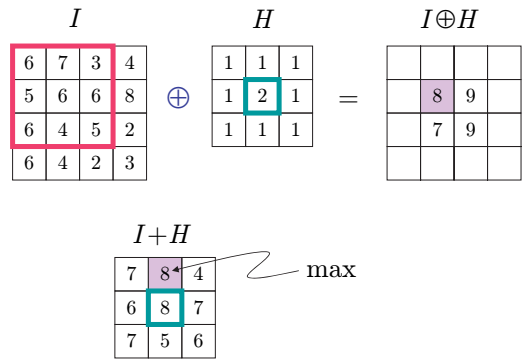


Fig. 10.19
Grayscale dilation $I \oplus H$. The 3×3 pixel structuring element H is placed on the image I in the upper left position. Each value of H is added to the corresponding element of I; the intermediate result $(I + H)$ for this particular position is shown below. Its maximum value $8 = 7 + 1$ 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.

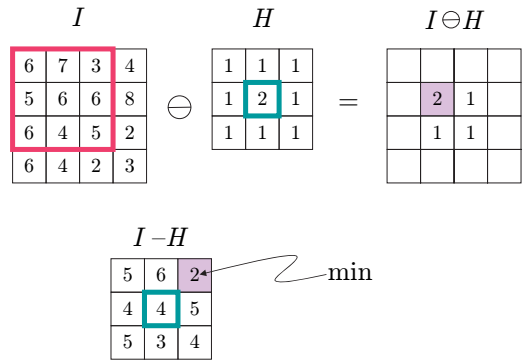


Fig. 10.20
Grayscale erosion $I \ominus H$. The 3×3 pixel structuring element H is placed on the image I in the upper left position. Each value of H is subtracted from the corresponding element of I; the intermediate result $(I - H)$ for this particular position is shown below. Its minimum value $3 - 1 = 2$ 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.

range of pixel values is restricted; for example, by clamping the results (see Sec. 5.1.2). Some examples of grayscale dilation and erosion on natural images using disk-shaped structuring elements of various sizes are shown in Fig. 10.21. Figure 10.22 demonstrates the same operations with some freely designed structuring elements.

10.4.3 Grayscale Opening and Closing

Opening and closing on grayscale images are defined, identical to the binary case (Eqns. (10.20) and (10.21)), as operations composed of dilation and erosion with the same structuring element. Some examples are shown in Fig. 10.23 for disk-shaped structuring elements and in Fig. 10.24 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.

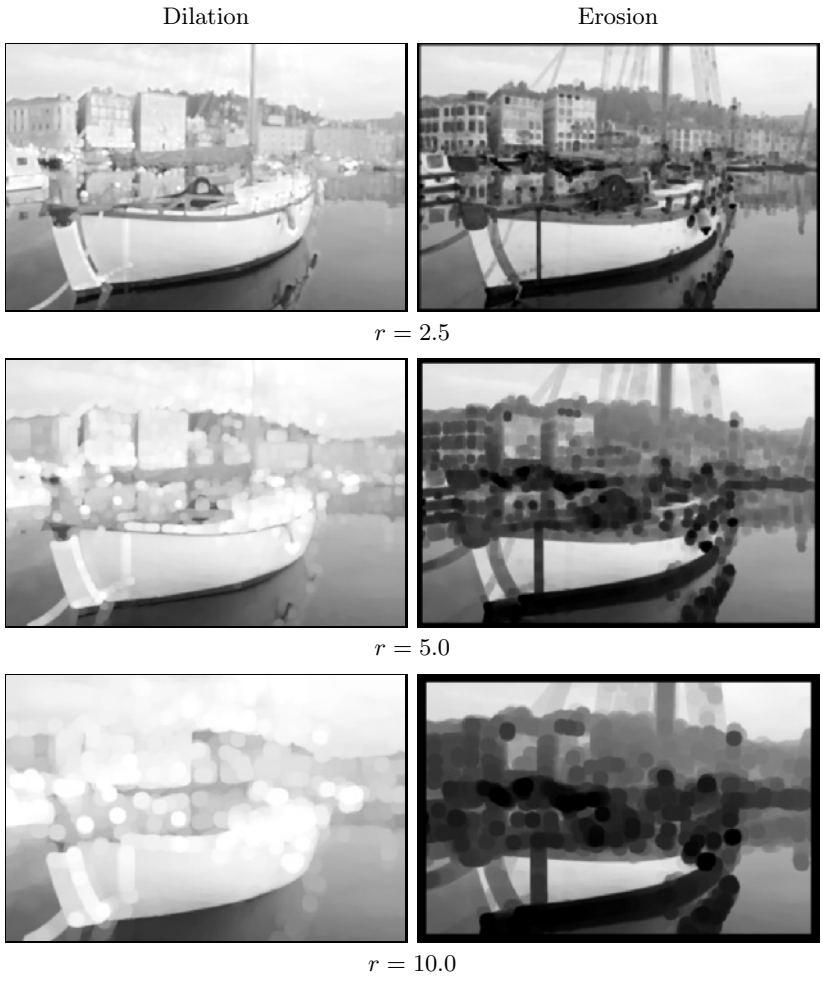


Fig. 10.21
Grayscale dilation and erosion with disk-shaped structuring elements. The radius r of the structuring element is 2.5 (top), 5.0 (center), or 10.0 (bottom).

10.5 Implementing Morphological Filters

10.5.1 Binary Images in ImageJ

In ImageJ, binary images contain 8 bits per pixel, the same as ordinary grayscale images.[4] A zero intensity value is interpreted as a binary 0, and any value greater than zero is considered a binary 1. Usually the intensity values 0 and 255 are used to represent the binaries 0 and 1, respectively, in which case the background pixels are displayed black and the foreground pixels are white by default. If an inverted display (black foreground) is desired, this can be easily accomplished by inverting the

[4] ImageJ does not provide a special (1-bit) data format for binary images. The class `BinaryProcessor` keeps image data as byte (8-bit) arrays, as does `ByteProcessor` for grayscale images.

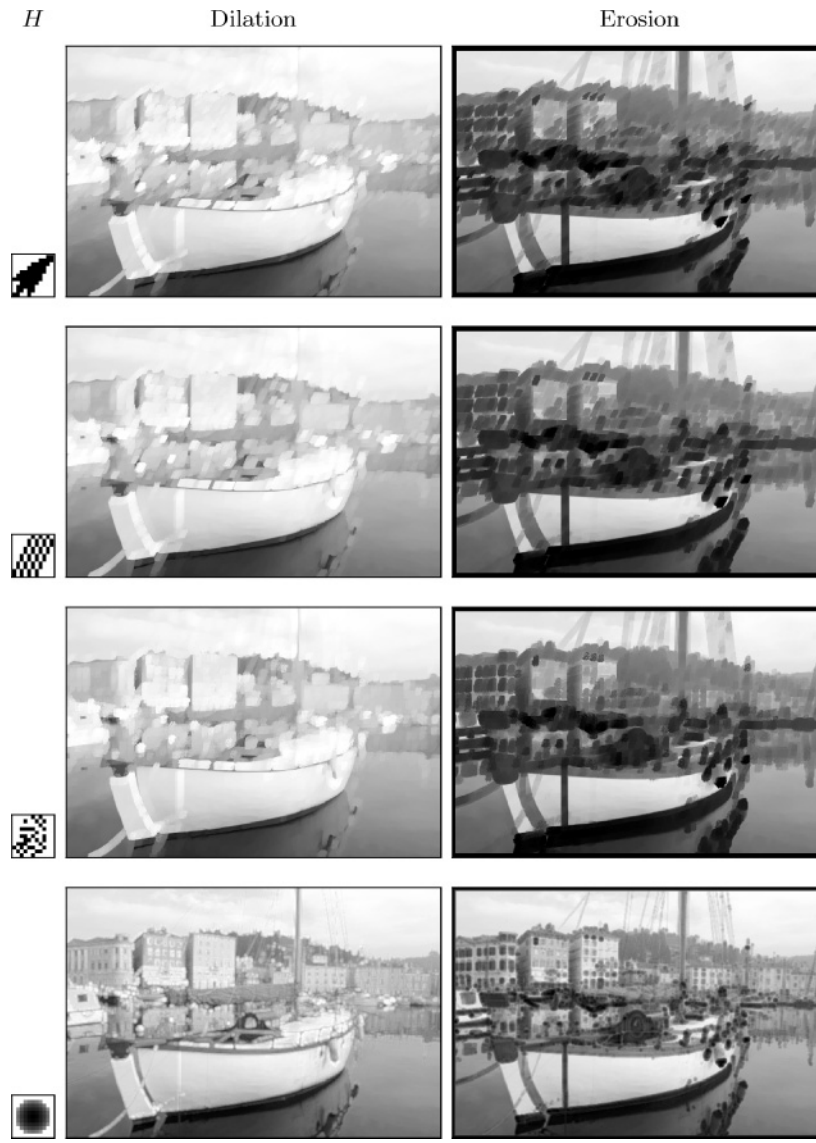


Fig. 10.22
Grayscale dilation and erosion with various free-form structuring elements.

display function or lookup table (LUT) either interactively through the menu

Image→Lookup Tables→Invert LUT

or within the Java program by invoking the `ImageProcessor` method

```
void invertLut()
```

on the corresponding `ImageProcessor` object. Any of these instructions changes only the screen presentation of the current image but not its contents (pixel values).

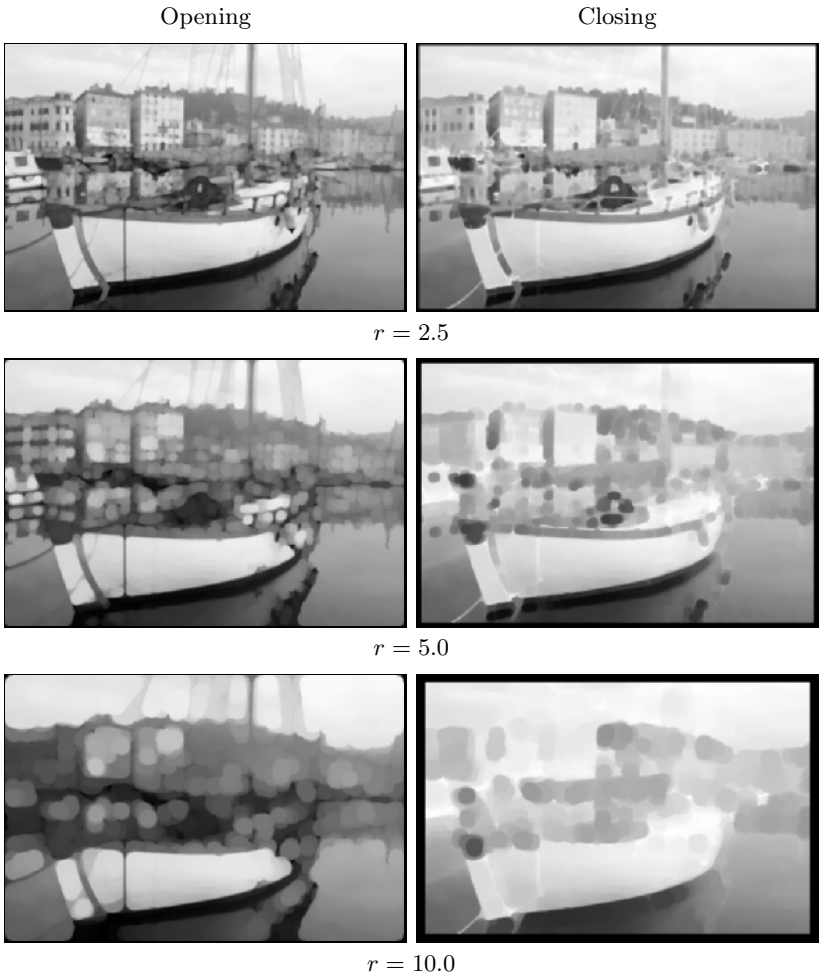


Fig. 10.23
Grayscale opening and closing with disk-shaped structuring elements. The radius r of the structuring element is 2.5 (top), 5.0 (center), or 10.0 (bottom).

10.5.2 Dilation and Erosion

Most morphological operations are already implemented in ImageJ as methods of the class `ImageProcessor` (see also Sec. 10.5.5); however, they are restricted to structuring elements of size 3×3 pixels.

In the following, we describe a sample implementation of binary *dilation* for arbitrary structuring elements that can be used (due to the duality of dilation and erosion; see Eqn. (10.16)) for implementing most other morphological operations. Input to `dilate()` is a binary image `I` with values 0 for the background and 255 for the foreground[5] and a two-dimensional structuring element `H` with 0/1-values whose origin (hot spot) is assumed at its center:

```
1  import ij.process.Blitter;
2  import ij.process.ImageProcessor;
```

[5] In fact, any value greater than 0 is considered a foreground pixel.

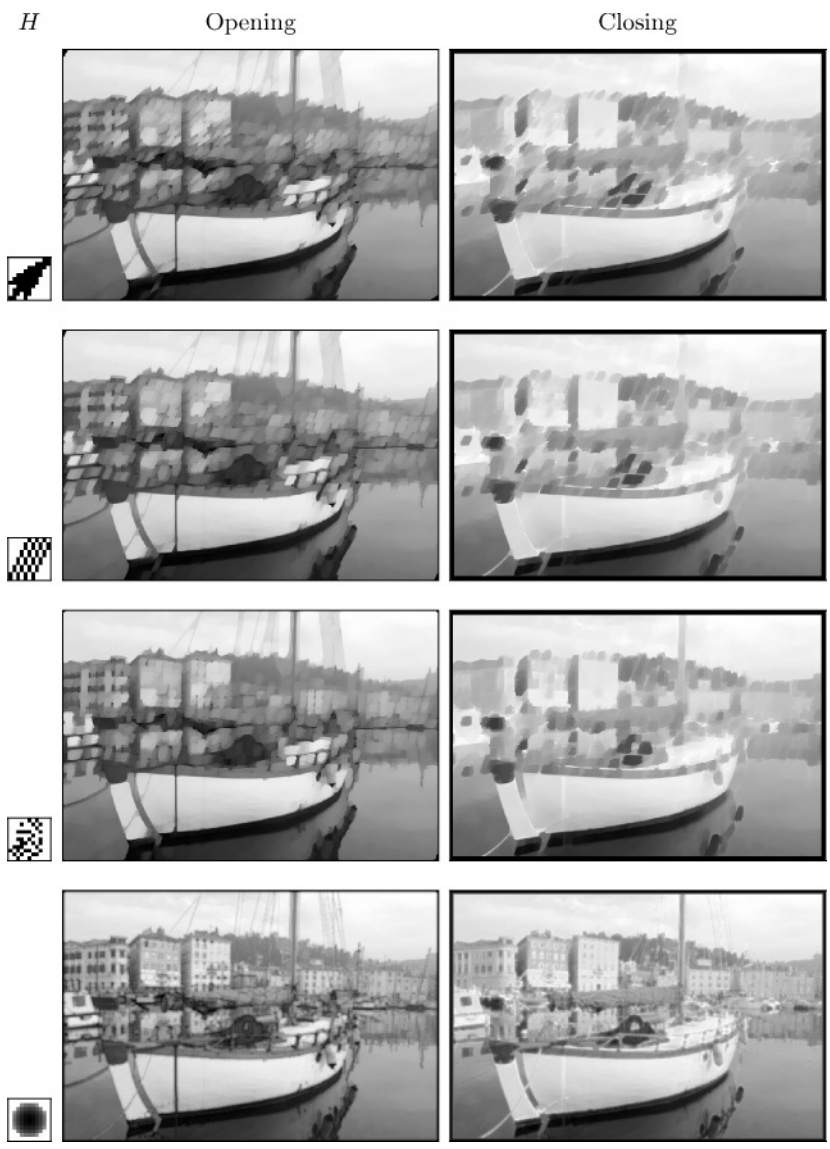


Fig. 10.24
Grayscale opening and closing with various free-form structuring elements.

```
3   ...
4   void dilate(ImageProcessor I, int[][] H){
5     //assume that the hot spot of H is at its center (ic,jc):
6     int ic = (H[0].length-1)/2;
7     int jc = (H.length-1)/2;
8
9     //create a temporary (empty) image:
10    ImageProcessor tmp
11          = I.createProcessor(I.getWidth(),I.getHeight());
12
```

```
13    for (int j=0; j<H.length; j++){
14      for (int i=0; i<H[j].length; i++){
15        if (H[j][i] > 0) { // this pixel is set
16          //copy image into position (i-ic,j-jc):
17          tmp.copyBits(I,i-ic,j-jc,Blitter.MAX);
18        }
19      }
20    }
21    //copy the temporary result back to original image
22    I.copyBits(np,0,0,Blitter.COPY);
23  }
```

The `dilate()` method destructively modifies the input image `I`. First (in line 10), a temporary (empty) image `tmp` of the same size as `I` is created, which is then modified and eventually (line 22) copied back to replace the input image. The actual dilation is performed iteratively by copying a shifted version of the original image into the temporary image `tmp` for every position (i, j) of the structuring element with $H(i, j) > 0$. This is done in line 17 using the `ImageProcessor` method `copyBits()` with `Blitter.MAX` as the operation parameter (see also Sec. 5.8.3). If the pixels are interpreted as binary values, the max-operation corresponds to a logical OR operation between the pixels in the intermediate image `tmp` and the shifted input image `I`.

Dilation is the only operation that must be implemented in detail since erosion can be performed as a dilation of the background by inverting the image, performing a dilation, and inverting again (see Alg. 10.1):

```
24  void erode(ImageProcessor I, int[][] H) {
25    ip.invert();
26    dilate(ip, reflect(H));
27    ip.invert();
28  }
```

In the above, the method `reflect(H)` (line 26) returns a mirrored copy of the structuring element H and `invert()` (lines 25, 27) is a standard ImageJ method defined by the class `ImageProcessor`.

10.5.3 Opening and Closing

Opening and closing operations are now easy to implement as combinations of dilation and erosion with the same structuring element `H`, as described in Sec. 10.3:

```
29  void open(ImageProcessor I, int[][] H) {
30    erode(I,H);
31    dilate(I,H);
32  }
```

```
33  void close(ImageProcessor I, int[][] H) {
34    dilate(I,H);
35    erode(I,H);
36  }
```

10.5.4 Outline

To implement the *outline* operation for extracting the boundary pixels, as described in Sec. 10.2.7, we use a 3×3 pixel structuring element `H` to represent the 4-neighborhood. First we create a duplicate (`Ie`) of the input image (`I`), which is then subject to erosion with `H` (line 43). The boundary pixels are obtained by computing the difference between the original and the eroded image (using the standard method `copyBits()` with the argument `Blitter.DIFFERENCE`). In binary terms, this is an exclusive-OR (XOR) operation between the pixels in `I` and `Ie`, which implements the set intersection (see Eqn. (10.19)). The differencing operation in line 44 stores its result in `I`, which finally contains the boundary pixels of the foreground structures:

```
37  void outline(ImageProcessor I) {
38    int[][] H = { //4-neighborhood structuring element
39      {0,1,0},
40      {1,1,1},
41      {0,1,0}};
42    ImageProcessor Ie = I.duplicate();
43    erode(Ie,H);    // I' ← I ⊖ H
44    I.copyBits(Ie,0,0,Blitter.DIFFERENCE); // I ← XOR(I, I')
45  }
```

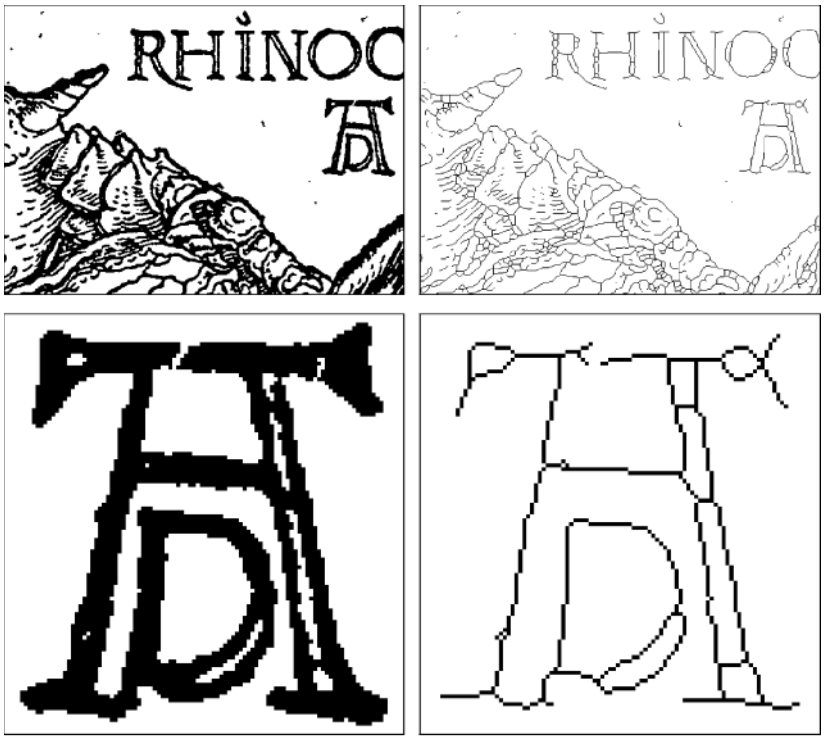
10.5.5 Morphological Operations in ImageJ

Class `ImageProcessor`

ImageJ defines several methods for basic morphological operations in the class `ImageProcessor`:

```
void dilate()
void erode()
void open()
void close()
```

All these methods apply a 3×3 pixel box-shaped structuring element (see Fig. 10.11 (b)) and perform either binary or grayscale operations, depending upon the image content. The class `ColorProcessor` uses the same methods for RGB images by processing the color channels individually like ordinary grayscale or binary images.



(a) (b)

Fig. 10.25
Example of thinning with the `skeletonize()` method: original image and detail (a) and results from thinning (b).

Class `BinaryProcessor`

The class `BinaryProcessor` (a subclass of `ByteProcessor`) offers the specific morphological methods

```
void outline()
void skeletonize()
```

which are only defined for binary images. The method `outline()` implements the extraction of boundary pixels using an 8-neighborhood structuring element, as described in Sec. 10.2.7.

The operation implemented by the method `skeletonize()` is often referred to as "thinning" or "skeletonization", which iteratively erodes structures down to a thickness of 1 pixel without splitting them. This requires a decision based on the current image content within the filter region (typically of size 3×3 pixels) as to whether another erosion should be applied or not. The operation repeats until no more changes can be made to the result (see, e. g., [38, p. 535] or [59, p. 517] for details). The actual implementation in ImageJ is based on an efficient algorithm by Zhang and Suen [107], and an example of applying the `skeletonize()` method is shown in Fig. 10.25.

The methods `outline()` and `skeletonize()` are only applicable to objects of type `BinaryProcessor`, which can be created from existing

ByteProcessor objects. This assumes, however, that the original image contains only values of 0 (background) and 255 (foreground). The following example shows the use of `outline()` within the `run()` method of an ImageJ plugin:

```
1  public void run(ImageProcessor ip) {
2      ByteProcessor byteP
3          = (ByteProcessor) ip.convertToByte(true); // scale!
4      BinaryProcessor binP
5          = new BinaryProcessor(byteP);
6      binP.outline();
7      ...
8  }
```

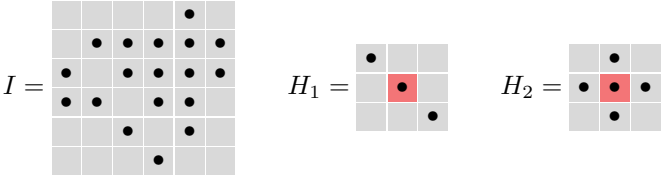
Notice that the new `BinaryProcessor` object `binP` does not allocate any new image data but only references the data of the parent image `byteP`. Thus any subsequent modification to `binP` (e. g., by invoking the method `outline()`) is also visible in `byteP`.

Other morphological filters

In addition to the morphological operations implemented in ImageJ itself, there are additional plugins and complete morphological packages available online,[6] including the morphology operators by Gabriel Landini and the Grayscale Morphology package by Dimiter Prodanov, which allows structuring elements to be interactively specified (a modified version was used for some examples in this chapter).

10.6 Exercises

Exercise 10.1. Manually compute the results of dilation and erosion for the following image I and the structuring elements H_1 and H_2:



Exercise 10.2. 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.

[6] http://rsb.info.nih.gov/ij/plugins/.

Exercise 10.3. Show that, in the special case of the structuring elements with the contents

 for binary images and  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 Sec. 6.4.1).

11

Regions in Binary Images

In binary images, a pixel 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 regions in images and how to isolate and describe such structures.

Our primary task then is to devise a program to interpret the number and type of objects in a figure like Fig. 11.1. 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*.

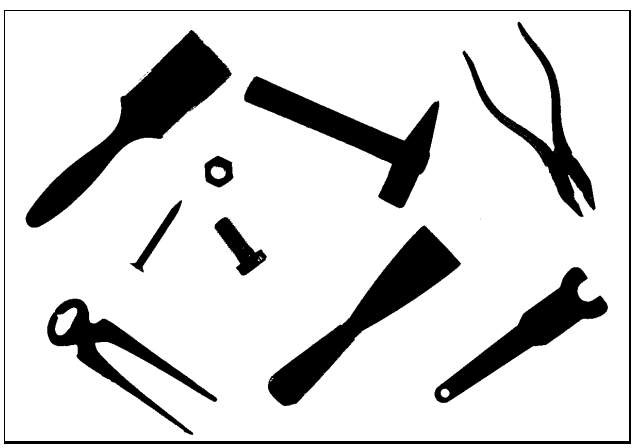


Fig. 11.1
Binary image with nine objects. Each object corresponds to a region of related foreground pixels.

11.1 Finding 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. 11.2.2, we describe a third method that combines two useful processes, region labeling and contour finding, in a single algorithm.

Independent of which of the methods above we use, we must first settle on either the 4- or 8-connected definition of neighboring (Fig. 10.5) 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, the resulting binary image $I(u, v)$ uses the values 0 for background and 1 for foreground, and any other value can be used for numbering (labeling) the regions:

$$I(u,v) = \begin{cases} 0 & \textit{background} \text{ pixel} \\ 1 & \textit{foreground} \text{ pixel} \\ 2, 3, \ldots & \text{region } \textit{label}. \end{cases}$$

11.1.1 Region Labeling with Flood Filling

The underlying algorithm for region marking using *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. 11.1):

(A) **Recursive Flood Filling:** The recursive version (Alg. 11.1, lines 8–15) 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

[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.

Algorithm 11.1
Region marking using *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. Three variations of the FloodFill() procedure are presented: *recursive*, *depth-first*, and *breadth-first*.

```
1:  RegionLabeling(I)
        I: binary image (0 = background, 1 = foreground)
        The image I is labeled (destructively modified) and returned.
2:      Initialize m ← 2 (the value of the next label to be assigned).
3:      Iterate over all image coordinates (u, v).
4:          if I(u, v) = 1 then
5:              FloodFill(I, u, v, m)        ▷ use any of the 3 versions below
6:              m ← m + 1.
7:      return the labeled image I.
```

```
8:  FloodFill(I, u, v, label)                          ▷ Recursive Version
9:      if coordinate (u, v) is within image boundaries and I(u, v) = 1 then
10:         Set I(u, v) ← label
11:         FloodFill(I, u+1, v, label)
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)                          ▷ Depth-First Version
17:     Create an empty stack S
18:     Put the seed coordinate ⟨u, v⟩ onto the stack: Push(S, ⟨u, v⟩)
19:     while S is not empty do
20:         Get the next coordinate from the top of the stack:
                ⟨x, y⟩ ← Pop(S)
21:         if coordinate (x, y) is within image boundaries and I(x, y) = 1
                then
22:             Set I(x, y) ← label
23:             Push(S, ⟨x+1, y⟩)
24:             Push(S, ⟨x, y+1⟩)
25:             Push(S, ⟨x, y−1⟩)
26:             Push(S, ⟨x−1, y⟩)
27:     return.
```

```
28: FloodFill(I, u, v, label)                          ▷ Breadth-First Version
29:     Create an empty queue Q
30:     Insert the seed coordinate ⟨u, v⟩ into the queue: Enqueue(Q, ⟨u, v⟩)
31:     while Q is not empty do
32:         Get the next coordinate from the front of the queue:
                ⟨x, y⟩ ← Dequeue(Q)
33:         if coordinate ⟨x, y⟩ is within image boundaries and I(x, y) = 1
                then
34:             Set I(x, y) ← label
35:             Enqueue(Q, ⟨x+1, y⟩)
36:             Enqueue(Q, ⟨x, y+1⟩)
37:             Enqueue(Q, ⟨x, y−1⟩)
38:             Enqueue(Q, ⟨x−1, y⟩)
39:     return.
```

tree structure, rooted at the starting point, is defined by the neighborhood relation between pixels. The recursive step corresponds to a *depth-first traversal* [25] of this tree and results in very short and elegant program code. Unfortunately, since the depth of the recursion necessary 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. 11.1, lines 16–27) by implementing and managing its own *stacks*. In this case, the stack records the "open" (that is, the not yet visited) elements. As in the recursive version (A), the tree of pixels is traversed *depth-first*. By making use of its own dedicated stack (which is created in the *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. 11.1, lines 28–39). 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 corresponds practically 1:1 to its Java implementation. However, a normal Java runtime does not support more than about 10,000 recursive calls of the FloodFill() procedure (Alg. 11.1, 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 11.1 gives the complete Java implementation for both variants of the iterative FloodFill() procedure. In line 1, a new Java class `Node` is implemented to represent a single pixel coordinate.

In implementing the stack (S) in the iterative *depth-first* Version (B), we use the stack data structure provided by the Java class `Stack` (Prog. 11.1, line 10), which serves as a container for generic Java objects. For the queue data structure (Q) in the *breadth-first* variant (C), we use the Java class `LinkedList`[2] with the methods `addFirst()`, `removeLast()`, and `isEmpty()` (Prog. 11.1, line 25). We have specified `<Node>` as a parameter for both container classes (i. e., parameterized) so that only objects of the type `Node` can be stored in these data structures.[3]

[2] The class `LinkedList` is a part of the *Java Collection Frameworks* (see also Appendix B, p. 462).

[3] Generic types and templates (i. e., the ability to specify a parameterization for a container) have only been available since Java 5 (1.5).

Program 11.1
Flood filling (Java implementation). The *depth-first* variant uses the stack operations provided by the methods `push()`, `pop()`, and `isEmpty()` of the Java class `Stack`. The *breadth-first* variant uses the Java class `LinkedList` (with access methods `addFirst()` for Enqueue() and `removeLast()` for Dequeue()) for implementing the queue data structure.

```
1 class Node {
2   int x, y;
3
4   Node(int x, int y) { // constructor method
5     this.x = x;
6     this.y = y;
7   }
8 }
```

Depth-first variant (using a *stack*):

```
9  void floodFill(ImageProcessor ip, int x, int y, int label) {
10   Stack<Node> s = new Stack<Node>(); // stack
11   s.push(new Node(x,y));
12   while (!s.isEmpty()){
13     Node n = s.pop();
14     if ((n.x>=0) && (n.x<width) && (n.y>=0) && (n.y<height)
15       && ip.getPixel(n.x,n.y)==1) {
16       ip.putPixel(n.x,n.y,label);
17       s.push(new Node(n.x+1,n.y));
18       s.push(new Node(n.x,n.y+1));
19       s.push(new Node(n.x,n.y-1));
20       s.push(new Node(n.x-1,n.y));
21     }
22   }
23 }
```

Breadth-first variant (using a *queue*):

```
24 void floodFill(ImageProcessor ip, int x, int y, int label) {
25   LinkedList<Node> q = new LinkedList<Node>(); // queue
26   q.addFirst(new Node(x,y));
27   while (!q.isEmpty()) {
28     Node n = q.removeLast();
29     if ((n.x>=0) && (n.x<width) && (n.y>=0) && (n.y<height)
30       && ip.getPixel(n.x,n.y)==1) {
31       ip.putPixel(n.x,n.y,label);
32       q.addFirst(new Node(n.x+1,n.y));
33       q.addFirst(new Node(n.x,n.y+1));
34       q.addFirst(new Node(n.x,n.y-1));
35       q.addFirst(new Node(n.x-1,n.y));
36     }
37   }
38 }
```

Figure 11.2 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, step by step, 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. 11.2 using the implementation given (Prog. 11.1), 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.

11.1.2 Sequential Region Labeling

Sequential region marking is a classical, nonrecursive technique that is known in the literature as "region labeling". The algorithm consists in essence of two steps: (1) a 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 region. Even though this algorithm is relatively complex, especially the second stage, its moderate memory requirements have made it the method of choice in practice over other simpler methods. The entire process is summarized in Alg. 11.2.

Stage 1: Preliminary 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 vicinity of each pixel must be examined ($\times$ marks the actual pixel at the position (u, v)):

$$\mathcal{N}_4(u,v) = \begin{array}{|c|c|c|}\hline & N_2 & \\ \hline N_1 & \times & \\ \hline & & \\ \hline\end{array} \quad \text{or} \quad \mathcal{N}_8(u,v) = \begin{array}{|c|c|c|}\hline N_2 & N_3 & N_4 \\ \hline N_1 & \times & \\ \hline & & \\ \hline\end{array}$$

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 $N_1 \ldots N_4$ must be examined. In the following example, we will use an 8-connected neighborhood and the image from Fig. 11.3 (a).

Propagating labels

Again we assume that, in the image, the value $I(u, v) = 0$ represents foreground pixels and the value $I(u, v) = 1$ represents background 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

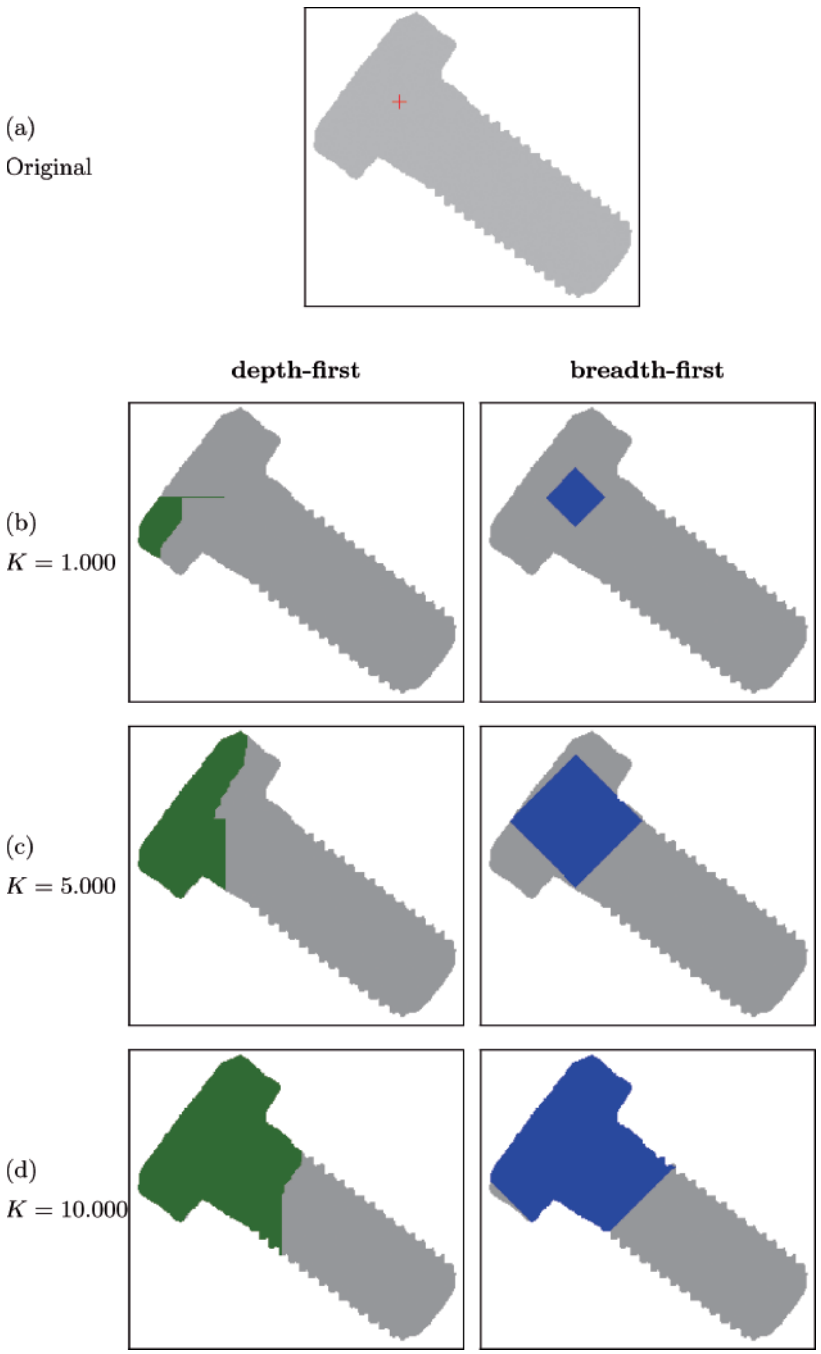


Fig. 11.2
Iterative *flood filling*—comparison between *depth-first* and *breadth-first* variations. The starting point, marked + in the original image (a), was arbitrarily chosen. Intermediate results of the *flood fill* algorithm after 1000, 5000, and 10,000 marked pixels are shown (b–d). The image size is 250 × 242 pixels.

Algorithm 11.2
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 $2 \ldots m-1$.

1: SequentialLabeling(I)
   I: binary image (0 = *background*, 1 = *foreground*)
   The image I is labeled (destructively modified) and returned.
   Pass 1—Assign Initial Labels:
2: Initialize $m \leftarrow 2$ (the value of the next label to be assigned).
3: Create an empty set $\mathcal{C}$ to hold the collisions: $\mathcal{C} \leftarrow \{\}$.
4: **for** $v \leftarrow 0 \ldots H-1$ **do** ▷ H = height of image I
5: **for** $u \leftarrow 0 \ldots W-1$ **do** ▷ W = width of image I
6: **if** $I(u,v) = 1$ **then** do one of:
7: **if** all neighbors of (u,v) are background pixels (all $n_i = 0$) **then**
8: $I(u,v) \leftarrow m$.
9: $m \leftarrow m + 1$.
10: **else if** exactly *one* of the neighbors has a label value $n_k > 1$ **then**
11: set $I(u,v) \leftarrow n_k$
12: **else if** *several* neighbors of (u,v) have label values $n_j > 1$ **then**
13: Select one of them as the new label: $I(u,v) \leftarrow k \in \{n_j\}$.
14: **for** all other neighbors of $u,v)$ with label values $n_i > 1$ and $n_i \neq k$ **do**
15: Create a new label collision $\boldsymbol{c}_i = \langle n_i, k \rangle$.
16: Record the collision: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\boldsymbol{c}_i\}$.
   Remark: The image I now contains label values $0, 2, \ldots m-1$.
   Pass 2—Resolve Label Collisions:
17: Let $\mathcal{L} = \{2, 3, \ldots m-1\}$ be the set of preliminary region labels.
18: Create a partitioning of $\mathcal{L}$ as a *vector of sets*, one set for each label value: $\mathcal{R} \leftarrow [\mathcal{R}_2, \mathcal{R}_3, \ldots, \mathcal{R}_{m-1}] = [\{2\}, \{3\}, \{4\}, \ldots, \{m-1\}]$, so $\mathcal{R}_i = \{i\}$ for all $i \in \mathcal{L}$.
19: **for all** collisions $\langle a, b \rangle \in \mathcal{C}$ **do**
20: Find in $\mathcal{R}$ the sets $\mathcal{R}_a$, $\mathcal{R}_b$ containing the labels a, b, resp.:
$\mathcal{R}_a \leftarrow$ the set that currently contains label a
$\mathcal{R}_b \leftarrow$ the set that currently contains label b
21: **if** $\mathcal{R}_a \neq \mathcal{R}_b$ (a and b are contained in different sets) **then**
22: Merge sets $\mathcal{R}_a$ and $\mathcal{R}_b$ by moving all elements of $\mathcal{R}_b$ to $\mathcal{R}_a$:
$\mathcal{R}_a \leftarrow \mathcal{R}_a \cup \mathcal{R}_b$
$\mathcal{R}_b \leftarrow \{\}$
   Remark: All *equivalent* label values (i. e., all labels of pixels in the same region) are now contained in the same set $\mathcal{R}_i$ within $\mathcal{R}$.
   Pass 3—Relabel the Image:
23: Iterate through all image pixels (u,v):
24: **if** $I(u,v) > 1$ **then**
25: Find the set $\mathcal{R}_i$ in $\mathcal{R}$ that contains label $I(u,v)$.
26: Choose one unique representative element k from the set $\mathcal{R}_i$ (e. g., the minimum value, $k = \min(\mathcal{R}_i)$).
27: Replace the image label: $I(u,v) \leftarrow k$.
28: **return** the labeled image I.

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 in (Fig. 11.3 (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. 11.3 (d)).

When two labels a, b collide, then we know that they are actually "equivalent"; i. e., 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 during 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 structure such as lists and hash tables. Upon the completion of the first steps, all the original foreground pixels have been provisionally marked, and all the collisions between markings within the same regions have been registered for subsequent processing.

The example in Fig. 11.4 illustrates the state upon completion of step 1: all foreground pixels have been assigned preliminary labels (Fig. 11.4 (a)), and the following collisions (depicted by circles) between the labels $\langle 2, 4\rangle$, $\langle 2, 5\rangle$, and $\langle 2, 6\rangle$ have been registered. The labels $\mathcal{L} = \{2, 3, 4, 5, 6, 7\}$ and collisions $\mathcal{C} = \{\langle 2, 4\rangle, \langle 2, 5\rangle, \langle 2, 6\rangle\}$ correspond to the nodes and edges of an undirected graph (Fig. 11.4 (b)).

Step 2: Resolving 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 different labels to be connected transitively (e. g., $\langle a, b\rangle \cap \langle b, c\rangle \Rightarrow \langle a, c\rangle$) 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 [25], where the labels $\mathcal{L}$ determined in Step 1 constitute the

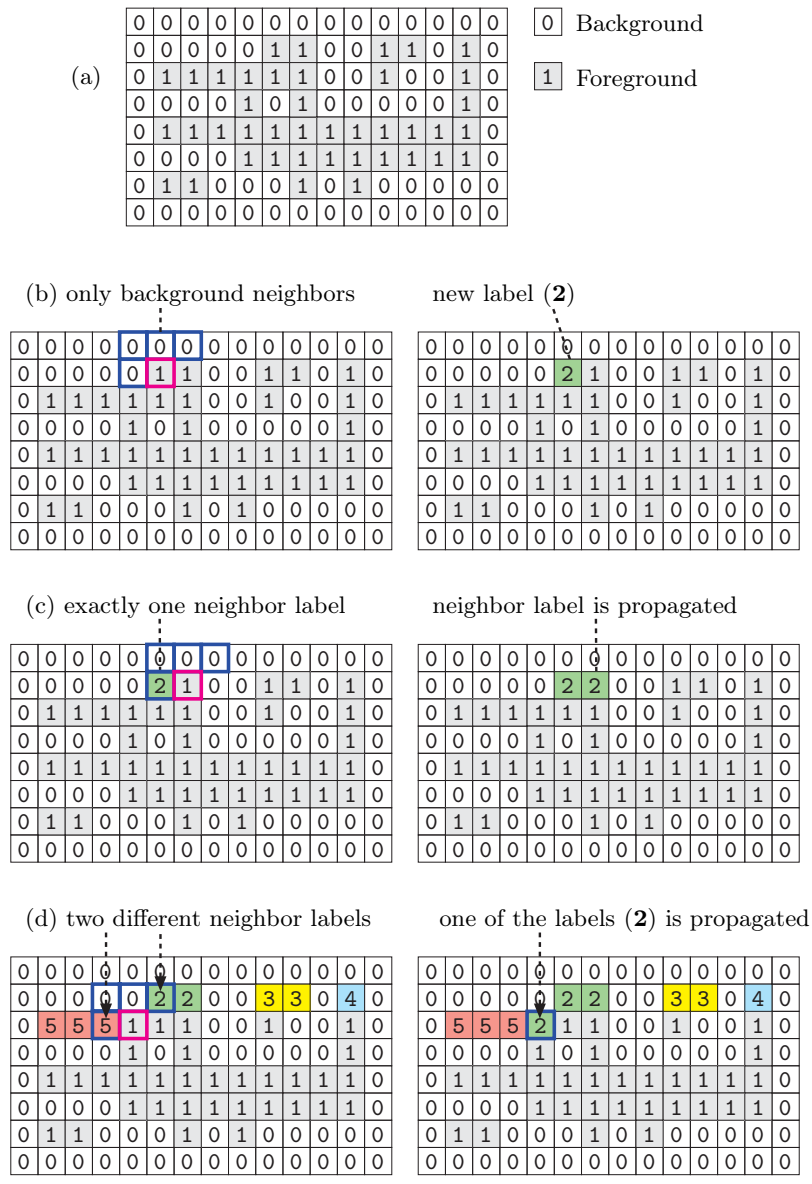


Fig. 11.3
Sequential region labeling—label propagation. Original image (a). The first foreground pixel [**1**] is found in (b): all neighbors are background pixels [**0**], and the pixel is assigned the first label [**2**]. In the next step, (c) is exactly *one* neighbor pixel with the label **2** marked, 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 $\langle \mathbf{2}, \mathbf{5} \rangle$ is registered.

"nodes" of the graph and the registered collisions $\mathcal{C}$ make up its "edges" (Fig. 11.4 (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 have the same value (for example, using the smallest original label in the region) as in Fig. 11.5.

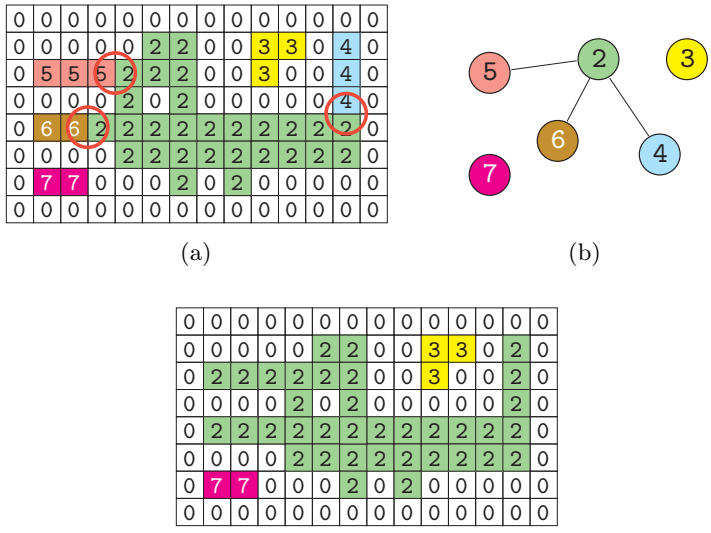


Fig. 11.4
Sequential region labeling—intermediate result after Step 1. Label collisions indicated by circles (a); the nodes of the undirected graph (b) correspond to the labels, and its edges correspond to the collisions.

Fig. 11.5
Sequential region labeling—final result after Step 2. All equivalent labels are replaced by the smallest label within that region.

11.1.3 Region Labeling—Summary

In this section, we 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 (Sec. 11.1.1) method was not useful because of practical limits on the depth of recursion and the high memory costs associated with it. We also saw that classical sequential region labeling (Sec. 11.1.2) 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.

11.2 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 task: 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.

11.2.1 External and Internal Contours

As we have already seen in Sec. 10.2.7, the pixels along the edge of a binary region (that is, its border) can be identified using simple morphological operations and difference images. It must be stressed, however,

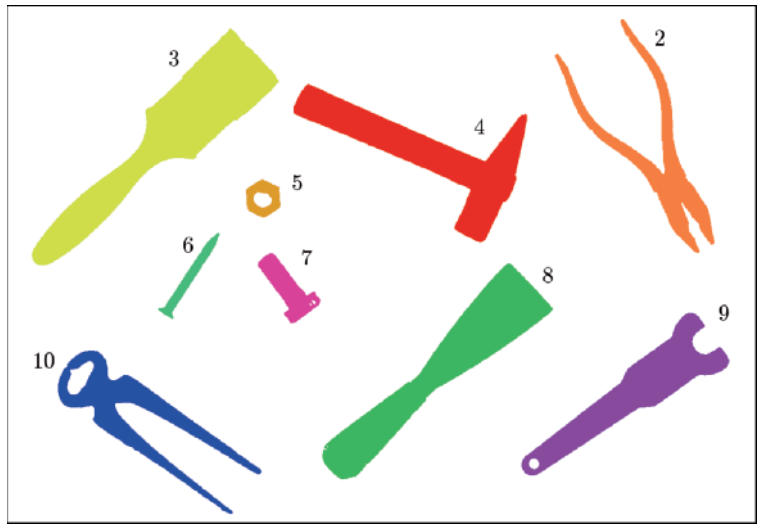


Fig. 11.6
Example of a complete region labeling. The pixels within each region have been colored according to the consecutive label values $2, 3, \ldots 10$ they were assigned. The corresponding region statistics are shown in the table below (total image size is 1212×836).

Label	Area (*pixels*)	Bounding Box (*left*, *top*, *right*, *bottom*)	Center (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)

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 pixels 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. 11.7).

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. 11.8). Therefore, when tracing a contour from a start point $\boldsymbol{x}_S$, returning to the start point is *not* a sufficient condition for terminating the contour tracing. Other factors, such as the current direction along which the contour is being traced, must be taken into account.

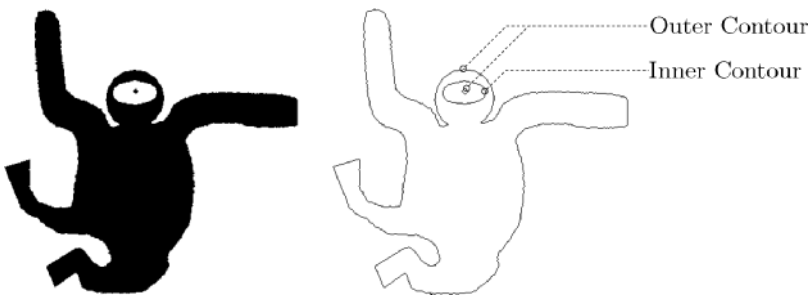


Fig. 11.7
Binary image with outer and inner contours. The outer contour lies along the outside of the foreground region (dark). The inner contour surrounds the space within the region, which may contain further regions (holes), and so on in a recursive manner.

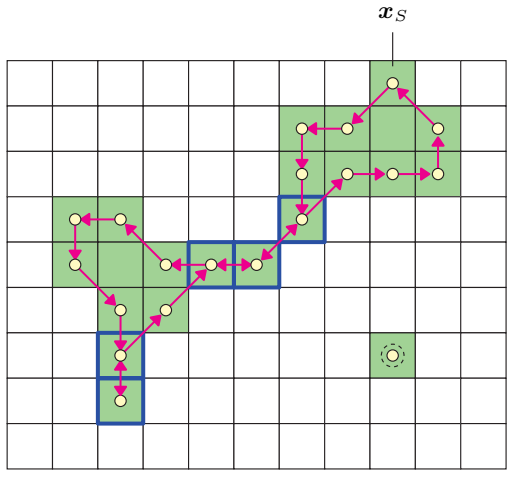


Fig. 11.8
The path along a contour as an ordered sequence of pixel coordinates with a given start point x_S. Individual pixels may occur (be visited) more than once within the path, and a region consisting of a single isolated pixel will also have a contour (bottom right).

One apparently simple way of determining a contour is to proceed based on the two-stage idea presented in the previous section (11.1); that is, 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 a similar way, the internal contour can be found starting from a region's interior. A wide range of algorithms based on first finding the regions and then following along their contours have been published, including [84], [77, pp. 142–148], and [90, p. 296], and while the idea is simple in essence, the implementation requires careful record-keeping and is complicated by special cases such as the single-pixel bridges described in the previous section.

As an alternative, we present the following *combined* algorithm that, in contrast to the classical methods above, combines contour finding and region labeling in a single process.

11.2.2 Combining Region Labeling and Contour Finding

This method, based on [23], combines the concepts of sequential region labeling (Sec. 11.1) 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 a complicated data structures and is very efficient when compared with other methods with similar capabilities.

We now sketch the fundamental idea of the algorithm. While the main idea of the algorithm can be sketched out in a few simple steps, the actual implementation requires attention to a number of details, so we have provided the complete Java source for an ImageJ plugin implementation in Appendix D (pp. 532–542). The most important steps of the method are illustrated in Fig. 11.9:

1. As in the sequential region labeling (Alg. 11.2), 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.

2. At a given position in the image, the following cases may occur:

   Case A: The transition from a foreground pixel to a previously unmarked foreground pixel A means that A lies on the outer edge of a new region. A new *label* is allocated and the associated *outer* contour is traversed and marked by calling the method TraceContour() (see Fig. 11.9 (a) and Alg. 11.3 (line 20)). Furthermore, all background pixels directly bordering the region are marked with the value -1.

   Case B: The transition from a foreground pixel B to an unmarked background pixel means that B lies on the edge of an *inner* contour (Fig. 11.9 (b)). Starting from B, the inner contour is traversed and its pixels are labeled with labels from the surrounding region (Fig. 11.9 (c)). Also, all bordering background pixels are again assigned the value of -1.

   Case C: When a foreground pixel does not lie on a contour (i. e., it is not on an edge), then the neighboring pixel to the left has already been labeled (Fig. 11.9 (d)) and this label is propagated to the current pixel.

In Algs. 11.3 and 11.4, the entire procedure is presented again and explained precisely. The method CombinedContourLabeling() traverses the image line-by-line and calls the method 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" LM by the method FindNextPoint() (Alg. 11.4).

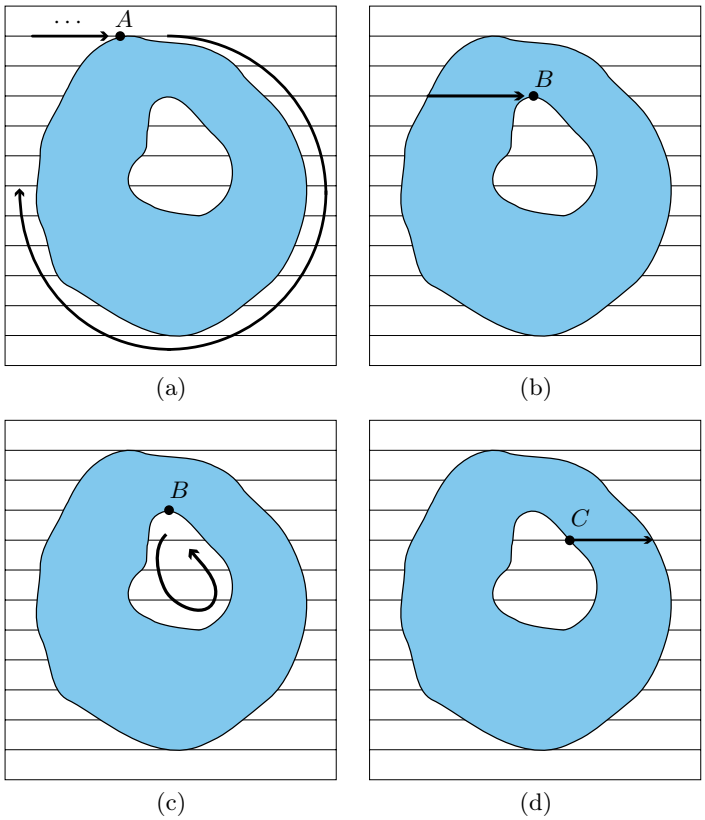


Fig. 11.9
Combined region labeling and contour following (after [23]). The image is traversed from the top left to the lower right a row at a time. In (a), the first point A on the outer edge of the region is found. Starting from point A, the pixels on the edge along the outer contour are visited and labeled until A is reached again. In (b), the first point B on an inner contour is found. The pixels along the inner contour are visited and labeled until arriving back at B (c). In (d), an already labeled point C on an inner contour is found. Its label is propagated along the image row within the region.

11.2.3 Implementation

The complete implementation of the algorithm in Java (ImageJ) can be found in Appendix D (beginning on page 532). The implementation closely follows the description in Algs. 11.3 and 11.4 but illustrates several additional details:[4]

- First the image I (`pixelMap`) and the associated label map LM (`labelMap`) are enlarged by adding one pixel around the borders. The new pixels are marked as *background* (0) in the image I. This simplifies contour following and eliminates the need to handle a number of special situations.
- As contours are found they are stored in an object of the class `ContourSet`, separated into outer and inner contours. The contours themselves are represented by the classes `OuterContour` and `InnerContour`, with a common superclass `Contour`. Every contour consists of an ordered sequence of coordinate points of the class `Node`

[4] In the following description the names in parentheses after the algorithmic symbols denote the corresponding identifiers used in the Java implementation.

Algorithm 11.3
Combined contour tracing and region labeling. Given a binary image I, the method CombinedContourLabeling() 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 the method TraceContour() (line 20 and line 27). TraceContour() itself is described in Alg. 11.4.

```
1:  CombinedContourLabeling (I)
        I: binary image
        Returns a set of contours and a label map (labeled image).
2:      Create an empty set of contours: C ← {}
3:      Create a label map LM of the same size as I and initialize:
4:      for all (u, v) do
5:          LM(u, v) ← 0                                  ▷ label map LM
6:      R ← 0                                             ▷ region counter R
7:      Scan the image from left to right and top to bottom:
8:      for v ← 0 ... N−1 do
9:          L_k ← 0                                       ▷ current label L_k
10:         for u ← 0 ... M−1 do
11:             if I(u, v) is a foreground pixel then
12:                 if (L_k ≠ 0) then                     ▷ continue existing region
13:                     LM(u, v) ← L
14:                 else
15:                     L_k ← LM(u, v)
16:                     if (L_k = 0) then                 ▷ hit new outer contour
17:                         R ← R + 1
18:                         L_k ← R
19:                         x_S ← (u, v)
20:                         c_outer ← TraceContour(x_S, 0, L_k, I, LM)
21:                         C ← C ∪ {c_outer}             ▷ collect new contour
22:                         LM(u, v) ← L_k
23:             else                                      ▷ I(u, v) is a background pixel
24:                 if (L ≠ 0) then
25:                     if (LM(u, v) = 0) then            ▷ hit new inner contour
26:                         x_S ← (u−1, v)
27:                         c_inner ← TraceContour(x_S, 1, L_k, I, LM)
28:                         C ← C ∪ {c_inner}             ▷ collect new contour
29:                     L ← 0
30:     return (C, LM).                                   ▷ return the set of contours and the label map
```

continued in Alg. 11.4 ▷▷

(defined on p. 203). The Java container class `ArrayList` (templated on the type `Node`) is used as a dynamic data structure for storing the point sequences of the outer and inner contours.

- The method `traceContour()` (see p. 538) traverses an outer or inner contour, beginning from the starting point $\boldsymbol{x}_S$ (`xS`, `yS`). It calls the method `findNextPoint()`, to determine the next contour point $\boldsymbol{x}_T$ (`xT`, `yT`) following $\boldsymbol{x}_S$:
  - In the case that no following point is found, then $\boldsymbol{x}_S = \boldsymbol{x}_T$ and the region (contour) consists of a single isolated pixel. The method `traceContour()` is finished.
  - In the other case the remaining contour points are found by repeatedly calling `findNextPoint()`, and for every successive pair of points the *current* point $\boldsymbol{x}_c$ (`xC`, `yC`) and the *previous* point $\boldsymbol{x}_p$

Algorithm 11.4
Combined contour finding and region labeling (continued from Alg. 11.3). Starting from $\boldsymbol{x}_S$, 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 LM. Given a point $\boldsymbol{x}_c$, TraceContour uses FindNextPoint() to determine the next point along the contour (line 10). The function Delta() returns the next coordinate in the sequence, taking into account the search direction d.

```
 1: TraceContour(x_S, d_S, L_k, I, LM)
        x_S: start position, d_S: initial search direction,
        L_c: label for this contour
        I: original image, LM: label map.
        Traces and returns the contour starting at x_S.
 2:   (x_T, d_next) ← FindNextPoint(x_S, d_S, I, LM)
 3:   c ← [x_T]                          ▷ create a contour starting with x_T
 4:   x_p ← x_S                          ▷ previous position x_p = (u_p, v_p)
 5:   x_c ← x_T                          ▷ current position x_c = (u_c, v_c)
 6:   done ← (x_S ≡ x_T)                 ▷ isolated pixel?
 7:   while (¬done) do
 8:     LM(u_c, v_c) ← L_c
 9:     d_search ← (d_next + 6) mod 8
10:     (x_n, d_next) ← FindNextPoint(x_c, d_search, I, LM)
11:     x_p ← x_c
12:     x_c ← x_n
13:     done ← (x_p ≡ x_S ∧ x_c ≡ x_T)  ▷ back at start point?
14:     if (¬done) then
15:       Append(c, x_n)                ▷ add point x_n to contour c
16:   return c.                         ▷ return this contour

17: FindNextPoint(x_c, d, I, LM)
        x_c: start point, d: search direction,
        I: original image, LM: label map.
18:   for i ← 0...6 do                  ▷ search in 7 directions
19:     x' ← x_c + Delta(d)             ▷ x' = (u', v')
20:     if I(u', v') is a background pixel then
21:       LM(u', v') ← −1               ▷ mark background as visited (−1)
22:       d ← (d + 1) mod 8
23:     else                            ▷ found a nonbackground pixel at x'
24:       return (x', d)
25:   return (x_c, d).                  ▷ found no next point, return start point
```

26: Delta$(d) = (\Delta x, \Delta y)$, with

d	0	1	2	3	4	5	6	7
Δx	1	1	0	−1	−1	−1	0	1
Δy	0	1	1	1	0	−1	−1	−1

(`xP`, `yP`) are recorded. Only when *both* points correspond to the original starting points on the contour, $\boldsymbol{x}_p = \boldsymbol{x}_S$ and $\boldsymbol{x}_c = \boldsymbol{x}_T$, we know that the contour has been completely traversed.

- The method `findNextPoint()` (see p. 539) determines which point on the contour follows the current point $\boldsymbol{x}_c$ (`Xc`) by searching in the *direction* d (`dir`), depending upon the position of the previous contour point. Starting in the first search direction, up to seven neighboring pixels (all neighbors except the previous contour point) are searched in clockwise direction until the next contour point is found. At the same time, all background pixels in the *label map* LM (`labelMap`) are marked with the value −1 to prevent them from being searched again. If no valid contour point is found among the

Program 11.2
Example of using the class `ContourTracer`.

```
1 import java.util.ArrayList;
2 ...
3 public class Trace_Contours implements PlugInFilter {
4   public void run(ImageProcessor ip) {
5     ContourTracer tracer = new ContourTracer(ip);
6     ContourSet cs = tracer.getContours();
7     // process outer and inner contours:
8     ArrayList<Contour> outer = cs.outerContours;
9     ArrayList<Contour> inner = cs.innerContours;
10    ...
11  }
12 }
```

seven possible neighbors, then `findNextPoint()` returns the original point $\boldsymbol{x}_c$ (`Xc`).

In this implementation the core of the algorithm is contained in the class `ContourTracer` (pp. 536–541). Program 11.2 provides an example of its usage within the `run()` method of an ImageJ plugin. An interesting detail is the class `ContourOverlay` (pp. 541–542) that is used to display the resulting contours by a vector graphics overlay. In this way graphic structures that are smaller and thinner than image pixels can be visualized on top of ImageJ's raster images at arbitrary magnification (zooming).

11.2.4 Example

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 11.10 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 trough 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 11.11 shows the results for a larger section taken from a real image (Fig. 10.12).

11.3 Representing Image Regions

11.3.1 Matrix Representation

A natural representation for images is a matrix (that is, 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

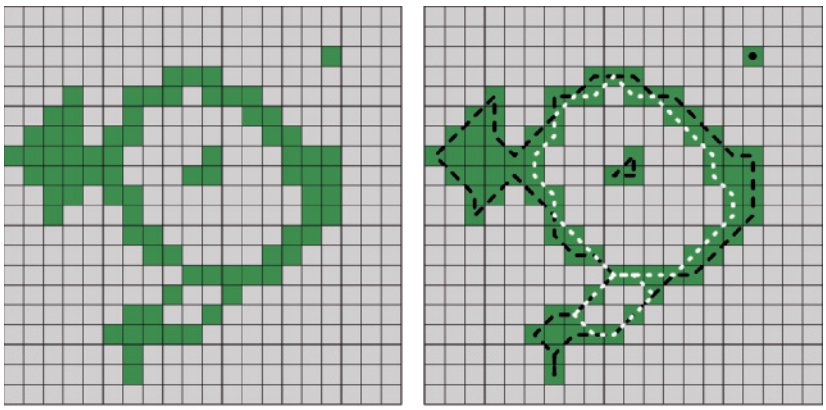


Fig. 11.10
Combined contour and region marking: original image in gray (a), located contours (b) with black lines for out and white lines for inner contours. The contour consisting of singe isolated pixels (for example, in the upper-right of (b)) are marked by a single circle in the approriate color.

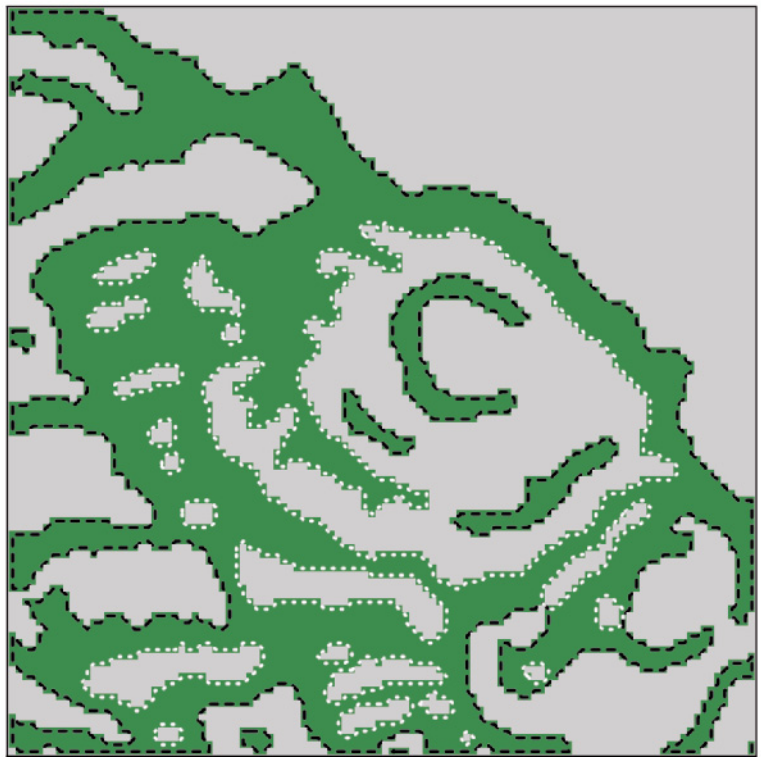


Fig. 11.11
Example of a complex contour (in a section cut from Fig. 10.12). Outer contours are marked in black and inner contours in white.

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.

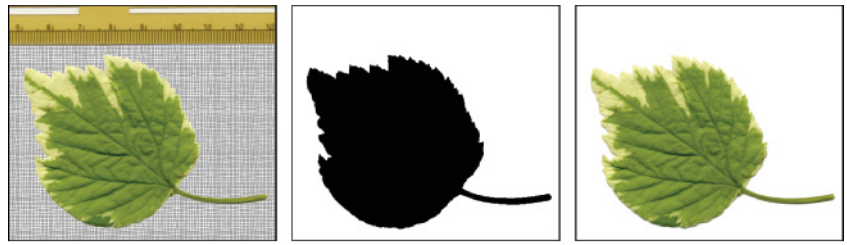


Fig. 11.12
Use of a binary mask to specify a region of an image: original image (a), logical (bit) mask (b), and masked image (c).

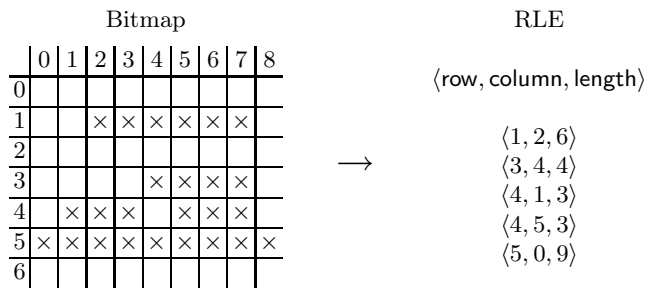


Fig. 11.13
Run length encoding in row direction. A run of pixels can be represented by its starting point $(1, 2)$ and its length (6).

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. 11.12). Since Boolean values can be represented by a single bit, such a matrix is often referred to as a "bitmap".[5]

11.3.2 Run Length Encoding

In *run length encoding* (RLE), sequences of adjacent foreground pixels can be represented compactly as "runs". A run, or contiguous 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,

$$Run_i = \langle \mathsf{row}_i, \mathsf{column}_i, \mathsf{length}_i \rangle,$$

two to represent the starting pixel (row, column) and a third for the length of the run as illustrated in Fig. 11.13. 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.

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

[5] In Java, variables of the type `boolean` are represented internally within the Java virtual machine (JVM) as 32-bit `int`s. There is currently no direct way to implement genuine bitmaps in Java.

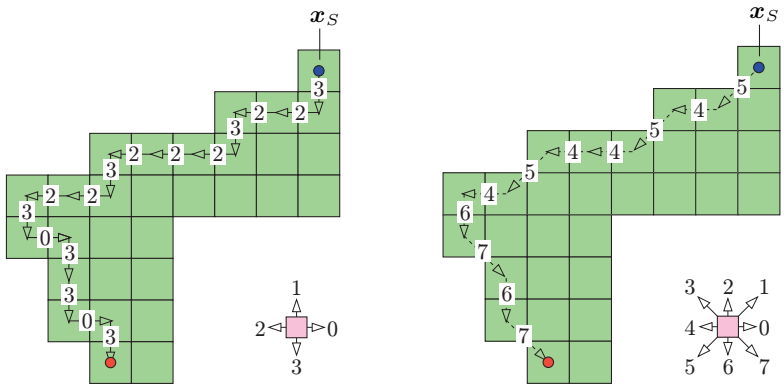


Fig. 11.14
Chain codes with 4- and 8-connected neighborhoods. To compute a chain code, begin traversing the contour from a given starting point $\boldsymbol{x}_S$. Encode the relative position between adjacent contour points using the directional code for either 4-connected (left) or 8-connected (right) 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.

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. 11.4.3).

11.3.3 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 [35], are a classical method of contour encoding. In this encoding, the contour beginning at a given start point $\boldsymbol{x}_S$ is represented by the sequence of directional changes it describes on the discrete image raster (Fig. 11.14).

Absolute chain code

For a closed contour of a region $\mathcal{R}$, described by the sequence of points $\boldsymbol{c}_\mathcal{R} = [\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots \boldsymbol{x}_{M-1}]$ with $\boldsymbol{x}_i = \langle u_i, v_i \rangle$, we create the elements of its chain code sequence $\boldsymbol{c}'_\mathcal{R} = [c'_0, c'_1, \ldots c'_{M-1}]$ by

$$c'_i = \textsc{Code}(\Delta u_i, \Delta v_i), \tag{11.1}$$

$$\text{where} \quad (\Delta u_i, \Delta v_i) = \begin{cases} (u_{i+1} - u_i, v_{i+1} - v_i) & \text{for } 0 \le i < M-1 \\ (u_0 - u_i, v_0 - v_i) & \text{for } i = M-1, \end{cases}$$

and $\textsc{Code}(\Delta u, \Delta v)$ being defined by the following table:[6]

[6] Assuming an 8-connected neighborhood.

Δu	1	1	0	−1	−1	−1	0	1
Δv	0	1	1	1	0	−1	−1	−1
$\text{Code}(\Delta u, \Delta v)$	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 $\boldsymbol{x}_S$, 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 $\boldsymbol{c}'_{\mathcal{R}} = [c'_0, c'_1, \ldots c'_{M-1}]$ can be converted element by element to a *differential* chain code $\boldsymbol{c}''_{\mathcal{R}} = [c''_0, c''_1, \ldots c''_{M-1}]$, with

$$c''_i = \begin{cases} (c'_{i+1} - c'_i) \bmod 8 & \text{for } 0 \le i < M-1 \\ (c'_0 - c'_i) \bmod 8 & \text{for } i = M-1, \end{cases} \tag{11.2}$$

again under the assumption of an 8-connected neighborhood.[7] 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 $\boldsymbol{c}'_{\mathcal{R}}$. For the contour in Fig. 11.14 (b), the results are

$$\begin{aligned} \boldsymbol{c}'_{\mathcal{R}} &= [5, 4, 5, 4, 4, 5, 4, 6, 7, 6, 7, \ldots 2, 2, 2], \\ \boldsymbol{c}''_{\mathcal{R}} &= [7, 1, 7, 0, 1, 7, 2, 1, 7, 1, 1, \ldots 0, 0, 3]. \end{aligned}$$

Given the starting point $\boldsymbol{x}_S$ 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 $\boldsymbol{c}''_1$, $\boldsymbol{c}''_2$, we must first ensure that the same start point was used when computing the codes.

[7] See Appendix B.1.2 for implementing the mod operator used in Eqn. (11.2).

A method that is often used [5,38] 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

$$\begin{aligned}\textsc{Val}(\boldsymbol{c}''_{\mathcal{R}}) &= c''_0 \cdot b^0 + c''_1 \cdot b^1 + \ \ldots \ + c''_{M-1} \cdot b^{M-1} \\ &= \sum_{i=0}^{M-1} c''_i \cdot b^i. \end{aligned} \tag{11.3}$$

Then the sequence $\boldsymbol{c}''_{\mathcal{R}}$ is shifted cyclically until the numeric value of the corresponding number reaches a maximum. We use the expression $\boldsymbol{c}''_{\mathcal{R}} \rhd k$ to denote the sequence $\boldsymbol{c}''_{\mathcal{R}}$ being cyclically shifted by k positions to the right,[8] such as (for $k = 2$)

$$\begin{aligned}\boldsymbol{c}''_{\mathcal{R}} &= [\,0, 1, 3, 2, \ldots 9, 3, 7, 4\,] \\ \boldsymbol{c}''_{\mathcal{R}} \rhd 2 &= [\,7, 4, 0, 1, 3, 2, \ldots 9, 3\,]\end{aligned}$$

and

$$k_{\max} = \arg \max_{0 \le k < M} \textsc{Val}(\boldsymbol{c}''_{\mathcal{R}} \rhd k) \tag{11.4}$$

to denote the shift required to maximize the corresponding arithmetic value. The resulting code sequence or *shape number*,

$$\boldsymbol{s}_{\mathcal{R}} = \boldsymbol{c}''_{\mathcal{R}} \rhd k_{\max}, \tag{11.5}$$

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 Val() in Eqn. (11.3) produces values that are in general too large to be actually computed, in practice the relation

$$\textsc{Val}(\boldsymbol{c}''_1) > \textsc{Val}(\boldsymbol{c}''_2)$$

is determined by comparing the *lexicographic ordering* between the sequences $\boldsymbol{c}''_1$ and $\boldsymbol{c}''_2$ 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 ($\neq 90°$) 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 11.4 presents a number of tools that are more appropriate in these types of cases.

Fourier descriptors

An elegant approach to describing contours are so-called Fourier descriptors, which interpret the two-dimensional contour $\boldsymbol{c}_{\mathcal{R}} = [\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots$

[8] $(\boldsymbol{c}''_{\mathcal{R}} \rhd k)[\,i\,] = \boldsymbol{c}''_{\mathcal{R}}[(i-k) \bmod M]$.

$\boldsymbol{x}_{M-1}]$ with $\boldsymbol{x}_i = (u_i, v_i)$ as a sequence of values $[z_0, z_1 \dots z_{M-1}]$ in the complex plane, where

$$z_i = (u_i + \mathrm{i} \cdot v_i) \in \mathbb{C}. \tag{11.6}$$

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 one-dimensional *Fourier spectrum* (see Sec. 13.3) 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 [38, 41, 59, 60, 95].

11.4 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 [27, 75, 98].

11.4.1 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 value 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*; 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.

11.4.2 Geometric Features

A region $\mathcal{R}$ of a binary image can be interpreted as a two-dimensional distribution of foreground points $\boldsymbol{x}_i = (u_i, v_i)$ within the discrete plane $\mathbb{Z}^2$,

$$\mathcal{R} = \{\boldsymbol{x}_0, \boldsymbol{x}_1 \dots \boldsymbol{x}_{N-1}\} = \{(u_0, v_0), (u_1, v_1) \dots (u_{N-1}, v_{N-1})\}.$$

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. 11.1, 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. 11.14, 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 vertical segments with 1 and diagonal segments with $\sqrt{2}$. Given an 8-connected chain code $\boldsymbol{c}'_{\mathcal{R}} = [c'_0, c'_1, \dots c'_{M-1}]$, the perimeter of the region is arrived at by

$$\mathsf{Perimeter}(\mathcal{R}) = \sum_{i=0}^{M-1} \mathrm{length}(c'_i) \tag{11.7}$$

$$\text{with} \quad \mathrm{length}(c) = \begin{cases} 1 & \text{for } c = 0, 2, 4, 6, \\ \sqrt{2} & \text{for } c = 1, 3, 5, 7. \end{cases}$$

However, with this conventional method of calculation,[9] the *real* perimeter ($P(\mathcal{R})$) is systematically overestimated. As a simple remedy, a general correction factor of 0.95 works satisfactory even for relatively small regions:

$$P(\mathcal{R}) \approx \mathsf{Perimeter}_{\mathrm{corr}}(\mathcal{R}) = 0.95 \cdot \mathsf{Perimeter}(\mathcal{R}). \tag{11.8}$$

[9] The function used in ImageJ's Analyze menu uses this method of perimeter computation.

Area

The area of a region $\mathcal{R}$ can be found by simply counting the image pixels that make up the region,

$$A(\mathcal{R}) = |\mathcal{R}| = N. \tag{11.9}$$

The area of a connected region without holes can also be approximated from its closed contour, defined by M coordinate points $(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots \boldsymbol{x}_{M-1})$, where $\boldsymbol{x}_i = (u_i, v_i)$, using the Gaussian area formula for polygons:

$$A(\mathcal{R}) \approx \frac{1}{2} \cdot \left| \sum_{i=0}^{M-1} \big(u_i \cdot v_{(i+1) \bmod M} \; - \; u_{(i+1) \bmod M} \cdot v_i\big) \right|. \tag{11.10}$$

When the contour is already encoded as a chain code $\boldsymbol{c}'_{\mathcal{R}} = [c'_0, c'_1, \ldots c'_{M-1}]$, then the region's area can be computed using Eqn. (11.10) by expanding $\boldsymbol{c}'_{\mathcal{R}}$ into a sequence of contour points, using an arbitrary starting point (e. g., $(0, 0)$).

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 described below, 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,

$$\mathsf{Circularity}(\mathcal{R}) = \; 4\pi \cdot \frac{A(\mathcal{R})}{P^2(\mathcal{R})}, \tag{11.11}$$

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. 11.15). If an absolute value for a region's roundness is required, the corrected perimeter estimate (Eqn. (11.8)) should be employed:

$$\mathsf{Circularity}(\mathcal{R}) \approx \; 4\pi \cdot \frac{A(\mathcal{R})}{\mathsf{Perimeter}^2_{\mathrm{corr}}(\mathcal{R})}. \tag{11.12}$$

Figure 11.15 shows the circularity values of different regions as computed with the formulation in Eqn. (11.12).

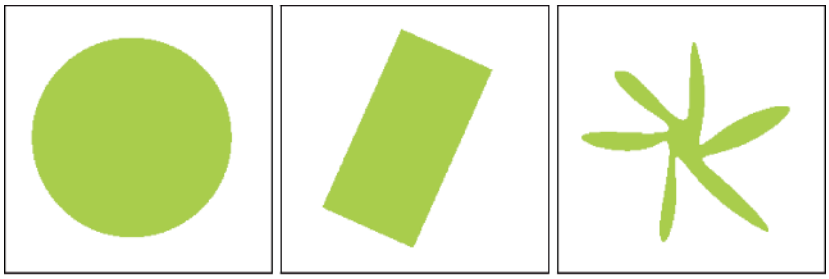


Fig. 11.15
Circularity values for different shapes. Shown are the corresponding $\mathsf{Circularity}(\mathcal{R})$ estimates as defined in Eqn. (11.12).

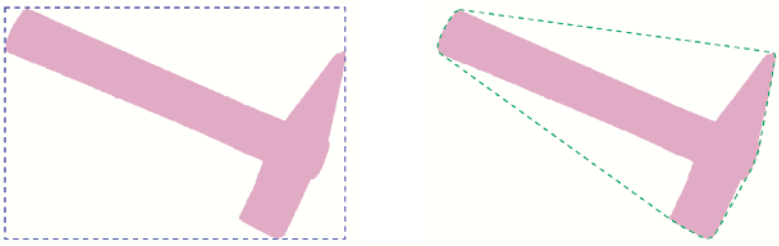


Fig. 11.16
Example bounding box (a) and convex hull (b) of a binary image region.

Bounding box

The bounding box of a region $\mathcal{R}$ is the minimal axis-parallel rectangle that encloses all points of $\mathcal{R}$,

$$\mathsf{BoundingBox}(\mathcal{R}) = \langle u_{\min}, u_{\max}, v_{\min}, v_{\max} \rangle, \tag{11.13}$$

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. 11.16 (a)).

Convex hull

The convex hull is the smallest polygon within which all points in the region fit. 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 (Fig. 11.16 (b)). The convex hull can be computed digitally using the *QuickHull* algorithm [6] for N contour points in time complexity $\mathcal{O}(NH)$, where H is the number of points in the polygon of the resulting convex hull.[10]

The convex hull is useful, for example, for determining the convexity or the *density* of a region. The *convexity* is defined as the relationship

[10] For $\mathcal{O}()$ complexity notation, see Appendix A (p. 454).

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.

11.4.3 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*.

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.[11]

The centroid $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y})$ of a binary (not necessarily connected) region is the arithmetic mean of the coordinates in the x and y directions,

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} u \qquad \text{and} \qquad \bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} v \,. \tag{11.14}$$

Moments

The formulation of the region's centroid in Eqn. (11.14) is only a special case of the more general statistical concept of a *moment*. Specifically, the expression

$$m_{pq} = \sum_{(u,v)\in\mathcal{R}} I(u,v) \cdot u^p v^q \tag{11.15}$$

describes the (ordinary) moment of the 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 computed directly from the coordinates of the contour points [89, 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. (11.15) can be simplified to

[11] 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.

$$m_{pq} = \sum_{(u,v)\in\mathcal{R}} u^p v^q. \tag{11.16}$$

In this way, the *area* of a binary region can be expressed as the *zero*-order moment,

$$A(\mathcal{R}) = |\mathcal{R}| = \sum_{(u,v)\in\mathcal{R}} 1 = \sum_{(u,v)\in\mathcal{R}} u^0 v^0 = m_{00}(\mathcal{R}), \tag{11.17}$$

and similarly the *centroid* $\bar{\boldsymbol{x}}$ Eqn. (11.14) as

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} u^1 v^0 = \frac{m_{10}(\mathcal{R})}{m_{00}(\mathcal{R})}, \tag{11.18}$$

$$\bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} u^0 v^1 = \frac{m_{01}(\mathcal{R})}{m_{00}(\mathcal{R})}. \tag{11.19}$$

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.

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 $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y})$ to obtain the *central* moments of order p, q:

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} I(u,v) \cdot (u-\bar{x})^p \cdot (v-\bar{y})^q. \tag{11.20}$$

For a binary image (with $I(u,v) = 1$ within the region $\mathcal{R}$), Eqn. (11.20) can be simplified to

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} (u-\bar{x})^p \cdot (v-\bar{y})^q. \tag{11.21}$$

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

$$s^{(p+q+2)}. \tag{11.22}$$

Thus size-invariant (normalized) moments are obtained by normalizing with the reciprocal of the area $\mu_{00} = m_{00}$ raised to the required power in the form

$$\bar{\mu}_{pq}(\mathcal{R}) = \mu_{pq} \cdot \left(\frac{1}{\mu_{00}(\mathcal{R})}\right)^{(p+q+2)/2} \tag{11.23}$$

for $(p + q) \geq 2$ [59, p. 529].

Program 11.3 gives a direct (brute force) conversion of the calculations for ordinary, central, and normalized central moments in Java for binary images (`BACKGROUND = 0`). This implementation is only meant to clarify the computation, and naturally much more efficient implementations are possible (for example, [61]).

11.4.4 Moment-Based Geometrical Properties

While normalized moments can be directly applied for classifying regions, further interesting and geometrically relevant features can be elegantly derived from moments.

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. 11.18 (a)). Since rotating the region around the major axis requires less effort (least 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), the pencil exhibits the least mass inertia (Fig. 11.17). 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

$$\tan(2\,\theta_{\mathcal{R}}) = \frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})} \tag{11.24}$$

and therefore

$$\theta_{\mathcal{R}} = \frac{1}{2}\tan^{-1}\left(\frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})}\right). \tag{11.25}$$

The resulting angle $\theta_{\mathcal{R}}$ is in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.[12] Orientation measurements based on region moments are very accurate in general.

[12] See Appendix B.1.6 concerning the computation of angles in Java using `Math.atan2()`.

Program 11.3
Example of directly computing moments in Java. The methods `moment()`, `centralMoment()`, and `normalCentralMoment()` compute for a binary image the moments m_{pq}, μ_{pq}, and $\bar{\mu}_{pq}$ (Eqns. (11.16), (11.21), and (11.23)).

```
1  import ij.process.ImageProcessor;
2
3  public class Moments {
4    static final int BACKGROUND = 0;
5
6    static double moment(ImageProcessor ip,int p,int q) {
7      double Mpq = 0.0;
8      for (int v = 0; v < ip.getHeight(); v++) {
9        for (int u = 0; u < ip.getWidth(); u++) {
10         if (ip.getPixel(u,v) != BACKGROUND) {
11           Mpq += Math.pow(u, p) * Math.pow(v, q);
12         }
13       }
14     }
15     return Mpq;
16   }
17   static double centralMoment(ImageProcessor ip,int p,int q)
18     {
19     double m00 = moment(ip, 0, 0); // region area
20     double xCtr = moment(ip, 1, 0) / m00;
21     double yCtr = moment(ip, 0, 1) / m00;
22     double cMpq = 0.0;
23     for (int v = 0; v < ip.getHeight(); v++) {
24       for (int u = 0; u < ip.getWidth(); u++) {
25         if (ip.getPixel(u,v) != BACKGROUND) {
26           cMpq +=
27             Math.pow(u - xCtr, p) *
28             Math.pow(v - yCtr, q);
29         }
30       }
31     }
32     return cMpq;
33   }
34   static double normalCentralMoment
35                         (ImageProcessor ip,int p,int q) {
36     double m00 = moment(ip, 0, 0);
37     double norm = Math.pow(m00, (double)(p + q + 2) / 2);
38     return centralMoment(ip, p, q) / norm;
39   }
40
41 } // end of class Moments
```

Computing orientation vectors

When visualizing region properties, a frequent task is to plot the region's orientation as a line or arrow, that are usually anchored at the center of gravity $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y})$; for example, by a parametric line of the form

$$\boldsymbol{x} = \bar{\boldsymbol{x}} + \lambda \cdot \boldsymbol{x}_d = \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} + \lambda \cdot \begin{pmatrix} \cos(\theta_{\mathcal{R}}) \\ \sin(\theta_{\mathcal{R}}) \end{pmatrix}, \tag{11.26}$$

Fig. 11.17 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.

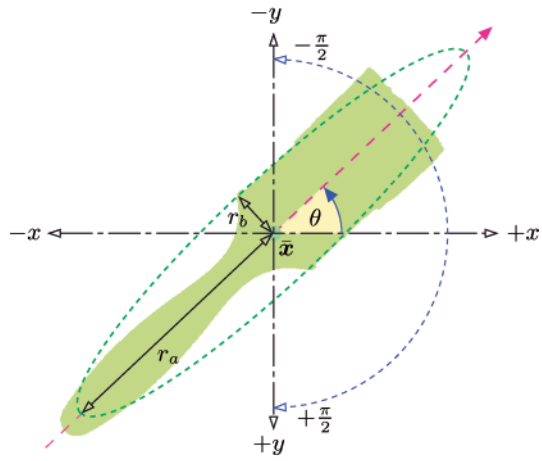


Fig. 11.18 Region orientation and eccentricity. The major axis of the region extends through its center of gravity $\bar{\boldsymbol{x}}$ at the orientation θ. Note that angles are in the range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ and increment in the *clockwise* direction because the y axis of the image coordinate system points downward (in this example, $\theta \approx -0.759 \approx -43.5°$). 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.

for some length $\lambda > 0$. To find the unit orientation vector $\boldsymbol{x}_d = (\cos\theta, \sin\theta)^T$, we could first compute the inverse tangent to get 2θ (Eqn. (11.25)) and then compute the cosine and sine of θ. However, the vector $\boldsymbol{x}_d$ can also be obtained without using trigonometric functions as follows. Rewriting Eqn. (11.24) as

$$\tan(2\theta_{\mathcal{R}}) = \frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})} = \frac{A}{B} = \frac{\sin(2\theta_{\mathcal{R}})}{\cos(2\theta_{\mathcal{R}})} \tag{11.27}$$

we get (by Pythagoras theorem)

$$\sin(2\theta_{\mathcal{R}}) = \frac{A}{\sqrt{A^2+B^2}} \quad \text{and} \quad \cos(2\theta_{\mathcal{R}}) = \frac{B}{\sqrt{A^2+B^2}},$$

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 region's orientation vector $\boldsymbol{x}_d = (x_d, y_d)^T$ as

$$x_d = \cos(\theta_{\mathcal{R}}) = \begin{cases} 0 & \text{for } A = B = 0 \\ \left[\frac{1}{2}\left(1 + \frac{B}{\sqrt{A^2+B^2}}\right)\right]^{\frac{1}{2}} & \text{otherwise,} \end{cases} \tag{11.28}$$

$$y_d = \sin(\theta_{\mathcal{R}}) = \begin{cases} 0 & \text{for } A = B = 0 \\ \left[\frac{1}{2}\left(1 - \frac{B}{\sqrt{A^2+B^2}}\right)\right]^{\frac{1}{2}} & \text{for } A \geq 0 \\ -\left[\frac{1}{2}\left(1 - \frac{b}{\sqrt{A^2+B^2}}\right)\right]^{\frac{1}{2}} & \text{for } A < 0, \end{cases} \tag{11.29}$$

straight from the central region moments $\mu_{11}(\mathcal{R})$, $\mu_{20}(\mathcal{R})$, and $\mu_{02}(\mathcal{R})$, as defined in Eqn. (11.27). The horizontal component (x_d) in Eqn. (11.28) is always positive, while the case switch in Eqn. (11.29) corrects the sign of the vertical component (y_d) to map to the same angular range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ as Eqn. (11.25). The resulting vector $\boldsymbol{x}_d$ 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 below.

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. (11.25)), 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. 11.4.2), and they can also handle nonconnected regions or point clouds. Several different formulations of region eccentricity can be found in the literature [5,59,60] (see also Exercise 11.11). We adopt the following definition because of its simple geometrical interpretation:

$$\mathsf{Ecc}(\mathcal{R}) = \frac{a_1}{a_2} = \frac{\mu_{20} + \mu_{02} + \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}}{\mu_{20} + \mu_{02} - \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}}, \tag{11.30}$$

where $a_1 = 2\lambda_1$, $a_2 = 2\lambda_2$ are multiples of the eigenvalues λ_1, λ_2 of the symmetric 2×2 matrix

$$\boldsymbol{A} = \begin{pmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{pmatrix}$$

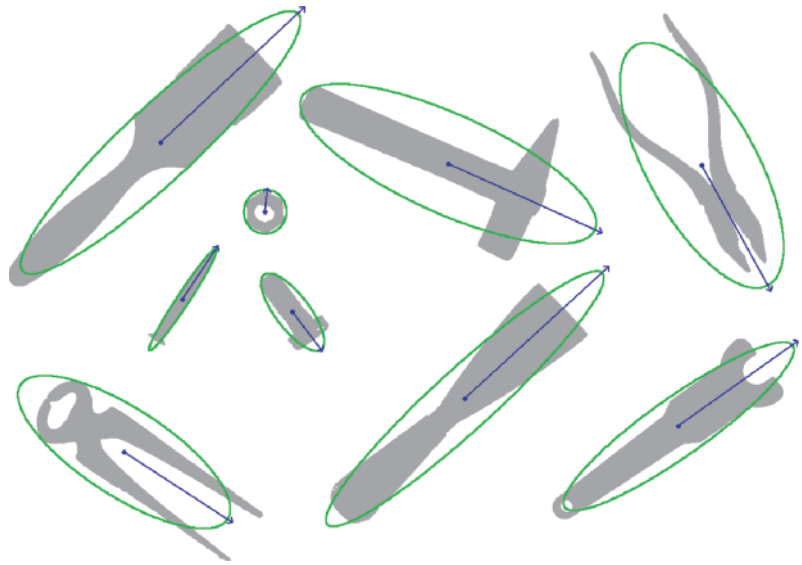


Fig. 11.19
Orientation and eccentricity examples. The orientation θ (Eqn. (11.25)) is displayed for each connected region as a vector with the length proportional to the region's eccentricity value $\mathsf{Ecc}(\mathcal{R})$ (Eqn. (11.30)). Also shown are the ellipses (Eqns. (11.31) and (11.32)) corresponding to the orientation and eccentricity parameters.

formed by the central moments μ_{pq} of the region $\mathcal{R}$. The values of Ecc are in the range $[1, \infty)$, where $\mathsf{Ecc} = 1$ corresponds to a circular disk and elongated regions have values > 1. Ecc itself is invariant to the region's orientation and size. However, the values a_1, a_2 contain information about the spatial extent 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. (11.25). The lengths of the major and minor axes, r_a and r_b, are

$$r_a = 2 \cdot \left(\frac{\lambda_1}{|\mathcal{R}|}\right)^{\frac{1}{2}} = \left(\frac{2\,a_1}{|\mathcal{R}|}\right)^{\frac{1}{2}}, \tag{11.31}$$

$$r_b = 2 \cdot \left(\frac{\lambda_2}{|\mathcal{R}|}\right)^{\frac{1}{2}} = \left(\frac{2\,a_2}{|\mathcal{R}|}\right)^{\frac{1}{2}}, \tag{11.32}$$

respectively, with a_1, a_2 as defined in Eqn. (11.30) and $|\mathcal{R}|$ being the number of pixels in the region. The resulting parametric equation of this ellipse is

$$\begin{aligned}\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} &= \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} + \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} r_a \cdot \cos(t) \\ r_b \cdot \sin(t) \end{pmatrix} \\ &= \begin{pmatrix} \bar{x} + \cos(\theta) \cdot r_a \cdot \cos(t) - \sin(\theta) \cdot r_b \cdot \sin(t) \\ \bar{y} + \sin(\theta) \cdot r_a \cdot \cos(t) + \cos(\theta) \cdot r_b \cdot \sin(t) \end{pmatrix}\end{aligned} \tag{11.33}$$

for $0 \le t < 2\pi$. If entirely *filled*, the region described by this ellipse would have the same central moments as the original region $\mathcal{R}$. Figure 11.19 shows a set of regions with overlaid orientation and eccentricity results.

Invariant 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. A classical solution to this problem is a clever combination of simpler features known as "Hu's Moments" [48]:[13]

$$
\begin{aligned}
H_1 &= \bar{\mu}_{20} + \bar{\mu}_{02}, \qquad (11.34)\\
H_2 &= (\bar{\mu}_{20} - \bar{\mu}_{02})^2 + 4\,\bar{\mu}_{11}^2,\\
H_3 &= (\bar{\mu}_{30} - 3\,\bar{\mu}_{12})^2 + (3\,\bar{\mu}_{21} - \bar{\mu}_{03})^2,\\
H_4 &= (\bar{\mu}_{30} + \bar{\mu}_{12})^2 + (\bar{\mu}_{21} + \bar{\mu}_{03})^2,\\
H_5 &= (\bar{\mu}_{30} - 3\,\bar{\mu}_{12}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot \left[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3(\bar{\mu}_{21} + \bar{\mu}_{03})^2\right]\\
&\quad + (3\,\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot \left[3\,(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2\right],\\
H_6 &= (\bar{\mu}_{20} - \bar{\mu}_{02}) \cdot \left[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2\right]\\
&\quad + 4\,\bar{\mu}_{11} \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}),\\
H_7 &= (3\,\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot \left[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3\,(\bar{\mu}_{21} + \bar{\mu}_{03})^2\right]\\
&\quad + (3\,\bar{\mu}_{12} - \bar{\mu}_{30}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot \left[3\,(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2\right].
\end{aligned}
$$

In practice, the logarithm of the results (that is, $\log(H_k)$) is used since the raw values can 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 grayscale images; for further information, see [38, p. 517].

11.4.5 Projections

Image projections are one-dimensional representations of the image contents, usually computed parallel to the coordinate axis; in this case, the horizontal, as well as the veritcal, projection of an image $I(u, v)$, with $0 \le u < M$, $0 \le v < N$, defined as

$$P_{\text{hor}}(v_0) = \sum_{u=0}^{M-1} I(u, v_0) \qquad \text{for } 0 < v_0 < N, \qquad (11.35)$$

$$P_{\text{ver}}(u_0) = \sum_{v=0}^{N-1} I(u_0, v) \qquad \text{for } 0 < u_0 < M. \qquad (11.36)$$

The *horizontal* projection $P_{\text{hor}}(v_0)$ (Eqn. (11.35)) is the sum of the pixel values in the image *row* v_0 and has length N corresponding to the height

[13] In order to improve the legibility of Eqn. (11.34) 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.

Program 11.4 Computation 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 a single traversal of the image. The projections are represented by the one-dimensional arrays `horProj` and `verProj` with elements of type `int`.

```
1  public void run(ImageProcessor ip) {
2    int M = ip.getWidth();
3    int N = ip.getHeight();
4    int[] horProj = new int[N];
5    int[] verProj = new int[M];
6    for (int v = 0; v < N; v++) {
7      for (int u = 0; u < M; u++) {
8        int p = ip.getPixel(u, v);
9        horProj[v] += p;
10       verProj[u] += p;
11     }
12   }
13   // use projections horProj, verProj now
14   // ...
15 }
```

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. (11.36)). 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 Prog. 11.4 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. 11.20). In practice, especially to account for document skew, projections are often computed along the major axis of an image region Eqn. (11.25). 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.

11.4.6 Topological Properties

Topological features do not describe the shape of a region in continuous terms; instead, they capture its structural properties. Topological properties are typically invariant even under strong image transformations. The convexity of a region, which can be calculated from the convex hull (Sec. 11.4.2), 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. 11.2.2.

Fig. 11.20
Example of the horizontal projection (right) and vertical projection (bottom) of a binary image.

A feature that can be derived 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,

$$N_E(\mathcal{R}) = N_R(\mathcal{R}) - N_L(\mathcal{R}). \tag{11.37}$$

When dealing with single connected regions, the formula simplifies to $1-N_L$. So, for a picture of the number "**8**", for example, $N_E = 1-2 = -1$ and for the letter "**D**" $N_E = 1-1 = 0$.

Topological features are often used in combination with numerical features for classification. A classic example of this combination is OCR (optical character recognition) [16].

11.5 Exercises

Exercise 11.1. Trace, by hand, the execution of both variations (*depth-first* and *breadth-first*) of the flood-fill algorithm using the following image region and starting at coordinates $(5, 1)$:

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	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	1	0
0	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	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 Background

1 Foreground

Exercise 11.2. The implementation of the flood-fill algorithm in Prog. 11.1 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 above. Modify the *depth-first* and *breadth-first* variants given in Prog. 11.1 accordingly and compare the new running times.

Exercise 11.3. Implement an ImageJ plugin that encodes a grayscale image using run length encoding (Sec. 11.3.2) and stores it in a file. Develop a second plugin that reads the file and reconstructs the image.

Exercise 11.4. 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 11.5. 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 11.6. 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.

Exercise 11.7. Implement an algorithm for comparing contours using their shape numbers Eqn. (11.3). 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 11.8. Using Eqn. (11.10) as the basis, develop and implement an algorithm that computes the area of a region from its 8-chain code encoded contour. What type of discrepancy from the region's actual area (the number of pixels it contains) do you expect?

Exercise 11.9. Sketch an example binary region where the centroid lies outside of the region.

Exercise 11.10. Implement the moment features developed by Hu (Eqn. (11.34)) and show that they are invariant under scaling and rotation for both binary and grayscale images.

Exercise 11.11. There are alternative definitions for the eccentricity of a region Eqn. (11.30); for example,

$$\mathsf{Ecc}_2(\mathcal{R}) = \frac{\big(\mu_{20} - \mu_{02}\big)^2 + 4 \cdot \mu_{11}^2}{\big(\mu_{20} + \mu_{02}\big)^2} \qquad \text{[60, p. 394]},$$

$$\mathsf{Ecc}_3(\mathcal{R}) = \frac{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}}{m_{00}} \qquad \text{[59, p. 531]},$$

$$\mathsf{Ecc}_4(\mathcal{R}) = \frac{\sqrt{\mu_{20} - \mu_{02}} + 4 \cdot \mu_{11}}{m_{00}} \qquad \text{[5, p. 255]}.$$

Implement all four variations (including the one in Eqn. (11.30)) and contrast the results using suitably designed regions. Determine how these measures work and what their range of values is, and propose a geometrical interpretation for each.

Exercise 11.12. 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. 11.19). Hint: Use Eqn. (11.33) to develop a method for drawing ellipses at arbitrary orientations (not available in ImageJ).

Exercise 11.13. The Java method in Prog. 11.4 computes an image's horizontal and vertical projections. For document image processing, projections in the diagonal directions are also useful. Implement these projections and consider what role they play in document image analysis.

12

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 interest of scientists, psychologists, philosophers, and artists for hundreds of years [88, 92]. 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.

12.1 RGB Color Images

The RGB color schema encodes colors as combinations of the three primary colors: red (R), green (G), and blue (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 CRT[1]-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 three-dimensional 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 **S** and **W** are shades of gray created from equal color components $R = G = B$.

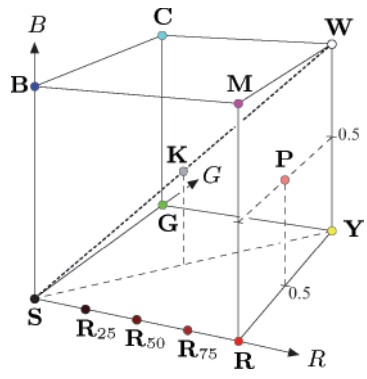


Fig. 12.1
Representation of the RGB color space as a three-dimensional unit cube. The primary colors red (R), green (G), and blue (B) form the coordinate system. The "pure" red color (**R**), green (**G**), blue (**B**), cyan (**C**), magenta (**M**), and yellow (**Y**) lie on the vertices of the color cube. All the shades of gray, of which **K** is an example, lie on the diagonal between black **S** and white **W**.

		RGB Value		
Point	Color	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
$\mathbf{R}_{75}$	75% Red	0.75	0.00	0.00
$\mathbf{R}_{50}$	50% Red	0.50	0.00	0.00
$\mathbf{R}_{25}$	25% Red	0.25	0.00	0.00
P	Pink	1.00	0.50	0.50

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. In this color space, 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 sense. While the simple RGB color system does not permit us to answer questions like these, we will return to them when we cover the CIE color space (Sec. 12.3.1).

[1] Cathode ray tube.

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 violet and therefore displays correspondingly higher values in its B channel.

12.1.1 Organization 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 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. Selecting which type of image to use depends on the requirements of the application.

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*.

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

$$I = \langle I_R, I_G, I_B \rangle$$

can be thought of as a group of related intensity images I_R, I_G, and I_B (Fig. 12.3), and the RGB component values of the color image I at position (u, v) are obtained by accessing all three intensity images as follows:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} \leftarrow \begin{pmatrix} I_R(u,v) \\ I_G(u,v) \\ I_B(u,v) \end{pmatrix}. \tag{12.1}$$

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) so that

$$I(u, v) = \langle R, G, B \rangle.$$

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 follows:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} \leftarrow \begin{pmatrix} \text{Red}(I(u,v)) \\ \text{Green}(I(u,v)) \\ \text{Blue}(I(u,v)) \end{pmatrix}. \tag{12.2}$$

The access functions, Red(), Green(), Blue(), will depend on the specific implementation used for encoding the color pixels.

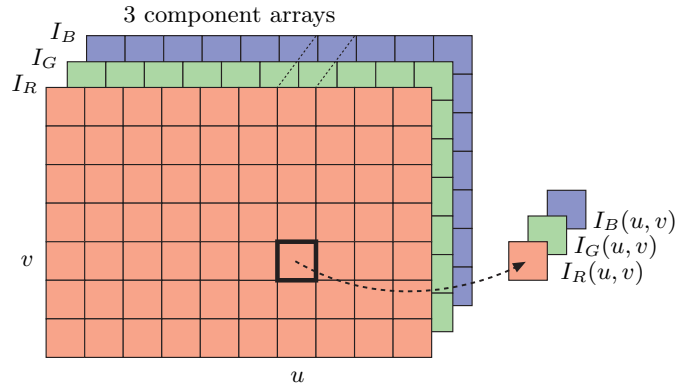


Fig. 12.3 RGB color image in component ordering. The three color components are laid out in separate arrays I_R, I_G, I_B of the same size.

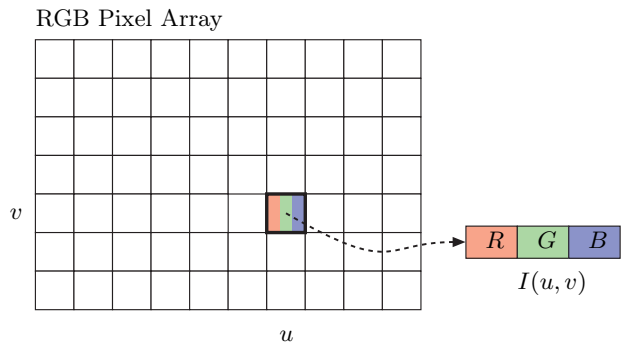


Fig. 12.4
RGB-color image using packed ordering. The three color components R, G, and B are placed together in a single array element.

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 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"

$$P[k] = (P_R[k], P_G[k], P_B[k])\,,$$

for $k = 0 \ldots N-1$ (Fig. 12.5). N is the size of the color table and therefore also the maximum number of distinct image colors (typically $N = 2$ to 256). Since the color table can contain any RGB color value $\langle P_R, P_G, P_B \rangle$, it must be saved as part of the image. The RGB component values of an indexed image I_{idx} at location (u, v) are obtained as

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} \leftarrow \begin{pmatrix} P_R[k] \\ P_G[k] \\ P_B[k] \end{pmatrix} = \begin{pmatrix} r_k \\ g_k \\ b_k \end{pmatrix}, \tag{12.3}$$

with $k = I_{\text{idx}}(u, v)$. During the transformation from a true color image to an indexed image (for example, from a JPEG image to a GIF image),

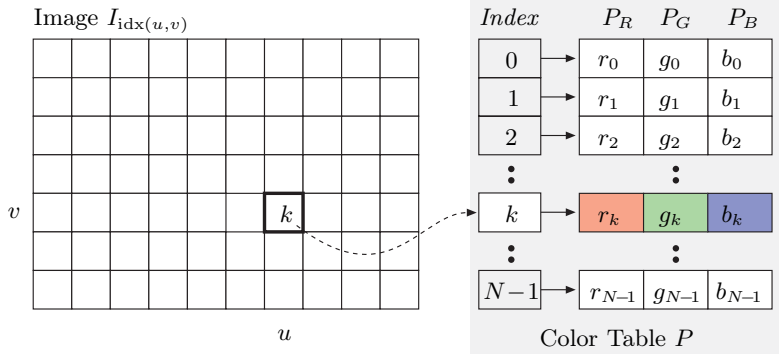


Fig. 12.5
RGB indexed image. Instead of a full color value, each pixel in $I_{\text{idx}(u,v)}$ contains an index k. The color value for each k is defined by an entry in the color table or "palette" $P[k]$.

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 the color quantization section (Sec. 12.5).

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 to 255. The remaining 8 bits are reserved for the transparency,[2] or *alpha* (α), component. This is also the usual ordering in Java[3] for RGB color images.

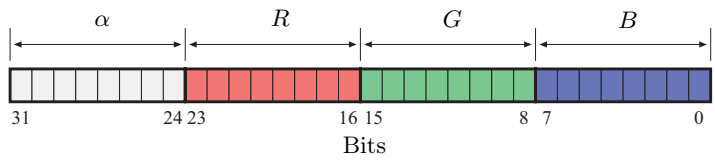


Fig. 12.6 Construction of an RGB color pixel in Java. Within a 32-bit `int`, 8 bits are allocated, in the following order, for each of the color components R, G, B as well as the transparency α (unused in ImageJ).

Accessing RGB pixel values

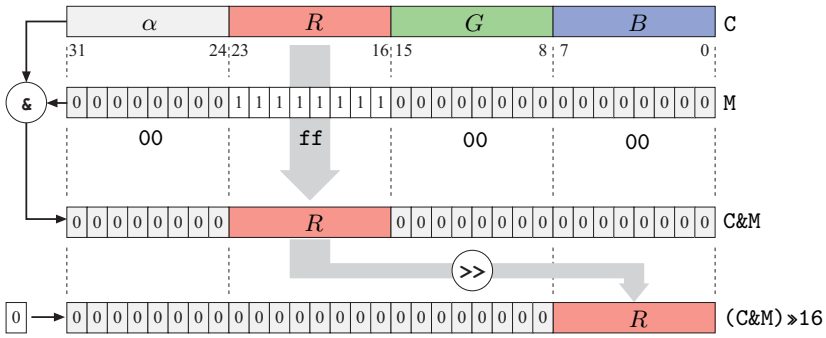
RGB color images are represented by an array of pixels, the elements of which are standard Java `int`s. To disassemble the packed `int` value into the three color components, you apply the appropriate bitwise shifting and masking operations. In the following example, we assume that the image processor `ip` contains an RGB color image:

```
1 int c = ip.getPixel(u,v); // a color pixel
2 int r = (c & 0xff0000) >> 16; // red value
3 int g = (c & 0x00ff00) >> 8;  // green value
4 int b = (c & 0x0000ff);       // blue value
```

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

[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).



(following convention, bit masks are given in hexadecimal[4] notation), and afterwards the extracted bits are shifted right 16 (for R) or 8 (for G) bit positions (see Fig. 12.7).

The "construction" of an RGB pixel from the individual R, G, and B values is done in the opposite direction using the bitwise OR operator (`|`) and shifting the bits left (`«`):

```
int r = 169; // red value
int g = 212; // green value
int b = 17; // blue value
int c = ((r & 0xff)<<16) | ((g & 0xff)<<8) | b & 0xff;
ip.putPixel(u,v,C);
```

Masking the component values with `0xff` works in this case because except for the bits in positions 0 to 7 (values in the range 0 to 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 (see also Sec. B.1.3).

The ImageJ class `ColorProcessor` provides an easy to use alternative which returns the separated RGB components (as an `int` array with three elements). In the following example that demonstrates its use, `ip` is of type `ColorProcessor`:

```
int[] RGB = new int[3];
...
RGB = ip.getPixel(u,v,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

[4] The mask `0xff0000` is of type `int` and represents the 32-bit binary pattern `00000000111111110000000000000000`.

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 = 0xff0000`. 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 (`»`), so that the R component is moved into the lowest 8 bits and its value lies in the range of 0 to 255. During the shift operation, zeros are filled in from the left.

Program 12.1
Working with RGB color images using bit operations (ImageJ plugin, variant 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 17), the separation of color components using bit operations (lines 19–21), and the reassembly of color pixels after modification (line 28). 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 10).

```
1  // File Brighten_Rgb_1.java
2
3  import ij.ImagePlus;
4  import ij.plugin.filter.PlugInFilter;
5  import ij.process.ImageProcessor;
6
7  public class Brighten_Rgb_1 implements PlugInFilter {
8
9    public int setup(String arg, ImagePlus im) {
10     return DOES_RGB; // this plugin works on RGB images
11   }
12
13   public void run(ImageProcessor ip) {
14     int[] pixels = (int[]) ip.getPixels();
15
16     for (int i = 0; i < pixels.length; i++) {
17       int c = pixels[i];
18       // split the color pixel into RGB components
19       int r = (c & 0xff0000) >> 16;
20       int g = (c & 0x00ff00) >> 8;
21       int b = (c & 0x0000ff);
22       // modify colors
23       r = r + 10; if (r > 255) r = 255;
24       g = g + 10; if (g > 255) g = 255;
25       b = b + 10; if (b > 255) b = 255;
26       // reassemble the color pixel and insert into pixel array
27       pixels[i]
28           = ((r & 0xff)<<16) | ((g & 0xff)<<8) | b & 0xff;
29     }
30   }
31
32 } // end of class Brighten_Rgb_1
```

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 to 255 range. Without this test, arithmetic overflow failures 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 variant in Prog. 12.1).

Opening and saving RGB images

ImageJ supports the following types of image formats for RGB true color images:

- **TIFF** (only uncompressed): 3×8-bit RGB. TIFF color images with 16-bit depth are opened as an image stack consisting of three 16-bit intensity images.
- **BMP**, **JPEG**: 3×8-bit RGB.

Program 12.2
Working with RGB color images without bit operations (ImageJ plugin, variant 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 22 and 26, respectively). The running time, because of the method calls, is approximately four times higher than that of variant 1 (Prog. 12.1).

```
1  // File Brighten_Rgb_2.java
2
3  import ij.ImagePlus;
4  import ij.plugin.filter.PlugInFilter;
5  import ij.process.ColorProcessor;
6  import ij.process.ImageProcessor;
7
8  public class Brighten_Rgb_2 implements PlugInFilter {
9    static final int R = 0, G = 1, B = 2; // component indices
10
11   public int setup(String arg, ImagePlus im) {
12     return DOES_RGB; // this plugin works on RGB images
13   }
14
15   public void run(ImageProcessor ip) {
16     //make sure the image is of type ColorProcessor
17     ColorProcessor cp = (ColorProcessor) ip;
18     int[] RGB = new int[3];
19
20     for (int v = 0; v < cp.getHeight(); v++) {
21       for (int u = 0; u < cp.getWidth(); u++) {
22         cp.getPixel(u, v, RGB);
23         RGB[R] = Math.min(RGB[R]+10, 255); // add 10 and
24         RGB[G] = Math.min(RGB[G]+10, 255); // limit to 255
25         RGB[B] = Math.min(RGB[B]+10, 255);
26         cp.putPixel(u, v, RGB);
27       }
28     }
29   }
30
31 } // end of class Brighten_Rgb_2
```

- **PNG** (read only): 3×8-bit 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 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:

```
1 int w = 640, h = 480;
2 ColorProcessor cip = new ColorProcessor(w, h);
3 ImagePlus cimg = new ImagePlus("My New Color Image", cip);
4 cimg.show();
```

When needed, the color image can be displayed by creating an instance of the class `ImagePlus` (line 3) and calling its `show()` method. Since `cip`

is of type `ColorProcessor`, the resulting `ImagePlus` object `cimg` is also a color image. The following code segment demonstrates how this could be verified:

```
5 if (cimg.getType()==ImagePlus.COLOR_RGB) {
6   int b = cimg.getBitDepth(); // b = 24
7   IJ.write("this is an RGB color image with " + b + " bits");
8 }
```

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 following types of image formats for indexed images:

- **GIF**: index values with 1 to 8 bits (2 to 256 colors), 3×8-bit color values.
- **PNG** (read only): index values with 1 to 8 Bits (2 to 256 colors), 3×8-bit color values.
- **BMP**, **TIFF** (uncompressed): index values with 1 to 8 bits (2 to 256 colors), 3×8-bit color values.

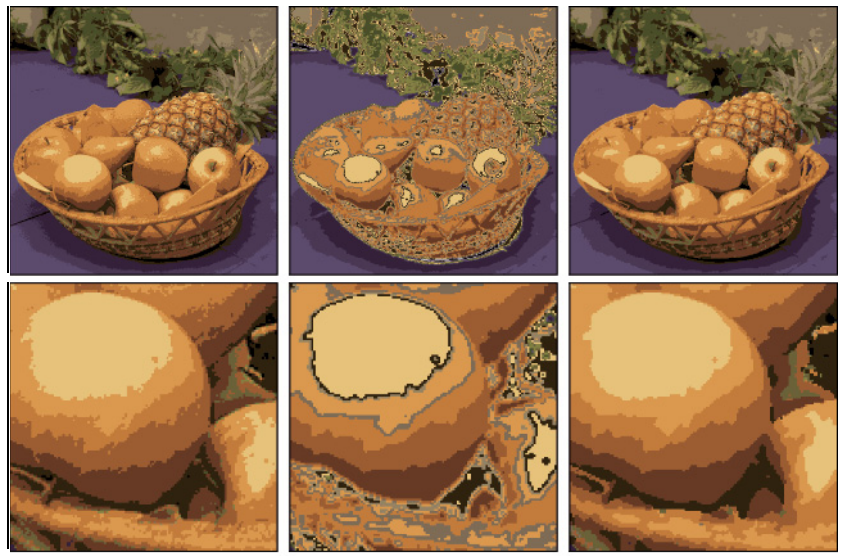
Working with indexed images

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. 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

Program 12.3
Working with indexed images (ImageJ plugin). This plugin increases the brightness of an image by 10 units by modifying the palette. The actual values in the pixel array, which are indices into the palette, are not changed.

```
1  // File Brighten_Index_Image.java
2
3  import ij.ImagePlus;
4  import ij.WindowManager;
5  import ij.plugin.filter.PlugInFilter;
6  import ij.process.ImageProcessor;
7  import java.awt.image.IndexColorModel;
8
9  public class Brighten_Index_Image implements PlugInFilter {
10
11   public int setup(String arg, ImagePlus im) {
12     return DOES_8C; // this plugin works on indexed color images
13   }
14
15   public void run(ImageProcessor ip) {
16     IndexColorModel icm =
17       (IndexColorModel) ip.getColorModel();
18     int pixBits = icm.getPixelSize();
19     int mapSize = icm.getMapSize();
20
21     //retrieve the current lookup tables (maps) for R,G,B
22     byte[] Rmap = new byte[mapSize]; icm.getReds(Rmap);
23     byte[] Gmap = new byte[mapSize]; icm.getGreens(Gmap);
24     byte[] Bmap = new byte[mapSize]; icm.getBlues(Bmap);
25
26     //modify the lookup tables
27     for (int idx = 0; idx < mapSize; idx++){
28       int r = 0xff & Rmap[idx]; //mask to treat as unsigned byte
29       int g = 0xff & Gmap[idx];
30       int b = 0xff & Bmap[idx];
31       Rmap[idx] = (byte) Math.min(r + 10, 255);
32       Gmap[idx] = (byte) Math.min(g + 10, 255);
33       Bmap[idx] = (byte) Math.min(b + 10, 255);
34     }
35
36     //create a new color model and apply to the image
37     IndexColorModel icm2 =
38       new IndexColorModel(pixBits, mapSize, Rmap, Gmap,Bmap);
39     ip.setColorModel(icm2);
40
41     //update the resulting image
42     WindowManager.getCurrentImage().updateAndDraw();
43   }
44
45 } // end of class Brighten_Index_Image
```



(a) (b) (c)

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 suitable ordering relation exists between the index values.

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 to 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 to 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 28–30).

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. (12.3). On the other hand, conversion in the other direction requires *quantization* of the RGB color space and is as a rule more difficult and involved (see Sec. 12.5 for more details). In practice, most applications make use of existing conversion methods such as those available in ImageJ (see pp. 252–253).

[5] Defined in the standard Java class `java.awt.image.ColorModel`.

Program 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
12   public int setup(String arg, ImagePlus im) {
13     return DOES_8C + NO_CHANGES; //does not alter original image
14   }
15
16   public void run(ImageProcessor ip) {
17     int w = ip.getWidth();
18     int h = ip.getHeight();
19
20     //retrieve the color table (palette) for R,G,B
21     IndexColorModel icm =
22         (IndexColorModel) ip.getColorModel();
23     int mapSize = icm.getMapSize();
24     byte[] Rmap = new byte[mapSize]; icm.getReds(Rmap);
25     byte[] Gmap = new byte[mapSize]; icm.getGreens(Gmap);
26     byte[] Bmap = new byte[mapSize]; icm.getBlues(Bmap);
27
28     //create new 24-bit RGB image
29     ColorProcessor cp = new ColorProcessor(w,h);
30     int[] RGB = new int[3];
31     for (int v = 0; v < h; v++) {
32       for (int u = 0; u < w; u++) {
33         int idx = ip.getPixel(u, v);
34         RGB[R] = Rmap[idx];
35         RGB[G] = Gmap[idx];
36         RGB[B] = Bmap[idx];
37         cp.set(u, v, RGB);
38       }
39     }
40     ImagePlus cimg = new ImagePlus("RGB Image",cp);
41     cimg.show();
42   }
43
44 } // end of class Index_To_Rgb
```

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:

```
1 ByteProcessor makeIndexColorImage(int w, int h, int nColors) {
2   // allocate red, green, blue color tables:
3   byte[] Rmap = new byte[nColors];
4   byte[] Gmap = new byte[nColors];
5   byte[] Bmap = new byte[nColors];
6   // color maps need to be filled here
7   byte[] pixels = new byte[w * h];
8   // pixel array (color indices) needs to be filled here
9   IndexColorModel cm
10    = new IndexColorModel(8, nColors, Rmap, Gmap, Bmap);
11  return new ByteProcessor(w, h, pixels, cm);
12 }
```

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 to 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, lines 37–39 would need to be modified as follows:

```
1   int tIdx = 2; // index of transparent color
2   IndexColorModel icm2 = new
3     IndexColorModel(pixBits, mapSize, Rmap, Gmap, Bmap, tIdx);
4   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.

Color image conversion in ImageJ

In ImageJ, the following methods for converting between different types of color and grayscale image objects of type `ImagePlus` and processor objects of type `ImageProcessor` are available:

Converting images of type `ImageProcessor`

ImageJ objects of type `ImageProcessor` can be converted using the methods listed in Table 12.1. Each of these methods returns a new

`ImageProcessor convertToByte(boolean doScaling)`	Converts to an 8-bit grayscale image (`ByteProcessor`).
`ImageProcessor convertToShort(boolean doScaling)`	Converts to a 16-bit grayscale image (`ShortProcessor`).
`ImageProcessor convertToFloat()`	Converts to a 32-bit floating-point image (`FloatProcessor`).
`ImageProcessor convertToRGB()`	Converts to a 32-bit RGB color image (`ColorProcessor`).

Table 12.1
Conversion methods for images of type `ImageProcessor`. If *`doScaling`* is `true` in the first two methods, the pixel values are automatically scaled to the maximum range of the new image.

`ImageProcessor` object, unless the original image is already of the desired type. If this is the case, only a reference to the original image processor is returned, i. e., *no* duplication or modification occurs. The following example demonstrates the conversion from an arbitrary image type to an RGB color image:

```
1  ImageProcessor ip1;
2  ...
3  ImageProcessor ip2 = ip1.convertToRGB();
4  // now ip2 is of type ColorProcessor, ip1 is unmodified.
5  ...
```

In this case, a new object (`ip2`) of type `ColorProcessor` is created and the original object (`ip1`) remains unchanged.

Converting images of type `ImagePlus`

ImageJ image objects of type `ImagePlus` can be converted with the help of methods from the ImageJ class `ImageConverter`, as summarized in Table 12.2. The following example demonstrates the conversion to an RGB color image:

```
1  import ij.process.ImageConverter;
2  ...
3  ImagePlus ipl;
4  ...
5  ImageConverter ic = new ImageConverter(ipl);
6  ic.convertToRGB();
7  // ipl is an RGB image now
```

Note that the method `convertToRGB()` does not return a new image object, but instead modifies the original `ImagePlus` object `ipl`.

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

Table 12.2
Methods of the ImageJ class `ImageConverter` for converting `ImagePlus` objects. Note that these methods do not create any new images, but instead modify the original `ImagePlus` object *ipl* used to instantiate the `ImageConverter`.

`ImageConverter(ImagePlus ipl)` Instantiates an `ImageConverter` object for the image *ipl*.
`void convertToGray8()` Converts *ipl* to an 8-bit grayscale image.
`void convertToGray16()` Converts *ipl* to a 16-bit grayscale image.
`void convertToGray32()` Converts *ipl* to a 32-bit grayscale image (`float`).
`void convertToRGB()` Converts *ipl* to an RGB color image.
`void convertRGBtoIndexedColor(int nColors)` Converts the RGB true color image *ipl* to an indexed image with 8-bit index values and *nColors* colors, performing color quantization.
`void convertToHSB()` Converts *ipl* to a color image using the HSB color space (see Sec. 12.2.3).
`void convertHSBToRGB()` Converts an HSB color space image *ipl* to an RGB color image.

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 any coordinate movement 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.

Figure 12.9 shows the distribution of the colors from a nature image in three different color spaces. The first half of this section introduces these 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.

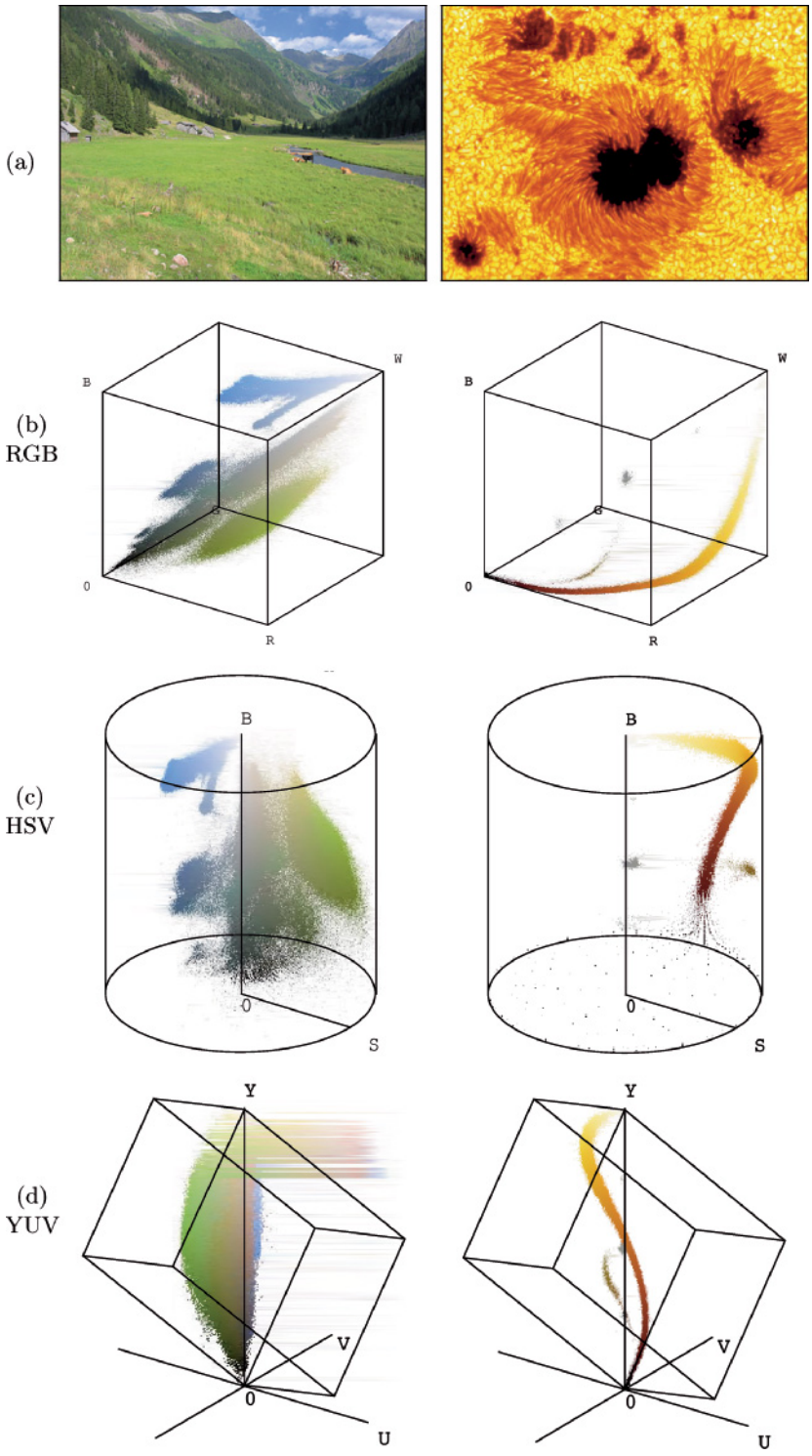


Fig. 12.9
Examples of the color distribution of natural images in three different color spaces. Original images: landscape photograph with dominant green and blue components and sun-spot image with rich red and yellow components (a). Color distribution in RGB- (b), HSV- (c), and YUV-space (d).

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

$$Y = \mathrm{Avg}(R, G, B) = \frac{R + G + B}{3} \tag{12.4}$$

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

$$Y = \mathrm{Lum}(R, G, B) = w_R \cdot R + w_G \cdot G + w_B \cdot B \tag{12.5}$$

is typically used to compute the equivalent luminance value. The weights most often used were originally developed for encoding analog color television signals (see Sec. 12.2.4):

$$w_R = 0.299\,, \qquad w_G = 0.587\,, \qquad w_B = 0.114\,. \tag{12.6}$$

Those recommended in ITU-BT.709 [55] for digital color encoding are

$$w_R = 0.2125\,, \qquad w_G = 0.7154\,, \qquad w_B = 0.072\,. \tag{12.7}$$

If each color component is assigned the same weight, as in Eqn. (12.4), this is of course just a special case of Eqn. (12.5).

Note that, although these weights were developed for use with TV signals, they are optimized for *linear* RGB component values, i. e., signals with no gamma correction. In many practical situations, however, the RGB components are actually *nonlinear*, particularly when we work with sRGB images (see Sec. 12.3.3). In this case, the RGB components must first be linearized to obtain the correct luminance values with the above weights. An alternative is to estimate the luminance without linearization by computing the weighted sum of the *nonlinear* component values and applying a different set of weights (for details see Sec. 12.3.3, p. 287).

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. (12.11) in Sec. 12.2.3) or the *luminance* L in HLS (Eqn. (12.21)), 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 $I(u, v) = (R, G, B)$ are the same; i. e., 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,

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} \leftarrow \begin{pmatrix} Y \\ Y \\ Y \end{pmatrix}, \tag{12.8}$$

by using $Y = \text{Lum}(R, G, B)$ from Eqn. (12.5), 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 method

`convertToByte(boolean doScaling)`,

which returns a new image of type `ByteProcessor` (see Table 12.1 and the example on page 253). ImageJ uses the default weights $w_R = w_G = w_B = \frac{1}{3}$ (as in Eqn. (12.4)) for the RGB components, or $w_R = 0.299$, $w_G = 0.587$, $w_B = 0.114$ (as in Eqn. (12.6)) if the "Weighted RGB Conversions" option is selected in the Edit→Options→Conversions dialog. Arbitrary weights (`wr`, `wg`, `wb`) can be specified for subsequent conversion operations through the static `ColorProcessor` method

`setWeightingFactors(double wr, double wg, double wb)`.

Similarly, the static method `ColorProcessor.getWeightingFactors()` can be used to retrieve the current 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 Sec. 12.3.3).

12.2.2 Desaturating 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 (R_d, G_d, B_d) computed by linear interpolation between the pixel's original color and the corresponding (Y, Y, Y) gray point in the RGB space, i. e.,

$$\begin{pmatrix} R_d \\ G_d \\ B_d \end{pmatrix} \leftarrow \begin{pmatrix} Y \\ Y \\ Y \end{pmatrix} + s_{\text{col}} \cdot \begin{pmatrix} R - Y \\ G - Y \\ B - Y \end{pmatrix}, \tag{12.9}$$

where the factor $s_{\text{col}} \in [0, 1]$ controls the remaining amount of color saturation (Fig. 12.10). This gradual transition is referred to as desaturating a color image. A value of $s_{\text{col}} = 0$ completely eliminates all color, resulting in a true grayscale image, and with $s_{\text{col}} = 1$ the color values will be unchanged. In Prog. 12.5, continuous desaturation as defined in Eqn. (12.9) is implemented as an ImageJ plugin.

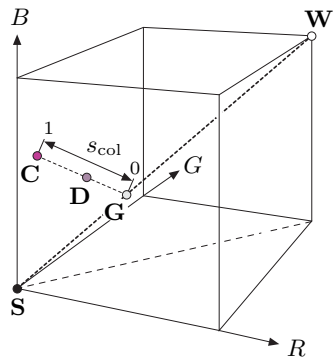
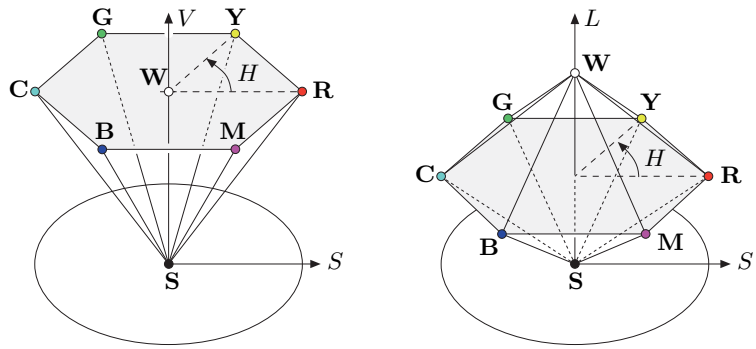


Fig. 12.10
Desaturation in RGB space: original color point $\mathbf{C} = (R, G, B)$, its corresponding gray point $\mathbf{G} = (Y, Y, Y)$, and the desaturated color point $\mathbf{D} = (R_d, G_d, B_d)$. Saturation is controlled by the factor s_{col}.



(a) HSV (b) HLS

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**).

12.2.3 HSV/HSB and HLS Color Space

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 = *brightness*),[6] it denotes the same 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 value, the horizontal distance from the axis the S value, and the angle the H 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 base. While this space is often represented as a pyramid, according to its mathematical definition, the space is actually a *cylinder*, as shown below (Fig. 12.13).

The **HLS** color space[7] (*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

[6] Sometimes the HSV space is also referred to as the "HSI" space, where 'I' stands for *intensity*.

[7] The acronyms HLS and HSL are used interchangeably.

Program 12.5
Continuous desaturation of an RGB color image (ImageJ plugin). The amount of color saturation is controlled by the variable `sCol` defined in line 9 (see Eqn. (12.9)).

```
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
9   static double sCol = 0.3; // color saturation factor
10
11  public int setup(String arg, ImagePlus im) {
12    return DOES_RGB;
13  }
14
15  public void run(ImageProcessor ip) {
16
17    // iterate over all pixels
18    for (int v = 0; v < ip.getHeight(); v++) {
19      for (int u = 0; u < ip.getWidth(); u++) {
20
21        // get int-packed color pixel
22        int c = ip.get(u, v);
23
24        // extract RGB components from color pixel
25        int r = (c & 0xff0000) >> 16;
26        int g = (c & 0x00ff00) >> 8;
27        int b = (c & 0x0000ff);
28
29        // compute equivalent gray value
30        double y = 0.299 * r + 0.587 * g + 0.114 * b;
31
32        // linearly interpolate (yyy) ↔ (rgb)
33        r = (int) (y + sCol * (r - y));
34        g = (int) (y + sCol * (g - y));
35        b = (int) (y + sCol * (b - y));
36
37        // reassemble color pixel
38        c = ((r & 0xff)<<16) | ((g & 0xff)<<8) | b & 0xff;
39        ip.set(u, v, c);
40      }
41    }
42  }
43
44 } // end of class Desaturate_Rgb
```

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 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

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

$$S_{\text{HSV}} = \begin{cases} \frac{C_{\text{rng}}}{C_{\text{high}}} & \text{for } C_{\text{high}} > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{12.10}$$

and the luminance (*value*)

$$V_{\text{HSV}} = \frac{C_{\text{high}}}{C_{\max}}\,, \tag{12.11}$$

with C_{high}, C_{low}, and C_{rng} defined as

$$C_{\text{high}} = \max(R, G, B), \quad C_{\text{low}} = \min(R, G, B), \quad C_{\text{rng}} = C_{\text{high}}. \tag{12.12}$$

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 compute H_{HSV} when $C_{\text{rng}} > 0$, we first normalize each component using

$$R' = \frac{C_{\text{high}} - R}{C_{\text{rng}}}, \qquad G' = \frac{C_{\text{high}} - G}{C_{\text{rng}}}, \qquad B' = \frac{C_{\text{high}} - B}{C_{\text{rng}}}. \tag{12.13}$$

Then, depending on which of the three original color components had the maximal value, we compute a preliminary hue H' as

$$H' = \begin{cases} B' - G' & \text{if } R = C_{\text{high}} \\ R' - B' + 2 & \text{if } G = C_{\text{high}} \\ G' - R' + 4 & \text{if } B = C_{\text{high}}. \end{cases} \tag{12.14}$$

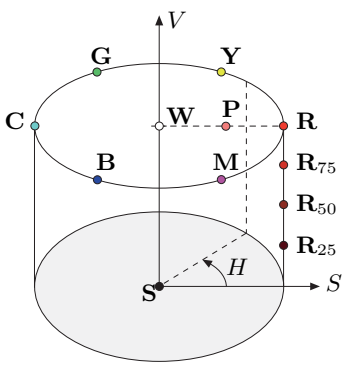
Since the resulting value for H' lies on the interval $[-1 \ldots 5]$, we obtain the final hue value by normalizing to the interval $[0, 1]$ as

$$H_{\text{HSV}} = \frac{1}{6} \cdot \begin{cases} (H' + 6) & \text{for } H' < 0 \\ H' & \text{otherwise.} \end{cases} \tag{12.15}$$

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_{HSV} S_{HSV} V_{HSV}

Fig. 12.12
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.



RGB/HSV Values

Pt.	Color	R	G	B	H	S	V
S	Black	0.00	0.00	0.00	—	0.00	0.00
R	Red	1.00	0.00	0.00	0	1.00	1.00
Y	Yellow	1.00	1.00	0.00	1/6	1.00	1.00
G	Green	0.00	1.00	0.00	2/6	1.00	1.00
C	Cyan	0.00	1.00	1.00	3/6	1.00	1.00
B	Blue	0.00	0.00	1.00	4/6	1.00	1.00
M	Magenta	1.00	0.00	1.00	5/6	1.00	1.00
W	White	1.00	1.00	1.00	—	0.00	1.00
$\mathbf{R}_{75}$	75% Red	0.75	0.00	0.00	0	1.00	0.75
$\mathbf{R}_{50}$	50% Red	0.50	0.00	0.00	0	1.00	0.50
$\mathbf{R}_{25}$	25% Red	0.25	0.00	0.00	0	1.00	0.25
P	Pink	1.00	0.50	0.50	0	0.5	1.00

Fig. 12.13
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 **S** and the white point **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 ($\mathbf{R}_{25}$, $\mathbf{R}_{50}$, $\mathbf{R}_{75}$, **R**).

$$H^{\circ}_{\text{HSV}} = H_{\text{HSV}} \cdot 360.$$

Under this definition, the RGB space unit cube is mapped to a *cylinder* with height and radius of length 1 (Fig. 12.13). 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 shows the individual HSV components (in grayscale) of the test image in Fig. 12.2. Figure 12.13 plots the location of some notable color points and compares them with their locations in RGB space (see also Fig. 12.1).

Java implementation

In Java, the RGB-HSV conversion is implemented in the class `java.awt.Color` by the 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 usage example:

```
1 import java.awt.Color;
2 ...
3 float[] hsv = new float[3];
4 int red = 128, green = 255, blue = 0;
5 hsv = Color.RGBtoHSB (red, green, blue, hsv);
6 float h = hsv[0];
7 float s = hsv[1];
8 float v = hsv[2];
9 ...
```

A possible implementation of the Java method `RGBtoHSB()` using the definition in Eqns. (12.11)–(12.15) is given in Prog. 12.6.

HSV→RGB

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

$$H' = (6 \cdot H_{\text{HSV}}) \bmod 6 \tag{12.16}$$

$(0 \leq H' < 6)$ is determined first, followed by computing the intermediate values

$$\begin{aligned} c_1 &= \lfloor H' \rfloor, & x &= (1 - S_{\text{HSV}}) \cdot v, \\ c_2 &= H' - c_1, & y &= (1 - (S_{\text{HSV}} \cdot c_2)) \cdot V_{\text{HSV}}, \\ & & z &= (1 - (S_{\text{HSV}} \cdot (1 - c_2))) \cdot V_{\text{HSV}}. \end{aligned} \tag{12.17}$$

Depending on the value of c_1, the normalized RGB values $R', G', B' \in [0, 1]$ are then computed from $v = V_{\text{HSV}}$, x, y, and z as follows:[8]

$$(R', G', B') = \begin{cases} (v, z, x) & \text{if } c_1 = 0 \\ (y, v, x) & \text{if } c_1 = 1 \\ (x, v, z) & \text{if } c_1 = 2 \\ (x, y, v) & \text{if } c_1 = 3 \\ (z, x, v) & \text{if } c_1 = 4 \\ (v, x, y) & \text{if } c_1 = 5. \end{cases} \tag{12.18}$$

The scaling of the RGB components to whole numbers in the range $[0, N - 1]$ (typically $N = 256$) is carried out as follows:

$$\begin{aligned} R &= \min\bigl(\text{round}(N \cdot R'), N - 1\bigr), \\ G &= \min\bigl(\text{round}(N \cdot G'), N - 1\bigr), \\ B &= \min\bigl(\text{round}(N \cdot B'), N - 1\bigr). \end{aligned} \tag{12.19}$$

[8] The variables x, y, z used here have no relation to those used in the CIEXYZ color space (Sec. 12.3.1).

Program 12.6
RGB→HSV conversion. 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.

```
1  static float[] RGBtoHSV (int R, int G, int B, float[] HSV) {
2    // R, G, B ∈ [0, 255]
3    float H = 0, S = 0, V = 0;
4    float cMax = 255.0f;
5    int cHi = Math.max(R,Math.max(G,B)); // highest color value
6    int cLo = Math.min(R,Math.min(G,B)); // lowest color value
7    int cRng = cHi - cLo;          // color range
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 highest color value
23       hh = bb - gg;
24     else if (G == cHi)               // G is highest color value
25       hh = rr - bb + 2.0f;
26     else                             // B is highest color value
27       hh = gg - rr + 4.0f;
28     if (hh < 0)
29       hh= hh + 6;
30     H = hh / 6;
31   }
32
33   if (HSV == null) // create a new HSV array if needed
34     HSV = new float[3];
35   HSV[0] = H; HSV[1] = S; HSV[2] = V;
36   return HSV;
37 }
```

Java implementation

In Java, HSV→RGB conversion is implemented in the standard AWT class `java.awt.Color` by the 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.

Program 12.7 HSV→RGB conversion. This Java method takes the same arguments and returns identical results as the standard method `Color.HSBtoRGB()`.

```
1  static int HSVtoRGB (float h, float s, float v) {
2    // h, s, v ∈ [0, 1]
3    float rr = 0, gg = 0, bb = 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: rr=v; gg=z; bb=x; break;
12     case 1: rr=y; gg=v; bb=x; break;
13     case 2: rr=x; gg=v; bb=z; break;
14     case 3: rr=x; gg=y; bb=v; break;
15     case 4: rr=z; gg=x; bb=v; break;
16     case 5: rr=v; gg=x; bb=y; break;
17   }
18   int N = 256;
19   int r = Math.min(Math.round(rr*N), N-1);
20   int g = Math.min(Math.round(gg*N), N-1);
21   int b = Math.min(Math.round(bb*N), N-1);
22   // create int-packed RGB color:
23   int rgb = ((r&0xff)<<16) | ((g&0xff)<<8) | b&0xff;
24   return rgb;
25 }
```

RGB→HLS

In the HLS model, the *hue* value H_{HLS} is computed in the same way as in the HSV model (Eqns. (12.13)–(12.15)), i. e.,

$$H_{\text{HLS}} = H_{\text{HSV}}. \tag{12.20}$$

The other values, L_{HLS} and S_{HLS}, are computed as follows (for C_{high}, C_{low}, and C_{rng}, see Eqn. (12.12)):

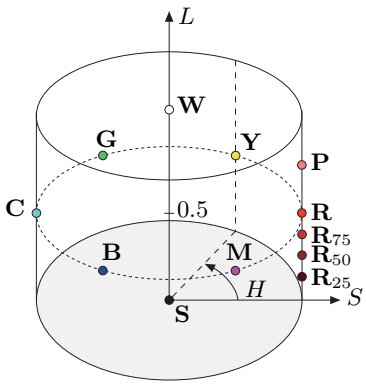
$$L_{\text{HLS}} = \frac{C_{\text{high}} + C_{\text{low}}}{2}, \tag{12.21}$$

$$S_{\text{HLS}} = \begin{cases} 0 & \text{for } L_{\text{HLS}} = 0 \\ 0.5 \cdot \frac{C_{\text{rng}}}{L_{\text{HLS}}} & \text{for } 0 < L_{\text{HLS}} \le 0.5 \\ 0.5 \cdot \frac{C_{\text{rng}}}{1 - L_{\text{HLS}}} & \text{for } 0.5 < L_{\text{HLS}} < 1 \\ 0 & \text{for } L_{\text{HLS}} = 1. \end{cases} \tag{12.22}$$

Figure 12.14 shows the individual HLS components of the test image as grayscale images. Using the above definitions, the unit cube in the RGB space is again mapped to a cylinder with height and length 1 (Fig. 12.15).

H_{HLS} S_{HLS} L_{HLS}

Fig. 12.14
HLS color components H_{HLS} (*hue*), S_{HLS} (*saturation*), and L_{HLS} (*luminance*). Note that the S and L images are swapped to appear in the same order as in HSV space (see Fig. 12.12).



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
$\mathbf{R}_{75}$	75% Red	0.75	0.00	0.00	0	1.00	0.375
$\mathbf{R}_{50}$	50% Red	0.50	0.00	0.00	0	1.00	0.250
$\mathbf{R}_{25}$	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

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 ($\mathbf{R}_{25}$, $\mathbf{R}_{50}$, $\mathbf{R}_{75}$, **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).

In contrast to the HSV space (Fig. 12.13), 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.

HLS→RGB

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

$$(R', G', B') = \begin{cases} (0,0,0) & \text{for } L_{\text{HLS}} = 0 \\ (1,1,1) & \text{for } L_{\text{HLS}} = 1. \end{cases} \tag{12.23}$$

Otherwise, we again determine the appropriate color sector

$$H' = (6 \cdot H_{\text{HLS}}) \bmod 6\,, \tag{12.24}$$

where $(0 \le H' < 6)$, and then, based on the resulting sector, we determine the values

$$
\begin{aligned}
c_1 &= \lfloor H' \rfloor & d &= \begin{cases} S_{\text{HLS}} \cdot L_{\text{HLS}} & \text{for } L_{\text{HLS}} \le 0.5 \\ S_{\text{HLS}} \cdot (L_{\text{HLS}} - 1) & \text{for } L_{\text{HLS}} > 0.5 \end{cases} \\
c_2 &= H' - c_1 & & \\
w &= L_{\text{HLS}} + d & y &= w - (w - x) \cdot c_2 \\
x &= L_{\text{HLS}} - d & z &= x + (w - x) \cdot c_2.
\end{aligned} \tag{12.25}
$$

The assignment of the RGB values is done similarly to Eqn. (12.18), i. e.,

$$
(R', G', B') = \begin{cases} (w, z, x) & \text{if } c_1 = 0 \\ (y, w, x) & \text{if } c_1 = 1 \\ (x, w, z) & \text{if } c_1 = 2 \\ (x, y, w) & \text{if } c_1 = 3 \\ (z, x, w) & \text{if } c_1 = 4 \\ (w, x, y) & \text{if } c_1 = 5. \end{cases} \tag{12.26}
$$

Finally, scaling the normalized ($[0, 1]$) R', G', B' color components back into the $[0, 255]$ range is done as in Eqn. (12.19).

Java implementation (RGB↔HLS)

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. (12.20)–(12.22). The HLS→RGB conversion is given in Prog. 12.9.

Comparing HSV and HLS

Despite the gross 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 sampling the RGB space at an interval of 0.1 in each dimension.

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* (that is, isolating objects according to their *hue*) on a homogeneously colored background where the brightness is not necessarily constant.

Program 12.8
RGB→HLS conversion (Java method).

```
1  static float[] RGBtoHLS (float R, float G, float B) {
2    // R, G, B assumed to be in [0,1]
3    float cHi = Math.max(R,Math.max(G,B)); // highest color value
4    float cLo = Math.min(R,Math.min(G,B)); // lowest color value
5    float cRng = cHi - cLo;        // color range
6
7    // compute luminance L
8    float L = (cHi + cLo)/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 / d;
15   }
16
17   // compute hue H
18   float H=0;
19   if (cHi > 0 && cRng > 0) {     // a color pixel
20     float rr = (float)(cHi - R) / cRng;
21     float gg = (float)(cHi - G) / cRng;
22     float bb = (float)(cHi - B) / cRng;
23     float hh;
24     if (R == cHi)                  // R is highest color value
25       hh = bb - gg;
26     else if (G == cHi)             // G is highest color value
27       hh = rr - bb + 2.0f;
28     else                           // B is highest color value
29       hh = gg - rr + 4.0f;
30
31     if (hh < 0)
32       hh= hh + 6;
33     H = hh / 6;
34   }
35
36   return new float[] {H,L,S};
37 }
```

12.2.4 TV 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 [49]. 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 sig-

Program 12.9 HLS→RGB conversion (Java method).

```
1  static float[] HLStoRGB (float H, float L, float S) {
2    // H, L, S assumed to be in [0,1]
3    float R = 0, G = 0, B = 0;
4
5    if (L <= 0)      // black
6      R = G = B = 0;
7    else if (L >= 1)  // white
8      R = G = B = 1;
9    else {
10     float hh = (6 * H) % 6;
11     int   c1 = (int) hh;
12     float c2 = hh - c1;
13     float d = (L <= 0.5f) ? (S * L) : (S * (1 - L));
14     float w = L + d;
15     float x = L - d;
16     float y = w - (w - x) * c2;
17     float z = x + (w - x) * c2;
18     switch (c1) {
19       case 0: R=w; G=z; B=x; break;
20       case 1: R=y; G=w; B=x; break;
21       case 2: R=x; G=w; B=z; break;
22       case 3: R=x; G=y; B=w; break;
23       case 4: R=z; G=x; B=w; break;
24       case 5: R=w; G=x; B=y; break;
25     }
26   }
27   return new float[] {R,G,B};
28 }
```

nal 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. (12.6), from the RGB components as

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{12.27}$$

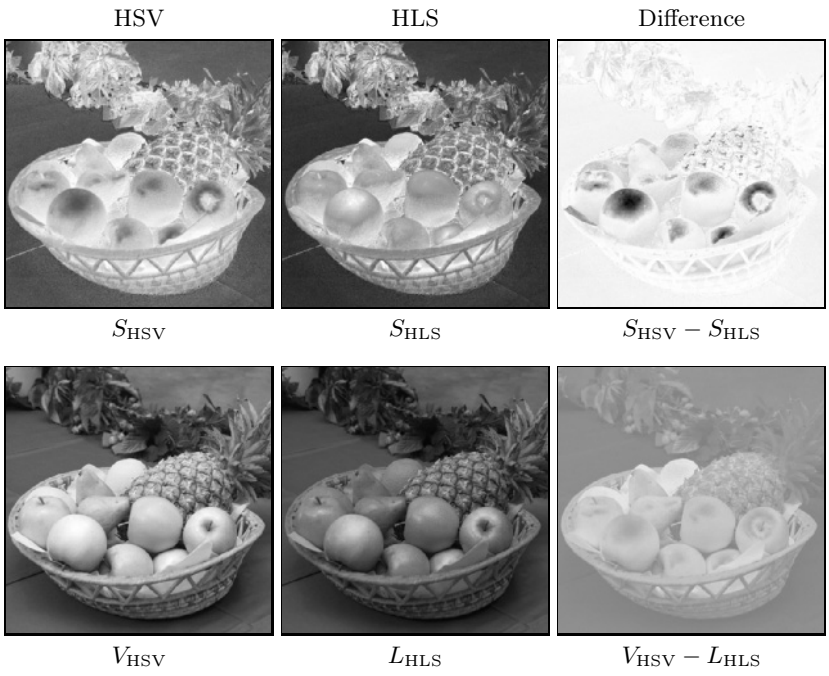


Fig. 12.16
Comparison between HSV and HLS components: *saturation* (top row) and *intensity* (bottom row). In the color *saturation* difference image $S_{\text{HSV}} - S_{\text{HLS}}$ (top), light areas correspond to positive values and dark areas to negative values. Saturation in the HLS representation, 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 H (not shown) is identical in both representations.

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 Sec. 5.7) for playback. The UV components are computed from a weighted difference between the luminance and the blue or red components as

$$U = 0.492 \cdot (B - Y) \qquad \text{and} \qquad V = 0.877 \cdot (R - Y), \tag{12.28}$$

and the entire transformation from RGB to YUV is

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}. \tag{12.29}$$

The transformation from YUV back to RGB is found by inverting the matrix in Eqn. (12.29):

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.000 & 1.140 \\ 1.000 & -0.395 & -0.581 \\ 1.000 & 2.032 & 0.000 \end{pmatrix} \cdot \begin{pmatrix} Y \\ U \\ V \end{pmatrix}. \tag{12.30}$$

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

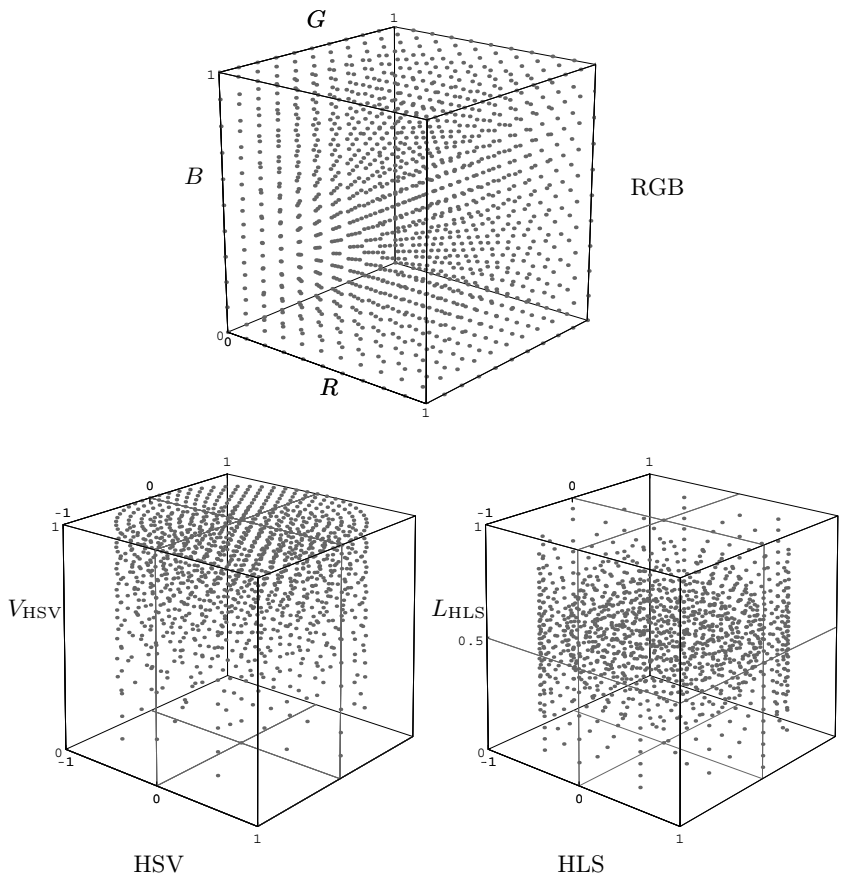


Fig. 12.17
Distribution of colors in the RGB, HSV, and HLS spaces. The starting point is the uniform distribution of colors in RGB space (top). The corresponding colors in the HSV and HLS spaces are distributed nonsymmetrically (HSV) and symmetrically (HLS) within the cylindrical space.

$$\begin{pmatrix} I \\ Q \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} \cos\beta & \sin\beta \\ -\sin\beta & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} U \\ V \end{pmatrix}, \tag{12.31}$$

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 [57, 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 (for example, 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 [82, p. 16]. For arbitrary weights w_B, w_R, the transformation is defined as

$$
\begin{aligned}
Y &= w_R \cdot R + (1 - w_B - w_R) \cdot G + w_B \cdot B, \\
C_b &= \frac{0.5}{1 - w_B} \cdot (B - Y), \\
C_r &= \frac{0.5}{1 - w_R} \cdot (R - Y),
\end{aligned} \tag{12.32}
$$

and the inverse transformation from $\mathrm{YC}_b\mathrm{C}_r$ to RGB is

$$
\begin{aligned}
R &= Y + \frac{1 - w_R}{0.5} \cdot C_r, \\
G &= Y - \frac{w_B \cdot (1 - w_B) \cdot C_b \; - \; w_R \cdot (1 - w_R) \cdot C_r}{0.5 \cdot (1 - w_B - w_R)}, \\
B &= Y + \frac{1 - w_B}{0.5} \cdot C_b.
\end{aligned} \tag{12.33}
$$

The ITU[9] recommendation BT.601 [56] specifies the values $w_R = 0.299$ and $w_B = 0.114$ ($w_G = 1 - w_B - w_R = 0.587$). Using these values, the transformation becomes

$$
\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}, \tag{12.34}
$$

and the inverse transformation becomes

$$
\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.000 & 1.403 \\ 1.000 & -0.344 & -0.714 \\ 1.000 & 1.773 & 0.000 \end{pmatrix} \cdot \begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix}. \tag{12.35}
$$

Different weights are recommended based on how the color space is used; for example, ITU-BT.709 [55] 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, e. g., $128 = 2^7$ in case of 8-bit components.

Figure 12.18 shows the three color spaces YUV, YIQ, and $\mathrm{YC}_b\mathrm{C}_r$ together for comparison. The U, V, I, Q, and C_b, C_r values in the 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 $\mathrm{YC}_b\mathrm{C}_r$ 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

[9] International Telecommunication Union (www.itu.int).

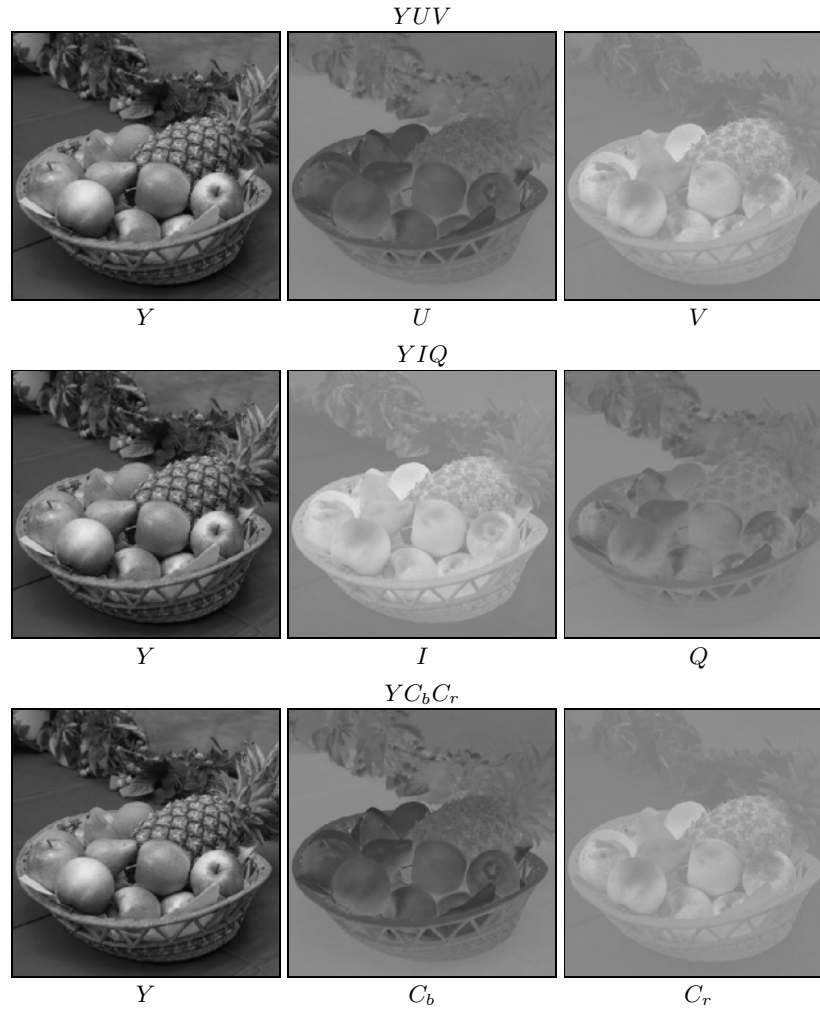


Fig. 12.18
Comparing YUV-, YIQ- and YC_bC_r values. The Y values are identical in all three color spaces.

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)[10] 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* (G), and yellow absorbs *blue* (B). The simplest form of the CMY model is defined as

$$\begin{aligned} C &= 1 - R, \\ M &= 1 - G, \\ Y &= 1 - B. \end{aligned} \tag{12.36}$$

[10] Note that in this case Y stands for *yellow* and has nothing to do with the Y luminance component in YUV or YC_bC_r.

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

$$K = \min(C, M, Y). \tag{12.37}$$

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 (Version 1): In this simple variant the C, M, Y values are reduced linearly with increasing K and the modified components C', M', Y', K' are defined as

$$\begin{pmatrix} C' \\ M' \\ Y' \\ K' \end{pmatrix} = \begin{pmatrix} C - K \\ M - K \\ Y - K \\ K \end{pmatrix}. \tag{12.38}$$

CMY→CMYK (Version 2): The second variant corrects the color by reducing the C, M, Y components with $\frac{1}{1-K}$, resulting in stronger colors in the dark areas of the image:

$$\begin{pmatrix} C' \\ M' \\ Y' \end{pmatrix} = \begin{pmatrix} C - K \\ M - K \\ Y - K \end{pmatrix} \cdot \begin{cases} \frac{1}{1-K} & \text{for } K < 1 \\ 1 & \text{otherwise.} \end{cases} \tag{12.39}$$
$$K' \quad = \quad K$$

In both versions, the fourth component is used directly (from Eqn. (12.37)) 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 12.19 (a) compares the result from version 2 with that produced with Adobe Photoshop (Fig. 12.19 (c)). The difference in the cyan component C is particularly noticeable and also the amount of black (K) 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.

CMY→CMYK (Version 3): In print production, special transfer functions are applied to tune the results. For example, the Adobe PostScript interpreter [62, p. 345] specifies an *undercolor-removal function* $f_{\mathrm{UCR}}(K)$ for gradually reducing the CMY components and a separate *black-generation function* $f_{\mathrm{BG}}(K)$ for controlling the amount of black. These functions are used in the form

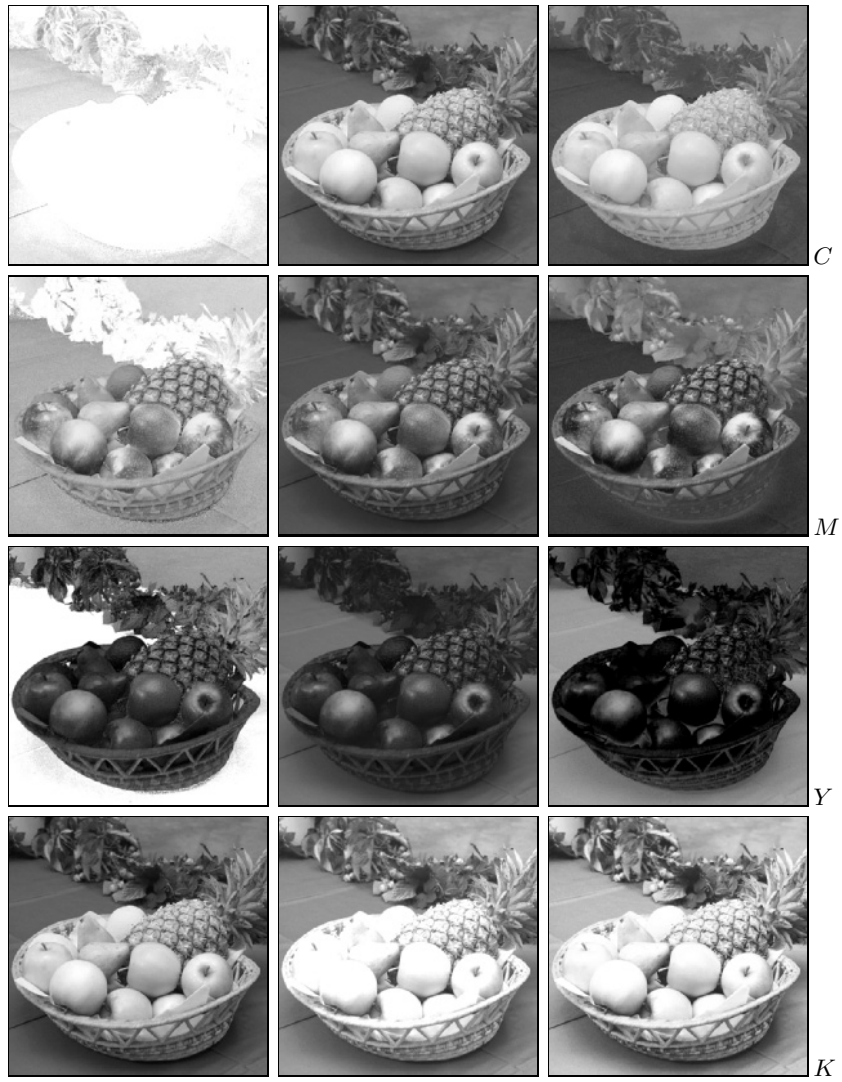


Fig. 12.19
RGB→CMYK conversion comparison. Simple conversion using Eqn. (12.39) (a), applying the *undercolor-removal* and *black-generation* functions of Eqn. (12.40) (b), and results obtained with Adobe Photoshop (c). The color intensities are shown inverted, i. e., darker areas represent higher CMYK 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.

$$\begin{pmatrix} C' \\ M' \\ Y' \\ K' \end{pmatrix} = \begin{pmatrix} C - f_{\text{UCR}}(K) \\ M - f_{\text{UCR}}(K) \\ Y - f_{\text{UCR}}(K) \\ f_{\text{BG}}(K) \end{pmatrix}, \tag{12.40}$$

where $K = \min(C, M, Y)$ again (as defined in Eqn. (12.37)). The functions f_{UCR} and f_{BG} are usually nonlinear, and the resulting values C', M', Y', K' are scaled (typically by means of *clamping*) to the interval $[0, 1]$. The example shown in Fig. 12.19 (b) was produced using the functions

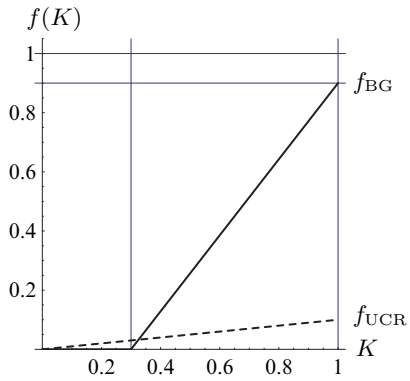


Fig. 12.20
Examples of *undercolor-removal function* f_{UCR} (Eqn. (12.41)) and *black generation function* f_{BG} (Eqn. (12.42)). The parameter settings are $s_K = 0.1$, $K_0 = 0.3$, and $K_{\max} = 0.9$.

$$f_{\text{UCR}}(K) = s_K \cdot K, \tag{12.41}$$

$$f_{\text{BG}}(K) = \begin{cases} 0 & \text{for } K < K_0 \\ K_{\max} \cdot \frac{K - K_0}{1 - K_0} & \text{for } K \geq K_0, \end{cases} \tag{12.42}$$

where $s_K = 0.1$, $K_0 = 0.3$, and $K_{\max} = 0.9$ (see Fig. 12.20). With this definition, f_{UCR} reduces the CMY components by 10% of the K value (by Eqn. (12.40)), which mostly affects the dark areas of the image with high K values. The effect of the function f_{BG} (Eqn. (12.42)) is that for values of $K < K_0$ (that is in the light areas of the image), no black ink is added at all. In the interval $K = K_0 \ldots 1.0$, the black component is increased linearly up to the maximum value $K_{\max}$. The result in Fig. 12.19 (b) is relatively close to the CMYK component values produced by Photoshop[11] in Fig. 12.19 (c). It could be further improved by adjusting the function parameters s_K, K_0, and $K_{\max}$ (Eqn. (12.40)).

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 argue in the following section, technically correct color conversions need to be based on precise, "colorimetric" grounds.

12.3 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

[11] Actually Adobe Photoshop does not convert directly from RGB to CMYK. Instead, it first converts to, and then from, the CIE L*a*b* color space (see Sec. 12.3.1).

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 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., [106] or [91] for an overview). While several colorimetric standards exist, we focus on the most widely used CIE systems in the remaining part of this section.

12.3.1 CIE Color Spaces

The XYZ color system, developed by the CIE (Commission Internationale d'Èclairage)[12] in the 1920s and standardized in 1931, is the foundation of most colorimetric color systems that are in use today [81, p. 22].

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 three-dimensional cone-shaped region (Fig. 12.21 (a)), which interestingly enough does not include the primary colors themselves.

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. 12.3.3. Thus, as shown in Fig. 12.21 (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 it, 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 12.3.

[12] International Commission on Illumination (www.cie.co.at).

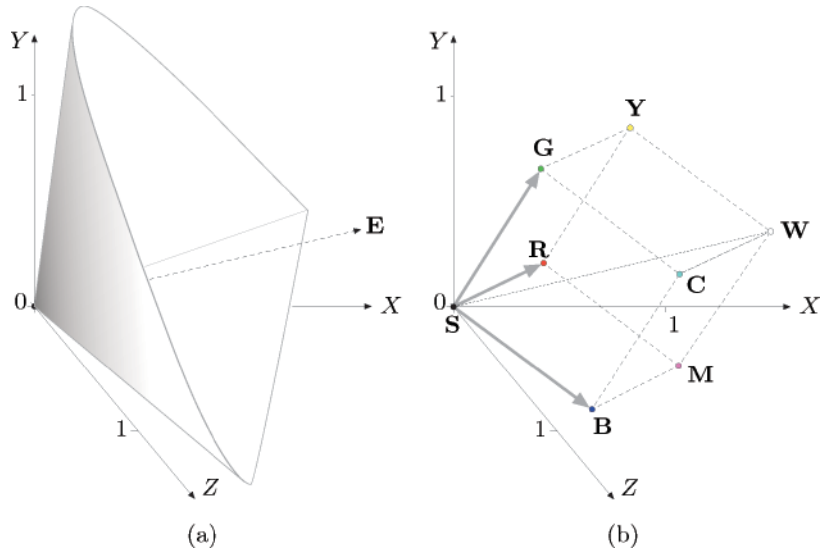


Fig. 12.21
CIE XYZ color space. The XYZ color space is defined by the three imaginary primary colors X, Y, Z, where the Y dimension corresponds to the perceived luminance. All visible colors are contained inside an open, cone-shaped volume that originates at the black point **S** (a), where E denotes the axis of neutral (gray) colors. The RGB color space maps to the XYZ space as a linearly distorted cube (b).

Table 12.3
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 12.4), x, y denote the corresponding CIE chromaticity coordinates.

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

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

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z}, \qquad (12.43)$$

where (obviously) $x + y + z = 1$ and thus one of the three values (e. g., z) is redundant. Equation (12.43) describes a central projection from X, Y, Z coordinates onto the three-dimensional plane

$$X + Y + Z = 1,$$

with the origin **S** as the projection center (Fig. 12.22). Thus, for an arbitrary XYZ color point $\mathbf{A} = (X_a, Y_a, Z_a)$, the corresponding chromaticity

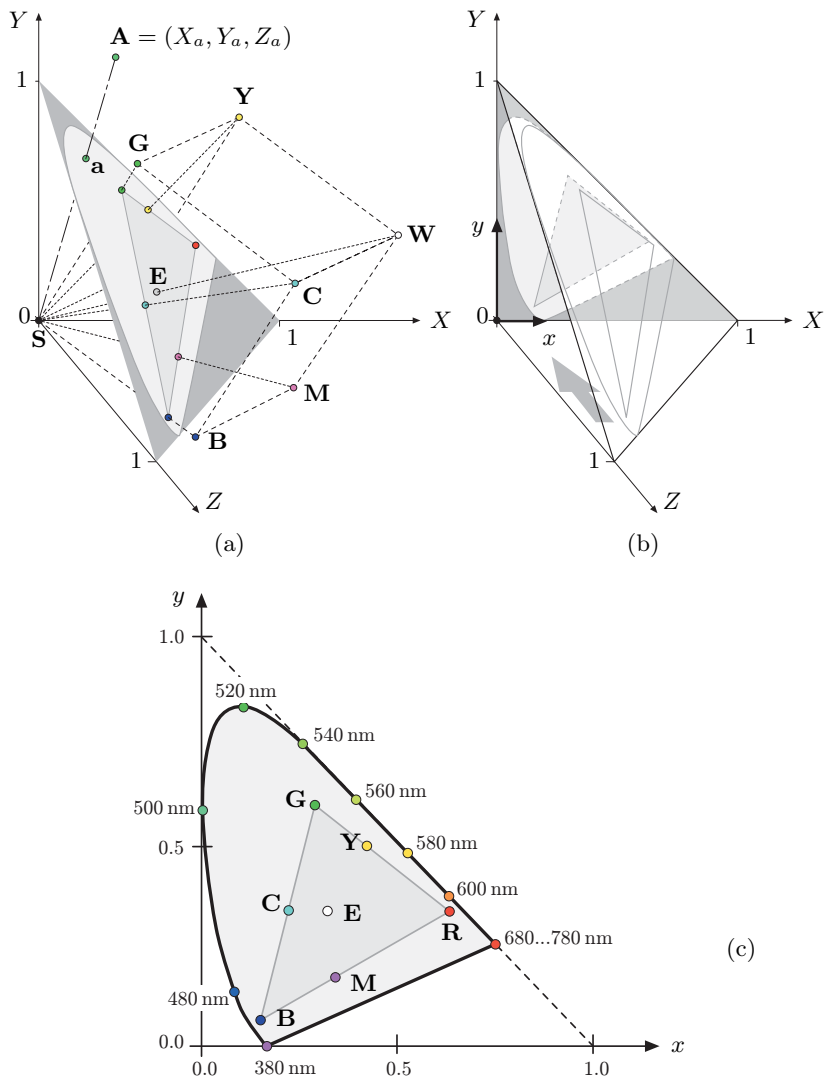


Fig. 12.22
CIE x, y chromaticity diagram. For an arbitrary XYZ color point $\mathbf{A} = (X_a, Y_a, Z_a)$, the chromaticity values $\mathbf{a} = (x_a, y_a, z_a)$ are obtained by a central projection onto the 3D plane $X + Y + Z = 1$ (a). The corner points of the RGB cube map to a triangle, and its white point **W** maps to the (colorless) neutral point **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., **R**, **G**, **B**), which define a triangle (linear hull) containing all representable colors.

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. 12.22 (a)). The final x, y coordinates are the result of projecting these intersection points onto the X/Y-plane (Fig. 12.22 (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. 12.22 (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. (12.43)) as

$$X = x \cdot \frac{Y}{y}\,, \qquad Z = z \cdot \frac{Y}{y} = (1 - x - y) \cdot \frac{Y}{y}. \tag{12.44}$$

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 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.

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) [106, Sec. 3.3.3]. The following daylight (D) illuminants are particularly important for the design of digital color spaces (Table 12.4):

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.

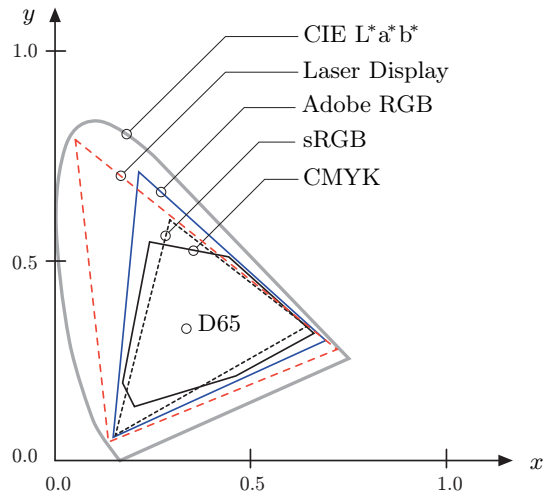


Fig. 12.23
Gamut regions for different color spaces and output devices inside the CIE diagram.

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 emittive 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. 12.3.3) 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 ±2°).

Table 12.4
CIE color parameters for the standard illuminants **D50** and **D65**. **E** denotes the absolute neutral point in CIE XYZ space.

Pt.	Temp.	X	Y	Z	x	y
D50	5000° K	0.964296	1.000000	0.825105	0.3457	0.3585
D65	6500° K	0.950456	1.000000	1.088754	0.3127	0.3290
E	5400° K	1	1	1	1/3	1/3

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 three-dimensional CIE XYZ color space or, reduced to the representable color hues and ignoring the luminance component, a convex region in the two-dimensional CIE chromaticity diagram. Figure 12.23 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. 12.23. 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 L*a*b* (described below), which covers the entire visible portion of the CIE diagram.

Variants of the CIE color space

The original CIE XYZ 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, 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′, L*u*v*, YC$_b$C$_r$, and particularly L*a*b*, which is described below.

In addition, CIE-compliant specifications exist for most common color spaces (see Sec. 12.2), which allow more or less dependable conversions between almost any pair of color spaces.

12.3.2 CIE L*a*b*

The L*a*b* 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, L*a*b*[13] 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 Sec. 5.7.6) is applied to all three components, as detailed below.

[13] Often L*a*b* is simply referred to as the "Lab" color space.

Table 12.5
CIE L*a*b* coordinates for selected RGB color points. The X_{65}, Y_{65}, Z_{65} values relate to the standard (ITU-R BT.709) primaries and white point D65 (see Tables 12.3 and 12.4).

Pt.	Color	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	0.00	1.00	1.00	0.5929	0.2848	0.9696	60.32	98.23	−60.83
W	white	1.00	1.00	1.00	0.9505	1.0000	1.0888	100.00	0.00	0.00

Transformation CIE XYZ → L*a*b*

Several specifications for converting to and from L*a*b* space exist that, however, differ marginally and for very small L values only. The current specification for converting between CIE XYZ and L*a*b* colors is defined by ISO Standard 13655 [53] as follows:

$$\begin{aligned} L^* &= 116 \cdot Y' - 16, \\ a^* &= 500 \cdot (X' - Y'), \\ b^* &= 200 \cdot (Y' - Z'), \end{aligned} \tag{12.45}$$

$$\text{where} \quad X' = f_1\big(\tfrac{X}{X_{\text{ref}}}\big), \quad Y' = f_1\big(\tfrac{Y}{Y_{\text{ref}}}\big), \quad Z' = f_1\big(\tfrac{Z}{Z_{\text{ref}}}\big),$$

$$\text{and} \quad f_1(c) = \begin{cases} c^{\frac{1}{3}} & \text{for } c > 0.008856 \\ 7.787 \cdot c + \frac{16}{116} & \text{for } c \le 0.008856\,. \end{cases}$$

Usually D65 is specified as the reference white point $(X_{\text{ref}}, Y_{\text{ref}}, Z_{\text{ref}})$ (see Table 12.4). The L^* values are positive and usually within the range $[0, 100]$ (often scaled to $[0, 255]$), but may theoretically be greater. The possible values for a^* and b^* are in the range $[-127, +127]$.

Transformation L*a*b* → CIE XYZ

The reverse transformation from L*a*b* space to XYZ coordinates is defined as follows:

$$\begin{aligned} X &= X_{\text{ref}} \cdot f_2\big(\tfrac{a^*}{500} + Y'\big), \\ Y &= Y_{\text{ref}} \cdot f_2\big(Y'\big), \\ Z &= Z_{\text{ref}} \cdot f_2\big(Y' - \tfrac{b^*}{200}\big), \end{aligned} \tag{12.46}$$

$$\text{where} \quad Y' = \tfrac{L^*+16}{116}$$

$$\text{and} \quad f_2(c) = \begin{cases} c^3 & \text{for } c^3 > 0.008856 \\ \frac{c - 16/116}{7.787} & \text{for } c^3 \le 0.008856\,. \end{cases}$$

The complete Java code for the L*a*b*/XYZ conversion and the implementation of the associated `ColorSpace` class can be found in Progs. 12.10 and 12.11 (pp. 297–298).

L^* a^* b^*

Fig. 12.24
L*a*b* components shown as grayscale images. The contrast of the a^* and b^* images has been increased by 40% for better viewing.

Table 12.5 lists the relation between L*a*b* and XYZ coordinates for selected RGB colors. Figure 12.24 shows the separation of a color image into the corresponding L*a*b* components.

Measuring color differences

Due to its high uniformity with respect to human color perception, the L*a*b* color space is a particularly good choice for determining the difference between colors (the same holds for the L*u*v* space) [40, p. 57]. The difference between two color points $\mathbf{C}_1$ and $\mathbf{C}_2$ can be found by simply measuring the *Euclidean distance* in L*a*b* space,

$$\begin{aligned} \text{ColorDist}_{\text{Lab}}(\mathbf{C}_1, \mathbf{C}_2) &= \|\mathbf{C}_1 - \mathbf{C}_2\| \qquad (12.47) \\ &= \sqrt{(L_1^* - L_2^*)^2 + (a_1^* - a_2^*)^2 + (b_1^* - b_2^*)^2}\,, \end{aligned}$$

where $\mathbf{C}_1 = (L_1^*, a_1^*, b_1^*)$ and $\mathbf{C}_2 = (L_2^*, a_2^*, b_2^*)$.

12.3.3 sRGB

CIE-based color spaces such as L*a*b* (and L*u*v*) are device-independent and have a gamut sufficiently large to represent virtually all visible colors in the CIE XYZ 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" [52]) 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 CIE XYZ 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 L*a*b*, sRGB exhibits a relatively small gamut (see Fig. 12.23), which, however, includes most colors that can be reproduced by

Table 12.6
sRGB tristimulus values **R**, **G**, **B** with reference to the white point D65 (**W**). R, G, B denote the *linearized* component values (which at 0 and 1 are identical to the nonlinear R', G', B' values).

Pt.	R	G	B	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

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 [45]. 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 [97]. In particular, sRGB was defined as the standard default color space for Internet/Web applications by the W3C consortium as part of the HTML 4 specification [96]. 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. 12.3.6 below).

Table 12.6 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 [55] (see Tables 12.3 and 12.4). Together, these values permit the unambiguous mapping of all other colors in the CIE diagram.

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 precorrected 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 CIE XYZ color space by a linear mapping and can thus be computed from X, Y, Z coordinates and vice versa by simple matrix multiplication,

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \boldsymbol{M}_{\text{RGB}} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \boldsymbol{M}_{\text{RGB}}^{-1} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}, \tag{12.48}$$

respectively, with

$$\boldsymbol{M}_{\mathrm{RGB}} = \begin{pmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix}, \tag{12.49}$$

$$\boldsymbol{M}_{\mathrm{RGB}}^{-1} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix}. \tag{12.50}$$

Notice that the three column vectors of $\boldsymbol{M}_{\mathrm{RGB}}^{-1}$ (Eqn. (12.50)) are the coordinates of the primary colors **R**, **G**, **B** (tristimulus values) in XYZ space (cf. Table 12.6) and thus

$$\mathbf{R} = \boldsymbol{M}_{\mathrm{RGB}}^{-1} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{G} = \boldsymbol{M}_{\mathrm{RGB}}^{-1} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \mathbf{B} = \boldsymbol{M}_{\mathrm{RGB}}^{-1} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}. \tag{12.51}$$

Transformation CIE XYZ→sRGB

To transform a given XYZ color to sRGB (Fig. 12.25), we first compute the *linear* R, G, B values by multiplying the (X, Y, Z) coordinate vector with the matrix $\boldsymbol{M}_{\mathrm{RGB}}$ (Eqn. (12.49)),

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \boldsymbol{M}_{\mathrm{RGB}} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}. \tag{12.52}$$

Subsequently, a modified gamma correction (see Sec. 5.7.6) with $\gamma = 2.4$ (which corresponds to an effective gamma value of ca. 2.2) is applied to the linear R, G, B values,

$$R' = f_\gamma(R), \quad G' = f_\gamma(G), \quad B' = f_\gamma(B),$$

$$\text{with} \quad f_\gamma(c) = \begin{cases} 1.055 \cdot c^{\frac{1}{2.4}} - 0.055 & \text{for } c > 0.0031308 \\ 12.92 \cdot c & \text{for } c \le 0.0031308\,. \end{cases} \tag{12.53}$$

The resulting nonlinear sRGB components R', G', B' are limited to the interval $[0, 1]$. To obtain discrete numbers, the R', G', B' values are finally scaled linearly to the 8-bit integer range $[0, 255]$.

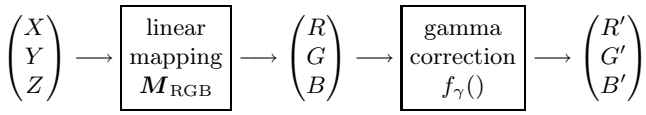


Fig. 12.25
Color transformation from CIE XYZ to sRGB.

Table 12.7
CIE XYZ 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 $\mathbf{R}_{25}$).

		sRGB *nonlinear*			sRGB *linearized*			CIE XYZ		
Pt.	Color	R'	G'	B'	R	G	B	X_{65}	Y_{65}	Z_{65}
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	1.0000	0.0000	0.0000	0.4125	0.2127	0.0193
Y	yellow	1.00	1.0	0.0	1.0000	1.0000	0.0000	0.7700	0.9278	0.1385
G	green	0.00	1.0	0.0	0.0000	1.0000	0.0000	0.3576	0.7152	0.1192
C	cyan	0.00	1.0	1.0	0.0000	1.0000	1.0000	0.5380	0.7873	1.0694
B	blue	0.00	0.0	1.0	0.0000	0.0000	1.0000	0.1804	0.0722	0.9502
M	magenta	1.00	0.0	1.0	1.0000	0.0000	1.0000	0.5929	0.2848	0.9696
W	white	1.00	1.0	1.0	1.0000	1.0000	1.0000	0.9505	1.0000	1.0888
K	50% gray	0.50	0.5	0.5	0.2140	0.2140	0.2140	0.2034	0.2140	0.2330
$\mathbf{R}_{75}$	75% red	0.75	0.0	0.0	0.5225	0.0000	0.0000	0.2155	0.1111	0.0101
$\mathbf{R}_{50}$	50% red	0.50	0.0	0.0	0.2140	0.0000	0.0000	0.0883	0.0455	0.0041
$\mathbf{R}_{25}$	25% red	0.25	0.0	0.0	0.0509	0.0000	0.0000	0.0210	0.0108	0.0010
P	pink	1.00	0.5	0.5	1.0000	0.2140	0.2140	0.5276	0.3812	0.2482

Transformation sRGB→CIE XYZ

To compute 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[14] (Eqn. (12.53)),

$$R = f_\gamma^{-1}(R'), \qquad G = f_\gamma^{-1}(G'), \qquad B = f_\gamma^{-1}(B'), \tag{12.54}$$

$$\text{with} \quad f_\gamma^{-1}(c') = \begin{cases} \left(\frac{c'+0.055}{1.055}\right)^{2.4} & \text{for } c' > 0.03928 \\ \frac{c'}{12.92} & \text{for } c' \le 0.03928. \end{cases} \tag{12.55}$$

Subsequently, the linearized (R, G, B) vector is transformed to XYZ coordinates by multiplication with the inverse of the matrix $\boldsymbol{M}_{\text{RGB}}$ (Eqn. (12.50)); i. e.,

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \boldsymbol{M}_{\text{RGB}}^{-1} \begin{pmatrix} R \\ G \\ B \end{pmatrix}. \tag{12.56}$$

Table 12.7 lists the nonlinear and the linear RGB component values for selected color points. Note that component values of 0 and 1 are not affected by the gamma correction because these values map to themselves. The coordinates of the extremal points of the RGB color cube are therefore identical in nonlinear and linear RGB spaces. However, intermediate values are strongly affected by the gamma correction, as illustrated by the coordinates for the color points **K** ... **P**, which emphasizes the importance of differentiating between linear and nonlinear color coordinates.

[14] See Eqn. (5.36) for a general formulation of the inverse modified gamma function.

Calculating with sRGB values

The sRGB color specification is widely used for representing colors in digital photography, graphics, multimedia, and Internet applications. For example, when a JPEG image is loaded 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. (12.55)) and converted back again with f_γ (Eqn. (12.53)).

Example: color to grayscale conversion

When we initially presented the process of converting RGB colors to grayscale values in Sec. 12.2.1 we simply ignored the possibility that the color component values used in the weighted sum might be nonlinear. However, the variables R, G, B, and Y in the color-to-grayscale conversion defined by Eqn. (12.7),

$$Y = 0.2125 \cdot R + 0.7154 \cdot G + 0.0721 \cdot B, \tag{12.57}$$

implicitly refer to *linear* color and gray values, respectively. We can refine Eqn. (12.57) to obtain the *correct* grayscale conversion from the raw nonlinear sRGB components R', G', B' as

$$Y' = f_\gamma\Big[0.2125 \cdot f_\gamma^{-1}(R') + 0.7154 \cdot f_\gamma^{-1}(G') + 0.0721 \cdot f_\gamma^{-1}(B')\Big], \tag{12.58}$$

with the functions $f_\gamma()$ and $f_\gamma^{-1}()$ as defined in Eqns. (12.53) and (12.55). The result (Y') is again a nonlinear, sRGB-compatible gray value; i. e., 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',

$$(R', G', B') \to (Y', Y', Y'),$$

always creates a gray (colorless) pixel, despite the nonlinearities of the sRGB space. This is due to the fact that the gamma correction (Eqns. (12.53) and (12.55)) 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. 12.21 (b)).

For many applications, however, the following *approximation* to the exact grayscale conversion in Eqn. (12.58) is sufficient. It works without

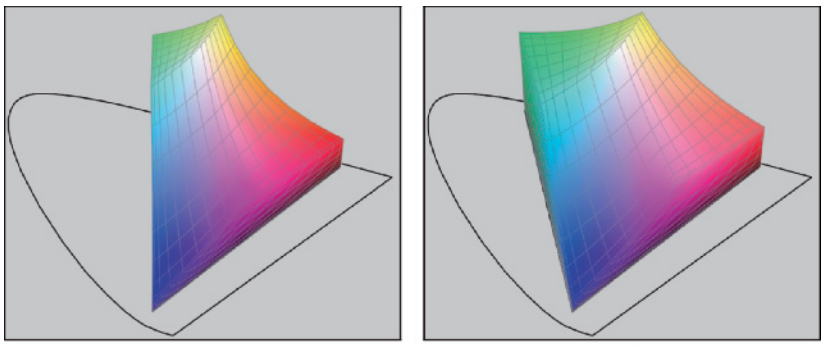


Fig. 12.26
Gamuts for sRGB (a) and Adobe RGB (b) in the three-dimensional CIE XYZ color space.

converting the sRGB values (i. e., directly on the nonlinear R', G', B' components) by computing a linear combination

$$Y' \approx w'_R \cdot R' + w'_G \cdot G' + w'_B \cdot B' \tag{12.59}$$

with a modified set of weights; e. g., $w'_R = 0.309$, $w'_G = 0.609$, $w'_B = 0.082$, as proposed in [78].

12.3.4 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. 12.23), which extends its use particularly to print applications. Figure 12.26 shows the noted difference between the sRGB and Adobe RGB gamuts in three-dimensional CIE XYZ color space.

The white point of Adobe RGB corresponds to the D65 standard (with $x = 0.3127$, $y = 0.3290$), and the gamma value is 2.199 (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 L*a*b* color space and for picture archive applications.

12.3.5 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 $\mathbf{W}_1 = (X_{\mathrm{W1}}, Y_{\mathrm{W1}}, Z_{\mathrm{W1}})$ and $\mathbf{W}_2 = (X_{\mathrm{W2}}, Y_{\mathrm{W2}}, Z_{\mathrm{W2}})$, they can be related by first applying a so-called *chromatic adaptation transformation* (CAT) [49, Ch. 34] in the 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 the alternate white point $\mathbf{W}_2$.

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:

$$X_2 = X_1 \cdot \frac{X_{\mathrm{W2}}}{X_{\mathrm{W1}}}, \qquad Y_2 = Y_1 \cdot \frac{Y_{\mathrm{W2}}}{Y_{\mathrm{W1}}}, \qquad Z_2 = Z_1 \cdot \frac{Z_{\mathrm{W2}}}{Z_{\mathrm{W1}}}\,. \tag{12.60}$$

For example, to convert colors from a system based on the white point $\mathbf{W}_1 = \mathbf{D65}$ to a system relative to $\mathbf{W}_2 = \mathbf{D50}$ (see Table 12.4), the resulting transformation is

$$\begin{aligned} X_{50} &= X_{65} \cdot \frac{X_{\mathbf{D50}}}{X_{\mathbf{D65}}} = X_{65} \cdot \frac{0.964296}{0.950456} = X_{65} \cdot 1.01456\,, \\ Y_{50} &= Y_{65} \cdot \frac{Y_{\mathbf{D50}}}{Y_{\mathbf{D65}}} = Y_{65} \cdot \frac{1.000000}{1.000000} = Y_{65}\,, \\ Z_{50} &= Z_{65} \cdot \frac{Z_{\mathbf{D50}}}{Z_{\mathbf{D65}}} = Z_{65} \cdot \frac{0.825105}{1.088754} = Z_{65} \cdot 0.757843\,. \end{aligned} \tag{12.61}$$

This form of scaling color coordinates in XYZ space is usually not considered a good color adaptation model and is not recommended for high-quality applications.

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

$$\begin{pmatrix} R^* \\ G^* \\ B^* \end{pmatrix} = \boldsymbol{M}_{\mathrm{CAT}} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}, \tag{12.62}$$

where $\boldsymbol{M}_{\mathrm{CAT}}$ is a 3×3 transformation matrix (defined below). 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 $\mathbf{W}_1$) to the new color coordinates X_2, Y_2, Z_2 (w.r.t. white point $\mathbf{W}_2$) takes the form

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \boldsymbol{M}_{\mathrm{CAT}}^{-1} \cdot \begin{pmatrix} \frac{R^*_{\mathrm{W2}}}{R^*_{\mathrm{W1}}} & 0 & 0 \\ 0 & \frac{G^*_{\mathrm{W2}}}{G^*_{\mathrm{W1}}} & 0 \\ 0 & 0 & \frac{B^*_{\mathrm{W2}}}{B^*_{\mathrm{W1}}} \end{pmatrix} \cdot \boldsymbol{M}_{\mathrm{CAT}} \cdot \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix}, \tag{12.63}$$

where $\frac{R^*_{\mathrm{W2}}}{R^*_{\mathrm{W1}}}$, $\frac{G^*_{\mathrm{W2}}}{G^*_{\mathrm{W1}}}$, $\frac{B^*_{\mathrm{W2}}}{B^*_{\mathrm{W1}}}$ are the (constant) ratios of the $R^*G^*B^*$ values of the white points $\mathbf{W}_2$, $\mathbf{W}_1$, respectively; i. e.,

$$\begin{pmatrix} R^*_{\mathrm{W1}} \\ G^*_{\mathrm{W1}} \\ B^*_{\mathrm{W1}} \end{pmatrix} = \boldsymbol{M}_{\mathrm{CAT}} \cdot \begin{pmatrix} X_{\mathrm{W1}} \\ Y_{\mathrm{W1}} \\ Z_{\mathrm{W1}} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} R^*_{\mathrm{W2}} \\ G^*_{\mathrm{W2}} \\ B^*_{\mathrm{W2}} \end{pmatrix} = \boldsymbol{M}_{\mathrm{CAT}} \cdot \begin{pmatrix} X_{\mathrm{W2}} \\ Y_{\mathrm{W2}} \\ Z_{\mathrm{W2}} \end{pmatrix}.$$

The popular "Bradford" model [49, p. 590] for chromatic adaptation specifies the transformation matrix

$$\boldsymbol{M}_{\mathrm{CAT}} = \begin{pmatrix} 0.8951 & 0.2664 & -0.1614 \\ -0.7502 & 1.7135 & 0.0367 \\ 0.0389 & -0.0685 & 1.0296 \end{pmatrix}. \tag{12.64}$$

Inserting this particular $\boldsymbol{M}_{\mathrm{CAT}}$ matrix in Eqn. (12.63) 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 12.4) is

$$\begin{aligned} \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} &= \boldsymbol{M}_{50|65} \cdot \begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix} \\ &= \begin{pmatrix} 1.047884 & 0.022928 & -0.050149 \\ 0.029603 & 0.990437 & -0.017059 \\ -0.009235 & 0.015042 & 0.752085 \end{pmatrix} \cdot \begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix}, \end{aligned} \tag{12.65}$$

and conversely from D50-based to D65-based colors (i. e., $\mathbf{W}_1 = \mathbf{D50}$, $\mathbf{W}_2 = \mathbf{D65}$),

$$\begin{aligned} \begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix} &= \boldsymbol{M}_{65|50} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} = \boldsymbol{M}_{50|65}^{-1} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} \\ &= \begin{pmatrix} 0.955513 & -0.023079 & 0.063190 \\ -0.028348 & 1.009992 & 0.021019 \\ 0.012300 & -0.020484 & 1.329993 \end{pmatrix} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix}. \end{aligned} \tag{12.66}$$

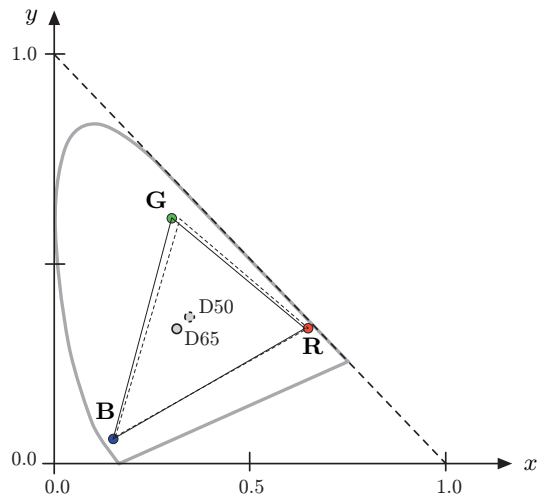


Fig. 12.27
Bradford chromatic adaptation from white point D65 to D50. The solid triangle represents the original RGB gamut for white point D65, with the primaries $(\mathbf{R}, \mathbf{G}, \mathbf{B})$ located at the corner points. The dashed triangle is the corresponding gamut after chromatic adaptation to white point D50.

Table 12.8
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. (12.65).

		sRGB			XYZ (**D65**)			XYZ (**D50**)		
Pt.	Color	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
$\mathbf{R}_{75}$	75% red	0.75	0.0	0.0	0.2155	0.1111	0.0101	0.2279	0.1163	0.0073
$\mathbf{R}_{50}$	50% red	0.50	0.0	0.0	0.0883	0.0455	0.0041	0.0933	0.0476	0.0030
$\mathbf{R}_{25}$	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

Fig. 12.27 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 12.8.

The Bradford model is a widely used chromatic adaptation scheme but several similar procedures have been proposed (see also Exercise 12.5). Generally speaking, chromatic adaptation and related problems have a long history in color engineering and are still active fields of scientific research [106, Sec. 5.12].

12.3.6 Colorimetric Support in Java

sRGB colors in Java

sRGB is the standard color space in Java; i. e., the components of color objects and RGB color images are gamma-corrected, *nonlinear* R', G', B' values (see Fig. 12.25). 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. (12.53) and (12.55)).

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 ICC[15] "color management architecture", which uses a CIE XYZ-based device-independent color space called the "profile connection space" (PCS) [51, 54]. 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. 12.3.6) 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 12.28 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 from the sRGB specification, the ICC specifies **D50** (and *not* D65) as the illuminant white point for its default PCS color space (see Table 12.4). 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 12.9) are different from the ones given for white point D65 (see Table 12.6)! This is a frequent cause of confusion since the sRGB 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. 12.3.5) is used to convert between XYZ color coordinates that are measured with respect to different white points. The ICC specification [51] recommends a linear chromatic adaptation based on the Bradford model to convert between the D65-related

[15] International Color Consortium (ICC, www.color.org).

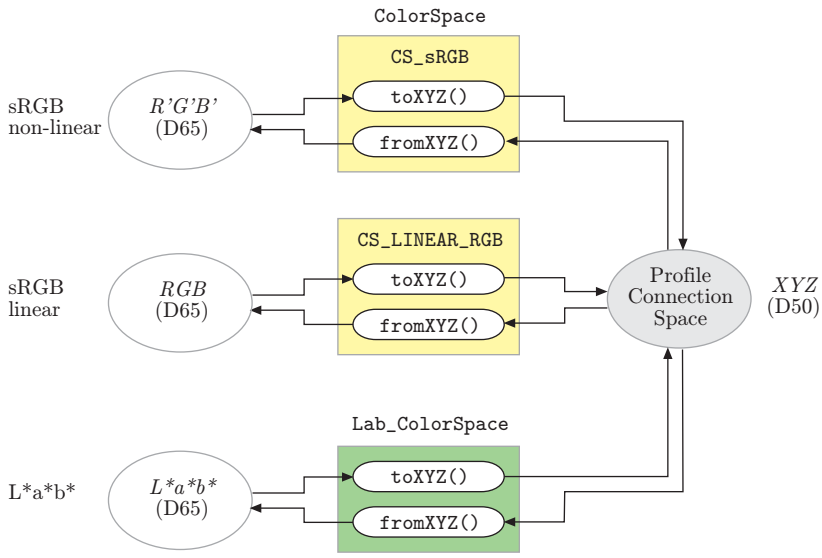


Fig. 12.28
XYZ-based color conversion in Java. `ColorSpace` objects implement the methods `fromCIEXYZ()` and `toCIEXYZ()` to convert color vectors from and to the CIE XYZ color space, respectively. Colorimetric transformations between color spaces can be accomplished as a two-step process via the XYZ space. For example, to convert from sRGB to $L^*a^*b^*$, the sRGB color is first converted to XYZ and subsequently from XYZ to $L^*a^*b^*$. 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

Table 12.9
Color coordinates for sRGB primaries and the white point in Java's default XYZ color space. The white point **W** is equal to D50.

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 RGB→XYZ_{65} transformation by matrix $\boldsymbol{M}_{\text{RGB}}$ (Eqns. (12.49) and (12.50)) and the chromatic adaptation transformation XYZ_{65}→XYZ_{50} defined by the matrix $\boldsymbol{M}_{50|65}$ (Eqn. (12.65)),

$$\begin{aligned}\begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} &= \boldsymbol{M}_{50|65} \cdot \boldsymbol{M}_{\text{RGB}}^{-1} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} \\ &= \left(\boldsymbol{M}_{\text{RGB}} \cdot \boldsymbol{M}_{65|50}\right)^{-1} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} \\ &= \begin{pmatrix} 0.436131 & 0.385147 & 0.143033 \\ 0.222527 & 0.716878 & 0.060600 \\ 0.013926 & 0.097080 & 0.713871 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}, \end{aligned} \tag{12.67}$$

and, in the reverse direction,

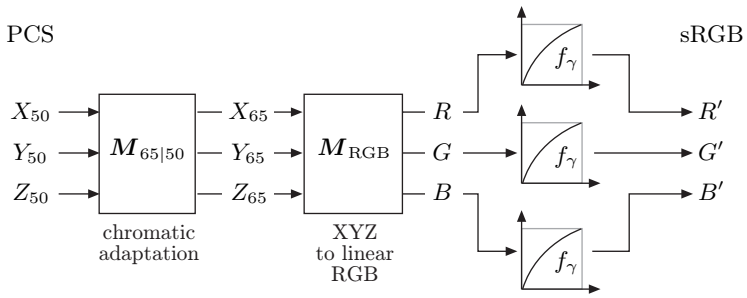


Fig. 12.29
Transformation from D50-based PCS coordinates (X_{50}, Y_{50}, Z_{50}) to nonlinear sRGB values (R', G', B').

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \boldsymbol{M}_{\mathrm{RGB}} \cdot \boldsymbol{M}_{65|50} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} \tag{12.68}$$

$$= \begin{pmatrix} 3.133660 & -1.617140 & -0.490588 \\ -0.978808 & 1.916280 & 0.033444 \\ 0.071979 & -0.229051 & 1.405840 \end{pmatrix} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix}.$$

Equations (12.67) and (12.68) 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 12.29 illustrates the complete transformation from D50-based PCS coordinates to nonlinear sRGB values.

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 the properties of entire color spaces.
- `ColorModel`: describes the structure of color images; e. g., full-color images or indexed-color images, as used in Sec. 12.1.2 (see Prog. 12.3).

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 in addition accept alpha (transparency) values. In addition, the `Color` class offers two useful static methods, `RGBtoHSB()` and `HSBtoRGB()`, for converting between sRGB and HSV[16] colors (see Sec. 12.2.3, p. 261).

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 CIE XYZ 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 (pink) to the corresponding (X_{50}, Y_{50}, Z_{50}) coordinates in Java's (D50-based) CIE XYZ 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 *nonlinear* R', G', B' components, as specified in [52],
- `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).

The color space objects returned by `getInstance()` are all instances of `ICC_ColorSpace`, which is the only implementation of (the abstract

[16] The HSV color space is referred to as "HSB" (hue, saturation, *brightness*) in the Java API.

class) ColorSpace provided by the Java standard API.Other color spaces can be implemented by creating additional implementations (subclasses) of ColorSpace, as demonstrated for L*a*b* in the example below.

An L*a*b* color space implementation

In the following, we show a complete implementation of the L*a*b* color space, which is not available in the current Java API, based on the specification given in Sec. 12.3.2. For this purpose, we define a 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 L*a*b* and sRGB (Progs. 12.10 and 12.11). These conversions are performed in two steps via XYZ coordinates, where care must be taken regarding the right choice of the associated white point (L*a*b* is based on D65 and Java XYZ on D50). The following examples demonstrate the principal use of the new Lab_ColorSpace class:

```
1 ColorSpace LABcsp = new LabColorSpace();
2 float[] cyan_sRGB = {0.0f, 1.0f, 1.0f};
3 // sRGB→L*a*b*:
4 float[] cyan_LAB = LABcsp.fromRGB(cyan_sRGB)
5 // L*a*b*→XYZ:
6 float[] cyan_XYZ = LABcsp.toXYZ(cyan_LAB);
```

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. ICC 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 [51], which is identical to ISO standard 15076 [54]. Profiles are thus a key element in the process of digital color management [102].

The standard Java 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.[17]

[17] In the Java API, the transformations for all standard color space types are specified through corresponding ICC profiles, which are part of the standard Java distribution (files sRGB.pf, etc., usually contained in jdk.../jre/lib/cmm). However, up to the current Java release (1.6.0), the methods toCIEXYZ() and fromCIEXYZ() do *not* properly invert; i. e., *col* ≠ csp.fromCIEXYZ(csp.toCIEXYZ(*col*)) for a color space object csp. (This has been a documented Java problem for some time.) A "clean" implemen-

```
1  public class LabColorSpace extends ColorSpace {
2
3    // D65 reference white point
4    static final double Xref = Illuminant.D65.X; // 0.950456;
5    static final double Yref = Illuminant.D65.Y; // 1.000000;
6    static final double Zref = Illuminant.D65.Z; // 1.088754;
7
8    // create two chromatic adaptation objects
9    ChromaticAdaptation catD65toD50 =
10       new BradfordAdaptation(Illuminant.D65, Illuminant.D50);
11   ChromaticAdaptation catD50toD65 =
12       new BradfordAdaptation(Illuminant.D50, Illuminant.D65);
13
14   // sRGB color space for methods toRGB() and fromRGB()
15   static final ColorSpace sRGBcs
16         = ColorSpace.getInstance(CS_sRGB);
17
18   // constructor method:
19   public LabColorSpace(){
20     super(TYPE_Lab,3);
21   }
22
23   // XYZ→CIELab: returns D65-related L*a*b values
24   // from D50-related XYZ values:
25   public float[] fromCIEXYZ(float[] XYZ50) {
26     float[] XYZ65 = catD50toD65.apply(XYZ50);
27     double xx = f1(XYZ65[0] / Xref);
28     double yy = f1(XYZ65[1] / Yref);
29     double zz = f1(XYZ65[2] / Zref);
30
31     float L = (float)(116 * yy - 16);
32     float a = (float)(500 * (xx - yy));
33     float b = (float)(200 * (yy - zz));
34     return new float[] {L, a, b};
35   }
36
37   // CIELab→XYZ: returns D50-related XYZ values
38   // from D65-related L*a*b* values:
39   public float[] toCIEXYZ(float[] Lab) {
40     double yy = ( Lab[0] + 16 ) / 116;
41     float X65 = (float) (Xref * f2(Lab[1] / 500 + yy));
42     float Y65 = (float) (Yref * f2(yy));
43     float Z65 = (float) (Zref * f2(yy - Lab[2] / 200));
44     float[] XYZ65 = new float[] {X65, Y65, Z65};
45     return catD65toD50.apply(XYZ65);
46   }
47
48 // continued...
```

Program 12.10
Java implementation of the L*a*b* color space. `Lab_ColorSpace` is a subclass of the standard AWT class `ColorSpace`. The conversion from the profile connection space (XYZ) to L*a*b* (Eqn. (12.45)) is implemented by the method `fromCIEXYZ()`, where a chromatic adaptation from D50 to D65 is applied first (line 26). The auxiliary method `f1()` (defined in line 50) performs the required gamma correction (lines 27–29). The method `toCIEXYZ()` implements the reverse transformation from L*a*b* to XYZ, where the method `f2()` (defined in line 58) does the inverse gamma correction (lines 41–43), followed by the chromatic adaptation from D65 to D50 in line 45. The definitions of the classes `Illuminant`, `ChromaticAdaptation`, and `BradfordAdaptation` can be found in the source code section of the book's Website.

Program 12.11 Java implementation of the L*a*b* color space (continued). Auxiliary methods `f1()` and `f2()` implementing the forward and inverse gamma corrections, respectively (as defined in Eqns. (12.45) and (12.46)). The methods `toRGB()` and `fromRGB()` perform the conversions to and from sRGB in two steps via XYZ coordinates.

```
49  // Gamma correction (forward)
50  double f1 (double c) {
51    if (c > 0.008856)
52      return Math.pow(c, 1.0 / 3);
53    else
54      return (7.787 * c) + (16.0 / 116);
55  }
56
57  // Gamma correction (inverse)
58  double f2 (double c) {
59    double c3 = Math.pow(c, 3.0);
60    if (c3 > 0.008856)
61      return c3;
62    else
63      return (c - 16.0 / 116) / 7.787;
64  }
65
66  //sRGB→CIELab
67  public float[] fromRGB(float[] sRGB) {
68    float[] XYZ50 = sRGBcs.toCIEXYZ(sRGB);
69    return this.fromCIEXYZ(XYZ50);
70  }
71
72  //CIELab→sRGB
73  public float[] toRGB(float[] Lab) {
74    float[] XYZ50 = this.toCIEXYZ(Lab);
75    return sRGBcs.fromCIEXYZ(XYZ50);
76  }
77
78 } // end of class LabColorSpace
```

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.[18] 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 CIE XYZ or sRGB values, as illustrated by the following example:

```
1 // load the scanner's ICC profile
2 ICC_ColorSpace scannerCS = new
3     ICC_ColorSpace(ICC_ProfileRGB.getInstance("scanner.icc"));
4 // convert to RGB color
```

tation of the sRGB color space can be found in the source code section of this book's Website.

[18] ICC profile files may also come with extensions `.icm` or `.pf` (as in the Java distribution).

```
5 float[] RGBColor = scannerCS.toRGB(scannerColor);
6 // convert to XYZ color
7 float[] XYZColor = scannerCS.toCIEXYZ(scannerColor);
```

Similarly, we can compute the accurate color values to be sent to the monitor by creating a suitable color space object from this device's ICC profile.

12.4 Statistics of Color Images

12.4.1 How Many 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. 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 (one-dimensional) 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. 12.12. 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.[19] Sorting of the one-dimensional array in Prog. 12.12 is accomplished (in line 9) with the generic Java method `Arrays.sort()`, which is implemented very efficiently.

12.4.2 Color Histograms

We briefly touched on histograms of color images in Sec. 4.5, where we only considered the one-dimensional 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();
```

[19] Java arrays implement the methods of the root class `Object`, including the `clone()` method specified by the `Cloneable` interface (see also Appendix B.2.5).

Program 12.12
Counting the colors contained in an RGB image. The method `countColors()` first creates a copy of the one-dimensional RGB (`int`) pixel array (line 8), then sorts that array, and finally counts the transitions between contiguous blocks of identical colors.

```
1  import ij.process.ColorProcessor;
2  import java.util.Arrays;
3
4  public class ColorStatistics {
5
6    static int countColors (ColorProcessor cp) {
7      // duplicate pixel array and sort
8      int[] pixels = ((int[]) cp.getPixels()).clone();
9      Arrays.sort(pixels);
10
11     int k = 1; // image contains at least one color
12     for (int i = 0; i < pixels.length-1; i++) {
13       if (pixels[i] != pixels[i+1])
14         k = k + 1;
15     }
16     return k;
17   }
18
19 } // end of class ColorStatistics
```

As an alternative, one could compute the individual intensity histograms of the three color channels, although (as discussed in Sec. 4.5.2) 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 $L^*a^*b^*$.

A *full* histogram of an RGB image is three-dimensional 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 large[20] but also difficult to visualize.

2D color histograms

A useful alternative to the full 3D RGB histogram are two-dimensional histogram projections (Fig. 12.30). 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

$$\begin{aligned} H_{\text{RG}}(r,g) &\leftarrow \text{number of pixels with } I_{\text{RGB}}(u,v)=(r,g,*), \\ H_{\text{RB}}(r,b) &\leftarrow \text{number of pixels with } I_{\text{RGB}}(u,v)=(r,*,b), \\ H_{\text{GB}}(g,b) &\leftarrow \text{number of pixels with } I_{\text{RGB}}(u,v)=(*,g,b), \end{aligned} \tag{12.69}$$

where $*$ denotes an arbitrary component value. The result is, independent of the original image size, a set of two-dimensional histograms of

[20] 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.

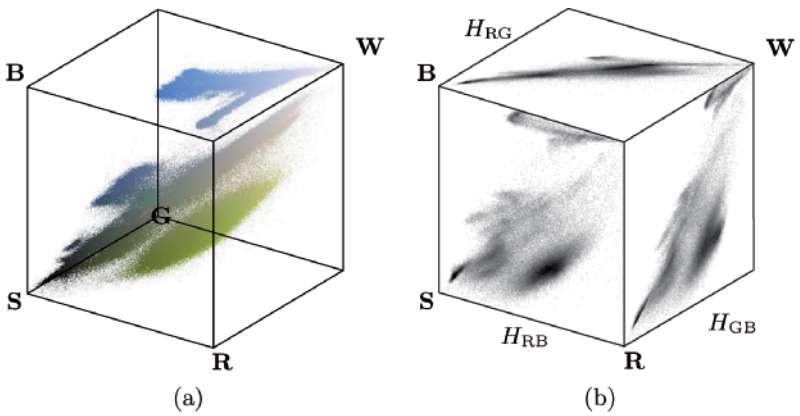


Fig. 12.30
Two-dimensional RGB histogram projections. Three-dimensional RGB cube illustrating an image's color distribution (a). The color points indicate the corresponding pixel colors and not the color frequency. The combined histograms for red-green (H_{RG}), red-blue (H_{RB}), and green-blue (H_{GB}) are 2D projections of the 3D histogram. The corresponding image is shown in Fig. 12.9 (a).

```
1  static int[][] get2dHistogram
2                 (ColorProcessor cp, int c1, int c2) {
3    // c1, c2: R = 0, G = 1, B = 2
4    int[] RGB = new int[3];
5    int[][] H = new int[256][256]; // histogram array H[c1][c2]
6
7    for (int v = 0; v < cp.getHeight(); v++) {
8      for (int u = 0; u < cp.getWidth(); u++) {
9        cp.getPixel(u, v, RGB);
10       int i = RGB[c1];
11       int j = RGB[c2];
12       // increment corresponding histogram cell
13       H[j][i]++; // i runs horizontal, j runs vertical
14     }
15   }
16   return H;
17 }
```

Program 12.13
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 two-dimensional `int` array. The method is defined in class `ColorStatistics` (Prog. 12.12).

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. 12.13).

As the examples in Fig. 12.31 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).

12.5 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

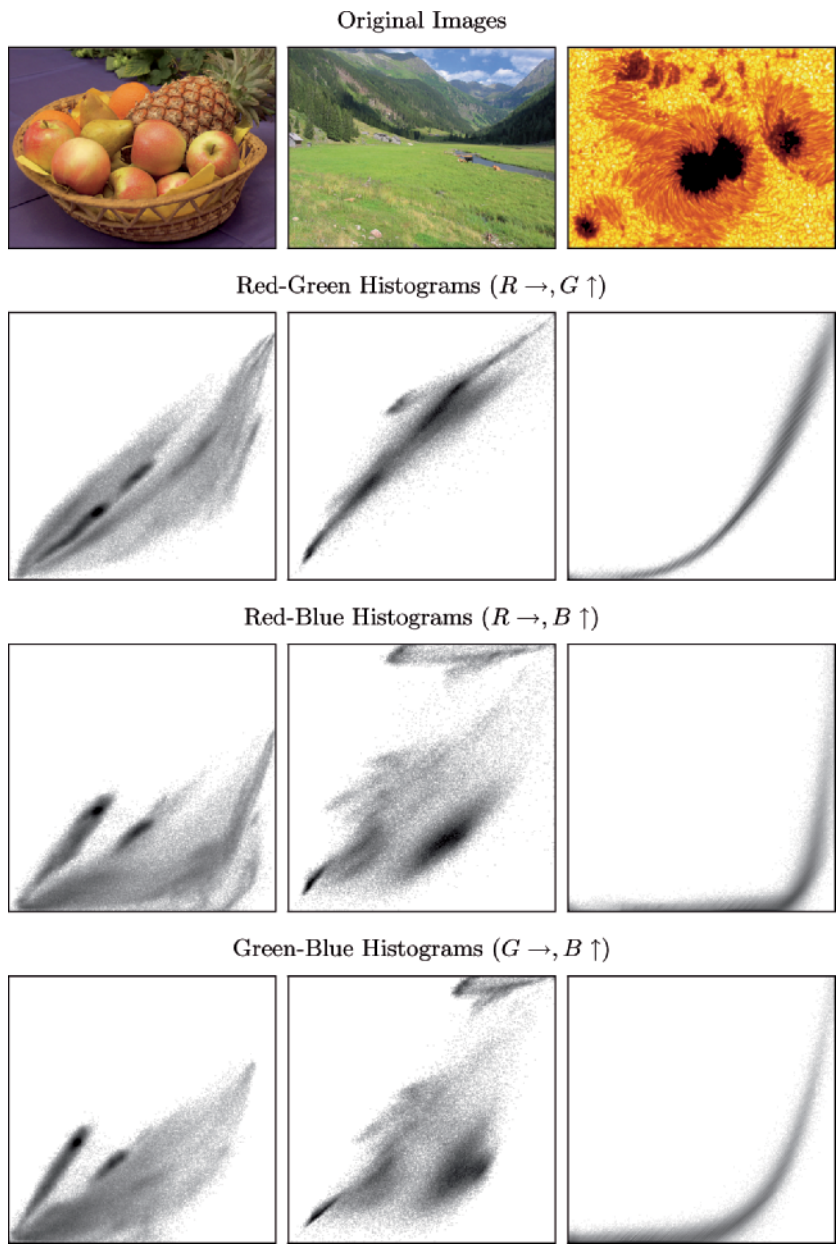


Fig. 12.31
Combined color histogram examples. For better viewing, the images are inverted (dark regions indicate high frequencies) and the gray value corresponds to the logarithm of the histogram entries (scaled to the maximum entries).

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 $\mathcal{C} = \{\mathbf{C}_1, \mathbf{C}_2, \ldots \mathbf{C}_m\}$, 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 $\mathcal{C}' = \{\mathbf{C}'_1, \mathbf{C}'_2, \ldots \mathbf{C}'_n\}$, 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.

12.5.1 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 $[0 \ldots m-1]$ is independently converted to the new range $[0 \ldots n-1]$, in the simplest case by a linear quantization in the form

$$c'_i \leftarrow \left\lfloor c_i \cdot \frac{n}{m} \right\rfloor, \tag{12.70}$$

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 (Fig. 12.32 (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 illustrated in Fig. 12.32 (b). In this case, $m = 256$ for all color components, $n_{\text{red}} = n_{\text{green}} = 8$, and $m_{\text{blue}} = 4$. This conversion to 3:3:2-packed single byte colors can be accomplished efficiently with simple bit operations, as illustrated in the Java code segment in Prog. 12.14. Naturally, due to the small number of colors available with this encoding (Fig. 12.33), the resulting image quality is poor.

Unlike the techniques described in the following, scalar quantization does not take into account the distribution of colors in the original image.

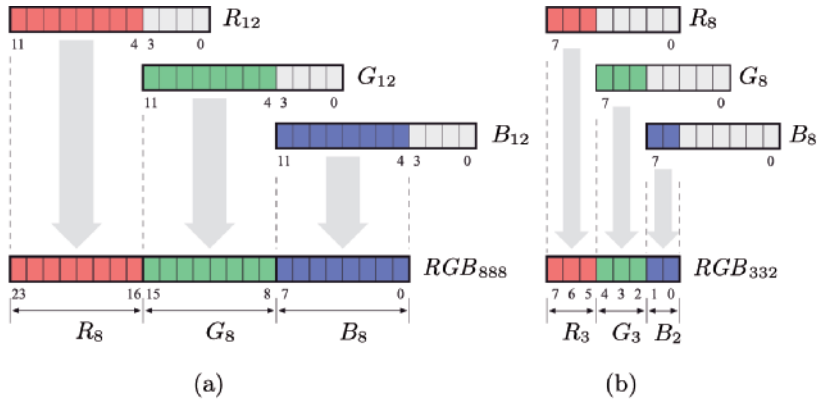


Fig. 12.32
Scalar quantization of color components by truncating lower bits. Quantization of 3 × 12-bit to 3 × 8-bit colors (a). Quantization of 3 × 8-bit to 3:3:2-packed 8-bit colors (b).

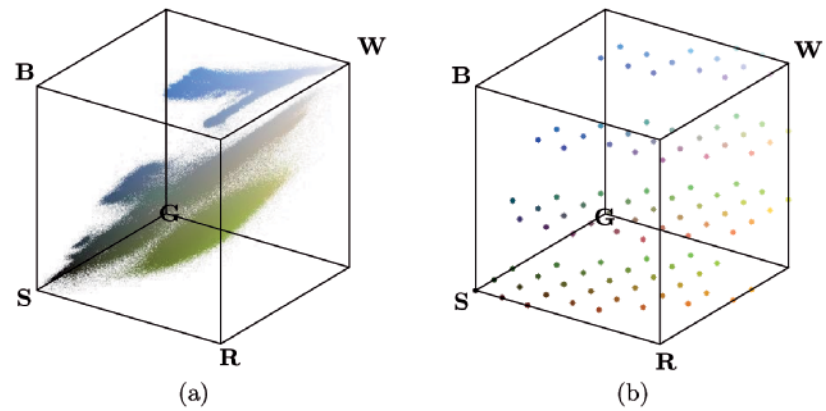


Fig. 12.33
Color distribution after a scalar 3:3:2 quantization. Distribution of the original 226,321 colors (a) and the remaining 8 × 8 × 4 = 256 colors after 3:3:2 quantization (b) in the RGB color cube.

Program 12.14
3:3:2 quantization of a 24-bit RGB color pixel using bit operations (see also Exercise 12.9).

```
1 ColorProcessor cp = (ColorProcessor) ip;
2 int C = cp.getPixel(u, v); // 24-bit color pixel
3 int R = (C & 0x00FF0000) >> 16;
4 int G = (C & 0x0000FF00) >> 8;
5 int B = (C & 0x000000FF);
6 byte RGB = (byte)  // 8-bit color pixel (3:3:2-packed)
7     (R & 0xE0 | (G & 0xE0) >> 3 | (B & 0xC0) >> 6);
```

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.

12.5.2 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 $\mathcal{C} = \{\mathbf{C}_1, \mathbf{C}_2, \ldots \mathbf{C}_m\}$, the task of vector quantization is

(a) to find a set of n representative color vectors $\mathcal{C}' = \{\mathbf{C}'_1, \mathbf{C}'_2, \ldots \mathbf{C}'_n\}$ and
(b) 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.

Populosity algorithm

The populosity algorithm[21] [43] 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. 12.4.1 can be used to determine the n most frequent image colors. Each original pixel $\mathbf{C}_i$ is then replaced by the closest representative color vector in $\mathcal{C}'$; i. e., 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.

Median-cut algorithm

The median-cut algorithm [43] 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.[22] The initial histogram

[21] Sometimes also called the "popularity" algorithm.

[22] 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.

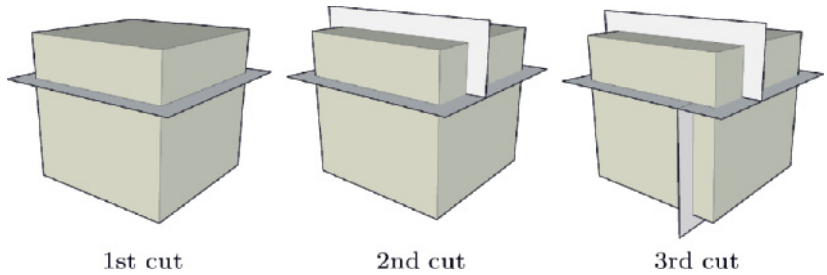


Fig. 12.34
Median-cut algorithm. The RGB color space is recursively split into smaller cubes along one of the color axes.

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. 12.34).

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 12.1–12.3 and a corresponding Java implementation can be found in the source code section of this book's Website.

Octree algorithm

Similar to the median-cut algorithm, this method is also based on partitioning the three-dimensional color space into cells of varying size. The octree algorithm [36] 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:

```
1:  MedianCut(I, K_max)
        I: color image, K_max: max. number of quantized colors
        Returns a new quantized image with at most K_max colors.
2:      C_R ← FindRepresentativeColors(I, K_max)
3:      return QuantizeImage(I, C_R)
---------------------------------------------------------------------
4:  FindRepresentativeColors(I, K_max)
5:      Determine C = {c_1, c_2, ... c_K}, the set of K distinct colors in I,
            where each color instance c_i is a tuple ⟨red, grn, blu, cnt⟩ consist-
            ing of the RGB color components (red, grn, blu) and the number
            of pixels (cnt) for that particular color.
6:      if |C| ≤ K_max then
7:          C_R ← C.
8:      else
            Create a color box b_0 at level 0 that contains all image colors C
            and make it the initial element in the set of color boxes B:
9:          b_0 ← CreateColorBox(C, 0)                    ▷ see Alg. 12.2
10:         B ← {b_0}                                     ▷ initial set of color boxes
11:         k ← 1
12:         done ← false
13:         while k < N_max and not done do
14:             b ← FindBoxToSplit(B)                     ▷ see Alg. 12.2
15:             if b ≠ nil then
16:                 ⟨b_1, b_2⟩ ← SplitBox(b)              ▷ see Alg. 12.2
17:                 B ← B − {b}                           ▷ remove b from B
18:                 B ← B ∪ {b_1, b_2}                    ▷ insert b_1, b_2 into B
19:                 k ← k + 1
20:             else                                      ▷ no more boxes to split
21:                 done ← true
            Determine the average color inside each color box in set B:
22:         C_R ← {c_j = AverageColors(b_j) | b_j ∈ B}    ▷ see Alg. 12.3
23:     return C_R.
---------------------------------------------------------------------
24: QuantizeImage(I, C_R)
        Returns a new image with color pixels from I replaced by their closest
        representative colors in C_R:
25:     Create a new image I' the same size as I.
26:     for all (u, v) do
27:         Find the color c ∈ C_R that is "closest" to I(u, v) (e. g., using the
                Euclidean distance in RGB space).
28:         I'(u, v) ← c
29:     return I'.
```

1. If the number of nodes is less than K, 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.

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

Algorithm 12.1
Median-cut color quantization (*part 1 of 3*). The input image 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 ($\mathcal{C}_R$) that are subsequently used by procedure QuantizeImage() to quantize the original image I. Note that (unlike in most common implementations) no prequantization is applied to the original image colors.

Algorithm 12.2
Median-cut color quantization (*part 2 of 3*).

1: CreateColorBox($\mathcal{C}, m$)
Creates and returns a new color box containing the colors $\mathcal{C}$. A color box $\boldsymbol{b}$ is a tuple $\langle \mathsf{colors}, \mathsf{level}, \mathsf{rmin}, \mathsf{rmax}, \mathsf{gmin}, \mathsf{gmax}, \mathsf{bmin}, \mathsf{bmax} \rangle$, where colors is the set of image colors represented by the box, level denotes the split-level, and rmin, ... bmax describe the color boundaries of the box in RGB space.

2: Find the RGB extrema of all colors in this box:

$$\left.\begin{aligned} r_{\min} &\leftarrow \min\ \mathsf{red}(\boldsymbol{c}) \\ r_{\max} &\leftarrow \max\ \mathsf{red}(\boldsymbol{c}) \\ g_{\min} &\leftarrow \min\ \mathsf{grn}(\boldsymbol{c}) \\ g_{\max} &\leftarrow \max\ \mathsf{grn}(\boldsymbol{c}) \\ b_{\min} &\leftarrow \min\ \mathsf{blu}(\boldsymbol{c}) \\ b_{\max} &\leftarrow \max\ \mathsf{blu}(\boldsymbol{c}) \end{aligned}\right\} \text{ for all colors } \boldsymbol{c} \in \mathcal{C}$$

3: Create a new color box $\boldsymbol{b}$:
$\boldsymbol{b} \leftarrow \langle \mathcal{C}, m, r_{\min}, r_{\max}, g_{\min}, g_{\max}, b_{\min}, b_{\max} \rangle$

4: **return** $\boldsymbol{b}$.

5: 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.
Let $\mathcal{B}_s$ be the set of all color boxes that can be split (i. e., contain at least 2 different colors):

6: $\mathcal{B}_s \leftarrow \{\, \boldsymbol{b} \mid \boldsymbol{b} \in \mathcal{B} \wedge |\mathsf{colors}(\boldsymbol{b})| \geq 2 \}$
7: **if** $\mathcal{B}_s = \{\}$ **then** ▷ no splittable box was found
8: **return** nil.
9: **else**
10: Select a box $\boldsymbol{b}_x \in \mathcal{B}_s$, such that $\mathsf{level}(\boldsymbol{b}_x)$ is a minimum.
11: **return** $\boldsymbol{b}_x$.

12: SplitBox($\boldsymbol{b}$)
Splits the color box $\boldsymbol{b}$ at the median plane perpendicular to its longest dimension and returns a pair of new color boxes.

13: $m \leftarrow \mathsf{level}(\boldsymbol{b})$
14: $d \leftarrow$ FindMaxBoxDimension($\boldsymbol{b}$) ▷ see Alg. 12.3
15: $\mathcal{C} \leftarrow \mathsf{colors}(\boldsymbol{b})$
16: From all color samples in $\mathcal{C}$ determine x_{med} as the **median** of the color distribution along dimension d.
17: Partition the set $\mathcal{C}$ into two disjoint sets $\mathcal{C}_1$ and $\mathcal{C}_2$ by splitting at x_{med} along dimension d.
18: $\boldsymbol{b}_1 \leftarrow$ CreateColorBox($C_1, m+1$)
19: $\boldsymbol{b}_2 \leftarrow$ CreateColorBox($C_2, m+1$)
20: **return** $\langle \boldsymbol{b}_1, \boldsymbol{b}_2 \rangle$.

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.

Algorithm 12.3
Median-cut color quantization (*part 3 of 3*).

```
1:  AverageColors(b)
    Returns the average color c_avg for the pixels represented by the color box b.
2:  C ← colors(b)
3:  n ← 0,  r_sum ← 0,  g_sum ← 0,  b_sum ← 0
4:  for all c ∈ C do
5:      k ← cnt(c)
6:      n ← n + k
7:      r_sum ← r_sum + k · red(c)
8:      g_sum ← g_sum + k · grn(c)
9:      b_sum ← b_sum + k · blu(c)
10: r_avg ← (1/n) · r_sum,  g_avg ← (1/n) · g_sum,  b_avg ← (1/n) · b_sum
11: c_avg ← ⟨r_avg, g_avg, b_avg⟩
12: return c_avg.

13: FindMaxBoxDimension(b)
    Returns the largest dimension of the color box b (i. e., Red, Green, or Blue).
14: size_r = rmax(b) − rmin(b)
15: size_g = gmax(b) − gmin(b)
16: size_b = bmax(b) − bmin(b)
17: size_max = max(size_r, size_g, size_b)
18: if size_max = size_r then
19:     return Red.
20: else if size_max = size_g then
21:     return Green.
22: else
23:     return Blue.
```

Figure 12.35 shows the resulting color distributions in RGB space after applying the median-cut and octree algorithms. In both cases, the original image (Fig. 12.31 (b)) 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 quantization, the resulting distances between the original pixels and the quantized colors are shown in Fig. 12.36. 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 considerably smaller, although the latter creates several large deviations, particularly inside the colored foreground regions and the forest region in the background.

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

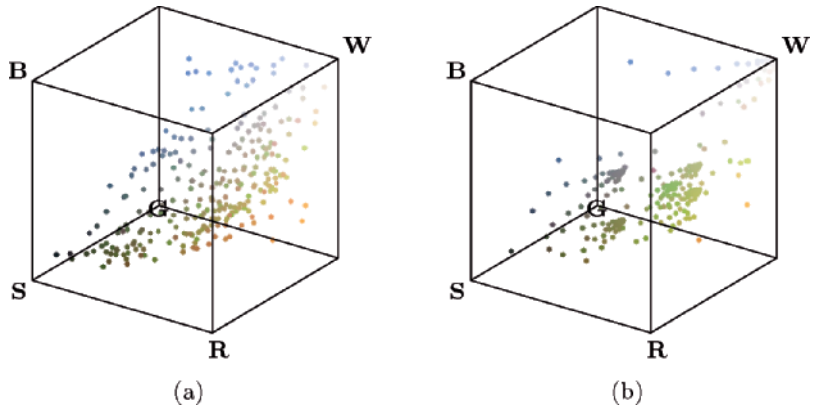


Fig. 12.35
Color distribution after application of the median-cut (a) and octree (b) algorithms. In both cases, the set of 226,321 colors in the original image (Fig. 12.31 (b)) was reduced to 256 representative colors.

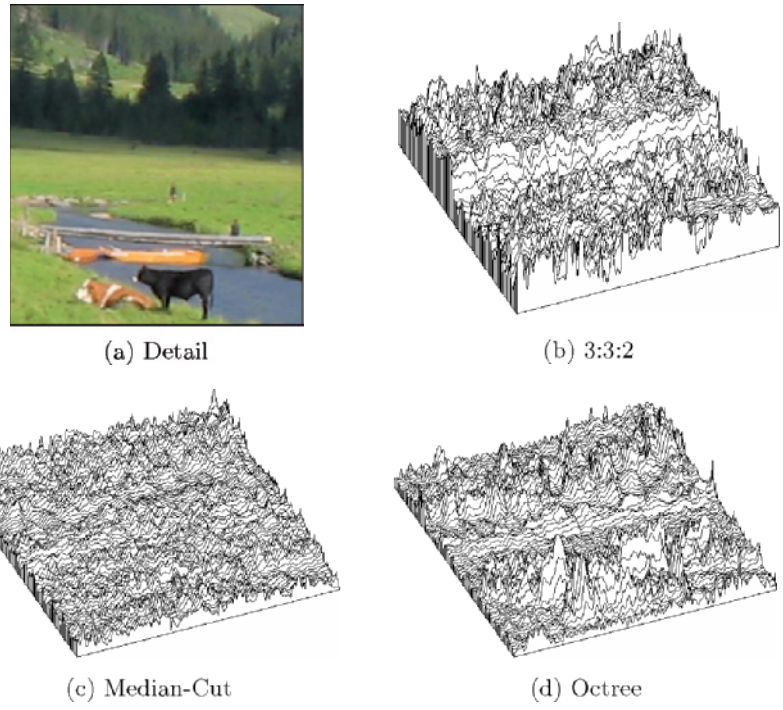


Fig. 12.36
Quantization errors. Original image (a), distance between original and quantized color pixels for scalar 3:3:2 quantization (b), median-cut (c), and octree (d) algorithms.

to use only 10% of randomly selected pixels to obtain a high probability that none of the important colors is lost.

In addition to the color quantization methods described above, 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 [93].

12.6 Exercises

Exercise 12.1. Create an ImageJ plugin that rotates the individual components of an RGB color image; i. e., $R \to G \to B \to R$.

Exercise 12.2. 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 12.3. Show that a "desaturated" RGB pixel produced in the form $(r, g, b) \to (y, y, y)$, where y is the equivalent luminance value (see Eqn. (12.8)), has the luminance y as well.

Exercise 12.4. 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 12.5. For chromatic adaptation (defined in Eqn. (12.63)), transformation matrices other than the Bradford model (Eqn. (12.64)) have been proposed; e. g. [97],

$$\boldsymbol{M}_{\text{Sharp}} = \begin{pmatrix} 1.2694 & -0.0988 & -0.1706 \\ -0.8364 & 1.8006 & 0.0357 \\ 0.0297 & -0.0315 & 1.0018 \end{pmatrix} \quad \text{and}$$

$$\boldsymbol{M}_{\text{CMC}} = \begin{pmatrix} 0.7982 & 0.3389 & -0.1371 \\ -0.5918 & 1.5512 & 0.0406 \\ 0.0008 & -0.0239 & 0.9753 \end{pmatrix}.$$

Derive the complete chromatic adaptation transformations $\boldsymbol{M}_{50|65}$ and $\boldsymbol{M}_{65|50}$ for converting between D65 and D50 colors, analogous to Eqns. (12.65) and (12.66), for each of the transformation matrices above.

Exercise 12.6. Implement the conversion of an sRGB color image to a colorless (grayscale) sRGB image using the three methods in Eqns. (12.57) (incorrectly applying standard weights to nonlinear $R'G'B'$ components), (12.58) (exact computation), and (12.59) (approximation using nonlinear components and modified weights). Compare the results by computing difference images, and also determine the total errors.

Exercise 12.7. Pseudocolors are sometimes used for displaying grayscale images (i. e., for viewing medical images with high dynamic 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 12.8. 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, two-dimensional color histograms (as described in Sec. 12.4.2) 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. Simplify the 3:3:2 quantization given in Prog. 12.14 such that only a single bit mask/shift step is performed for each color component.

Exercise 12.10. The median-cut algorithm for color quantization (Sec. 12.5.2) is implemented in the *Independent JPEG Group's*[23] `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.

[23] www.ijg.org.

13

Introduction to Spectral Techniques

The following three chapters deal with the representation and analysis of images in the frequency domain, based on the decomposition of image signals into sine and cosine functions—which are also known as *harmonic* functions—using the well-known *Fourier transform*. Students often consider this a difficult topic, mainly because of its mathematical flavor and that its practical applications are not immediately obvious. Indeed, most common operations and methods in digital image processing can be sufficiently described in the original signal or image space without even mentioning spectral techniques. This is the reason why we pick up this topic relatively late in this text.

While spectral techniques were often used to improve the efficiency of image-processing operations, this has become increasingly less important due to the high power of modern computers. There exist, however, some important effects, concepts, and techniques in digital image processing that are considerably easier to describe in the frequency domain or cannot otherwise be understood at all. The topic should therefore not be avoided all together. Fourier analysis not only owns a very elegant (perhaps not always sufficiently appreciated) mathematical theory but interestingly enough also complements some important concepts we have seen earlier, in particular linear filters and linear *convolution* (Sec. 6.2). Equally important are applications of spectral techniques in many popular methods for image and video compression, and they provide valuable insight into the mechanisms of sampling (discretization) of continuous signals as well as the reconstruction and interpolation of discrete signals.

In the following, we first give a basic introduction to the concepts of frequency and spectral decomposition that tries to be minimally formal and thus should be easily "digestible" even for readers without previous exposure to this topic. We start with the representation of one-dimensional signals and will then extend the discussion to two-

dimensional signals (images) in the next chapter. Subsequently, Ch. 15 briefly explains the *discrete cosine transform*, a popular variant of the discrete Fourier transform that is frequently used in image compression.

13.1 The Fourier Transform

The concept of frequency and the decomposition of waveforms into elementary "harmonic" functions first arose in the context of music and sound. The idea of describing acoustic events in terms of "pure" sinusoidal functions does not seem unreasonable, considering that sine waves appear naturally in every form of oscillation (e. g., on a free-swinging pendulum).

13.1.1 Sine and Cosine Functions

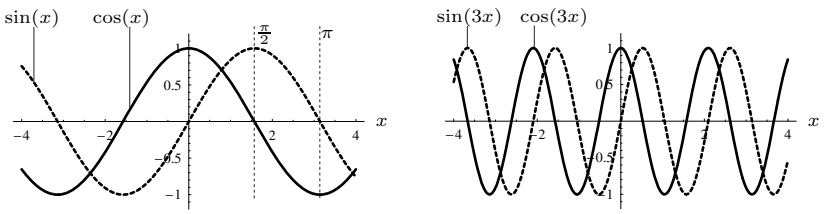
The well-known *cosine* function

$$f(x) = \cos(x) \tag{13.1}$$

has the value 1 at the origin ($\cos(0) = 1$) and performs exactly one full cycle between the origin and the point $x = 2\pi$ (Fig. 13.1 (a)). We say that the function is periodic with a cycle length (period) $T = 2\pi$; i. e.,

$$\cos(x) = \cos(x + 2\pi) = \cos(x + 4\pi) = \cdots = \cos(x + k2\pi) \tag{13.2}$$

for any $k \in \mathbb{Z}$. The same is true for the corresponding *sine* function, except that its value is zero at the origin ($\sin(0) = 0$).



(a) (b)

Fig. 13.1 Cosine and sine functions. The expression $\cos(\omega x)$ describes a cosine function with angular frequency ω at position x. The angular frequency ω of this periodic function corresponds to a cycle length (period) $T = 2\pi/\omega$. For $\omega = 1$, the period is $T_1 = 2\pi$ (a), and for $\omega = 3$ it is $T_3 = 2\pi/3 \approx 2.0944$ (b). The same holds for the sine function $\sin(\omega x)$.

Frequency and amplitude

The number of oscillations of $\cos(x)$ over the distance $T = 2\pi$ is *one* and thus the value of the *angular frequency*

$$\omega = \frac{2\pi}{T} = 1. \tag{13.3}$$

If we modify the function to

$$f(x) = \cos(3x), \tag{13.4}$$

we obtain a compressed cosine wave that oscillates three times faster than the original function $\cos(x)$ (Fig. 13.1 (b)). The function $\cos(3x)$ performs three full cycles over a distance of 2π and thus has the angular frequency $\omega = 3$ and a period $T = \frac{2\pi}{3}$. In general, the period T relates to the angular frequency ω as

$$T = \tfrac{2\pi}{\omega} \tag{13.5}$$

for $\omega > 0$. A sine or cosine function oscillates between peak values $+1$ and -1, and its *amplitude* is 1. Multiplying by a constant $a \in \mathbb{R}$ changes the peak values of the function to $\pm a$ and its *amplitude* to a. In general, the expression

$$a \cdot \cos(\omega x) \qquad \text{and} \qquad a \cdot \sin(\omega x)$$

denotes a cosine or sine function with amplitude a and angular frequency ω, evaluated at position (or point in time) x. The relation between the angular frequency ω and the "common" frequency f is given by

$$f = \frac{1}{T} = \frac{\omega}{2\pi} \quad \text{or} \quad \omega = 2\pi f, \tag{13.6}$$

where f is measured in cycles per length or time unit.[1] In the following, we use either ω or f as appropriate, and the meaning should always be clear from the symbol used.

Phase

Shifting a cosine function along the x axis by a distance φ,

$$\cos(x) \to \cos(x - \varphi),$$

changes the *phase* of the cosine wave, and φ denotes the *phase angle* of the resulting function. Thus a sine function is really just a cosine function shifted to the right[2] by a quarter period ($\varphi = \frac{2\pi}{4} = \frac{\pi}{2}$), so

$$\sin(\omega x) = \cos\left(\omega x - \tfrac{\pi}{2}\right). \tag{13.7}$$

If we take the cosine function as the reference with phase $\varphi_{\cos} = 0$, then the phase angle of the corresponding sine function is $\varphi_{\sin} = \frac{\pi}{2} = 90°$.

Cosine and sine functions are "orthogonal" in a sense and we can use this fact to create new "sinusoidal" functions with arbitrary frequency,

[1] For example, a temporal oscillation with frequency $f = 1000$ cycles/s (Hertz) has the period $T = 1/1000$ s and therefore the angular frequency $\omega = 2000\pi$. The latter is a unitless magnitude.

[2] In general, the function $f(x-d)$ is the original function $f(x)$ shifted to the right by a distance d.

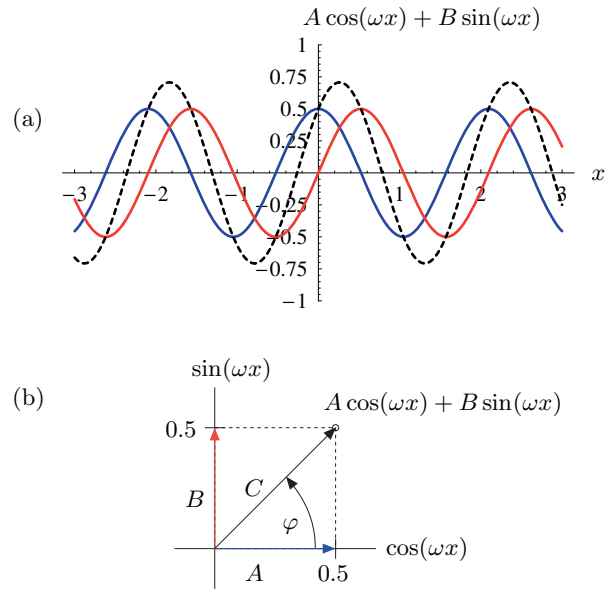


Fig. 13.2
Adding cosine and sine functions with identical frequencies, $A \cdot \cos(\omega x) + B \cdot \sin(\omega x)$, with $\omega = 3$ and $A = B = 0.5$. The result is a phase-shifted cosine function (dotted curve) with amplitude $C = \sqrt{0.5^2 + 0.5^2} \approx 0.707$ and phase angle $\varphi = 45°$ (a). If the cosine and sine components are treated as orthogonal vectors (A, B) in 2-space, the amplitude and phase of the resulting sinusoid (C) can be easily determined by vector summation (b).

phase, and amplitude. In particular, adding a cosine and a sine function with the identical frequencies ω and arbitrary amplitudes A and B, respectively, creates another sinusoid:

$$A \cdot \cos(\omega x) + B \cdot \sin(\omega x) = C \cdot \cos(\omega x - \varphi). \tag{13.8}$$

The resulting amplitude C and the phase angle φ are defined only by the two original amplitudes A and B as

$$C = \sqrt{A^2 + B^2} \quad \text{and} \quad \varphi = \tan^{-1}\left(\tfrac{B}{A}\right). \tag{13.9}$$

Figure 13.2 (a) shows an example with amplitudes $A = B = 0.5$ and a resulting phase angle $\varphi = 45°$.

Complex-valued sine functions—Euler's notation

Figure 13.2 (b) depicts the contributing cosine and sine components of the new function as a pair of orthogonal vectors in 2-space whose *lengths* correspond to the amplitudes A and B. Not coincidentally, this reminds us of the representation of real and imaginary components of complex numbers

$$z = a + \mathrm{i}\, b$$

in the two-dimensional plane $\mathbb{C}$, where i is the imaginary unit ($\mathrm{i}^2 = -1$). This association becomes even stronger if we look at Euler's famous notation of complex numbers along the unit circle,

$$z = e^{\mathrm{i}\theta} = \cos(\theta) + \mathrm{i} \cdot \sin(\theta), \tag{13.10}$$

where $e \approx 2.71828$ is the Euler number. If we take the expression $e^{\mathrm{i}\theta}$ as a function of the angle θ rotating around the unit circle, we obtain a "complex-valued sinusoid" whose real and imaginary parts correspond to a cosine and a sine function, respectively,

$$\begin{aligned} \mathrm{Re}\{e^{\mathrm{i}\theta}\} &= \cos(\theta), \\ \mathrm{Im}\{e^{\mathrm{i}\theta}\} &= \sin(\theta). \end{aligned} \tag{13.11}$$

Since $z = e^{\mathrm{i}\theta}$ is placed on the unit circle, the *amplitude* of the complex-valued sinusoid is $|z| = r = 1$. We can easily modify the amplitude of this function by multiplying it by some real value $a \geq 0$,

$$|a \cdot e^{\mathrm{i}\theta}| = a \cdot |e^{\mathrm{i}\theta}| = a. \tag{13.12}$$

Similarly, we can alter the *phase* of a complex-valued sinusoid by adding a phase angle φ in the function's exponent or, equivalently, by multiplying it by a complex-valued constant $c = e^{\mathrm{i}\varphi}$,

$$e^{\mathrm{i}(\theta+\varphi)} = e^{\mathrm{i}\theta} \cdot e^{\mathrm{i}\varphi}. \tag{13.13}$$

In summary, multiplying by some real value affects only the *amplitude* of a sinusoid, while multiplying by some complex value c (with unit amplitude $|c| = 1$) modifies only the function's *phase* (without changing its amplitude). In general, of course, multiplying by some arbitrary complex value changes both the amplitude *and* the phase of the function (also see Appendix 1.3).

The complex notation makes it easy to combine orthogonal pairs of sine functions $\cos(\omega x)$ and $\sin(\omega x)$ with identical frequencies ω into a single functional expression

$$e^{\mathrm{i}\theta} = e^{\mathrm{i}\omega x} = \cos(\omega x) + \mathrm{i} \cdot \sin(\omega x). \tag{13.14}$$

We will make more use of this notation later in Sec. 13.1.4 to explain the Fourier transform.

13.1.2 Fourier Series of Periodic Functions

As we demonstrated in Eqn. (13.8), sinusoidal functions of arbitrary frequency, amplitude, and phase can be described as the sum of suitably weighted cosine and sine functions. One may wonder if non-sinusoidal functions can also be decomposed into a sum of cosine and sine functions. The answer is yes, of course. It was Fourier[3] who first extended this idea to arbitrary functions and showed that (almost) any periodic function $g(x)$ with a fundamental frequency ω_0 can be described as a—possibly infinite—sum of "harmonic" sinusoids; i. e.,

[3] Jean Baptiste Joseph de Fourier (1768–1830).

$$g(x) = \sum_{k=0}^{\infty} \left[A_k \cos(k\omega_0 x) + B_k \sin(k\omega_0 x) \right]. \tag{13.15}$$

This is called a *Fourier series*, and the constant factors A_k, B_k are the *Fourier coefficients* of the function $g(x)$. Notice that in Eqn. (13.15) the frequencies of the sine and cosine functions contributing to the Fourier series are integral multiples ("harmonics") of the fundamental frequency ω_0, including the zero frequency for $k = 0$. The corresponding coefficients A_k and B_k, which are initially unknown, can be uniquely derived from the original function $g(x)$. This process is commonly referred to as *Fourier analysis*.

13.1.3 Fourier Integral

Fourier did not want to limit this concept to periodic functions and postulated that nonperiodic functions, too, could be described as sums of sine and cosine functions. While this proved to be true in principle, it generally requires—beyond multiples of the fundamental frequency $(k\omega_0)$—infinitely many, densely spaced frequencies! The resulting decomposition

$$g(x) = \int_0^{\infty} A_\omega \cos(\omega x) + B_\omega \sin(\omega x)\, \mathrm{d}\omega \tag{13.16}$$

is called a *Fourier integral*, and the coefficients A_ω, B_ω are again the weights for the corresponding cosine and sine functions with the (continuous) frequency ω. The Fourier integral is the basis of the Fourier spectrum and the Fourier transform, as described below (for details, see e. g., [15, Sec. 15.3]).

In Eqn. (13.16), every coefficient A_ω and B_ω specifies the *amplitude* of the corresponding cosine or sine function, respectively. The coefficients thus define "how much of each frequency" contributes to a given function or signal $g(x)$. But what are the proper values of these coefficients for a given function $g(x)$, and can they be determined uniquely? The answer is yes again, and the "recipe" for computing the coefficients is amazingly simple:

$$A_\omega = A(\omega) = \frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \cos(\omega x)\, \mathrm{d}x, \tag{13.17}$$

$$B_\omega = B(\omega) = \frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \sin(\omega x)\, \mathrm{d}x. \tag{13.18}$$

Since this representation of the function $g(x)$ involves infinitely many densely spaced frequency values ω, the corresponding coefficients $A(\omega)$ and $B(\omega)$ are indeed continuous functions as well. They hold the continuous distribution of frequency components contained in the original signal, which is called a "spectrum".

Thus the Fourier integral in Eqn. (13.16) describes the original function $g(x)$ as a sum of infinitely many cosine and sine functions, with the corresponding Fourier coefficients contained in the functions $A(\omega)$ and $B(\omega)$. In addition, a signal $g(x)$ is uniquely and fully represented by the corresponding coefficient functions $A(\omega)$ and $B(\omega)$. We know from Eqn. (13.17) how to compute the spectrum for a given function $g(x)$, and Eqn. (13.16) explains how to reconstruct the original function from its spectrum if it is ever needed.

13.1.4 Fourier Spectrum and Transformation

There is now only a small remaining step from the decomposition of a function $g(x)$, as shown in Eqn. (13.17), to the "real" Fourier transform. In contrast to the Fourier *integral*, the Fourier *transform* treats both the original signal and the corresponding spectrum as *complex-valued* functions, which considerably simplifies the resulting notation. Based on the functions $A(\omega)$ and $B(\omega)$ defined in the Fourier integral (Eqn. (13.17)), the *Fourier spectrum* $G(\omega)$ of a function $g(x)$ is given as

$$\begin{aligned} G(\omega) &= \sqrt{\tfrac{\pi}{2}}\Big[A(\omega) - \mathrm{i}\cdot B(\omega)\Big] \\ &= \sqrt{\tfrac{\pi}{2}}\left[\frac{1}{\pi}\int_{-\infty}^{\infty} g(x)\cdot\cos(\omega x)\,\mathrm{d}x \;-\; \mathrm{i}\cdot\frac{1}{\pi}\int_{-\infty}^{\infty} g(x)\cdot\sin(\omega x)\,\mathrm{d}x\right] \\ &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} g(x)\cdot\Big[\cos(\omega x) - \mathrm{i}\cdot\sin(\omega x)\Big]\,\mathrm{d}x, \end{aligned} \tag{13.19}$$

with $g(x), G(\omega) \in \mathbb{C}$. Using Euler's notation of complex values (Eqn. (13.14)) yields the continuous Fourier spectrum from Eqn. (13.19) in its most popular form:

$$\boxed{\begin{aligned} G(\omega) &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} g(x)\cdot\Big[\cos(\omega x) - \mathrm{i}\cdot\sin(\omega x)\Big]\,\mathrm{d}x \\ &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} g(x)\cdot e^{-\mathrm{i}\omega x}\,\mathrm{d}x. \end{aligned}} \tag{13.20}$$

The transition from the function $g(x)$ to its Fourier spectrum $G(\omega)$ is called the *Fourier transform*[4] ($\mathcal{F}$). Conversely, the original function $g(x)$ can be reconstructed completely from its Fourier spectrum $G(\omega)$ using the *inverse Fourier transform*[5] ($\mathcal{F}^{-1}$), defined as

$$\boxed{\begin{aligned} g(x) &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} G(\omega)\cdot\Big[\cos(\omega x) + \mathrm{i}\cdot\sin(\omega x)\Big]\,\mathrm{d}\omega \\ &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} G(\omega)\cdot e^{\mathrm{i}\omega x}\,\mathrm{d}\omega. \end{aligned}} \tag{13.21}$$

[4] Also called the "direct" or "forward" transformation.
[5] Also called "backward" transformation.

In general, even if one of the involved functions ($g(x)$ or $G(\omega)$) is real-valued (which is usually the case for physical signals $g(x)$), the other function is complex-valued. One may also note that the forward transformation $\mathcal{F}$ (Eqn. (13.20)) and the inverse transformation $\mathcal{F}^{-1}$ (Eqn. (13.21)) are almost completely symmetrical, the sign of the exponent being the only difference.[6] The spectrum produced by the Fourier transform is a new representation of the signal in a space of frequencies. Apparently, this "frequency space" and the original "signal space" are *dual* and interchangeable mathematical representations.

13.1.5 Fourier Transform Pairs

The relationship between a function $g(x)$ and its Fourier spectrum $G(\omega)$ is unique in both directions: the Fourier spectrum is uniquely defined for a given function, and for any Fourier spectrum there is only one matching signal—the two functions $g(x)$ and $G(\omega)$ constitute a "transform pair",

$$g(x) \circ\!\!-\!\!\bullet\, G(\omega).$$

Table 13.1 lists the transform pairs for some selected analytical functions, which are also shown graphically in Figs. 13.3 and 13.4.

Table 13.1
Fourier transforms of selected analytical functions; $\delta()$ denotes the "impulse" or *Dirac* function (see Sec. 13.2.1).

Function	*Transform Pair* $g(x) \circ\!\!-\!\!\bullet\, G(\omega)$	*Figure*
Cosine function with frequency ω_0	$g(x) = \cos(\omega_0 x)$ $G(\omega) = \sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega-\omega_0) + \delta(\omega+\omega_0)\big)$	13.3 (a, c)
Sine function with frequency ω_0	$g(x) = \sin(\omega_0 x)$ $G(\omega) = \mathrm{i}\sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega-\omega_0) - \delta(\omega+\omega_0)\big)$	13.3 (b, d)
Gaussian function of width σ	$g(x) = \frac{1}{\sigma} \cdot e^{-\frac{x^2}{2\sigma^2}}$ $G(\omega) = e^{-\frac{\sigma^2\omega^2}{2}}$	13.4 (a, b)
Rectangular pulse of width $2b$	$g(x) = \Pi_b(x) = \begin{cases} 1 & \text{for } \lvert x\rvert \le b \\ 0 & \text{otherwise} \end{cases}$ $G(\omega) = \frac{2b\sin(b\omega)}{\sqrt{2\pi}\omega}$	13.4 (c, d)

The Fourier spectrum of a *cosine function* $\cos(\omega_0 x)$, for example, consists of two separate thin pulses arranged symmetrically at a distance ω_0 from the origin (Fig. 13.3 (a, c)). Intuitively, this corresponds to our physical understanding of a spectrum (e. g., if we think of a pure

[6] Various definitions of the Fourier transform are in common use. They are contrasted mainly by the constant factors outside the integral and the signs of the exponents in the forward and inverse transforms, but all versions are equivalent in principle. The symmetric variant shown here uses the same factor $(1/\sqrt{2\pi})$ in the forward and inverse transforms.

monophonic sound in acoustics or the thin line produced by some extremely pure color in the optical spectrum). Increasing the frequency ω_0 would move the corresponding pulses in the spectrum away from the origin. Notice that the spectrum of the cosine function is real-valued, the imaginary part being zero. Of course, the same relation holds for the sine function (Fig. 13.3 (b, d)), with the only difference being that the pulses have different polarities and appear in the imaginary part of the spectrum. In this case, the real part of the spectrum $G(\omega)$ is zero.

The *Gaussian function* is particularly interesting because its Fourier spectrum is also a Gaussian function (Fig. 13.4 (a, b))! It is one of the few examples where the function type in frequency space is the same as in signal space. With the Gaussian function, it is also clear to see that *stretching* a function in signal space corresponds to *shortening* its spectrum and vice versa.

The Fourier transform of a *rectangular pulse* (Fig. 13.4 (c, d)) is the "Sinc" function of type $\sin(x)/x$. With increasing frequencies, this function drops off quite slowly, which shows that the components contained in the original rectangular signal are spread out over a large frequency range. Thus a rectangular pulse function exhibits a very wide spectrum in general.

13.1.6 Important Properties of the Fourier Transform

Symmetry

The Fourier spectrum extends over positive and negative frequencies and could, in principle, be an arbitrary complex-valued function. However, in many situations, the spectrum is symmetric about its origin (see, e. g., [20, p. 178]). In particular, the Fourier transform of a real-valued signal $g(x) \in \mathbb{R}$ is a so-called *Hermite* function with the property

$$G(\omega) = G^*(-\omega), \tag{13.22}$$

where G^* denotes the complex conjugate of G (see also Appendix 1.3).

Linearity

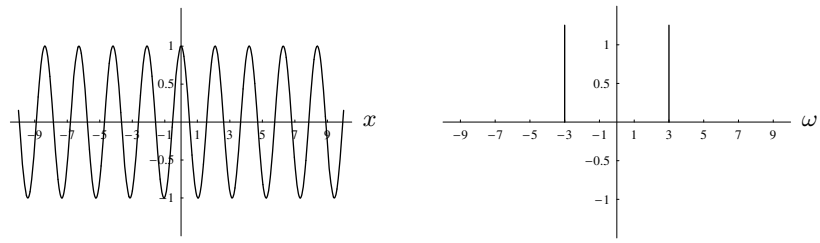
The Fourier transform is also a *linear* operation such that multiplying the signal by a constant value $c \in \mathbb{C}$ scales the corresponding spectrum by the same amount,

$$c \cdot g(x) \circ\!\!-\!\!\bullet\, c \cdot G(\omega). \tag{13.23}$$

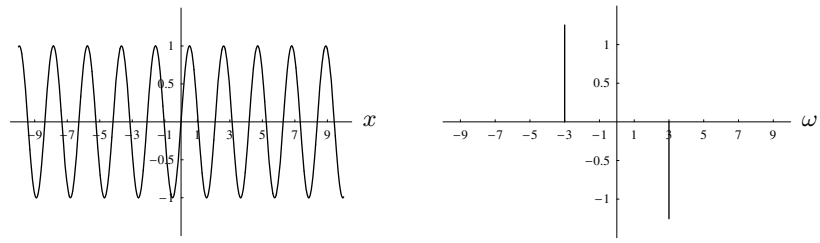
Linearity also means that the transform of the sum of two signals $g(x) = g_1(x) + g_2(x)$ is identical to the sum of their individual transforms $G_1(\omega)$ and $G_2(\omega)$ and thus

$$g_1(x) + g_2(x) \circ\!\!-\!\!\bullet\, G_1(\omega) + G_2(\omega). \tag{13.24}$$

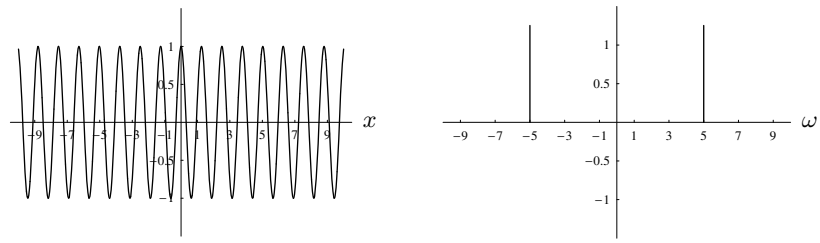
Fig. 13.3
Fourier transform pairs—cosine and sine functions.



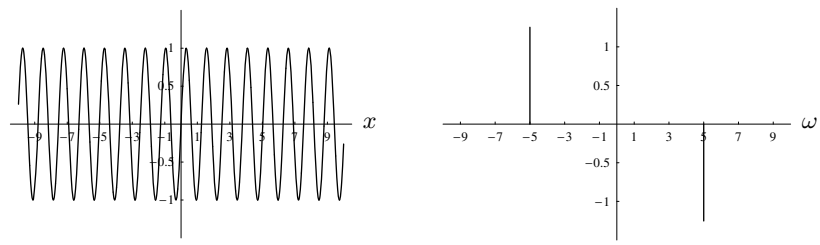
(a) cosine ($\omega_0 = 3$): $g(x) = \cos(3x)$ $\quad\multimap\bullet\quad$ $G(\omega) = \sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega-3) + \delta(\omega+3)\big)$



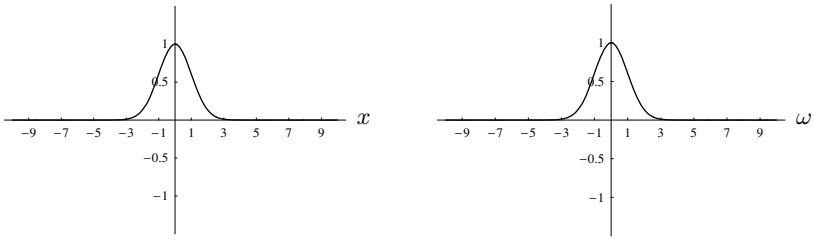
(b) sine ($\omega_0 = 3$): $g(x) = \sin(3x)$ $\quad\multimap\bullet\quad$ $G(\omega) = \mathrm{i}\sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega-3) - \delta(\omega+3)\big)$



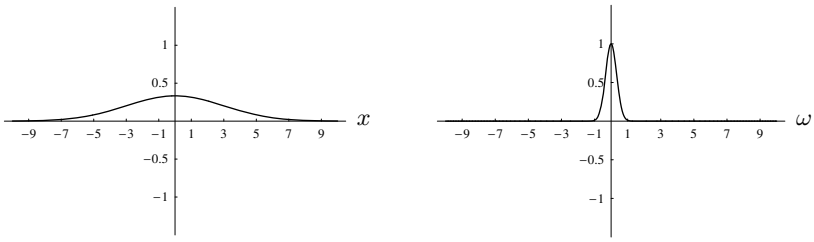
(c) cosine ($\omega_0 = 5$): $g(x) = \cos(5x)$ $\quad\multimap\bullet\quad$ $G(\omega) = \sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega-5) + \delta(\omega+5)\big)$



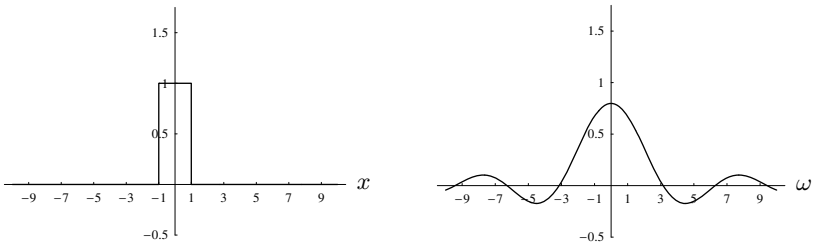
(d) sine ($\omega_0 = 5$): $g(x) = \sin(5x)$ $\quad\multimap\bullet\quad$ $G(\omega) = \mathrm{i}\sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega-5) - \delta(\omega+5)\big)$



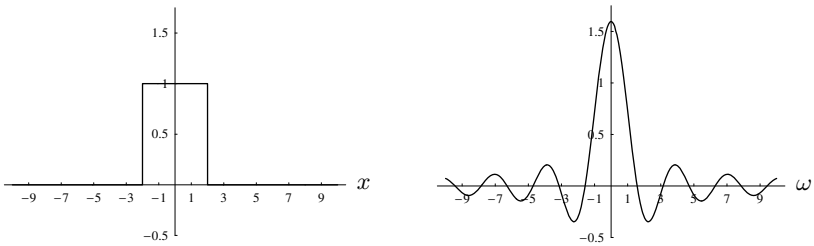
(a) Gauss ($\sigma=1$): $g(x) = e^{-\frac{x^2}{2}}$ $\circ\!\!-\!\!\bullet$ $G(\omega) = e^{-\frac{\omega^2}{2}}$



(b) Gauss ($\sigma=3$): $g(x) = \frac{1}{3} \cdot e^{-\frac{x^2}{2 \cdot 9}}$ $\circ\!\!-\!\!\bullet$ $G(\omega) = e^{-\frac{9\omega^2}{2}}$



(c) rect. pulse ($b=1$): $g(x) = \Pi_1(x)$ $\circ\!\!-\!\!\bullet$ $G(\omega) = \frac{2\sin(\omega)}{\sqrt{2\pi}\omega}$



(d) rect. pulse ($b=2$): $g(x) = \Pi_2(x)$ $\circ\!\!-\!\!\bullet$ $G(\omega) = \frac{4\sin(2\omega)}{\sqrt{2\pi}\omega}$

Fig. 13.4
Fourier transform pairs—Gaussian functions and rectangular pulses.

Similarity

If the original function $g(x)$ is scaled in space or time, the opposite effect appears in the corresponding Fourier spectrum. In particular, as observed on the Gaussian function in Fig. 13.4, *stretching* a signal by a factor s (i. e., $g(x) \rightarrow g(sx)$) leads to a *shortening* of the Fourier spectrum:

$$g(sx) \circ\!\!-\!\!\bullet \frac{1}{|s|} \cdot G\left(\frac{\omega}{s}\right). \tag{13.25}$$

Similarly, the signal is shortened if the corresponding spectrum is stretched.

Shift property

If the original function $g(x)$ is shifted by a distance d along its coordinate axis (i. e., $g(x) \rightarrow g(x-d)$), then the Fourier spectrum multiplies by the complex value $e^{-\mathrm{i}\omega d}$ dependent on ω:

$$g(x-d) \circ\!\!-\!\!\bullet e^{-\mathrm{i}\omega d} \cdot G(\omega). \tag{13.26}$$

Since $e^{-\mathrm{i}\omega d}$ lies on the unit circle, the multiplication causes a phase shift on the spectral values (i. e., a redistribution between the real and imaginary components) without altering the magnitude $|G(\omega)|$. Obviously, the amount (angle) of phase shift (ωd) is proportional to the angular frequency ω.

Convolution property

From the image-processing point of view, the most interesting property of the Fourier transform is its relation to linear convolution, which we described in Sec. 6.3.1. Let us assume that we have two functions $g(x)$ and $h(x)$ and their corresponding Fourier spectra $G(\omega)$ and $H(\omega)$, respectively. If the original functions are subject to linear convolution (i. e., $g(x) * h(x)$), then the Fourier transform of the result equals the (pointwise) product of the individual Fourier transforms $G(\omega)$ and $H(\omega)$:

$$g(x) * h(x) \circ\!\!-\!\!\bullet G(\omega) \cdot H(\omega). \tag{13.27}$$

Due to the duality of signal space and frequency space, the same also holds in the opposite direction; i. e., a pointwise multiplication of two signals is equivalent to convolving the corresponding spectra:

$$g(x) \cdot h(x) \circ\!\!-\!\!\bullet G(\omega) * H(\omega). \tag{13.28}$$

A multiplication of the functions in *one* space (signal or frequency space) thus corresponds to a linear convolution of the Fourier spectra in the *opposite* space.

13.2 Working with Discrete Signals

The definition of the continuous Fourier transform above is of little use for numerical computation on a computer. Neither can arbitrary continuous (and possibly infinite) functions be represented in practice. Nor can the required integrals be computed. In reality, we must always deal with *discrete* signals, and we therefore need a new version of the Fourier transform that treats signals and spectra as finite data vectors—the "discrete" Fourier transform. Before continuing with this issue we want to use our existing wisdom to take a closer look at the process of discretizing signals in general.

13.2.1 Sampling

We first consider the question of how a continuous function can be converted to a discrete signal in the first place. This process is usually called "sampling" (i. e., taking samples of the continuous function at certain points in time (or in space), usually spaced at regular distances). To describe this step in a simple but formal way, we require an inconspicuous but nevertheless important piece from the mathematician's toolbox.

The impulse function $\delta(x)$

We casually encountered the impulse function (also called the *delta* or *Dirac* function) earlier when we looked at the impulse response of linear filters (Sec. 6.3.4) and in the Fourier transforms of the cosine and sine functions (Fig. 13.3). This function, which models a continuous "ideal" impulse, is unusual in several respects: its value is zero everywhere except at the origin, where it is nonzero (though undefined), but its integral is one; i. e.,

$$\delta(x) = 0 \;\text{ for }\; x \neq 0 \quad \text{and} \quad \int_{-\infty}^{\infty} \delta(x)\,\mathrm{d}x = 1. \tag{13.29}$$

One could imagine $\delta(x)$ as a single pulse at position $x = 0$ that is infinitesimally narrow but still contains finite energy (1). Also remarkable is the impulse function's behavior under scaling along the time (or space) axis (i. e., $\delta(x) \to \delta(sx)$), with

$$\delta(sx) = \frac{1}{|s|} \cdot \delta(x) \quad \text{for } s \neq 0. \tag{13.30}$$

Despite the fact that $\delta(x)$ does not exist in physical reality and cannot be plotted (the corresponding plots in Fig. 13.3 are for illustration only), this function is a useful mathematical tool for describing the sampling process, as shown below.

Sampling with the impulse function

Using the concept of the ideal impulse, the sampling process can be described in a straightforward and intuitive way.[7] If a continuous function $g(x)$ is multiplied with the impulse function $\delta(x)$, we obtain a new function

$$\bar{g}(x) = g(x) \cdot \delta(x) = \begin{cases} g(0) & \text{for } x = 0 \\ 0 & \text{otherwise.} \end{cases} \tag{13.31}$$

The resulting function $\bar{g}(x)$ consists of a single pulse at position 0 whose height corresponds to the original function value $g(0)$ (at position 0). Thus, by multiplying the function $g(x)$ by the impulse function, we obtain a single discrete sample value of $g(x)$ at position $x = 0$. If the impulse function $\delta(x)$ is shifted by a distance x_0, we can sample $g(x)$ at an arbitrary position $x = x_0$,

$$\bar{g}(x) = g(x) \cdot \delta(x - x_0) = \begin{cases} g(x_0) & \text{for } x = x_0 \\ 0 & \text{otherwise.} \end{cases} \tag{13.32}$$

Here $\delta(x - x_0)$ is the impulse function shifted by x_0, and the resulting function $\bar{g}(x)$ is zero except at position x_0, where it contains the original function value $g(x_0)$. This relationship is illustrated in Fig. 13.5 for the sampling position $x_0 = 3$.

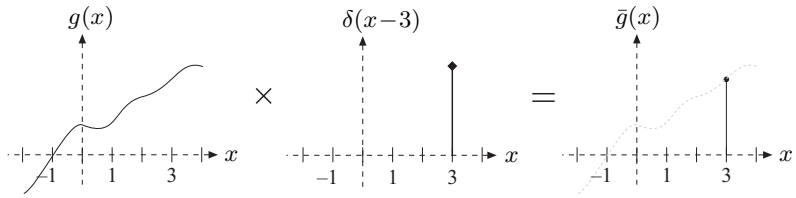


Fig. 13.5
Sampling with the impulse function. The continuous signal $g(x)$ is sampled at position $x_0 = 3$ by multiplying $g(x)$ by a shifted impulse function $\delta(x-3)$.

To sample the function $g(x)$ at more than one position simultaneously (e. g., at positions x_1 and x_2), we use two separately shifted versions of the impulse function, multiply $g(x)$ by both of them, and simply add the resulting function values. In this particular case, we get

$$\bar{g}(x) = g(x) \cdot \delta(x - x_1) + g(x) \cdot \delta(x - x_2) \tag{13.33}$$

$$= g(x) \cdot \big[\delta(x - x_1) + \delta(x - x_2)\big] \tag{13.34}$$

$$= \begin{cases} g(x_1) & \text{for } x = x_1 \\ g(x_2) & \text{for } x = x_2 \\ 0 & \text{otherwise.} \end{cases} \tag{13.35}$$

From Eqn. (13.34), sampling a continuous function $g(x)$ at N positions $x_i = 1, 2, \ldots N$ can thus be described as the sum of the N individual samples,

[7] The following description is intentionally casual and superficial in a mathematical sense. See, e. g., [20, 60] for more precise coverage of these topics.

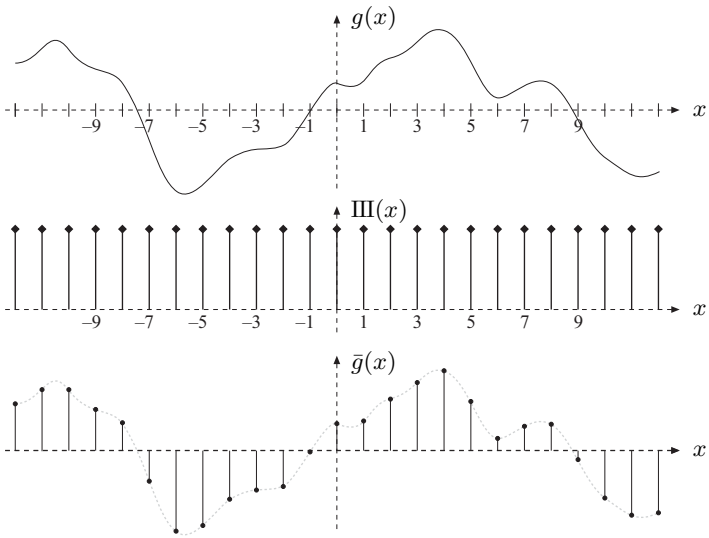


Fig. 13.6
Sampling with the comb function. The original continuous signal $g(x)$ is multiplied by the comb function $\text{Ш}(x)$. The function value $g(x)$ is transferred to the resulting function $\bar{g}(x)$ only at integral positions $x = x_i \in \mathbb{Z}$ and ignored at all nonintegral positions.

$$\begin{aligned}\bar{g}(x) &= g(x) \cdot \big[\delta(x-1) + \delta(x-2) + \ldots + \delta(x-N)\big] \\ &= g(x) \cdot \sum_{i=1}^{N} \delta(x-i). \end{aligned} \tag{13.36}$$

The comb function

The sum of shifted impulses $\sum_{i=1}^{N} \delta(x-i)$ in Eqn. (13.36) is called a *pulse sequence* or *pulse train*. Extending this sequence to infinity in both directions, we obtain the "comb" or "Shah" function

$$\text{Ш}(x) = \sum_{i=-\infty}^{\infty} \delta(x-i). \tag{13.37}$$

The process of discretizing a continuous function by taking samples at regular integral intervals can thus be written simply as

$$\bar{g}(x) = g(x) \cdot \text{Ш}(x), \tag{13.38}$$

i. e., as a pointwise multiplication of the original signal $g(x)$ with the comb function $\text{Ш}(x)$. As Fig. 13.6 illustrates, the function values of $g(x)$ at integral positions $x_i \in \mathbb{Z}$ are transferred to the discrete function $\bar{g}(x_i)$ and ignored at all nonintegral positions.

Of course, the sampling interval (i. e., the distance between adjacent samples) is not restricted to 1. To take samples at regular but *arbitrary* intervals τ, the sampling function $\text{Ш}(x)$ is simply scaled along the time or space axis; i. e.,

$$\bar{g}(x) = g(x) \cdot \text{Ш}\left(\tfrac{x}{\tau}\right) \quad \text{for } \tau > 0. \tag{13.39}$$

Effects of sampling in frequency space

Despite the elegant formulation made possible by the use of the comb function, one may still wonder why all this math is necessary to describe a process that appears intuitively to be so simple anyway. The Fourier spectrum gives one answer to this question. Sampling a continuous function has massive—though predictable—effects upon the frequency spectrum of the resulting (discrete) signal. Using the comb function as a formal model for the sampling process makes it relatively easy to estimate and interpret those spectral effects. Similar to the Gaussian (see Sec. 13.1.5), the comb function features the rare property that its Fourier transform

$$\text{Ш}(x) \circ\!\!-\!\!\bullet\, \text{Ш}(\tfrac{1}{2\pi}\omega) \tag{13.40}$$

is again a comb function (i. e., the same type of function). In general, the Fourier transform of a comb function scaled to an arbitrary sampling interval τ is

$$\text{Ш}(\tfrac{x}{\tau}) \circ\!\!-\!\!\bullet\, \tau\text{Ш}\left(\tfrac{\tau}{2\pi}\omega\right) \tag{13.41}$$

due to the similarity property of the Fourier transform (Eqn. (13.25)). Figure 13.7 shows two examples of the comb function $\text{Ш}_\tau(x)$ with sampling intervals $\tau = 1$ and $\tau = 3$ and the corresponding Fourier transforms.

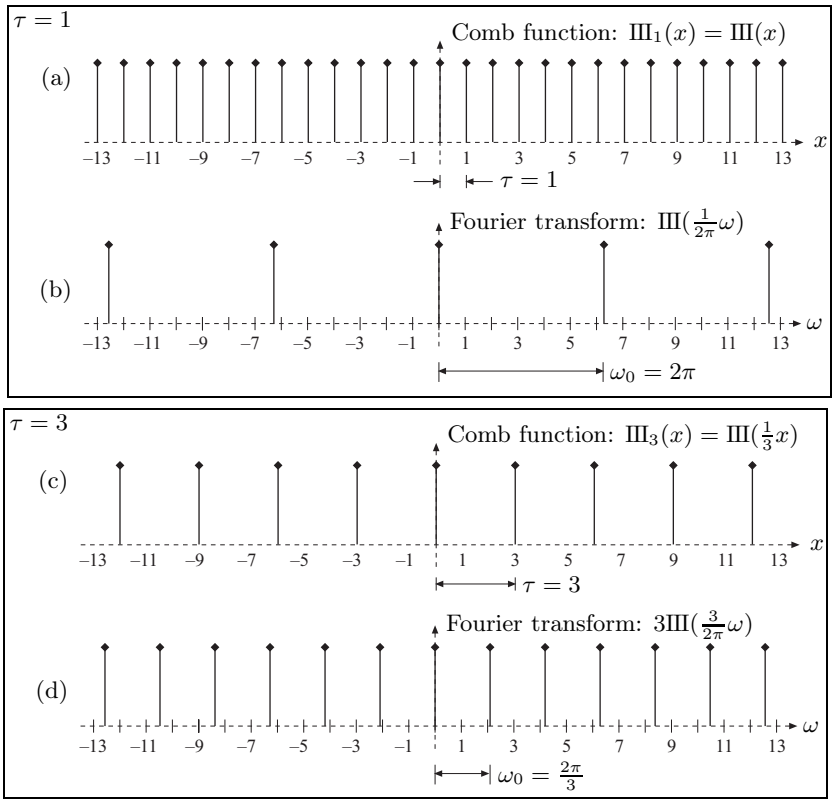
Now, what happens to the Fourier spectrum during discretization; i. e., when we multiply a function in signal space by the comb function $\text{Ш}(\frac{x}{\tau})$? We get the answer by recalling the convolution property of the Fourier transform (Eqn. (13.27)): the product of two functions in one space (signal or frequency space) corresponds to the linear convolution of the transformed functions in the opposite space, and thus

$$g(x) \cdot \text{Ш}(\tfrac{x}{\tau}) \circ\!\!-\!\!\bullet\, G(\omega) * \tau\text{Ш}\left(\tfrac{\tau}{2\pi}\omega\right). \tag{13.42}$$

We already know that the Fourier spectrum of the sampling function is a comb function again and therefore consists of a sequence of regularly spaced pulses (Fig. 13.7). In addition, we know that convolving an arbitrary function with the impulse $\delta(x)$ returns the original function; i. e., $f(x) * \delta(x) = f(x)$ (see Sec. 6.3.4). Convolving with a *shifted* pulse $\delta(x-d)$ also reproduces the original function $f(x)$, though shifted by the same distance d; i. e.,

$$f(x) * \delta(x-d) = f(x-d). \tag{13.43}$$

As a consequence, the spectrum $G(\omega)$ of the original continuous signal becomes *replicated* in the Fourier spectrum $\bar{G}(\omega)$ of a sampled signal at every pulse of the sampling function's spectrum; i. e., infinitely many times (see Fig. 13.8 (a, b))! Thus the resulting Fourier spectrum is repetitive with a period $\frac{2\pi}{\tau}$, which corresponds to the sampling frequency ω_s.



13.2 Working with Discrete Signals

Fig. 13.7
Comb function and its Fourier transform. Comb function $\text{Ш}_\tau(x)$ for the sampling interval $\tau = 1$ (a) and its Fourier transform. Comb function for $\tau = 3$ (c) and its Fourier transform (d). Note that the actual height of the δ-pulses is undefined and shown only for illustration.

Aliasing and the sampling theorem

As long as the spectral replicas in $\bar{G}(\omega)$ created by the sampling process do not overlap, the original spectrum $G(\omega)$—and thus the original continuous function—can be reconstructed without loss from any isolated replica of $G(\omega)$ in the periodic spectrum $\bar{G}(\omega)$. As we can see in Fig. 13.8, this requires that the frequencies contained in the original signal $g(x)$ be within some upper limit $\omega_{\max}$; i. e., the signal contains no components with frequencies greater than $\omega_{\max}$. The maximum allowed signal frequency $\omega_{\max}$ depends upon the sampling frequency ω_s used to discretize the signal, with the requirement

$$\omega_{\max} \leq \tfrac{1}{2}\omega_s \quad \text{or} \quad \omega_s \geq 2\omega_{\max}. \tag{13.44}$$

Discretizing a continuous signal $g(x)$ with frequency components in the range $0 \leq \omega \leq \omega_{\max}$ thus requires a sampling frequency ω_s of at least twice the maximum signal frequency $\omega_{\max}$. If this condition is not met, the replicas in the spectrum of the sampled signal overlap (Fig. 13.8 (c)) and the spectrum becomes corrupted. Consequently, the original signal cannot be recovered flawlessly from the sampled signal's spectrum. This effect is commonly called "aliasing".

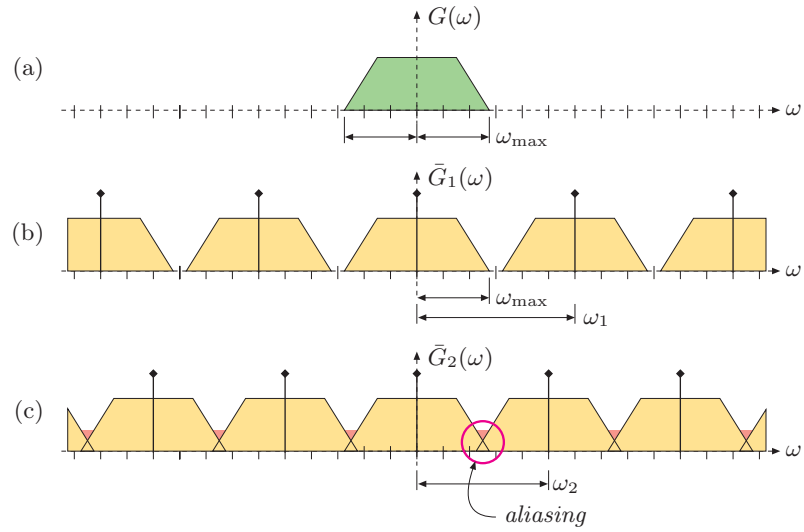


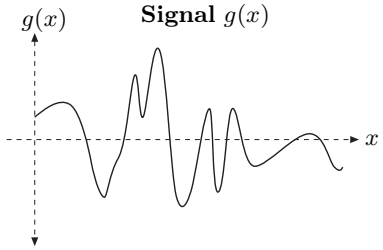
Fig. 13.8
Spectral effects of sampling. The spectrum $G(\omega)$ of the original continuous signal is assumed to be band-limited within the range $\pm\omega_{\max}$ (a). Sampling the signal at a rate (sampling frequency) $\omega_s = \omega_1$ causes the signal's spectrum $G(\omega)$ to be replicated at multiples of ω_1 along the frequency (ω) axis (b). Obviously, the replicas in the spectrum do not overlap as long as $\omega_s > 2\omega_{\max}$. In (c), the sampling frequency $\omega_s = \omega_2$ is less than $2\omega_{\max}$, so there is overlap between the replicas in the spectrum, and frequency components are mirrored at $2\omega_{\max}$ and superimpose the original spectrum. This effect is called "aliasing" because the original spectrum (and thus the original signal) cannot be reproduced from such a corrupted spectrum.

What we just said in simple terms is nothing but the essence of the famous "sampling theorem" formulated by Shannon and Nyquist (see e. g. [20, p. 256]). It actually states that the sampling frequency must be at least twice the *bandwidth*[8] of the continuous signal to avoid aliasing effects. However, if we assume that a signal's frequency range starts at zero, then bandwidth and maximum frequency are the same anyway.

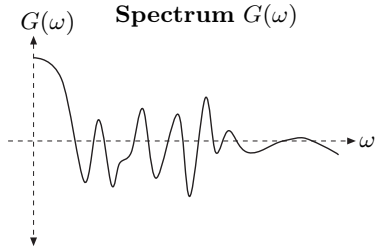
13.2.2 Discrete and Periodic Functions

Assume that we are given a continuous signal $g(x)$ that is periodic with a period of length T. In this case, the corresponding Fourier spectrum $G(\omega)$ is a sequence of thin spectral lines equally spaced at a distance $\omega_0 = 2\pi/T$. As discussed in Sec. 13.1.2, the Fourier spectrum of a periodic function can be represented as a Fourier series and is therefore *discrete*. Conversely, if a continuous signal $g(x)$ is *sampled* at regular intervals τ, then the corresponding Fourier spectrum becomes *periodic* with a period of length $\omega_s = 2\pi/\tau$. Sampling in signal space thus leads to periodicity in frequency space and vice versa. Figure 13.9 illustrates this relationship and the transition from a continuous nonperiodic signal to a discrete periodic function, which can be represented as a finite vector of numbers and thus easily processed on a computer.

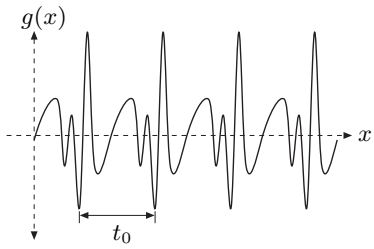
[8] This may be surprising at first because it allows a signal with high frequency—but low bandwidth—to be sampled (and correctly reconstructed) at a relatively low sampling frequency, even well below the maximum signal frequency. This is possible because one can also use a filter with suitably low bandwidth for reconstructing the original signal. For example, it may be sufficient to strike (i. e., "sample") a church bell (a low-bandwidth oscillatory system with small internal damping) to uniquely generate a sound wave of relatively high frequency.



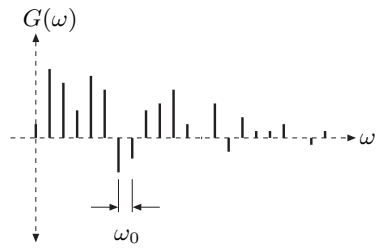
(a) Continuous nonperiodic signal.



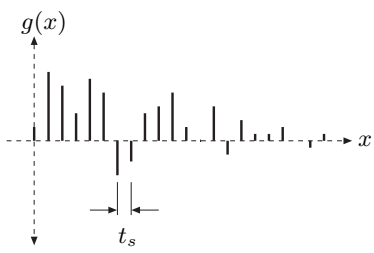
(b) Continuous nonperiodic spectrum.



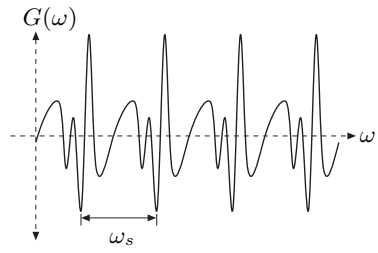
(c) Continuous periodic signal with period t_0.



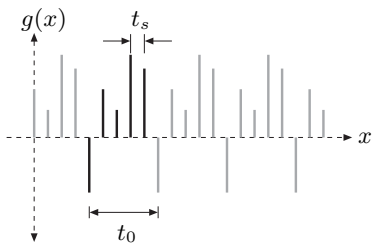
(d) Discrete nonperiodic spectrum with values spaced at $\omega_0 = 2\pi/t_0$.



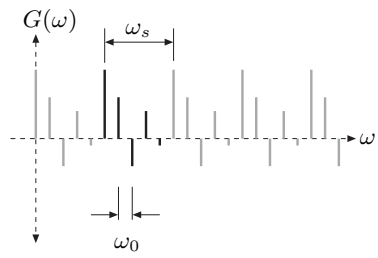
(e) Discrete nonperiodic signal with samples spaced at t_s.



(f) Continuous periodic spectrum with period $\omega_s = 2\pi/t_s$.



(g) Discrete periodic signal with samples spaced at t_s and period $t_0 = t_s M$.



(h) Discrete periodic spectrum with values spaced at $\omega_0 = 2\pi/t_0$ and period $\omega_s = 2\pi/t_s = \omega_0 M$.

Fig. 13.9
Transition from continuous to discrete periodic functions.

Thus, in general, the Fourier spectrum of a continuous, nonperiodic signal $g(x)$ is also continuous and nonperiodic (Fig. 13.9 (a, b)). However, if the signal $g(x)$ is *periodic*, then the corresponding spectrum is *discrete* (Fig. 13.9 (c,d)). Conversely, a discrete—but not necessarily periodic—signal leads to a periodic spectrum (Fig. 13.9 (e, f)). Finally, if a signal is discrete *and* periodic with M samples per period, then its spectrum is also discrete and periodic with M values (Fig. 13.9 (g, h)). Note that the particular signals and spectra in Fig. 13.9 were chosen for illustration only and do not really correspond with each other.

13.3 The Discrete Fourier Transform (DFT)

In the case of a discrete periodic signal, only a finite sequence of M sample values is required to completely represent either the signal $g(u)$ itself or its Fourier spectrum $G(m)$.[9] This representation as finite vectors makes it straightforward to store and process signals and spectra on a computer. What we still need is a version of the Fourier transform applicable to discrete signals.

13.3.1 Definition of the DFT

The discrete Fourier transform is, just like its continuous counterpart, identical in both directions. For a discrete signal $g(u)$ of length M ($u = 0 \ldots M-1$), the forward transform (DFT) is defined as

$$\begin{aligned} G(m) &= \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g(u) \cdot \left[\cos\left(2\pi \frac{mu}{M}\right) - \mathrm{i} \cdot \sin\left(2\pi \frac{mu}{M}\right)\right] \\ &= \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g(u) \cdot e^{-\mathrm{i}2\pi \frac{mu}{M}} \qquad \text{for } 0 \le m < M \end{aligned} \tag{13.45}$$

and the *inverse* transform (DFT^{-1}) as

$$\begin{aligned} g(u) &= \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G(m) \cdot \left[\cos\left(2\pi \frac{mu}{M}\right) + \mathrm{i} \cdot \sin\left(2\pi \frac{mu}{M}\right)\right] \\ &= \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G(m) \cdot e^{\mathrm{i}2\pi \frac{mu}{M}} \qquad \text{for } 0 \le u < M. \end{aligned} \tag{13.46}$$

(Compare these definitions with the corresponding expressions for the *continuous* forward and inverse Fourier transforms in Eqns. (13.20) and (13.21), respectively.) Both the *signal* $g(u)$ and the discrete *spectrum* $G(m)$ are complex-valued vectors of length M,

[9] Notation: we use $g(x)$, $G(\omega)$ for a *continuous* signal or spectrum, respectively, and $g(u)$, $G(m)$ for the *discrete* versions.

u	$g(u)$			$G(m)$		m
0	1.0000	0.0000		14.2302	0.0000	0
1	3.0000	0.0000	DFT	−5.6745	−2.9198	1
2	5.0000	0.0000	⟶	*0.0000	*0.0000	2
3	7.0000	0.0000		−0.0176	−0.6893	3
4	9.0000	0.0000		*0.0000	*0.0000	4
5	8.0000	0.0000		0.3162	0.0000	5
6	6.0000	0.0000		*0.0000	*0.0000	6
7	4.0000	0.0000	DFT^{-1}	−0.0176	0.6893	7
8	2.0000	0.0000	⟵	*0.0000	*0.0000	8
9	0.0000	0.0000		−5.6745	2.9198	9
	Re	Im		Re	Im	

Fig. 13.10
Complex-valued vectors (example). In the discrete Fourier transform (DFT), both the original signal $g(u)$ and its spectrum $G(m)$ are complex-valued vectors of length M ($M = 10$ in this example); $*$ indicates values with $|G(m)| < 10^{-15}$.

$$\begin{aligned} g(u) &= g_{\mathrm{Re}}(u) + \mathrm{i}\cdot g_{\mathrm{Im}}(u), \\ G(m) &= G_{\mathrm{Re}}(m) + \mathrm{i}\cdot G_{\mathrm{Im}}(m), \end{aligned} \tag{13.47}$$

for $u, m = 0 \ldots M-1$ (Fig. 13.10). Expanding the first line of Eqn. (13.45), we obtain the complex values of the Fourier spectrum in component notation as

$$G(m) = \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} \Big[\underbrace{g_{\mathrm{Re}}(u) + \mathrm{i}\cdot g_{\mathrm{Im}}(u)}_{g(u)}\Big] \cdot \Big[\underbrace{\cos\big(2\pi\tfrac{mu}{M}\big)}_{\boldsymbol{C}_m^M(u)} - \mathrm{i}\cdot\underbrace{\sin\big(2\pi\tfrac{mu}{M}\big)}_{\boldsymbol{S}_m^M(u)}\Big], \tag{13.48}$$

where we denote as $\boldsymbol{C}_m^M$ and $\boldsymbol{S}_m^M$ the discrete (cosine and sine) basis functions, as described in the next section. Applying the usual complex multiplication, we obtain the real and imaginary parts of the discrete Fourier spectrum as

$$G_{\mathrm{Re}}(m) = \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g_{\mathrm{Re}}(u) \cdot \boldsymbol{C}_m^M(u) + g_{\mathrm{Im}}(u) \cdot \boldsymbol{S}_m^M(u), \tag{13.49}$$

$$G_{\mathrm{Im}}(m) = \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g_{\mathrm{Im}}(u) \cdot \boldsymbol{C}_m^M(u) - g_{\mathrm{Re}}(u) \cdot \boldsymbol{S}_m^M(u), \tag{13.50}$$

for $m = 0 \ldots M-1$. Analogously, the *inverse* DFT in Eqn. (13.46) expands to

$$g(u) = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} \Big[\underbrace{G_{\mathrm{Re}}(m) + \mathrm{i}\cdot G_{\mathrm{Im}}(m)}_{G(m)}\Big] \cdot \Big[\underbrace{\cos\big(2\pi\tfrac{mu}{M}\big)}_{\boldsymbol{C}_m^M(u)} + \mathrm{i}\cdot\underbrace{\sin\big(2\pi\tfrac{mu}{M}\big)}_{\boldsymbol{S}_m^M(u)}\Big], \tag{13.51}$$

and thus the real and imaginary parts of the reconstructed signal are

$$g_{\mathrm{Re}}(u) = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G_{\mathrm{Re}}(m) \cdot \boldsymbol{C}_m^M(u) - G_{\mathrm{Im}}(m) \cdot \boldsymbol{S}_m^M(u), \tag{13.52}$$

$$g_{\mathrm{Im}}(u) = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G_{\mathrm{Im}}(m) \cdot \boldsymbol{C}_m^M(u) + G_{\mathrm{Re}}(m) \cdot \boldsymbol{S}_m^M(u), \tag{13.53}$$

for $u = 0 \ldots M-1$.

13.3.2 Discrete Basis Functions

Equation (13.51) describes the decomposition of the discrete function $g(u)$ into a finite sum of M discrete cosine and sine functions $(\boldsymbol{C}_m^M, \boldsymbol{S}_m^M)$ whose weights (or "amplitudes") are determined by the DFT coefficients in $G(m)$. Each of these one-dimensional basis functions,

$$\boldsymbol{C}_m^M(u) = \boldsymbol{C}_u^M(m) = \cos\bigl(2\pi \tfrac{mu}{M}\bigr) = \cos(\omega_m u), \tag{13.54}$$

$$\boldsymbol{S}_m^M(u) = \boldsymbol{S}_u^M(m) = \sin\bigl(2\pi \tfrac{mu}{M}\bigr) = \sin(\omega_m u), \tag{13.55}$$

is periodic with M and has a discrete frequency (wave number) m, which corresponds to the angular frequency

$$\omega_m = 2\pi \frac{m}{M}.$$

As an example, Figs. 13.11 and 13.12 show the discrete basis functions (with integer ordinate values $u \in \mathbb{Z}$) for the DFT of length $M = 8$ as well as their continuous counterparts (with ordinate values $x \in \mathbb{R}$).

For wave number $m = 0$, the cosine function $\boldsymbol{C}_0^M(u)$ (Eqn. (13.54)) has the constant value 1. The corresponding DFT coefficient $G_{\mathrm{Re}}(0)$—the real part of $G(0)$—thus specifies the constant part of the signal or the average value of the signal $g(u)$ in Eqn. (13.52). In contrast, the zero-frequency sine function $\boldsymbol{S}_0^M(u)$ is zero for any value of u and thus cannot contribute anything to the signal. The corresponding DFT coefficients $G_{\mathrm{Im}}(0)$ in Eqn. (13.52) and $G_{\mathrm{Re}}(0)$ in Eqn. (13.53) are therefore of no relevance. For a real-valued signal (i. e., $g_{\mathrm{Im}}(u) = 0$ for all u), the coefficient $G_{\mathrm{Im}}(0)$ in the corresponding Fourier spectrum must also be zero.

As shown in Fig. 13.11, the wave number $m = 1$ relates to a cosine or sine function that performs exactly one full cycle over the signal length $M = 8$. Similarly, the wave numbers $m = 2 \ldots 7$ correspond to $2 \ldots 7$ complete cycles over the signal length M (Figs. 13.11 and 13.12).

13.3.3 Aliasing Again!

A closer look at Figs. 13.11 and 13.12 reveals an interesting fact: the sampled (discrete) cosine and sine functions for $m = 3$ and $m = 5$ are *identical*, although their continuous counterparts are different! The same

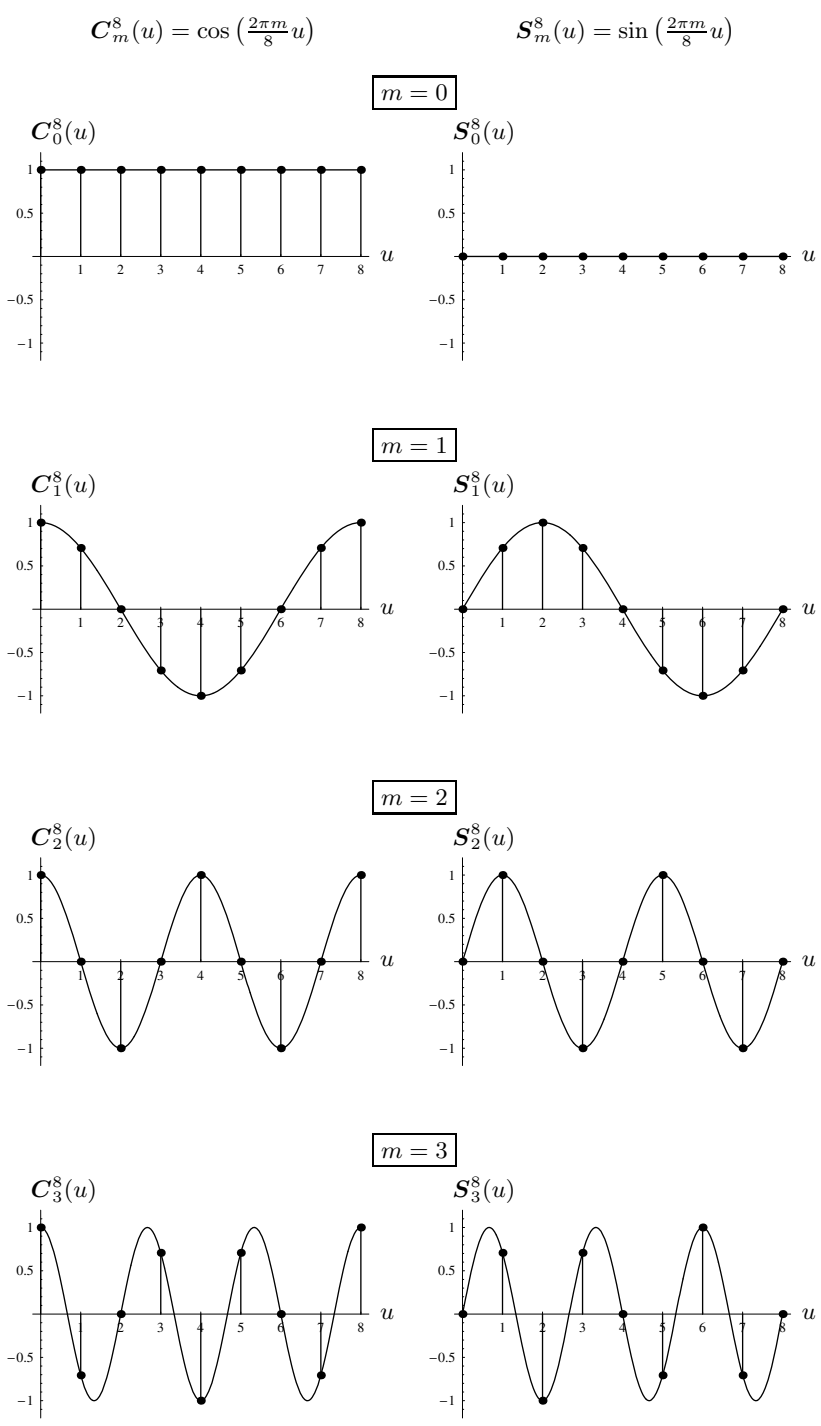


Fig. 13.11
Discrete basis functions $\boldsymbol{C}_m^M(u)$ and $\boldsymbol{S}_m^M(u)$ for the signal length $M = 8$ and wave numbers $m = 0 \ldots 3$. Each plot shows both the discrete function (round dots) and the corresponding continuous function.

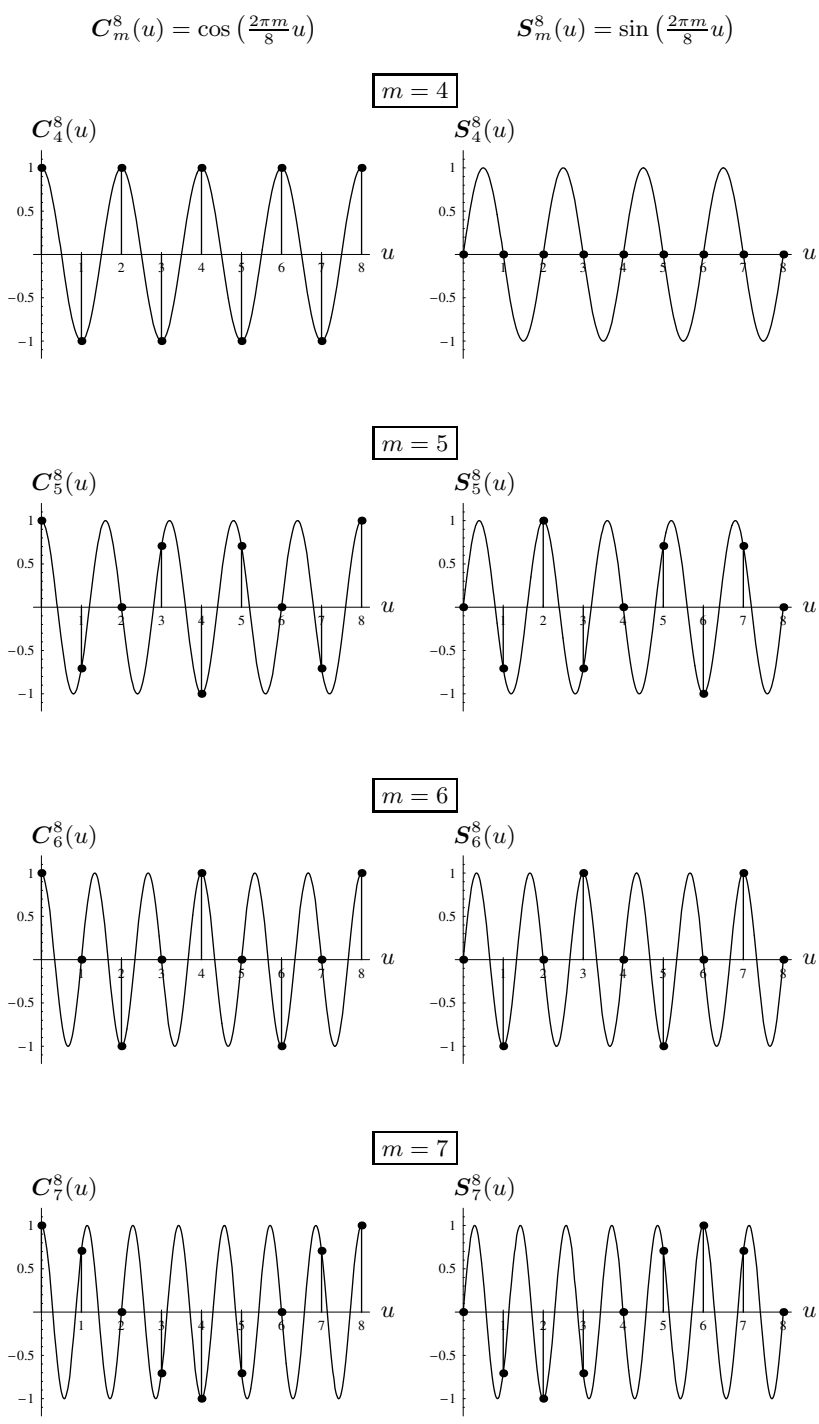


Fig. 13.12
Discrete basis functions (continued). Signal length $M = 8$ and wave numbers $m = 4 \ldots 7$. Notice that, for example, the discrete functions for $m = 5$ and $m = 3$ (Fig. 13.11) are identical because $m = 4$ is the maximum wave number that can be represented in a discrete spectrum of length $M = 8$.

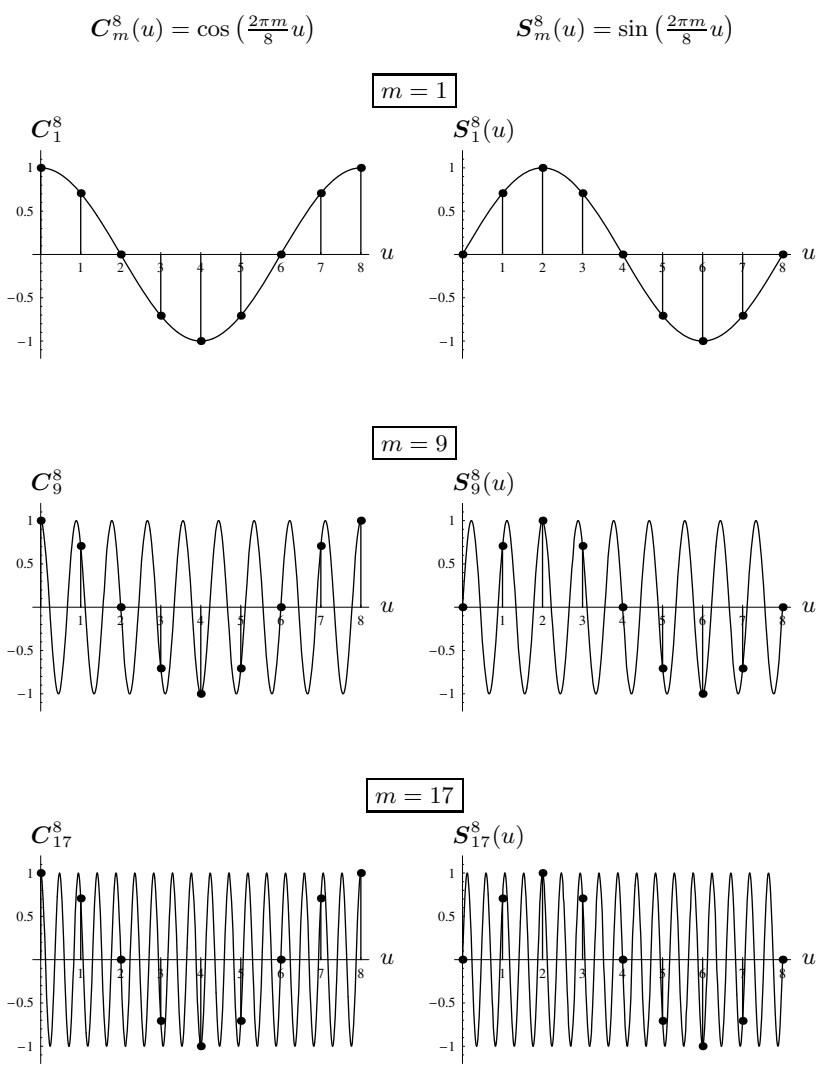


Fig. 13.13
Aliasing in signal space. For the signal length $M = 8$, the discrete cosine and sine basis functions for the wave numbers $m = 1, 9, 17, \ldots$ (round dots) are all identical. The sampling frequency itself corresponds to the wave number $m = 8$.

is true for the frequency pairs $m = 2, 6$ and $m = 1, 7$. What we see here is another manifestation of the sampling theorem—which we had originally encountered (Sec. 13.2.1) in frequency space—in *signal space*.

Obviously, $m = 4$ is the maximum frequency component that can be represented by a discrete signal of length $M = 8$. Any discrete function with a higher frequency ($m = 5 \ldots 7$ in this case) has an identical counterpart with a lower wave number and thus cannot be reconstructed from the sampled signal!

If a continuous signal is sampled at a regular distance τ, the corresponding Fourier spectrum is repeated at multiples of $\omega_s = 2\pi/\tau$, as we have shown earlier (Fig. 13.8). In the discrete case, the spectrum is periodic with length M. Since the Fourier spectrum of a real-valued signal is

symmetric about the origin (Eqn. (13.22)), there is for every coefficient with wave number m an equal-sized duplicate with wave number $-m$. Thus the spectral components appear pairwise and mirrored at multiples of M; i. e.,

$$\begin{aligned} |G(m)| = |G(M-m)| &= |G(M+m)| \\ = |G(2M-m)| &= |G(2M+m)| \\ &\dots \\ = |G(kM-m)| &= |G(kM+m)| \end{aligned} \tag{13.56}$$

for all $k \in \mathbb{Z}$. If the original continuous signal contains "energy" with the frequencies

$$\omega_m > \omega_{M/2}$$

(i. e., signal components with wave numbers $m > M/2$), then, according to the sampling theorem, the overlapping parts of the spectra are superimposed in the resulting periodic spectrum of the discrete signal.

13.3.4 Units in Signal and Frequency Space

The relationship between the units in signal and frequency space and the interpretation of wave numbers m is a common cause of confusion. While the discrete signal and its spectrum are simple numerical vectors and units of measurement are irrelevant for computing the DFT itself, it is nevertheless important to understand how the coordinates in the spectrum relate to physical dimensions in the real world.

Clearly, every complex-valued spectral coefficient $G(m)$ corresponds to one pair of cosine and sine functions with a particular frequency in signal space. Assume a continuous signal is sampled at M consecutive positions spaced at τ (an interval in time or distance in space). The *wave number* $m = 1$ then corresponds to the *fundamental period* of the discrete signal (which is now assumed to be periodic) with a period of length $M\tau$; i. e., to the *frequency*

$$f_1 = \frac{1}{M\tau}. \tag{13.57}$$

In general, the wave number m of a discrete spectrum relates to the physical frequency as

$$f_m = m\frac{1}{M\tau} = m \cdot f_1 \tag{13.58}$$

for $0 \le m < M$, which is equivalent to the angular frequency

$$\omega_m = 2\pi f_m = m\frac{2\pi}{M\tau} = m \cdot \omega_1. \tag{13.59}$$

Obviously then, the sampling frequency $f_s = 1/\tau = M \cdot f_1$ corresponds to the wave number $m_s = M$. As expected, the maximum nonaliased wave number in the spectrum is

$$m_{\max} = \frac{M}{2} = \frac{m_s}{2}, \tag{13.60}$$

half the wave number of the sampling frequency m_s.

Example 1: Time-domain signal

We assume for this example that $g(u)$ is a signal in the time domain (e. g., a discrete sound signal) that contains $M = 500$ sample values taken at regular intervals $\tau = 1\,\text{ms} = 10^{-3}\,\text{s}$. Thus the sampling frequency is $f_s = 1/\tau = 1000\,\text{Hertz}$ (cycles per second) and the total duration (fundamental period) of the signal is $M\tau = 0.5\,\text{s}$.

The signal is implicitly periodic, and from Eqn. (13.57) we obtain its fundamental frequency as $f_1 = \frac{1}{500 \cdot 10^{-3}} = \frac{1}{0.5} = 2\,\text{Hertz}$. The wave number $m = 2$ in this case corresponds to a real frequency $f_2 = 2f_1 = 4\,\text{Hertz}$, $f_3 = 6\,\text{Hertz}$, etc. The maximum frequency that can be represented by this discrete signal without aliasing is $f_{\max} = \frac{M}{2} f_1 = \frac{1}{2\tau} = 500\,\text{Hertz}$, exactly half the sampling frequency f_s.

Example 2: Space-domain signal

Assume we have a one-dimensional print pattern with a resolution (i. e., spatial sampling frequency) of 120 dots per cm, which equals approximately 300 dots per inch (dpi) and a total signal length of $M = 1800$ samples. This corresponds to a spatial sampling interval of $\tau = 1/120\,\text{cm} \approx 83\,\mu\text{m}$ and a physical signal length of $(1800/120)\,\text{cm} = 15\,\text{cm}$.

The fundamental frequency of this signal (again implicitly assumed to be periodic) is $f_1 = \frac{1}{15}$, expressed in cycles per cm. The sampling frequency is $f_s = 120$ cycles per cm and thus the maximum signal frequency is $f_{\max} = \frac{f_s}{2} = 60$ cycles per cm. The maximum signal frequency specifies the finest structure ($\frac{1}{60}$ cm) that can be reproduced by this print raster.

13.3.5 Power Spectrum

The *magnitude* of the complex-valued Fourier spectrum

$$|G(m)| = \sqrt{G_{\text{Re}}^2(m) + G_{\text{Im}}^2(m)} \tag{13.61}$$

is commonly called the "power spectrum" of a signal. It specifies the energy that individual frequency components in the spectrum contribute to the signal. The power spectrum is real-valued and positive and thus often used for graphically displaying the results of Fourier transforms (see also Sec. 14.2).

Since all phase information is lost in the power spectrum, the original signal cannot be reconstructed from the power spectrum alone. However,

because of the missing phase information, the power spectrum is insensitive to shifts of the original signal and can thus be efficiently used for comparing signals. To be more precise, the power spectrum of a cyclically shifted signal is identical to the power spectrum of the original signal. Thus, given a discrete periodic signal $g_1(u)$ of length M and a second signal $g_2(u)$ shifted by some offset d, such that

$$g_2(u) = g_1(u-d), \tag{13.62}$$

the corresponding power spectra are the same,

$$|G_2(m)| = |G_1(m)|, \tag{13.63}$$

although in general the complex-valued spectra $G_1(m)$ and $G_2(m)$ are different. Furthermore, from the symmetry property of the Fourier spectrum, it follows that

$$|G(m)| = |G(-m)| \tag{13.64}$$

for real-valued signals $g(u) \in \mathbb{R}$.

13.4 Implementing the DFT

13.4.1 Direct Implementation

Based on the definitions in Eqns. (13.49)–(13.50) the DFT can be directly implemented, as shown in Prog. 13.1. The main method `DFT()` transforms a signal vector of arbitrary length M (not necessarily a power of 2). It requires roughly M^2 operations (multiplications and additions); i. e., the time complexity of this DFT algorithm is $\mathcal{O}(M^2)$.

One way to improve the efficiency of the DFT algorithm is to use lookup tables for the sin and cos functions (which are relatively "expensive" to compute) since only function values for a set of M different angles φ_m are ever needed. The angles $\varphi_m = 2\pi\frac{m}{M}$ corresponding to $m = 0 \ldots M-1$ are evenly distributed over the full 360° circle. Any integral multiple $\varphi_m \cdot u$ (for $u \in \mathbb{Z}$) can only fall onto one of these angles again because

$$\varphi_m \cdot u = 2\pi\tfrac{mu}{M} \quad \equiv \quad \tfrac{2\pi}{M}(\underbrace{mu \bmod M}_{0 \le k < M}) = 2\pi\tfrac{k}{M} = \varphi_k, \tag{13.65}$$

where mod denotes the "modulus" operator.[10] Thus we can set up two constant tables (floating-point arrays) $\mathbf{W}_C[k]$ and $\mathbf{W}_S[k]$ of size M with the values

$$\begin{aligned} \mathbf{W}_C[k] \leftarrow \cos(\omega_k) &= \cos\left(2\pi\tfrac{k}{M}\right), \\ \mathbf{W}_S[k] \leftarrow \sin(\omega_k) &= \sin\left(2\pi\tfrac{k}{M}\right), \end{aligned}$$

[10] See also Appendix B.1.2.

Program 13.1
Direct implementation of the DFT based on the definition in Eqns. (13.49) and (13.50). The method `DFT()` returns a complex-valued vector with the same length as the complex-valued input (signal) vector `g`. This method implements both the forward and the inverse transforms, controlled by the Boolean parameter `forward`. The class `Complex` (top) defines the structure of the complex-valued vector elements.

```
1  class Complex {
2      double re, im;
3
4      Complex(double re, double im) { //constructor method
5          this.re = re;
6          this.im = im;
7      }
8  }
```

```
9      Complex[] DFT(Complex[] g, boolean forward) {
10         int M = g.length;
11         double s = 1 / Math.sqrt(M); //common scale factor
12         Complex[] G = new Complex[M];
13         for (int m = 0; m < M; m++) {
14             double sumRe = 0;
15             double sumIm = 0;
16             double phim = 2 * Math.PI * m / M;
17             for (int u = 0; u < M; u++) {
18                 double gRe = g[u].re;
19                 double gIm = g[u].im;
20                 double cosw = Math.cos(phim * u);
21                 double sinw = Math.sin(phim * u);
22                 if (!forward) // inverse transform
23                     sinw = -sinw;
24                 //complex mult: [g_Re + i g_Im] · [cos(ω) + i sin(ω)]
25                 sumRe += gRe * cosw + gIm * sinw;
26                 sumIm += gIm * cosw - gRe * sinw;
27             }
28             G[m] = new Complex(s * sumRe, s * sumIm);
29         }
30         return G;
31     }
```

for $0 \le k < M$. For computing the DFT, the necessary cosine and sine values (Eqn. (13.48)) can be read from these tables as

$$\boldsymbol{C}_k^M(u) = \cos\bigl(2\pi\tfrac{mu}{M}\bigr) = \mathbf{W}_C[mu \bmod M], \tag{13.66}$$

$$\boldsymbol{S}_k^M(u) = \sin\bigl(2\pi\tfrac{mu}{M}\bigr) = \mathbf{W}_S[mu \bmod M], \tag{13.67}$$

for arbitrary values of m and u, without any additional computation. The necessary modification of the `DFT()` method in Prog. 13.1 is left as an exercise (Exercise 13.5).

Despite this significant improvement, the direct implementation of the DFT remains computationally intensive. As a matter of fact, it has been impossible for a long time to compute this form of DFT in sufficiently short time on off-the-shelf computers, and this is still true today for many real applications.

13.4.2 Fast Fourier Transform (FFT)

Fortunately, for computing the DFT in practice, fast algorithms exist that lay out the sequence of computations in such a way that intermediate results are only computed once and optimally reused many times. This "fast Fourier transform", which exists in many variations, generally reduces the time complexity of the computation from $\mathcal{O}(M^2)$ to $\mathcal{O}(M \log_2 M)$. The benefits are substantial, in particular for longer signals. For example, with a signal of length $M = 10^3$, the FFT leads to a speedup by a factor of 100 over the direct DFT implementation and an impressive gain of 10,000 times for a signal of length $M = 10^6$. Since its invention, the FFT has therefore become an indispensable tool in almost any application of spectral signal analysis [14].

Most FFT algorithms, including the one described in the famous publication by Cooley and Tukey in 1965 (see [38, p. 156] for a historic overview), are designed for signals of length $M = 2^k$ (i. e., powers of 2). However, FFT algorithms have also been developed for other lengths, including several small prime numbers [8]. It is important to remember, though, that the DFT and FFT compute exactly the *same* result and the FFT is only a special—though ingenious—method for *implementing* the discrete Fourier transform (Eqn. (13.45)).

13.5 Exercises

Exercise 13.1. Compute the values of the cosine function $f(x) = \cos(\omega x)$ with angular frequency $\omega = 5$ for the positions $x = -3, -2, \ldots, 2, 3$. What is the length of this function's period?

Exercise 13.2. Determine the phase angle φ of the function $f(x) = A \cdot \cos(\omega x) + B \cdot \sin(\omega x)$ for $A = -1$ and $B = 2$.

Exercise 13.3. Compute the real part, the imaginary part, and the magnitude of the complex value $z = 1.5 \cdot e^{-\mathrm{i}\,2.5}$.

Exercise 13.4. A one-dimensional optical scanner for sampling film transparencies is supposed to resolve image structures with a precision of 4,000 dpi (dots per inch). What spatial distance (in mm) between samples is required such that no aliasing occurs?

Exercise 13.5. Modify the direct implementation of the one-dimensional DFT given in Prog. 13.1 by using lookup tables for the cos and sin functions as described in Eqns. (13.66) and (13.67).

14

The Discrete Fourier Transform in 2D

The Fourier transform is defined not only for one-dimensional signals but for functions of arbitrary dimension. Thus, two-dimensional images are nothing special from a mathematical point of view.

14.1 Definition of the 2D DFT

For a two-dimensional, periodic function (e. g., an intensity image) $g(u, v)$ of size $M \times N$, the discrete Fourier transform (2D DFT) is defined as

$$\begin{aligned} G(m,n) &= \frac{1}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u,v) \cdot e^{-\mathrm{i}2\pi \frac{mu}{M}} \cdot e^{-\mathrm{i}2\pi \frac{nv}{N}} \\ &= \frac{1}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u,v) \cdot e^{-\mathrm{i}2\pi (\frac{mu}{M} + \frac{nv}{N})} \end{aligned} \tag{14.1}$$

for the spectral coordinates $m = 0 \ldots M-1$ and $n = 0 \ldots N-1$. As we see, the resulting Fourier transform is again a two-dimensional function of the same size ($M \times N$) as the original signal. Similarly, the *inverse* 2D DFT is defined as

$$\begin{aligned} g(u,v) &= \frac{1}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(m,n) \cdot e^{\mathrm{i}2\pi \frac{mu}{M}} \cdot e^{\mathrm{i}2\pi \frac{nv}{N}} \\ &= \frac{1}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(m,n) \cdot e^{\mathrm{i}2\pi (\frac{mu}{M} + \frac{nv}{N})} \end{aligned} \tag{14.2}$$

for the image coordinates $u = 0 \ldots M-1$ and $v = 0 \ldots N-1$.

14.1.1 2D Basis Functions

Equation (14.2) shows that a discrete two-dimensional, periodic function $g(u, v)$ can be represented as a linear combination (i. e., as a weighted sum) of 2D sinusoids of the form

$$e^{\mathrm{i}2\pi\left(\frac{mu}{M}+\frac{nv}{N}\right)} = e^{\mathrm{i}(\omega_m u+\omega_n v)} \tag{14.3}$$

$$= \underbrace{\cos\left[2\pi\left(\frac{mu}{M}+\frac{nv}{N}\right)\right]}_{\boldsymbol{C}^{M,N}_{m,n}(u,v)} + \mathrm{i}\cdot\underbrace{\sin\left[2\pi\left(\frac{mu}{M}+\frac{nv}{N}\right)\right]}_{\boldsymbol{S}^{M,N}_{m,n}(u,v)}. \tag{14.4}$$

$\boldsymbol{C}^{M,N}_{m,n}(u, v)$ and $\boldsymbol{S}^{M,N}_{m,n}(u, v)$ are discrete, two-dimensional cosine and sine functions with horizontal and vertical wave numbers n and m, respectively, and the corresponding angular frequencies ω_m, ω_n:

$$\boldsymbol{C}^{M,N}_{m,n}(u, v) = \cos\left[2\pi\left(\frac{mu}{M}+\frac{nv}{N}\right)\right] = \cos(\omega_m u + \omega_n v), \tag{14.5}$$

$$\boldsymbol{S}^{M,N}_{m,n}(u, v) = \sin\left[2\pi\left(\frac{mu}{M}+\frac{nv}{N}\right)\right] = \sin(\omega_m u + \omega_n v). \tag{14.6}$$

Each of these basis functions is periodic with M units in the horizontal direction and N units in the vertical direction.

Examples

Figures 14.1 and 14.2 show a set of 2D cosine functions $\boldsymbol{C}^{M,N}_{m,n}$ of size $M \times N = 16 \times 16$ for various combinations of wave numbers $m, n = 0 \ldots 3$. As we can clearly see, these functions correspond to a directed, cosine-shaped waveform whose orientation is determined by the wave numbers m and n. For example, the wave numbers $m = n = 2$ specify a cosine function $\boldsymbol{C}^{M,N}_{2,2}(u, v)$ that performs two full cycles in both the horizontal and vertical directions, thus creating a diagonally oriented, two-dimensional wave. Of course, the same holds for the corresponding sine functions.

14.1.2 Implementing the Two-Dimensional DFT

As in the one-dimensional case, we could directly use the definition in Eqn. (14.1) to write a program or procedure that implements the 2D DFT. However, this is not even necessary. A minor rearrangement of Eqn. (14.1) into

$$G(m, n) = \frac{1}{\sqrt{N}} \sum_{v=0}^{N-1} \underbrace{\left[\frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g(u, v) \cdot e^{-\mathrm{i}2\pi\frac{mu}{M}}\right]}_{\text{1-dim. DFT of row } g(\cdot, v)} \cdot\ e^{-\mathrm{i}2\pi\frac{nv}{N}} \tag{14.7}$$

Algorithm 14.1
In-place implementation of the two-dimensional DFT as a sequence of one-dimensional DFTs on row and column vectors.

```
1: Separable 2D-DFT (g)          ▷ g(u,v) ∈ ℂ, 0 ≤ u < M, 0 ≤ v < N
2:     for v ← 0 ... N−1 do
3:         Let g(·,v) be the vth row vector of g:
               Replace g(·,v) by DFT(g(·,v))
4:     for u ← 0 ... M−1 do
5:         Let g(u,·) be the uth column vector of g:
               Replace g(u,·) by DFT(g(u,·))
       Remark: g(u,v) ≡ G(u,v) ∈ ℂ now contains the discrete 2D spectrum.
6:     return g
```

shows that its core contains a *one-dimensional* DFT (see Eqn. (13.45)) of the vth row vector $g(\cdot, v)$ that is independent of the "vertical" position v and size N (noting the fact that v and N are placed outside the square brackets in Eqn. (14.7)). If, in a first step, we *replace* each *row* vector $g(\cdot, v)$ of the original image by its one-dimensional Fourier transform,

$$g'(\cdot, v) \leftarrow \mathsf{DFT}\big(g(\cdot, v)\big) \quad \text{for } 0 \le v < N,$$

then we only need to replace each resulting *column* vector by its one-dimensional DFT in a second step:

$$g''(u, \cdot) \leftarrow \mathsf{DFT}\big(g'(u, \cdot)\big) \quad \text{for } 0 \le u < M.$$

The resulting function $g''(u, v)$ is precisely the two-dimensional Fourier transform $G(m, n)$. Thus the *two-dimensional* DFT can be separated into a sequence of one-dimensional DFTs over the row and column vectors, respectively, as summarized in Alg. 14.1. The advantage of this is twofold: first, the 2D-DFT can be implemented more efficiently, and second, only a one-dimensional implementation of the DFT (or the one-dimensional FFT, as described in Sec. 13.4.2) is needed to implement any multidimensional DFT.

As we can see from Eqn. (14.7), the 2D DFT could equally be performed in the opposite way, by first doing a 1D DFT on all *rows* and subsequently on all *columns*. One should also note that all operations in Alg. 14.1 are done "in place", which means that the original signal $g(u, v)$ is destructively modified and successively replaced by its Fourier transform $G(m, n)$ of the same size, without allocating any additional storage space. This feature is certainly desirable and also quite common, based on the fact that most one-dimensional FFT algorithms (which should be used for implementing the DFT in practice) work "in place".

14.2 Visualizing the 2D Fourier Transform

Unfortunately, there is no simple method for visualizing two-dimensional complex-valued functions, such as the result of a 2D DFT. One alternative is to display the real and imaginary parts individually as 2D surfaces. Mostly, however, the absolute value of the complex functions

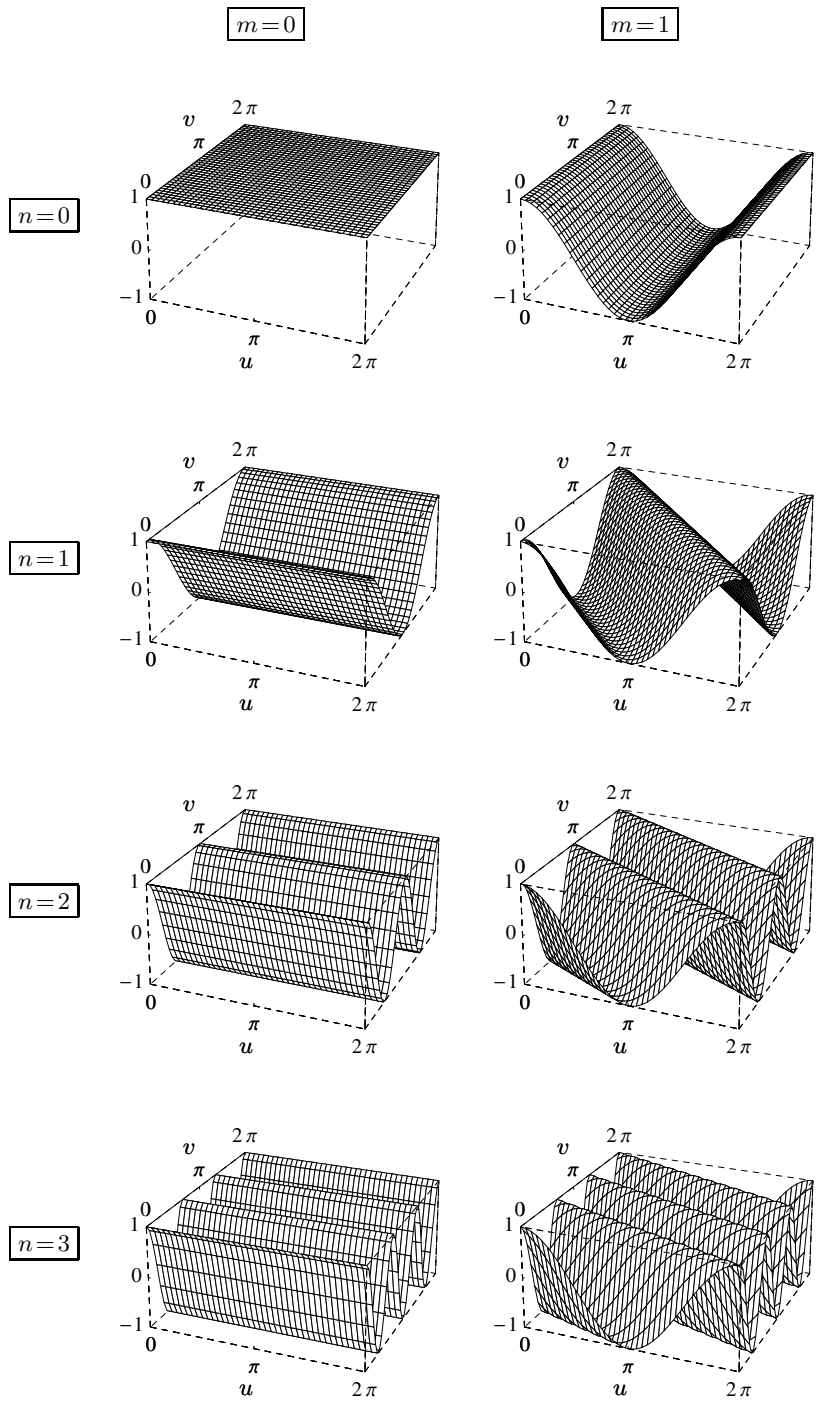


Fig. 14.1
Two-dimensional cosine functions. $\boldsymbol{C}^{M,N}_{m,n}(u,v) = \cos\left[2\pi\left(\frac{mu}{M}+\frac{nv}{N}\right)\right]$ for $M = N = 16$, $n = 0\ldots3$, $m = 0, 1$.

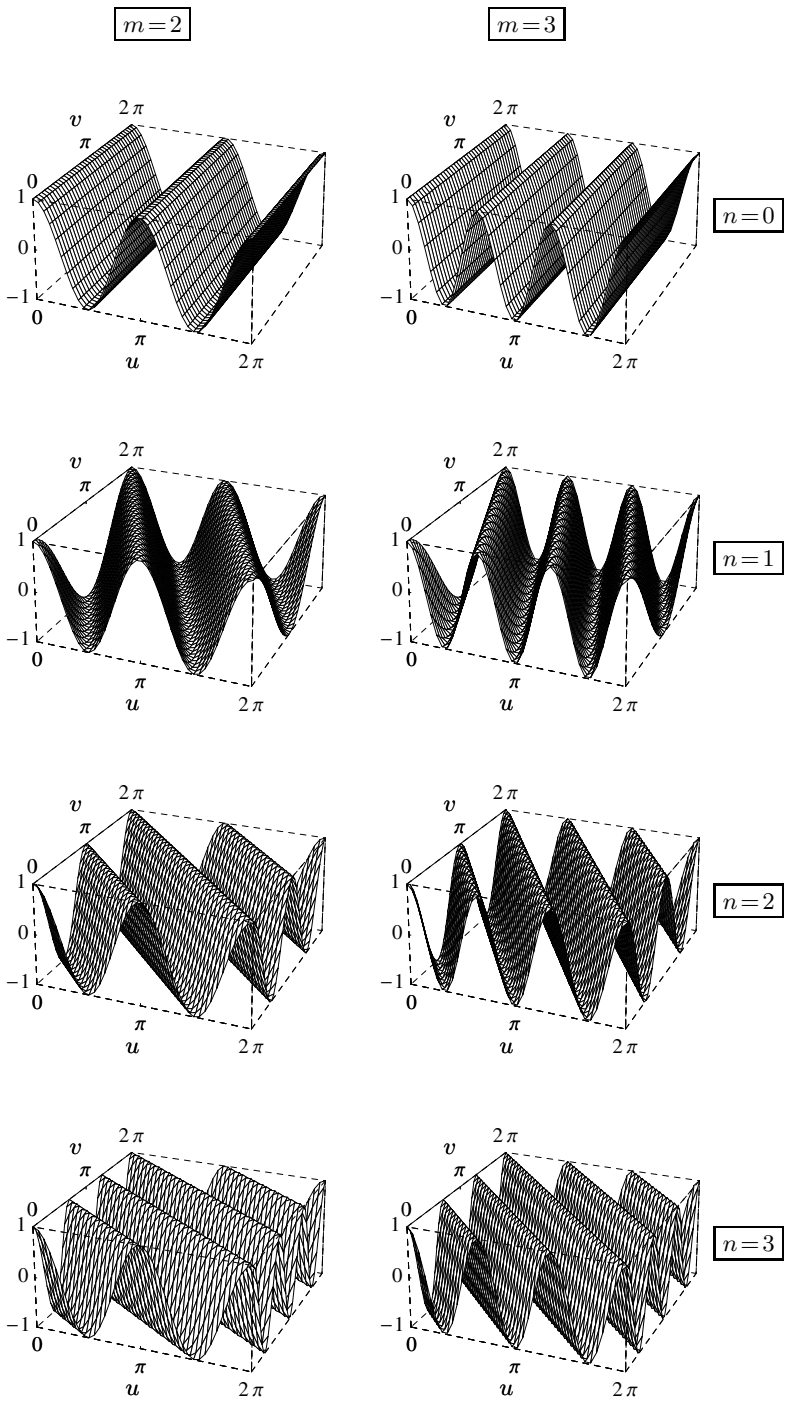


Fig. 14.2
Two-dimensional cosine functions (*continued*). $\boldsymbol{C}^{M,N}_{m,n}(u,v) = \cos\left[2\pi\left(\frac{mu}{M} + \frac{nv}{N}\right)\right]$ for $M = N = 16$, $n = 0 \ldots 3$, $m = 2, 3$.

is displayed, which in the case of the Fourier transform is $|G(m,n)|$, the *power spectrum* (see Sec. 13.3.5).

14.2.1 Range of Spectral Values

For most natural images, the "spectral energy" concentrates at the lower frequencies with a clear maximum at wave numbers $(0,0)$; i. e., at the co-ordinate center (see also Sec. 14.4). The values of the power spectrum usually cover a wide range, and displaying them linearly often makes the smaller values invisible. To show the full range of spectral values, in particular the smaller values for the high frequencies, it is common to display the square root or the logarithm of the power spectrum, $\sqrt{|G(m,n)|}$ or $\log|G(m,n)|$, respectively.

14.2.2 Centered Representation

Analogous to the one-dimensional case, the 2D spectrum is a periodic function in both dimensions,

$$G(m,n) = G(m+pM, n+qN), \tag{14.8}$$

for arbitrary $p,q \in \mathbb{Z}$. In the case of a real-valued 2D signal ($g(u,v) \in \mathbb{R}$, see Eqn. (13.56)), the power spectrum is also *symmetric* about the origin,

$$|G(m,n)| = |G(-m,-n)|\,. \tag{14.9}$$

It is thus common to use a centered representation of the spectrum with coordinates m,n in the ranges

$$-\left\lfloor \tfrac{M}{2} \right\rfloor \le m \le \left\lfloor \tfrac{M-1}{2} \right\rfloor \quad \text{and} \quad -\left\lfloor \tfrac{N}{2} \right\rfloor \le n \le \left\lfloor \tfrac{N-1}{2} \right\rfloor,$$

respectively. This can be easily accomplished by swapping the four quadrants of the Fourier transform, as illustrated in Fig. 14.3. In the resulting representation, the low-frequency coefficients are found at the center and the high-frequency entries along the outer boundaries. Figure 14.4 shows the plot of a 2D power spectrum as an intensity image in its original and centered form, with the intensity proportional to the logarithm of the spectral values ($\log_{10}|G(m,n)|$).

14.3 Frequencies and Orientation in 2D

As we could see in Figs. 14.1 and 14.2, each 2D basis function is an oriented cosine or sine function whose orientation and frequency are determined by its wave numbers m and n for the horizontal and vertical directions, respectively. If we moved along the main direction of such a basis function (i. e., perpendicular to the crest of the waves), we would follow a one-dimensional cosine or sine function of some frequency $\hat{f}$, which we call the *directional* or *effective frequency* of the waveform (see Fig. 14.5).

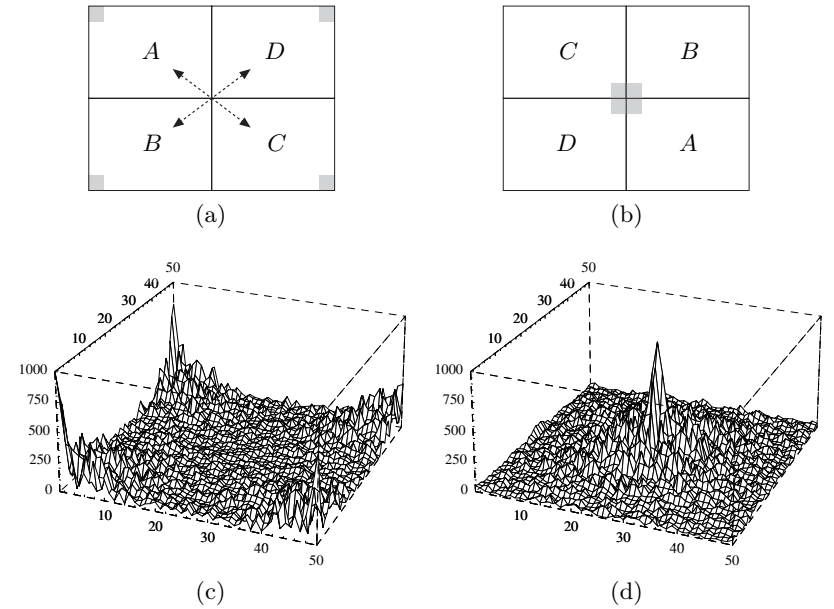


Fig. 14.3
Centering the 2D Fourier spectrum. In the original (non-centered) spectrum, the coordinate center (i. e., the region of low frequencies) is located in the upper left corner and, due to the periodicity of the spectrum, also at all other corners (a). In this case, the coefficients for the highest wave numbers (frequencies) lie at the center. Swapping the quadrants pairwise, as shown in (b), moves all low-frequency coefficients to the center and high frequencies to the periphery. A real 2D power spectrum is shown in its original form in (c) and in centered form in (d).

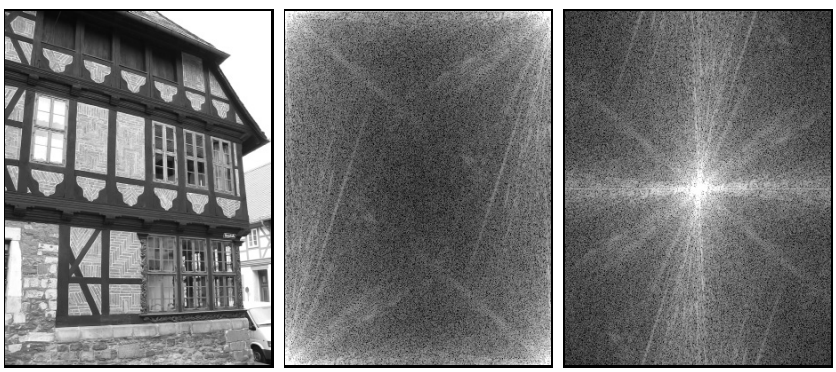


Fig. 14.4
Intensity plot of a 2D power spectrum: original image (a), non-centered spectrum (b), and centered spectrum (c).

14.3.1 Effective Frequency

As we remember, the wave numbers m, n specify how many full cycles the corresponding 2D basis function performs over a distance of M units in the horizontal direction or N units in the vertical direction. The effective frequency along the main direction of the wave can be derived from the one-dimensional case (Eqn. (13.57)) as

$$\hat{f}_{(m,n)} = \frac{1}{\tau}\sqrt{\left(\frac{m}{M}\right)^2 + \left(\frac{n}{N}\right)^2}, \tag{14.10}$$

where we assume the same (fixed) spatial sampling interval along the x and y axes (i. e., $\tau = \tau_x = \tau_y$). Thus the maximum signal frequency in the directions of the coordinate axes is

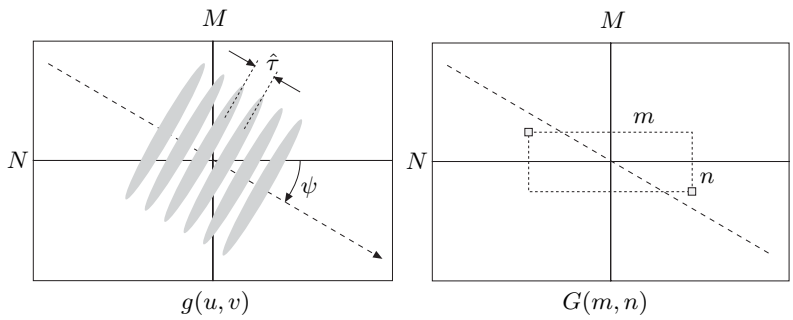


Fig. 14.5
Frequency and orientation in 2D. The image (left) contains some periodic pattern with effective frequency $\hat{f} = 1/\hat{\tau}$ and orientation ψ. The frequency coefficient corresponding to this pattern is located at position $(m, n) = \pm\hat{f} \cdot (M \cos\psi, N \sin\psi)$ in the 2D power spectrum (right). Thus in general the spectral coefficients (m, n) are not placed at the same direction (with respect to the origin) as the orientation of the image pattern implies.

$$\hat{f}_{(\pm\frac{M}{2},0)} = \hat{f}_{(0,\pm\frac{N}{2})} = \tfrac{1}{\tau}\sqrt{\left(\tfrac{1}{2}\right)^2} = \tfrac{1}{2\tau} = \tfrac{1}{2}f_s, \tag{14.11}$$

where $f_s = \frac{1}{\tau}$ denotes the sampling frequency. Notice that the effective signal frequency at the corners of the spectrum is

$$\hat{f}_{(\pm\frac{M}{2},\pm\frac{N}{2})} = \tfrac{1}{\tau}\sqrt{\left(\tfrac{1}{2}\right)^2 + \left(\tfrac{1}{2}\right)^2} = \tfrac{1}{\sqrt{2}\cdot\tau} = \tfrac{1}{\sqrt{2}}f_s, \tag{14.12}$$

which is a factor $\sqrt{2}$ higher than along the coordinate axes (Eqn. (14.11)).

14.3.2 Frequency Limits and Aliasing in 2D

Figure 14.6 illustrates the relationship described in Eqns. (14.11) and (14.12). The highest permissible signal frequencies in any direction lie along the boundary of the centered 2D spectrum of size $M \times N$. Any signal with all frequency components *within* this region complies with the sampling theorem (Nyquist rule) and can thus be reconstructed without aliasing. In contrast, any spectral component *outside* these limits is reflected across the boundary of this box toward the coordinate center onto lower frequencies, which would appear as visual aliasing in the reconstructed image.

Apparently the lowest effective sampling frequency (Eqn. (14.10)) occurs in the directions of the coordinate axes of the sampling grid. To ensure that a certain image pattern can be reconstructed without aliasing at any orientation, the effective signal frequency $\hat{f}$ of that pattern must be limited to $\frac{f_s}{2} = \frac{1}{2\tau}$ in every direction, again assuming that the sampling interval τ is the same along both coordinate axes.

14.3.3 Orientation

The spatial orientation of a two-dimensional cosine or sine wave with spectral coordinates m, n (wave numbers $0 \le m < M$, $0 \le n < N$) is

$$\psi_{(m,n)} = \mathrm{ArcTan}\left(\tfrac{m}{M}, \tfrac{n}{N}\right) = \mathrm{ArcTan}\big(mN, nM\big), \tag{14.13}$$

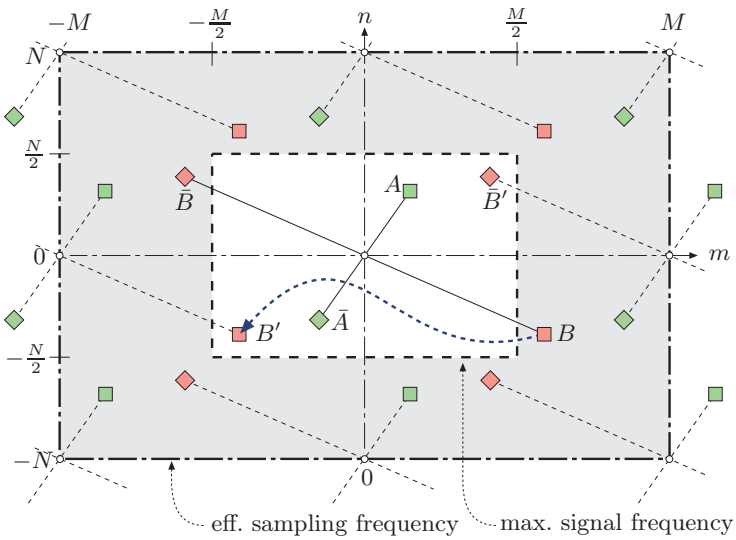


Fig. 14.6
Maximum signal frequencies and aliasing in 2D. The boundary of the $M \times N$ 2D spectrum (inner rectangle) marks the region of permissible signal frequencies along any direction. The outer rectangle corresponds to the effective sampling frequency, which is twice the maximum signal frequency in the same direction. The signal component with spectral position A ($\bar{A}$, respectively) lies *within* the maximum representable frequency range and thus causes no *aliasing*. In contrast, component B ($\bar{B}$, respectively) is outside the permissible frequency range. Due to the periodicity of the spectrum, the components repeat, as in the one-dimensional case, at every multiple of the sampling frequencies along the m and n axes. Thus component B appears as an "alias" at position B' (and $\bar{B}$ appears at position $\bar{B}'$) in the visible part of the spectrum. Notice that aliasing changes both the frequencies and directions of the affected components in signal space.

where $\psi_{(m,n)}$ for $m = n = 0$ is of course undefined.[1] Conversely, a two-dimensional sinusoid with effective frequency $\hat{f}$ and spatial orientation ψ is represented by the spectral coordinates

$$(m, n) = \pm\hat{f} \cdot (M \cos\psi, N \sin\psi)\,, \tag{14.14}$$

as shown before in Fig. 14.5.

14.3.4 Correcting the Geometry of a 2D Spectrum

From Eqn. (14.14) we can derive that in the special case of a sinusoid with spatial orientation $\psi = 45°$ the corresponding spectral coefficients are found at the frequency coordinates

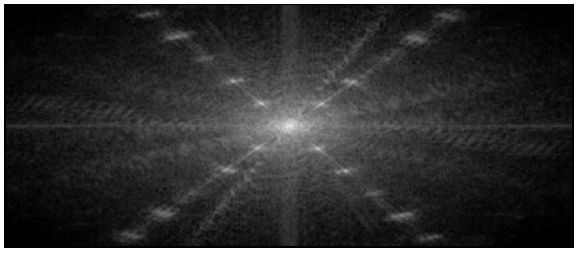
$$(m, n) = \pm(\lambda M, \lambda N) \quad \text{for } -\tfrac{1}{2} \le \lambda \le +\tfrac{1}{2}\,; \tag{14.15}$$

i. e., at the diagonals of the spectrum (see also Eqn. (14.12)). Unless the image (and thus the spectrum) is quadratic ($M = N$), the angle of orientation in the image and in the spectrum are not the same, though they coincide along the directions of the coordinate axes. This means that rotating an image by some angle α does turn the spectrum in the same direction but in general not by the same angle α!

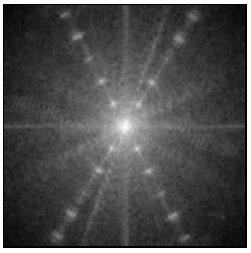
To obtain identical orientations and turning angles in both the image and the spectrum, it is sufficient to scale the spectrum to square size such that the spectral resolution is the same along both frequency axes (as shown in Fig. 14.7).

[1] ArcTan(x, y) in Eqn. (14.13) denotes the inverse tangent function $\tan^{-1}(y/x)$ (also see Appendix B.1.6).

(a)



(b)



(c)

Fig. 14.7
Correcting the geometry of the 2D spectrum. Original image (a) with dominant oriented patterns that show up as clear peaks in the corresponding spectrum (b). Because the image and the spectrum are not square ($M \neq N$), orientations in the image are not the same as in the actual spectrum (b). After the spectrum is scaled to square size (c), we can clearly observe that the cylinders of this (Harley-Davidson *V-Rod*) engine are really spaced at a 60° angle.

14.3.5 Effects of Periodicity

When interpreting the 2D DFT of images, one must always be aware of the fact that with any discrete Fourier transform, the signal function is implicitly assumed to be periodic in every dimension. Thus the transitions at the borders between the replicas of the image are also part of the signal, just like the interior of the image itself. If there is a large intensity difference between opposing borders of an image (e. g., between the upper and lower parts of a landscape image), then this causes strong transitions in the resulting periodic signal. Such steep discontinuities are of high bandwidth (i. e., the corresponding signal energy is spread over a wide range along the coordinate axes of the sampling grid; see Fig. 14.8). This broadband energy distribution along the main axes, which is often observed with real images, may lead to a suppression of other relevant signal components in the spectrum.

14.3.6 Windowing

One solution to this problem is to multiply the image function $g(u, v) = I(u, v)$ by a suitable windowing function $w(u, v)$,

$$\tilde{g}(u, v) = g(u, v) \cdot w(u, v),$$

for $0 \leq u < M$, $0 \leq v < N$, prior to computing the DFT. The windowing function $w(u, v)$ should drop off continuously toward the image borders such that the transitions between image replicas are effectively eliminated. But multiplying the image with $w(u, v)$ has additional effects

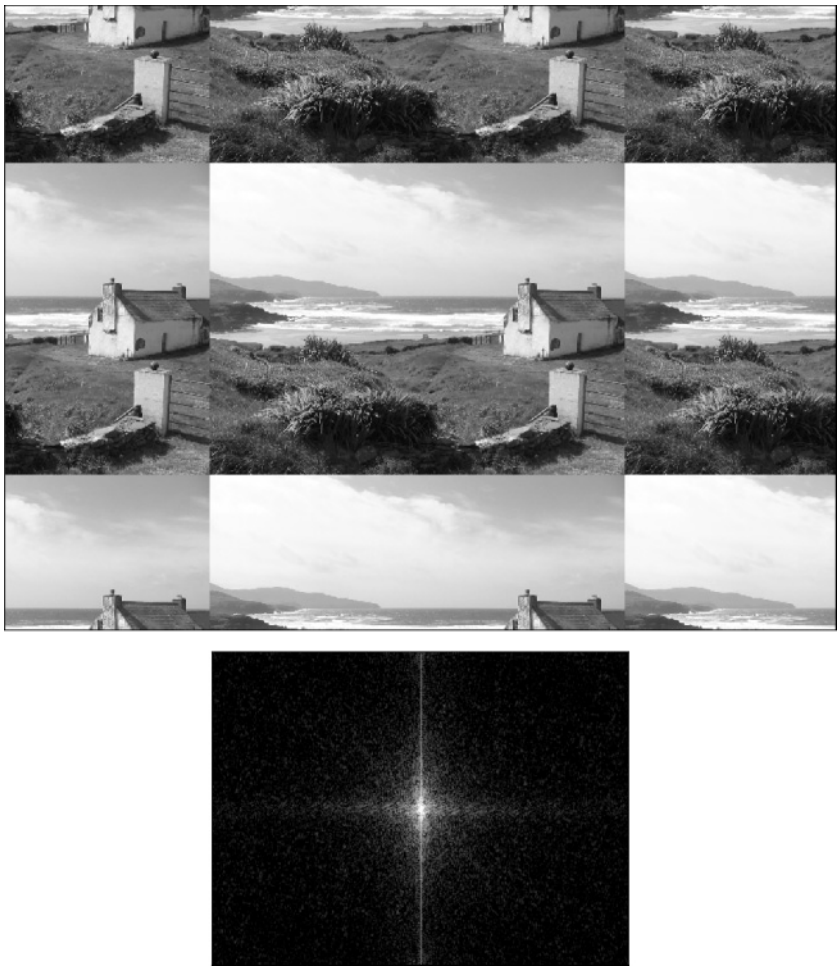


Fig. 14.8
Effects of periodicity in the 2D spectrum. The discrete Fourier transform is computed under the implicit assumption that the image signal is periodic along both dimensions (top). Large differences in intensity at opposite image borders—here most notably in the vertical direction—lead to broad-band signal components that in this case appear as a bright line along the spectrum's vertical axis (bottom).

upon the spectrum. As we already know (from Eqn. (13.27)), a *multiplication* of two functions in signal space corresponds to a convolution of the corresponding spectra in frequency space:

$$\tilde{G}(m,n) \leftarrow G(m,n) * W(m,n).$$

To apply the least possible damage to the Fourier transform of the image, the ideal spectrum of $w(u,v)$ would be the impulse function $\delta(m,n)$. Unfortunately, this again corresponds to the constant windowing function $w(u,v) = 1$ with no windowing effect at all. In general, we can say that a broader spectrum of the windowing function $w(u,v)$ smoothes the resulting spectrum more strongly and individual frequency components are harder to isolate.

Taking a picture is equivalent to cutting out a finite (usually rectangular) region from an infinite image plane, which can be simply modeled as a multiplication with a rectangular pulse function of width M and

height N. So, in this case, the spectrum of the original intensity function is convolved with the spectrum of the rectangular pulse (box). The problem is that the spectrum of the rectangular box (see Fig. 14.9 (a)) is of extremely high bandwidth and thus far off the ideal narrow impulse function.

These two examples demonstrate a dilemma: windowing functions should for one be as wide as possible to include a maximum part of the original image, and they should also drop off to zero toward the image borders but then again not too steeply to maintain a narrow windowing spectrum.

14.3.7 Windowing Functions

Suitable windowing functions should therefore exhibit soft transitions, and many variants have been proposed and analyzed both theoretically and for practical use (see, e. g., [14, Sec. 9.3], [80, Ch. 10]). Table 14.1 lists the definitions of several popular windowing functions; the corresponding 2D (logarithmic) power spectra are displayed in Figs. 14.9 and 14.10.

The spectrum of the *rectangular pulse* function (Fig. 14.9 (a)), which assigns identical weights to all image elements, exhibits a relatively narrow peak at the center, which promises little smoothing in the resulting windowed spectrum. Nevertheless, the spectral energy drops off quite slowly toward the higher frequencies, thus in all creating a rather wide spectrum. Not surprisingly, the behavior of the *elliptical* windowing function in Fig. 14.9 (b) is quite similar. The *Gaussian* window in Fig. 14.9 (c) demonstrates how the off-center spectral energy can be significantly suppressed by narrowing the windowing function, however, at the cost of a much wider peak at the center. In fact, none of the functions in Fig. 14.9 make a good windowing function.

Obviously, the choice of a suitable windowing function is a delicate compromise since even apparently similar functions may exhibit largely different behaviors in the frequency spectrum. For example, good overall results can be obtained with the *Hanning* window (Fig. 14.10 (c)) or the *Parzen* window (Fig. 14.10 (d)), which are both easy to compute and frequently used in practice.

Figure 14.11 illustrates the effects of selected windowing functions upon the spectrum of an intensity image. As can be seen clearly, narrowing the windowing function leads to a suppression of the artifacts caused by the signal's implicit periodicity. At the same time, however, it also reduces the resolution of the spectrum; the spectrum becomes blurred, and individual peaks are widened.

Definitions:

$$r_u = \frac{u - M/2}{M/2} = \frac{2u}{M} - 1 \qquad r_v = \frac{v - N/2}{N/2} = \frac{2v}{N} - 1 \qquad r_{u,v} = \sqrt{r_u^2 + r_v^2}$$

Elliptical window:	$w(u,v) = \begin{cases} 1 & \text{for } 0 \le r_{u,v} \le 1 \\ 0 & \text{otherwise} \end{cases}$
Gaussian window:	$w(u,v) = e^{\left(\frac{-r_{u,v}^2}{2\sigma^2}\right)}, \quad \sigma = 0.3 \ldots 0.4$
Super-Gaussian window:	$w(u,v) = e^{\left(\frac{-r_{u,v}^n}{\kappa}\right)}, \quad n = 6,\ \kappa = 0.3 \ldots 0.4$
Cosine^2 window:	$w(u,v) = \begin{cases} \cos\left(\frac{\pi}{2} r_u\right) \cdot \cos\left(\frac{\pi}{2} r_v\right) & \text{for } 0 \le r_u, r_v \le 1 \\ 0 & \text{otherwise} \end{cases}$
Bartlett window:	$w(u,v) = \begin{cases} 1 - r_{u,v} & \text{for } 0 \le r_{u,v} \le 1 \\ 0 & \text{otherwise} \end{cases}$
Hanning window:	$w(u,v) = \begin{cases} 0.5 \cdot \cos(\pi r_{u,v} + 1) & \text{for } 0 \le r_{u,v} \le 1 \\ 0 & \text{otherwise} \end{cases}$
Parzen window:	$w(u,v) = \begin{cases} 1 - 6r_{u,v}^2 + 6r_{u,v}^3 & \text{for } 0 \le r_{u,v} < 0.5 \\ 2 \cdot (1 - r_{u,v})^3 & \text{for } 0.5 \le r_{u,v} < 1 \\ 0 & \text{otherwise} \end{cases}$

Table 14.1
2D windowing functions. The functions $w(u,v)$ have their maximum values at the image center, $w(M/2, N/2) = 1$. The values r_u, r_v, and $r_{u,v}$ used in the definitions are specified at the top of the table.

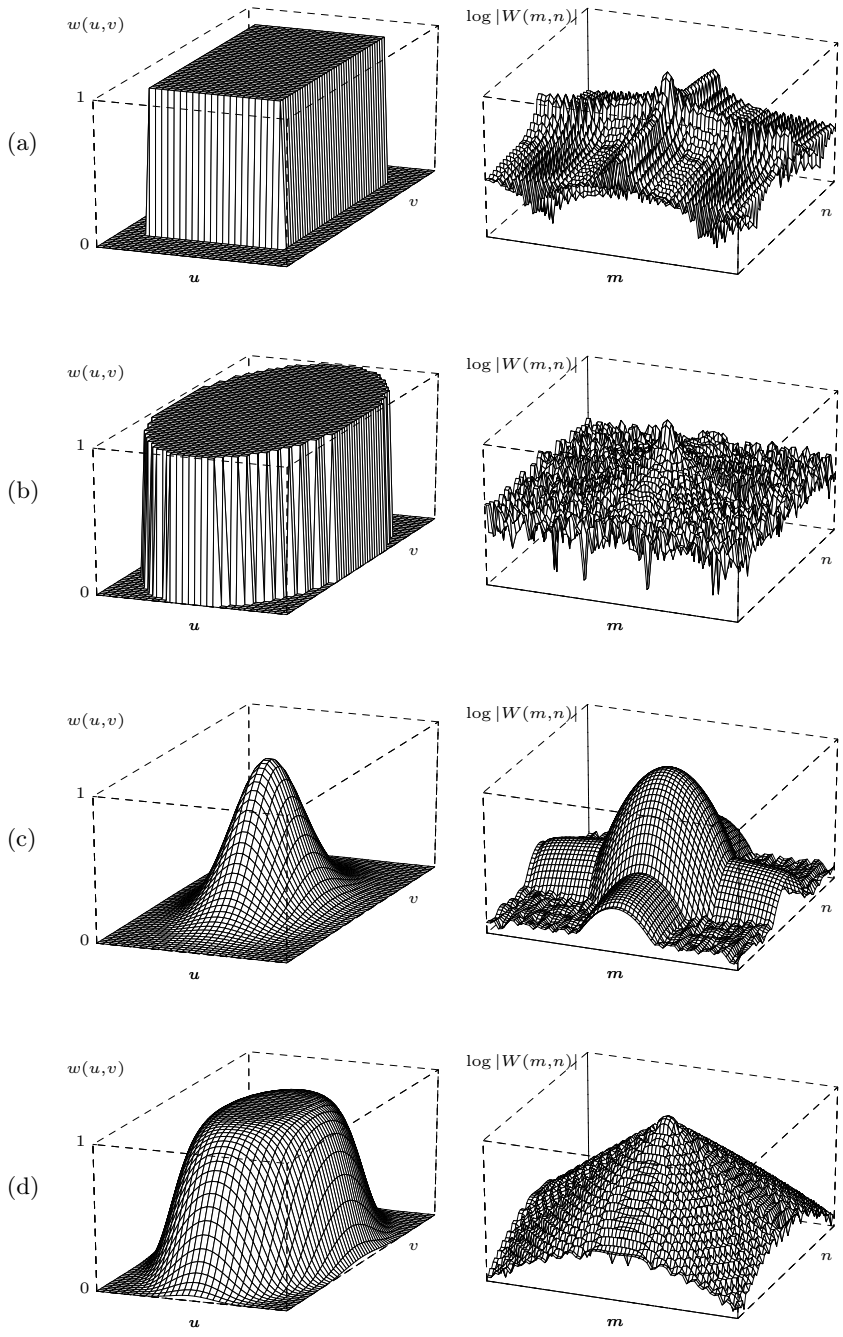


Fig. 14.9
Windowing functions and their logarithmic power spectra. Rectangular pulse (a), elliptical window (b), Gaussian window with $\sigma = 0.3$ (c), and super-Gaussian window of order $n = 6$ and $\kappa = 0.3$ (d). The windowing functions are deliberately of nonsquare size ($M : N = 1 : 2$).

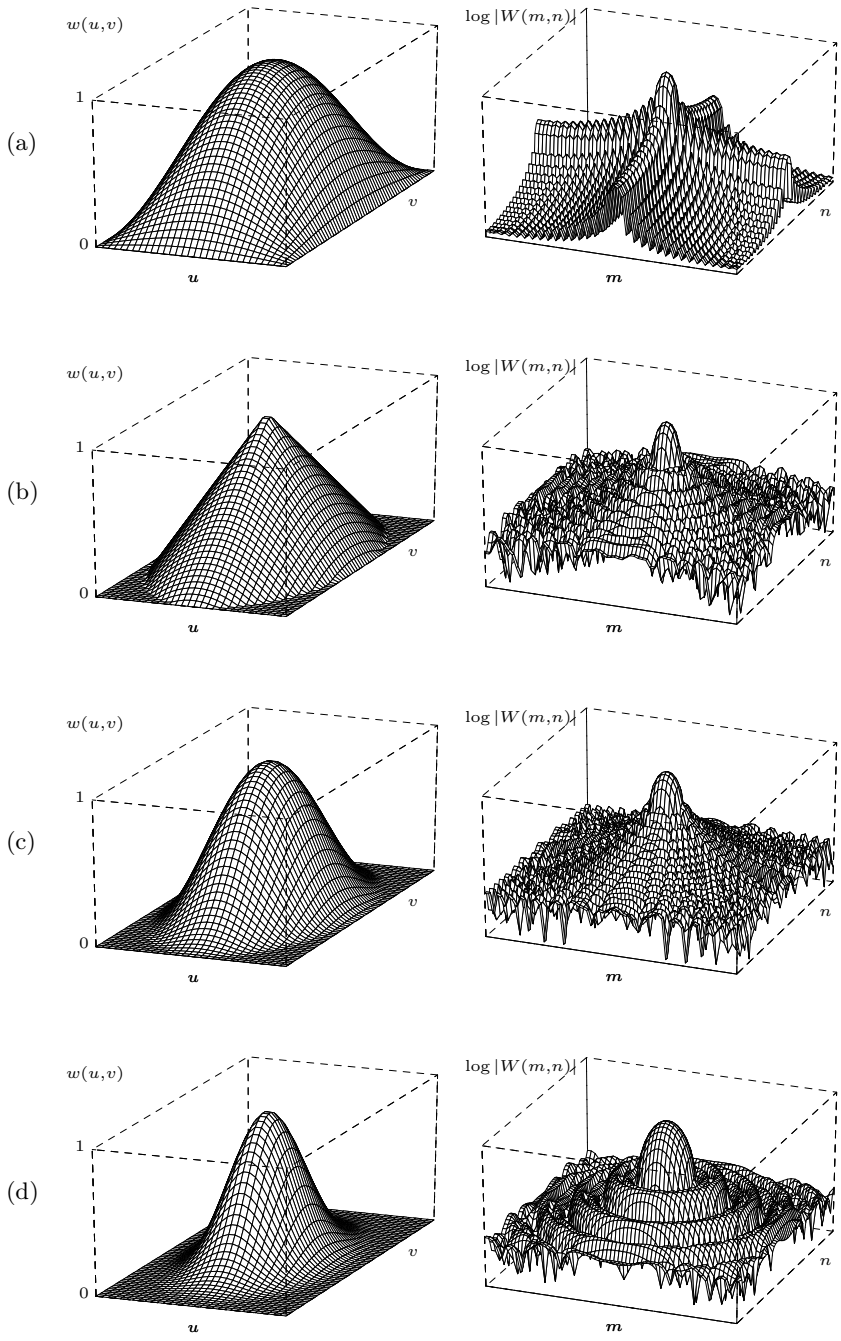


Fig. 14.10
Windowing functions and their logarithmic power spectra (*continued*). Cosine^2 window (a), Bartlett window (b), Hanning window (c), and Parzen window (d).

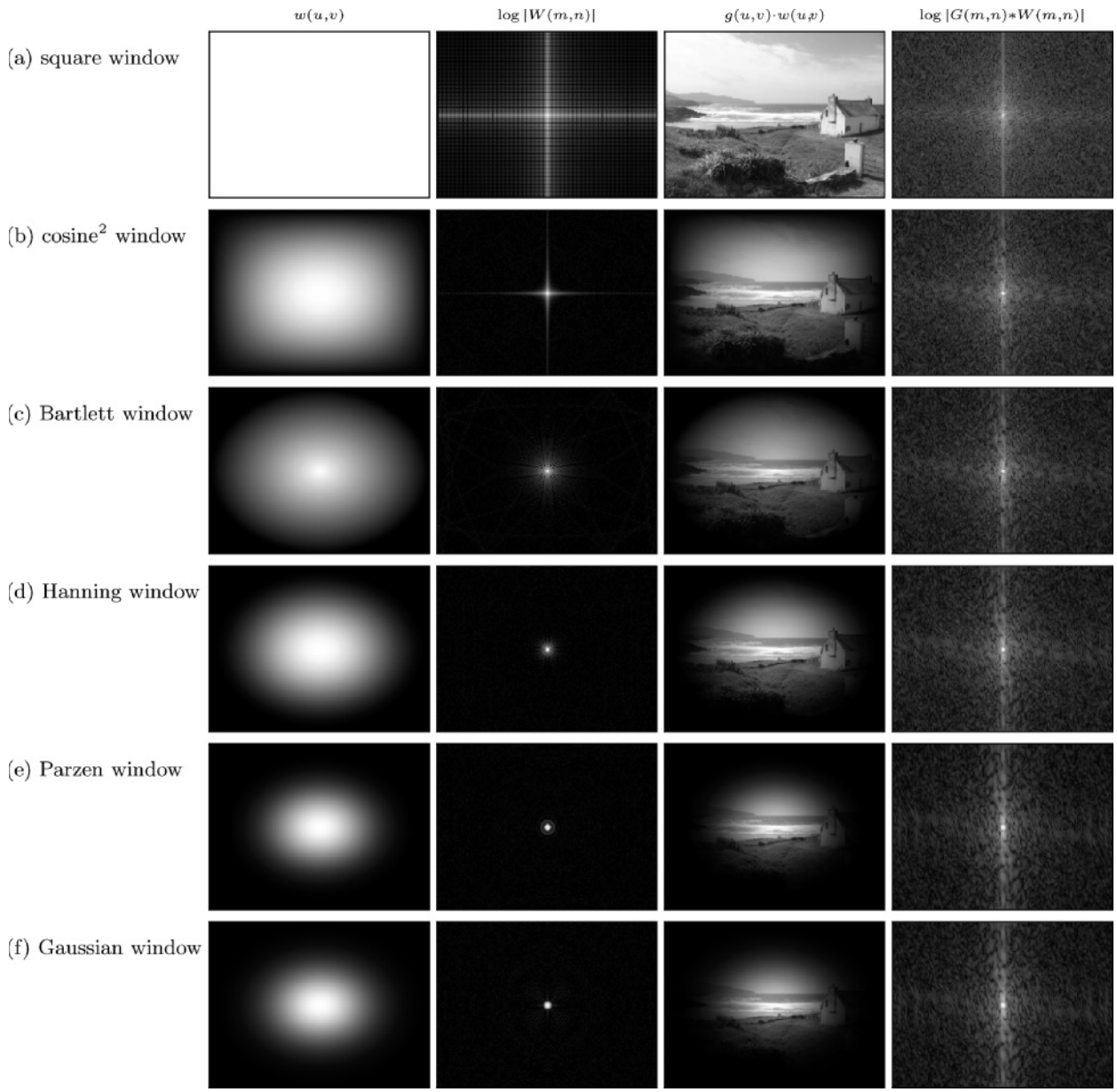


Fig. 14.11. Application of windowing functions on images. The plots show the windowing function $w(u,v)$, the logarithmic power spectrum of the windowing function $\log|W(m,n)|$, the windowed image $g(u,v) \cdot w(u,v)$, and the power spectrum of the windowed image $\log|G(m,n) * W(m,n)|$.

14.4 2D Fourier Transform Examples

The following examples demonstrate some basic properties of the two-dimensional DFT on real intensity images. All examples in Figs. 14.12–14.18 show a centered and squared spectrum with logarithmic intensity values (see Sec. 14.2).

Scaling

Figure 14.12 shows that scaling the image in signal space has the opposite effect in frequency space, analogous to the one-dimensional case (Fig. 13.4).

Periodic image patterns

The images in Fig. 14.13 contain repetitive periodic patterns at various orientations and scales. They appear as distinct peaks at the corresponding positions (see Eqn. (14.14)) in the spectrum.

Rotation

Figure 14.14 shows that rotating the image by some angle α rotates the spectrum in the same direction and—if the image is square—by the same angle.

Oriented, elongated structures

Pictures of artificial objects often exhibit regular patterns or elongated structures that appear dominantly in the spectrum. The images in Fig. 14.15 show several elongated structures that show up in the spectrum as bright streaks oriented perpendicularly to the main direction of the image patterns.

Natural images

Straight and regular structures are usually less dominant in images of natural objects and scenes, and thus the visual effects in the spectrum are not as obvious as with artificial objects. Some examples of this class of images are shown in Figs. 14.16 and 14.17.

Print pattern

The regular patterns generated by the common raster print techniques (Fig. 14.18) are typical examples for periodic multidirectional structures, which stand out clearly in the corresponding Fourier spectrum.

14.5 Applications of the DFT

The Fourier transform and the DFT in particular are important tools in many engineering disciplines. In digital signal and image processing, the DFT (and the FFT) is an indispensable "workhorse" with many applications in image analysis, filtering, and image reconstruction, just to name a few.

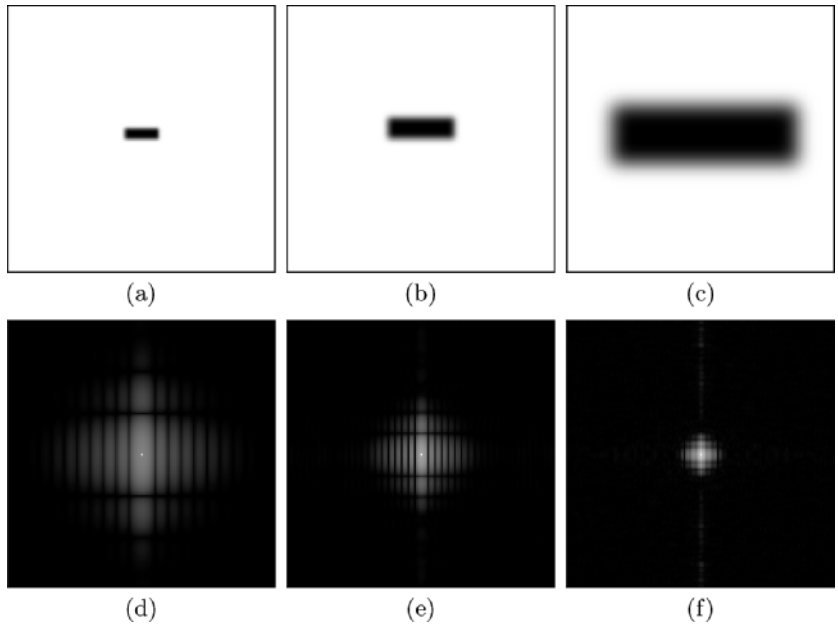


Fig. 14.12
DFT—image scaling. The rectangular pulse in the image function (a–c) creates a strongly oscillating power spectrum (d–f), as in the one-dimensional case. Stretching the image causes the spectrum to contract and vice versa.

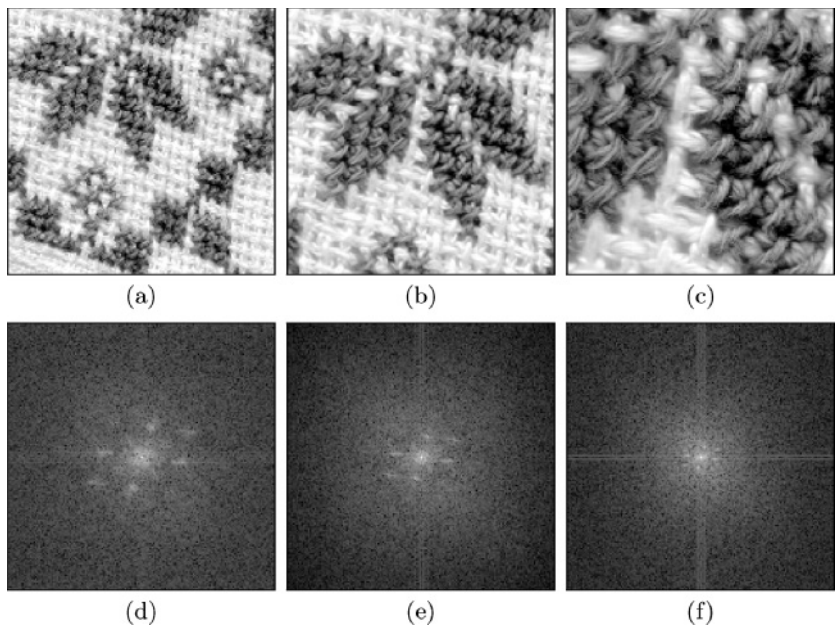


Fig. 14.13
DFT—oriented, repetitive patterns. The image function (a–c) contains patterns with three dominant orientations, which appear as pairs of corresponding frequency spots in the spectrum (c–f). Enlarging the image causes the spectrum to contract.

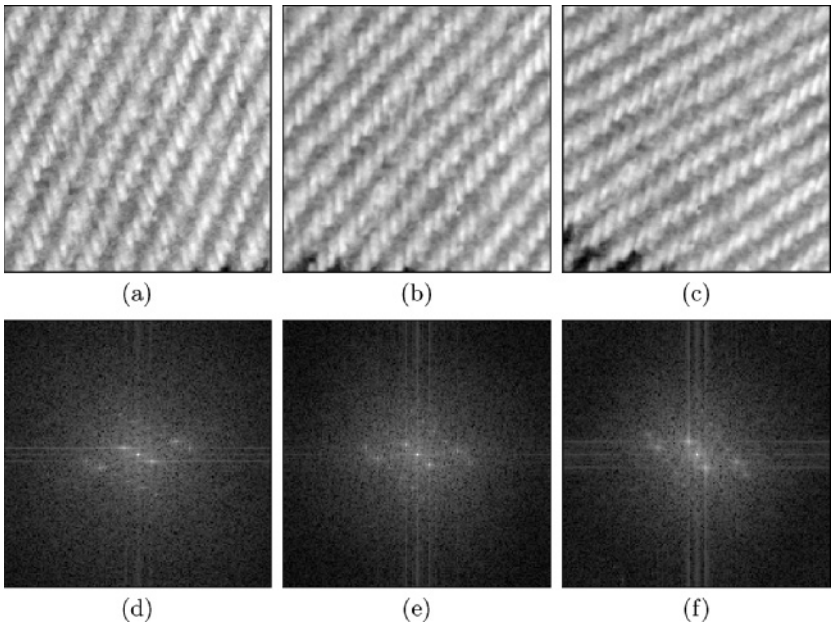


Fig. 14.14
DFT —image rotation. The original image (a) is rotated by 15° (b) and 30° (c). The corresponding (squared) spectrum turns in the same direction and by exactly the same amount (d–f).

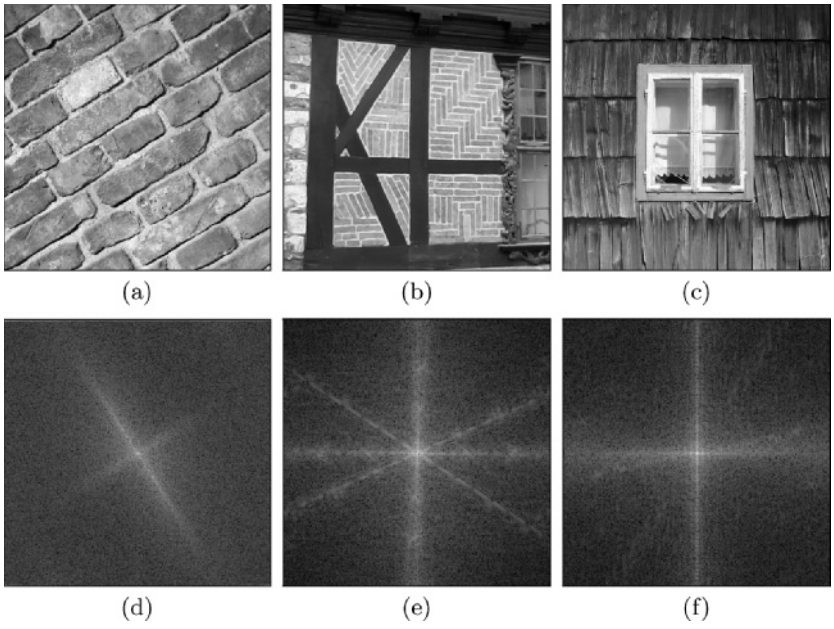


Fig. 14.15
DFT—superposition of image patterns. Strong, oriented subpatterns (a–c) are easy to identify in the corresponding spectrum (d–f). Notice the broadband effects caused by straight structures, such as the dark beam on the wall in (b, e).

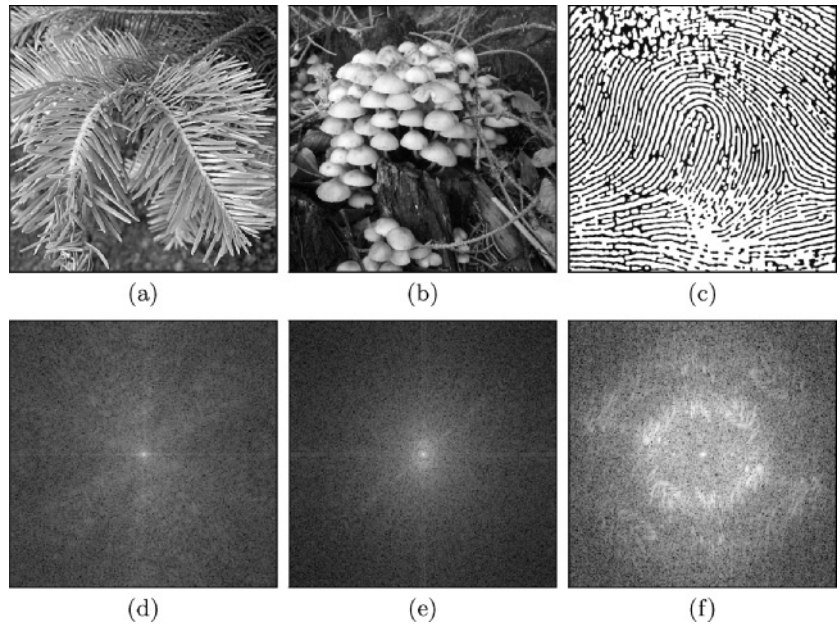


Fig. 14.16
DFT—natural image patterns. Examples of repetitive structures in natural images (a–c) that are also visible in the corresponding spectrum (d–f).

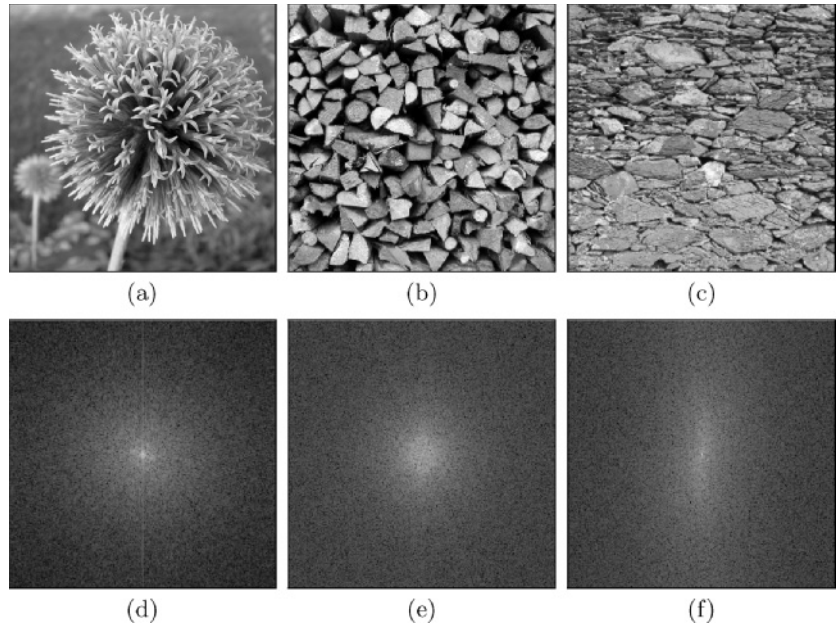
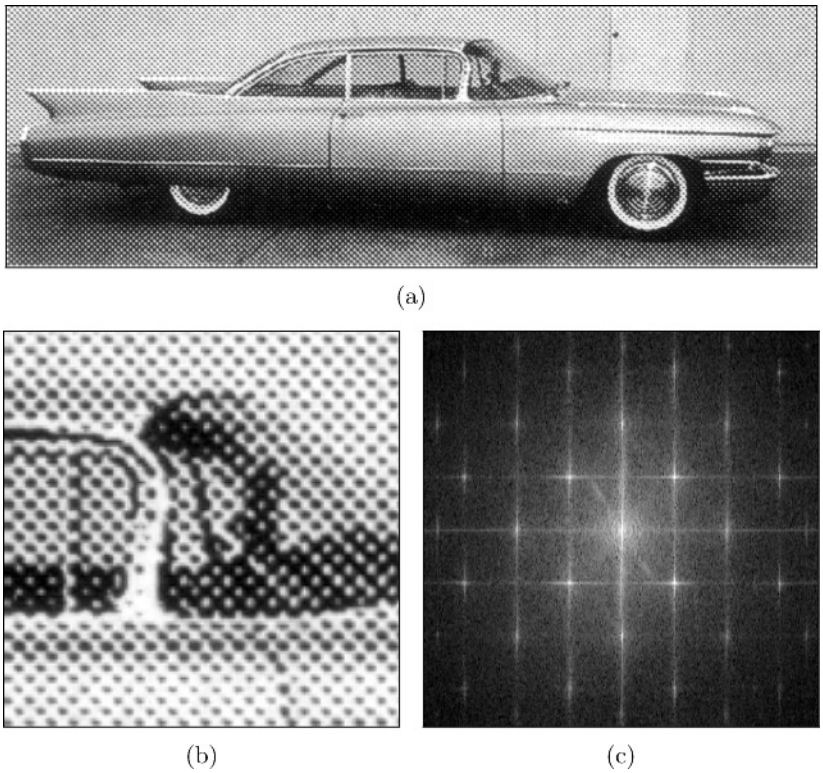


Fig. 14.17
DFT—natural image patterns with no dominant orientation. The repetitive patterns contained in these images (a–c) have no common orientation or sufficiently regular spacing to stand out locally in the corresponding Fourier spectra (d–f).



(a) (b) (c)

Fig. 14.18
DFT of a print pattern. The regular diagonally oriented raster pattern (a, b) is clearly visible in the corresponding power spectrum (c). It is possible (at least in principle) to remove such patterns by erasing these peaks in the Fourier spectrum and reconstructing the smoothed image from the modified spectrum using the inverse DFT.

14.5.1 Linear Filter Operations in Frequency Space

Performing linear filter operations in frequency space is an interesting option because it provides an efficient way to apply filters of large spatial extent. The approach is based on the *convolution property* of the Fourier transform (see Sec. 13.1.6), which states that a linear convolution in image space corresponds to a pointwise multiplication in frequency space. Thus the linear convolution $g * h \rightarrow g'$ between an image $g(u,v)$ and a filter matrix $h(u,v)$ can be accomplished by the following steps:

$$\begin{array}{rccccc} \text{Image space:} & g(u,v) & * & h(u,v) & = & g'(u,v) \\ & \downarrow & & \downarrow & & \uparrow \\ & \mathsf{DFT} & & \mathsf{DFT} & & \mathsf{DFT}^{-1} \\ & \downarrow & & \downarrow & & \uparrow \\ \text{Frequency space:} & G(m,n) & \cdot & H(m,n) & \longrightarrow & G'(m,n) \end{array} \tag{14.16}$$

First, the image g and the filter function h are transformed to frequency space using the two-dimensional DFT. The corresponding spectra G and H are then multiplied (pointwise), and the result G' is subsequently transformed back to image space using the inverse DFT, thus generating the filtered image g'.

The main advantage of this "detour" lies in its possible efficiency. A direct convolution for an image of size $M \times M$ and a filter matrix of size

$N \times N$ requires $\mathcal{O}(M^2N^2)$ operations. Thus, time complexity increases quadratically with filter size, which is usually no problem for small filters but may render some larger filters too costly to implement. For example, a filter of size 50×50 already requires about 2500 multiplications and additions for every image pixel. In comparison, the transformation from image to frequency space and back can be performed in $\mathcal{O}(M \log_2 M)$ using the FFT, and the pointwise multiplication in frequency space requires M^2 operations, independent of the filter size.

In addition, certain types of filters are easier to specify in frequency space than in image space; for example, an ideal low-pass filter, which can be described very compactly in frequency space. Further details on filter operations in frequency space can be found, for example, in [38, Sec. 4.4].

14.5.2 Linear Convolution versus Correlation

As described already in Sec. 6.3, a linear correlation is the same as a linear convolution with a mirrored filter function. Therefore, the correlation can be computed just like the convolution operation in the frequency domain by following the steps described in Eqn. (14.16). This could be advantageous for comparing images using correlation methods (see Sec. 17.1.1) because in this case the image and the "filter" matrix (i. e., the second image) are of similar size and thus usually too large to be processed in image space.

Some operations in ImageJ, such as *correlate*, *convolve*, or *deconvolve* (see below), are also implemented in the "Fourier domain" (FD) using the two-dimensional DFT. They can be invoked through the menu Process →FFT→FD Math.

14.5.3 Inverse Filters

Filtering in the frequency domain opens another interesting perspective: reversing the effects of a filter, at least under restricted conditions. In the following, we describe the basic idea only.

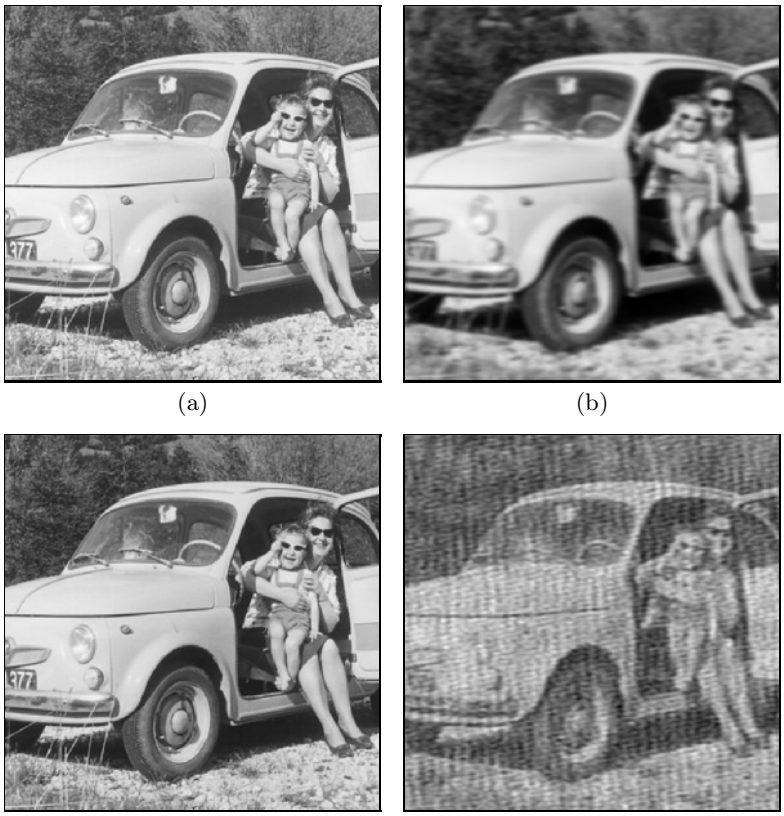
Assume we are given an image g_{blur} that has been generated from an original image g_{orig} by some linear filter, for example, motion blur induced by a moving camera. Under the assumption that this image modification can be modeled sufficiently by a linear filter function h_{blur}, we can state that

$$g_{\text{blur}} = g_{\text{orig}} * h_{\text{blur}}.$$

Knowing that in frequency space this corresponds to a multiplication of the corresponding spectra,

$$G_{\text{blur}} = G_{\text{orig}} \cdot H_{\text{blur}},$$

it should be possible to reconstruct the original (nonblurred) image by computing the inverse Fourier transform of the expression



(a) (b)

(c) (d)

Fig. 14.19
Removing motion blur by inverse filtering: original image (a); image blurred by horizontal motion (b); reconstruction using the exact (known) filter function (c); result of the inverse filter when the filter function deviates marginally from the real filter (d).

$$G_{\text{orig}}(m,n) = \frac{G_{\text{blur}}(m,n)}{H_{\text{blur}}(m,n)}\,.$$

Unfortunately, this "inverse filter" only works if the spectral coefficients H_{blur} are nonzero, because otherwise the resulting values are infinite. But even small values of H_{blur}, which are typical at high frequencies, lead to large coefficients in the reconstructed spectrum and, as a consequence, large amounts of image noise.

It is also important that the real filter function be accurately approximated because otherwise the reconstructed image may strongly deviate from the original. The example in Fig. 14.19 shows an image that has been blurred by smooth horizontal motion, whose effect can easily be modeled as a linear convolution. If the filter function that caused the blurring is known exactly, then the reconstruction of the original image can be accomplished without any problems (Fig. 14.19 (c)). However, as shown in Fig. 14.19 (d), large errors occur if the inverse filter deviates only marginally from the real filter, which quickly renders the method useless.

Beyond this simple idea (which is often referred to as "deconvolution"), better methods for inverse filtering exist, such as the *Wiener filter*

and related techniques (see, e. g., [38, Sec. 5.4], [60, Sec. 8.3], [59, Sec. 17.8], [20, Ch. 16]).

14.6 Exercises

Exercise 14.1. Implement the two-dimensional DFT using the one-dimensional DFT, as described in Sec. 14.1.2. Apply the 2D DFT to real intensity images of arbitrary size and display the results (by converting to ImageJ `float` images). Implement the inverse transform and verify that the back-transformed result is identical to the original image.

Exercise 14.2. Assume that the two-dimensional Fourier spectrum of an image with size 640×480 and a spatial resolution of 72 dpi shows a dominant peak at position $\pm(100, 100)$. Determine the orientation and effective frequency (in cycles per cm) of the corresponding image pattern.

Exercise 14.3. An image with size 800×600 contains a wavy intensity pattern with an effective period of 12 pixels, oriented at 30°. At which frequency coordinates will this pattern manifest itself in the discrete Fourier spectrum?

Exercise 14.4. Generalize Eqn. (14.10) and Eqns. (14.12)–(14.14) for the case where the sampling intervals are *not* identical along the x and y axes (i. e., for $\tau_x \neq \tau_y$).

Exercise 14.5. Implement the *elliptical* and the *super-Gauss* windowing functions (Table 14.1) as ImageJ plugins, and investigate the effects of these windows upon the resulting spectra. Also compare the results to the case where *no* windowing function is used.

15

The Discrete Cosine Transform (DCT)

The Fourier transform and the DFT are designed for processing complex-valued signals, and they always produce a complex-valued spectrum even in the case where the original signal was strictly real-valued. The reason is that neither the real nor the imaginary part of the Fourier spectrum alone is sufficient to represent (i. e., reconstruct) the signal completely. In other words, the corresponding cosine (for the real part) or sine functions (for the imaginary part) alone do not constitute a complete set of basis functions.

On the other hand, we know (see Eqn. (13.22)) that a real-valued signal has a symmetric Fourier spectrum, so only one half of the spectral coefficients need to be computed without losing any signal information.

There are several spectral transformations that have properties similar to the DFT but do not work with complex function values. The discrete cosine transform (DCT) is a well known example that is particularly interesting in our context because it is frequently used for image and video compression. The DCT uses only cosine functions of various wave numbers as basis functions and operates on real-valued signals and spectral coefficients. Similarly, there is also a discrete sine transform (DST) based on a system of sine functions [60].

15.1 One-Dimensional DCT

The discrete cosine transform is not, as one may falsely assume, only a "one-half" variant of the discrete Fourier transform. In the one-dimensional case, the *forward* cosine transform for a signal $g(u)$ of length M is defined as

$$G(m) = \sqrt{\tfrac{2}{M}} \sum_{u=0}^{M-1} g(u) \cdot c_m \cos\left(\pi \tfrac{m(2u+1)}{2M}\right) \tag{15.1}$$

for $0 \le m < M$, and the *inverse* transform is

$$g(u) = \sqrt{\tfrac{2}{M}} \sum_{m=0}^{M-1} G(m) \cdot c_m \cos\left(\pi \tfrac{m(2u+1)}{2M}\right) \tag{15.2}$$

for $0 \le u < M$, respectively, with

$$c_m = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } m = 0 \\ 1 & \text{otherwise.} \end{cases} \tag{15.3}$$

Note that the index variables (u, m) are used differently in the forward transform (Eqn. (15.1)) and the inverse transform (Eqn. (15.2)), so the two transforms are—in contrast to the DFT—*not* symmetric.

15.1.1 DCT Basis Functions

One may ask why it is possible that the DCT can work without any sine functions, while they are essential in the DFT. The trick is to divide all frequencies in half such that they are spaced more densely and thus the frequency resolution in the spectrum is doubled. Comparing the cosine parts of the DFT basis functions (Eqn. (13.48)) and those of the DCT (Eqn. (15.1)),

$$\begin{aligned} &\text{DFT: } \boldsymbol{C}_m^M(u) = \cos\bigl(2\pi \tfrac{mu}{M}\bigr), \\ &\text{DCT: } \boldsymbol{D}_m^M(u) = \cos\bigl(\pi \tfrac{m(2u+1)}{2M}\bigr) = \cos\bigl(2\pi \tfrac{m(u+0.5)}{2M}\bigr), \end{aligned} \tag{15.4}$$

one can clearly see that the period of the DCT basis functions $(2M/m)$ is double the period of DFT functions (M/m) and the DCT functions are also phase-shifted by 0.5 units.

Figure 15.1 shows the DCT basis functions $\boldsymbol{D}_m^M(u)$ for the signal length $M = 8$ and wave numbers $m = 0 \ldots 7$. For example, the basis function $\boldsymbol{D}_7^8(u)$ for wave number $m = 7$ performs seven full cycles over a length of $2M = 16$ units and therefore has the radial frequency $\omega = m/2 = 3.5$.

15.1.2 Implementing the One-Dimensional DCT

Since the DCT does not create any complex values and the forward and inverse transforms (Eqns. (15.1) and (15.2)) are almost identical, the whole procedure is fairly easy to implement in Java, as shown in Prog. 15.1. The only minor thing to note is that the factor c_m in Eqn. (15.1) is independent of the iteration variable u and can thus be computed outside the inner summation loop (Prog. 15.1, line 7).

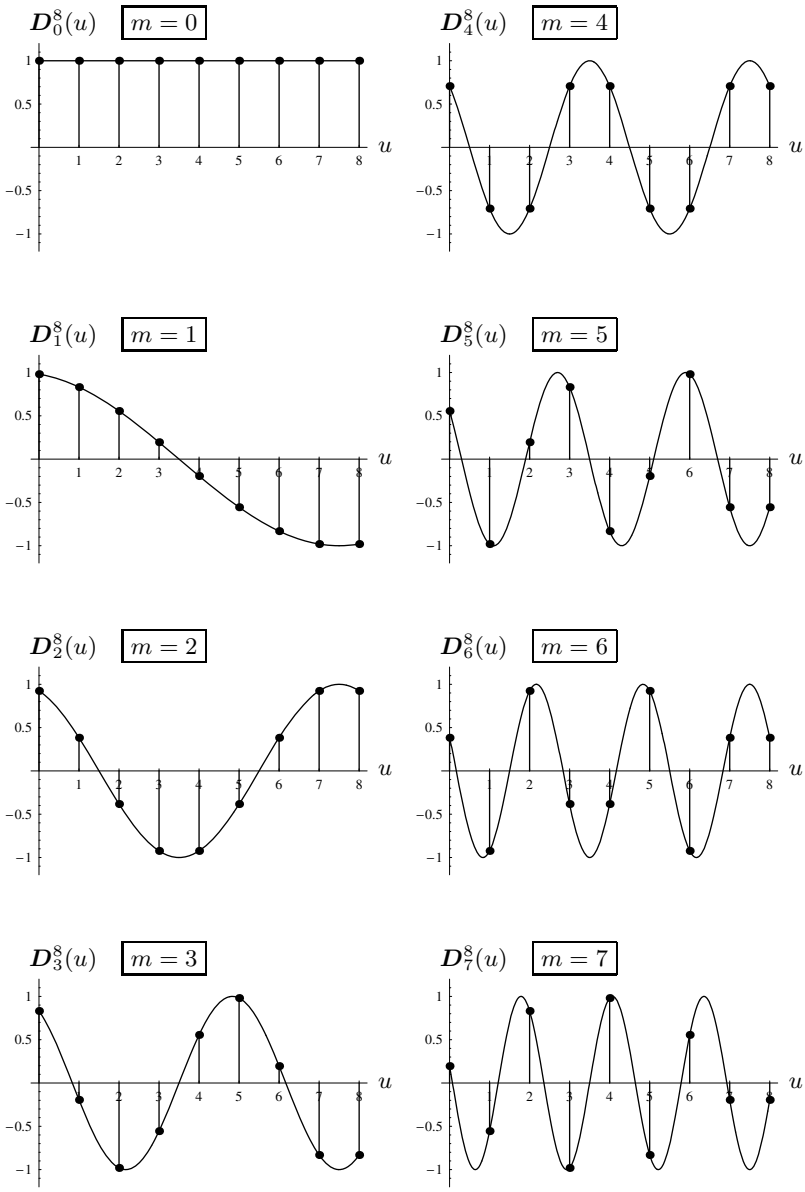


Fig. 15.1
DCT basis functions $\boldsymbol{D}_0^M(u) \ldots \boldsymbol{D}_7^M(u)$ for $M = 8$. Each plot shows both the discrete function (round dots) and the corresponding continuous function. Compared with the basis functions of the DFT (Figs. 13.11 and 13.12), all frequencies are divided in half and the DCT basis functions are phase-shifted by 0.5 units. All DCT basis functions are thus periodic over the length $2M = 16$ (as compared with M for the DFT).

Of course, much more efficient ("fast") DCT algorithms exist. Moreover, the DCT can also be computed in $\mathcal{O}(M \log_2 M)$ time using the FFT [60, p. 152].[1] The DCT is often used for image compression, in particular for JPEG compression, where the size of the transformed subimages is fixed at 8×8 and the processing can be highly optimized.

[1] See Appendix A (p. 454) for a brief explanation of the $\mathcal{O}()$ notation.

Program 15.1
One-dimensional DCT (Java implementation). The method `DCT()` computes the forward transform for a real-valued signal vector `g` of arbitrary length according to the definition in Eqn. (15.1). The method returns the DCT spectrum as a real-valued vector of the same length as the input vector `g`. The inverse transform `iDCT()` computes the inverse DCT for the real-valued cosine spectrum `G`.

```
1  double[] DCT (double[] g) { // forward DCT of signal g(u)
2    int M = g.length;
3    double s = Math.sqrt(2.0 / M); //common scale factor
4    double[] G = new double[M];
5    for (int m = 0; m < M; m++) {
6      double cm = 1.0;
7      if (m == 0) cm = 1.0 / Math.sqrt(2);
8      double sum = 0;
9      for (int u = 0; u < M; u++) {
10       double Phi = (Math.PI * m * (2 * u + 1)) / (2.0 * M);
11       sum += g[u] * cm * Math.cos(Phi);
12     }
13     G[m] = s * sum;
14   }
15   return G;
16 }
```

```
17 double[] iDCT (double[] G) { // inverse DCT of spectrum G(m)
18   int M = G.length;
19   double s = Math.sqrt(2.0 / M); //common scale factor
20   double[] g = new double[M];
21   for (int u = 0; u < M; u++) {
22     double sum = 0;
23     for (int m = 0; m < M; m++) {
24       double cm = 1.0;
25       if (m == 0) cm = 1.0 / Math.sqrt(2);
26       double Phi = (Math.PI * (2 * u + 1) * m) / (2.0 * M);
27       double cosPhi = Math.cos(Phi);
28       sum += cm * G[m] * cosPhi;
29     }
30     g[u] = s * sum;
31   }
32   return g;
33 }
```

15.2 Two-Dimensional DCT

The two-dimensional form of the DCT follows immediately from the the one-dimensional definition (Eqns. (15.1) and (15.2)), resulting in the 2D forward transform

$$\begin{aligned} G(m,n) &= \frac{2}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u,v) \cdot c_m \cos\Big(\tfrac{\pi(2u+1)m}{2M}\Big) \cdot c_n \cos\Big(\tfrac{\pi(2v+1)n}{2N}\Big) \\ &= \frac{2c_m c_n}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u,v) \cdot \boldsymbol{D}_m^M(u) \cdot \boldsymbol{D}_n^N(v) \end{aligned} \tag{15.5}$$

for $0 \le m < M$, $0 \le n < N$, and the inverse transform

$$g(u,v) = \frac{2}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(m,n) \cdot c_m \cos\Big(\tfrac{\pi(2u+1)m}{2M}\Big) \cdot c_n \cos\Big(\tfrac{\pi(2v+1)n}{2N}\Big)$$
$$= \frac{2}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(m,n) \cdot c_m \boldsymbol{D}_m^M(u) \cdot c_n \boldsymbol{D}_n^N(v) \qquad (15.6)$$

for $0 \le u < M$, $0 \le v < N$. The coefficients c_m and c_n in Eqns. (15.5) and (15.6) are the same as in the one-dimensional case (Eqn. (15.3)). Notice that in the forward transform (and only there!) the factors c_m, c_n are independent of the iteration variables u, v and can thus be placed outside the summation (as shown in Eqn. (15.5)).

15.2.1 Separability

Similar to the DFT (see Eqn. (14.7)), the two-dimensional DCT can also be separated into two successive one-dimensional transforms. To make this fact clear, the forward DCT can be expressed in the following way:

$$G(m,n) = \sqrt{\tfrac{2}{N}} \sum_{v=0}^{N-1} \Bigg[\underbrace{\sqrt{\tfrac{2}{M}} \sum_{u=0}^{M-1} g(u,v) \cdot c_m \boldsymbol{D}_m^M(u)}_{\text{one-dimensional } \mathsf{DCT}[g(\cdot,v)]} \Bigg] \cdot c_n \boldsymbol{D}_n^N(v)\,. \qquad (15.7)$$

The inner expression in Eqn. (15.7) corresponds to a one-dimensional DCT of the vth line $g(\cdot, v)$ of the 2D signal function. Thus, as with the 2D DFT, one can first apply a one-dimensional DCT to every line of an image and subsequently a DCT to each column. Of course ,one could equally follow the reverse order by doing a DCT on the columns first and then on the rows.

15.2.2 Examples

Figure 15.2 shows several examples of the DCT in comparison with the results of the DFT. Since the DCT spectrum is (in contrast to the DFT spectrum) not symmetric, it does not get centered but is displayed in its original form with its origin at the upper left corner. The intensity corresponds to the logarithm of the absolute value in the case of the (real-valued) DCT spectrum. Similarly, the usual logarithmic power spectrum is shown for the DFT. Notice that the DCT is not simply a section of the DFT but obviously combines structures from adjacent quadrants of the Fourier spectrum.

15.3 Other Spectral Transforms

Apparently, the Fourier transform is not the only way to represent a given signal in frequency space; in fact, numerous similar transforms exist.

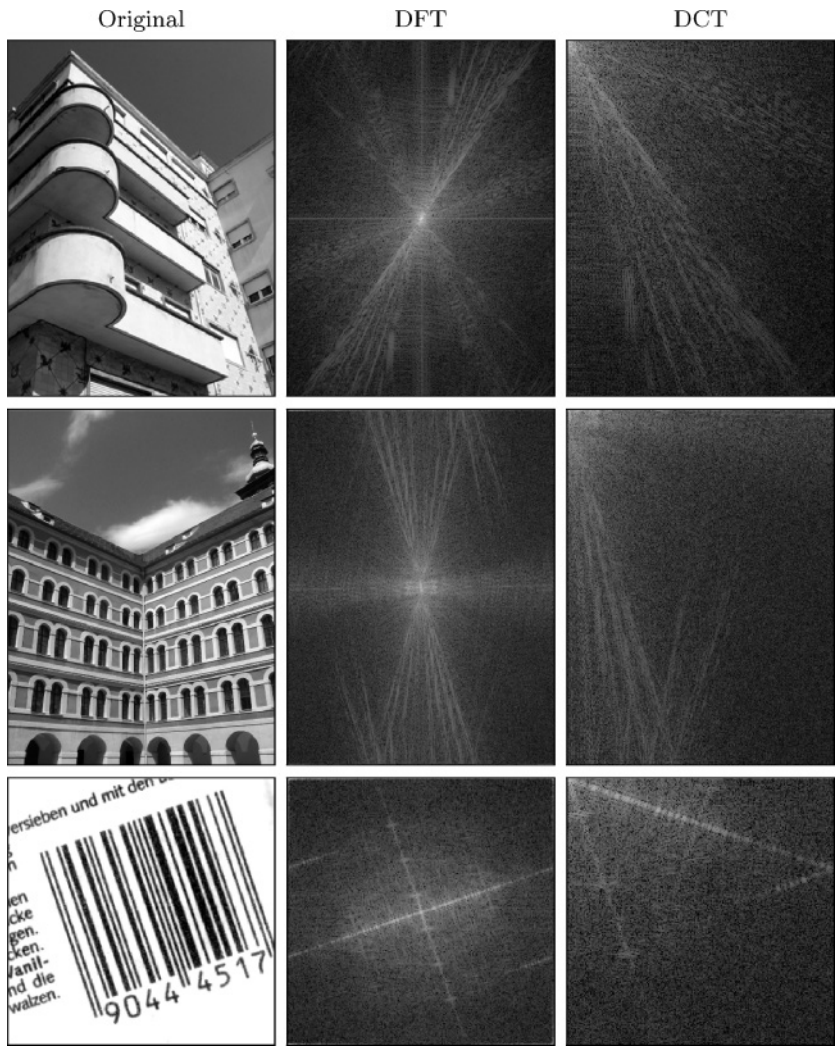


Fig. 15.2
Comparing the results of the two-dimensional DFT and DCT. Apparently both transforms show the frequency effects of image structures in a similar fashion. In the real-valued DCT spectrum (right), all coefficients are contained in a single quadrant and the frequency resolution is doubled compared with the DFT power spectrum (center). The DFT spectrum is centered as usual, while the origin of the DCT spectrum is located at the upper left corner. Both spectral plots show logarithmic intensity values.

Some of these, such as the discrete cosine transform, also use sinusoidal basis functions, while others, such as the *Hadamard* transform (also known as the *Walsh* transform), build on binary 0/1-functions [20, 59].

All of these transforms are of *global* nature; i. e., the value of any spectral coefficient is equally influenced by all signal values, independent of the spatial position in the signal. Thus a peak in the spectrum could be caused by a high-amplitude event of local extent as well as by a widespread, continuous wave of low amplitude. Global transforms are therefore of limited use for the purpose of detecting or analyzing local events because they are incapable of capturing the spatial position and extent of events in a signal.

A solution to this problem is to use a set of *local*, spatially limited basis functions ("wavelets") in place of the global, spatially fixed basis functions. The corresponding *wavelet transform*, of which several versions have been proposed, allows the simultaneous localization of repetitive signal components in both signal space *and* frequency space [68].

15.4 Exercises

Exercise 15.1. Implement the two-dimensional DCT (Sec. 15.2) as an ImageJ plugin for images of arbitrary size.

Exercise 15.2. Implement an efficient ("hard-coded") Java method for computing the one-dimensional DCT of length $M = 8$ that operates without iterations (loops) and contains all necessary coefficients as precomputed constants.

Exercise 15.3. Verify by numerical computation that the DCT basis functions $\boldsymbol{D}_m^M(u)$ for $0 \leq m, u < M$ (Eqn. (15.4)) are pairwise orthogonal; i. e., the inner product of the vectors $\boldsymbol{D}_m^M \cdot \boldsymbol{D}_n^M$ is zero for any pair $m \neq n$.

16

Geometric Operations

Common to the filters and point operations described so far is the fact that they may change the intensity function of an image but the position of each pixel and thus the geometry of the image remains the same. The purpose of geometric operations, which are discussed in this chapter, is to deform an image by altering its geometry. Typical examples are shifting, rotating, or scaling images, as shown in Fig. 16.1. Geometric operations are frequently needed in practical applications, for example, in virtually any modern graphical computer interface. Today we take for granted that windows and images in graphic or video applications can be zoomed continuously to arbitrary size. Geometric image operations are also important in computer graphics where textures, which are usually raster images, are deformed to be mapped onto the corresponding 3D surfaces, possibly in real time.

Of course, geometric operations are not as simple as their commonality may suggest. While it is obvious, for example, that an image could be enlarged by some integral factor n simply by replicating each pixel $n \times n$ times, the results would probably not be appealing, and it also gives us no immediate idea how to handle nonintegral scale factors, rotating images, or other image deformations. In general, geometric operations that achieve high-quality results are not trivial to implement and are also computationally demanding, even on today's fast computers.

In principle, a geometric operation transforms a given image I to a new image I' by modifying the *coordinates* of image pixels,

$$I(x, y) \rightarrow I'(x', y'); \tag{16.1}$$

i. e., the value of the image function I originally located at (x, y) moves to the position (x', y') in the new image I'.

To model this process, we first need a *mapping function*

$$T : \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

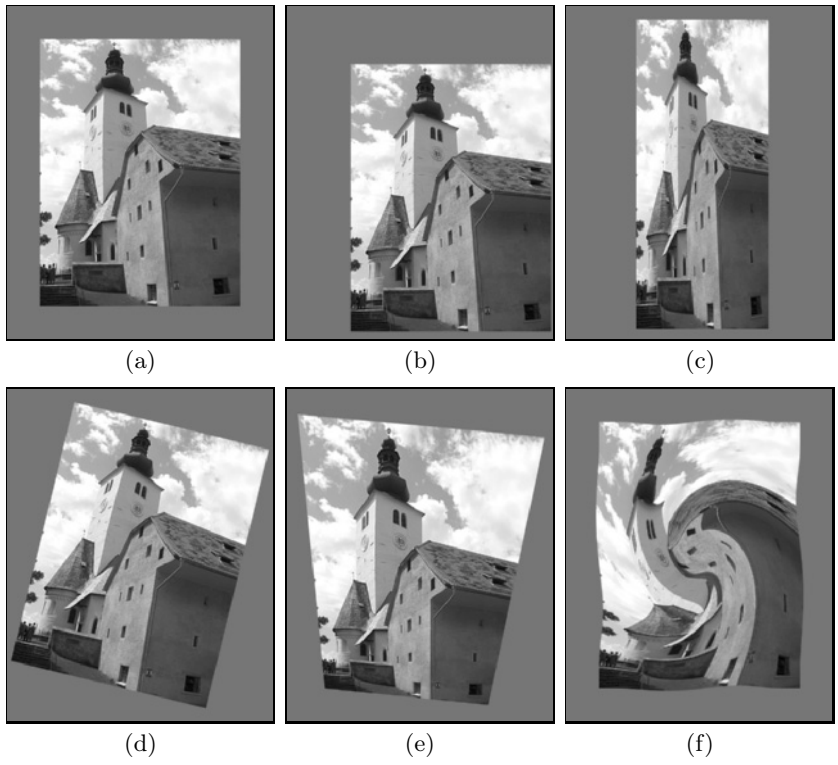


Fig. 16.1
Typical examples for geometric operations: original image (a), translation (b), scaling (contracting or stretching) in x and y directions (c), rotation about the center (d), projective transformation (e), and nonlinear distortion (f).

that specifies for each original 2D coordinate point $\boldsymbol{x} = (x, y)$ the corresponding target point $\boldsymbol{x}' = (x', y')$ in the new image I',

$$\boldsymbol{x}' = T(\boldsymbol{x}). \tag{16.2}$$

Notice that the coordinates (x, y) and (x', y') specify real-valued points in the continuous image plane $\mathbb{R} \times \mathbb{R}$. The main problem in transforming digital images is that the pixels $I(u, v)$ are defined not on a continuous plane but on a discrete raster $\mathbb{Z} \times \mathbb{Z}$. Obviously, a transformed coordinate $(u', v') = T(u, v)$ produced by the mapping function $T()$ will, in general, no longer fall onto a discrete raster point. The solution to this problem is to compute intermediate pixel values for the transformed image by a process called *interpolation*, which is the second essential element in any geometric operation. Let us first take a closer look at the continuous coordinate transform $T()$ and subsequently attend to the issue of interpolation in Sec. 16.3.

16.1 2D Mapping Function

The mapping function $T()$ in Eqn. (16.2) is an arbitrary continuous function that for reasons of simplicity is often specified as two separate functions,

$$x' = T_x(x, y) \qquad \text{and} \qquad y' = T_y(x, y), \tag{16.3}$$

for the x and y components.

16.1.1 Simple Mappings

The simple mapping functions include translation, scaling, shearing, and rotation, defined as follows:

Translation (shift) by a vector (d_x, d_y):

$$\begin{matrix} T_x : x' = x + d_x \\ T_y : y' = y + d_y \end{matrix} \quad \text{or} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix}. \tag{16.4}$$

Scaling (contracting or stretching) along the x or y axis by the factor s_x or s_y, respectively:

$$\begin{matrix} T_x : x' = s_x \cdot x \\ T_y : y' = s_y \cdot y \end{matrix} \quad \text{or} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \tag{16.5}$$

Shearing along the x and y axis by the factor b_x and b_y, respectively (for shearing in only one direction, the other factor is set to zero):

$$\begin{matrix} T_x : x' = x + b_x \cdot y \\ T_y : y' = y + b_y \cdot x \end{matrix} \quad \text{or} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & b_x \\ b_y & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \tag{16.6}$$

Rotation by an angle α (the coordinate origin being the center of rotation):

$$\begin{matrix} T_x : x' = x \cdot \cos\alpha - y \cdot \sin\alpha \\ T_y : y' = x \cdot \sin\alpha + y \cdot \cos\alpha \end{matrix} \quad \text{or} \tag{16.7}$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \tag{16.8}$$

16.1.2 Homogeneous Coordinates

The operations listed in Eqns. (16.4)–(16.8) constitute the important class of "affine" mapping functions or *affine transformations* (see also Sec. 16.1.3). To simplify the concatenation of mappings, it is advantageous to specify all operations in the form of vector-matrix multiplications, as in Eqns. (16.5)–(16.8). Notice that the translation Eqn. (16.4), which is a vector addition, cannot be formulated as vector-matrix multiplication.

Fortunately, this difficulty can be elegantly resolved with *homogeneous coordinates* (see, e. g., [31, p. 204]). To turn regular coordinates into homogeneous coordinates, each vector is extended by one additional element (h); i. e., in the two-dimensional case,

$$\boldsymbol{x} = \begin{pmatrix} x \\ y \end{pmatrix} \quad \text{converts to} \quad \hat{\boldsymbol{x}} = \begin{pmatrix} \hat{x} \\ \hat{y} \\ h \end{pmatrix} = \begin{pmatrix} h\,x \\ h\,y \\ h \end{pmatrix}. \tag{16.9}$$

Thus every ordinary 2D (Cartesian) coordinate pair $\boldsymbol{x} = (x, y)^T$ is replaced by a three-dimensional homogeneous coordinate vector $\hat{\boldsymbol{x}} = (\hat{x}, \hat{y}, h)^T$ with arbitrary $h \neq 0$. If the last component (h) of the homogeneous vector $\hat{\boldsymbol{x}}$ is nonzero, the components of the corresponding Cartesian vector $(x, y)^T$ are found to be

$$x = \frac{\hat{x}}{h} \qquad \text{and} \qquad y = \frac{\hat{y}}{h}. \tag{16.10}$$

Since the value of h is arbitrary, there exist infinitely many homogeneous vectors that are equivalent to a particular ordinary vector. In particular, two homogeneous coordinates $\hat{\boldsymbol{x}}_1$, $\hat{\boldsymbol{x}}_2$ represent the same Cartesian point $\boldsymbol{x}$ if they are multiples of each other; i. e.,

$$\text{if} \quad \hat{\boldsymbol{x}}_1 = s \cdot \hat{\boldsymbol{x}}_2, \quad \text{then} \quad \boldsymbol{x}_1 = \boldsymbol{x}_2 = \boldsymbol{x} \tag{16.11}$$

for $s \neq 0$. For example, the homogeneous coordinates $\hat{\boldsymbol{x}}_1 = (3, 2, 1)^T$, $\hat{\boldsymbol{x}}_2 = (-6, -4, -2)^T$, and $\hat{\boldsymbol{x}}_3 = (30, 20, 10)^T$ are all equivalent to the Cartesian coordinate vector $\boldsymbol{x} = (3, 2)^T$.

16.1.3 Affine (Three-Point) Mapping

Using homogeneous coordinates, we can rewrite the 2D translation (Eqn. (16.4)) as

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x+d_x \\ y+d_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{16.12}$$

which had been our motive for introducing homogeneous coordinates in the first place. Consequently, we can now express any combination of 2D translation, scaling, and rotation as vector-matrix multiplication with homogeneous coordinates in the form $\hat{\boldsymbol{x}}' = \boldsymbol{A} \cdot \hat{\boldsymbol{x}}$ or

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \tag{16.13}$$

This 2D coordinate transformation is called an "affine mapping" with the six parameters $a_{11} \ldots a_{23}$, where a_{13}, a_{23} specify the translation (equivalent to d_x, d_y in Eqn. (16.4)) and a_{11}, a_{12}, a_{21}, a_{22} aggregate the scaling, shearing, and rotation terms (Eqns. (16.5)–(16.8)). An affine mapping transforms straight lines to straight lines, triangles to triangles, and rectangles to parallelograms, as illustrated in Fig. 16.2. The distance ratio between points on a straight line remains unchanged by this type of mapping function.

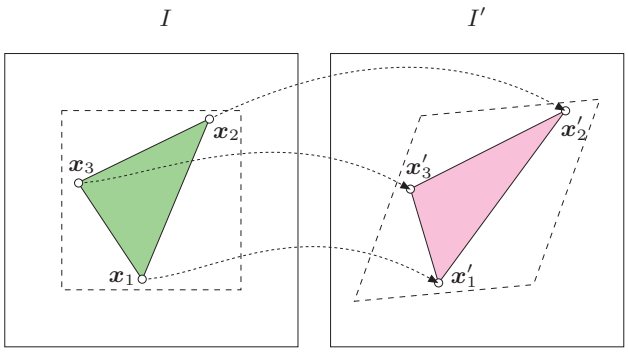


Fig. 16.2
Affine mapping. This mapping transforms straight lines to straight lines, triangles to triangles, and rectangles to parallelograms. Parallel lines remain parallel, and the distance ratio between points on a straight line does not change. An affine 2D transformation is uniquely specified by three pairs of corresponding points; e. g., $(\boldsymbol{x}_1, \boldsymbol{x}'_1)$, $(\boldsymbol{x}_2, \boldsymbol{x}'_2)$, and $(\boldsymbol{x}_3, \boldsymbol{x}'_3)$.

Determining transformation parameters

The six parameters of the 2D affine mapping (Eqn. (16.13)) are uniquely determined by three pairs of corresponding points $(\boldsymbol{x}_1, \boldsymbol{x}'_1)$, $(\boldsymbol{x}_2, \boldsymbol{x}'_2)$, $(\boldsymbol{x}_3, \boldsymbol{x}'_3)$, with the first point $\boldsymbol{x}_i = (x_i, y_i)$ of each pair located in the original image and the corresponding point $\boldsymbol{x}'_i = (x'_i, y'_i)$ located in the target image. From these six coordinate values, the six transformation parameters $a_{11} \ldots a_{23}$ are derived by solving the system of linear equations

$$\begin{aligned} x'_1 &= a_{11}\cdot x_1 + a_{12}\cdot y_1 + a_{13}, \qquad & y'_1 &= a_{21}\cdot x_1 + a_{22}\cdot y_1 + a_{23}, \\ x'_2 &= a_{11}\cdot x_2 + a_{12}\cdot y_2 + a_{13}, \qquad & y'_2 &= a_{21}\cdot x_2 + a_{22}\cdot y_2 + a_{23}, \\ x'_3 &= a_{11}\cdot x_3 + a_{12}\cdot y_3 + a_{13}, \qquad & y'_3 &= a_{21}\cdot x_3 + a_{22}\cdot y_3 + a_{23}, \end{aligned} \tag{16.14}$$

provided that the points (vectors) $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{x}_3$ are linearly independent (i. e., that they do not lie on a common straight line). Since Eqn. (16.14) consists of two independent sets of linear 3×3 equations for x'_i and y'_i, the solution can be written in closed form as

$$\begin{aligned} a_{11} &= \tfrac{1}{d}\cdot\big[y_1(x'_2 - x'_3) &&+ y_2(x'_3 - x'_1) &&+ y_3(x'_1 - x'_2)\big], \\ a_{12} &= \tfrac{1}{d}\cdot\big[x_1(x'_3 - x'_2) &&+ x_2(x'_1 - x'_3) &&+ x_3(x'_2 - x'_1)\big], \\ a_{21} &= \tfrac{1}{d}\cdot\big[y_1(y'_2 - y'_3) &&+ y_2(y'_3 - y'_1) &&+ y_3(y'_1 - y'_2)\big], \\ a_{22} &= \tfrac{1}{d}\cdot\big[x_1(y'_3 - y'_2) &&+ x_2(y'_1 - y'_3) &&+ x_3(y'_2 - y'_1)\big], \\ a_{13} &= \tfrac{1}{d}\cdot\big[x_1(y_3x'_2 - y_2x'_3) &&+ x_2(y_1x'_3 - y_3x'_1) &&+ x_3(y_2x'_1 - y_1x'_2)\big], \\ a_{23} &= \tfrac{1}{d}\cdot\big[x_1(y_3y'_2 - y_2y'_3) &&+ x_2(y_1y'_3 - y_3y'_1) &&+ x_3(y_2y'_1 - y_1y'_2)\big], \end{aligned} \tag{16.15}$$

with $d = x_1(y_3 - y_2) + x_2(y_1 - y_3) + x_3(y_2 - y_1)$.

Inverse mapping

The inverse $T^{-1}()$ of the affine mapping function, which is often required in practice (see Sec. 16.2.2), can be found by computing the inverse of the transformation matrix (Eqn. (16.13)),

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \tag{16.16}$$

$$= \frac{1}{a_{11}a_{22} - a_{12}a_{21}} \begin{pmatrix} a_{22} & -a_{12} & a_{12}a_{23} - a_{13}a_{22} \\ -a_{21} & a_{11} & a_{13}a_{21} - a_{11}a_{23} \\ 0 & 0 & a_{11}a_{22} - a_{12}a_{21} \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}.$$

Of course, the inverse of the affine mapping can also be found directly (i. e., without inverting the transformation matrix) from the given point coordinates $(\boldsymbol{x}_i, \boldsymbol{x}'_i)$ using Eqn. (16.15) with source and target coordinates interchanged.

16.1.4 Projective (Four-Point) Mapping

In contrast to the affine transformation, which provides a mapping between arbitrary triangles, the projective transformation is a linear mapping between arbitrary *quadrilaterals* (Fig. 16.3). This is particularly useful for deforming images controlled by mesh partitioning, as described in Sec. 16.1.7. To map from an arbitrary sequence of four 2D points $(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4)$ to a set of corresponding points $(\boldsymbol{x}'_1, \boldsymbol{x}'_2, \boldsymbol{x}'_3, \boldsymbol{x}'_4)$, the transformation requires eight degrees of freedom, two more than needed for the affine transformation. Similar to the affine transformation, the projective transformation can be expressed as a linear mapping in homogeneous coordinates, with two additional parameters (a_{31}, a_{32}):

$$\begin{pmatrix} \hat{x}' \\ \hat{y}' \\ h' \end{pmatrix} = \begin{pmatrix} h'x' \\ h'y' \\ h' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \tag{16.17}$$

In Cartesian coordinates, the resulting mapping functions

$$x' = \frac{1}{h'} \cdot (a_{11}\, x + a_{12}\, y + a_{13}) = \frac{a_{11}\, x + a_{12}\, y + a_{13}}{a_{31}\, x + a_{32}\, y + 1}, \tag{16.18}$$

$$y' = \frac{1}{h'} \cdot (a_{21}\, x + a_{22}\, y + a_{23}) = \frac{a_{21}\, x + a_{22}\, y + a_{23}}{a_{31}\, x + a_{32}\, y + 1}, \tag{16.19}$$

are apparently nonlinear. Despite this nonlinearity, straight lines are preserved under this transformation. In fact, this is the most general transformation that maps straight lines to straight lines in 2D, and it actually maps any Nth-order algebraic curve onto another Nth-order algebraic curve. In particular, circles and ellipses always transform into other second-order curves (i. e., conic sections). Unlike the affine transformation, however, parallel lines do not generally map to parallel lines under a projective transformation (cf. Fig. 16.3) and the distance ratios between points on a line are not preserved. The projective mapping is therefore sometimes referred to as "perspective" or "pseudo-perspective".

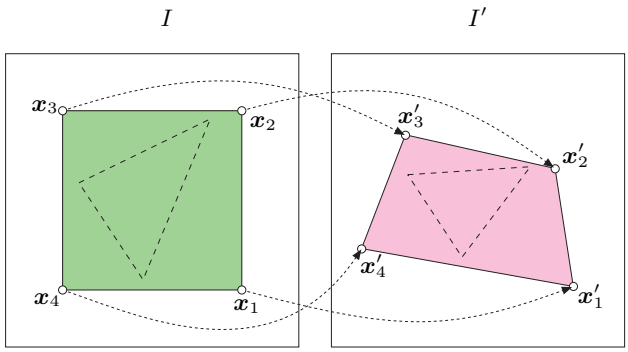


Fig. 16.3
Projective mapping. Four pairs of corresponding 2D points uniquely specify a projective transformation. Straight lines are again mapped to straight lines, and a rectangle is mapped to some quadrilateral. In general, neither parallelism between straight lines nor distance ratios are preserved.

Determining transformation parameters

Given four pairs of corresponding 2D points, $(\boldsymbol{x}_1, \boldsymbol{x}'_1) \dots (\boldsymbol{x}_4, \boldsymbol{x}'_4)$, with one point $\boldsymbol{x}_i = (x_i, y_i)$ in the source image and the second point $\boldsymbol{x}'_i = (x'_i, y'_i)$ in the target image, the eight unknown transformation parameters $a_{11} \dots a_{32}$ can be found by solving a system of linear equations. Inserting the given point coordinates x'_i, y'_i into Eqn. (16.18), we get for each point pair $i = 1 \dots 4$ a pair of linear equations

$$\begin{aligned} x'_i &= a_{11}\, x_i + a_{12}\, y_i + a_{13} - a_{31}\, x_i\, x'_i - a_{32}\, y_i\, x'_i, \\ y'_i &= a_{21}\, x_i + a_{22}\, y_i + a_{23} - a_{31}\, x_i\, y'_i - a_{32}\, y_i\, y'_i, \end{aligned} \tag{16.20}$$

for the eight unknowns $a_{11} \dots a_{32}$. Combining the resulting eight equations in matrix notation gives

$$\begin{pmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4 \end{pmatrix} \cdot \begin{pmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \end{pmatrix} \tag{16.21}$$

or

$$\boldsymbol{x}' = \boldsymbol{M} \cdot \boldsymbol{a}\,. \tag{16.22}$$

The unknown parameters $\boldsymbol{a} = (a_{11}, a_{12}, \dots a_{32})^T$ can be found by solving the system of linear equations above using standard numerical methods such as the Gauss algorithm [15, p. 276].[1]

[1] We recommend relying on existing numerical software libraries for this purpose. Several free packages are available for C/C++ but only a few exist for Java; e. g., *JAMA*—A Java Matrix Package (http://math.nist.gov/javanumerics/jama/), which we use here.

Inverse mapping

In general, a *linear* transformation of the form $\boldsymbol{x}' = \boldsymbol{A}\cdot\boldsymbol{x}$ can be inverted by computing the inverse of the matrix $\boldsymbol{A}$ (i. e., $\boldsymbol{x} = \boldsymbol{A}^{-1}\cdot\boldsymbol{x}'$), provided that $\boldsymbol{A}$ is nonsingular ($\mathrm{Det}(\boldsymbol{A}) \neq 0$). The inverse of a 3×3 matrix $\boldsymbol{A}$ is relatively easy to find using the relation

$$\boldsymbol{A}^{-1} = \frac{1}{\mathrm{Det}(\boldsymbol{A})} \cdot \boldsymbol{A}_{\mathrm{adj}} \tag{16.23}$$

between the determinant $\mathrm{Det}(\boldsymbol{A})$ and the corresponding adjoint matrix $\boldsymbol{A}_{\mathrm{adj}}$ [15, pp. 251, 260]. For an arbitrary 3×3 matrix

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix},$$

the determinant is

$$\begin{aligned} \mathrm{Det}(\boldsymbol{A}) = \quad & a_{11}\,a_{22}\,a_{33} + a_{12}\,a_{23}\,a_{31} + a_{13}\,a_{21}\,a_{32} \\ & - a_{11}\,a_{23}\,a_{32} - a_{12}\,a_{21}\,a_{33} - a_{13}\,a_{22}\,a_{31} \end{aligned} \tag{16.24}$$

and its adjoint matrix is

$$\boldsymbol{A}_{\mathrm{adj}} = \begin{pmatrix} a_{22}\,a_{33} - a_{23}\,a_{32} & a_{13}\,a_{32} - a_{12}\,a_{33} & a_{12}\,a_{23} - a_{13}\,a_{22} \\ a_{23}\,a_{31} - a_{21}\,a_{33} & a_{11}\,a_{33} - a_{13}\,a_{31} & a_{13}\,a_{21} - a_{11}\,a_{23} \\ a_{21}\,a_{32} - a_{22}\,a_{31} & a_{12}\,a_{31} - a_{11}\,a_{32} & a_{11}\,a_{22} - a_{12}\,a_{21} \end{pmatrix}. \tag{16.25}$$

In the special case of a projective mapping, the coefficient $a_{33} = 1$ (Eqn. (16.17)), which slightly simplifies the computation. Since scalar multiples of homogeneous vectors are all equivalent in Cartesian space, the multiplication by the factor $1/\mathrm{Det}(\boldsymbol{A})$ in Eqn. (16.23) can be omitted. Thus, to invert the linear transformation, we only need to multiply the homogeneous coordinate vector with the adjoint matrix $\boldsymbol{A}_{\mathrm{adj}}$ and (if needed) "homogenize" the resulting vector,

$$\begin{pmatrix} \hat{x} \\ \hat{y} \\ h \end{pmatrix} = \boldsymbol{A}_{\mathrm{adj}} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \quad \text{and subsequently} \quad \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \frac{1}{h} \begin{pmatrix} \hat{x} \\ \hat{y} \\ h \end{pmatrix}. \tag{16.26}$$

This method can be used to invert any linear mapping function in 2D, including the affine and projective mapping functions described above. Consequently, the inversion of the *affine* transformation shown earlier (Eqn. (16.16)) is only a special case of this general method.

Projective mapping via the unit square

An alternative method for finding the projective mapping parameters for a given set of image points is to use a two-stage mapping through

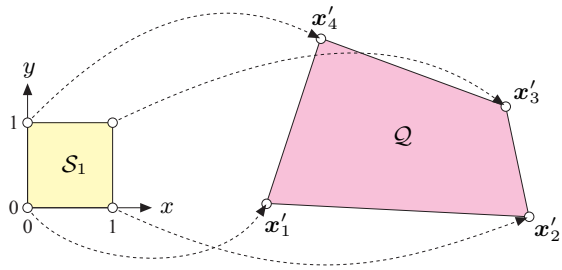


Fig. 16.4
Projective mapping from the unit square $\mathcal{S}_1$ to an arbitrary quadrilateral $\mathcal{Q} = (\boldsymbol{x}'_1, \dots \boldsymbol{x}'_4)$.

the unit square, which avoids iteratively solving a system of equations. The projective mapping, shown in Fig. 16.4, from the corner points of the unit square $\mathcal{S}_1$ to an arbitrary quadrilateral $\mathcal{Q} = \boldsymbol{x}'_1, \dots \boldsymbol{x}'_4$ with

$$\begin{aligned} (0,0) &\to \boldsymbol{x}'_1 \qquad & (1,1) &\to \boldsymbol{x}'_3 \\ (1,0) &\to \boldsymbol{x}'_2 \qquad & (0,1) &\to \boldsymbol{x}'_4 \end{aligned}$$

reduces the system of equations in Eqn. (16.21) to

$$\begin{aligned} x'_1 &= a_{13}, \\ y'_1 &= a_{23}, \\ x'_2 &= a_{11} + a_{13} - a_{31} \cdot x'_2, \\ y'_2 &= a_{21} + a_{23} - a_{31} \cdot y'_2, \\ x'_3 &= a_{11} + a_{12} + a_{13} - a_{31} \cdot x'_3 - a_{32} \cdot x'_3, \\ y'_3 &= a_{21} + a_{22} + a_{23} - a_{31} \cdot y'_3 - a_{32} \cdot y'_3, \\ x'_4 &= a_{12} + a_{13} - a_{32} \cdot x'_4, \\ y'_4 &= a_{22} + a_{23} - a_{32} \cdot y'_4. \end{aligned} \tag{16.27}$$

This set of equations has the following closed-form solution for the eight unknown transformation parameters $a_{11}, a_{12}, \dots a_{32}$:

$$a_{31} = \frac{(x'_1 - x'_2 + x'_3 - x'_4) \cdot (y'_4 - y'_3) - (y'_1 - y'_2 + y'_3 - y'_4) \cdot (x'_4 - x'_3)}{(x'_2 - x'_3) \cdot (y'_4 - y'_3) \;-\; (x'_4 - x'_3) \cdot (y'_2 - y'_3)}, \tag{16.28}$$

$$a_{32} = \frac{(y'_1 - y'_2 + y'_3 - y'_4) \cdot (x'_2 - x'_3) - (x'_1 - x'_2 + x'_3 - x'_4) \cdot (y'_2 - y'_3)}{(x'_2 - x'_3) \cdot (y'_4 - y'_3) \;-\; (x'_4 - x'_3) \cdot (y'_2 - y'_3)}, \tag{16.29}$$

$$a_{11} = x'_2 - x'_1 + a_{31}\, x'_2, \quad a_{12} = x'_4 - x'_1 + a_{32}\, x'_4, \quad a_{13} = x'_1, \tag{16.30}$$

$$a_{21} = y'_2 - y'_1 + a_{31}\, y'_2, \quad a_{22} = y'_4 - y'_1 + a_{32}\, y'_4, \quad a_{23} = y'_1. \tag{16.31}$$

By computing the inverse of the corresponding 3×3 transformation matrix (Eqn. (16.23)), the mapping may be *reversed* to transform an arbitrary quadrilateral to the unit square. A mapping T between two arbitrary quadrilaterals

$$\mathcal{Q}_1 \xrightarrow{T} \mathcal{Q}_2$$

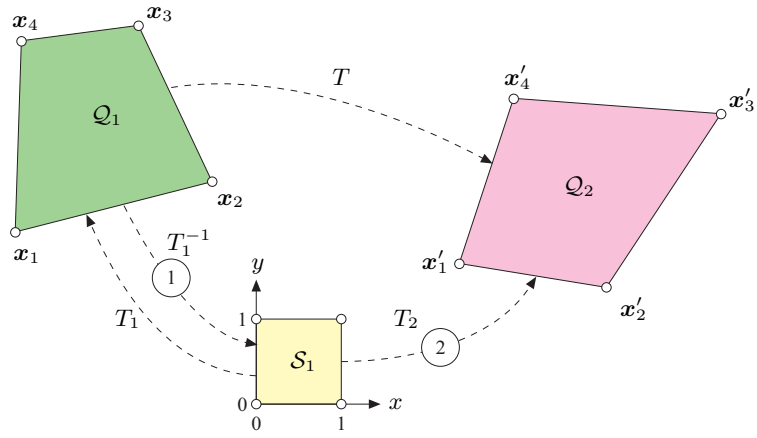


Fig. 16.5
Two-step projective transformation between arbitrary quadrilaterals. In the first step, quadrilateral $\mathcal{Q}_1$ is transformed to the unit square $\mathcal{S}_1$ by the inverse mapping function T_1^{-1}. In the second step, T_2 transforms the square $\mathcal{S}_1$ to the target quadrilateral $\mathcal{Q}_2$. The complete mapping T results from the concatenation of the mappings T_1^{-1} and T_2.

can thus be implemented by combining a reversed mapping and a forward mapping via the unit square [105, p. 55] [44]. As illustrated in Fig. 16.5, the transformation of the first quadrilateral $\mathcal{Q}_1 = (\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4)$ to the second quadrilateral $\mathcal{Q}_2 = (\boldsymbol{x}'_1, \boldsymbol{x}'_2, \boldsymbol{x}'_3, \boldsymbol{x}'_4)$ is accomplished in two steps involving the linear transformations T_1 and T_2 between the two quadrilaterals and the unit square $\mathcal{S}_1$:

$$\mathcal{Q}_1 \xrightarrow{T_1^{-1}} \mathcal{S}_1 \xrightarrow{T_2} \mathcal{Q}_2. \tag{16.32}$$

The projective transformation parameters for T_1 and T_2 are obtained by inserting the corresponding point coordinates of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ ($\boldsymbol{x}_i$ and $\boldsymbol{x}'_i$, respectively) into Eqns. (16.28)–(16.31). The complete transformation T is then the concatenation of the two transformations T_1^{-1} and T_2,

$$\boldsymbol{x}' = T(\boldsymbol{x}) = T_2(T_1^{-1}(\boldsymbol{x})), \tag{16.33}$$

or, expressed in matrix notation,

$$\boldsymbol{x}' = \boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{A}_2 \cdot \boldsymbol{A}_1^{-1} \cdot \boldsymbol{x}. \tag{16.34}$$

Of course, the matrix $\boldsymbol{A} = \boldsymbol{A}_2 \cdot \boldsymbol{A}_1^{-1}$ needs to be computed only once for a particular transformation and can then be used repeatedly for mapping all required image points.

Example

The source and the target quadrilaterals $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are specified by the following coordinate points:

$$\begin{aligned} \mathcal{Q}_1: \quad & \boldsymbol{x}_1 = (2,5) \quad \boldsymbol{x}_2 = (4,6) \quad \boldsymbol{x}_3 = (7,9) \quad \boldsymbol{x}_4 = (5,9) \\ \mathcal{Q}_2: \quad & \boldsymbol{x}'_1 = (4,3) \quad \boldsymbol{x}'_2 = (5,2) \quad \boldsymbol{x}'_3 = (9,3) \quad \boldsymbol{x}'_4 = (7,5) \end{aligned}$$

Using Eqns. (16.28)–(16.31), the transformation parameters (matrices) for the projective mappings from the unit $\mathcal{S}_1$ square to the quadrilaterals $\boldsymbol{A}_1 : \mathcal{S}_1 \to \mathcal{Q}_1$ and $\boldsymbol{A}_2 : \mathcal{S}_1 \to \mathcal{Q}_2$ are obtained as

$$\boldsymbol{A}_1 = \begin{pmatrix} 3.3\dot{3} & 0.50 & 2.00 \\ 3.00 & -0.50 & 5.00 \\ 0.3\dot{3} & -0.50 & 1.00 \end{pmatrix} \quad \text{and} \quad \boldsymbol{A}_2 = \begin{pmatrix} 1.00 & -0.50 & 4.00 \\ -1.00 & -0.50 & 3.00 \\ 0.00 & -0.50 & 1.00 \end{pmatrix}.$$

Concatenating the inverse mapping $\boldsymbol{A}_1^{-1}$ with $\boldsymbol{A}_2$ (by matrix multiplication), we get the complete mapping $\boldsymbol{A} = \boldsymbol{A}_2 \cdot \boldsymbol{A}_1^{-1}$ with

$$\boldsymbol{A}_1^{-1} = \begin{pmatrix} 0.60 & -0.45 & 1.05 \\ -0.40 & 0.80 & -3.20 \\ -0.40 & 0.55 & -0.95 \end{pmatrix} \quad \text{and} \quad \boldsymbol{A} = \begin{pmatrix} -0.80 & 1.35 & -1.15 \\ -1.60 & 1.70 & -2.30 \\ -0.20 & 0.15 & 0.65 \end{pmatrix}.$$

The Java method `makeMapping()` in class `ProjectiveMapping` (p. 420) shows an implementation of this two-step technique.

16.1.5 Bilinear Mapping

Similar to the projective transformation (Eqn. (16.17)), the bilinear mapping function

$$\begin{aligned} T_x: \quad x' &= a_1 x + a_2 y + a_3 xy + a_4, \\ T_y: \quad y' &= b_1 x + b_2 y + b_3 xy + b_4, \end{aligned} \tag{16.35}$$

is specified with four pairs of corresponding points and has eight parameters ($a_1 \ldots a_4,\ b_1 \ldots b_4$). The transformation is nonlinear because of the mixed term xy and cannot be described by a linear transformation, even with homogeneous coordinates. In contrast to the projective transformation, the straight lines are not preserved in general but map onto quadratic curves. Similarly, circles are not mapped to ellipses by a bilinear transform.

A bilinear mapping is uniquely specified by four corresponding pairs of 2D points $(\boldsymbol{x}_1, \boldsymbol{x}'_1) \ldots (\boldsymbol{x}_4, \boldsymbol{x}'_4)$. In the general case, for a bilinear mapping between arbitrary quadrilaterals, the coefficients $a_1 \ldots a_4$, $b_1 \ldots b_4$ (Eqn. (16.35)) are found as the solution of two separate systems of equations, each with four unknowns:

$$\begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & x_1 & y_1 & 1 \\ x_2 & y_2 & x_2 & y_2 & 1 \\ x_3 & y_3 & x_3 & y_3 & 1 \\ x_4 & y_4 & x_4 & y_4 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \quad \text{or} \quad \boldsymbol{x} = \boldsymbol{M} \cdot \boldsymbol{a}, \tag{16.36}$$

$$\begin{pmatrix} y'_1 \\ y'_2 \\ y'_3 \\ y'_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & x_1 & y_1 & 1 \\ x_2 & y_2 & x_2 & y_2 & 1 \\ x_3 & y_3 & x_3 & y_3 & 1 \\ x_4 & y_4 & x_4 & y_4 & 1 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \quad \text{or} \quad \boldsymbol{y} = \boldsymbol{M} \cdot \boldsymbol{b}. \tag{16.37}$$

These equations can again be solved using standard numerical techniques, as described on page 381. A sample implementation of this computation is shown by the Java method `makeInverseMapping()` inside the class `BilinearMapping` on page 422.

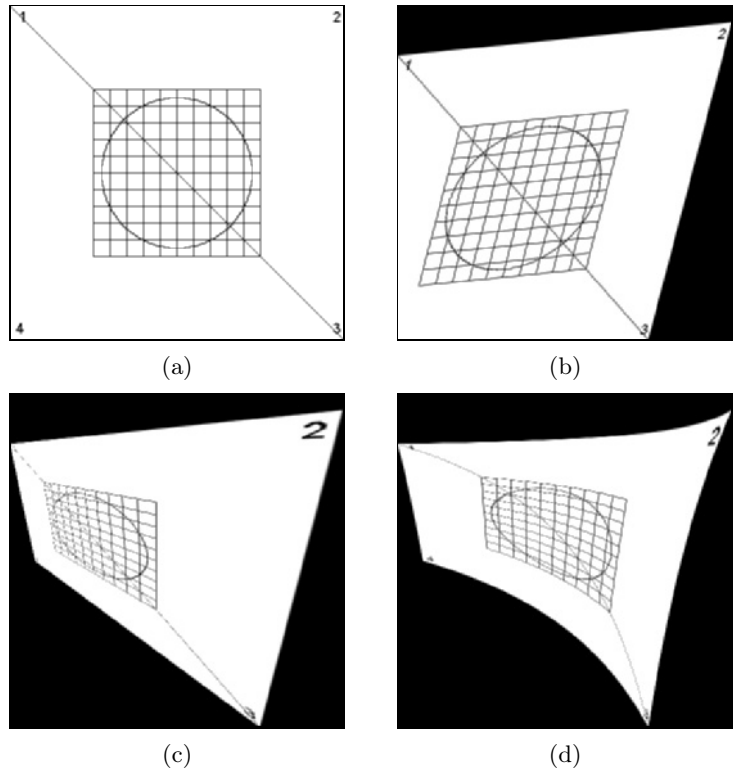


Fig. 16.6
Geometric transformations compared: original image (a), affine transformation with respect to the triangle 1-2-3 (b), projective transformation (c), and bilinear transformation (d).

In the special case of bilinearly mapping the unit square $\mathcal{S}_1$ to an arbitrary quadrilateral $\mathcal{Q} = (\boldsymbol{x}'_1, \ldots \boldsymbol{x}'_4)$, the parameters $a_1 \ldots a_4$ and $b_1 \ldots b_4$ are

$$
\begin{aligned}
a_1 &= x'_2 - x'_1, & b_1 &= y'_2 - y'_1,\\
a_2 &= x'_4 - x'_1, & b_2 &= y'_4 - y'_1,\\
a_3 &= x'_1 - x'_2 + x'_3 - x'_4, & b_3 &= y'_1 - y'_2 + y'_3 - y'_4,\\
a_4 &= x'_1, & b_4 &= y'_1.
\end{aligned}
$$

Figure 16.6 shows results of the affine, projective, and bilinear transformations applied to a simple test pattern. The affine transformation (Fig. 16.6 (b)) is specified by mapping to the triangle 1-2-3, while the four points of the quadrilateral 1-2-3-4 define the projective and the bilinear transforms (Fig. 16.6 (c, d)).

16.1.6 Other Nonlinear Image Transformations

The bilinear transformation discussed in the previous section is only one example of a nonlinear mapping in 2D that cannot be expressed as a simple matrix-vector multiplication in homogeneous coordinates. Many other types of nonlinear deformations exist; for example, to implement

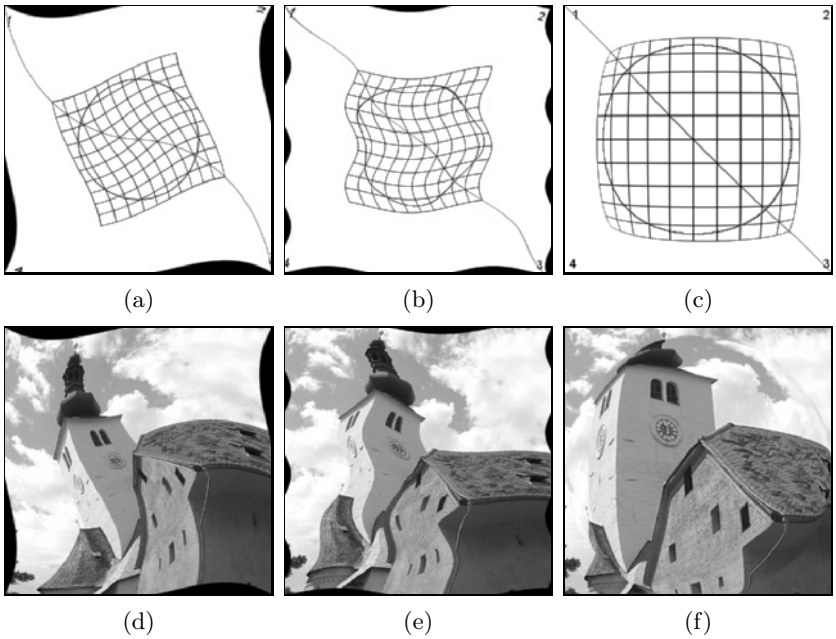


Fig. 16.7
Various nonlinear image deformations: *twirl* (a, d), *ripple* (b, e), and *sphere* (c, f) transformations. The original (source) images are shown in Fig. 16.6 (a) and Fig. 16.1 (a).

various artistic effects for creative imaging. This type of image deformation is often called "image warping".

Depending on the type of transformation used, the derivation of the *inverse* transformation function—which is required for the practical computation of the mapping using *target-to-source mapping* (see Sec. 16.2.2)—is not always easy or may even be impossible. In the following three examples, we therefore look straight at the inverse maps

$$\boldsymbol{x} = T^{-1}(\boldsymbol{x}')$$

without really bothering about the corresponding forward transformations.

"Twirl" transformation

The twirl mapping causes the image to be rotated around a given anchor point $\boldsymbol{x}_c = (x_c, y_c)$ with a space-variant rotation angle, which has a fixed value α at the center $\boldsymbol{x}_c$ and decreases linearly with the radial distance from the center. The image remains unchanged outside the limiting radius $r_{\max}$. The corresponding (inverse) mapping function is defined as

$$T_x^{-1}: \quad x = \begin{cases} x_c + r \cdot \cos(\beta) & \text{for } r \le r_{\max} \\ x' & \text{for } r > r_{\max}, \end{cases} \tag{16.38}$$

$$T_y^{-1}: \quad y = \begin{cases} y_c + r \cdot \sin(\beta) & \text{for } r \le r_{\max} \\ y' & \text{for } r > r_{\max}, \end{cases} \tag{16.39}$$

with

$$d_x = x' - x_c, \qquad r = \sqrt{d_x^2 + d_y^2},$$
$$d_y = y' - y_c, \qquad \beta = \mathrm{ArcTan}(d_x, d_y) + \alpha \cdot \left(\tfrac{r_{\max} - r}{r_{\max}}\right).$$

Figure 16.7 (a, d) shows a twirl mapping with the anchor point $\boldsymbol{x}_c$ placed at the image center. The limiting radius $r_{\max}$ is half the length of the image diagonal, and the rotation angle is $\alpha = 43°$ at the center. A Java implementation of this transformation is shown in the class `TwirlMapping` on page 422.

"Ripple" transformation

The ripple transformation causes a local wavelike displacement of the image along both the x and y directions. The parameters of this mapping function are the period lengths $\tau_x, \tau_y \neq 0$ (in pixels) and the corresponding amplitude values a_x, a_y for the displacement in both directions:

$$T_x^{-1}: \quad x = x' + a_x \cdot \sin\left(\tfrac{2\pi \cdot y'}{\tau_x}\right), \tag{16.40}$$
$$T_y^{-1}: \quad y = y' + a_y \cdot \sin\left(\tfrac{2\pi \cdot x'}{\tau_y}\right). \tag{16.41}$$

An example for the ripple mapping with $\tau_x = 120$, $\tau_y = 250$, $a_x = 10$, and $a_y = 15$ is shown in Fig. 16.7 (b, e).

Spherical transformation

The spherical deformation imitates the effect of viewing the image through a transparent hemisphere or lens placed on top of the image. The parameters of this transformation are the position $\boldsymbol{x}_c = (x_c, y_c)$ of the lens center, the radius of the lens $r_{\max}$ and its refraction index ρ. The corresponding mapping functions are defined as

$$T_x^{-1}: \quad x = x' - \begin{cases} z \cdot \tan(\beta_x) & \text{for } r \le r_{\max} \\ 0 & \text{for } r > r_{\max}, \end{cases} \tag{16.42}$$
$$T_y^{-1}: \quad y = y' - \begin{cases} z \cdot \tan(\beta_y) & \text{for } r \le r_{\max} \\ 0 & \text{for } r > r_{\max}, \end{cases} \tag{16.43}$$

with

$$d_x = x' - x_c, \quad r = \sqrt{d_x^2 + d_y^2}\,, \quad \beta_x = \left(1 - \tfrac{1}{\rho}\right) \cdot \sin^{-1}\left(\tfrac{d_x}{\sqrt{(d_x^2 + z^2)}}\right),$$
$$d_y = y' - y_c, \quad z = \sqrt{r_{\max}^2 - r^2}\,, \quad \beta_y = \left(1 - \tfrac{1}{\rho}\right) \cdot \sin^{-1}\left(\tfrac{d_y}{\sqrt{(d_y^2 + z^2)}}\right).$$

Figure 16.7 (c, f) shows a spherical transformation with the lens positioned at the image center. The lens radius $r_{\max}$ is set to half of the image width, and the refraction index is $\rho = 1.8$.

16.1.7 Local Image Transformations

All the geometric transformations discussed so far are *global* (i. e., the same mapping function is applied to all pixels in the given image). It is often necessary to deform an image such that a larger number of n original image points $\boldsymbol{x}_1 \ldots \boldsymbol{x}_n$ are precisely mapped onto a given set of target points $\boldsymbol{x}'_1 \ldots \boldsymbol{x}'_n$. For $n = 3$, this problem can be solved with an affine mapping (see Sec. 16.1.3), and for $n = 4$ we could use a projective or bilinear mapping (see Secs. 16.1.4 and 16.1.5). A precise global mapping of $n > 4$ points requires a more complicated function $T(\boldsymbol{x})$ (e. g., a two-dimensional nth-order polynomial or a spline function).

An alternative is to use *local* or *piecewise* transformations, where the image is partitioned into disjoint patches that are transformed separately, applying an individual mapping function to each patch. In practice, it is common to partition the image into a *mesh* of triangles or quadrilaterals, as illustrated in Fig. 16.8.

For a *triangular* mesh partitioning (Fig. 16.8 (a)), the transformation between each pair of triangles $\mathcal{D}_i \rightarrow \mathcal{D}'_i$ could be accomplished with an *affine* mapping, whose parameters must be computed individually for every patch. Similarly, the *projective* transformation would be suitable for mapping each patch in a mesh partitioning composed of *quadrilaterals*

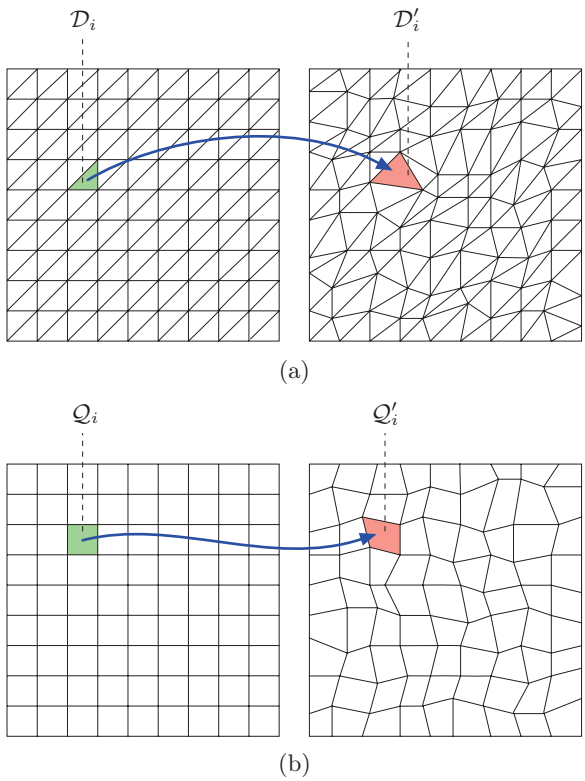


Fig. 16.8
Mesh partitioning. Almost arbitrary image deformations can be implemented by partitioning the image plane into nonoverlapping triangles $\mathcal{D}_i, \mathcal{D}'_i$ (a) or quadrilaterals $\mathcal{Q}_i, \mathcal{Q}'_i$ (b) and applying simple local transformations. Every patch in the resulting mesh is transformed separately with the required transformation parameters derived from the corresponding three or four corner points, respectively.

$\mathcal{Q}_i$ (Fig. 16.8 (b)). Since both the affine and the projective transformations preserve the straightness of lines, we can be certain that no holes or overlaps will arise and the deformation will appear continuous between adjacent mesh patches.

Local transformations of this type are frequently used; for example, to register aerial and satellite images or to undistort images for panoramic stitching. In computer graphics, similar techniques are used to map texture images onto polygonal 3D surfaces in the rendered 2D image. Another popular application of this technique is "morphing" [105], which performs a stepwise geometric transformation from one image to another while simultaneously blending their intensity (or color) values.[2]

16.2 Resampling the Image

In the discussion of geometric transformations, we have so far considered the 2D image coordinates as being continuous (i. e., real-valued). In reality, the picture elements in digital images reside at discrete (i. e., integer-valued) coordinates, and thus transferring a discrete image into another discrete image without introducing significant losses in quality is a nontrivial subproblem in the implementation of geometric transformations.

Based on the original image $I(u, v)$ and some (continuous) geometric transformations $T(x, y)$, the aim is to create a transformed image $I'(u', v')$ where all coordinates are discrete (i. e., $u, v \in \mathbb{Z}$ and $u', v' \in \mathbb{Z}$).[3] This can be accomplished in one of two ways, which differ by the mapping direction and are commonly referred to as *source-to-target* or *target-to-source* mapping, respectively.

16.2.1 Source-to-Target Mapping

In this approach, which appears quite natural at first sight, we compute for every pixel (u, v) of the original (*source*) image I the corresponding transformed position

$$(x', y') = T(u, v)$$

in the target image I'. In general, the result will *not* coincide with any of the raster points, as illustrated in Fig. 16.9. Subsequently, we would have to decide in which pixel in the target image I' the original intensity or color value from $I(u, v)$ should be stored. We could perhaps even think of somehow distributing this value onto all adjacent pixels.

The problem with the source-to-target method is that, depending on the geometric transformation T, some elements in the target image I'

[2] Image morphing has also been implemented in ImageJ as a plugin (*iMorph*) by Hajime Hirase (http://rsb.info.nih.gov/ij/plugins/morph.html).

[3] Remark on notation: We use (u, v) or (u', v') to denote *discrete* (integer) coordinates and (x, y) or (x', y') for *continuous* (real-valued) coordinates.

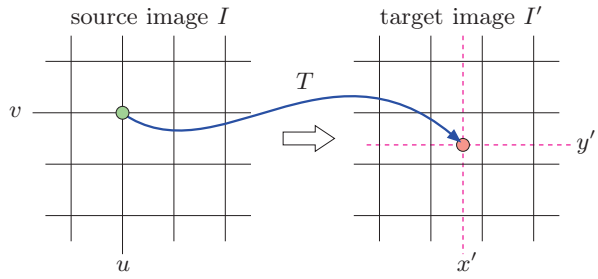


Fig. 16.9
Source-to-target mapping. For each discrete pixel position (u, v) in the source image I, the corresponding (continuous) target position (x', y') is found by applying the geometric transformation $T(u, v)$. In general, the target position (x', y') does not coincide with any discrete raster point. The source pixel value $I(u, v)$ is subsequently transferred to one of the adjacent target pixels.

may never be "hit" at all (i. e., never receive a source pixel value)! This happens, for example, when the image is enlarged (even slightly) by the geometric transformation. The resulting holes in the target image would be difficult to close in a subsequent processing step. Conversely, one would have to consider (e. g., when the image is shrunk) that a single element in the target image I' may be hit by multiple source pixels and thus image content may get lost. In the light of all these complications, source-to-target mapping is not really the method of choice.

16.2.2 Target-to-Source Mapping

This method avoids most difficulties encountered in the source-to-target mapping by simply reversing the image generation process. For every discrete pixel position (u', v') in the *target* image, we compute the corresponding (continuous) point

$$(x, y) = T^{-1}(u', v')$$

in the source image plane using the inverse geometric transformation T^{-1}. Of course, the coordinate (x, y) again does not fall onto a raster point in general and thus we have to decide from which of the neighboring source pixels to extract the resulting target pixel value. This problem of interpolating among intensity values will be discussed in detail in Sec. 16.3.

The major advantage of the target-to-source method is that all pixels in the target image I' (and only these) are computed and filled exactly

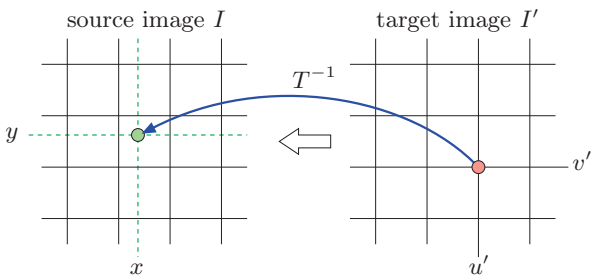


Fig. 16.10
Target-to-source mapping. For each discrete pixel position (u', v') in the target image I', the corresponding continuous source position (x, y) is found by applying the inverse mapping function $T^{-1}(u', v')$. The new pixel value $I'(u', v')$ is determined by interpolating the pixel values in the source image within some neighborhood of (x, y).

once such that no holes or multiple hits can occur. This, however, requires the *inverse* geometric transformation T^{-1} to be available, which is no disadvantage in most cases since the forward transformation T itself is never really needed. Due to its simplicity, which is also demonstrated in Alg. 16.1, *target-to-source* mapping is the common method for geometrically transforming 2D images.

Algorithm 16.1
Geometric image transformation using target-to-source mapping. Given are the original (source) image I and the continuous coordinate transformation T. getInterpolatedValue(I, x, y) returns the interpolated value of the source image I at the continuous position (x, y).

```
1: TransformImage (I, T)
       I: source image
       T: coordinate transform function
2:     Create target image I'.
3:     for all target image coordinates (u', v') do
4:         Let (x, y) ← T^{-1}(u', v')
5:         I'(u', v') ← getInterpolatedValue(I, x, y)
6:     return target image I'.
```

16.3 Interpolation

Interpolation is the process of estimating the intermediate values of a sampled function or signal at continuous positions or the attempt to reconstruct the original continuous function from a set of discrete samples. In the context of geometric operations this task arises from the fact that discrete pixel positions in one image are generally not mapped to discrete raster positions in the other image under some continuous geometric transformation T (or T^{-1}, respectively). The concrete goal is to obtain an optimal estimate for the value of the two-dimensional image function $I(x, y)$ at any continuous position $(x, y) \in \mathbb{R}^2$. In practice, the interpolated function should preserve as much detail (i. e., sharpness) as possible without causing visible artifacts such as ringing or moiré patterns.

16.3.1 Simple Interpolation Methods

To illustrate the problem, we first attend to the one-dimensional case (Fig. 16.11). Several simple methods exist for interpolating the values of a discrete function $g(u)$, with $u \in \mathbb{Z}$, at arbitrary continuous positions $x \in \mathbb{R}$. While these ad hoc methods are easy to implement, they lack a theoretical justification and usually give poor results.

Nearest-neighbor interpolation

The simplest of all interpolation methods is to round the continuous coordinate x to the closest integer u_0 and use the sample $g(u_0)$ as the estimated function value $\hat{g}(x)$,

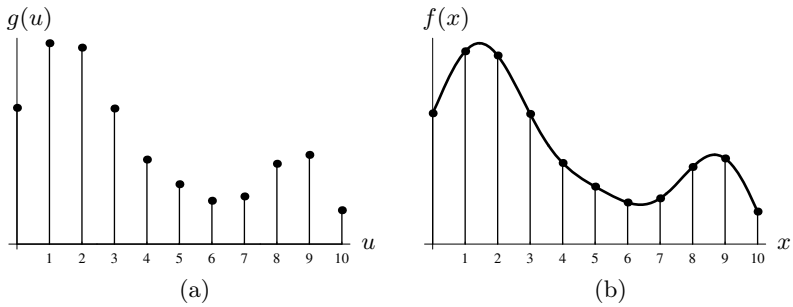


Fig. 16.11
Interpolating a discrete function in 1D. Given the discrete function values $g(u)$ (a), the goal is to estimate the original function $f(x)$ at arbitrary continuous positions $x \in \mathbb{R}$ (b).

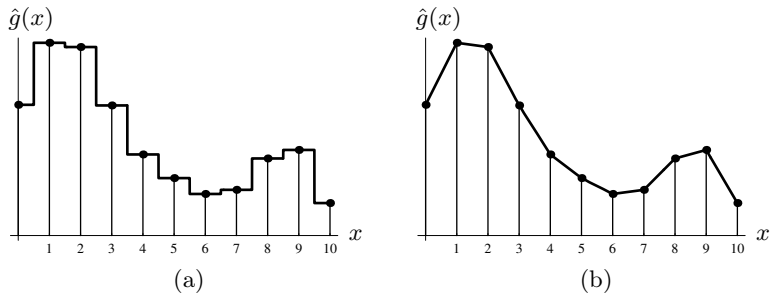


Fig. 16.12
Simple interpolation methods. The *nearest-neighbor interpolation* (a) simply selects the discrete sample $g(u)$ closest to the given continuous coordinate x as the interpolating value $\hat{g}(x)$. Under *linear interpolation* (b), the result is a piecewise linear function connecting adjacent samples $g(u)$ and $g(u+1)$.

$$\hat{g}(x) = g(u_0), \tag{16.44}$$

$$\text{where} \quad u_0 = \text{round}(x) = \lfloor x + 0.5 \rfloor. \tag{16.45}$$

A typical result of this so-called *nearest-neighbor interpolation* is shown in Fig. 16.12 (a).

Linear interpolation

Another simple method is *linear interpolation*. Here the estimated value is the sum of the two closest samples $g(u_0)$ and $g(u_0+1)$, with $u_0 = \lfloor x \rfloor$. The weight of each sample is proportional to its closeness to the continuous position x,

$$\begin{aligned} \hat{g}(x) &= g(u_0) + (x - u_0) \cdot \big(g(u_0+1) - g(u_0)\big) \\ &= g(u_0) \cdot \big(1 - (x - u_0)\big) + g(u_0+1) \cdot (x - u_0). \end{aligned} \tag{16.46}$$

As shown in Fig. 16.12 (b), the result is a piecewise linear function made up of straight line segments between consecutive sample values.

16.3.2 Ideal Interpolation

Obviously the results of these simple interpolation methods do not well approximate the original continuous function (Fig. 16.11). But how can we obtain a better approximation from the discrete samples only when the original function is unknown? This may appear hopeless at first,

because the discrete samples $g(u)$ could possibly originate from any continuous function $f(x)$ with identical values at the discrete sample positions.

We find an intuitive answer to this question (once again) by looking at the functions in the spectral domain. If the original function $f(x)$ was discretized in accordance with the *sampling theorem* (see Sec. 13.2.1), then $f(x)$ must have been "band limited"—it could not contain any signal components with frequencies higher than half the sampling frequency ω_s. This means that the reconstructed signal can only contain a limited set of frequencies and thus its trajectory between the discrete sample values is not arbitrary but naturally constrained.

In this context, absolute units of measure are of no concern since in a digital signal all frequencies relate to the sampling frequency. In particular, if we take $\tau_s = 1$ as the (unitless) sampling interval, the resulting sampling frequency is

$$\omega_s = 2\pi$$

and thus the maximum signal frequency is $\omega_{\max} = \frac{\omega_s}{2} = \pi$. To isolate the frequency range $-\omega_{\max} \dots \omega_{\max}$ in the corresponding (periodic) Fourier spectrum, we multiply the spectrum $G(\omega)$ by a square windowing function $\mathit{\Pi}_\pi(\omega)$ of width $\pm\omega_{\max} = \pm\pi$,

$$\hat{G}(\omega) \;=\; G(\omega) \cdot \mathit{\Pi}_\pi(\omega) \;=\; G(\omega) \cdot \begin{cases} 1 & \text{for } -\pi \le \omega \le \pi \\ 0 & \text{otherwise.} \end{cases}$$

This is called an *ideal low-pass filter*, which cuts off all signal components with frequencies greater than π and keeps all lower-frequency components unchanged. In the signal domain, the operation in Eqn. (16.47) corresponds (see Eqn. (13.28)) to a *linear convolution* with the inverse Fourier transform of the windowing function $\mathit{\Pi}_\pi(\omega)$, which is the *Sinc* function, defined as

$$\mathrm{Sinc}(x) = \begin{cases} 1 & \text{for } |x| = 0 \\ \frac{\sin(\pi x)}{\pi x} & \text{for } |x| > 0 \end{cases} \tag{16.47}$$

and shown in Fig. 16.13 (see also Table 13.1). This correspondence, which was already discussed in Sec. 13.1.6, between convolution in the signal domain and simple multiplication in the frequency domain is summarized in Fig. 16.14.

So theoretically $\mathrm{Sinc}(x)$ is the ideal interpolation function for reconstructing a frequency-limited continuous signal. To compute the interpolated value for the discrete function $g(u)$ at an arbitrary position x_0, the Sinc function is shifted to x_0 (such that its origin lies at x_0), multiplied with all sample values $g(u)$, with $u \in \mathbb{Z}$, and the results are summed—i. e., $g(u)$ and $\mathrm{Sinc}(x)$ are *convolved*. The reconstructed value of the continuous function at position x_0 is thus

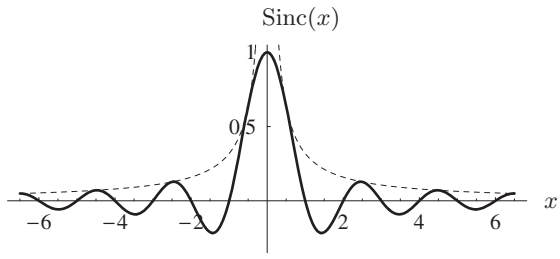


Fig. 16.13
Sinc function in 1D. The function $\mathrm{Sinc}(x)$ has the value 1 at the origin and zero values at all integer positions. The dashed line plots the amplitude $|\frac{1}{\pi x}|$ of the underlying sine function.

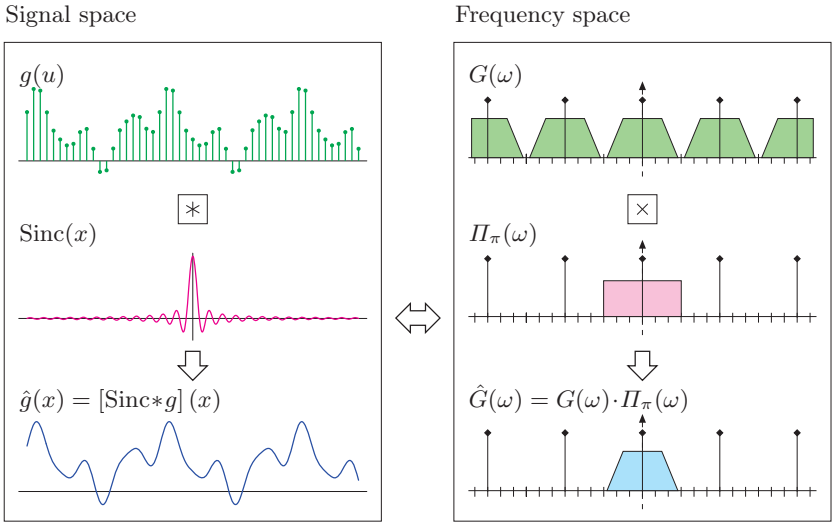


Fig. 16.14
Interpolation of a discrete signal—relation between signal and frequency space. The discrete signal $g(u)$ in signal space (left) corresponds to the periodic Fourier spectrum $G(\omega)$ in frequency space (right). The spectrum $\hat{G}(\omega)$ of the continuous signal is isolated from $G(\omega)$ by pointwise multiplication ($\times$) with the square function $\Pi_\pi(\omega)$, which constitutes an ideal low-pass filter (right). In signal space (left), this operation corresponds to a linear convolution ($*$) with the function $\mathrm{Sinc}(x)$.

$$\hat{g}(x_0) = [\mathrm{Sinc} * g]\,(x_0) = \sum_{u=-\infty}^{\infty} \mathrm{Sinc}(x_0 - u) \cdot g(u), \tag{16.48}$$

where $*$ is the linear convolution operator (see Sec. 6.3.1). If the discrete signal $g(u)$ is *finite* with length N (as is usually the case), it is assumed to be *periodic* (i. e., $g(u) = g(u + kN)$ for all $k \in \mathbb{Z}$).[4] In this case, Eqn. (16.48) modifies to

$$\hat{g}(x_0) = \sum_{u=-\infty}^{\infty} \mathrm{Sinc}(x_0 - u) \cdot g(u \bmod N). \tag{16.49}$$

It may be surprising that the ideal interpolation of a discrete function $g(u)$ at a position x_0 apparently involves not only a few neighboring sample points but in general *infinitely many* values of $g(u)$ whose weights decrease continuously with their distance from the given interpolation point x_0 (at the rate $|\frac{1}{\pi(x_0-u)}|$).

[4] This assumption is explained by the fact that a discrete Fourier spectrum implicitly corresponds to a periodic signal (also see Sec. 13.2.2).

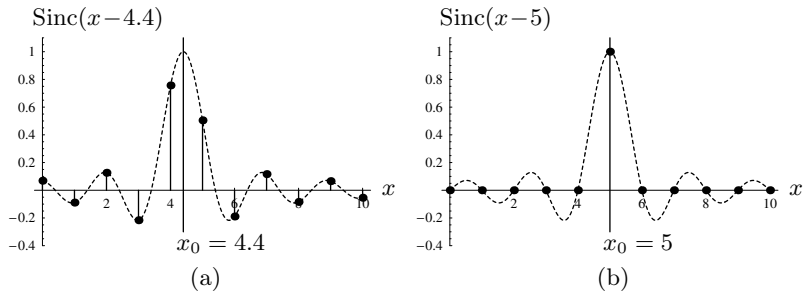


Fig. 16.15
Interpolation by convolving with the Sinc function. The Sinc function is shifted by aligning its origin with the interpolation points $x_0 = 4.4$ (a) and $x_0 = 5$ (b). The values of the shifted Sinc function (dashed curve) at the integral positions are the weights (coefficients) for the corresponding sample values $g(u)$.

Figure 16.15 shows two examples for interpolating the function $g(u)$ at positions $x_0 = 4.4$ and $x_0 = 5$. If the function is interpolated at some integral position, such as $x_0 = 5$, the sample $g(u)$ at $u = x_0$ receives the weight 1, while all other samples coincide with the zero positions of the Sinc function and are thus ignored. Consequently, the resulting interpolation values $\hat{g}(x)$ are identical to the sample values $g(u)$ at all integral positions $x = u$.

If a continuous signal is properly frequency limited (by half the sampling frequency $\frac{\omega_s}{2}$), it can be exactly reconstructed from the discrete signal by interpolation with the Sinc function, as Fig. 16.16 (a) demonstrates. Problems occur, however, around local high-frequency signal events, such as rapid transitions or pulses, as shown in Fig. 16.16 (b, c). In those situations, the Sinc interpolation causes strong overshooting or "ringing" artifacts, which are perceived as visually disturbing. For practical applications, the Sinc function is therefore not suitable as an interpolation kernel—not only because of its infinite extent (and the resulting noncomputability).

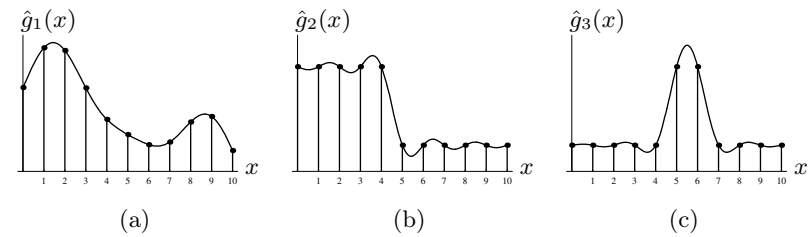


Fig. 16.16
Sinc interpolation on various signal types. The reconstructed function in (a) is identical to the continuous, band-limited original. The results for the step function (b) and the pulse function (c) show the strong ringing caused by Sinc (ideal low-pass) interpolation.

A useful interpolation function implements a low-pass filter that on the one hand introduces minimal blurring by maintaining the maximum the signal bandwidth but also delivers a good reconstruction at rapid signal transitions on the other hand. In this regard, the Sinc function is an extreme choice—it implements an ideal low-pass filter and thus preserves a maximum bandwidth and signal continuity but gives inferior results at signal transitions. At the opposite extreme, nearest-neighbor interpolation (Fig. 16.12) can perfectly handle steps and pulses but generally fails to produce a continuous signal reconstruction between sample

points. The design of an interpolation function thus always involves a trade-off, and the quality of the results often depends on the particular application and subjective judgment. In the following, we discuss some common interpolation functions that come close to this goal and are therefore frequently used in practice.

16.3.3 Interpolation by Convolution

As we saw earlier in the context of Sinc interpolation (Eqn. (16.47)), the reconstruction of a continuous signal can be described as a linear convolution operation. In general, we can express interpolation as a convolution of the given discrete function $g(u)$ with some continuous *interpolation kernel* $w(x)$ as

$$\hat{g}(x_0) = [w * g](x_0) = \sum_{u=-\infty}^{\infty} w(x_0 - u) \cdot g(u). \tag{16.50}$$

The Sinc interpolation in Eqn. (16.48) is obviously only a special case with $w(x) = \mathrm{Sinc}(x)$. Similarly, the one-dimensional *nearest-neighbor interpolation* (Eqn. (16.45), Fig. 16.12 (a)) can be expressed as a linear convolution with the kernel

$$w_{\mathrm{nn}}(x) = \begin{cases} 1 & \text{for } -0.5 \le x < 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{16.51}$$

and the *linear interpolation* (Eqn. (16.46), Fig. 16.12 (b)) with the kernel

$$w_{\mathrm{lin}}(x) = \begin{cases} 1 - x & \text{for } |x| < 1 \\ 0 & \text{for } |x| \ge 1. \end{cases} \tag{16.52}$$

The interpolation kernels $w_{\mathrm{nn}}(x)$ and $w_{\mathrm{lin}}(x)$ are both shown in Fig. 16.17, and sample results for various function types are plotted in Fig. 16.18.

16.3.4 Cubic Interpolation

Because of its infinite extent and the ringing artifacts caused by its slowly decaying oscillations, the Sinc function is not a useful interpolation kernel in practice. Therefore, several interpolation methods employ a truncated

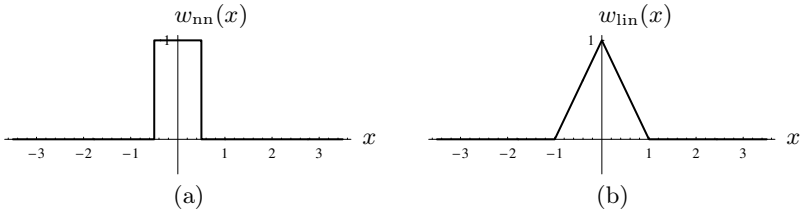


Fig. 16.17
Convolution kernels for nearest-neighbor interpolation $w_{\mathrm{nn}}(x)$ and linear interpolation $w_{\mathrm{lin}}(x)$.

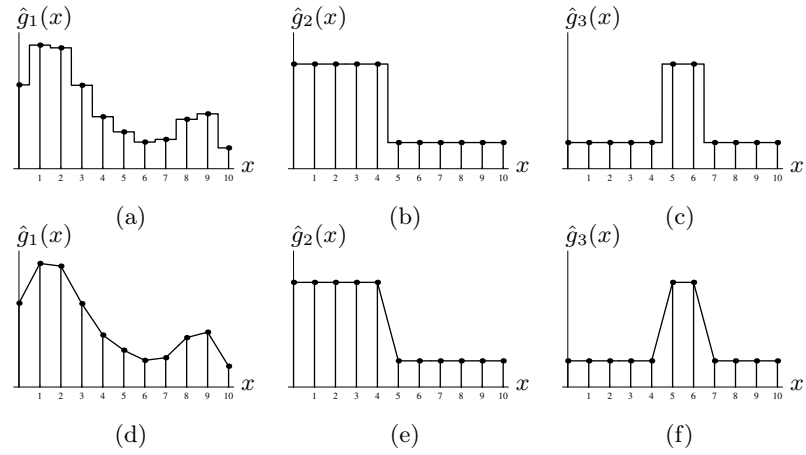


Fig. 16.18
Interpolation examples: nearest-neighbor interpolation (a–c), linear interpolation (d–f).

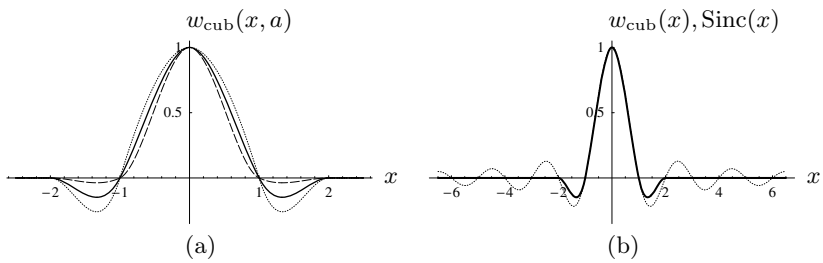


Fig. 16.19
Cubic interpolation kernel. Function $w_{\text{cub}}(x,a)$ with control parameter a set to $a = 0.25$ (dashed curve), $a = 1$ (continuous curve), and $a = 1.75$ (dotted curve) (a). Cubic function $w_{\text{cub}}(x)$ and Sinc function compared (b).

version of the Sinc function or an approximation of it, thereby making the convolution kernel more compact and reducing the ringing. A frequently used approximation of a truncated Sinc function is the so-called cubic interpolation, whose convolution kernel is defined as the piecewise cubic polynomial

$$w_{\text{cub}}(x,a) = \begin{cases} (-a+2)\cdot|x|^3 + (a-3)\cdot|x|^2 + 1 & \text{for } 0 \le |x| < 1 \\ -a\cdot|x|^3 + 5a\cdot|x|^2 - 8a\cdot|x| + 4a & \text{for } 1 \le |x| < 2 \\ 0 & \text{for } |x| \ge 2\,. \end{cases} \tag{16.53}$$

The single control parameter a can be used to adjust the slope of this spline[5] function (Fig. 16.19), which affects the amount of overshoot and thus the perceived "sharpness" of the interpolated signal. For $a = 1$, which is often recommended as a standard setting, Eqn. (16.53) simplifies to

$$w_{\text{cub}}(x) = \begin{cases} |x|^3 - 2\cdot|x|^2 + 1 & \text{for } 0 \le |x| < 1 \\ -|x|^3 + 5\cdot|x|^2 - 8\cdot|x| + 4 & \text{for } 1 \le |x| < 2 \\ 0 & \text{for } |x| \ge 2\,. \end{cases} \tag{16.54}$$

[5] The family of functions described by Eqn. (16.53) are commonly referred to as *cardinal splines* [10] (see also Sec. 16.3.5).

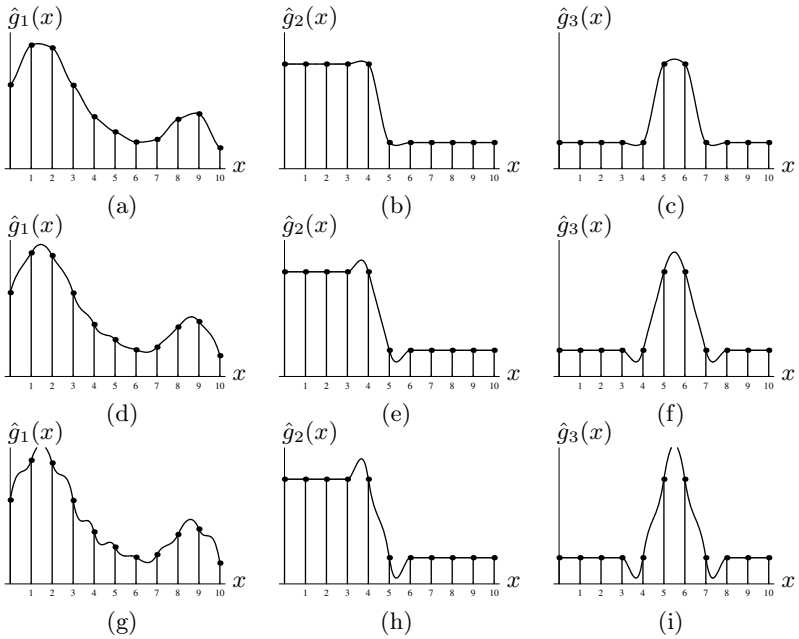


Fig. 16.20
Cubic interpolation examples. Parameter a in Eqn. (16.53) controls the amount of signal overshoot or perceived sharpness: $a = 0.25$ (a–c), standard setting $a = 1$ (d–f), $a = 1.75$ (g–i). Notice in (d) the ripple effects incurred by interpolating with the standard settings in smooth signal regions.

Figure 16.20 shows the results of cubic interpolation with different settings of the control parameter a. Notice that the cubic reconstruction obtained with the popular standard setting ($a = 1$) exhibits substantial overshooting at edges as well as strong ripple effects in the continuous parts of the signal (Fig. 16.20 (d)). With $a = 0.5$, the expression in Eqn. (16.53) corresponds to a *Catmull-Rom* spline [21] (see also Sec. 16.3.5), which produces significantly better results than the standard setup (with $a = 1$), particularly in smooth signal regions (see Fig. 16.22 (a–c)).

In contrast to the Sinc function, the cubic interpolation kernel $w_{\text{cub}}(x)$ has a very small extent and is therefore efficient to compute (Fig. 16.19 (b)). Since $w_{\text{cub}}(x, a) = 0$ for $|x| \geq 2$, only *four* discrete values $g(u)$ need to be accounted for in the convolution operation (Eqn. (16.50)) at any continuous position $x_0 \in \mathbb{R}$,

$$g(u_0-1),\ g(u_0),\ g(u_0+1),\ g(u_0+2), \quad \text{where}\ \ u_0 = \lfloor x_0 \rfloor.$$

This reduces the one-dimensional cubic interpolation to computing the expression

$$\hat{g}(x_0) = \sum_{u=\lfloor x_0 \rfloor -1}^{\lfloor x_0 \rfloor +2} w_{\text{cub}}(x_0 - u, a) \cdot g(u). \tag{16.55}$$

16.3.5 Spline Interpolation

The cubic interpolation kernel (Eqn. (16.53)) described in the previous section is a piecewise cubic polynomial function, also known as a *cubic*

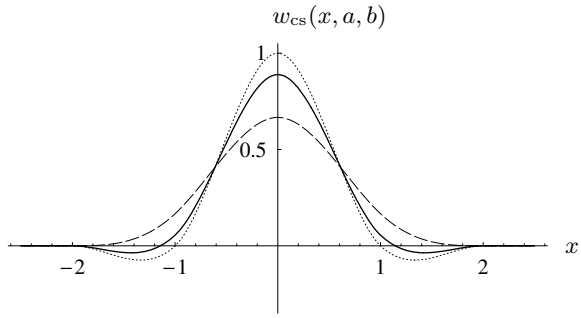


Fig. 16.21
Examples of cardinal spline functions $w_{cs}(x, a, b)$ as specified by Eqn. (16.56): *Catmull-Rom* spline $w_{cs}(x, 0.5, 0)$ (dotted line), *cubic B-spline* $w_{cs}(x, 0, 1)$ (dashed line), and *Mitchell-Netravali* function $w_{cs}(x, \frac{1}{3}, \frac{1}{3})$ (solid line).

spline in computer graphics. In its general form, this function takes not only one but *two* control parameters (a, b) [72],[6]

$$w_{cs}(x, a, b) = \frac{1}{6} \cdot \begin{cases} (-6a - 9b + 12) \cdot |x|^3 \\ \quad + (6a + 12b - 18) \cdot |x|^2 - 2b + 6 & \text{for } 0 \le |x| < 1 \\ (-6a - b) \cdot |x|^3 + (30a + 6b) \cdot |x|^2 \\ \quad + (-48a - 12b) \cdot |x| + 24a + 8b & \text{for } 1 \le |x| < 2 \\ 0 & \text{for } |x| \ge 2\,. \end{cases} \tag{16.56}$$

Equation (16.56) describes a family of C2-continuous functions; i. e., their first and second derivatives are continuous everywhere and thus their trajectories exhibit no discontinuities, corners, or abrupt changes of curvature. For $b = 0$, the function $w_{cs}(x, a, b)$ specifies a one-parameter family of so-called *cardinal splines* equivalent to the cubic interpolation function $w_{cub}(x, a)$ in Eqn. (16.53),

$$w_{cs}(x, a, 0) = w_{cub}(x, a),$$

and for the standard setting $a = 1$ (Eqn. (16.54)) in particular

$$w_{cs}(x, 1, 0) = w_{cub}(x, 1) = w_{cub}(x).$$

Figure 16.21 shows three additional examples of this function type that are important in the context of interpolation: *Catmull-Rom* splines, *cubic B-splines*, and the *Mitchell-Netravali* function. All three functions are briefly described below. The actual computation of the interpolated signal follows exactly the same scheme as used for the cubic interpolation described in Eqn. (16.55).

Catmull-Rom interpolation

With the control parameters set to $a = 0.5$ and $b = 0$, the function in Eqn. (16.56) is a *Catmull-Rom spline* [21], as already mentioned in Sec. 16.3.4:

[6] In [72], the parameters a and b were originally named C and B, respectively, with $B \equiv b$ and $C \equiv a$.

$$w_{\text{crm}}(x) = w_{\text{cs}}(x, 0.5, 0)$$
$$= \frac{1}{2} \cdot \begin{cases} 3 \cdot |x|^3 - 5 \cdot |x|^2 + 2 & \text{for } 0 \le |x| < 1 \\ -|x|^3 + 5 \cdot |x|^2 - 8 \cdot |x| + 4 & \text{for } 1 \le |x| < 2 \\ 0 & \text{for } |x| \ge 2\,. \end{cases} \tag{16.57}$$

Examples of signals interpolated with this kernel are shown in Fig. 16.22 (a–c). The results are similar to ones produced by cubic interpolation (with $a = 1$, see Fig. 16.20) with regard to sharpness, but the Catmull-Rom reconstruction is clearly superior in smooth signal regions (compare, e. g., Fig. 16.20 (d) vs. Fig. 16.22 (a)).

Cubic B-spline approximation

With parameters set to $a = 0$ and $b = 1$, Eqn. (16.56) corresponds to a cubic B-spline function [10] of the form

$$w_{\text{cbs}}(x) = w_{\text{cs}}(x, 0, 1)$$
$$= \frac{1}{6} \cdot \begin{cases} 3 \cdot |x|^3 - 6 \cdot |x|^2 - 4 & \text{for } 0 \le |x| < 1 \\ -|x|^3 + 6 \cdot |x|^2 - 12 \cdot |x| + 8 & \text{for } 1 \le |x| < 2 \\ 0 & \text{for } |x| \ge 2\,. \end{cases} \tag{16.58}$$

This function is positive everywhere and, when used as an interpolation kernel, causes a pure smoothing effect similar to a Gaussian smoothing filter (see Fig. 16.22 (d–f)). Notice also that—in contrast to all previously described interpolation methods—the reconstructed function does *not* pass through all discrete sample points. Thus, to be precise, the reconstruction with cubic B-splines is not called an *interpolation* but an *approximation* of the signal.

Mitchell-Netravali approximation

The design of an optimal interpolation kernel is always a trade-off between high bandwidth (sharpness) and good transient response (low ringing). Catmull-Rom interpolation, for example, emphasizes high sharpness, whereas cubic B-spline interpolation blurs but creates no ringing. Based on empirical tests, Mitchell and Netravali [72] proposed a cubic interpolation kernel as described in Eqn. (16.56) with parameter settings $a = \frac{1}{3}$ and $b = \frac{1}{3}$, and the resulting interpolation function

$$w_{\text{mn}}(x) = w_{\text{cs}}\left(x, \tfrac{1}{3}, \tfrac{1}{3}\right)$$
$$= \frac{1}{18} \cdot \begin{cases} 21 \cdot |x|^3 - 36 \cdot |x|^2 + 16 & \text{for } 0 \le |x| < 1 \\ -7 \cdot |x|^3 + 36 \cdot |x|^2 - 60 \cdot |x| + 32 & \text{for } 1 \le |x| < 2 \\ 0 & \text{for } |x| \ge 2. \end{cases} \tag{16.59}$$

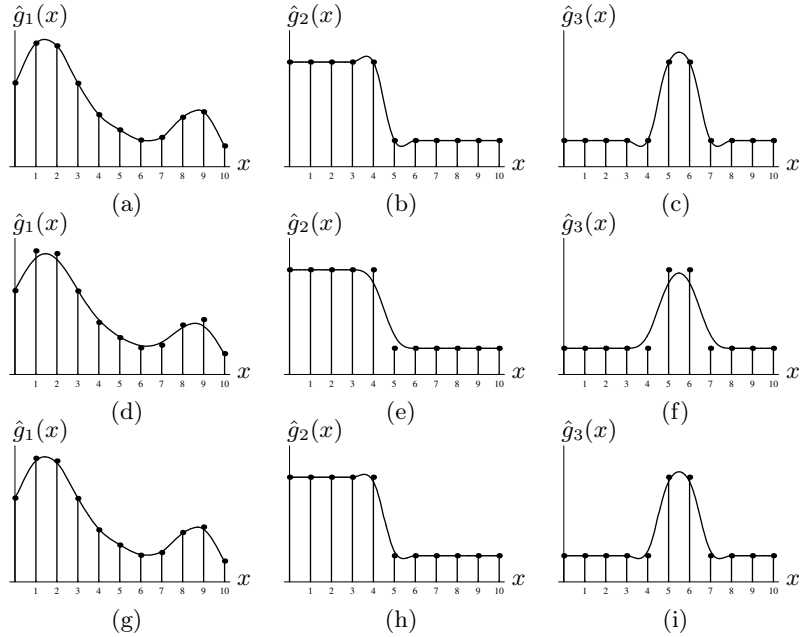


Fig. 16.22
Cardinal spline reconstruction examples: *Catmull-Rom* interpolation (a–c), *cubic B-spline* approximation (d–f), and *Mitchell-Netravali* approximation (g–i).

This function is the weighted sum of a Catmull-Rom spline (Eqn. (16.57)) and a cubic B-spline (Eqn. (16.58)), as is apparent in Fig. 16.21.[7] The examples in Fig. 16.22 (g–i) show that this method is a good compromise, producing little overshoot, high edge sharpness, and good signal continuity in smooth regions. Since the resulting function does not pass through the original sample points, the Mitchell-Netravali method is again an *approximation* and not an interpolation.

16.3.6 Lanczos Interpolation

The Lanczos[8] interpolation belongs to the family of "windowed Sinc" methods. In contrast to the methods described in the previous sections, these do *not* use a polynomial (or other) approximation of the Sinc function but the Sinc function *itself* combined with a suitable window function $\psi(x)$; i. e., an interpolation kernel of the form

$$w(x) = \psi(x) \cdot \mathrm{Sinc}(x). \tag{16.60}$$

The particular window functions for the Lanczos interpolation are defined as

$$\psi_{\mathrm{L}n}(x) = \begin{cases} 1 & \text{for } |x| = 0 \\ \frac{\sin(\pi \frac{x}{n})}{\pi \frac{x}{n}} & \text{for } 0 < |x| < n \\ 0 & \text{for } |x| \geq n\,, \end{cases} \tag{16.61}$$

[7] See also Exercise 16.5.
[8] Cornelius Lanczos (1893–1974).

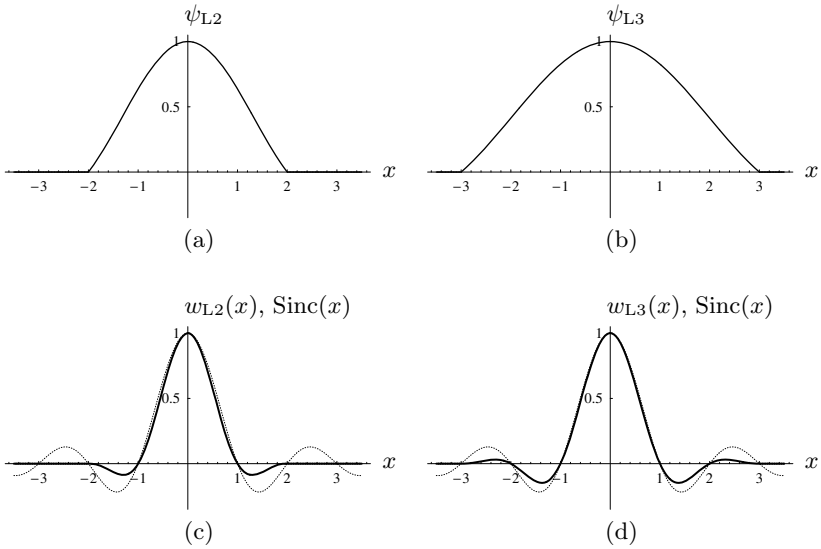


Fig. 16.23
One-dimensional Lanczos interpolation kernels. Lanczos window functions ψ_{L2} (a), ψ_{L3} (b), and the corresponding interpolation kernels w_{L2} (c), w_{L3} (d). The original Sinc function (dotted curve) is shown for comparison.

where $n \in \mathbb{N}$ denotes the *order* of the filter [76, 100]. Notice that the window function is again a truncated Sinc function! For the Lanczos filters of order $n = 2, 3$, which are the most commonly used in image processing, the corresponding window functions are

$$\psi_{L2}(x) = \begin{cases} 1 & \text{for } |x| = 0 \\ \frac{\sin(\pi \frac{x}{2})}{\pi \frac{x}{2}} & \text{for } 0 < |x| < 2 \\ 0 & \text{for } |x| \geq 2\,, \end{cases} \tag{16.62}$$

$$\psi_{L3}(x) = \begin{cases} 1 & \text{for } |x| = 0 \\ \frac{\sin(\pi \frac{x}{3})}{\pi \frac{x}{3}} & \text{for } 0 < |x| < 3 \\ 0 & \text{for } |x| \geq 3. \end{cases} \tag{16.63}$$

Both functions are shown in Fig. 16.23 (a, b). The corresponding one-dimensional interpolation kernels w_{L2} and w_{L3}, respectively, are obtained by multiplying the window function by the Sinc function (Eqns. (16.47) and (16.61)) as

$$\begin{aligned} w_{L2}(x) &= \psi_{L2}(x) \cdot \text{Sinc}(x) \\ &= \begin{cases} 1 & \text{for } |x| = 0 \\ 2 \cdot \frac{\sin(\pi \frac{x}{2}) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 < |x| < 2 \\ 0 & \text{for } |x| \geq 2 \end{cases} \end{aligned} \tag{16.64}$$

and

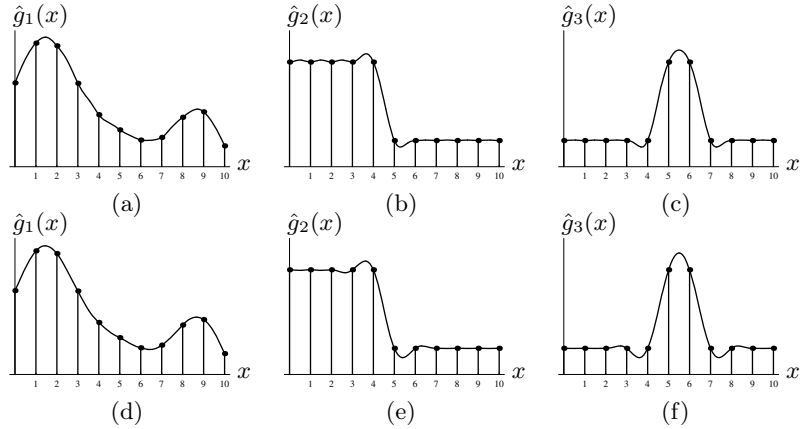


Fig. 16.24 Lanczos interpolation examples: Lanczos-2 (a–c), Lanczos-3 (d–f).

$$w_{\mathrm{L3}}(x) = \psi_{\mathrm{L3}}(x) \cdot \mathrm{Sinc}(x)$$
$$= \begin{cases} 1 & \text{for } |x| = 0 \\ 3 \cdot \frac{\sin(\pi\frac{x}{3}) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 < |x| < 3 \\ 0 & \text{for } |x| \geq 3\,. \end{cases} \tag{16.65}$$

Figure 16.23 (c, d) shows the resulting interpolation kernels together with the original Sinc function. The function $w_{\mathrm{L2}}(x)$ is quite similar to the Catmull-Rom kernel $w_{\mathrm{crm}}(x)$ (Eqn. (16.57), Fig. 16.21), so the results can be expected to be similar as well, as shown in Fig. 16.24 (a–c) (cf. Fig. 16.22 (a–c)). Notice, however, the relatively poor reconstruction in the smooth signal regions (Fig. 16.24 (a)) and the bumps introduced in the constant high-amplitude regions (Fig. 16.24 (b)). The "3-tap" kernel $w_{\mathrm{L3}}(x)$ reduces these artifacts but at the cost of additional overshoot and ringing at edges (Fig. 16.22 (d–f)).

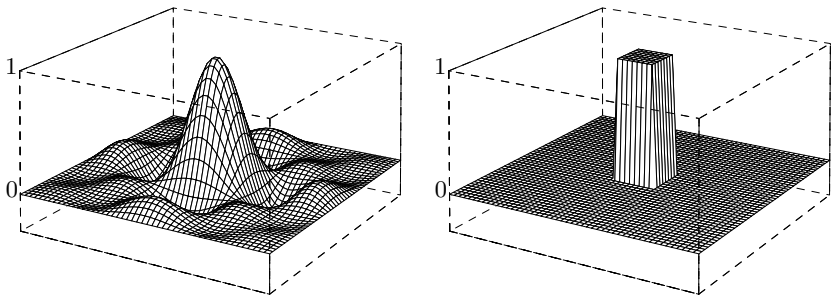
In summary, although Lanczos filters have seen revived interest and popularity in recent years, they do not seem to offer much (if any) advantage over other established methods, particularly the cubic, Catmull-Rom, or Mitchell-Netravali interpolations.

16.3.7 Interpolation in 2D

So far we have only looked at interpolating (or reconstructing) *one-dimensional* signals from discrete samples. Images are *two*-dimensional signals but, as we shall see in this section, the techniques for interpolating images are very similar and can be derived from the one-dimensional approach. In particular, "ideal" (low-pass filter) interpolation requires a two-dimensional Sinc function defined as

$$\textsc{Sinc}(x, y) = \mathrm{Sinc}(x) \cdot \mathrm{Sinc}(y) = \frac{\sin(\pi x)}{\pi x} \cdot \frac{\sin(\pi y)}{\pi y}, \tag{16.66}$$

which is shown in Fig. 16.25 (a). Just as in 1D, the 2D Sinc function is not a practical interpolation function for various reasons. In the following, we



(a) (b)

Fig. 16.25
Interpolation kernels in 2D: Sinc kernel $\text{Sinc}(x, y)$ (a) and nearest-neighbor kernel $W_{\text{nn}}(x, y)$ (b) for $-3 \le x, y \le 3$.

look at some common interpolation methods for images, particularly the nearest-neighbor, bilinear, bicubic, and Lanczos interpolations, whose 1D versions were described in the previous sections.

Nearest-neighbor interpolation in 2D

The pixel closest to a given continuous point (x_0, y_0) is found by rounding the x and y coordinates independently to integral values,

$$\begin{aligned} & \hat{I}(x_0, y_0) = I(u_0, v_0), \\ \text{with} \quad & u_0 = \text{round}(x_0) = \lfloor x_0 + 0.5 \rfloor, \\ & v_0 = \text{round}(y_0) = \lfloor y_0 + 0.5 \rfloor. \end{aligned} \tag{16.67}$$

As in the 1D case, the interpolation in 2D can be described as a linear convolution (linear filter). The 2D kernel for the nearest-neighbor interpolation is, analogous to Eqn. (16.51), defined as

$$W_{\text{nn}}(x, y) = \begin{cases} 1 & \text{for } -0.5 \le x, y < 0.5 \\ 0 & \text{otherwise.} \end{cases} \tag{16.68}$$

This function is shown in Fig. 16.25 (b). Nearest-neighbor interpolation is known for its strong blocking effects (Fig. 16.26 (b)) and thus is rarely used for geometric image operations. However, in some situations, this effect may be intended; for example, if an image is to be enlarged by replicating each pixel without any smoothing.

Bilinear interpolation

The 2D counterpart to the linear interpolation (Sec. 16.3.1) is the so-called *bilinear* interpolation,[9] whose operation is illustrated in Fig. 16.27. For the given interpolation point (x_0, y_0), we first find the four closest (surrounding) pixels A, B, C, D in the image I with

[9] Not to be confused with the bilinear *mapping* (transformation) described in Sec. 16.1.5.

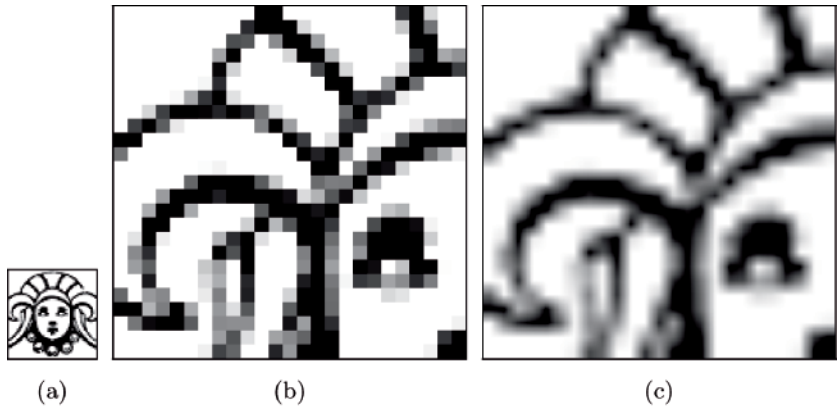


Fig. 16.26
Image enlargement (8×): original (a), nearest-neighbor interpolation (b), and bilinear interpolation (c).

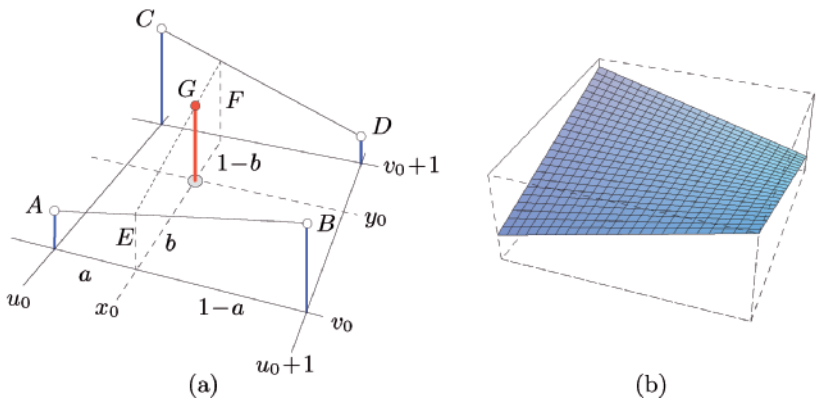


Fig. 16.27
Bilinear interpolation. For a given position (x_0, y_0), the interpolated value is computed from the values A, B, C, D of the four closest pixels in two steps (a). First the intermediate values E and F are computed by linear interpolation in the horizontal direction between A, B and C, D, respectively, where $a = x_0 - u_0$ is the distance to the nearest pixel to the left of x_0. Subsequently, the intermediate values E, F are interpolated in the vertical direction, where $b = y_0 - v_0$ is the distance to the nearest pixel below y_0. An example for the resulting surface between four adjacent pixels is shown in (b).

$$\begin{aligned} A &= I(u_0, v_0), & B &= I(u_0+1, v_0), \\ C &= I(u_0, v_0+1), & D &= I(u_0+1, v_0+1), \end{aligned} \tag{16.69}$$

where $u_0 = \lfloor x_0 \rfloor$ and $v_0 = \lfloor y_0 \rfloor$. Then the pixel values A, B, C, D are interpolated in horizontal and subsequently in vertical direction. The intermediate values E, F are determined by the distance $a = x_0 - u_0$ between the interpolation point (x_0, y_0) and the horizontal raster coordinate u_0 as

$$\begin{aligned} E &= A + (x_0 - u_0) \cdot (B - A) = A + a \cdot (B - A), \\ F &= C + (x_0 - u_0) \cdot (D - C) = C + a \cdot (D - C), \end{aligned}$$

and the final interpolation value G is computed from the vertical distance $b = y_0 - v_0$ as

$$\begin{aligned} \hat{I}(x_0, y_0) &= G = E + (y_0 - v_0) \cdot (F - E) = E + b \cdot (F - E) \\ &= (a-1)(b-1)\, A + a(1-b)\, B + (1-a)\, b\, C + a\, b\, D. \end{aligned} \tag{16.70}$$

Expressed as a linear convolution filter, the corresponding 2D kernel $W_{\text{bil}}(x, y)$ is the product of the two one-dimensional kernels $w_{\text{lin}}(x)$ and

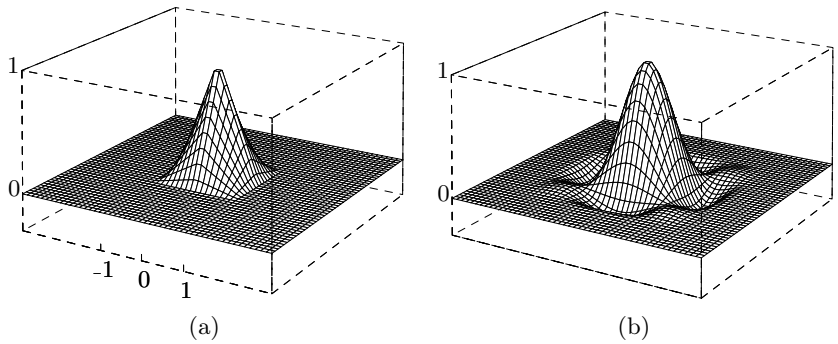


Fig. 16.28
2D interpolation kernels: bilinear kernel $W_{\text{bil}}(x, y)$ (a) and bicubic kernel $W_{\text{bic}}(x, y)$ (b) for $-3 \leq x, y \leq 3$.

$w_{\text{lin}}(y)$ (Eqn. (16.52)),

$$W_{\text{bil}}(x,y) = w_{\text{lin}}(x) \cdot w_{\text{lin}}(y) = \begin{cases} 1 - x - y - x \cdot y & \text{for } 0 \leq |x|, |y| < 1 \\ 0 & \text{otherwise.} \end{cases} \tag{16.71}$$

In this function (plotted in Fig. 16.28 (a)), we can recognize the bilinear term that gives this method its name.

Bicubic interpolation

The convolution kernel for the two-dimensional cubic interpolation is also defined as the product of the corresponding one-dimensional kernels (Eqn. (16.54)),

$$W_{\text{bic}}(x,y) = w_{\text{cub}}(x) \cdot w_{\text{cub}}(y). \tag{16.72}$$

The resulting kernel is plotted in Fig. 16.28 (b). Due to the decomposition into one-dimensional kernels (Eqn. (16.72)), the computation of the bicubic interpolation is *separable* in x, y and can thus be expressed as

$$\begin{aligned} \hat{I}(x_0,y_0) &= \sum_{v=\lfloor y_0 \rfloor - 1}^{\lfloor y_0 \rfloor + 2} \left[\sum_{u=\lfloor x_0 \rfloor - 1}^{\lfloor x_0 \rfloor + 2} I(u,v) \cdot W_{\text{bic}}(x_0 - u, y_0 - v) \right] \\ &= \sum_{j=0}^{3} \left[w_{\text{cub}}(y_0 - v_j) \cdot \underbrace{\sum_{i=0}^{3} I(u_i, v_j) \cdot w_{\text{cub}}(x_0 - u_i)}_{p_j} \right], \end{aligned} \tag{16.73}$$

with $u_i = \lfloor x_0 \rfloor - 1 + i$ and $v_j = \lfloor y_0 \rfloor - 1 + j$. The value p_j denotes the intermediate result of the cubic interpolation in the x direction in line j, as illustrated in Fig. 16.29. Equation (16.73) describes a simple and efficient procedure for computing the bicubic interpolation using only a one-dimensional kernel $w_{\text{cub}}(x)$. The interpolation is based on a 4×4 neighborhood of pixels and requires a total of $16 + 4 = 20$ additions and multiplications.

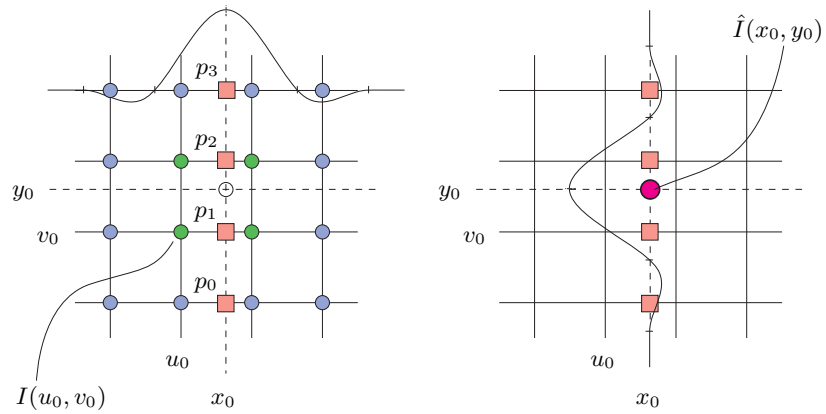


Fig. 16.29
Bicubic interpolation in two steps. The discrete image I (pixels are marked ○) is to be interpolated at some continuous position (x_0, y_0). In step 1 (left), a one-dimensional interpolation is performed in the horizontal direction with $w_{\text{cub}}(x)$ over four pixels $I(u_i, v_j)$ in four lines. One intermediate result p_j (marked □) is computed for each line j. In step 2 (right), the result $\hat{I}(x_0, y_0)$ is computed by a single cubic interpolation in the vertical direction over the intermediate results $p_0 \ldots p_3$.

Algorithm 16.2
Bicubic interpolation of image I at position (x_0, y_0). The one-dimensional cubic function $w_{\text{cub}}(\cdot)$ (Eqn. (16.53)) is used for the separate interpolation in the x and y directions based on a neighborhood of 4×4 pixels.

```
1:  BicubicInterpolation (I, x0, y0)                    ▷ x0, y0 ∈ ℝ
        Returns the interpolated value of the image I at the continuous po-
        sition (x0, y0).
2:      q ← 0
3:      for j ← 0 ... 3 do                              ▷ iterate over 4 lines
4:          v ← ⌊y0⌋ − 1 + j
5:          p ← 0
6:          for i ← 0 ... 3 do                          ▷ iterate over 4 columns
7:              u ← ⌊x0⌋ − 1 + i
8:              p ← p + I(u, v) · w_cub(x0 − u)
9:          q ← q + p · w_cub(y0 − v)
10:     return q.
```

This method, which is summarized in Alg. 16.2, can be used to implement any x/y-separable 2D interpolation kernel of size 4×4, such as the two-dimensional *Catmull-Rom* interpolation (Eqn. (16.57)) with

$$W_{\text{crm}}(x, y) = w_{\text{crm}}(x) \cdot w_{\text{crm}}(y) \tag{16.74}$$

or the *Mitchell-Netravali* interpolation (Eqn. (16.59)) with

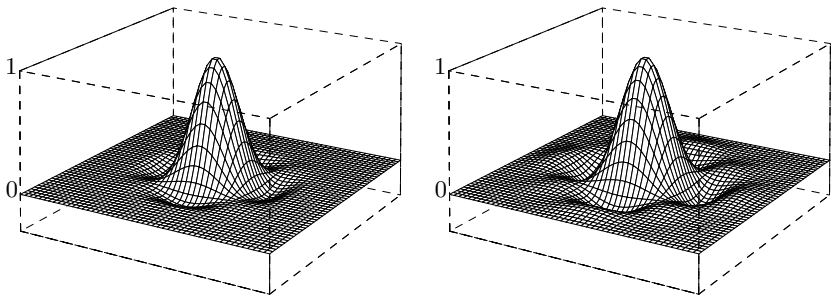
$$W_{\text{mn}}(x, y) = w_{\text{mn}}(x) \cdot w_{\text{mn}}(y). \tag{16.75}$$

Lanczos interpolation

The kernels for the 2D Lanczos interpolation are also x/y-separable into one-dimensional kernels (Eqns. (16.64) and (16.65)),

$$W_{\text{Ln}}(x, y) = w_{\text{Ln}}(x) \cdot w_{\text{Ln}}(y). \tag{16.76}$$

The resulting kernels for $n = 2$ and $n = 3$ are shown in Fig. 16.30. Because of the separability the 2D Lanczos interpolation can be com-



(a) (b)

Fig. 16.30
Two-dimensional Lanczos kernels for $n = 2$ and $n = 3$: kernels $W_{\mathrm{L}2}(x, y)$ (a) and $W_{\mathrm{L}3}(x, y)$ (b), with $-3 \le x, y \le 3$.

puted, similar to the bicubic interpolation, separately in the x and y directions. Like the bicubic kernel, the 2-tap Lanczos kernel $W_{\mathrm{L}2}$ (Eqn. (16.64)) is *zero* outside the interval $-2 \le x, y \le 2$, and thus the procedure described in Eqn. (16.73) and Alg. 16.2 can be used without modification. The 3-tap kernel $W_{\mathrm{L}3}$ (Eqn. (16.65)) requires two additional rows and columns, and thus the 2D interpolation changes to

$$\begin{aligned}\hat{I}(x_0, y_0) &= \sum_{\substack{v=\\ \lfloor y_0 \rfloor - 2}}^{\lfloor y_0 \rfloor + 3} \left[\sum_{\substack{u=\\ \lfloor x_0 \rfloor - 2}}^{\lfloor x_0 \rfloor + 3} I(u, v) \cdot W_{\mathrm{L}3}(x_0 - u, y_0 - v) \right] \\ &= \sum_{j=0}^{5} \left[w_{\mathrm{L}3}(y_0 - v_j) \cdot \sum_{i=0}^{5} I(u_i, v_j) \cdot w_{\mathrm{L}3}(x_0 - u_i) \right], \qquad (16.77)\end{aligned}$$

$$\text{with} \quad u_i = \lfloor x_0 \rfloor + i - 2 \quad \text{and} \quad v_j = \lfloor y_0 \rfloor + j - 2.$$

Thus, the L3 Lanczos interpolation in 2D uses a support region of $6 \times 6 = 36$ pixels from the original image, 20 pixels more than the bicubic interpolation.

Examples and discussion

Figure 16.31 compares the three most common interpolation methods: nearest-neighbor, bilinear, and bicubic interpolation. The original image, consisting of black lines on a gray background, is rotated counterclockwise by 15°.

Nearest-neighbor interpolation (Fig. 16.31 (b)) creates no new pixel values but forms, as expected, coarse blocks of pixels with the same intensity. The effect of the *bilinear* interpolation (Fig. 16.31 (c)) is local smoothing over four neighboring pixels. The weights for these four pixels are positive, and thus no result can be smaller than the smallest neighboring pixel value or greater than the greatest neighboring pixel value. In other words, bilinear interpolation cannot create any over- or undershoot at edges. This is not the case for the *bicubic* interpolation (Fig. 16.31 (d)): some of the coefficients in the bicubic interpolation

kernel are negative, which makes pixels near edges clearly brighter or darker, respectively, thus increasing the perceived sharpness. In general, bicubic interpolation produces clearly better results than the bilinear method at comparable computing cost, and it is thus widely accepted as the standard technique and used in most image manipulation programs. By adjusting the control parameter a (Eqn. (16.53)), the bicubic kernel can be easily tuned to fit the need of particular applications. For example, the *Catmull-Rom* method can be implemented with the bicubic interpolation by setting $a = 0.5$ (Eqns. (16.57) and (16.74)).

Results from the 2D *Lanczos* interpolation (not shown here) using the 2-tap kernel $W_{\mathrm{L}2}$ cannot be expected to be better than the bicubic interpolation, which can be adjusted to give similar results without producing the artifacts shown in Fig. 16.24. The 3-tap Lanczos kernel $W_{\mathrm{L}3}$ on the other hand should produce slightly sharper edges at the cost of increased ringing (see also Exercise 16.7).

For high-quality applications one may consider the *Mitchell-Netravali* method (Eqns. (16.59) and (16.75)), which offers exceptional reconstruction at the same computational cost as the bicubic interpolation (see Exercise 16.6).

16.3.8 Aliasing

As we described in the previous parts of this chapter, the usual approach for implementing geometric image transformations can be summarized by the following three steps (Fig. 16.32):

1. Each discrete image point (u'_0, v'_0) of the *target* image is projected by the inverse geometric transformation T^{-1} to the continuous coordinate (x_0, y_0) in the source image.
2. The continuous image function $\hat{I}(x, y)$ is reconstructed from the discrete source image $I(u, v)$ by interpolation (using one of the methods described above).
3. The interpolated function is sampled at position (x_0, y_0), and the sample value $\hat{I}(x_0, y_0)$ is transferred to the target pixel $I'(u'_0, v'_0)$.

Sampling the interpolated image

One problem not considered so far concerns the process of sampling the reconstructed, continuous image function in step 3 above. The problem occurs when the geometric transformation T causes parts of the image to be *contracted*. In this case, the distance between adjacent sample points on the source image is locally *increased* by the corresponding inverse transformation T^{-1}. Now, widening the sampling distance reduces the spatial sampling rate and thus the maximum permissible frequencies in the reconstructed image function $\hat{I}(x, y)$. Eventually this leads to a violation of the sampling criterion and causes visible aliasing in the transformed image. The problem does not occur when the image is enlarged

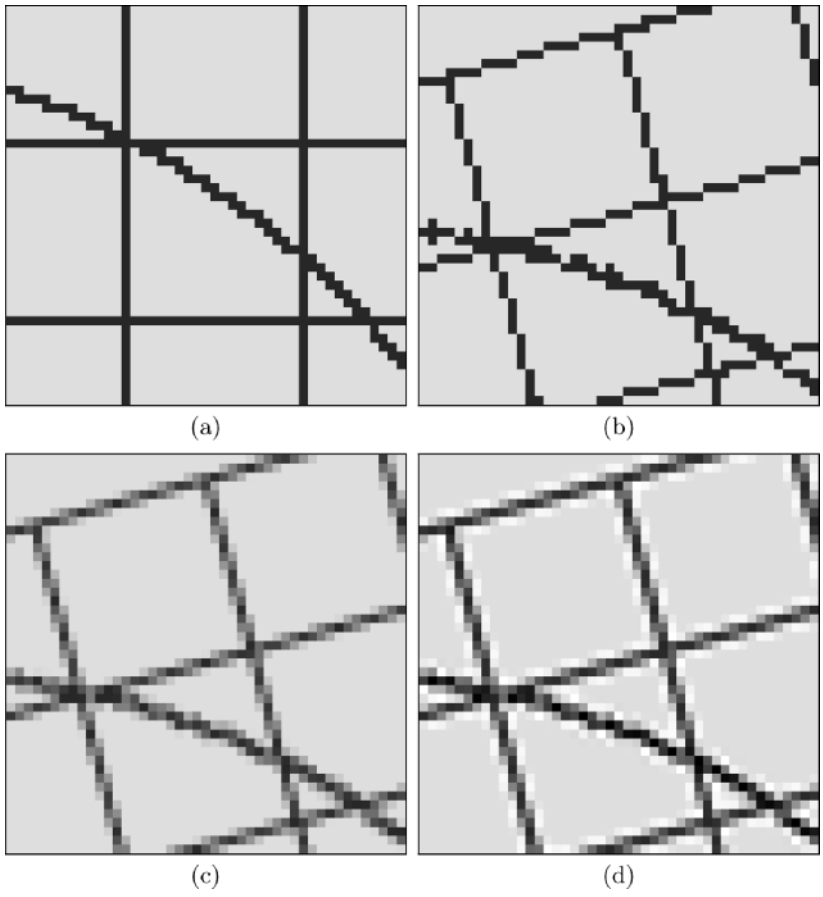


Fig. 16.31
Image interpolation methods compared: part of the original image (a), which is subsequently rotated by 15°, nearest-neighbor interpolation (b), bilinear interpolation (c), and bicubic interpolation (d).

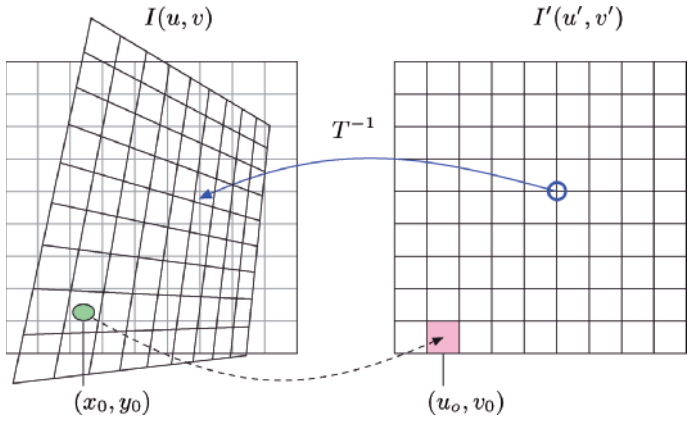


Fig. 16.32
Sampling errors in geometric operations. If the geometric transformation T leads to a local contraction of the image (which corresponds to a local enlargement by T^{-1}), the distance between adjacent sample points in I is increased. This reduces the local sampling frequency and thus the maximum signal frequency allowed in the source image, which eventually leads to aliasing.

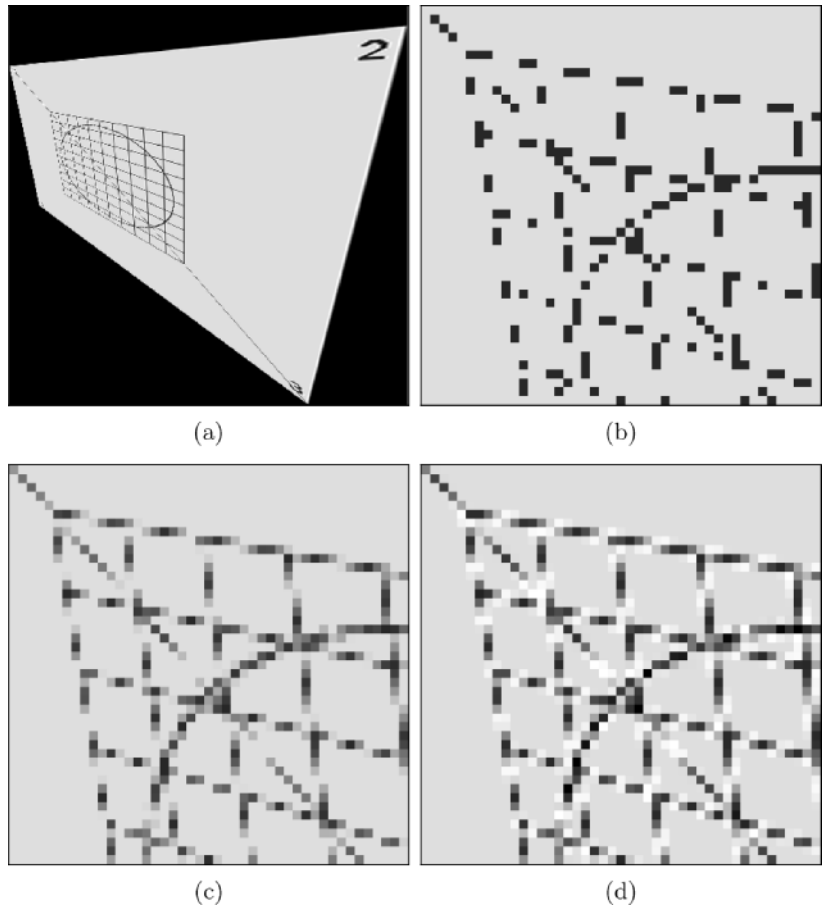


Fig. 16.33
Aliasing caused by local image contraction. Aliasing is caused by a violation of the sampling criterion and is largely unaffected by the interpolation method used: complete transformed image (a), detail using nearest-neighbor interpolation (b), bilinear interpolation (c), and bicubic interpolation (d).

by the geometric transformation because in this case the sampling interval on the source image is shortened (corresponding to a higher sampling frequency) and no aliasing can occur.

Notice that this effect is largely unrelated to the interpolation method, as demonstrated by the examples in Fig. 16.33. The effect is most noticeable under nearest-neighbor interpolation in Fig. 16.33 (b), where the thin lines are simply not "hit" by the widened sampling raster and thus disappear in some places. Important image information is thereby lost. The bilinear and bicubic interpolation methods in Fig. 16.33 (c, d) have wider interpolation kernels but still cannot avoid the aliasing effect. The problem of course gets worse with increasing reduction factors.

Low-pass filtering

One solution to the aliasing problem is to make sure that the interpolated image function is properly frequency-limited before it gets resampled.

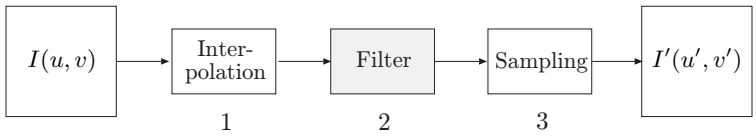


Fig. 16.34
Low-pass filtering to avoid aliasing in geometric operations. After interpolation (step 1), the reconstructed image function is subjected to low-pass filtering (step 2) before being resampled (step 3).

This can be accomplished with a suitable low-pass filter, as illustrated in Fig. 16.34.

The cutoff frequency of the low-pass filter is determined by the amount of local scale change, which may—depending upon the type of transformation—be different in various parts of the image. In the simplest, case the amount of scale change is the same throughout the image (e. g., under global scaling or affine transformations, where the same filter can be used everywhere in the image).

In general, however, the low-pass filter is *space-variant* or *nonhomogeneous*, and the local filter parameters are determined by the transformation T and the current image position. If convolution filters are used for both interpolation and low-pass filtering, they could be combined into a common, space-variant reconstruction filter. Unfortunately, space-variant filtering is computationally expensive and thus is often avoided, even in professional applications (e. g., Adobe Photoshop). The technique is nevertheless used in certain applications, such as high-quality texture mapping in computer graphics [31, 44, 105].

16.4 Java Implementation

In ImageJ, only a few simple geometric operations are currently implemented as methods in the `ImageProcessor` class, such as rotation and flipping. Additional operations, including affine transformations, are available as plugin classes as part of the optional `TransformJ` package [70]. In the following, we develop a rudimentary Java implementation for a set of geometric operations with the class structure summarized in Fig. 16.35. The Java classes form two groups: the first group implements the geometric transformations discussed in Sec. 16.1,[10] while the second group implements the most important interpolation methods described in Sec. 16.3. Finally, we show a sample ImageJ plugin to demonstrate the use of this implementation.

16.4.1 Geometric Transformations

The following Java classes represent geometric transformations in 2D and provide methods for computing the transformation parameters from corresponding point pairs.

[10] The standard Java API currently only implements the *affine transformation* (in class java.awt.geom.AffineTransform).

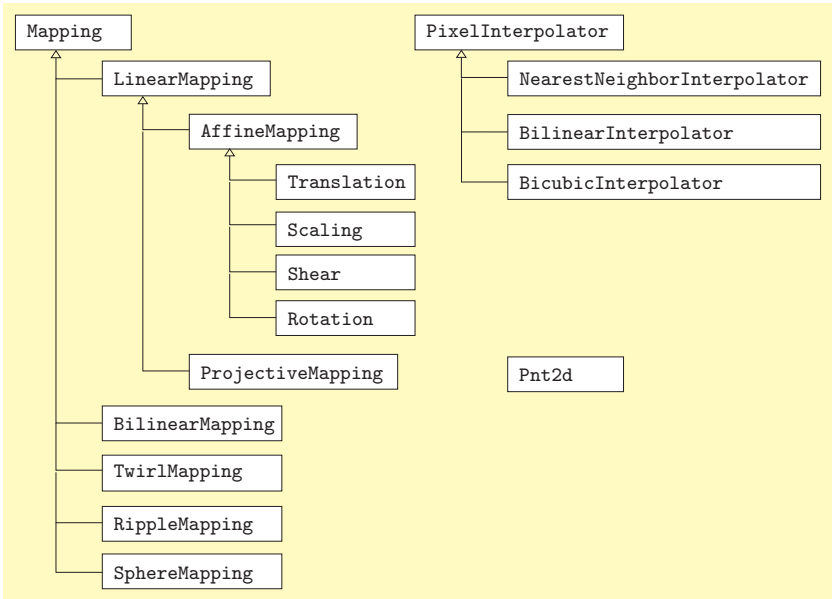


Fig. 16.35 Java class structure for the implementation of geometric operations. The class `Mapping` and its subclasses implement the geometric transformations, and `PixelInterpolator` implements various interpolation methods. `Pnt2d` is an auxiliary class for representing 2D coordinates.

`Pnt2d` (class)

Two-dimensional, continuous coordinates $\boldsymbol{x} = (x, y) \in \mathbb{R}^2$ are represented as objects of the class `Pnt2d`:

```
1 public class Pnt2d {
2   double x, y;
3
4   Pnt2d (double x, double y){
5     this.x = x; this.y = y;
6   }
7 }
```

`Mapping` (class)

The abstract class `Mapping` is the superclass for all subsequent transformations. All subclasses of `Mapping` are required to implement the method

```
applyTo (Pnt2d pnt)
```

which applies the specific transformation to a given coordinate point `pnt`. The method

```
applyTo (ImageProcessor ip, PixelInterpolator intPol)
```

on the other hand is implemented by the class `Mapping` itself and is not supposed to be overwritten by subclasses. It is used to apply some coordinate transformation to the entire image `ip`, with the specified `PixelInterpolator` object `intPol` taking care of the pixel interpolation (see line 36 in the code segment below).

The actual image transformation is based on the *target-to-source* method (Sec. 16.2.2) and thus requires the *inverse* coordinate transform T^{-1}, which can be obtained via the method `getInverse()` (see lines 14 and 24). The inverse mapping is computed and returned unless the particular mapping is already an inverse mapping (`isInverse` is `true`). Note that the inversion is only implemented for *linear* transformations (class `LinearMapping` and derived subclasses). In all other cases, an inverse mapping is created immediately when the `Mapping` object is instantiated, so no inversion is ever needed.

```
1  import ij.process.ImageProcessor;
2
3  public abstract class Mapping implements Cloneable {
4    boolean isInverse = false;
5
6    // subclasses must implement this method:
7    abstract Pnt2d applyTo(Pnt2d pnt);
8
9    Mapping invert() {
10     throw new
11       IllegalArgumentException("cannot invert mapping");
12   }
13
14   Mapping getInverse() {
15     if (isInverse)
16       return this;
17     else
18       return this.invert(); // only linear mappings invert
19   }
20
21   void applyTo(ImageProcessor ip, PixelInterpolator intPol){
22     ImageProcessor targetIp = ip;
23     ImageProcessor sourceIp = ip.duplicate();
24     Mapping invMap = this.getInverse(); // get inverse mapping
25     intPol.setImageProcessor(sourceIp);
26     int w = sourceIp.getWidth();
27     int h = sourceIp.getHeight();
28
29     Pnt2d pt = new Pnt2d(0,0);
30     for (int v=0; v<h; v++){
31       for (int u=0; u<w; u++){
32         pt.x = u;
33         pt.y = v;
34         invMap.applyTo(pt);
35         int p =
36           (int) Math.rint(intPol.getInterpolatedPixel(pt));
37         targetIp.putPixel(u,v,p);
38       }
39     }
40   }
41
```

```
  Mapping duplicate() { //clones any mapping object
    Mapping newMap = null;
    try {
      newMap = (Mapping) this.clone();
    }
    catch (CloneNotSupportedException e){};
    return newMap;
  }

} // end of class Mapping
```

LinearMapping (class)

`LinearMapping` is a subclass of `Mapping` that implements an arbitrary linear transformation in 2D using homogeneous coordinates. The nine elements $a_{11}, a_{12}, \ldots a_{33}$ of the 3×3 transformation matrix are represented by corresponding instance variables. This class is normally not instantiated (only subclasses are) but supplies the general functionality of linear mappings, in particular the transformation of 2D points (method `applyTo(Pnt2d pnt)`), inversion (method `invert()`), and concatenation with another linear mapping (method `concat(LinearMapping B)`):

```
public class LinearMapping extends Mapping {
  double
    a11 = 1, a12 = 0, a13 = 0, // transformation matrix
    a21 = 0, a22 = 1, a23 = 0,
    a31 = 0, a32 = 0, a33 = 1;

  LinearMapping() {}

  LinearMapping (    // constructor method
      double a11, double a12, double a13,
      double a21, double a22, double a23,
      double a31, double a32, double a33,
      boolean inv) {
    this.a11 = a11; this.a12 = a12; this.a13 = a13;
    this.a21 = a21; this.a22 = a22; this.a23 = a23;
    this.a31 = a31; this.a32 = a32; this.a33 = a33;
    isInverse = inv;
  }

  Pnt2d applyTo (Pnt2d pnt) {      // see Eqn. (16.17)
    double h = (a31*pnt.x + a32*pnt.y + a33);
    double x = (a11*pnt.x + a12*pnt.y + a13) / h;
    double y = (a21*pnt.x + a22*pnt.y + a23) / h;
    pnt.x = x;
    pnt.y = y;
    return pnt;
  }

```

```
29   Mapping invert() {                    // see Eqn. (16.23)
30     LinearMapping lm = (LinearMapping) duplicate();
31     double det =
32       a11*a22*a33 + a12*a23*a31 + a13*a21*a32 -
33       a11*a23*a32 - a12*a21*a33 - a13*a22*a31;
34
35     lm.a11 = (a22*a33 - a23*a32) / det;
36     lm.a12 = (a13*a32 - a12*a33) / det;
37     lm.a13 = (a12*a23 - a13*a22) / det;
38
39     lm.a21 = (a23*a31 - a21*a33) / det;
40     lm.a22 = (a11*a33 - a13*a31) / det;
41     lm.a23 = (a13*a21 - a11*a23) / det;
42
43     lm.a31 = (a21*a32 - a22*a31) / det;
44     lm.a32 = (a12*a31 - a11*a32) / det;
45     lm.a33 = (a11*a22 - a12*a21) / det;
46
47     lm.isInverse = !isInverse;
48     return lm;
49   }
50
51   // concatenates this transform matrix A with B: C ← B · A
52   LinearMapping concat(LinearMapping B) {
53     LinearMapping lm = (LinearMapping) duplicate();
54     lm.a11 = B.a11*a11 + B.a12*a21 + B.a13*a31;
55     lm.a12 = B.a11*a12 + B.a12*a22 + B.a13*a32;
56     lm.a13 = B.a11*a13 + B.a12*a23 + B.a13*a33;
57
58     lm.a21 = B.a21*a11 + B.a22*a21 + B.a23*a31;
59     lm.a22 = B.a21*a12 + B.a22*a22 + B.a23*a32;
60     lm.a23 = B.a21*a13 + B.a22*a23 + B.a23*a33;
61
62     lm.a31 = B.a31*a11 + B.a32*a21 + B.a33*a31;
63     lm.a32 = B.a31*a12 + B.a32*a22 + B.a33*a32;
64     lm.a33 = B.a31*a13 + B.a32*a23 + B.a33*a33;
65     return lm;}
66
67 } // end of class LinearMapping
```

AffineMapping (class)

AffineMapping extends its superclass LinearMapping with two additional functions. First, it contains a special constructor method that initializes the elements a_{31}, a_{32}, a_{33} of the transformation matrix to $0, 0, 1$, as required by the affine transformation. Second, it defines the method makeMapping(), which is used to compute the parameters of the affine transformation T from three pairs of corresponding points $(\mathtt{A}_i, \mathtt{B}_i)$ as described in Eqn. (16.15):

```
1  public class AffineMapping extends LinearMapping {
2
3    AffineMapping ( // constructor method
4        double a11, double a12, double a13,
5        double a21, double a22, double a23,
6        boolean inv) {
7      super(a11,a12,a13,a21,a22,a23,0,0,1,inv);
8    }
9
10   // create the affine transform between
11   // arbitrary triangles (A1..A3) and (B1..B3)
12   static AffineMapping makeMapping (
13       Pnt2d A1, Pnt2d A2, Pnt2d A3,
14       Pnt2d B1, Pnt2d B2, Pnt2d B3) {
15
16     double ax1 = A1.x, ax2 = A2.x,  ax3 = A3.x;
17     double ay1 = A1.y, ay2 = A2.y,  ay3 = A3.y;
18     double bx1 = B1.x, bx2 = B2.x,  bx3 = B3.x;
19     double by1 = B1.y, by2 = B2.y,  by3 = B3.y;
20
21     double S = ax1*(ay3-ay2) + ax2*(ay1-ay3) + ax3*(ay2-ay1);
22     double a11 =
23         (ay1*(bx2-bx3)+ay2*(bx3-bx1)+ay3*(bx1-bx2)) / S;
24     double a12 =
25         (ax1*(bx3-bx2)+ax2*(bx1-bx3)+ax3*(bx2-bx1)) / S;
26     double a21 =
27         (ay1*(by2-by3)+ay2*(by3-by1)+ay3*(by1-by2)) / S;
28     double a22 =
29         (ax1*(by3-by2)+ax2*(by1-by3)+ax3*(by2-by1)) / S;
30     double a13 =
31         (ax1*(ay3*bx2-ay2*bx3) + ax2*(ay1*bx3-ay3*bx1)
32         + ax3*(ay2*bx1-ay1*bx2)) / S;
33     double a23 =
34         (ax1*(ay3*by2-ay2*by3) + ax2*(ay1*by3-ay3*by1)
35         + ax3*(ay2*by1-ay1*by2)) / S;
36
37     return new AffineMapping(a11,a12,a13,a21,a22,a23,false);
38   }
39
40 } // end of class AffineMapping
```

Translation, Scaling, Shear, Rotation (classes)

The classes `Translation`, `Scaling`, `Shear`, and `Rotation` are direct subclasses of `AffineMapping`. The definition of each class only contains the corresponding constructor method. The remaining functionality is derived from the superclasses `AffineTransform` and `LinearTransform`. The call `super()` in the following code segments refers to the constructor method of the direct superclass `AffineMapping`:

```
1 class Translation extends AffineMapping { // see Eqn. (16.4)
2   Translation (double dx, double dy) {
3     super(
4       1, 0, dx,
5       0, 1, dy,
6       false );
7   }
8 }  // end of class Translation
```

```
1 class Scaling extends AffineMapping { // see Eqn. (16.5)
2   Scaling(double sx, double sy) {
3     super(
4       sx, 0, 0,
5       0, sy, 0,
6       false );
7   }
8 }  // end of class Scaling
```

```
1 class Shear extends AffineMapping { // see Eqn. (16.6)
2   Shear(double bx, double by) {
3     super(
4       1, bx, 0,
5       by, 1, 0,
6       false );
7   }
8 }  // end of class Shear
```

```
1 class Rotation extends AffineMapping { // see Eqn. (16.8)
2   Rotation(double alpha) {
3     super(
4        Math.cos(alpha), Math.sin(alpha), 0,
5       -Math.sin(alpha), Math.cos(alpha), 0,
6        false);
7   }
8 }  // end of class Rotation
```

`ProjectiveMapping` (class)

The class `ProjectiveMapping` implements a linear projective transformation as defined in Eqn. (16.17). The class provides a constructor method for initializing the eight transformation parameters $a_{11}, a_{12}, \ldots a_{32}$ ($a_{33} = 1$) and two different (overloaded) methods to compute the transformation for given pairs of quadrilaterals.

The first version of `makeMapping()`, in line 12, creates a projective mapping from the unit square $\mathcal{S}_1$ to an arbitrary quadrilateral $\mathcal{Q}$, defined by the coordinate points `P1`...`P4` (see Eqns. (16.28)–(16.31)). The second version of `makeMapping()`, in line 37, computes the projective mapping between two arbitrary quadrilaterals `A1`...`A4` and `B1`...`B4` in two steps via the unit square (see Eqn. (16.33)). This method makes

use of the methods `invert()` and `concat()` defined earlier in the class `LinearMapping`:

```
1  class ProjectiveMapping extends LinearMapping {
2
3    ProjectiveMapping(
4        double a11, double a12, double a13,
5        double a21, double a22, double a23,
6        double a31, double a32, boolean inv) {
7      super(a11,a12,a13,a21,a22,a23,a31,a32,1,inv);
8    }
9
10   // creates the projective mapping from the unit square S1 to
11   // the arbitrary quadrilateral Q given by points P1...P4:
12   static ProjectiveMapping makeMapping(
13           Pnt2d P1, Pnt2d P2, Pnt2d P3, Pnt2d P4) {
14     double x1 = P1.x, x2 = P2.x, x3 = P3.x, x4 = P4.x;
15     double y1 = P1.y, y2 = P2.y, y3 = P3.y, y4 = P4.y;
16     double S = (x2-x3)*(y4-y3) - (x4-x3)*(y2-y3);
17
18     double a31 =
19         ((x1-x2+x3-x4)*(y4-y3)-(y1-y2+y3-y4)*(x4-x3))/S;
20     double a32 =
21         ((y1-y2+y3-y4)*(x2-x3)-(x1-x2+x3-x4)*(y2-y3))/S;
22
23     double a11 = x2 - x1 + a31*x2;
24     double a12 = x4 - x1 + a32*x4;
25     double a13 = x1;
26
27     double a21 = y2 - y1 + a31*y2;
28     double a22 = y4 - y1 + a32*y4;
29     double a23 = y1;
30
31     return new
32       ProjectiveMapping(a11,a12,a13,a21,a22,a23,a31,a32,false);
33   }
34
35   // creates the projective mapping between arbitrary
36   // quadrilaterals Qa, Qb via the unit square S1: Qa → S1 → Qb
37   static ProjectiveMapping makeMapping (
38       Pnt2d A1, Pnt2d A2, Pnt2d A3, Pnt2d A4,
39       Pnt2d B1, Pnt2d B2, Pnt2d B3, Pnt2d B4) {
40     ProjectiveMapping T1 = makeMapping(A1, A2, A3, A4);
41     ProjectiveMapping T2 = makeMapping(B1, B2, B3, B4);
42     LinearMapping T1i = (LinearMapping) T1.invert();
43     LinearMapping T = T1i.concat(T2);
44     T.isInverse = false;
45     return (ProjectiveMapping) T;
46   }
47 }  // end of class ProjectiveMapping
```

BilinearMapping (class)

This class implements the bilinear transformation described in Sec. 16.1.5. As a nonlinear mapping, it is a direct subclass of `Mapping`, it has eight transformation parameters (`a1`...`b4`), and (since it does not inherit the corresponding method from class `LinearTransformation`) defines its own `applyTo(Pnt2d pnt)` method for transforming individual coordinate points (see Eqn. (16.35)):

```
1  import Jama.Matrix; // use the JAMA linear algebra package
2
3  class BilinearMapping extends Mapping {
4    double a1, a2, a3, a4;
5    double b1, b2, b3, b4;
6
7    c(  // constructor method
8        double a1, double a2, double a3, double a4,
9        double b1, double b2, double b3, double b4,
10       boolean inv) {
11     this.a1 = a1; this.a2 = a2; this.a3 = a3; this.a4 = a4;
12     this.b1 = b1; this.b2 = b2; this.b3 = b3; this.b4 = b4;
13     isInverse = inv;
14   }
15
16   Pnt2d applyTo (Pnt2d pnt){
17     double x = pnt.x;
18     double y = pnt.y;
19     pnt.x = a1 * x + a2 * y + a3 * x * y + a4;
20     pnt.y = b1 * x + b2 * y + b3 * x * y + b4;
21     return pnt;
22   }
23
24    // (continued below)
```

To avoid the problem of inverting this transformation, the method `makeInverseMapping()` creates the inverse mapping T^{-1} directly. This method computes the (inverse) bilinear mapping from a given quadrilateral $\mathcal{P}$ (specified by the coordinate points `P1`...`P4`) to another quadrilateral $\mathcal{Q}$ (`Q1`...`Q4`):

```
25   // map between arbitrary quadrilaterals P → Q
26   public static BilinearMapping makeInverseMapping(
27       Pnt2d P1, Pnt2d P2, Pnt2d P3, Pnt2d P4, // source quad P
28       Pnt2d Q1, Pnt2d Q2, Pnt2d Q3, Pnt2d Q4) // target quad Q
29     {
30
31     //define column vectors x, y
32     Matrix X = new Matrix(
33       new double[][] {{Q1.x},{Q2.x},{Q3.x},{Q4.x}});
34     Matrix Y = new Matrix(
35       new double[][] {{Q1.y},{Q2.y},{Q3.y},{Q4.y}});
36
```

```
37     //define matrix M
38     Matrix M = new Matrix(new double[][]
39       {{P1.x, P1.y, P1.x * P1.y, 1},
40        {P2.x, P2.y, P2.x * P2.y, 1},
41        {P3.x, P3.y, P3.x * P3.y, 1},
42        {P4.x, P4.y, P4.x * P4.y, 1}}
43       );
44
45     Matrix A = M.solve(X);   // solve x = M · a (Eqn. (16.36))
46     Matrix B = M.solve(Y);   // solve y = M · b (Eqn. (16.37))
47
48     double a1 = A.get(0,0);  double b1 = B.get(0,0);
49     double a2 = A.get(1,0);  double b2 = B.get(1,0);
50     double a3 = A.get(2,0);  double b3 = B.get(2,0);
51     double a4 = A.get(3,0);  double b4 = B.get(3,0);
52
53     return new BilinearMapping(a1,a2,a3,a4,b1,b2,b3,b4,true);
54   }
55 }  // end of class BilinearMapping
```

In the method above, a 4×4 system of linear equations is solved in lines 45–46 using the method `solve()` of the `Matrix` class in the JAMA[11] numerical library.

`TwirlMapping` (class)

This class implements the twirl mapping (Eqns. (16.38) and (16.39)) as a typical example of a warp transformation. The mapping is nonlinear, and thus `TwirlMapping` is a subclass of the general mapping class `Mapping`. Again the inverse transformation T^{-1} is created directly using the method `makeInverseMapping()`:

```
1 public class TwirlMapping extends Mapping {
2   double xc;
3   double yc;
4   double angle;
5   double rad;
6
7   TwirlMapping (
8       double xc, double yc, double angle,
9       double rad, boolean inv)
10  {
11    this.xc = xc;
12    this.yc = yc;
13    this.angle = angle;
14    this.rad = rad;
15    this.isInverse = inv;
16  }
17
```

[11] http://math.nist.gov/javanumerics/jama/.

```
18   static TwirlMapping makeInverseMapping (
19       double xc, double yc, double angle, double rad)
20   {
21     return new TwirlMapping(xc, yc, angle, rad, true);
22   }
23
24   Pnt2d applyTo (Pnt2d pnt) {
25     double x = pnt.x;
26     double y = pnt.y;
27     double dx = x - xc;
28     double dy = y - yc;
29     double d = Math.sqrt(dx*dx + dy*dy);
30     if (d < rad) {
31       double a = Math.atan2(dy,dx) + angle * (rad-d) / rad;
32       pnt.x = xc + d*Math.cos(a);
33       pnt.y = yc + d*Math.sin(a);
34     }
35     return pnt;
36   }
37 }  // end of class TwirlMapping
```

Similar classes could be defined to implement the ripple transformation and the spherical distortion described in Sec. 16.1.6 (see Exercise 16.3).

16.4.2 Pixel Interpolation

The following class definitions implement three of the interpolation methods described in Sec. 16.3. Each class provides its own version of the method `getInterpolatedPixel(Pnt2d pnt)`, which returns the interpolated value of the image function at the given continuous coordinate `pnt` $= (x_0, y_0)$ as a floating-point value.

`PixelInterpolator` (class)

`PixelInterpolator` is the (abstract) superclass for the actual interpolator classes. In particular, it specifies the method `getInterpolatedPixel (Pnt2d pnt)`, which must be implemented by all subclasses and is invoked by the method `applyTo()` in class `Mapping` (p. 415):

```
1  import ij.process.ImageProcessor;
2
3  public abstract class PixelInterpolator {
4    ImageProcessor ip;
5
6    PixelInterpolator() {}
7
8    void setImageProcessor(ImageProcessor ip) {
9      this.ip = ip;
10   }
11
```

```
12   abstract double getInterpolatedPixel(Pnt2d pnt);
13
14 } // end of class PixelInterpolator
```

NearestNeighborInterpolator (class)

This class implements the two-dimensional *nearest-neighbor* interpolation (see Eqn. (16.67)):

```
1 public class NearestNeighborInterpolator extends
      PixelInterpolator {
2
3   double getInterpolatedPixel(Pnt2d pnt) {
4     int u = (int) Math.rint(pnt.x);
5     int v = (int) Math.rint(pnt.y);
6     return ip.getPixel(u,v);
7   }
8 } // end of class NearestNeighborInterpolator
```

BilinearInterpolator (class)

This class implements the *bilinear* interpolation method (see Eqn. (16.70)):

```
1 public class BilinearInterpolator extends PixelInterpolator
2 {
3   double getInterpolatedPixel(Pnt2d pnt) {
4     int u = (int) Math.floor(pnt.x);
5     int v = (int) Math.floor(pnt.y);
6     double a = pnt.x - u;
7     double b = pnt.y - v;
8     int A = ip.getPixel(u,v);
9     int B = ip.getPixel(u+1,v);
10    int C = ip.getPixel(u,v+1);
11    int D = ip.getPixel(u+1,v+1);
12    double E = A + a*(B-A);
13    double F = C + a*(D-C);
14    double G = E + b*(F-E);
15    return G;
16  }
17 } // end of class BilinearInterpolator
```

The bilinear interpolation is also implemented in the current ImageJ distribution by the method

```
double getInterpolatedPixel (double x, double y)
```

in class `ImageProcessor`.

BicubicInterpolator (class)

This class implements the bicubic interpolation method described in Eqn. (16.73) and Alg. 16.2. The control parameter a (with $a = 1$ as the default value) can be modified by the corresponding parameter in the second constructor method (line 6). The one-dimensional cubic interpolation (Eqn. (16.55)) is implemented by the method `double cubic (double x)`:

```
1  public class BicubicInterpolator extends PixelInterpolator {
2    double a = 1;  // control parameter a (default setting)
3
4    BicubicInterpolator() {}
5
6    BicubicInterpolator(double a) {
7      this.a = a;
8    }
9
10   public double getInterpolatedPixel(Pnt2d pnt) {
11     double x0 = pnt.x;
12     double y0 = pnt.y;
13     // use floor to correctly handle negative coordinates:
14     int u0 = (int) Math.floor(x0);
15     int v0 = (int) Math.floor(y0);
16
17     double q = 0;
18     for (int j = 0; j <= 3; j++) {
19       int v = v0 - 1 + j;
20       double p = 0;
21       for (int i = 0; i <= 3; i++) {
22         int u = u0 - 1 + i;
23         p = p + ip.getPixel(u,v) * cubic(x0 - u);
24       }
25       q = q + p * cubic(y0 - v);
26     }
27     return q;
28   }
29
30   double cubic(double x) {
31     if (x < 0) x = -x;
32     double z = 0;
33     if (x < 1)
34       z = (-a+2)*x*x*x + (a-3)*x*x + 1;
35     else if (x < 2)
36       z = -a*x*x*x + 5*a*x*x - 8*a*x + 4*a;
37     return z;
38   }
39 } // end of class BicubicInterpolator
```

Setting $a = 0.5$, this class implements a *Catmull-Rom* interpolation (see Sec. 16.3.5).

16.4.3 Sample Applications

The following ImageJ plugins show two simple examples of the use of the classes above for implementing geometric operations.

Example 1: Rotation

The plugin class `PluginRotation_` performs a rotation of the image by 15°. First the geometric mapping object (`map`) is created as an instance of class `Rotation`, with the given angle being converted from degrees to radians (line 14). Subsequently, the interpolator object `ipol` is created (line 15). Here we chose a `BicubicInterpolator` with $a = 0.5$ (i. e., *Catmull-Rom* interpolation). The actual transformation of the image is accomplished by invoking the method `applyTo()` in line 16:

```
1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.*;
4
5 public class Geometry_Rotate implements PlugInFilter {
6
7   double angle = 15; // rotation angle (in degrees)
8
9   public int setup(String arg, ImagePlus imp) {
10    return DOES_8G;
11  }
12
13  public void run(ImageProcessor ip) {
14    Rotation map = new Rotation((2 * Math.PI * angle) / 360);
15    PixelInterpolator ipol = new BicubicInterpolator(0.5);
16    map.applyTo(ip, ipol);
17  }
18 } // end of class Geometry_Rotate
```

Example 2: Projective transformation

The second examples demonstrates a projective transformation, the mapping T being specified by two corresponding quadrilaterals $\mathcal{P} =$ `p1`...`p4` and $\mathcal{Q} =$ `q1`...`q4`. Of course, in a real application, these points would probably be specified interactively or given as the result of a mesh partitioning.

The mapping object `map`, which represents the forward transformation T, is created by invoking the static method `ProjectiveMapping.makeMapping()` in line 22. In this case, we used a *bilinear* interpolator (line 24), which is applied as in the previous example (line 25):

```
1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
```

```
3 import ij.process.*;
4
5 public class Geometry_ProjectiveMapping implements
      PlugInFilter
6 {
7   public int setup(String arg, ImagePlus imp) {
8       return DOES_8G;
9   }
10
11   public void run(ImageProcessor ip) {
12     Pnt2d p1 = new Pnt2d(0,0);
13     Pnt2d p2 = new Pnt2d(400,0);
14     Pnt2d p3 = new Pnt2d(400,400);
15     Pnt2d p4 = new Pnt2d(0,400);
16
17     Pnt2d q1 = new Pnt2d(0,60);
18     Pnt2d q2 = new Pnt2d(400,20);
19     Pnt2d q3 = new Pnt2d(300,400);
20     Pnt2d q4 = new Pnt2d(30,200);
21
22     ProjectiveMapping map =
23       ProjectiveMapping.makeMapping(p1,p2,p3,p4,q1,q2,q3,q4);
24     PixelInterpolator ipol = new BilinearInterpolator();
25     map.applyTo(ip, ipol);
26   }
27 } // end of class Geometry_ProjectiveMapping
```

16.5 Exercises

Exercise 16.1. Show that a straight line $y = kx + d$ in 2D is mapped to another straight line under a projective transformation (Eqn. (16.17)).

Exercise 16.2. Show that parallel lines remain parallel under affine transformation (Eqn. (16.13)).

Exercise 16.3. Implement the nonlinear geometric transformations `RippleMapping` (Eqns. (16.40) and (16.41)) and `SphereMapping` (Eqns. (16.42) and (16.43)) as Java classes analogous to the implementation of `TwirlMapping` (p. 422). Also create suitable ImageJ plugins and use them to test these mappings.

Exercise 16.4. Design a nonlinear geometric transformation similar to the ripple transformation (Eqns. (16.40) and (16.41)) that uses a *sawtooth* function instead of a sinusoid for the distortions in the horizontal and vertical directions. Use the class `TwirlMapping` (p. 422) as a template for your implementation.

Exercise 16.5. The one-dimensional interpolation function by Mitchell and Natravali $w_{\mathrm{mn}}(x)$ is defined as a general spline function $w_{\mathrm{cs}}(x, a, b)$

(Eqn. (16.59)). Show that this function can be expressed as the weighted sum of a Catmull-Rom function $w_{\mathrm{crm}}(x)$ (Eqn. (16.57)) and a cubic B-spline $w_{\mathrm{cbs}}(x)$ (Eqn. (16.58)) in the form

$$\begin{aligned} w_{\mathrm{mn}}(x) &= w_{\mathrm{cs}}\big(x, \tfrac{1}{3}, \tfrac{1}{3}\big) \\ &= \tfrac{1}{3} \cdot \big[2 \cdot w_{\mathrm{cs}}(x, 0.5, 0) + w_{\mathrm{cs}}(x, 0, 1)\big] \\ &= \tfrac{1}{3} \cdot \big[2 \cdot w_{\mathrm{crm}}(x) + w_{\mathrm{cbs}}(x)\big]. \end{aligned}$$

Exercise 16.6. Implement the two-dimensional *Mitchell-Netravali* interpolation as defined by Eqn. (16.59) and Eqn. (16.75) as a Java class analogous to the class `BicubicInterpolator` (p. 425). Compare the results with those of the bicubic interpolation.

Exercise 16.7. Implement the two-dimensional *Lanczos* interpolation with a W_{L3} kernel as defined in Eqn. (16.77) as a Java class analogous to the class `BicubicInterpolator` (p. 425). Compare the results to the bicubic interpolation.

Exercise 16.8. The one-dimensional Lanczos interpolation kernel of order $n = 4$ is (analogous to Eqn. (16.65)) defined as

$$w_{\mathrm{L4}} = \begin{cases} 4 \cdot \frac{\sin(\pi \frac{x}{4}) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 \le |x| < 4 \\ 0 & \text{for } |x| \ge 4. \end{cases} \tag{16.78}$$

Generalize the two-dimensional L3 kernel in Eqn. (16.77) to Ln, where n (the number of "taps") can be chosen arbitrarily, and implement this interpolator as a Java class analogous to `BicubicInterpolator` (p. 425). How many image pixels are used in the interpolation for a given n? Perform tests on suitable images to see how the interpolation performs when n is increased.

17

Comparing Images

When we compare two images, we are faced with the following basic question: when are two images the same or similar, and how can this similarity be measured? Of course one could trivially define two images I_1, I_2 as being identical when all pixel values are the same (i. e., the difference $I_1 - I_2$ is zero). Although this kind of definition may be useful in specific applications, such as for detecting changes in successive images under constant lighting and camera conditions, simple pixel differencing is usually too inflexible to be of much practical use. Noise, quantization errors, small changes in lighting, and minute shifts or rotations can all create large numerical pixel differences for pairs of images that would still be perceived as perfectly identical by a human viewer. Obviously, human perception incorporates a much wider concept of similarity and uses cues such as structure and content to recognize similarity between images, even when a direct comparison between individual pixels would not indicate any match. The problem of comparing images at a structural or semantic level is a difficult problem and an interesting research field, for example in the context of image-based searches on the Internet or database retrieval.

This chapter deals with the much simpler problem of comparing images at the pixel level; in particular, localizing a given subimage—often called a "template"—within some larger image. This task is frequently required, for example, to find matching patches in stereo images, to localize a particular pattern in a scene, or to track a certain pattern through an image sequence. The principal idea behind "template matching" is simple: move the given pattern (template) over the search image, measure the difference against the corresponding subimage at each position, and record those positions where the highest similarity is obtained. But this is not as simple as it may initially sound. After all, what is a suitable distance measure, what total difference is acceptable for a match,

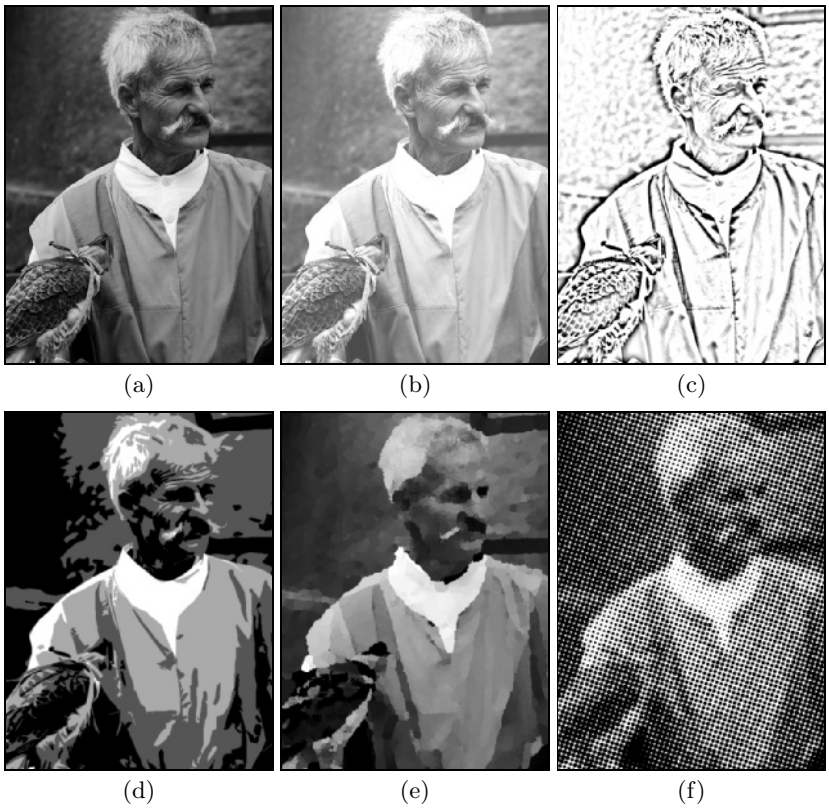


Fig. 17.1
Are these images the "same"? Simply measuring the difference between pixel values will return a large distance between the original image (a) and any of the five other images (b–f).

and what happens when brightness or contrast changes (Fig. 17.1)? We already touched on this problem of invariance under geometric transformations when we discussed the shape properties of segmented regions in Sec. 11.4.2. However, geometric invariance is not our main concern in the remaining part of this chapter, where we describe only the most basic template-matching techniques: correlation-based methods for intensity images and "chamfer-matching" for binary images.

17.1 Template Matching in Intensity Images

First we look at the problem of localizing a given *reference image* (template) R within a larger intensity (grayscale) image I, which we call the *search image*. The task is to find those positions where the contents of the reference image R and the corresponding subimage of I are either the same or most similar. If we denote by

$$R_{r,s}(u,v) = R(u-r, v-s)$$

the reference image R *shifted* by the distance (r,s) in the horizontal and vertical directions, respectively, then the matching problem (illustrated in Fig. 17.2) can be summarized as

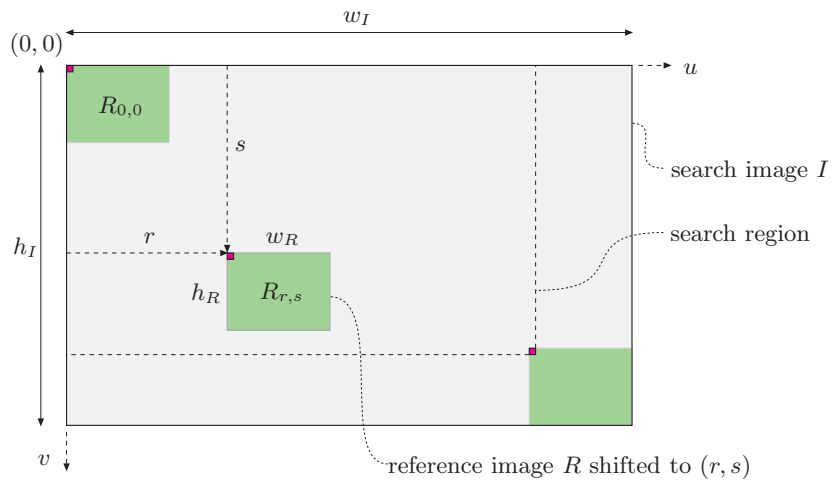


Fig. 17.2
Geometry of template matching. The reference image R is shifted across the search image I by an offset (r, s) using the origins of the two images as the reference points. The dimensions of the search image ($w_I \times h_I$) and the reference image ($w_R \times h_R$) determine the maximal search region for this comparison.

> Given are the search image I and the reference image R. Find the offset (r, s) such that the similarity between the shifted reference image $R_{r,s}$ and the corresponding subimage of I is a maximum.

To successfully solve this task, several issues need to be addressed such as determining a minimum similarity value for accepting a match and developing a good search strategy for finding the optimal displacement. First and most important, a suitable measure of similarity between subimages must be found that is reasonably tolerant against intensity and contrast variations.

17.1.1 Distance between Image Patterns

To quantify the amount of agreement, we compute a "distance" $d(r, s)$ between the shifted reference image R and the corresponding subimage of I for each offset position (r, s) (Fig. 17.3). Several distance measures have been proposed for two-dimensional intensity images, including the following three basic definitions:[1]

Sum of absolute differences:

$$\mathrm{d}_A(r,s) = \sum_{(i,j)\in R} |I(r+i, s+j) - R(i,j)|\,; \tag{17.1}$$

Maximum difference:

$$\mathrm{d}_M(r,s) = \max_{(i,j)\in R} |I(r+i, s+j) - R(i,j)|\,; \tag{17.2}$$

[1] We use the short notation $(i, j) \in R$ to specify the set of all possible template coordinates $\{\,(i,j)\,|\,0 \le i < w_R, 0 \le j < h_R\,\}$.

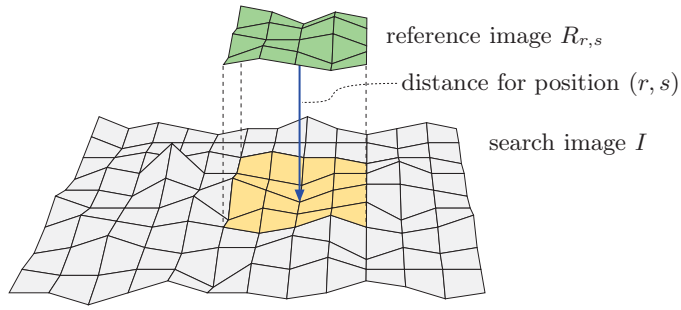


Fig. 17.3
Measuring the distance between two-dimensional image functions. The reference image R is positioned at offset (r, s) on top of the search image I.

Sum of squared differences:

$$\mathrm{d}_E(r,s) = \Big[\sum_{(i,j)\in R} \big(I(r+i, s+j) - R(i,j)\big)^2 \Big]^{1/2}. \tag{17.3}$$

This is also called the N-dimensional *Euclidean distance*, with N being the number of pixels (treated as elements of N-dimensional vectors) used in the distance computation.

Distance and correlation

Because of its formal properties, the N-dimensional distance d_E (Eqn. (17.3)) is of special importance and well-known in statistics and optimization. To find the best-matching position between the reference image R and the search image I, it is sufficient to *minimize the square* of d_E (which is always positive), which can be expanded to

$$\begin{aligned} \mathrm{d}_E^2(r,s) &= \sum_{(i,j)\in R} \big(I(r+i, s+j) - R(i,j)\big)^2 \qquad (17.4)\\ &= \underbrace{\sum_{(i,j)\in R} I^2(r+i, s+j)}_{A(r,s)} + \underbrace{\sum_{(i,j)\in R} R^2(i,j)}_{B} - \underbrace{2\sum_{(i,j)\in R} I(r+i, s+j)\cdot R(i,j)}_{C(r,s)}. \end{aligned}$$

Notice that the term B in Eqn. (17.4) is the sum of the squared pixel values in the reference image R, a constant value (independent of r, s) that can thus be ignored. The term $A(r, s)$ is the sum of the squared values within the subimage of I at the current offset (r, s). $C(r, s)$ is the so-called *linear cross correlation* ($\circledast$) between I and R, which is defined in the general case as

$$(I \circledast R)(r,s) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(r+i, s+j)\cdot R(i,j), \tag{17.5}$$

which—since R and I are assumed to have zero values outside their boundaries—is furthermore equivalent to

$$\sum_{i=0}^{w_R-1}\sum_{j=0}^{h_R-1} I(r+i,s+j)\cdot R(i,j) = \sum_{(i,j)\in R} I(r+i,s+j)\cdot R(i,j),$$

and thus the same as $C(r,s)$ in Eqn. (17.4). As we can see in Eqn. (17.5), correlation is in principle the same operation as linear *convolution* (see Sec. 6.3.1, Eqn. (6.14)), with the only difference being that the convolution kernel ($R(i,j)$ in this case) is implicitly mirrored.

If we assume for a minute that $A(r,s)$—the "signal energy"— in Eqn. (17.4) is constant throughout the image I, then $A(r,s)$ can also be ignored and the position of maximum cross correlation $C(r,s)$ coincides with the best match between R and I. In this case, the minimum of $d_E^2(r,s)$ (Eqn. (17.4)) can be found by computing the maximum value of the correlation $I \circledast R$ only. This could be interesting for practical reasons if we consider that the linear convolution (and thus the correlation) with large kernels can be computed very efficiently in the frequency domain (see also Sec. 14.5).

Normalized cross correlation

Unfortunately, the assumption made above that $A(r,s)$ is constant does not hold for most images, and thus the result of the cross correlation strongly varies with intensity changes in the image I. The *normalized cross correlation* compensates for this dependency by taking into account the energy in the reference image and the current subimage:

$$\begin{aligned} C_N(r,s) &= \frac{C(r,s)}{\sqrt{A(r,s)\cdot B}} = \frac{C(r,s)}{\sqrt{A(r,s)}\cdot\sqrt{B}} \\ &= \frac{\displaystyle\sum_{(i,j)\in R} I(r+i,s+j)\cdot R(i,j)}{\Big[\displaystyle\sum_{(i,j)\in R} I^2(r+i,s+j)\Big]^{1/2}\cdot\Big[\displaystyle\sum_{(i,j)\in R} R^2(i,j)\Big]^{1/2}}. \end{aligned} \tag{17.6}$$

If the values in the search and reference images are all positive (which is usually the case), then the result of $C_N(r,s)$ is always in the range $[0,1]$, independent of the remaining contents in I and R. In this case, the result $C_N(r,s)=1$ indicates a maximum match between R and the current subimage of I at the offset (r,s), while $C_N(r,s)=0$ signals no agreement. Thus the normalized correlation has the additional advantage of delivering a standardized match value that can be used directly (using a suitable threshold between 0 and 1) to decide about the acceptance or rejection of a match position.

In contrast to the "global" cross correlation in Eqn. (17.5), the expression in Eqn. (17.6) is a "local" distance measure. However, it, too, has the problem of measuring the *absolute* distance between the template and the subimage. If, for example, the overall intensity of the image I is

altered, then even the result of the normalized cross correlation $C_N(r,s)$ may also change dramatically.

Correlation coefficient

One solution to this problem is to compare not the original function values but the differences with respect to the average value of R and the average of the current subimage of I. This modification turns Eqn. (17.6) into

$$C_L(r,s) = \frac{\sum\limits_{(i,j)\in R} \big(I(r+i,s+j) - \bar{I}(r,s)\big) \cdot \big(R(i,j) - \bar{R}\big)}{\Big[\sum\limits_{(i,j)\in R} \big(I(r+i,s+j) - \bar{I}_{r,s}\big)^2\Big]^{1/2} \cdot \underbrace{\Big[\sum\limits_{(i,j)\in R} \big(R(i,j) - \bar{R}\big)^2\Big]^{1/2}}_{S_R^2 = K\cdot\sigma_R^2}}, \tag{17.7}$$

where the average values $\bar{I}_{r,s}$ and $\bar{R}$ are defined as

$$\bar{I}_{r,s} = \frac{1}{K} \cdot \sum_{(i,j)\in R} I(r+i,s+j) \quad \text{and} \quad \bar{R} = \frac{1}{K} \cdot \sum_{(i,j)\in R} R(i,j), \tag{17.8}$$

respectively ($K = |R|$ being the number of pixels in the reference image R). In statistics, the expression in Eqn. (17.7) is known as the *correlation coefficient*. However, different from the usual application as a global measure in statistics, $C_L(r,s)$ describes a *local*, piecewise correlation between the template R and the current subimage (at offset r,s) of I. The resulting values of $C_L(r,s)$ are in the range $[-1,1]$ regardless of the contents in R and I. Again a value of 1 indicates maximum agreement between the compared image patterns, while -1 corresponds to a maximum mismatch. The term

$$S_R^2 = K \cdot \sigma_R^2 = \sum_{(i,j)\in R} \big(R(i,j) - \bar{R}\big)^2 \tag{17.9}$$

in the denominator of Eqn. (17.7) is K times the *variance* (σ_R^2) of the values in the template R, which is constant and thus needs to be computed only once. Due to the fact that $\sigma_R = \frac{1}{K}\sum R^2(i,j) - \bar{R}^2$, the expression in Eqn. (17.9) can be reformulated as

$$\begin{aligned} S_R^2 &= \sum_{(i,j)\in R} R^2(i,j) \;-\; K \cdot \bar{R}^2 \\ &= \sum_{(i,j)\in R} R^2(i,j) \;-\; \frac{1}{K} \cdot \Big(\sum_{(i,j)\in R} R(i,j)\Big)^2. \end{aligned} \tag{17.10}$$

By inserting the results from Eqns. (17.8) and (17.10) we can rewrite Eqn. (17.7) as

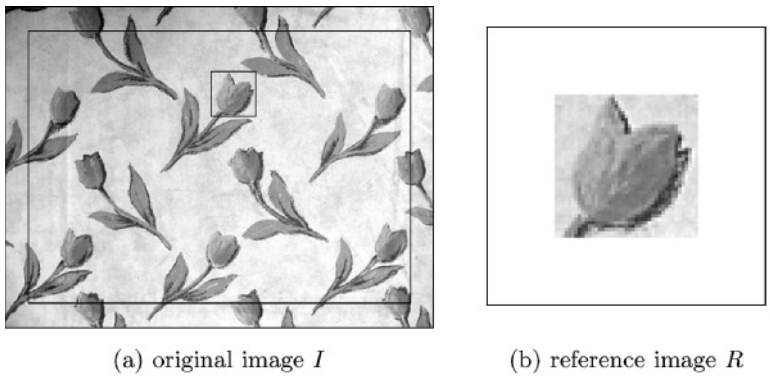
$$C_L(r,s) = \frac{\sum_{(i,j)\in R} \big(I(r+i,s+j)\cdot R(i,j)\big) \;-\; K\cdot \bar{I}_{r,s}\cdot \bar{R}}{\Big[\sum_{(i,j)\in R} I^2(r+i,s+j) \;-\; K\cdot \bar{I}^2_{r,s}\Big]^{1/2} \cdot S_R} \tag{17.11}$$

and thereby obtain an efficient way to compute the local correlation coefficient. Since $\bar{R}$ and $S_R = \sqrt{S^2_R}$ must be computed only once and the local average of the current subimage $\bar{I}_{r,s}$ is not immediately required for summing up the differences, the whole expression in Eqn. (17.11) can be computed in one common iteration, as described in Alg. 17.1. A sample Java implementation of this procedure is given by the class `CorrCoeffMatcher` in Progs. 17.1 and 17.2 (Sec. 17.1.2).

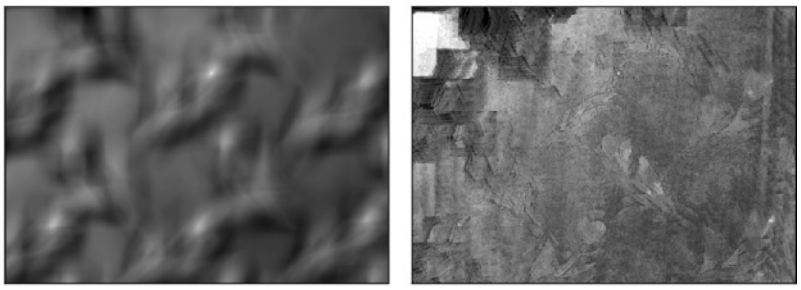
Examples and discussion

Figure 17.4 compares the performance of the described distance functions in a typical example. The original image (Fig. 17.4 (a)) shows a repetitive flower pattern under uneven lighting and due to differences in local intensity. One instance of the repetitive pattern was extracted as the reference image (Fig. 17.4 (b)).

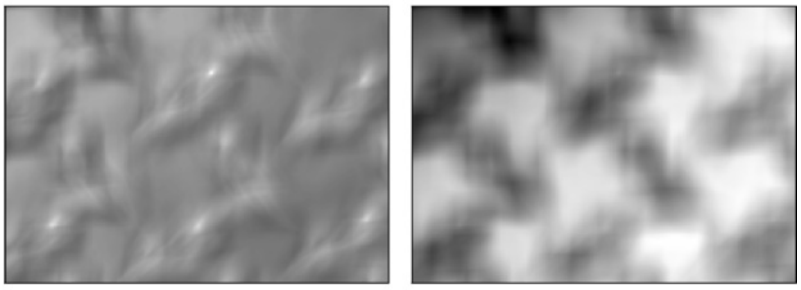
- The *sum of absolute differences* (Eqn. (17.1)) in Fig. 17.4 (c) shows a distinct peak value at the original template position, as does the *Euclidean distance* (Eqn. (17.3)) in Fig. 17.4 (e). Both measures work satisfactorily in this regard but are strongly affected by global intensity changes, as demonstrated in Figs. 17.5 and 17.6.
- The *maximum difference* (Eqn. (17.2)) in Fig. 17.4 (d) proves completely useless as a distance measure since it responds more strongly to the lighting changes than to pattern similarity. As expected, the behavior of the *global cross correlation* in Fig. 17.4 (f) is also unsatisfactory. Although the result exhibits a *local* maximum at the true template position (hardly visible in the printed image), it is completely dominated by the high-intensity responses in the brighter parts of the image.
- The result from the *normalized cross correlation* in Fig. 17.4 (g) appears naturally very similar to the Euclidean distance (Fig. 17.4 (e)) because in principle it is the same measure. As expected, the *correlation coefficient* (Eqn. (17.7)) in Fig. 17.4 (h) yields the best results. Distinct peaks of similar intensity are produced for all six instances of the template pattern, and the result is unaffected by changing lighting conditions. In this case, the values range from -1.0 (black) to $+1.0$ (white), and zero values are shown as gray.



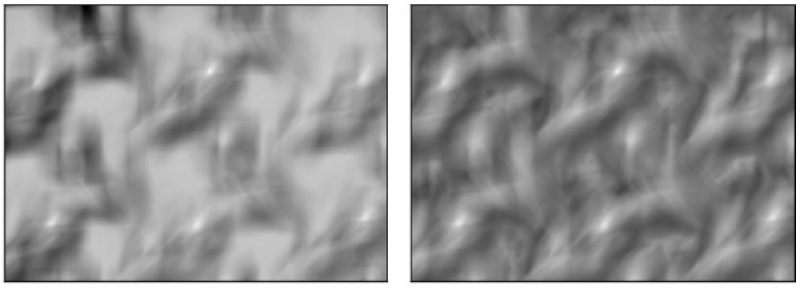
(a) original image I (b) reference image R



(c) sum of absolute differences (d) maximum difference



(e) sum of squared distances (f) global cross correlation



(g) normalized cross correlation (h) correlation coefficient

Fig. 17.4
Comparison of various distance functions. From the original image (a), the marked section is used as the reference image R, shown enlarged in (b). In the resulting difference images (c–h), brightness corresponds to the amount of agreement.

Algorithm 17.1
Computing the correlation coefficient. Given is the search image I and the reference image (template) R. In Step 1, the template's average $\bar{R}$ and variance term S_R are computed once. In Step 2, the match function is computed for every template position (r, s) as prescribed by Eqn. (17.11). The result is a map of correlation values $C(r, s) \in [-1, 1]$ that is returned. Notice that the computation in line 22 can be performed in two different ways—the second version does not require the average $\bar{I}_{r,s}$ (computed in line 21).

1: CorrelationCoefficient (I, R)

$I(u, v)$: search image of size $w_I \times h_I$
$R(i, j)$: reference image of size $w_R \times h_R$
Returns $C(r, s)$ containing the values of the correlation coefficient between I and R positioned at (r, s).

Step 1–initialize:

2: $K \leftarrow w_R \cdot h_R$
3: $\Sigma_R \leftarrow 0,\ \Sigma_{R2} \leftarrow 0$
4: **for** $i \leftarrow 0 \ldots (w_R - 1)$ **do**
5: **for** $j \leftarrow 0 \ldots (h_R - 1)$ **do**
6: $\Sigma_R \leftarrow \Sigma_R + R(i, j)$
7: $\Sigma_{R2} \leftarrow \Sigma_{R2} + \big(R(i, j)\big)^2$
8: $\bar{R} \leftarrow \Sigma_R / K$ ▷ Eqn. (17.8)
9: $S_R \leftarrow \sqrt{\Sigma_{R2} - K \cdot \bar{R}^2} = \sqrt{\Sigma_{R2} - \Sigma_R^2 / K}$ ▷ Eqn. (17.10)

Step 2—compute the correlation map:

10: $C \leftarrow$ new map of size $(w_I - w_R + 1) \times (h_I - h_R + 1)$, $C(r, s) \in \mathbb{R}$
11: **for** $r \leftarrow 0 \ldots (w_I - w_R)$ **do** ▷ place R at position (r, s)
12: **for** $s \leftarrow 0 \ldots (h_I - h_R)$ **do**

Compute correlation coefficient for position (r, s):

13: $\Sigma_I \leftarrow 0,\ \Sigma_{I2} \leftarrow 0,\ \Sigma_{IR} \leftarrow 0$
14: **for** $i \leftarrow 0 \ldots (w_R - 1)$ **do**
15: **for** $j \leftarrow 0 \ldots (h_R - 1)$ **do**
16: $a_I \leftarrow I(r + i, s + j)$
17: $a_R \leftarrow R(i, j)$
18: $\Sigma_I \leftarrow \Sigma_I + a_I$
19: $\Sigma_{I2} \leftarrow \Sigma_{I2} + a_I^2$
20: $\Sigma_{IR} \leftarrow \Sigma_{IR} + a_I \cdot a_R$
21: $\bar{I}_{r,s} \leftarrow \Sigma_I / K$ ▷ Eqn. (17.8)
22: $C(r, s) \leftarrow \dfrac{\Sigma_{IR} - K \cdot \bar{I}_{r,s} \cdot \bar{R}}{\sqrt{\Sigma_{I2} - K \cdot \bar{I}_{r,s}^2} \cdot S_R} = \dfrac{\Sigma_{IR} - \Sigma_I \cdot \bar{R}}{\sqrt{\Sigma_{I2} - \Sigma_I^2 / K} \cdot S_R}$
23: **return** C. ▷ $C(r, s) \in [-1, 1]$

Figure 17.5 compares the results of the *Euclidean distance* against the *correlation coefficient* under globally changing intensity. For this purpose, the intensity of the reference image R is raised by 50 units such that the template is different from any subpattern in the original image. As can be seen clearly, the initially distinct peaks disappear in the Euclidean distance (Fig. 17.5 (c)), while the correlation coefficient (Fig. 17.5 (d)) naturally remains unaffected by this change.

In summary, the correlation coefficient can be recommended as a reliable measure for template matching in intensity images under realistic lighting conditions. This method proves relatively robust against global changes of brightness or contrast and tolerates small deviations from the reference pattern. Since the resulting values are in the fixed range of

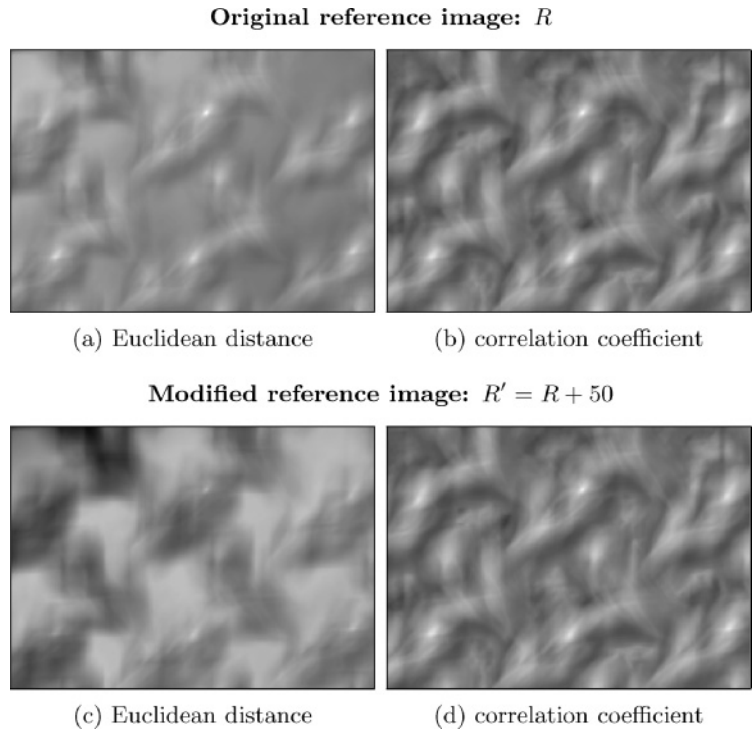


Fig. 17.5
Effects of changing global intensity. Original reference image R: the results from both the Euclidean distance (a) and the correlation coefficient (b) show distinct peaks at the positions of maximum agreement. Modified reference image $R' = R + 50$: the peak values disappear in the Euclidean distance (c), while the correlation coefficient (d) remains unaffected.

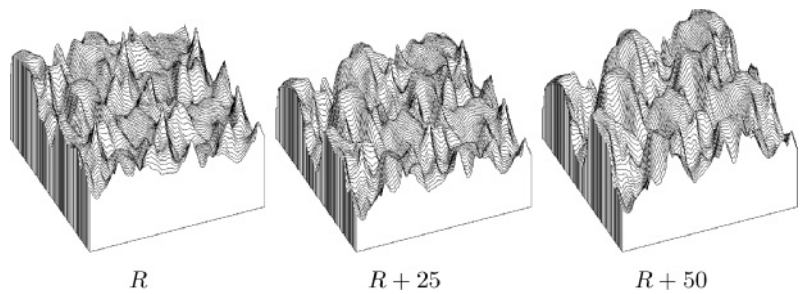
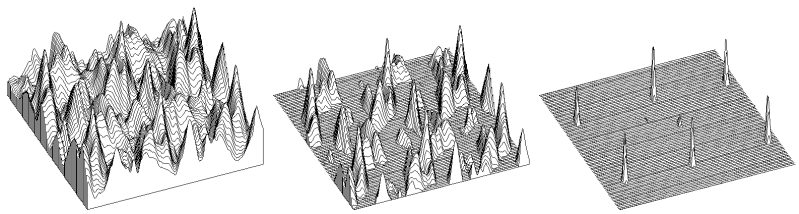


Fig. 17.6
Euclidean distance under global intensity changes. Distance function for the original template R (left), with the template intensity increased by 25 units (center) and 50 units (right). Notice that the local peaks disappear as the template intensity (and thus the total distance between the image and the template) is increased.

$[-1, 1]$, a simple threshold operation can be used to localize the best match points (Fig. 17.7).

17.1.2 Implementation

Programs 17.1 and 17.2 lists a Java implementation of template matching based on the local correlation coefficient (Eqn. (17.7)). The application assumes that the search image (`imgFp`) and the reference image (`refFp`) are already available as objects of type `FloatProcessor`. They are used to create a new instance of class `CorrCoeffMatcher`, as illustrated in the following example:



(a) (b) (c)

Fig. 17.7
Detection of match points by simple thresholding: correlation coefficient (a), positive values only (b), and values greater than 0.5 (c). The remaining peaks indicate the positions of the six similar (but not identical) tulip patterns in the original image (Fig. 17.4 (a)).

```
1 FloatProcessor imgP = ...  // search image
2 FloatProcessor refP = ...  // reference image
3 CorrCoeffMatcher matcher = new CorrCoeffMatcher(imgP, refP);
4 FloatProcessor matchP = matcher.computeMatch();
```

The correlation coefficient is computed by the method `computeMatch()` and returned as a new image (`matchFp`) of type `FloatProcessor`. The performance can be improved by using direct access to the pixel arrays (instead of the access methods `getf()` and `setf()`; see also Sec. 3.7).

Shape of the template

The shape of the reference image does not need to be rectangular as in the previous examples, although it is convenient for the processing. In some applications, circular, elliptical, or custom-shaped templates may be more applicable than a rectangle. In such a case, the template may still be stored in a rectangular array, but the relevant pixels must somehow be marked (e. g., using a binary mask). Even more general is the option to assign individual continuous weights to the template elements such that, for example, the center of a template can be given higher significance in the match than the peripheral regions. Implementing such a "windowed matching" technique should be straightforward and require only minor modifications to the standard approach.

17.1.3 Matching under Rotation and Scaling

Correlation-based matching methods applied in the way described in this section cannot handle significant rotation or scale differences between the search image and the template. One obvious way to overcome rotation is to match using multiple rotated versions of the template, of course at the price of additional computation time. Similarly, one could try to match using several scaled versions of the template to achieve scale independence to some extent. Although this could be combined by using a set of rotated *and* scaled template patterns, the combinatorially growing number of required matching steps could soon become prohibitive for a practical implementation.

Program 17.1
Class `CorrCoeffMatcher`. This is a direct implementation of Alg. 17.1. The constructor method (lines 11–36) computes the mean $\bar{R} =$ `meanR` (Eqn. (17.8)) and the variance term $S_R =$ `varR` (Eqn. (17.10)) of the reference image R.

```
1  class CorrCoeffMatcher {
2    FloatProcessor I; // image
3    FloatProcessor R; // template
4    int wI, hI;       // width/height of image
5    int wR, hR;       // width/height of template
6    int K;            // size of template
7
8    float meanR;      // mean value of template (R̄)
9    float varR;       // square root of template variance (σ_R)
10
11   public CorrCoeffMatcher( // constructor method
12          FloatProcessor img,  // search image (I)
13          FloatProcessor ref)  // reference image (R)
14   {
15     I = img;
16     R = ref;
17     wI = I.getWidth();
18     hI = I.getHeight();
19     wR = R.getWidth();
20     hR = R.getHeight();
21     K = wR * hR;
22
23     // compute the mean (R̄) and variance term (S_R) of the template:
24     float sumR = 0;      // Σ_R = Σ R(i,j)
25     float sumR2 = 0;     // Σ_R2 = Σ R²(i,j)
26     for (int j = 0; j < hR; j++) {
27       for (int i = 0; i < wR; i++) {
28         float aR = R.getf(i, j);
29         sumR  += aR;
30         sumR2 += aR * aR;
31       }
32     }
33     meanR = sumR / K;    // R̄ = [Σ R(i,j)]/K
34     varR =               // S_R = [Σ R²(i,j) − K·R̄²]^1/2
35          (float) Math.sqrt(sumR2 - K * meanR * meanR);
36   }
37
38   // continued...
```

Solutions to the rotation and scaling problems (such as matching in *logarithmic-polar space* [108]) exist but go beyond the elementary techniques described here. Also interesting in this context are *affine matching* methods, which have received strong interest in recent years, particularly for wide-baseline stereo applications, motion tracking, image retrieval, and panoramic image stitching. These methods rely on local statistical features that are invariant under affine image transformations (including rotation and scaling) [66, 86, 101].

```
40  public FloatProcessor computeMatch() {
41    FloatProcessor C = new FloatProcessor(wI-wR+1, hI-hR+1);
42    for (int r = 0; r <= wI-wR; r++) {
43      for (int s = 0; s <= hI-hR; s++) {
44        float d = getMatchValue(r,s);
45        C.setf(r, s, d);
46      }
47    }
48    return C;
49  }
50
51  float getMatchValue(int r, int s) {
52    float sumI = 0;    // ΣI = Σ I(r+i,s+j)
53    float sumI2 = 0;   // ΣI2 = Σ(I(r+i,s+j))^2
54    float sumIR = 0;   // ΣIR = Σ I(r+i,s+j)·R(i,j)
55
56    for (int j = 0; j < hR; j++) {
57      for (int i = 0; i < wR; i++) {
58        float aI = I.getf(r+i, s+j);
59        float aR = R.getf(i, j);
60        sumI  += aI;
61        sumI2 += aI * aI;
62        sumIR += aI * aR;
63      }
64    }
65    float meanI = sumI / K;          // Ī_{r,s} = ΣI/K
66    return (sumIR - K * meanI * meanR) /
67      ((float)Math.sqrt(sumI2 - K * meanI * meanI) * varR);
68  }
69
70 } // end of class CorrCoeffMatcher
```

Program 17.2
Class `CorrCoeffMatcher` (*continued*). The method `computeMatch()` (lines 40–49) computes the correlation coefficient for the reference image `R` and the corresponding subimage of `I` at all positions (`r`, `s`). The method `getMatchValue(r,s)` (lines 51–70) returns the local correlation coefficient $C(r, s)$ (Eqn. (17.11)).

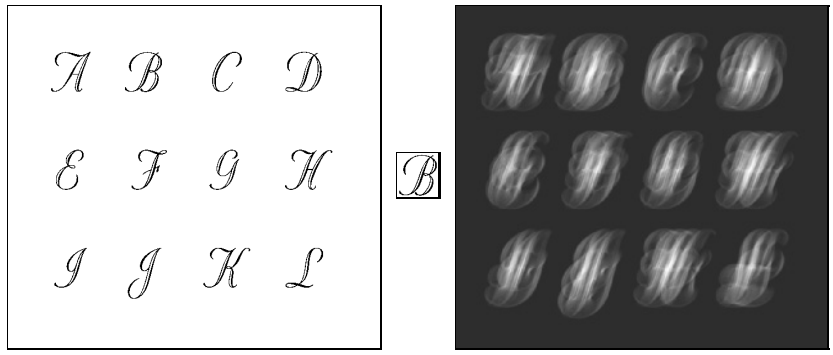
17.2 Matching Binary Images

As became evident in the previous section, the comparison of intensity images based on correlation may not be an optimal solution but is sufficiently reliable and efficient under certain restrictions.

17.2.1 Direct Comparison

If we compare binary images in the same way, by counting the number of identical pixels in the search image and the template, the total difference will only be small when most pixels are in exact agreement. Since there is no continuous transition between pixel values, the distribution produced by a simple distance function will generally be ill-behaved (i. e., highly discontinuous with many local maxima; see Fig. 17.8).

The problem with directly comparing binary images is that even the smallest deviations between image patterns, such as those caused by a



(a) (b) (c)

Fig. 17.8
Direct comparison of binary images. Given are a binary search image (a) and a binary reference image (b). The local similarity value for any template position corresponds to the relative number of matching (black) foreground pixels. High similarity values are shown as bright spots in the result (c). While the maximum similarity is naturally found at the correct position (at the center of the glyph *B*) the match function behaves wildly, with many local maxima.

small shift, rotation, or distortion, can create very high distance values. Shifting a thin line drawing by only a single pixel, for example, may be sufficient to switch from full agreement to no agreement at all (i. e., from zero difference to maximum difference). Thus a simple distance function gives no indication how far away and in which direction to search for a better match position.

Our goal is to find the position where a maximum number of foreground pixels in the search image and the template coincide using a distance function that is smooth and more tolerant against small deviations between the binary image patterns.

17.2.2 The Distance Transform

A first step in this direction is to record the distance to the closest foreground pixel for every position (u, v) in the search image I. This gives us the minimum distance (though not the direction) for shifting a particular pixel onto a foreground pixel. Starting from a binary image $I(u, v) = I(\boldsymbol{p})$, we denote

$$FG(I) = \{\boldsymbol{p} \mid I(\boldsymbol{p}) = 1\}, \tag{17.12}$$

$$BG(I) = \{\boldsymbol{p} \mid I(\boldsymbol{p}) = 0\}, \tag{17.13}$$

as the set of coordinates of the foreground and background pixels, respectively. The so-called distance transform of I, $D(\boldsymbol{p}) \in \mathbb{R}$, is defined as

$$D(\boldsymbol{p}) = \min_{\boldsymbol{p}' \in FG(I)} \text{dist}(\boldsymbol{p}, \boldsymbol{p}') \tag{17.14}$$

for all $\boldsymbol{p} = (u, v)$, where $u = 0 \ldots M-1$, $v = 0 \ldots N-1$ (for image size $M \times N$). If $I(\boldsymbol{p})$ is a foreground pixel itself (i. e., $p \in FG$), then the distance $D(\boldsymbol{p}) = 0$ since no shift is necessary for moving this pixel onto a foreground pixel.

The function $\text{dist}(\boldsymbol{p}, \boldsymbol{p}')$ in Eqn. (17.14) measures the geometric distance between the two coordinate points $\boldsymbol{p} = (u, v)$ and $\boldsymbol{p}' = (u', v')$. Examples of suitable distance functions are the Euclidean distance

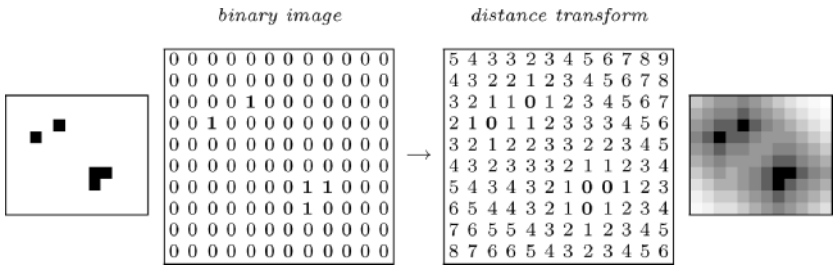


Fig. 17.9
Example of a distance transform of a binary image using the Manhattan distance $\mathrm{d}_M()$.

$$\mathrm{d}_E(\boldsymbol{p}, \boldsymbol{p}') = \|\boldsymbol{p} - \boldsymbol{p}'\| = \sqrt{(u-u')^2 + (v-v')^2} \quad \in \mathbb{R}^+ \tag{17.15}$$

or the *Manhattan distance*[2]

$$\mathrm{d}_M(\boldsymbol{p}, \boldsymbol{p}') = |u-u'| + |v-v'| \quad \in \mathbb{N}_0. \tag{17.16}$$

Figure 17.9 shows a simple example of a distance transform using the Manhattan distance $\mathrm{d}_M()$.

The direct implementation of the distance transform (following the definition in Eqn. (17.14)) is computationally expensive because the closest foreground pixel must be found for each pixel position $\boldsymbol{p}$ (unless $I(\boldsymbol{p})$ is a foreground pixel itself).[3]

Chamfer algorithm

The so-called *chamfer* algorithm [11] is an efficient method for computing the distance transform. Similar to the sequential region labeling algorithm (Alg. 11.2 in Sec. 11.1.2), the chamfer algorithm traverses the image twice by propagating the computed values across the image like a wave. The first traversal starts at the upper left corner of the image and propagates the distance values downward in a diagonal direction. The second traversal proceeds in the opposite direction from the bottom to the top. For each traversal, a "distance mask" is used for the propagation of the distance values; that is,

$$M^L = \begin{bmatrix} m_2^L & m_3^L & m_4^L \\ m_1^L & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \text{and} \quad M^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & m_1^R \\ m_4^R & m_3^R & m_2^R \end{bmatrix} \tag{17.17}$$

for the first and second traversals, respectively. The values in M^L and M^R describe the geometric distance between the current pixel (marked $\times$) and the relevant neighboring pixels. They depend upon the distance

[2] Also called "city block distance".

[3] A simple (brute force) algorithm for the distance transform would perform a full scan over the entire image for each processed pixel, resulting in $\mathcal{O}(N^2 \cdot N^2) = \mathcal{O}(N^4)$ steps for an image of size $N \times N$.

Algorithm 17.2 Chamfer algorithm for computing the distance transform. From the binary image I, the distance transform D (Eqn. (17.14)) is computed using a pair of distance masks (Eqn. (17.17)) for the first and second passes. Notice that the image borders require special treatment.

```
1:  DistanceTransform (I)
        I: binary image of size M × N.
        Returns the distance transform of image I.

        Step 1—initialize:
2:      D ← new distance map of size M × N, D(u, v) ∈ ℝ
3:      for all image coordinates (u, v) do
4:          if I(u, v) = 1 then
5:              D(u, v) ← 0                    ▷ foreground pixel (zero distance)
6:          else
7:              D(u, v) ← ∞                    ▷ background pixel (infinite distance)
        Step 2—L→R pass (using distance mask M^L = m_i^L):
8:      for v ← 1, 2, ..., N−1 do              ▷ top → bottom
9:          for u ← 1, 2, ..., M−2 do          ▷ left → right
10:             if D(u, v) > 0 then
11:                 d1 ← m_1^L + D(u−1, v)
12:                 d2 ← m_2^L + D(u−1, v−1)
13:                 d3 ← m_3^L + D(u, v−1)
14:                 d4 ← m_4^L + D(u+1, v−1)
15:                 D(u, v) ← min(d1, d2, d3, d4)
        Step 3—R→L pass (using distance mask M^R = m_i^R):
16:     for v ← N−2, ..., 1, 0 do              ▷ bottom → top
17:         for u ← M−2, ..., 2, 1 do          ▷ right → left
18:             if D(u, v) > 0 then
19:                 d1 ← m_1^R + D(u+1, v)
20:                 d2 ← m_2^R + D(u+1, v+1)
21:                 d3 ← m_3^R + D(u, v+1)
22:                 d4 ← m_4^R + D(u−1, v+1)
23:                 D(u, v) ← min(D(u, v), d1, d2, d3, d4)
24:     return D.
```

function $\mathrm{dist}(\boldsymbol{p}, \boldsymbol{p}')$ used. In particular, the distance masks for the Manhattan distance (Eqn. (17.16)) are

$$M_M^L = \begin{bmatrix} 2 & 1 & 2 \\ 1 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}, \qquad M_M^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & 1 \\ 2 & 1 & 2 \end{bmatrix}, \tag{17.18}$$

and similarly for the Euclidean distance (Eqn. (17.15))

$$M_E^L = \begin{bmatrix} \sqrt{2} & 1 & \sqrt{2} \\ 1 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}, \qquad M_E^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & 1 \\ \sqrt{2} & 1 & \sqrt{2} \end{bmatrix}. \tag{17.19}$$

Algorithm 17.2 outlines the chamfer method for computing the distance transform $D(u, v)$ for a binary image $I(u, v)$ using 3×3 pixel distance masks. For the Manhattan distance, the chamfer algorithm computes the distance transform *exactly* using the masks in Eqn. (17.18). The result obtained with the mask for the Euclidean distance (Eqn.

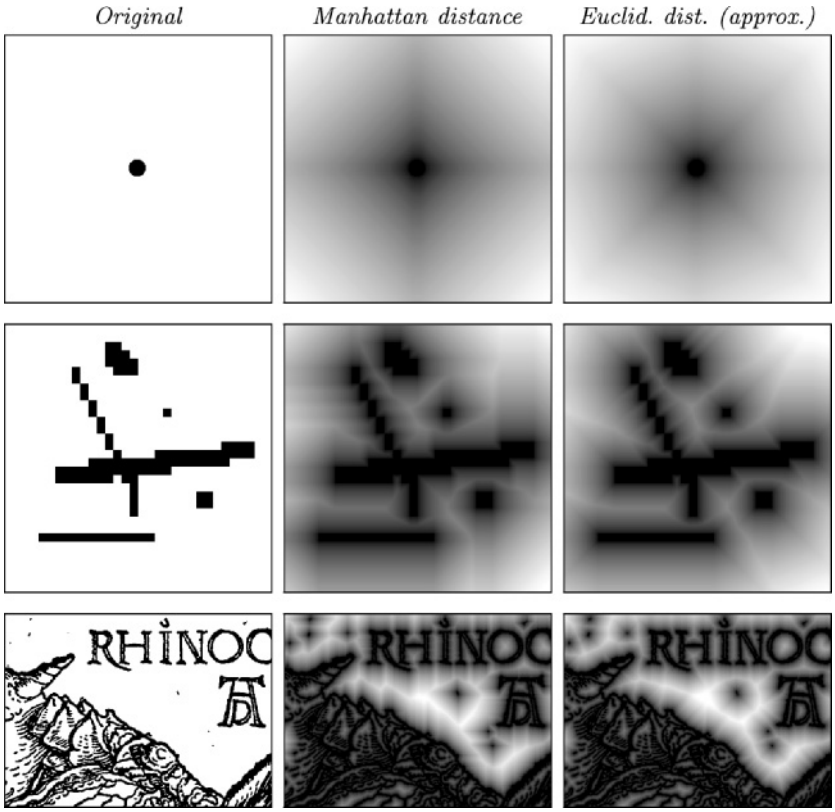


Fig. 17.10
Distance transform with the chamfer algorithm: original image with black foreground pixels (left), and results of distance transforms using the Manhattan distance (center) and the Euclidean distance (right). The brightness (scaled to maximum contrast) corresponds to the estimated distance to the nearest foreground pixel.

(17.19)) is only an approximation to the actual distance to the nearest foreground pixel, which is nevertheless more accurate than the estimate produced by the Manhattan distance. As demonstrated by the examples in Fig. 17.10, the distances obtained with the Euclidean masks are exact along the coordinate axes and the diagonals but are overestimated (i. e., too high) for all other directions.

A more precise approximation can be obtained with distance masks of greater size (e. g., 5×5 pixels; see Exercise 17.3), which include the exact distances to pixels in a larger neighborhood [11]. Furthermore, floating point-operations can be avoided by using distance masks with scaled integer values, such as the masks

$$M^L_{E'} = \begin{bmatrix} 4 & 3 & 4 \\ 3 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \text{and} \quad M^R_{E'} = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & 3 \\ 4 & 3 & 4 \end{bmatrix} \tag{17.20}$$

for the Euclidean distance. Compared with the original masks (Eqn. (17.19)), the resulting distance values are scaled by about a factor of 3.

17.2.3 Chamfer Matching

The chamfer algorithm offers an efficient way to approximate the distance transform for a binary image of arbitrary size. The next step is to use the distance transform for matching binary images. *Chamfer matching* (first described in [7]) uses the distance transform to localize the points of maximum agreement between a binary search image I and a binary reference image (template) R. Instead of counting the overlapping foreground pixels as in the direct approach (Sec. 17.2.1), chamfer matching uses the accumulated values of the distance transform as the match score Q. At each position (r, s) of the template R, the distance values corresponding to all foreground pixels in R are accumulated,

$$Q(r,s) = \frac{1}{K} \cdot \sum_{(i,j)\in FG(R)} D(r+i,\, s+j), \tag{17.21}$$

where $K = |FG(R)|$ denotes the number of foreground pixels in the template R.

The complete procedure for computing the match score Q is summarized in Alg. 17.3. If at some position each foreground pixel in the template R coincides with a foreground pixel in the image I, the sum of the distance values is zero, which indicates a perfect match. The more foreground pixels of the template fall onto distance values greater than zero, the larger is the resulting score value Q (sum of distances). The best match is found at the global minimum of Q,

$$\boldsymbol{p}_{\text{opt}} = (r_{\text{opt}}, s_{\text{opt}}) = \operatorname*{argmin}_{(r,s)} Q(r,s). \tag{17.22}$$

The example in Fig. 17.11 demonstrates the difference between direct pixel comparison and chamfer matching using the binary image shown in Fig. 17.8. Obviously the match score produced by the chamfer method is considerably smoother and exhibits only a few distinct local maxima. This is of great advantage because it facilitates the detection of optimal match points using simple local search methods. Figure 17.12 shows another example with circles and squares. The circles have different diameters and the medium-sized circle is used as the template. As this example illustrates, chamfer matching is tolerant against small-scale changes between the search image and the template and even in this case yields a smooth score function with distinct peaks.

While chamfer matching is not a "silver bullet", it is efficient and works sufficiently well if the applications and conditions are suitable. It is most suited for matching line or edge images where the percentage of foreground pixels is small, such as for registering aerial images or aligning wide-baseline stereo images. The method tolerates deviations between the image and the template to a small extent but is of course not generally invariant under scaling, rotation, and deformation. Because the method is based on minimizing the distances to foreground pixels, the

```
1:  ChamferMatch (I, R)
        I: binary search image of size w_I × h_I
        R: binary reference image of size w_R × h_R
        Returns a two-dimensional map of match scores.

        Step 1—initialize:
2:      D ← DistanceTransform(I)                    ▷ see Alg. 17.2
3:      K ← number of foreground pixels in R
4:      Q ← new match map of size (w_I−w_R+1) × (h_I−h_R+1), Q(r,s) ∈ ℝ

        Step 2—compute the match score:
5:      for r ← 0...(w_I−w_R) do                    ▷ place R at (r,s)
6:          for s ← 0...(h_I−h_R) do
                Get match score for template placed at (r,s):
7:              q ← 0
8:              for i ← 0...(w_R−1) do
9:                  for j ← 0...(h_R−1) do
10:                     if R(i,j) = 1 then          ▷ foreground pixel in template
11:                         q ← q + D(r+i, s+j)
12:             Q(r,s) ← q/K
13:     return Q.
```

quality of the results deteriorates quickly when images contain random noise ("clutter") or large foreground regions. One way to reduce the probability of false matches is not to use a *linear* summation (as in Eqn. (17.21)) but add up the *squared* distances,

$$Q_{rms}(r,s) = \sqrt{\frac{1}{K} \cdot \sum_{(i,j)\in FG(R)} D^2(r+i, s+i)} \tag{17.23}$$

("root mean square" of the distances) as the match score between the template R and the current subimage, as suggested in [11]. Also, hierarchical variants of the chamfer method have been proposed [12] to reduce the search effort as well as to increase robustness.

17.3 Exercises

Exercise 17.1. Implement the chamfer-matching method (Alg. 17.2) for binary images using the Euclidean distance and the Manhattan distance.

Exercise 17.2. Implement the exact Euclidean distance transform using a "brute-force" search for each closest foreground pixel (this may take a while to compute). Compare your results with the approximation obtained with the chamfer method (Alg. 17.2), and compute the maximum deviation (in percent).

Algorithm 17.3
Chamfer matching. Given is a binary search image I and a binary reference image (template) R. In step 1, the distance transform D is computed for the image I using the chamfer algorithm (Alg. 17.2). In step 2, the sum of distance values is accumulated for all foreground pixels in template R for each template position (r,s). The resulting scores are stored in the two-dimensional match map D that is returned.

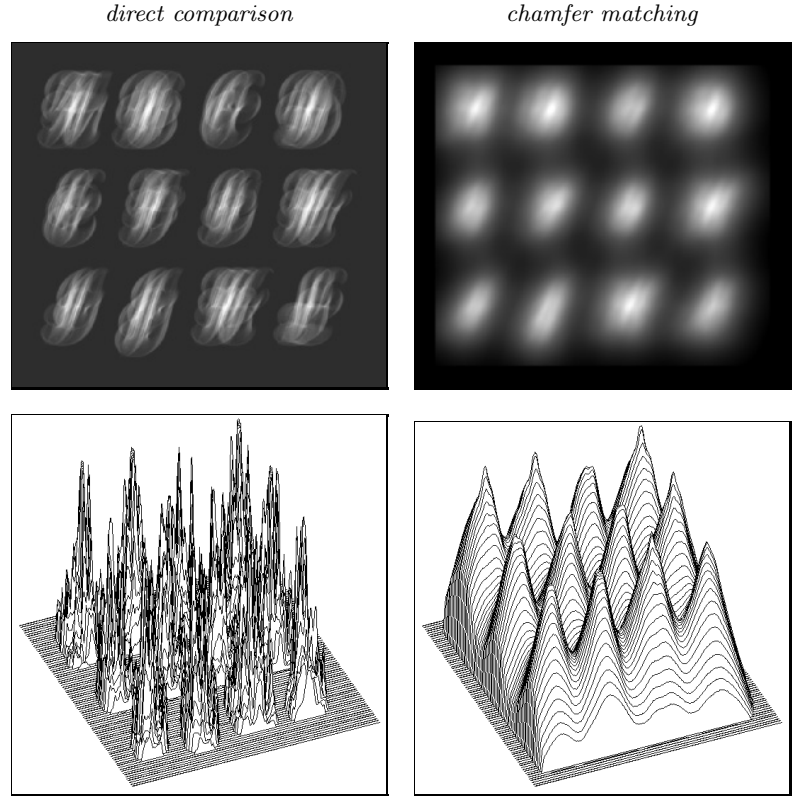


Fig. 17.11 Direct pixel comparison vs. chamfer matching (see original images in Fig. 17.8). Unlike the results of the direct pixel comparison (a), the chamfer match score Q (b) is much smoother. It shows distinct peak values in places of high agreement that are easy to track down with local search methods. The match score Q (Eqn. (17.21)) in (b) is shown inverted for easy comparison.

Exercise 17.3. Modify the chamfer algorithm for computing the distance transform (Alg. 17.2) by replacing the 3×3 pixel Euclidean distance masks (Eqn. (17.19)) with the following masks of size 5×5 pixels:

$$M^L = \begin{bmatrix} \cdot & 2.236 & \cdot & 2.236 & \cdot \\ 2.236 & 1.414 & 1.000 & 1.414 & 2.236 \\ \cdot & 1.000 & \times & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}, \quad M^R = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \times & 1.000 & \cdot \\ 2.236 & 1.414 & 1.000 & 1.414 & 2.236 \\ \cdot & 2.236 & \cdot & 2.236 & \cdot \end{bmatrix}.$$

Compare the results with those obtained with the standard masks. Why are no additional mask elements required along the coordinate axes and the diagonals?

Exercise 17.4. Implement the chamfer-matching technique using (a) the linear summation of distances (Eqn. (17.21)) and (b) the summation of squared distances (Eqn. (17.23)) for computing the match score. Select suitable test images to find out if version (b) is really more robust in terms of reducing the number of false matches.

Exercise 17.5. Adapt the template-matching method described in Sec. 17.1 for the comparison of RGB color images.

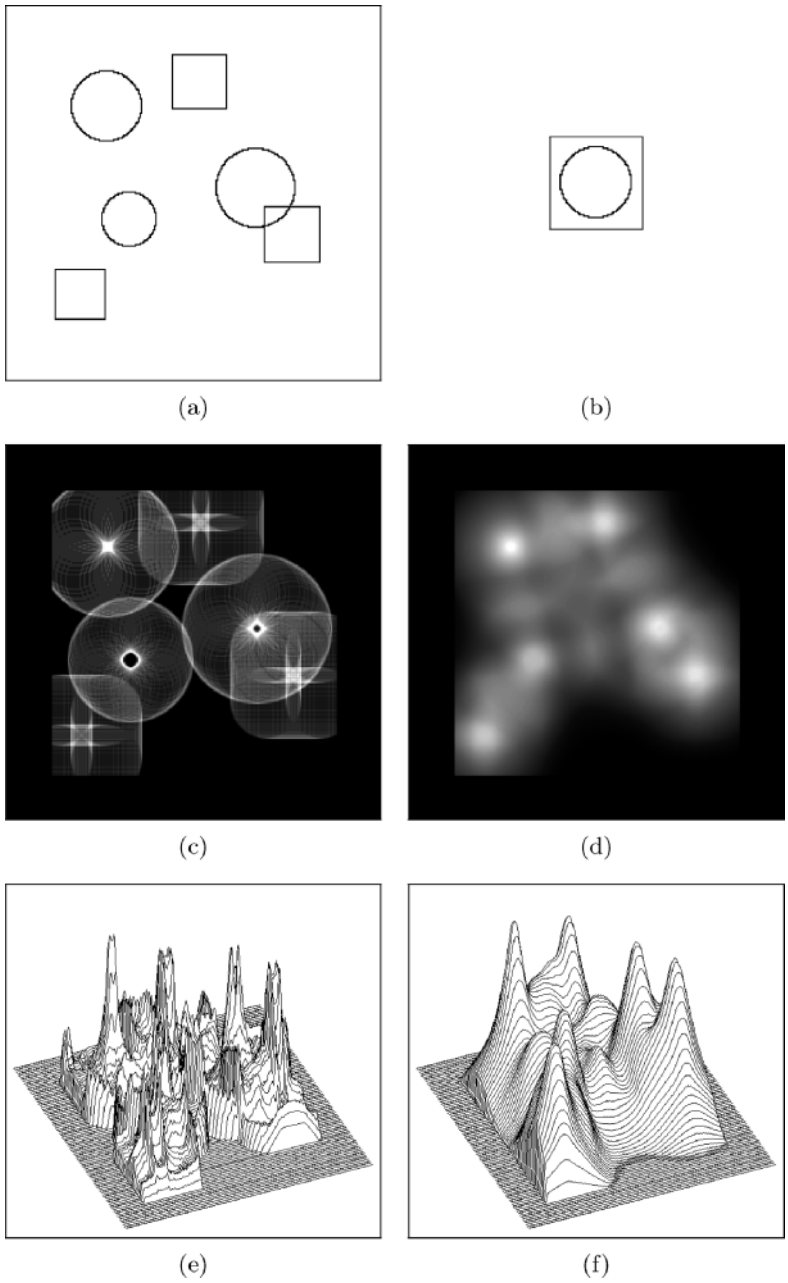


Fig. 17.12
Chamfer matching under varying scales. Binary search image with three circles of different diameters and three identical squares (a). The medium-sized circle at the top is used as the template (b). The result from a direct pixel comparison (c, e) and the result from chamfer matching (d, f). Again the chamfer match produces a much smoother score, which is most notable in the 3D plots shown in the bottom row (e, f). Notice that the three circles and the squares produce high match scores with similar absolute values (f).

Appendix A

Mathematical Notation

A.1 Symbols

The following symbols are used in the main text primarily with the denotations given below. While some symbols may be used for purposes other than the ones listed, the meaning should always be clear in the particular context.

$\{a, b, c, d, \ldots\}$	A *set*; i. e., an unordered collection of distinct elements. A particular element x can be contained in a set at most once. A set may also be empty ({ }).
$(a_1, a_2, \ldots a_n)$	A *vector*; i. e., a fixed-size collection of elements of the same type. $(a_1, a_2, \ldots a_n)^T$ denotes the *transposed* (i. e., column) vector. In programming, vectors are usually implemented as one-dimensional arrays, with elements being referred to by position (index).
$[c_1, c_2, \ldots c_m]$	A *sequence* or *list*; i. e., a collection of elements of variable length. Elements can be added to the sequence (inserted) or deleted from the sequence. A sequence may be empty ([ ]). In programming, sequences are usually implemented with dynamic data structures, such as linked lists. Java's *Collections* framework (see also Appendix B.2.7) provides numerous ready-to-use implementations.
$\langle \alpha_1, \alpha_2, \ldots \alpha_k \rangle$	A *tuple*; i. e., an ordered list of elements, each possibly of a different type. Tuples are typically implemented as *objects* (in Java or C++) or *structures* (in C) with elements being referred to by name.

$*$	Linear convolution operator (Sec. 6.3.1).
$\circledast$	Linear correlation operator (Sec. 17.1.1).
$\oplus$	Morphological dilation operator (Sec. 10.2.3).
$\ominus$	Morphological erosion operator (Sec. 10.2.4).
∂	Partial derivative operator (Sec. 7.2.1). For example, $\frac{\partial f}{\partial x}(x, y)$ denotes the *first* derivative of the function $f(x, y)$ along the x variable at position (x, y), $\frac{\partial^2 f}{\partial^2 x}(x, y)$ is the *second* derivative, etc.
∇	Gradient. ∇f is the vector of partial derivatives of a multidimensional function f (Sec. 7.2.1).
$\lfloor x \rfloor$	"Floor" of x, the largest integer $z \in \mathbb{Z}$ smaller than $x \in \mathbb{R}$ (i. e., $z = \lfloor x \rfloor \leq x$). For example, $\lfloor 3.141 \rfloor = 3$, $\lfloor -1.2 \rfloor = -2$.
a	Pixel value (usually $0 \leq a < K$).
$\mathrm{ArcTan}(x, y)$	Inverse tangent function, similar to $\arctan\left(\frac{y}{x}\right) = \tan^{-1}\left(\frac{y}{x}\right)$ but with two arguments and returning angles in the range $[-\pi, +\pi]$ (i. e., covering all four quadrants). It corresponds to the `ArcTan(x,y)` function in Mathematica and the Java method `Math.atan2(y,x)` (Secs. 7.3, B.1.6).
$\mathrm{card}\{\ldots\}$	Cardinality (size) of a set, $\mathrm{card}\, A \equiv \lvert A \rvert$ (Sec. 4.1).
DFT	Discrete Fourier transform (Sec. 13.3).
$\mathcal{F}$	Continuous Fourier transform (Sec. 13.1.4).
$g(x)$, $g(x, y)$	One- and two-dimensional *continuous* functions $(x, y \in \mathbb{R})$.
$g(u)$, $g(u, v)$	One- and two-dimensional *discrete* functions $(u, v \in \mathbb{Z})$.
$G(m)$, $G(m, n)$	One- and two-dimensional discrete Fourier spectra $(m, n \in \mathbb{Z})$.
$\mathsf{h}(i)$	Histogram of an image at pixel value (or bin) i (Sec. 4.1).
$\mathsf{H}(i)$	Cumulative histogram of an image at pixel value (or bin) i (Sec. 4.6).
$I(u, v)$	Intensity value of the image I at (integer) position (u, v).
i	Imaginary unit, $\mathrm{i}^2 = -1$ (see Sec. 1.3).

K	Number of possible pixel values.
M, N	Number of columns (width) and rows (height) of an image ($0 \leq u < M$, $0 \leq v < N$).
mod	Modulus operator: $(a \bmod b)$ is the remainder of the integer division a/b (Sec. 13.4, B.1.2).
$\mathsf{p}(i)$	Probability density function (Sec. 5.6.1).
$\mathsf{P}(i)$	Probability distribution function or cumulative probability density (Sec. 5.6.1).
$\mathcal{Q}$	Quadrilateral (Sec. 16.1.4).
round(x)	Rounding function: rounds x to the nearest integer. round$(x) = \lfloor x + 0.5 \rfloor$ (Sec. 16.3.1).
truncate(x)	Truncation function: truncates x toward zero to the closest integer. For example, truncate$(3.141) = 3$, truncate$(-2.5) = -2$.
$\mathcal{S}_1$	Unit square (Sec. 16.1.4).

A.2 Set Operators

$\lvert A \rvert$	The size (number of elements) of the set A (equivalent to card A).
$\forall_x \ldots$	"All" quantifier (for all x, ...).
$\exists_x \ldots$	"Exists" quantifier (there is some x for which ...).
$\cup$	Set union (e. g., $A \cup B$).
$\cap$	Set intersection (e. g., $A \cap B$).
$\bigcup_{\mathcal{R}_i}$	Union over multiple sets $\mathcal{R}_i$.
$\bigcap_{\mathcal{R}_i}$	Intersection over multiple sets $\mathcal{R}_i$.

A.3 Complex Numbers

Definitions:

$$z = a + \mathrm{i}b, \qquad z, \mathrm{i} \in \mathbb{C},\ a, b \in \mathbb{R},\ i^2 = -1, \tag{A.1}$$

$$z^* = a - \mathrm{i}b \qquad \text{(conjugate complex)}, \tag{A.2}$$

$$sz = sa + \mathrm{i}sb, \qquad s \in \mathbb{R}, \tag{A.3}$$

$$|z| = \sqrt{a^2 + b^2}, \quad |sz| = s|z|, \tag{A.4}$$

$$\begin{aligned} z &= a + \mathrm{i}b \\ &= |z| \cdot (\cos\psi + \mathrm{i}\sin\psi) && \text{(A.5)} \\ &= |z| \cdot e^{\mathrm{i}\psi}, \quad \text{where } \psi = \tan^{-1}(b/a), && \text{(A.6)} \end{aligned}$$

$$\mathrm{Re}\big(a + \mathrm{i}b\big) = a, \quad \mathrm{Re}\big(e^{\mathrm{i}\varphi}\big) = \cos\varphi, \tag{A.7}$$

$$\mathrm{Im}\big(a + \mathrm{i}b\big) = b, \quad \mathrm{Im}\big(e^{\mathrm{i}\varphi}\big) = \sin\varphi, \tag{A.8}$$

$$e^{\mathrm{i}\varphi} = \cos\varphi + \mathrm{i} \cdot \sin\varphi, \tag{A.9}$$

$$e^{-\mathrm{i}\varphi} = \cos\varphi - \mathrm{i} \cdot \sin\varphi, \tag{A.10}$$

$$\cos(\varphi) = \tfrac{1}{2} \cdot \big(e^{\mathrm{i}\varphi} + e^{-\mathrm{i}\varphi}\big), \tag{A.11}$$

$$\sin(\varphi) = \tfrac{1}{2i} \cdot \big(e^{\mathrm{i}\varphi} - e^{-\mathrm{i}\varphi}\big), \tag{A.12}$$

Arithmetic operations:

$$\begin{aligned} z_1 &= (a_1 + \mathrm{i}b_1) = |z_1|\, e^{\mathrm{i}\varphi_1}, \\ z_2 &= (a_2 + \mathrm{i}b_2) = |z_2|\, e^{\mathrm{i}\varphi_2}, \end{aligned}$$

$$z_1 + z_2 = (a_1 + b_1) + \mathrm{i}(b_1 + b_2), \tag{A.13}$$

$$\begin{aligned} z_1 \cdot z_2 &= (a_1 a_2 - b_1 b_2) + \mathrm{i}(a_1 b_2 + b_1 a_2) && \text{(A.14)} \\ &= |z_1| \cdot |z_2| \cdot e^{\mathrm{i}(\varphi_1 + \varphi_2)}, && \text{(A.15)} \end{aligned}$$

$$\begin{aligned} \frac{z_1}{z_2} &= \frac{a_1 a_2 + b_1 b_2}{a_2^2 + b_2^2} + \mathrm{i}\,\frac{a_2 b_1 - a_1 b_2}{a_2^2 + b_2^2} && \text{(A.16)} \\ &= \frac{|z_1|}{|z_2|} \cdot e^{\mathrm{i}(\varphi_1 - \varphi_2)}. && \text{(A.17)} \end{aligned}$$

A.4 Algorithmic Complexity and $\mathcal{O}$ Notation

The term "complexity" describes the effort (i. e., computing time or storage) required by an algorithm or procedure to solve a particular problem in relation to the "problem size" n. Often complexity is reported in the literature using "big O" ($\mathcal{O}$) notation [39, Sec. 9.2], as in the following example. Consider a spreadsheet with 20 columns and 30 rows. Obviously, adding up all the entries in the spreadsheet requires performing $30 \cdot 20$ additions. We can be more general by representing the number of columns and rows by M and N, respectively, and saying it requires $M \cdot N$ additions. What if we want to replace each location with the sum of its eight neighbors? Then it would require $M \cdot N \cdot 8$ operations. If we compare these two algorithms, we see that, at their core, both require

doing some number of operations $M \cdot N$ times. Since big O notation factors out constants (such as 8), we could say that the complexity of both of these algorithms is $\mathcal{O}(MN)$.

$\mathcal{O}(MN)$ is an upper bound on the number of operations an algorithm requires on an input of size MN. We can simplify this, since typical images have roughly the same number of rows and columns, by selecting the larger of the rows and columns $n = \max(M, N)$ and replacing it with n. Now, since we know $n \cdot n \geq M \cdot N$ we can say their complexity is $\mathcal{O}(n \cdot n)$ or, more commonly, $\mathcal{O}(n^2)$. Big O notation lets us compare *classes* of algorithms—in this case we discovered that both our algorithms belong to the $\mathcal{O}(n^2)$ class. This tells us that, no matter how much we optimize our code, at the heart our algorithm will require n^2 operations.

Similarly, the direct computation of the linear convolution (Sec. 6.3.1) for an image of size $n \times n$ and a convolution kernel of size $k \times k$ has the time complexity $\mathcal{O}(n^2 k^2)$. As another example, the *fast Fourier transform* (FFT, see Sec. 13.4.2) of a signal vector of length $n = 2^k$ requires only $\mathcal{O}\big(n \log_2(n)\big)$ time.

Additional details on complexity can be found in any good book on computer algorithms, such as [2, 25].

Appendix B

Java Notes

As an undergraduate text for engineering curricula, this book assumes basic programming skills in a procedural language, such as C or Java. The examples in the main text should be easy to understand with the help of some introductory book on Java or one of the many online tutorials. Experience shows, however, that difficulties with some basic Java concepts pertain even at higher levels and frequently cause complications. The following sections aim at resolving some of these typical problem spots.

B.1 Arithmetic

Java is a "strongly typed" programming language, which means in particular that any variable has a fixed type that cannot be altered dynamically. Also, the result of an expression is determined by the types of the involved operands and *not* (in the case of an assignment) by the type of the "receiving" variable.

B.1.1 Integer Division

Division involving integer operands is a frequent cause of errors. If the variables `a` and `b` are both of type `int`, then the expression `(a / b)` is evaluated according to the rules of integer division. The result—the number of times `b` is contained in `a`—is again of type `int`. For example, after the Java statements

```
int a = 2;
int b = 5;
double c = a/b;
```

the value of `c` is *not* 0.4 but 0.0 because the expression `a/b` on the right produces the `int` value 0, which is then automatically converted to the `double` value 0.0.

If we wanted to evaluate `a/b` as a *floating-point* operation (as most pocket calculators do), at least one of the involved operands must be converted to a floating-point value, for example by an explicit type cast `(double)`:

```
double c = (double) a / b;
```

Notice that the type cast `(double)` only applies to the immediately following term (`a`) and not the entire expression `a / b`; i. e., the value of the second operand (`b`) in this division is still of type `int`.

Example

Assume, for example, that we want to scale any pixel value a of an image such that the maximum pixel value $a_{\max}$ is mapped to 255 (see Ch. 5). In mathematical notation, the scaling of the pixel values is simply expressed as

$$c \leftarrow \frac{a}{a_{\max}} \cdot 255,$$

and it may be tempting to convert this 1:1 into Java code, such as

```
int a_max = ip.getMaxValue();
...
  int a = ip.getPixel(u,v);
  int c = (a / a_max) * 255;   ← problem!
  ip.putPixel(u,v,a);
...
```

As we can easily predict, the image will be all black (zero values), except those pixels whose value was `a_max` originally (they are set to 255). The reason is again the division `(a / a_max)` with two operands of type `int`, where the result is zero whenever the divisor (`a_max`) is greater than the dividend (`a`).

Of course, the entire operation could be performed in the floating-point domain by converting one of the operands (as shown earlier), but this is not even necessary in this case. Instead, we may simply swap the order of operations and start with the multiplication,

```
int c = a * 255 / a_max;
```

Why does this work? The subexpression `a * 255` is evaluated first,[1] generating large intermediate values that pose no problem for the subsequent (integer) division.In addition, *rounding* should always be considered to obtain more accurate results when computing fractions of integers (see Sec. B.1.5).

[1] In Java, expressions at the same level are always evaluated in left-to-right order, and therefore no parentheses are required in this example (though they would not do any harm either).

B.1.2 Modulus Operator

The result of the modulus operator

$$a \bmod b$$

(used in several places in the main text) is defined [39, p. 82] as the remainder of the integer division a/b,

$$a \bmod b \triangleq \begin{cases} a & \text{for } b = 0 \\ a - b \cdot \left\lfloor \frac{a}{b} \right\rfloor & \text{otherwise.} \end{cases} \tag{B.1}$$

Unfortunately, this type of mod operator (or an equivalent library method) is not available in the standard Java API. Java's native `%` (*remainder*) operator, defined as

$$a \;\%\; b \triangleq a - b \cdot \text{truncate}\left(\frac{a}{b}\right) \quad \text{for } b \neq 0, \tag{B.2}$$

is often used in this context, but produces the same results only for *positive* operands $a \geq 0$ and $b > 0$. For example,

$$\begin{array}{rcrcr} 13 & \bmod & 4 & \to & 1 \\ 13 & \bmod & -4 & \to & -3 \\ -13 & \bmod & 4 & \to & 3 \\ -13 & \bmod & -4 & \to & -1 \end{array} \qquad \begin{array}{rcrcr} 13 & \% & 4 & \to & 1 \\ 13 & \% & -4 & \to & 1 \\ -13 & \% & 4 & \to & -1 \\ -13 & \% & -4 & \to & -1 \end{array}$$

The following Java method implements the mod operation according to the definition in Eqn. (B.1):

```
static int Mod(int a, int b) {
  if (b == 0)
    return a;
  if (a * b >= 0)
    return a - b * (a / b);
  else
    return a - b * (a / b - 1);
}
```

B.1.3 Unsigned Bytes

Most grayscale and indexed images in Java and ImageJ are composed of pixels of type `byte`, and the same holds for the individual components of most color images. A single byte consists of eight bits and can thus represent $2^8 = 256$ different bit patterns or values, usually mapped to the numeric range $0 \ldots 255$. Unfortunately, Java (unlike C and C++) does *not* provide a suitable "unsigned" 8-bit data type. The primitive Java type `byte` is "signed", using one of its eight bits for the $\pm$ sign, and can represent values in the range $-128 \ldots 127$.

Java's `byte` data can still be used to represent the values 0 to 255, but conversions must take place to perform proper arithmetic computation. For example, after execution of the statements

```
int  a = 200;
byte b = (byte) a;
```

the variables `a` (32-bit `int`) and `b` (8-bit `byte`) contain the binary patterns

```
a = 00000000000000000000000011001000
                              b = 11001000
```

respectively. Interpreted as a (signed) `byte` value, with the leftmost bit[2] as the sign bit, the variable `b` has the decimal value −56. Thus, after the statement

```
int a1 = b;              // a1 == -56
```

the value of the new `int` variable `a1` is −56! To (ab-)use signed `byte` data as *unsigned* data, we can circumvent Java's standard conversion mechanism by disguising the content of `b` as a logic (i. e., nonarithmetic) *bit pattern*; e. g., by

```
int  a2 = (0xff & b);     // a2 == 200
```

where `0xff` (in hexadecimal notation) is an `int` value with the binary bit pattern `00000000000000000000000011111111` and `&` is the bitwise AND operator. Now the variable `a2` contains the right integer value (200) and we thus have a way to use Java's (signed) `byte` data type for storing *unsigned* values. Within ImageJ, access to pixel data is routinely implemented in this way, which is considerably faster than using the convenience methods `getPixel()` and `putPixel()`.

B.1.4 Mathematical Functions (Class `Math`)

Java provides the standard mathematical functions as static methods in class `Math`, as listed in Table B.1. The `Math` class is part of the `java.lang` package and thus requires no explicit import to be used. Most `Math` methods accept arguments of type `double` and also return values of type `double`. As a simple example, a typical use of the cosine function $y = \cos(x)$ is

```
double x;
double y = Math.cos(x);
```

Similarly, the `Math` class defines some common numerical constants as static variables; e. g., the value of π could be obtained by

```
double x = Math.PI;
```

[2] Java uses the standard "2s-complement" representation, where a sign bit = 1 stands for a negative value.

Table B.1
Methods and constants defined by Java's `Math` class.

`double abs(double a)`	`double max(double a, double b)`
`int abs(int a)`	`float max(float a, float b)`
`float abs(float a)`	`int max(int a, int b)`
`long abs(long a)`	`long max(long a, long b)`
`double ceil(double a)`	`double min(double a, double b)`
`double floor(double a)`	`float min(float a, float b)`
`double rint(double a)`	`int min(int a, int b)`
`long round(double a)`	`long min(long a, long b)`
`int round(float a)`	`double random()`
`double toDegrees(double rad)`	`double toRadians(double deg)`
`double sin(double a)`	`double asin(double a)`
`double cos(double a)`	`double acos(double a)`
`double tan(double a)`	`double atan(double a)`
`double atan2(double y, double x)`	
`double log(double a)`	`double exp(double a)`
`double sqrt(double a)`	`double pow(double a, double b)`
`double E`	`double PI`

B.1.5 Rounding

Java's `Math` class (confusingly) offers three different methods for rounding floating-point values:

```
double rint (double x)
long  round (double x)
int   round (float x)
```

For example, a `double` value `x` can be rounded to `int` in one of the following ways:

```
double x; int k;
k = (int) Math.rint(x);
k = (int) Math.round(x);
k = Math.round((float)x);
```

If the argument `x` is known to be positive (as is typically the case with pixel values) rounding can be accomplished without using any method calls by

```
k = (int) (x + 0.5); // works for x ≥ 0 only!
```

In this case, the expression `(x + 0.5)` is first computed as a floating-point (`double`) value, which is then truncated (toward zero) by the explicit `(int)` typecast.

B.1.6 Inverse Tangent Function

The inverse tangent function $\varphi = \tan^{-1}(a)$ or $\varphi = \arctan(a)$ is used in several places in the main text. This function is implemented by the method `atan(double a)` in Java's `Math` class (Table B.1). The return value of `atan()` is in the range $[-\pi \ldots \pi]$ and thus restricted to only two of the four quadrants. Without any additional constraints, the resulting angle is ambiguous. In many practical situations, however, a is given as the ratio of two catheti $(\Delta x, \Delta y)$ of a right-angled triangle in the form

$$\varphi = \tan^{-1}\Big(\frac{\Delta y}{\Delta x}\Big),$$

for which we used the (self-defined) two-parameter function

$$\varphi = \mathrm{ArcTan}(\Delta x, \Delta y)$$

in the main text. The function $\mathrm{ArcTan}(\Delta x, \Delta y)$ is implemented by the static method `atan2(dy,dx)` in Java's `Math` class and returns an unambiguous angle φ in the range $[-\pi \ldots \pi]$; i. e., in any of the four quadrants of the unit circle.[3]

B.1.7 `Float` and `Double` (Classes)

The representation of floating-point numbers in Java follows the IEEE standard, and thus the types `float` and `double` include the values

`POSITIVE_INFINITY`
`NEGATIVE_INFINITY`
`NaN` ("not a number")

These values are defined as constants in the corresponding wrapper classes `Float` and `Double`, respectively. If such a value occurs in the course of some computation (e. g., `POSITIVE_INFINITY` as the result of dividing by zero),[4] Java continues without raising an error.

B.2 Arrays and Collections

B.2.1 Creating Arrays

Unlike in most traditional programming languages (such as FORTRAN or C), arrays in Java can be created *dynamically*, meaning that the size of an array can be specified at runtime using the value of some variable or arithmetic expression. For example:

[3] The function `atan2(dy,dx)` is available in most current programming languages, including Java, C, and C++.

[4] In Java, this only holds for floating-point operations. Integer division by zero still causes an *exception*.

```
int N = 20;
int[] A = new int[N];
int[] B = new int[N*N];
```

Once allocated, however, the size of any Java array is fixed and cannot be subsequently altered. For additional variability, Java provides a number of universal container classes (e. g., the class `Vector`) for a wide range of applications.

After its definition, an array variable can be assigned any other compatible array or the constant value `null`; e. g.,

```
A = B;     // A now points to B's data
B = null;
```

Through the assignment `A = B` above, the array initially referenced by `A` becomes unaccessible and thus turns into *garbage*. In contrast to C and C++, where unnecessary storage needs to be *de*allocated explicitly, this is taken care of in Java by its built-in "garbage collector". It is also convenient that newly created arrays of numerical element types (`int`, `float`, `double`, etc.) are automatically initialized to zero.

B.2.2 Array Size

Since an array may be created dynamically, it is important that its actual size can be determined at runtime. This is done by accessing the `length` attribute[5] of the array:

```
int k = A.length; // number of elements in A
```

It may be surprising that Java arrays may have *zero* (not `null`) elements! If an array has more than one dimension, the size (`length`) along every dimension must be derived separately. The size is a property of the array itself and can therefore be obtained inside any method from array arguments passed to it. Thus (unlike in C, for example) it is not necessary to pass the size of an array as a separate function argument.

B.2.3 Accessing Array Elements

In Java, the index of the first array element is always 0 and the index of the last element is $N-1$ for an array with a total of N elements. To iterate through a one-dimensional array `A` of arbitrary size, one would typically use a construct like

```
for (int i = 0; i < A.length; i++) {
    // do something with A[i]
}
```

Since images in Java and ImageJ are stored as one-dimensional arrays (accessible through the `ImageProcessor` method `getPixels()`), most point operations can be efficiently implemented in this way (see Prog. C.1 on p. 491 for an example).

[5] Notice that the `length` attribute of an array is not a method!

B.2.4 Two-Dimensional Arrays

Multidimensional arrays are a common cause of misunderstanding. In Java, all arrays are one-dimensional, and multidimensional arrays are implemented as one-dimensional arrays of subarrays (Fig. B.1). If, for example, the 3×3 matrix

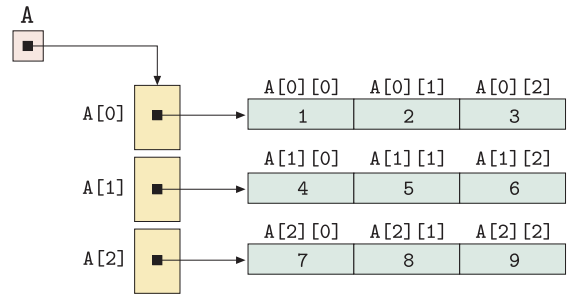
$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{B.3}$$

is represented as a two-dimensional floating-point array,

```
double[][] A = {{1,2,3},
                {4,5,6},
                {7,8,9}};
```

then `A` is really a *one*-dimensional array containing three items, each of which is again a one-dimensional array of type `double`.

Fig. B.1
Multidimensional arrays are implemented in Java as *one*-dimensional arrays whose elements are again one-dimensional arrays.



The usual assumption is that the array elements are arranged in *row-first* ordering, as illustrated in Fig. B.1. The first index thus corresponds to the *row* number r and the second index corresponds to the *column* number c,

$$a_{r,c} \equiv \texttt{A[r][c]}\,.$$

This is quite convenient, because the array initialization in the code segment above looks exactly the same as the original matrix in Eqn. (B.3).

If the matrix represents an image or filter kernel, then of course the row index corresponds to the *vertical* coordinate v (or j) and the column index corresponds to the *horizontal* coordinate u (or i). For example, if we represent the filter kernel

$$H(i,j) = \begin{bmatrix} H(0,0) & H(1,0) & H(2,0) \\ H(0,1) & H(1,1) & H(2,1) \\ H(0,2) & H(1,2) & H(2,2) \end{bmatrix} = \begin{bmatrix} -1 & -2 & 0 \\ -2 & 0 & 2 \\ 0 & 2 & 1 \end{bmatrix}$$

as a two-dimensional Java array,

```
double[][] H = {{-1,-2, 0},
                {-2, 0, 2},
                { 0, 2, 1}};
```

then the indices must be *reversed* in order to access the right elements. In this particular case,

$$H(i,j) \equiv \texttt{H[j][i]}.$$

This scheme was used, for example, for implementing the 3×3 filter plugin in Prog. 6.2 (p. 94).

Size of Multi-Dimensional Arrays

The size of a multidimensional array can be obtained by querying the size of its subarrays. For example, given the following three-dimensional array with dimensions $P \times Q \times R$,

```
int B[][][] = new int[P][Q][R];
```

the size of `B` along its three dimensions is obtained by the statements

```
int p = B.length;         // = P
int q = B[0].length;      // = Q
int r = B[0][0].length;   // = R
```

At least this works for "rectangular" Java arrays, multidimensional arrays with all subarrays at any level being of the *same* length. If this is not the case, the length of each subarray must be determined individually to avoid "index-out-of-bounds" errors. Thus a "bullet-proof" iteration over all elements of a three-dimensional—potentially "non-rectangular"—array `C` should be implemented as follows:

```
for (int i = 0; i < C.length; i++) {
  for (int j = 0; j < C[i].length; j++) {
    for (int k = 0; k < C[i][j].length; k++) {
      // do something with C[i][j][k]
    }
  }
}
```

B.2.5 Cloning Arrays

Java arrays implement the standard `java.lang.Cloneable` interface and provide `clone()` methods to perform a single-level ("shallow") form of duplication; i. e., to make a copy of the top-level structure of the array. Applied to a one-dimensional array of primitive element type, e. g.,

```
int[] A1 = {1,2,3,4};
int[] A2 = (int[]) A1.clone();
```

Program B.1
Utility method `duplicateArray()` for cloning arrays of any element type and dimensionality. Objects inside the array are not duplicated.

```
1  import java.lang.reflect.Array;
2
3  public static Object duplicateArray(Object orig) {
4    Class origClass = orig.getClass();
5    if (!origClass.isArray())
6      return null; // no array to duplicate
7    Class compType = origClass.getComponentType();
8    int n = Array.getLength(orig);
9    Object dup = Array.newInstance(compType, n);
10   if (compType.isArray()) // array elements are arrays again:
11     for (int i = 0; i < n; i++)
12       Array.set(dup, i, duplicateArray(Array.get(orig, i)));
13   else // array elements are objects or primitives:
14     System.arraycopy(orig, 0, dup, 0, n);
15   return dup;
16 }
```

the result `A2` is an exact and independent copy of the array `A1`, as one would expect. If the original array contains real (i. e., nonprimitive) Java *objects*, `clone()` does *not* duplicate the individual objects themselves, but the cells of both arrays refer to the same original objects.

Similarly, applying `clone()` to a two-dimensional (or multidimensional) array duplicates only the top-level structure of that array but none of its subarrays. Java has no standard method for doing a *full-depth* duplication of multidimensional arrays. The (nontrivial) method `duplicateArray()` in Prog. B.1 shows how this could be accomplished recursively for arrays of any element type and dimensionality.

B.2.6 Arrays of Objects, Sorting

In Java, as mentioned earlier, we can create arrays dynamically; i. e., the size of an array can be specified during execution. This is convenient because we can adapt the size of the arrays to the actual problem. For example, we could write

```
Corner[] cornerArray = new Corner[n];
```

to create an array that can hold `n` objects of type `Corner` (as defined in Sec. 8.3). But be aware that the new array is not filled with corners yet but initialized with `null` (i. e., empty references), so the array is really empty. We can insert a `Corner` object into its first (or any other) cell by

```
cornerArray[0] = new Corner(10,20,6789.0f);
```

Arrays can be sorted quickly using the static utility methods in the `java.util.Arrays` class,

```
Arrays.sort(type[] arr)
```

where `arr` can be any array of primitive *`type`* (`int`, `float`, etc.) or an array of objects. In the latter case, the array may not have `null` entries. Also, the class of every contained object must implement the `Comparable` interface, i. e., provide a public method

```
int compareTo(Object obj)
```

that must return an `int` value of −1, 0, or 1, depending upon the intended order relation to the other object `obj`. For example, within the `Corner` class, the `compareTo()` method could be defined as follows:

```
public int compareTo (Object obj){    // in class Corner
  Corner c2 = (Corner) obj;
  if (this.q > c2.q) return -1;
  if (this.q < c2.q) return 1;
  else return 0;
}
```

which implicitly assumes that objects of class `Corner` need never be compared with any other type of object.[6]

In summary, arrays are highly efficient data structures that allow fast searching and sorting and therefore should be used whenever fixed size is not a problem.

B.2.7 Collections

Once created, arrays in Java are of fixed size and cannot be expanded or shrunk. To use an array for collecting the corners detected in an image may thus not be a good idea because we do not know a priori how many corners the image contains. If we make the initial array too small, we will run out of space during the process. If we make the array as large as possibly needed, we will probably waste a lot of memory most of the time.

When we try to extract entities (e. g., corner points) from images, we do not know in advance how many of them we are going to find. Also, the properties of these items of interest may vary. This is a frequent situation, and while most simple processes in digital imaging are done with fixed-sized arrays of numbers, dynamic data structures are often needed for advanced tasks. Incidentally, this is also one of Java's strongest aspects. In fact, Java provides a complete collection framework with several convenient data structures that would be complicated to implement by oneself.

A "collection" represents a group of objects, known as its elements. So arrays, which we have been using over and over again, are of course collections. The Java collections framework is a unified architecture for

[6] Note that the typecast `(Corner)obj` (line 2 in method `compareTo`) is potentially dangerous and will create a runtime exception if `obj` is not of type `Corner`.

representing and manipulating collections, allowing them to be manipulated independently of the details of their representation. It reduces programming effort while delivering high performance. It allows for interoperability among unrelated APIs, reduces effort in designing and learning new APIs, and fosters software reuse. The framework is based on six collection interfaces. It includes implementations of these interfaces and algorithms to manipulate them. Some types of collections allow duplicate elements and others do not, and some collections are ordered and others unordered.

The Java SDK does not provide any *direct* implementations of this interface but implements more specific subinterfaces such as `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired. Concrete implementations of the `Collection` interface include the classes `Vector` (used to collect corners in Sec. 8.3 (p. 150)) and `ArrayList`, as well as `HashSet` for the convenient construction of hash tables.

Additional details and application examples can be found in the Java SDK documentation. For general hints on effective programming in Java, [9] is a particularly valuable source.

Appendix C

ImageJ Short Reference

C.1 Installation and Setup

The most up-to-date information about downloading and installation is found on the ImageJ Website

http://rsb.info.nih.gov/ij/.

Currently this site contains complete installation packages for Linux (x86), Macintosh (OS 9, OS X), and Windows. The following information mainly refers to the Windows installation but is quite similar for the other platforms.

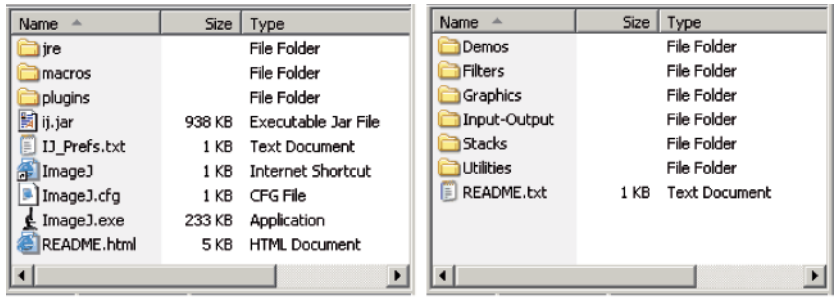
ImageJ can be installed in any file directory (which we refer to as `<ij>`) and can be used without installing any additional software (including the Java runtime). Figure C.1 (a) shows the contents of the installation directory (under Windows) with the following main contents:

`<ij>/jre`
: A complete Java runtime environment, the "Java Virtual Machine" (JVM). This is required for actually executing Java programs.

`<ij>/macros`
: Directory containing ImageJ *macros*, short programs written in ImageJ's macro language (not covered here).

`<ij>/plugins`
: This directory contains all ImageJ plugins written by the user. It comes with some simple example plugins stored in subdirectories (Fig. C.1 (b)). Notice that user-defined plugins may not be located deeper than one level below the `plugins` directory. Otherwise the plugins are not recognized by ImageJ.

`<ij>/ij.jar`
: A Java archive file that contains the entire core functionality of ImageJ. Only this file needs to be replaced when ImageJ is updated to



(a) (b)

Fig. C.1 ImageJ installation under Windows. Contents of the installation directory `<ij>` (a) and its subdirectory `<ij>/plugins` (b).

a newer version. JAR files are ZIP-compressed archives containing collections of binary Java (`.class`) files.

`<ij>/IJ_Prefs.txt`
: A text file used to define various settings and user options for ImageJ.

`<ij>/ImageJ.cfg`
: Specifies the path to launch Java and startup parameters for the Java runtime. Under Windows, this is typically by the lines

```
.
jre\bin\javaw.exe
-Xmx340m -cp ij.jar ij.ImageJ
```

The option `-Xmx340m` in this case specifies that 340 MB of storage are allocated for the Java process. This may be too small for some applications and can be increased (up to about 1.7 GB on a 32-bit system) by editing this file or through the Edit→Options→Memory menu in ImageJ.

`<ij>/ImageJ.exe`
: A small launch program that invokes Java and ImageJ and can be used like any native Windows program.

For writing new plugin programs, we also need a text *editor* for editing the Java source files and a Java *compiler*. The Java runtime environment (JRE) included with ImageJ contains both, even a compiler,[1] such that no additional software is required to get started. However, this basic programming environment is insufficient in practice even for small projects. Instead, it is recommended to embark on one of the freely available integrated Java programming environments, such as *Eclipse*,[2] *NetBeans*,[3] or *Borland JBuilder*.[4] These products also give superior sup-

[1] Unfortunately, the built-in compiler (contained in `jre/lib/ext/tools.jar`) does *not* support the language features introduced with Java 1.5 or higher and is thus incompatible with many examples in this book.
[2] www.eclipse.org.
[3] www.netbeans.org.
[4] www.borland.com/jbuilder.

port for managing larger plugin projects and provide context-dependent editing capabilities and advanced syntax analysis, which help to avoid many programming errors that may otherwise cause fatal execution errors.

C.2 ImageJ API

The complete documentation and source code for the ImageJ API[5] is available online at

http://rsb.info.nih.gov/ij/developer/.

Both are extremely helpful resources for developing new ImageJ plugins, as is the ImageJ programming tutorial written by Werner Bailer [4]. In addition, the standard Java API documentation (available online at Sun Microsystems[6]) should always be at hand for any serious Java programming. In the following, we give a brief description of the most important packages and classes in the ImageJ API.[7]

C.2.1 Images and Processors

While the ImageJ API makes it easy to work with images on the programming level, their internal representation is fairly complex and incorporates several objects of different classes. Some of these classes are unique to ImageJ, while others are standard Java (AWT) classes or derived from standard classes. Figure C.2 contains a simplified diagram that shows the relationships between the key image objects.

The actual image data (pixels) are stored in either an `ImageProcessor` or `ImageStack` object, depending on whether it is a single image or a sequence (stack) of images, respectively. `ImageProcessor` or `ImageStack` objects can be used to process images but have no screen representation. Visible images are based on an `ImagePlus` object, which links to an AWT `Image` and `ImageWindow` (a subclass of `java.awt.Frame`) to map the image's pixel data onto the screen.

C.2.2 Images (Package `ij`)

`ImagePlus` (class)
: This is an extended variant of the standard Java class `java.awt.Image` for representing images (Fig. C.3). An `ImagePlus` object represents an image (or image sequence) that can be displayed on the screen. It contains an instance of the class `ImageProcessor` (see below) that is not visible but provides the functionality for processing the corresponding image.

[5] Application programming interface.
[6] http://java.sun.com/reference/api/.
[7] The UML diagrams in Figs. C.3–C.6 are taken from the ImageJ Website.

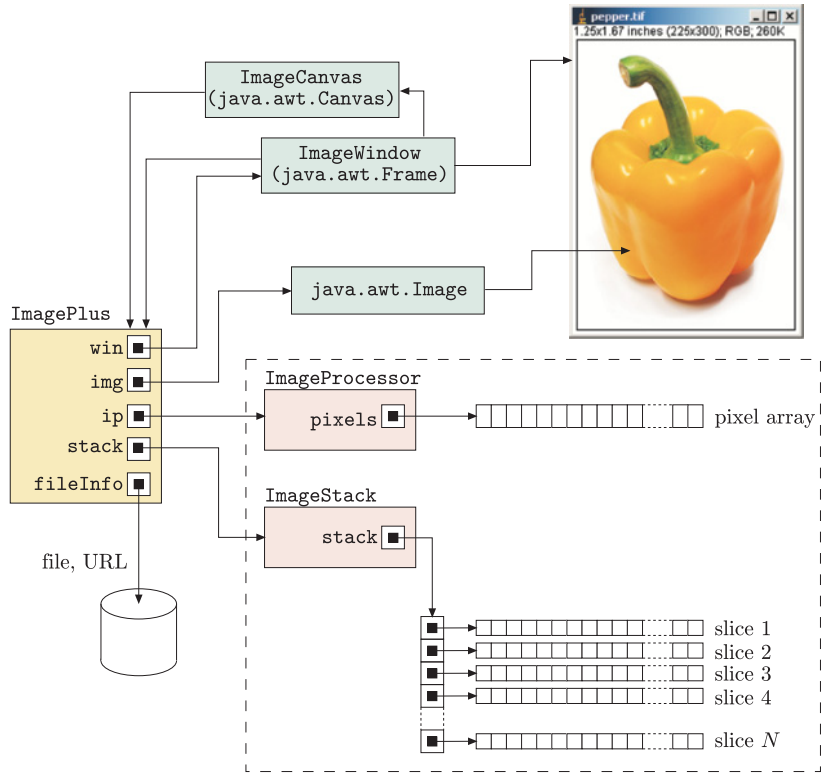


Fig. C.2
Internal representation of images and image stacks in ImageJ (simplified). `ImageProcessor` and `ImageStack` objects contain the actual pixel data of images and image sequences (stacks), respectively. A single image is stored in memory as a one-dimensional array of numerical pixel values. Image stacks are stored as a one-dimensional array of pixel arrays. `ImageProcessor` and `ImageStack` objects can be used to process and convert images but are not necessarily visible on screen. Opening, storing, and displaying an image or image stack requires an `ImagePlus` object, which uses standard AWT mechanisms for mapping to the screen (classes `Image`, `ImageWindow`, `ImageCanvas`).

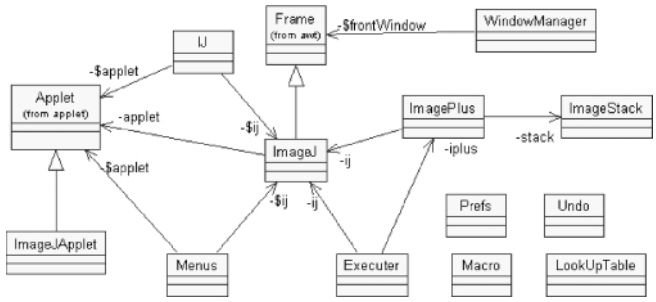
`ImageStack` (class)
An extensible sequence ("stack") of images that is usually attached to an `ImagePlus` object (see Fig. C.2).

C.2.3 Image Processors (Package `ij.process`)

`ImageProcessor` (class)
This is the (abstract) superclass for the four image processor classes available in ImageJ: `ByteProcessor`, `ShortProcessor`, `FloatProcessor`, `ColorProcessor` (Fig. C.4). Processing images is mainly accomplished with objects of class `ImageProcessor` or one of its subclasses, while `ImagePlus` objects (see above) are mostly used for displaying and interacting with images.

`ByteProcessor` (class)
Image processor for 8-bit (`byte`) grayscale and indexed color images. The derived subclass `BinaryProcessor` implements binary images that may only contain pixel values 0 and 255.

`ShortProcessor` (class)
Image processor for 16-bit grayscale images.



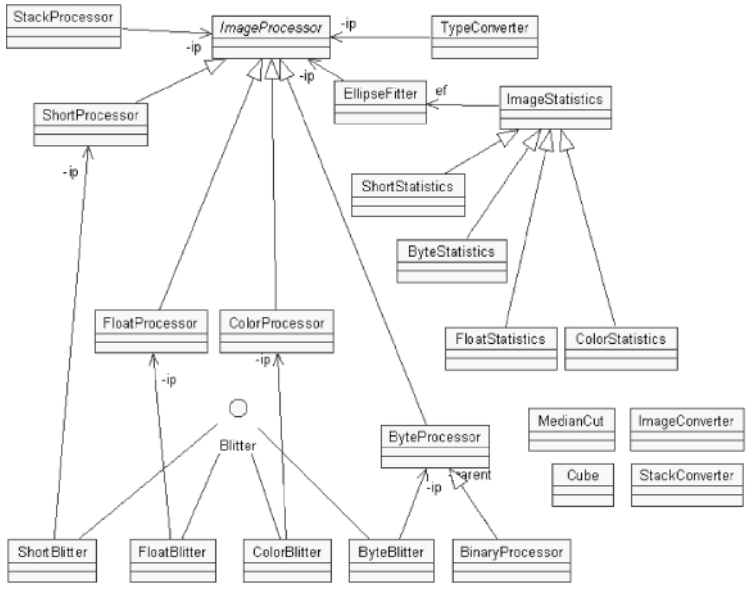


Fig. C.3
Class diagram for ImageJ package `ij`.

Fig. C.4
Class diagram for the ImageJ package `ij.process`.

`FloatProcessor` (class)
: Image processor for 32-bit floating-point (`float`) images.

`ColorProcessor` (class)
: Image processor for 32-bit color (3×8 bits RGB plus 8-bit α-channel) images.

C.2.4 Plugins (Packages `ij.plugin`, `ij.plugin.filter`)

`PlugIn` (interface)
: Interface for generic ImageJ plugins that import or display images or plugins that do not use any images.

`PlugInFilter` (interface)
: Interface for ImageJ plugins that process (and usually modify) images.

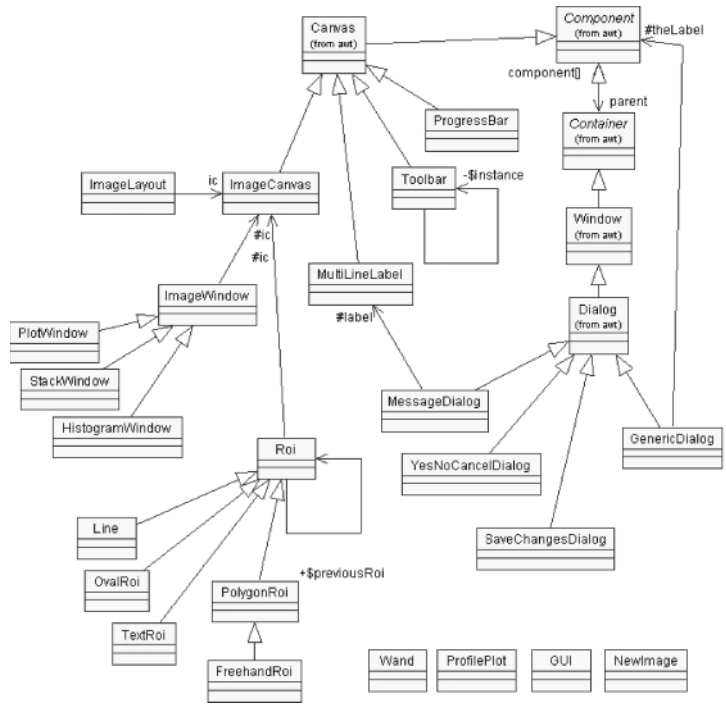


Fig. C.5
Class diagram for the ImageJ package `ij.gui`.

C.2.5 GUI Classes (Package `ij.gui`)

ImageJ's GUI[8] classes provide the basic functionality for displaying and interacting with images (Fig. C.5):

`ColorChooser` (class)
: Displays a dialog window for interactive color selection.

`NewImage` (class)
: Provides the functionality for creating new images interactively and through static methods (see Sec. C.3.4).

`GenericDialog` (class)
: Provides configurable dialog windows with a set of standard interaction fields.

`ImageCanvas` (class)
: This subclass of the standard Java class `java.awt.Canvas` describes the mapping (source rectangle, zoom factor) for displaying the image in a window. It also handles the mouse and keyboard events sent to that window.

`ImageWindow` (class)
: This subclass of the standard Java class `java.awt.Frame` represents a screen window for displaying images of type `ImagePlus`. An object of class `ImageWindow` contains an instance of class

[8] Graphical user interface.

`ImageCanvas` (see above) for the actual presentation of the image.

`Roi` (class)
Defines a rectangular "region of interest" (ROI) and is the superclass of other ROI classes: `Line`, `OvalRoi`, `PolygonRoi` (with subclass `FreehandRoi`), and `TextRoi`.

C.2.6 Window Management (Package `ij`)

`WindowManager` (class)
Provides a set of static methods to manage ImageJ's screen windows (Fig. C.3).

C.2.7 Utility Classes (Package `ij`)

`IJ` (class)
Provides a set of static utility methods, including methods for selecting, creating, opening, and saving images and obtaining information about the operating environment (Sec. C.21.2).

C.2.8 Input-Output (Package `ij.io`)

The `ij.io` package contains classes for reading (loading) and writing images from and to files in various image formats and encodings (Fig. C.6).

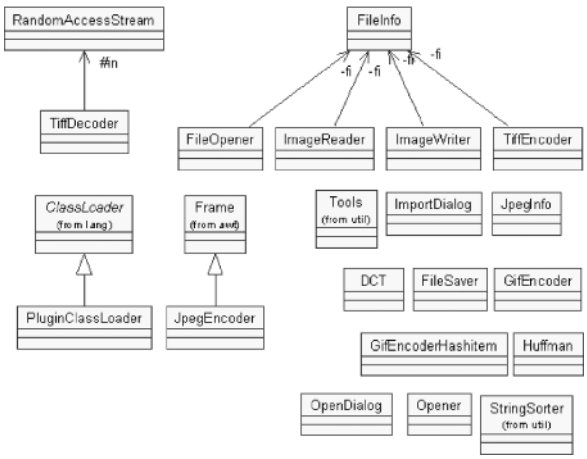


Fig. C.6
Class diagram for ImageJ package `ij.io`.

C.3 Creating Images and Image Stacks

In ImageJ, images, image stacks, and image processors can be created in a variety of different ways, either from existing images or from scratch.

C.3.1 ImagePlus (Class)

The class `ImagePlus` implements the following constructor methods for creating images:

`ImagePlus ()`
: Constructor method: creates a new `ImagePlus` object without initialization.

`ImagePlus (String` *`pathOrURL`*`)`
: Constructor method: opens the image *file* (TIFF, BMP, DICOM, FITS, PGM, GIF, or JPEG format) or *URL* (TIFF, DICOM, GIF, or JPEG format) specified by the location *`pathOrURL`* in a new `ImagePlus` object.

`ImagePlus (String` *`title`*`, Image` *`img`*`)`
: Constructor method: creates a new `ImagePlus` object with the name *`title`* from a given image *`img`* of the standard Java type `java.awt.Image`.

`ImagePlus (String` *`title`*`, ImageProcessor` *`ip`*`)`
: Constructor method: creates a new `ImagePlus` image with the name *`title`* from a given `ImageProcessor` object *`ip`*.

`ImagePlus (String` *`title`*`, ImageStack` *`stack`*`)`
: Constructor method: creates a new `ImagePlus` object with the name *`title`* from a given image stack.

Other methods:

`ImageStack createEmptyStack ()`
: Creates a new, *empty* stack with the same width, height, and color table as the given `ImagePlus` object to which this method is applied.

`ImageStack getStack ()`
: Returns the image stack associated with the `ImagePlus` object to which this method is applied. If no stack exists, a new single-slice stack is created with the contents of that image (by calling `createEmptyStack()`).

C.3.2 ImageStack (Class)

The class `ImageStack` (in package `ij`) provides the following constructor methods for creating image stacks (usually contained inside an `ImagePlus` object):

`ImageStack (int width, int height)`
Constructor method: creates a new, empty image stack of size *width* × *height*.

`ImageStack (int width, int height, ColorModel cm)`
Constructor method: creates a new, empty image stack of size *width* × *height* with the color model *cm* (of type `java.awt.image.ColorModel`).

C.3.3 IJ (Class)

`static ImagePlus createImage (String title, String type, int width, int height, int slices)`
Creates a new `ImagePlus` object. *type* should contain the string `"8"`, `"16"`, `"32"`, or `"RGB"` for creating 8-bit grayscale, 16-bit grayscale, float, or RGB images, respectively. In addition, *type* can be used to specify a fill option by attaching the string `"white"`, `"black"`, or `"ramp"` (the default is `"white"`). For example, the *type* string `"16ramp"` would specify a 16-bit grayscale image initially filled with a black-to-white ramp. *width* and *height* specify the size of the image, and *slices* specifies the number of stack slices (use 1 for a single image). The new image is returned but not automatically displayed (use `show()`).

`static void newImage (String title, String type, int width, int height)`
Creates a new image and displays it. The meaning of the parameters is the same as above. No reference to the new image is returned (`IJ.getimage()` may be used to obtain the active image).

C.3.4 NewImage (Class)

The class `NewImage` (in package `ij.gui`) implements several static methods for creating single images of type `ImagePlus` and image stacks:

`static ImagePlus createByteImage (String title, int width, int height, int slices, int fill)`
Creates a single 8-bit grayscale image or stack (if *slices* > 1) of size *width* × *height* with the name *title*. Admissible values for the *fill* argument are the constants `NewImage.FILL_BLACK`, `NewImage.FILL_WHITE`, and `NewImage.FILL_RAMP`.

`static ImagePlus createShortImage (String title, int width, int height, int slices, int fill)`
Creates a single 16-bit grayscale image or stack.

`static ImagePlus createFloatImage (String title, int width, int height, int slices, int fill)`
Creates a single 32-bit `float` image or stack.

`static ImagePlus createRGBImage (String title, int width, int height, int slices, int fill)`
Creates a single 32-bit RGB image or stack.

`static ImagePlus createImage (String title, int width, int height, int slices, int bitDepth, int fill)`
Generic method that creates and returns an 8-bit grayscale, 16-bit grayscale, float, or RGB image depending upon the value of *bitDepth*, which can be 8, 16, 32, or 24, respectively. The other parameters have the same meanings as above.

C.3.5 ImageProcessor (Class)

`java.awt.Image createImage ()`
Creates a copy of the `ImageProcessor` object and returns it as a standard Java AWT image.

C.4 Creating Image Processors

In ImageJ, `ImageProcessor` objects represent images that can be created, processed, and destroyed but are not generally visible on the screen (see Sec. 3.14 on how to display images).

C.4.1 ImagePlus (Class)

`ImageProcessor getProcessor ()`
Returns a reference to the image's `ImageProcessor` object. If there is no `ImageProcessor`, a new one is created. Returns `null` if this image contains no `ImageProcessor` and no AWT image.

`void setProcessor (String title, ImageProcessor ip)`
Replaces the image's current `ImageProcessor`, if any, by *ip*. If *title* is `null`, the image title remains unchanged.

C.4.2 ImageProcessor (Class)

`ImageProcessor createProcessor (int width, int height)`
Returns a new, blank `ImageProcessor` object of the specified size and the same type as the processor to which this method is applied. This is an abstract method that is implemented by every subclass of `ImageProcessor`.

`ImageProcessor duplicate ()`
Returns a copy of the image processor to which this method is applied. This is an abstract method that is implemented by every subclass of `ImageProcessor`.

C.4.3 `ByteProcessor` (Class)

`ByteProcessor(Image` *`img`*`)`
: Constructor method: creates a new `ByteProcessor` object from an 8-bit image *`img`* of type `java.awt.Image`.

`ByteProcessor(int` *`width`*`, int` *`height`*`)`
: Constructor method: creates a blank `ByteProcessor` object of size *`width`* × *`height`*.

`ByteProcessor(int` *`width`*`, int` *`height`*`, byte[]` *`pixels`*`, ColorModel` *`cm`*`)`
: Constructor method: creates a new `ByteProcessor` object of the specified size and the color model *`cm`* (of type `java.awt.image.ColorModel`), with the pixel values taken from the one-dimensional `byte` array *`pixels`*.

C.4.4 `ColorProcessor` (Class)

`ColorProcessor(Image` *`img`*`)`
: Constructor method: creates a new `ColorProcessor` object from the RGB image *`img`* of type `java.awt.Image`.

`ColorProcessor(int` *`width`*`, int` *`height`*`)`
: Constructor method: creates a blank `ColorProcessor` object of size *`width`* × *`height`*.

`ColorProcessor(int` *`width`*`, int` *`height`*`, int[]` *`pixels`*`)`
: Constructor method: creates a new `ColorProcessor` object of the specified size with the pixel values taken from the one-dimensional `int` array *`pixels`*.

C.4.5 `FloatProcessor` (Class)

`FloatProcessor float[][]` *`pixels`*
: Constructor method: creates a new `FloatProcessor` object from the two-dimensional `float` array *`pixels`*, which is assumed to store the image data as *`pixels`*`[u][v]` (i. e., in column-first order).

`FloatProcessor int[][]` *`pixels`*
: Constructor method: creates a new `FloatProcessor` object from the two-dimensional `int` array *`pixels`*; otherwise the same as above.

`FloatProcessor(int` *`width`*`, int` *`height`*`)`
: Constructor method: creates a blank `FloatProcessor` object of size *`width`* × *`height`*.

`FloatProcessor(int` *`width`*`, int` *`height`*`, double[]` *`pixels`*`)`
: Constructor method: creates a new `FloatProcessor` object of the specified size with the pixel values taken from the one-dimensional `double` array *`pixels`*. The resulting image uses the default grayscale color model.

`FloatProcessor(int width, int height, int[] pixels)`
Same as above with *pixels* being an `int` array.

`FloatProcessor(int` *width*`, int` *height*`, float[]` *pixels*`, ColorModel` *cm*`)`
Constructor method: creates a new `FloatProcessor` object of the specified size with the pixel values taken from the one-dimensional `float` array *pixels*. The resulting image uses the color model *cm* (of type `java.awt.image.ColorModel`), or the default grayscale model if *cm* is `null`.

C.4.6 `ShortProcessor` (Class)

`ShortProcessor(int` *width*`, int` *height*`)`
Constructor method: creates a new `ShortProcessor` object of the specified size. The resulting image uses the default grayscale color model, which maps zero to black.

`ShortProcessor(int` *width*`, int` *height*`, short[]` *pixels*`, ColorModel` *cm*`)`
Constructor method: creates a new `ShortProcessor` object of the specified size with the pixel values taken from the one-dimensional `short` array *pixels*. The resulting image uses the color model *cm* (of type `java.awt.image.ColorModel`), or the default grayscale model if *cm* is `null`.

C.5 Loading and Storing Images

C.5.1 `IJ` (Class)

The class `IJ` provides the static method `void run()` for executing commands that apply to the currently active image. I/O commands include:

`IJ.run("Open...")`
Displays a file open dialog and then opens the image file selected by the user. Displays an error message if the selected file is not in one of the supported formats or if it is not found. The opened image becomes the active image.

`IJ.run("Revert")`
Reverts the active image to the original file version.

`IJ.run("Save")`
Saves the currently active image.

Class `IJ` also defines the following static methods for image I/O:

`static void open()`
Displays a file open dialog and then opens the image file (TIFF, DICOM, FITS, PGM, JPEG, BMP, GIF, LUT, ROI, or text format) selected by the user. Displays an error message if the selected file is not in one of the supported formats or if it is not found. No

reference to the opened image is returned (use `IJ.getimage()` to obtain the active image).

`static void open (String path)`
Opens and displays an image file specified by *path*; otherwise the same as `open()` above. Displays an error message if the specified file is not in one of the supported formats or if it is not found.

`static ImagePlus openImage (String path)`
Tries to open the image file specified by *path* and returns a ClassImagePlus object (which is not automatically displayed) if successful. Otherwise `null` is returned and no error is raised.

`static void save (String path)`
Saves the currently active image, lookup table, selection, or text window to the specified file *path*, whose extension encodes the file type. *path* must therefore end in `".tif"`, `".jpg"`, `".gif"`, `".zip"`, `".raw"`, `".avi"`, `".bmp"`, `".lut"`, `".roi"`, or `".txt"`.

`static void saveAs (String format, String path)`
Saves the currently active image, lookup table, selection (region of interest), measurement results, XY coordinates, or text window to the specified file *path*. The *format* argument must be `"tif"`, `"jpeg"`, `"gif"`, `"zip"`, `"raw"`, `"avi"`, `"bmp"`, `"text image"`, `"lut"`, `"selection"`, `"measurements"`, `"xy"`, or `"text"`

C.5.2 `Opener` (Class)

`Opener` is used to open TIFF (and TIFF stacks), DICOM, FITS, PGM, JPEG, BMP, or GIF images, and lookup tables, using a file open dialog or a path.

`Opener ()`
Constructor method: creates a new `Opener` object.

`void open ()`
Displays a file open dialog box and then opens the file selected by the user. Displays an error message if the selected file is not in one of the supported formats. No reference to the opened image is returned (use `IJ.getimage()` to obtain the active image).

`void open (String path)`
Opens and displays a TIFF, DICOM, FITS, PGM, JPEG, BMP, GIF, LUT, ROI, or text file. Displays an error message if the file specified by *path* is not in one of the supported formats. No reference to the opened image is returned (use `IJ.getimage()` to obtain the active image).

`ImagePlus openImage (String path)`
Attempts to open the specified file as a TIF, BMP, DICOM, FITS, PGM, GIF or JPEG image. Returns a new `ImagePlus` object if successful, otherwise `null`. Activates the plugin `HandleExtraFileTypes` if the file type is not recognized.

`ImagePlus openImage (String` *`directory`*`, String` *`name`*`)`
Same as above, with *`path`* split into *`directory`* and *`name`*.

`void openMultiple ()`
Displays a standard file chooser and then opens the files selected by the user. Displays error messages if one or more of the selected files is not in one of the supported formats. No reference to the opened images is returned (methods in class `WindowManager` can be used to access these images; see Sec. C.20.1).

`ImagePlus openTiff (String` *`directory`*`, String` *`name`*`)`
Attempts to open the specified file as a TIFF image or image stack. Returns an `ImagePlus` object if successful, `null` otherwise.

`ImagePlus openURL (String` *`url`*`)`
Attempts to open the specified URL as a TIFF, ZIP-compressed TIFF, DICOM, GIF, or JPEG image. Returns an `ImagePlus` object if successful, `null` otherwise.

`void setSilentMode (boolean` *`mode`*`)`
Turns silent mode on or off. The "Opening: path" status message is not displayed in silent mode.

C.5.3 `FileSaver` (Class)

Saves images in TIFF, GIF, JPEG, RAW, ZIP, and text formats.

`FileSaver (ImagePlus` *`im`*`)`
Constructor method: creates a new `FileSaver` for a given `ImagePlus` object.

`boolean save ()`
Tries to save the image associated with this `FileSaver` as a TIFF file. Returns `true` if successful or `false` if the user cancels the file save dialog.

`boolean saveAsBmp ()`
Saves the image associated with this `FileSaver` in BMP format using a save file dialog.

`boolean saveAsBmp (String` *`path`*`)`
Saves the image associated with this `FileSaver` in BMP format at the specified path.

`boolean saveAsGif ()`
Saves the image associated with this `FileSaver` in GIF format using a save file dialog.

`boolean saveAsGif (String` *`path`*`)`
Saves the image associated with this `FileSaver` in GIF format at the specified path.

`boolean saveAsJpeg ()`
Saves the image associated with this `FileSaver` in JPEG format using a save file dialog.

`boolean saveAsJpeg (String path)`
Saves the image associated with this `FileSaver` in JPEG format at the specified path.

`boolean saveAsLut ()`
Saves the lookup table (LUT) of the image associated with this `FileSaver` using a save file dialog.

`boolean saveAsLut (String path)`
Saves the lookup table (LUT) of the image associated with this `FileSaver` at the specified path.

`boolean saveAsPng ()`
Saves the image associated with this `FileSaver` in PNG format using a save file dialog.

`boolean saveAsPng (String path)`
Saves the image associated with this `FileSaver` in PNG format at the specified path.

`boolean saveAsRaw ()`
Saves the image associated with this `FileSaver` in raw format using a save file dialog.

`boolean saveAsRaw (String path)`
Saves the image associated with this `FileSaver` in raw format at the specified path.

`boolean saveAsRawStack (String path)`
Saves the stack associated with this `FileSaver` in raw format at the specified path.

`boolean saveAsRaw ()`
Saves the image associated with this `FileSaver` in raw format using a save file dialog.

`boolean saveAsRaw (String path)`
Saves the image associated with this `FileSaver` in raw format at the specified path.

`boolean saveAsText ()`
Saves the image associated with this `FileSaver` as tab-delimited text using a save file dialog.

`boolean saveAsText (String path)`
Saves the image associated with this `FileSaver` as tab-delimited text at the specified path.

`boolean saveAsTiff ()`
Saves the image associated with this `FileSaver` in TIFF format using a save file dialog.

`boolean saveAsTiff (String path)`
Saves the image associated with this `FileSaver` in TIFF format at the specified path.

`boolean saveAsTiffStack (String path)`
Saves the stack associated with this `FileSaver` as a multiimage TIFF at the specified path.

`boolean saveAsZip ()`
Saves the image associated with this `FileSaver` as a TIFF in a ZIP archive using a save file dialog.

`boolean saveAsZip (String path)`
Saves the image associated with this `FileSaver` as a TIFF in a ZIP archive at the specified path.

C.5.4 `FileOpener` (Class)

`FileOpener (FileInfo fi)`
Constructor method: creates a new `FileOpener` from a given `FileInfo` object *fi*. Use *im*`.getFileInfo()` or *im*`.getOriginalFileInfo()` to retrieve the `FileInfo` from a given `ImagePlus` object *im*.

`void open ()`
Opens the image from the location specified by this `FileOpener` and displays it. No reference to the opened image is returned (use `IJ.getimage()` to obtain the active image).

`ImagePlus open (boolean show)`
Opens the image from the location specified by this `FileOpener`. The image is displayed if *show* is `true`. Returns an `ImagePlus` object if successful, otherwise `null`.

`void revertToSaved (ImagePlus im)`
Restores *im* to its original disk or network version.

Here is a simple example that uses the classes `Opener`, `FileInfo`, and `FileOpener` for opening and subsequently reverting an image to its original:

```
Opener op = new Opener();
op.open();
ImagePlus im = IJ.getImage();
ImageProcessor ip = im.getProcessor();
ip.invert();
im.updateAndDraw();
// .... more modifications
// revert to original:
FileInfo fi = im.getOriginalFileInfo();
FileOpener fo = new FileOpener(fi);
fo.revertToSaved(im);
```

C.6 Image Parameters

C.6.1 ImageProcessor (Class)

`int getHeight ()`
Returns this image processor's height (number of lines).

`int getWidth ()`
Returns this image processor's width (number of columns).

`boolean getInterpolate ()`
Returns `true` if bilinear interpolation is turned on for this processor.

`void setInterpolate (boolean interpolate)`
Turns pixel interpolation for this processor on or off. If turned *on*, the processor uses bilinear interpolation for `getLine()` and geometric operations such as `scale()`, `resize()`, and `rotate()`.

C.6.2 ColorProcessor (Class)

`static double[] getWeightingFactors ()`
Returns the weights used for the red, green, and blue component (as a 3-element `double`-array) for converting RGB colors to grayscale or intensity (see Sec. 12.2.1). These weights are used, for example, by the methods `getPixelValue()`, `getHistogram()`, and `convertToByte()` to perform color conversions. The weights can be set with the static method `setWeightingFactors()` described below.

`static void setWeightingFactors (double wr, double wg, double wb)`
Sets the weights used for the red, green, and blue components for color-to-gray conversion (see Sec. 12.2.1). The default weights in ImageJ are $w_R = w_G = w_B = \frac{1}{3}$. Alternatively, if the "Weighted RGB Conversions" option is selected in the Edit→Options→Conversions dialog, the standard ITU-BT.709 [55] weights ($w_R = 0.299$, $w_G = 0.587$, $w_B = 0.114$) are used.

C.7 Accessing Pixels

The ImageJ class `ImageProcessor` provides a large variety of methods for accessing image pixels. All methods described in this section are defined for objects of class `ImageProcessor`.

C.7.1 Accessing Pixels by 2D Image Coordinates

Methods performing coordinate checking

The following methods are tolerant against passing out-of-bounds coordinate values. Reading pixel values from positions outside the image

canvas usually returns a zero value, while writing to such positions has no effect.

`int getPixel (int` *u*`, int` *v*`)`
: Returns the pixel value at the image coordinate (*u*, *v*). Zero is returned for all positions outside the image boundaries (no error). Applied to a `ByteProcessor` or `ShortProcessor`, the returned `int` value is identical to the numerical pixel value. For images of type `ColorProcessor`, the αRGB bytes are arranged inside the `int` value in the standard way (see Fig. 12.6). For a `FloatProcessor`, the returned 32-bit `int` value contains the *bit-pattern* of the corresponding `float` pixel value, and *not* a converted numerical value! A bit pattern *p* may be converted to a numeric `float`-value using the method `Float.intBitsToFloat(`*p*`)` (in package `java.lang.Number`).

`void putPixel (int` *u*`, `*v*`, int` *value*`)`
: Sets the pixel at image coordinate (*u*, *v*) to *value*. Coordinates outside the image boundaries are ignored (no error). For images of types `ByteProcessor` and `ShortProcessor`, *value* is clamped to the admissible range. For a `ColorProcessor`, the 8-bit αRGB values are packed inside *value* in the standard arrangement. For a `FloatProcessor`, *value* is assumed to contain the 32-bit pattern of a `float` value, which can be obtained using the `Float.floatToIntBits()` method.

`int[] getPixel (int` *u*`, int` *v*`, int[]` *iArray*`)`
: Returns the pixel value at the image coordinate (*u*, *v*) as an `int` array containing *one* element or, for a `ColorProcessor`, *three* elements (RGB component values with *iArray*`[0]`$= R$, *iArray*`[1]`$= G$, *iArray*`[2]`$= B$). If the argument passed to *iArray* is a suitable array (i. e., of proper size and not `null`), that array is filled with the pixel value(s) and returned; otherwise a new array is returned.

`void putPixel (int` *u*`, int` *v*`, int[]` *iArray*`)`
: Sets the pixel at position (*u*, *v*) to the value specified by the contents of *iArray*, which contains either *one* element or, for a `ColorProcessor`, *three* elements (RGB component values, with *iArray*`[0]`$= R$, *iArray*`[1]`$= G$, *iArray*`[2]`$= B$).

`float getPixelValue (int` *u*`, int` *v*`)`
: Returns the pixel value at the image coordinate (*u*, *v*) as a `float` value. For images of types `ByteProcessor` and `ShortProcessor`, a calibrated value is returned that is determined by the processor's (optional) calibration table. Invoked on a `FloatProcessor`, the method returns the actual (numeric) pixel value. In the case of a `ColorProcessor`, the gray value of the corresponding RGB pixel is returned (computed as a weighted sum of the RGB compo-

nents). The RGB component weights can be set using the method `setWeightingFactors()` (see p. 485).

`void putPixelValue(int u, int v, double value)`
Sets the pixel at position (*u*, *v*) to *value* (after clamping to the appropriate range and rounding). On a `ColorProcessor`, *value* is clamped to [0 . . . 255] and assigned to all three color components, thus creating a gray color with the luminance equivalent to *value*.

Methods without *coordinate checking*

The following methods are faster at the cost of not checking the validity of the supplied coordinates, i. e., passing out-of-bounds coordinate values will result in a runtime exception.

`int get(int u, v)`
This is a faster version of `getPixel()` that does not do bounds checking on the coordinates.

`void set(int u, int v, int value)`
This is a faster version of `putPixel()` that does not clamp out-of-range values and does not do bounds checking on the coordinates.

`float getf(int u, v)`
Returns the pixel value as a `float`; otherwise the same as `get()`.

`void setf(int u, int v, float value)`
Sets the pixel at (*u*, *v*) to the `float` value *value*; otherwise the same as `set()`.

C.7.2 Accessing Pixels by 1D Indices

These methods are useful for processing images if the individual pixel coordinates are not relevant, e. g., for performing point operations on all image pixels.

`int get(int i)`
Returns the content of the `ImageProcessor`'s pixel array at position *i* as an `int` value, with $0 \leq i < w \cdot h$ (w, h are the width and height of the image, respectively). The method `getPixelCount()` (see below) retrieves the size of the pixel array.

`void set(int i, int value)`
Inserts *value* at the image's pixel array at position *i*.

`float getf(int i)`
Returns the content of the image processor's pixel array at position *i* as a `float` value.

`void setf(int i, float value)`
Inserts *value* at the image's pixel array at position *i*.

`int getPixelCount()`
Returns the number of pixels in this image, i. e., the length of the pixel array.

Pixel access is faster with the above methods because the supplied index *i* directly addresses the one-dimensional pixel array (see p. 490 for a directly accessing the pixel array without using method calls, which is still faster). Note that these methods are efficient at the cost of not checking the validity of their arguments, i. e., passing an illegal index will result in a runtime exception. The typical use of these methods is demonstrated by the following example:

```
...
int M = ip.getPixelCount();
for (int i = 0; i < M; i++) {
  int a = ip.get(i);
  int b = ... ; // compute the new pixel value
  ip.set(i, b);
}
...
```

Note that we explicitly define the range variable `M` instead of writing

```
for (int i = 0; i < ip.getPixelCount(); i++) ...
```

directly, because in this case the method `getPixelCount()` would be (unnecessarily) invoked in every single iteration of the `for`-loop.

C.7.3 Accessing Multiple Pixels

`Object getPixels()`
: Returns a reference (not a copy!) to the processor's one-dimensional pixel array, and thus any changes to the returned pixel array immediately affect the contents of the corresponding image. The array's element type depends on the type of processor:

  `ByteProcessor` → `byte[]`
  `ShortProcessor` → `short[]`
  `FloatProcessor` → `float[]`
  `ColorProcessor` → `int[]`

  Since the type (`Object`) of the returned object is generic, type casting is required to actually use the returned array; e. g.,

  ```
  ByteProcessor ip = new ByteProcessor(200, 300);
  byte[] pixels = (byte[]) ip.getPixels();
  ```

  Note that this typecast is potentially dangerous. To avoid a runtime exception one should assure that processor and array types match; e. g., using Java's `instanceof` operator:

  ```
  if (ip instanceof ByteProcessor) ...   or
  if (ip.getPixels() instanceof byte[]) ...
  ```

`void setPixels(Object pixels)`
: Replaces the processor's pixel array by *pixels*. The type and size of this one-dimensional array must match the specifications of the target processor (see `getpixels()`). The processor's snapshot array is reset.

`Object getPixelsCopy()`
Returns a reference to the image's snapshot (undo) array. If the snapshot array is `null`, a *copy* of the processor's pixel data is returned. Otherwise the use of the result is the same as with `getPixels()`.

`void getColumn(int u, int v, int[] data, int n)`
Returns *n* contiguous pixel values along the vertical column *u*, starting at position (*u*, *v*). The result is stored in the array *data* (which must not be `null` and at least of size *n*).

`void putColumn(int u, int v, int[] data, int n)`
Inserts the first *n* pixels contained in *data* into the vertical column *u*, starting at position (*u*, *v*). *data* must not be `null` and at least of size *n*.

`void getRow(int u, int v, int[] data, int m)`
Returns *m* contiguous pixel values along the horizontal line *v*, starting at position (*u*, *v*). The result is stored in the array *data* (which must not be `null` and at least of size *m*).

`void putRow(int u, int v, int[] data, int m)`
Inserts the first *m* pixels contained in *data* into the horizontal row *v*, starting at position (*u*, *v*). *data* must not be `null` and at least of size *m*.

`double[] getLine(double x1, double y1, double x2, double y2)`
Returns a one-dimensional array containing the pixel values along the straight line starting at position (*x1*, *y1*) and ending at (*x2*, *y2*). The length of the returned array corresponds to the rounded integer distance between the start and endpoint, any of which may be outside the image bounds. Interpolated pixel values are used if the processor's interpolation setting is on (see `setInterpolate()`).

`void insert(ImageProcessor ip, int u, int v)`
Inserts (pastes) the image contained in *ip* into this image at position (*u*, *v*).

C.7.4 Accessing All Pixels at Once

`int[][] getIntArray()`
Returns the contents of the image as a new two-dimensional `int` array, by storing the pixels as *array*`[u][v]` (i. e., in column-first order).

`void setIntArray(int[][] pixels)`
Replaces the image pixels with the contents of the two-dimensional `int` array *pixels*, which must be of exactly the same size as the target image. Pixels are assumed to be arranged in column-first order (as above).

`float[][] getFloatArray()`
: Returns the contents of the image as a new two-dimensional `int` array, by storing the pixels as *`array`*`[u][v]` (i. e., in column-first order).

`void setFloatArray(float[][] pixels)`
: Replaces the image pixels with the contents of the two-dimensional `float` array *`pixels`*, which must be of exactly the same size as the target image. Pixels are assumed to be arranged in column-first order (as above).

C.7.5 Specific Access Methods for Color Images

The following methods are only defined for objects of type `ColorProcessor`.

`void getRGB(byte[] R, byte[] G, byte[] B)`
: Stores the red, green, and blue color planes into three separate `byte` arrays *`R`*, *`G`*, *`B`*, whose size must be at least equal to the number of pixels in this image.

`void setRGB(byte[] R, byte[] G, byte[] B)`
: Fills the pixel array of this color image from the contents of the `byte` arrays *`R`*, *`G`*, *`B`*, whose size must be at least equal to the number of pixels in this image.

`void getHSB(byte[] H, byte[] S, byte[] B)`
: Stores the *hue*, *saturation*, and *brightness* values into three separate `byte` arrays *`H`*, *`S`*, *`B`*, whose size must be at least equal to the number of pixels in this image.

`void setHSB(byte[] H, byte[] S, byte[] B)`
: Fills the pixel array of this color image from the contents of the `byte` arrays *`H`* (*hue*), *`S`* (*saturation*), and *`B`* (*brightness*), whose size must be at least equal to the number of pixels in this image.

`FloatProcessor getBrightness()`
: Returns the *brightness* values (as defined by the HSV color model) of this color image as a new `FloatProcessor` of the same size.

`void setBrightness(FloatProcessor fp)`
: Replaces the *brightness* values (as defined by the HSV color model) of this color image by the values of the corresponding pixels in the specified `FloatProcessor` *`fp`*, which must be of the same size as this image.

C.7.6 Direct Access to Pixel Arrays

The use of pixel access methods (such as `getPixel()` and `putPixel`) is relatively time-consuming because these methods perform careful bounds checking on the given pixel coordinates. The alternative methods `set()` and `get()` do *no* bounds checking and are thus somewhat faster but still carry the overhead of method invocation.

Program C.1
Direct pixel access for images of type `ByteProcessor`. Notice the use of the `instanceof` operator (lines 3–4) to verify the correct type of the processor. In this case (since the pixel coordinates (`u`, `v`) are not used in the computation), a single loop over the one-dimensional pixel array could be used instead.

```
1  public void run (ImageProcessor ip) {
2    // check if ip is really a valid ByteProcessor:
3    if (!(ip instanceof ByteProcessor)) return;
4    if (!(ip.getPixels() instanceof byte[])) return;
5    // get pixel array:
6    byte[] pixels = (byte[]) ip.getPixels();
7    int w = ip.getWidth();
8    int h = ip.getHeight();
9    // process pixels:
10   for (int v = 0; v < h; v++) {
11     for (int u = 0; u < w; u++) {
12       int p = 0xFF & pixels[v * w + u];
13       p = p + 1;
14       pixels[v * w + u] = (byte) (0xFF & p);
15     }
16   }
17 }
```

If many pixels must be processed, *direct* access to the elements of the processor's pixel array may be considerably more efficient. For this we have to consider that the pixel arrays of Java and ImageJ images are one-dimensional and arranged in row-first order (also see Sec. B.2.3).

A reference to a processor's one-dimensional pixel array *pixels* is obtained by the method `getPixels()`. To retrieve a particular pixel at position (u, v) we must first compute its one-dimensional index i, where the width w (i. e., the length of each line) of the image must be known:

$$I(u, v) \equiv pixels[i] = pixels[v \cdot w + u] .$$

Program C.1 shows an example for direct pixel access inside the `run` method of a ImageJ plugin for an image of type `ByteProcessor`. The bit operations `0xFF & pixels[]` and `0xFF & p` (lines 12 and 14, respectively) are needed to use unsigned `byte` data in the range $[0 \ldots 255]$ (as described in Sec. B.1.3). Analogously, the bit-mask `0xFFFF` and a typecast to `(short)` would be required when processing unsigned 16-bit images of type `ShortProcessor`.

If, as in Prog. C.1, the pixels' coordinates (u, v) are not used in the computation and the order of pixels being accessed is irrelevant, a single loop can be used to iterate over all elements of the one-dimensional pixel array (of length $w \cdot h$). This approach is used, for example, to process all pixels of a color image in Prog. 12.1 (p. 246). Also, one should consider the 1D access methods in Sec. C.7.2 as a simple and similarly efficient alternative to the direct access scheme described above.

C.8 Converting Images

C.8.1 ImageProcessor (Class)

The class `ImageProcessor` implements the following basic methods for converting between different types of images. Each method returns a new `ImageProcessor` object unless the original image is of the desired type already. If this is the case, only a reference to the source image is returned, i. e., *no* duplication occurs.

`ImageProcessor convertToByte (boolean doScaling)`
: Copies the contents of the source image to a new object of type `ByteProcessor`. If *`doScaling`* is `true`, the pixel values are automatically scaled to the range of the target image; otherwise the values are clamped without scaling. If applied to an image of type `colorProcessor`, the intensity values are computed as the weighted sum of the RGB component values. The RGB weights can be set using the method `setWeightingFactors()` (see p. 485).

`ImageProcessor convertToShort (boolean doScaling)`
: Copies the contents of the source image to a new object of type `ShortProcessor`. If *`doScaling`* is `true`, the pixel values are automatically scaled to the range of the target image; otherwise the values are clamped without scaling.

`ImageProcessor convertToFloat ()`
: Copies the contents of the source image to a new object of type `FloatProcessor`.

`ImageProcessor convertToRGB ()`
: Copies the contents of the source image to a new object of type `ColorProcessor`.

C.8.2 ImagePlus, ImageConverter (Classes)

Images of type `ImagePlus` can be converted by instances of the class `ImageConverter` (package `ij.process`). To convert a given `ImagePlus` object `imp`, we first create an instance of the class `ImageConverter` for that image and then invoke a conversion method; for example,

```
ImageConverter iConv = new ImageConverter(imp);
iConv.convertToGray8();
```

This *destructively* modifies the image `imp` to an 8-bit grayscale image by replacing the attached `ImageProcessor` (among other things). No conversion takes place if the original image is of the target type already. The complete ImageJ plugin in Prog. C.2 illustrates how `ImageConverter` could be used to convert any image to an 8-bit grayscale image before processing.

In summary, the following methods are applicable to `ImageConverter` objects:

Program C.2
ImageJ sample plugin for converting any type of `ImagePlus` image to 8-bit grayscale. The actual conversion takes place on line 22. The updated image processor is retrieved (line 23) and can subsequently be used to process the converted image. Notice that the original `ImagePlus` object `imp` is not passed to the plugin's `run()` method but only to the `setup()` method, which is called first (by ImageJ's plugin mechanism) and keeps a reference in the instance variable `imp` (line 16) for later use.

```
1  import ij.ImagePlus;
2  import ij.plugin.filter.PlugInFilter;
3  import ij.process.ImageConverter;
4  import ij.process.ImageProcessor;
5
6  public class Convert_ImagePlus_To_Gray8
7                   implements PlugInFilter {
8
9    ImagePlus imp = null;
10
11   public int setup(String arg, ImagePlus imp) {
12     if (imp == null) {
13       IJ.noImage();
14       return DONE;
15     }
16     this.imp = imp;
17     return DOES_ALL;  // this plugin accepts any type of image
18   }
19
20   public void run(ImageProcessor ip) {
21     ImageConverter iConv = new ImageConverter(imp);
22     iConv.convertToGray8();
23     ip = imp.getProcessor(); // ip is now of type ByteProcessor
24     // process grayscale image ...
25   }
26
27 } // end of class Convert_ImagePlus_To_Gray8
```

`void convertToGray8 ()`
: Converts the source image to an 8-bit (`byte`) grayscale image.

`void convertToGray16 ()`
: Converts the source image to a 16-bit (`short`) grayscale image.

`void convertToGray32 ()`
: Converts the source image to a 32-bit (`float`) grayscale image.

`void convertToRGB ()`
: Converts the source image to a 32-bit (`int`) RGB color image.

`void convertToHSB ()`
: Converts a given RGB image to an HSB[9] (hue, saturation, brightness) stack, a stack of three independent grayscale images. May not be applied to another type of image.

`void convertHSBToRGB ()`
: Converts an HSB image stack to a single RGB image.

`void convertRGBStackToRGB ()`
: Converts an RGB image stack to a single RGB image.

[9] HSB is identical to the HSV color space (see Sec. 12.2.3).

`void convertToRGBStack ()`
Converts an RGB image to a three-slice RGB image stack.

`void convertRGBtoIndexedColor (int` *`nColors`*`)`
Converts an RGB image to an indexed color image with *`nColors`* colors.

`void setDoScaling (boolean` *`doScaling`*`)`
Enables or disables the scaling of pixel values. If *`doScaling`* is `true`, pixel values are scaled to [0 . . . 255] when converted to 8-bit images and to [0 . . . 65,535] for 16-bit images. Otherwise no scaling is applied.

`void getDoScaling ()`
Returns `true` if scaling is enabled for that `ImageConverter` object.

C.9 Histograms and Image Statistics

C.9.1 `ImageProcessor` (Class)

`int[] getHistogram ()`
Returns the histogram of the image or the region of interest (ROI), if selected. For images of type `ColorProcessor`, the intensity histogram is returned, where intensities are computed as weighted sums of the RGB components. The RGB weights can be set using the method `setWeightingFactors()` (see p. 485).

`double getHistogramMax ()`
Returns the maximum pixel value used for computing histograms of float images.

`double getHistogramMin ()`
Returns the minimum pixel value used for computing histograms of float images.

`int getHistogramSize ()`
Returns the number of bins used for computing histograms of float images.

`void setHistogramRange (double` *`histMin`*`, double` *`histMax`*`)`
Specifies the range of pixel values used for computing histograms of float images.

`int setHistogramSize (int` *`size`*`)`
Specifies the number of bins used for computing histograms of float images.

Additional statistics can be obtained through the class `ImageStatistics` and its subclasses `ByteStatistics`, `ShortStatistics`, `FloatStatistics`, `ColorStatistics`, and `StackStatistics`.

C.10 Point Operations

C.10.1 `ImageProcessor` (Class)

Single-image operations

The following methods for objects of type `ImageProcessor` perform arithmetic or logic operations with a constant scalar value as the second operand. All operations are applied either to the whole image or to the pixels within the region of interest, if selected.

`void abs ()`
Replaces every pixel by its absolute value.

`void add (int value)`
Increments every pixel by *value*.

`void add (double value)`
Increments every pixel by *value*.

`void and (int value)`
Bitwise AND operation between the pixel and *value*.

`void applyTable (int[] lut)`
Applies the mapping specified by the lookup table *lut* to each pixel.

`void autoThreshold ()`
Converts the image to binary using a threshold determined automatically from the original histogram.

`void gamma (double g)`
Applies a gamma correction with the gamma value *g*.

`void log ()`
Replaces every pixel a by $\log_{10}(a)$.

`void max (double value)`
Maximum operation: pixel values greater than *value* are set to *value*.

`void min (double value)`
Minimum operation: pixel values smaller than *value* are set to *value*.

`void multiply (double value)`
All pixels are multiplied by *value*.

`void noise (double r)`
Increments every pixel by a random value with normal distribution in the range $\pm r$.

`void or (int value)`
Bitwise OR operation between the pixel and *value*.

`void sqr ()`
Replaces every pixel a by a^2.

`void sqrt ()`
Replaces every pixel a by $\sqrt{a}$.

`void threshold (int th)`
Threshold operation: sets every pixel a with $a \leq th$ to 0 and all other pixels to 255.

`void xor (int value)`
Bitwise exclusive-OR (XOR) operation between the pixel and *value*.

Multi-image operations

The class `ImageProcessor` defines a single method for combining two images:

`void copyBits (ImageProcessor B, int u, int v, int mode)`
Copies the image *B* into the target image at position (*u*, *v*) using the transfer mode *mode*. The target image is destructively modified, and *B* remains unchanged.

Admissible *mode* values are defined as constants by the `Blitter` interface (see below); for example,

```
ipA.copyBits(ipB, 0, 0, Blitter.COPY);
```

for copying (pasting) the contents of image `ipB` into `ipA`. Another example for the use of `copyBits()` can be found in Sec. 5.8.3 (page 81).

In summary, `ij.process.Blitter` defines the following *mode* values for the `copyBits()` method (A refers to the target image, B to the source image):

`ADD`
$A(u,v) \leftarrow A(u,v) + B(u,v)$

`AND`
$A(u,v) \leftarrow A(u,v) \wedge B(u,v)$
Bitwise AND operation.

`AVERAGE`
$A(u,v) \leftarrow (A(u,v) + B(u,v))/2$

`COPY`
$A(u,v) \leftarrow B(u,v)$

`COPY_INVERTED`
$A(u,v) \leftarrow 255 - B(u,v)$
Only applicable to 8-bit grayscale and RGB images.

`DIFFERENCE`
$A(u,v) \leftarrow |A(u,v) - B(u,v)|$

`DIVIDE`
$A(u,v) \leftarrow A(u,v)/B(u,v)$

`MAX`
$A(u,v) \leftarrow \max(A(u,v), B(u,v))$

`MIN`
$A(u,v) \leftarrow \min(A(u,v), B(u,v))$

`MULTIPLY`
$A(u,v) \leftarrow A(u,v) \cdot B(u,v)$

`OR`
$A(u,v) \leftarrow A(u,v) \vee B(u,v)$
Bitwise OR operation.

`SUBTRACT`
$A(u,v) \leftarrow A(u,v) - B(u,v)$

`XOR`
$A(u,v) \leftarrow A(u,v)$ xor $B(u,v)$
Bitwise exclusive OR (XOR) operation.

C.11 Filters

C.11.1 `ImageProcessor` (Class)

`void convolve (float[] kernel, int w, int h)`
Performs a linear convolution of the image with the filter matrix *kernel* (of size $w \times h$), specified as a one-dimensional `float` array.

`void convolve3x3 (int[] kernel)`
Performs a linear convolution of the image with the filter matrix *kernel* (of size 3×3), specified as a one-dimensional `int` array.

`void dilate ()`
Dilation using a 3×3 minimum filter.

`void erode ()`
Erosion using a 3×3 maximum filter.

`void findEdges ()`
Applies a 3×3 edge filter (Sobel operator).

`void medianFilter ()`
Applies a 3×3 median filter.

`void smooth ()`
Applies a simple 3×3 average filter (box filter).

`void sharpen ()`
Sharpens the image using a 3×3 Laplacian-like filter kernel.

C.12 Geometric Operations

C.12.1 `ImageProcessor` (Class)

`ImageProcessor crop ()`
Creates a new `ImageProcessor` object with the contents of the current region of interest.

`void flipHorizontal ()`
Destructively mirrors the contents of the image (or region of interest) horizontally.

`void flipVertical ()`
Destructively mirrors the contents of the image (or region of interest) vertically.

`ImageProcessor resize (int` *`width`*`, int` *`height`*`)`
Creates a new `ImageProcessor` object containing a scaled copy of this image (or region of interest) of size *`width`* × *`height`*.

`void rotate (double` *`angle`*`)`
Destructively rotates the image (or region of interest) *`angle`* degrees clockwise.

`ImageProcessor rotateLeft ()`
Rotates the entire image 90° counterclockwise and returns a new `ImageProcessor` object that contains the rotated image.

`ImageProcessor rotateRight ()`
Rotates the entire image 90° clockwise and returns a new `ImageProcessor` object that contains the rotated image.

`void scale (double` *`xScale`*`, double` *`yScale`*`)`
Destructively scales (zooms) the image (or region of interest) in x and y by the factors *`xScale`* and *`yScale`*, respectively. The size of the image does not change.

`void setBackgroundValue (double` *`value`*`)`
Sets the background fill value used by the `rotate()` and `scale()` methods.

`boolean getInterpolate ()`
Returns `true` if (bilinear) interpolation is turned on for this image processor.

`void setInterpolate (boolean` *`interpolate`*`)`
Activates bilinear interpolation for geometric operations (otherwise nearest-neighbor interpolation is used).

`double getInterpolatedPixel (double` *`x`*`, double` *`y`*`)`
Returns the interpolated pixel value for the continuous coordinates (*`x`*, *`y`*) using bilinear interpolation. In case of a `ColorProcessor`, the gray value resulting from nearest-neighbor interpolation is returned (use `getInterpolatedRGBPixel()` to obtain interpolated color values).

`double getInterpolatedRGBPixel (double` *`x`*`, double` *`y`*`)`
Returns the interpolated RGB pixel value for the continuous coordinates (*`x`*, *`y`*) using bilinear interpolation. This method is defined for `ColorProcessor` only.

C.13 Graphic Operations

C.13.1 ImageProcessor (Class)

`void drawDot (int u, int v)`
Draws a dot centered at position (*u*, *v*) using the current line width and fill/draw value.

`void drawLine (int u1, int v1, int u2, int v2)`
Draws a line from position (*u1*, *v1*) to position (*u2*, *v2*).

`void drawOval (int u, int v, int w, int h)`
Draws an axis-parallel ellipse with a bounding rectangle of size *w* × height *h*, positioned at (*u*, *v*). See also `fillOval()`.

`void drawPixel (int u, int v)`
Sets the pixel at position (*u*, *v*) to the current fill/draw value.

`void drawPolygon (java.awt.Polygon p)`
Draws the polygon *p* using the current fill/draw value. See also `fillPolygon()`.

`void drawRect (int u, int v, int width, int height)`
Draws a rectangle of size *width* × *height* and parallel to the coordinate axes.

`void drawString (String s)`
Draws the string *s* at the *current drawing position* (set with `moveTo()` or `lineTo()`) using the current fill/draw value and font. Use `setAntialiasedText()` to control anti-aliasing for text rendering.

`void drawString (String s, int u, int v)`
Draws the string *s* at position (*u*, *v*) using the current fill/draw value and font.

`void fill ()`
Fills the image or region of interest (if selected) with the current fill/draw value.

`void fill (ImageProcessor mask)`
Fills the pixels that are inside both the region of interest and the mask image *mask*, which must be of the same size as the image or region of interest. A position is considered *inside* the mask if the corresponding mask pixel has a nonzero value.

`void fillOval (int u, int v, int w, int h)`
Draws and fills an axis-parallel ellipse with a bounding rectangle of size *w* × height *h*, positioned at (*u*, *v*). See also `drawOval()`.

`void fillPolygon (java.awt.Polygon p)`
Draws and fills the polygon *p* using the current fill/draw value. See also `drawPolygon()`.

`void getStringWidth (String s)`
Returns the width (in pixels) of the string *s* using the current font.

`void insert (ImageProcessor src, int u, int v)`
Inserts the image contained in *src* at position (*u*, *v*).

`void lineTo (int u, int v)`
Draws a line from the *current drawing position* to (*u*, *v*). Updates the current drawing position to (*u*, *v*).

`void moveTo (int u, int v)`
Sets the *current drawing position* to (*u*, *v*).

`void setAntialiasedText (boolean antialiasedText)`
Specifies whether or not text is rendered using anti-aliasing.

`void setClipRect (Rectangle clipRect)`
Sets the clipping rectangle used by the methods `lineTo()`, `drawLine()`, `drawDot()`, and `drawPixel()`.

`void setColor (java.awt.Color color)`
Sets the default fill/draw value for subsequent drawing operations to the pixel value closest to the specified color.

`void setFont (java.awt.Font font)`
Sets the font to be used by `drawString()`.

`void setJustification (int justification)`
Sets the justification used by `drawString()`. Admissible values for *justification* are the constants `CENTER_JUSTIFY`, `RIGHT_JUSTIFY`, and `LEFT_JUSTIFY` (defined in class `ImageProcessor`).

`void setLineWidth (int width)`
Sets the line width used by `lineTo()` and `drawDot()`.

`void setValue (double value)`
Sets the fill/draw value for subsequent drawing operations. Notice that the `double` parameter *value* is interpreted differently depending on the type of image. For `ByteProcessor`, `ShortProcessor`, and `FloatProcessor`, the numerical value of *value* is simply converted by typecasting to the corresponding pixel type. For images of type `ColorProcessor`, *value* is first typecast to `int` and the result is interpreted as a packed αRGB value.

C.14 Displaying Images and Image Stacks

Only images of type `ImagePlus` (which include stacks of images) may be displayed on the screen using the methods below. In contrast, objects of type `ImageProcessor` are not visible themselves but can only be displayed through an associated `ImagePlus` object, as described in Sec. C.14.2.

C.14.1 `ImagePlus` (Class)

`void draw ()`
Draws the image and the outline of the region of interest (if se-

lected). Does nothing if there is no window associated with this image (i. e., `show()` has not been called).

`void draw (int` *u*`, int` *v*`, int` *width*`, int` *height*`)`
Draws the image and the outline of the region of interest (as above) using the clipping rectangle specified by the four parameters.

`int getCurrentSlice ()`
Returns the index of the currently displayed stack slice or 1 if this `ImagePlus` is a single image. Use `setSlice()` to display a particular slice.

`int getID ()`
Returns this image's unique ID number. This ID can be used with the `WindowManager`'s method `getImage()` to reference a particular image.

`String getShortTitle ()`
Returns a shortened version of the image's name.

`String getTitle ()`
Returns the image's full name.

`ImageWindow getWindow ()`
Returns the window (of type `ij.gui.ImageWindow`, a subclass of `java.awt.Frame`) that is being used to display this `ImagePlus` image.

`void hide ()`
Closes any window currently displaying this image.

`boolean isInvertedLut ()`
Returns `true` if this image's `ImageProcessor` uses an inverting lookuptable (LUT) for displaying zero pixel values as white and 255 as black. The LUT can be inverted by calling `invertLut()` on the corresponding `ImageProcessor` (which is obtained with `getProcessor()`).[10]

`void repaintWindow ()`
Calls `draw()` to draw the image and also repaints the image window to update the header information (dimension, type, size).

`void setSlice (int` *index*`)`
Displays the specified slice of a stack. The parameter *index* must be $1 \leq$ *index* $\leq N$, where N is the number of slices in the stack. Redisplays the (single) image if this `ImagePlus` does not contain a stack.

`void setTitle (String` *title*`)`
Sets the image name to *title*.

`void show ()`
Opens a window to display this image and clears the status bar in the main ImageJ window.

[10] The class `ImagePlus` also defines a method `invertLookupTable()`, but this method is not `public`.

`void show (String statusMsg)`
Opens a window to display this image and displays the text *statusMsg* in the status bar.

`void updateAndDraw ()`
Updates this image from the pixel data in its associated `ImageProcessor` object and then displays it (by calling `draw()`).

`void updateAndRepaintWindow ()`
Calls `updateAndDraw()` to repaint the current pixel data and also updates the header information (dimension, type, size).

C.14.2 `ImageProcessor` (Class)

As mentioned above, objects of type `ImageProcessor` are not visible automatically but require an associated`ImagePlus` object to be seen on the screen.

The `ImageProcessor` object passed to a typical ImageJ plugin (of class `PlugInFilter`) belongs to a visible image and thus already has an associated `ImagePlus`, which is passed to the `setup()` method of that plugin. This `ImagePlus` object can be used to redisplay the image at any time during plugin execution, as exemplified in Prog. C.3.

To display the contents of a new `ImageProcessor`, a corresponding `ImagePlus` object must first be created for it using the constructor methods described in Sec. C.3.1; e. g.,

```
ByteProcessor ip = new ByteProcessor(200,300);
ImagePlus im = new ImagePlus ("A New Image", ip);
im.show();              // show image on screen
ip.smooth();            // modify this image
im.updateAndDraw();     // redisplay modified image
```

Notice that there is no simple way to access the `ImagePlus` object associated with a given `ImageProcessor` or determine if one exists at all. In reverse, the image processor of a given `ImagePlus` can be obtained directly with the `getProcessor()` method (see Sec. C.21.1). Analogously, an image *stack* associated with a given `ImagePlus` object is retrieved by the method `getStack()` (see Sec. C.15.1).

The following methods for the class `ImageProcessor` control the mapping between the original pixel values and display intensities:

`ColorModel getColorModel ()`
Returns this image processor's color model (of type `java.awt.image.ColorModel`): `IndexColorModel` for grayscale and indexed color images, and `DirectColorModel` for RGB color images. For processors other than `ColorProcessor`, this is the base lookup table, not the one that may have been modified by `setMinAndMax()` or `setThreshold()`. An `ImageProcessor`'s color model can be changed with the method `setColorModel()`.

Program C.3
Animation example (redisplaying the image passed to an ImageJ plugin). The `ImagePlus` object associated with the `ImageProcessor` passed to the plugin is initially received by the `setup()` method, where a reference is stored in variable `im` (line 14). Inside the `run()` method, the `ImageProcessor` is repeatedly modified (lines 21–22) and subsequently redisplayed by invoking `updateAndDraw()` on the associated `ImagePlus` object `im` (line 24).

```
1  import ij.IJ;
2  import ij.ImagePlus;
3  import ij.plugin.filter.PlugInFilter;
4  import ij.process.ImageProcessor;
5
6  public class Display_Demo implements PlugInFilter {
7    ImagePlus im = null;
8
9    public int setup(String arg, ImagePlus im) {
10     if (im == null) {
11       IJ.noImage();
12       return DONE;
13     }
14     this.im = im;  // keep reference to associated ImagePlus
15     return DOES_ALL;
16   }
17
18   public void run(ImageProcessor ip) {
19     for (int i = 0; i < 10; i++) {
20       // modify this image:
21       ip.smooth();
22       ip.rotate(30);
23       // redisplay this image:
24       im.updateAndDraw();
25       // sleep 100 ms so user can watch:
26       IJ.wait(100);
27     }
28   }
29
30 } // end of class Display_Demo
```

`ColorModel getCurrentColorModel ()`
: Returns the current color model, which may have been modified by `setMinAndMax()` or `setThreshold()`.

`double getMax ()`
: Returns the largest displayed pixel value $a_{\max}$ (pixels $I(u,v) > a_{\max}$ are mapped to 255). $a_{\max}$ can be modified with the method `setMinMax()`.

`double getMin ()`
: Returns the smallest displayed pixel value $a_{\min}$ (pixels $I(u,v) < a_{\min}$ are mapped to 0). $a_{\min}$ can be modified with the method `setMinMax()`.

`void invertLut ()`
: Inverts the values in this `ImageProcessor`'s lookuptable for displaying zero pixel values as white and 255 as black. Does nothing if this is a `ColorProcessor`.

`boolean isInvertedLut ()`
Returns `true` if this `ImageProcessor` uses an inverting lookuptable for displaying zero pixel values as white and 255 as black.

`void resetMinAndMax ()`
For `ShortProcessor` and `FloatProcessor` images, the $a_{\min}$ and $a_{\max}$ values are recalculated to correctly display the image. For `ByteProcessor` and `ColorProcessor`, the lookuptables are reset to default values.

`void setMinAndMax (double amin, double amax)`
Sets the parameters $a_{\min}$ and $a_{\max}$ to the specified values. The image is displayed by mapping the pixel values in the range $[a_{\min} \ldots a_{\max}]$ to screen values in the range $[0 \ldots 255]$.

`void setColorModel (java.awt.image.ColorModel cm)`
Sets the color model. Except for `ColorProcessor`, *cm* must be of type `IndexColorModel`.

C.15 Operations on Image Stacks

C.15.1 ImagePlus (Class)

For creating ready-to-use multislice stack images, see the methods for class `NewImage` (Sec. C.3.4).

`ImageStack createEmptyStack ()`
Returns a new, *empty* stack with the same width, height, and color table as the given `ImagePlus` object to which this method is applied. Notice that the new stack is *not* automatically attached to this `ImagePlus` by this method (use `setStack()` for this purpose).

`ImageStack getImageStack ()`
Returns the image stack associated with the `ImagePlus` object to which this method is applied. Calls `getStack()` if the image has no stack yet.

`ImageStack getStack ()`
Returns the image stack associated with the `ImagePlus` object to which this method is applied. If no stack exists, a new single-slice stack is created with the contents of that image (by calling `createEmptyStack()`). After adding or removing slices to/from the returned `ImageStack` object, `setStack()` should be called to update the image and the window that is displaying it.

`int getStackSize ()`
If this `ImagePlus` contains a stack, the number of slices is returned; 1 is returned if this is a single image.

`void setStack (String title, ImageStack stack)`
Replaces the current stack of this `ImagePlus`, if any, with *stack* and assigns the name *title*.

C.15.2 ImageStack (Class)

For creating new `ImageStack` objects, see the constructor methods in Sec. C.3.2.

`void addSlice(String label, ImageProcessor ip)`
Adds the image specified by *ip* to the end of the stack, assigning the title *label* to the new slice. No pixel data are duplicated.

`void addSlice(String label, ImageProcessor ip, int n)`
Adds the image specified by *ip* to the stack following slice *n*, assigning the title *label* to the new slice. The slice is added to the beginning of the stack if *n* is zero. No pixel data are duplicated.

`void addSlice(String label, Object pixels)`
Adds the image specified by *pixels* (which must be a suitable pixel array) to the end of the stack.

`void deleteLastSlice()`
Deletes the last slice in the stack.

`void deleteSlice(int n)`
Deletes the *n*th slice from the stack, where $1 \leq n \leq$ `getsize()`.

`int getHeight()`
Returns the height of the images in this stack.

`Object[] getImageArray()`
Returns the whole stack as an array of one-dimensional pixel arrays. Note that the size of the returned array may be greater than the number of slices currently in the stack, with unused elements set to `null`. No pixel data are duplicated.

`Object getPixels(int n)`
Returns the one-dimensional pixel array for the *n*th slice of the stack, where $1 \leq n \leq$ `getsize()`. No pixel data are duplicated.

`ImageProcessor getProcessor(int n)`
Creates and returns an `ImageProcessor` for the *n*th slice of the stack, where $1 \leq n \leq$ `getsize()`. No pixel data are duplicated. The method returns `null` if the stack is empty.

`int getSize()`
Returns the number of slices in this stack.

`String getSliceLabel(int n)`
Returns the label of the *n*th slice, where $1 \leq n \leq$ `getsize()`. Returns `null` if the slice has no label.

`String[] getSliceLabels()`
Returns the labels of all slices as an array of strings. Note that the size of the returned array may be greater than the number of slices currently in the stack. Returns `null` if the stack is empty or the label of the first slice is `null`.

`String getShortSliceLabel(int n)`
Returns a shortened version (up to the first 60 characters or first

newline character and suffix removed) of the *n*th slice's label, where $1 \leq n \leq$ `getsize()`. Returns `null` if the slice has no label.

`int getWidth ()`
Returns the height of the images in this stack.

`void setPixels (Object pixels, int n)`
Assigns the pixel array *pixels* to the *n*th slice, where $1 \leq n \leq$ `getsize()`. No pixel data are duplicated.

`void setSliceLabel (String label, int n)`
Assigns the title *label* to the *n*th slice, where $1 \leq n \leq$ `getsize()`.

C.15.3 Stack Example

Programs C.4 and C.5 shows a working example for the use of image stacks that blends one image into another by a simple technique called "alpha blending" by producing a sequence of intermediate images stored in a stack. This is an extension of Progs. 5.5 and 5.6 (see p. 85), which produce only a single blended image.

The background image (`bgIp`) is the current image (i. e., the image to which the plugin is applied) that is passed to the plugin's `run()` method. The foreground image (`fgIp`) is selected through a dialog window (created with `GenericDialog`), as well as the number of slices in the stack to be created.

In the plugin's `run()` method (Prog. C.5, line 64), a stack with the required number of slices is created first using the static method `NewImage.createByteImage()`. In the following loop, a varying transparency value α (see Eqn. (5.42)) is computed for each frame, and the corresponding stack image (slice) is replaced by the weighted sum of the two original images. Note that the slices of a stack of size N are numbered $1 \ldots N$ (`getProcessor()` in line 71), in contrast to the usual numbering scheme. A sample result and the corresponding dialog window are shown in Fig. C.7.

C.16 Regions of Interest

A region of interest (ROI) is used to select a particular image region for subsequent processing and is usually specified interactively by the user. ImageJ supports several types of ROI, including:

- rectangular (class `Roi`)
- elliptical (class `OvalRoi`)
- straight line (class `Line`)
- polygon/polyline (classes `PolygonRoi`, `FreehandRoi`)
- point set (class `PointRoi`)

Program C.4
Stack example—alpha blending (*part 1 of 2*). This is an extended version of the alpha blending example described in the main text (Progs. 5.5 and 5.6). It blends two given images with incrementally changing alpha weights and stores the results in a new stack of images.

```
1  import ij.IJ;
2  import ij.ImagePlus;
3  import ij.ImageStack;
4  import ij.WindowManager;
5  import ij.gui.*;
6  import ij.plugin.filter.PlugInFilter;
7  import ij.process.*;
8
9  public class Alpha_Blending_Stack implements PlugInFilter {
10   static int nFrames = 10;
11   ImagePlus fgIm;  // foreground image (chosen interactively)
12
13   public int setup(String arg, ImagePlus imp) {
14     return DOES_8G;}
15
16   boolean runDialog() {
17     // get list of open images
18     int[] windowList = WindowManager.getIDList();
19     if(windowList==null) {
20       IJ.noImage();
21       return false;
22     }
23     String[] windowTitles = new String[windowList.length];
24     for (int i = 0; i < windowList.length; i++) {
25       ImagePlus imp = WindowManager.getImage(windowList[i]);
26       if (imp != null)
27         windowTitles[i] = imp.getShortTitle();
28       else
29         windowTitles[i] = "untitled";
30     }
31     GenericDialog gd = new GenericDialog("Alpha Blending");
32     gd.addChoice("Foreground image:",
33         windowTitles, windowTitles[0]);
34     gd.addNumericField("Frames:", nFrames, 0);
35     gd.showDialog();
36     if (gd.wasCanceled())
37       return false;
38     else {
39       int img2Index = gd.getNextChoiceIndex();
40       fgIm = WindowManager.getImage(windowList[img2Index]);
41       nFrames = (int) gd.getNextNumber();
42       if (nFrames < 2)
43         nFrames = 2;
44       return true;
45     }
46   } // continued...
```

Program C.5
Stack example—alpha blending (*part 2 of 2*).

```
48  // class Alpha_Blending_Stack (continued)
49
50  public void run(ImageProcessor bgIp) {
51    // bgIp = background image
52
53    if(runDialog()) { //open dialog box (returns false if cancelled)
54      int w = bgIp.getWidth();
55      int h = bgIp.getHeight();
56
57      // prepare foreground image
58      ImageProcessor fgIp =
59          fgIm.getProcessor().convertToByte(false);
60      ImageProcessor fgTmpIp = bgIp.duplicate();
61
62      // create image stack
63      ImagePlus movie =
64        NewImage.createByteImage("Movie",w,h,nFrames,0);
65      ImageStack stack = movie.getStack();
66
67      // loop over stack frames
68      for (int i=0; i<nFrames; i++) {
69        // transparency of foreground image
70        double iAlpha = 1.0 - (double)i/(nFrames-1);
71        ImageProcessor iFrame = stack.getProcessor(i+1);
72
73        // copy background image to frame i
74        iFrame.insert(bgIp,0,0);
75        iFrame.multiply(iAlpha);
76
77        // copy foreground image and make transparent
78        fgTmpIp.insert(fgIp,0,0);
79        fgTmpIp.multiply(1-iAlpha);
80
81        // add foreground image frame i
82        ByteBlitter blitter =
83            new ByteBlitter((ByteProcessor)iFrame);
84        blitter.copyBits(fgTmpIp,0,0,Blitter.ADD);
85      }
86
87      // display movie (image stack)
88      movie.show();
89    }
90  }
91
92 } // end of class Alpha_Blending_Stack
```

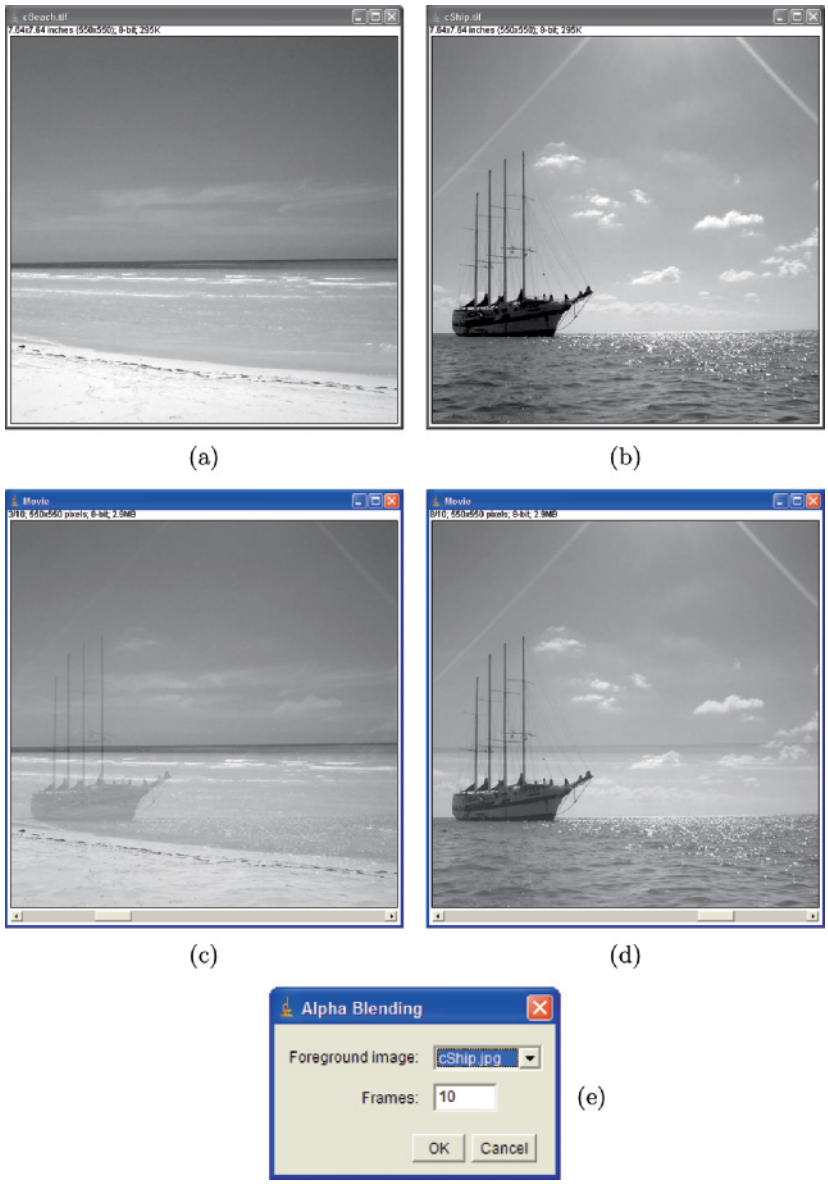


Fig. C.7
Alpha blending example using an image stack (results of Progs. C.4 and C.5). Original images: foreground image (a) and background image (b); frames 3 and 6 of the created image stack (c, d). Note the horizontal "slider" button at the window's bottom for navigating through the stack. Dialog window for selecting the foreground image and the stack size (e).

The corresponding classes are defined in the `ij.gui` package. ROI objects are usually associated with objects of type `ImagePlus`, as described below.

C.16.1 `ImagePlus` (Class)

`Roi getRoi ()`
Returns the current ROI object (of type `Roi` or one of its sub-

classes `Line`, `OvalRoi`, `PolygonRoi`, `TextRoi`) of this image. Returns `null` if the image has no ROI.

`void killRoi ()`
Deletes the image's current region of interest.

`void setRoi (int` *u*`, int` *v*`, int` *w*`, int` *h*`)`
Assigns a rectangular ROI (of size $w \times h$ and upper left corner positioned at (u, v)) to this image and displays it.

`void setRoi (java.awt.Rectangle` *rect*`)`
Assigns the specified rectangular ROI to this image and displays it.

`void setRoi (Roi` *roi*`)`
Assigns the specified ROI (of type `Roi` or any of its subclasses) to this image and displays it. Any existing ROI is deleted if *roi* is `null` or its width or height is zero.

`ImageProcessor getMask ()`
For images with nonrectangular ROIs, this method returns a mask image (of type `ByteProcessor`); otherwise it returns `null`. This method calls the `getMask()` method on the image's `ImageProcessor` object and returns the result (see Sec. C.16.3 for details).

C.16.2 `Roi`, `Line`, `OvalRoi`, `PointRoi`, `PolygonRoi` (Classes)

`Roi (int` *u*`, int` *v*`, int` *width*`, int` *height*`)`
Constructor method: creates a rectangular ROI from the specified parameters.

`Roi (java.awt.Rectangle` *rect*`)`
Constructor method: creates a rectangular ROI from a given AWT `Rectangle` object *rect*.

`Roi (int` *u*`, int` *v*`, ImagePlus` *imp*`)`
Constructor method: starts the process of creating a user-defined rectangular ROI from starting point (*u*, *v*) in the image *imp*. The user determines the size of the region interactively using rubber banding.

`Line (int` *u1*`, int` *v1*`, int` *u2*`, int` *v2*`)`
Constructor method: creates a straight-line ROI between points (*u1*, *v1*) and (*u2*, *v2*).

`Line (int` *u*`, int` *v*`, ImagePlus` *imp*`)`
Constructor method: starts the process of creating a user-defined straight-line ROI from starting point (*u*, *v*) in the image *imp*. The user determines the end of the line interactively using rubber banding.

`OvalRoi (int` *u*`, int` *v*`, int` *width*`, int` *height*`)`
Constructor method: creates an elliptic ROI whose bounding box is determined by the given parameters.

`OvalRoi (int u, int v, ImagePlus imp)`
Constructor method: starts the process of creating a user-defined oval ROI from starting point (u, v) in the image *imp*.

`PolygonRoi (int[] xPnts, int[] yPnts, int n, int type)`
Constructor method: creates a new polygon or polyline ROI from the coordinate arrays *xPnts* and *yPnts*, where *n* is the number of polygon points. Admissible values for *type* are `Roi.POLYGON`, `Roi.FREEROI`, `Roi.TRACED_ROI`, `Roi.POLYLINE`, `Roi.FREELINE`, or `Roi.ANGLE`.

`PolygonRoi (java.awt.Polygon p, int type)`
Creates a new polygon or polyline ROI from a given AWT `Polygon` object. *type* is used as above.

`PolygonRoi (int u, int v, ImagePlus imp)`
Constructor method: starts the process of creating a user-defined polygon or polyline ROI from starting point (u, v) in the image *imp*.

`PointRoi (int[] xPnts, int[] yPnts, int n)`
Constructor method: creates a new point-set ROI from the coordinate arrays *xPnts* and *yPnts*, where *n* is the number of polygon points.

`PointRoi (int u, int v)`
Constructor method: creates a new single-point `PointRoi` at position (u, v).

`PointRoi (int u, int v, ImagePlus imp)`
Creates a new `PointRoi` for the image *imp* using the screen coordinates (u, v).

`boolean contains (int u, int v)`
Returns `true` if the point (u, v) is within this region of interest and `false` otherwise.

C.16.3 `ImageProcessor` (Class)

An `ImageProcessor` object may also have an associated region of interest. The mechanism is similar to but nevertheless different from the one used for `ImagePlus` objects. In particular, a nonrectangular ROI is represented by the bounding rectangle in combination with a mask of the same size specified by a one-dimensional `int` array.

`ImageProcessor getMask ()`
For images with nonrectangular ROIs, this method returns a mask image (of type `ByteProcessor`); otherwise it returns `null`. Pixels "inside" the region of interest have nonzero mask values. This mask image is used for efficiently testing whether a particular (u, v) coordinate is inside or outside the ROI. Note that the origin of the mask image is *not* the same as for the original image but is anchored at the upper left corner of the ROI's bounding box (see Prog. C.6 and Fig. C.8 for an example).

`byte[] getMaskArray ()`
Returns the mask's `byte` array, or `null` if this image has no mask. Note that the origin and the dimensions of the underlying mask image are *not* the same as for the original image. The origin of the mask is anchored at the upper left corner of the ROI's bounding box, and its size is identical to the size of the bounding box.

`Rectangle getRoi ()`
Returns a rectangle (of type `java.awt.Rectangle`) that represents the current region of interest.

`void resetRoi ()`
Sets the region of interest to include the entire image.

`void setMask (ImageProcessor mask)`
Defines a byte mask that limits processing to an irregular ROI. The size of *`mask`* must be the same as the current region of interest. Pixels "inside" the region of interest have nonzero mask values.

`void setRoi (int u, int v, int width, int height)`
Defines a rectangular region of interest and deletes the associated mask if *`rect`* is not the same size as the previous ROI.

`void setRoi (java.awt.Rectangle rect)`
Defines a rectangular region of interest and deletes the associated mask if *`rect`* is not the same size as the previous ROI. If *`rect`* is `null`, the ROI is reset (by calling `resetRoi()`).

`void setRoi (Roi roi)`
Defines a rectangular or nonrectangular region of interest that consists of a rectangular ROI and a mask.

`void setRoi (java.awt.Polygon poly)`
Defines a polygon-shaped region of interest that consists of a rectangular ROI and a mask.

C.16.4 `ImageStack` (Class)

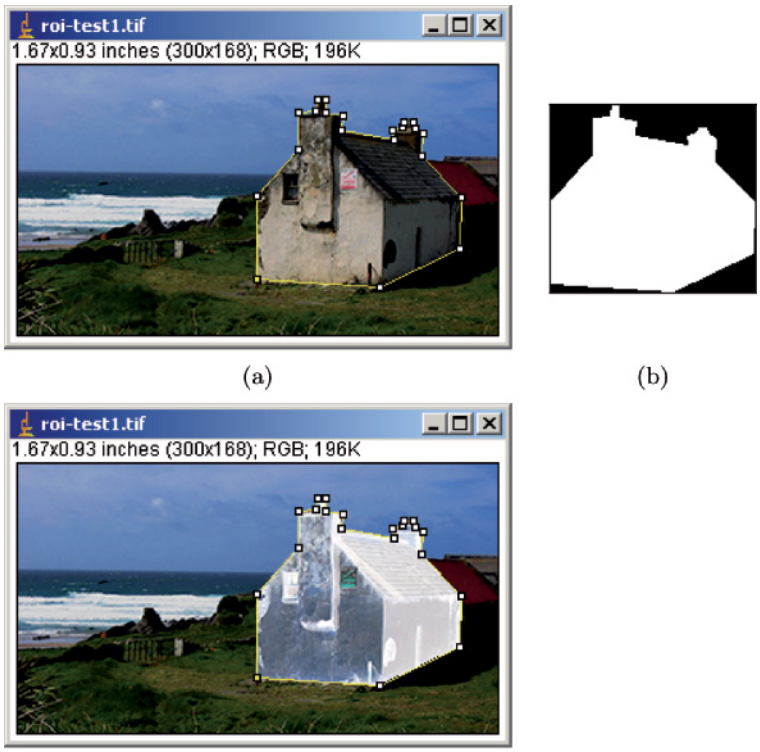
Only rectangular ROIs are applicable to image stacks:

`java.awt.Rectangle getRoi ()`
Returns an AWT `Rectangle` object that represents the current region of interest for this image stack.

`void setRoi (java.awt.Rectangle rect)`
Specifies a rectangular region of interest for this entire image stack.

C.16.5 `IJ` (Class)

The following static ROI methods in class `IJ` apply to the currently active (user-selected) image:



C.17 Image Properties

Fig. C.8
Nonrectangular ROI example. Original image with polygon-shaped selection (a). Binary mask image returned by the `ImageProcessor`'s `getMaskArray()` method (b). Note that the origin of the mask image is positioned at the upper left corner of the ROI's bounding box. Result with pixels inside the ROI being modified (c). See Prog. C.6 for implementation details.

`static void makeLine (int u1, int v1, int u2, int v2)`
Creates a straight-line selection (region of interest) on the currently active image (i. e., the image selected by the user).

`static void makeOval (int u, int v, int w, int h)`
Creates an elliptical region of interest of size ($w \times h$) on the currently active image.

`static void makeRectangle (int u, int v, int w, int h)`
Creates a rectangular region of interest of size ($w \times h$) on the currently active image.

C.17 Image Properties

Sometimes it is necessary to pass results from one plugin to another, but the `run()` method itself does not provide a return value. One solution is to deposit the results of a plugin as a *property* in the corresponding `ImagePlus` object. A property consists of a *key*/*value* pair, where *key* is a string and *value* may be any Java object. In ImageJ, this mechanism is implemented as a hash table and supported by the following methods.

Program C.6
Working with a nonrectangular region of interest (ROI). This example shows the use of a mask image for processing nonrectangular ROIs. Objects of class `ImageProcessor` always return a valid bounding box (`Rectangle`), whether an ROI is selected or not (line 15). If no ROI is selected, the resulting rectangle covers the full image. `ImageProcessor` only returns a mask image (line 16) if the specified ROI is nonrectangular (e. g., `OvalRoi`, `PolygonRoi`); otherwise `null` is returned. The processing loop (lines 29–30) only scans over the bounding rectangle of the ROI. Inside this loop (line 31), the mask image is used to determine if pixels are inside the ROI or not. Only the pixels inside the ROI are modified (see Fig. C.8 for example images).

```
1  import ij.ImagePlus;
2  import ij.plugin.filter.PlugInFilter;
3  import ij.process.ImageProcessor;
4  import java.awt.Rectangle;
5
6  public class Roi_Demo implements PlugInFilter {
7    boolean showMask = true;
8
9    public int setup(String arg, ImagePlus imp) {
10     return DOES_RGB;
11   }
12
13   public void run(ImageProcessor ip) {
14
15     Rectangle roi = ip.getRoi();
16     ImageProcessor mask = ip.getMask();
17     boolean hasMask = (mask != null);
18     if (hasMask && showMask) {
19       (new ImagePlus("The Mask", mask)).show();
20     }
21
22     // ROI corner coordinates:
23     int rLeft = roi.x;
24     int rTop = roi.y;
25     int rRight = rLeft + roi.width;
26     int rBottom = rTop + roi.height;
27
28     // process all pixels inside the ROI
29     for (int v = rTop; v < rBottom; v++) {
30       for (int u = rLeft; u < rRight; u++) {
31         if (!hasMask || mask.getPixel(u-rLeft, v-rTop) > 0) {
32           int p = ip.getPixel(u, v);
33           ip.putPixel(u, v, ~p); // invert pixel values
34         }
35       }
36     }
37   }
38
39 } // end of class Roi_Demo
```

C.17.1 `ImagePlus` (Class)

`java.util.Properties getProperties()`
Returns the `Properties` object (a hash table) with all property entries of this image, or `null` if the image has no properties.

`Object getProperty(String key)`
Returns the property value associated with ***key***, or `null` if no such property exists.

`void setProperty(String key, Object value)`
Adds a property with name *key* and content *value* to this image's properties. If a property with the same key already exists for this image, it is replaced by the new value. If *value* is `null`, the corresponding property is deleted.

Example

Program C.7 shows a simple example for the use of properties involving two ImageJ plugins. The first plugin (`Plugin_1`) computes the histogram of the image and inserts the result as a property with the key `HISTOGRAM` (line 17). The second plugin (`Plugin_2`) uses the same key to retrieve the histogram from the image's properties (line 36) for further processing. In this example, the common key is made available through the static variable `HistKey`, defined in class `Plugin_1` (line 35).

C.18 User Interaction

C.18.1 IJ (Class)

Text output, logging

`static void error(String msg)`
Displays the message `msg` in a dialog box titled "Image".

`static void error(String title, String msg)`
Displays the message `msg` in a dialog box with the specified title.

`static void log(String msg)`
Displays a line of text (*msg*) in ImageJ's "Log" window.

`static void write(String msg)`
Writes a line of text (*msg*) in ImageJ's "Results" window.

Dialog boxes

`static double getNumber(String prompt, double defVal)`
Allows the user to enter a number in a dialog box.

`static String getString(String prompt, double defStr)`
Allows the user to enter a string in a dialog box.

`static void noImage()`
Displays a "no images are open" dialog box.

`static void showMessage(String msg)`
Displays a message in a dialog box titled "Message".

`static void showMessage(String title, String msg)`
Displays a message in a dialog box with the specified title.

`static boolean showMessageWithCancel(String title, String msg)`
Displays a message in a dialog box with the specified title. Returns `false` if the user pressed "Cancel".

Program C.7
Use of image properties (example). Properties can be used to pass results from one plugin to another. Here, the first plugin (`Plugin_1`) computes the histogram of the image in its `run()` method and attaches the result as a property to the `ImagePlus` object `imp` (line 17). The second plugin retrieves the histogram from the image (line 36) for further processing. Note that the typecast to `int[]` in line 36 is potentially dangerous and should not be used without additional measures.

`Plugin_1.java`:

```
1  import ij.ImagePlus;
2  import ij.plugin.filter.PlugInFilter;
3  import ij.process.ImageProcessor;
4
5  public class Plugin_1 implements PlugInFilter {
6    ImagePlus im;
7    public static final String HistKey = "HISTOGRAM";
8
9    public int setup(String arg, ImagePlus im) {
10     this.im = im;
11     return DOES_ALL + NO_CHANGES;
12   }
13
14   public void run(ImageProcessor ip) {
15     int[] hist = ip.getHistogram();
16     // add histogram to image properties:
17     im.setProperty(HistKey, hist);
18   }
19
20 } // end of class Plugin_1
```

`Plugin_2.java`:

```
21 import ij.IJ;
22 import ij.ImagePlus;
23 import ij.plugin.filter.PlugInFilter;
24 import ij.process.ImageProcessor;
25
26 public class Plugin_2 implements PlugInFilter {
27   ImagePlus im;
28
29   public int setup(String arg, ImagePlus im) {
30     this.im = im;
31     return DOES_ALL;
32   }
33
34   public void run(ImageProcessor ip) {
35     String key = Plugin1_.HistKey;
36     int[] hist = (int[]) im.getProperty(key);
37     if (hist == null){
38       IJ.error("This image has no histogram");
39     }
40     else {
41       // process histogram ...
42     }
43   }
44
45 } // end of class Plugin_2
```

Progress and status bar

`static void showProgress(double progress)`
Updates the progress bar in ImageJ's main window, where $0 \leq$ *progress* < 1. The length of the displayed bar is *progress* times its maximum length. The progress bar is not displayed if the time between the first and second calls to this method is less than 30 milliseconds. The bar is erased if *progress* ≥ 1.

`static void showProgress(int i, int n)`
Updates the progress bar in ImageJ's main window, where $0 \leq i < n$ is the current index and n is the maximum index. The length of the displayed bar is (i/n) times its maximum length. The bar is erased if $i \geq n$.

`static void showStatus(String msg)`
Displays a message in the ImageJ status bar.

Keyboard queries

`static boolean altKeyDown()`
Returns `true` if the alt key is down.

`static boolean escapePressed()`
Returns `true` if the esc key was pressed since the last ImageJ command started to execute or since `resetEscape()` was called.

`static void resetEscape()`
This method sets the esc key to the "up" position.

`static boolean shiftKeyDown()`
Returns `true` if the shift key is down.

`static boolean spaceBarDown()`
Returns `true` if the space bar is down.

Miscellaneous

`static void beep()`
Emits a beep signal.

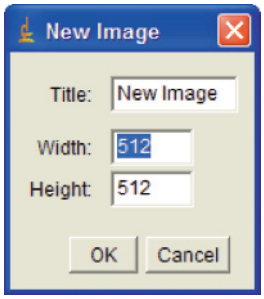
`static ImagePlus getImage()`
Returns a reference to the *active* image, i. e., the `ImagePlus` object currently selected by the user.

`static void wait(int msecs)`
Waits (suspends processing) for *msecs* milliseconds.

C.18.2 `GenericDialog` (Class)

The class `GenericDialog` offers a simple mechanism for creating dialog windows containing multiple fields of various types. The layout of these dialog windows is created automatically. A small example and the corresponding results are shown in Prog. C.8. Other examples can be found in Secs. 5.8.5 and C.15.3. For additional details, see the ImageJ online documentation and the tutorial in [4].

Program C.8
Use of the `GenericDialog` class (example). This ImageJ plugin creates a new image with the title and size specified interactively by the user. The corresponding dialog window is shown below.




```
1  import ij.ImagePlus;
2  import ij.gui.GenericDialog;
3  import ij.gui.NewImage;
4  import ij.plugin.PlugIn;
5
6  public class Generic_Dialog_Example implements PlugIn {
7    static String title = "New Image";
8    static int width = 512;
9    static int height = 512;
10
11   public void run(String arg) {
12     GenericDialog gd = new GenericDialog("New Image");
13     gd.addStringField("Title:", title);
14     gd.addNumericField("Width:", width, 0);
15     gd.addNumericField("Height:", height, 0);
16     gd.showDialog();
17     if (gd.wasCanceled())
18       return;
19     title = gd.getNextString();
20     width = (int) gd.getNextNumber();
21     height = (int) gd.getNextNumber();
22
23     ImagePlus imp = NewImage.createByteImage(
24       title, width, height, 1, NewImage.FILL_WHITE);
25     imp.show();
26   }
27
28 } // end of class Generic_Dialog_Example
```

C.19 Plugins

ImageJ plugins come in two different variants, both of which are implemented as Java "interfaces":

- `PlugIn`: can be applied without any image and is used to acquire images, display windows, etc.
- `PlugInFilter`: is applied to process an existing image.

C.19.1 `PlugIn` (Interface)

The `PlugIn` interface only specifies the implementation of a single method:

`void run (String` ***`arg`***`)`
: Starts this plugin, where ***`arg`*** is used to specify options (may be an empty string).

C.19.2 PlugInFilter (Interface)

The PlugInFilter interface requires the implementation of the following two methods:

`void run (ImageProcessor ip)`
: Starts this plugin. The parameter *ip* specifies the image (ImageProcessor object) to which this plugin is applied.

`int setup (String arg, ImagePlus im)`
: When the plugin is applied, the setup() method is called before the run() method. The parameter *im* refers to the target image (ImagePlus object) and *not* its ImageProcessor! If access to *im* is required within the run() method, it is usually assigned to a suitable object variable of this plugin by the setup() method (see, e. g., Prog. C.3). Note that the plugin's setup() method is called even when no images are open—in this case null is passed instead of the currently active image! The return value of the setup() method is a 32-bit (int) pattern, where each bit is a flag that corresponds to a certain feature of that plugin. Different flags can be easily combined by summing predefined constants, as listed below. The run() method of the plugin is not invoked if setup() returns DONE.

The following *return flags* for the setup() method are defined as int constants in the class PlugInFilter:

DOES_8G
: This plugin accepts (unsigned) 8-bit grayscale images.

DOES_8C
: This plugin accepts 8-bit indexed color images.

DOES_16
: This plugin accepts (unsigned) 16-bit grayscale images.

DOES_32
: This plugin accepts 32-bit float images.

DOES_RGB
: This plugin accepts 3×8 bit RGB color images.

DOES_ALL
: This plugin accepts any type of image (DOES_ALL = DOES_8G + DOES_8C + DOES_16 + DOES_32 + DOES_RGB).

DOES_STACKS
: The plugin's run method shall be applied to all slices of a stack.

DONE
: The plugin's run method shall not be invoked.

NO_CHANGES
: The plugin does not modify the original image.

NO_IMAGE_REQUIRED
: This plugin does not require that an image be open. In this case, null is passed to the run() method as the argument for *ip*.

`NO_UNDO`
: This plugin does not require undo.

`ROI_REQUIRED`
: This plugin requires a region of interest (ROI) to be explicitly specified.

`STACK_REQUIRED`
: This plugin requires a stack of images.

`SUPPORTS_MASKING`
: For nonrectangular ROIs, this plugin wants ImageJ to automatically restore that part of the image that is inside the bounding rectangle but outside of the ROI. This greatly simplifies the use of nonrectangular ROIs.

The flags above are integer values, each with only a single bit set (1) and the remaining bits being zero. Flags can be combined either by a bitwise OR operation (e. g., `DOES_8G | DOES_16`) or by simple arithmetic addition. For example, the `setup()` method for a `PlugInFilter` that can handle 8- *and* 16-bit grayscale images *and* does not modify the original image could be defined as follows:

```
public int setup (String arg, ImagePlus im) {
   return DOES_8G + DOES_16G + NO_CHANGES;
}
```

C.19.3 Executing Plugins: `IJ` (Class)

`static Object runPlugIn(String` *`className`*`, String` *`arg`*`)`
: Creates a new plugin object of class *`className`* and executes its `run()` method, passing the string argument *`arg`*. If the plugin is of type `PlugInFilter`, the new instance is applied to the currently active image by first invoking the `setup()` method. The `runPlugIn()` method returns a reference to the new plugin object.

C.20 Window Management

C.20.1 `WindowManager` (Class)

The class `ij.WindowManager` defines a set of static methods for manipulating the screen windows in ImageJ:

`static boolean closeAllWindows ()`
: Closes all windows and returns `true` if successful. Stops and returns `false` if the "save changes" dialog is canceled for any unsaved image.

`static ImagePlus getCurrentImage ()`
: Returns the currently active image of type `ImagePlus`.

`static ImageWindow getCurrentWindow ()`
: Returns the currently active window of type `ImageWindow`.

`static int[] getIDList ()`
: Returns an array containing the IDs of all open images, or `null` if no image is open. The image IDs are negative integer values.

`static ImagePlus getImage (int` *`imageID`*`)`
: For *`imageID`* less than zero, this method returns the `ImagePlus` object with the specified *`imageID`*. It returns `null` if either *`imageID`* is zero, no open image has a matching ID, or no images are open at all. For *`imageID`* greater than zero, it returns the image at the corresponding position in the image array delivered by `getIDList()`.

`static ImagePlus getImage (String` *`title`*`)`
: Returns the first image that has the specified title or `null` if no such image is found.

`static int getImageCount ()`
: Returns the number of open images.

`static ImagePlus getTempCurrentImage ()`
: Returns the image temporarily made current (by `setTempCurrentImage()`), which may be `null`.

`static int getWindowCount ()`
: Returns the number of open image windows.

`static void putBehind ()`
: Moves the current active image to the back and activates the next image in a cyclic fashion.

`static void repaintImageWindows ()`
: Repaints all open image windows.

`static void setCurrentWindow (ImageWindow` *`win`*`)`
: Makes the specified image active.

`static void setTempCurrentImage (ImagePlus` *`im`*`)`
: Makes *`im`* temporarily the active image and thus allows processing of images that are currently not displayed in a window. Another call with the argument `null` reverts to the previously active image.

C.21 Additional Functions

C.21.1 ImagePlus (Class)

Locking and unlocking images

ImageJ plugins may execute simultaneously as different Java threads in the same runtime environment. Locking may be required to avoid mutual interferences between plugins that operate on the same image.

`boolean lock ()`
: Locks this image so other threads can test to see if it is in use. Returns `true` if the image was successfully locked. Beeps, displays a message in the status bar, and returns `false` if the image is already locked.

`boolean lockSilently ()`
: Similar to lock but does not beep. Displays an error message if the attempt to lock the image fails.

`void unlock ()`
: Unlocks this image.

Internal clipboard

ImageJ maintains a single internal clipboard image (as an `ImagePlus` object) that can be manipulated interactively with the Edit menu or accessed through the following methods:

`void copy (boolean cut)`
: Copies the contents of the current selection (region of interest) to the internal clipboard. The entire image is copied if there is no selection. The selected part of the image is *cleared* (i. e., filled with the current background value or color) if `cut` is `true`.

`void paste ()`
: Inserts the contents of the internal clipboard into this (`ImagePlus`) image. If the target image has a selection the same size as the image on the clipboard, the clipboard content is inserted into that selection, otherwise the clipboard content is inserted into the center of the image.

`static ImagePlus getClipboard ()`
: Returns the internal clipboard (as an `ImagePlus` object) or `null` if the internal clipboard is empty. Note that this is a *static* method and is thus called in the form `ImagePlus.getClipboard()`.

File information

`FileInfo getFileInfo ()`
: Returns a `FileInfo` object containing information, including the pixel array, needed to *save* this image. Use `getOriginalFileInfo()` to get a copy of the `FileInfo` object used to *open* the image.

`FileInfo getOriginalFileInfo()`
Returns the `FileInfo` object that was used to *open* this image. This includes fields such as `fileName` (`String`), `directory` (`String`), and `description` (`String`). Returns `null` for images created internally or using the File→New command.

C.21.2 IJ (Class)

Directory information

`static String getDirectory(String target)`
Returns the path to ImageJ's *home*, *startup*, *plugins*, *macros*, *temp*, or *image* directory, depending on the value of *`target`* (`"home"`, `"startup"`, `"plugins"`, `"macros"`, `"temp"`, or `"image"`). If *`target`* (which may not be `null`) is none of the above, the method displays a dialog and returns the path to the directory selected by the user. `null` is returned if the specified directory is not found or the user cancels the dialog.

Memory management

`static long currentMemory()`
Returns the amount of memory (in bytes) currently being used by ImageJ.

`static String freeMemory()`
Runs the garbage collector and returns a string showing how much of the available memory is in use.

`static long maxMemory()`
Returns the maximum amount of memory available to ImageJ or zero if ImageJ is unable to determine this limit.

System information

`static String getVersion()`
Returns ImageJ's version number as a string.

`static boolean isJava2()`
Returns `true` if ImageJ is running on Java 2.

`static boolean isJava14()`
Returns `true` if ImageJ is running on a Java 1.4 or greater JVM.

`static boolean isMacintosh()`
Returns `true` if the current platform is a Macintosh computer.

`static boolean isMacOSX()`
Returns `true` if the current platform is a Macintosh computer running OS X.

`static boolean isWindows()`
Returns `true` if this machine is running Windows.

`static boolean versionLessThan(String version)`
Displays an error message and returns false if the current version of ImageJ is less than the one specified.

Appendix D

Source Code

D.1 Harris Corner Detector

The following Java source code represents a complete implementation of the Harris corner detector, as described in Ch. 8. It consists of the following classes (files):

- `Harris_Corner_Plugin`: a sample ImageJ plugin that demonstrates the use of the corner detector.
- `Corner` (p. 527): a class representing an individual corner object.
- `HarrisCornerDetector` (p. 527): the actual corner detector. This class is instantiated to create a corner detector for a given image.

D.1.1 `Harris_Corner_Plugin` (Class)

```
1  import harris.HarrisCornerDetector;
2  import ij.IJ;
3  import ij.ImagePlus;
4  import ij.gui.GenericDialog;
5  import ij.plugin.filter.PlugInFilter;
6  import ij.process.ImageProcessor;
7
8  public class Harris_Corner_Plugin implements PlugInFilter {
9      ImagePlus im;
10     static float alpha = HarrisCornerDetector.DEFAULT_ALPHA;
11     static int threshold = HarrisCornerDetector.
           DEFAULT_THRESHOLD;
12     static int nmax = 0; //points to show
13
14     public int setup(String arg, ImagePlus im) {
15       this.im = im;
```

```
16        if (arg.equals("about")) {
17            showAbout();
18            return DONE;
19        }
20        return DOES_8G + NO_CHANGES;
21    }
22
23    public void run(ImageProcessor ip) {
24        if (!showDialog()) return; //dialog canceled or error
25        HarrisCornerDetector hcd =
26            new HarrisCornerDetector(ip,alpha,threshold);
27        hcd.findCorners();
28        ImageProcessor result = hcd.showCornerPoints(ip);
29        ImagePlus win =
30            new ImagePlus("Corners from " + im.getTitle(),
                  result);
31        win.show();
32    }
33
34    void showAbout() {
35        String cn = getClass().getName();
36        IJ.showMessage("About "+cn+" ...",
37            "Harris Corner Detector");
38    }
39
40    private boolean showDialog() {
41      // display dialog, and return false if canceled or in error.
42      GenericDialog dlg = new GenericDialog("Harris Corner
            Detector", IJ.getInstance());
43      float def_alpha = HarrisCornerDetector.DEFAULT_ALPHA;
44      dlg.addNumericField("Alpha (default: "+def_alpha+")",
            alpha, 3);
45      int def_threshold = HarrisCornerDetector.
            DEFAULT_THRESHOLD;
46      dlg.addNumericField("Threshold (default: "+def_threshold+
            ")", threshold, 0);
47      dlg.addNumericField("Max. points (0 = show all)", nmax,
            0);
48      dlg.showDialog();
49      if(dlg.wasCanceled())
50        return false;
51      if(dlg.invalidNumber()) {
52        IJ.showMessage("Error", "Invalid input number");
53        return false;
54      }
55      alpha = (float) dlg.getNextNumber();
56      threshold = (int) dlg.getNextNumber();
57      nmax = (int) dlg.getNextNumber();
58      return true;
59    }
60 } // end of class Harris_Corner_Plugin
```

D.1.2 File Corner (Class)

```
1  package harris;
2  import ij.process.ImageProcessor;
3
4  class Corner implements Comparable {
5    int u;
6    int v;
7    float q;
8
9    Corner (int u, int v, float q) {
10     this.u = u;
11     this.v = v;
12     this.q = q;
13   }
14
15   public int compareTo (Object obj) {
16     // used for sorting corners by corner strength q
17     Corner c2 = (Corner) obj;
18     if (this.q > c2.q) return -1;
19     if (this.q < c2.q) return 1;
20     else return 0;
21   }
22
23   double dist2 (Corner c2) {
24     // returns the squared distance between this corner and corner c2
25     int dx = this.u - c2.u;
26     int dy = this.v - c2.v;
27     return (dx*dx)+(dy*dy);
28   }
29
30   void draw(ImageProcessor ip) {
31     // draw this corner as a black cross in ip
32     int paintvalue = 0; // black
33     int size = 2;
34     ip.setValue(paintvalue);
35     ip.drawLine(u-size,v,u+size,v);
36     ip.drawLine(u,v-size,u,v+size);
37   }
38
39 } // end of class Corner
```

D.1.3 File HarrisCornerDetector (Class)

```
1  package harris;
2  import ij.IJ;
3  import ij.ImagePlus;
4  import ij.plugin.filter.Convolver;
5  import ij.process.Blitter;
6  import ij.process.ByteProcessor;
```

```
7  import ij.process.FloatProcessor;
8  import ij.process.ImageProcessor;
9  import java.util.Arrays;
10 import java.util.Collections;
11 import java.util.List;
12 import java.util.Vector;
13
14 public class HarrisCornerDetector {
15
16   public static final float DEFAULT_ALPHA = 0.050f;
17   public static final int DEFAULT_THRESHOLD = 20000;
18   float alpha = DEFAULT_ALPHA;
19   int threshold = DEFAULT_THRESHOLD;
20   double dmin = 10;
21   final int border = 20;
22
23   // filter kernels (1D part of separable 2D filters)
24   final float[] pfilt = {0.223755f,0.552490f,0.223755f};
25   final float[] dfilt = {0.453014f,0.0f,-0.453014f};
26   final float[] bfilt = {0.01563f,0.09375f,0.234375f,0.3125f
         ,0.234375f,0.09375f,0.01563f};
27                 // = [1, 6, 15, 20, 15, 6, 1]/64
28   ImageProcessor ipOrig;
29   FloatProcessor A;
30   FloatProcessor B;
31   FloatProcessor C;
32   FloatProcessor Q;
33   List<Corner> corners;
34
35   HarrisCornerDetector(ImageProcessor ip) {
36     this.ipOrig = ip;
37   }
38
39   public HarrisCornerDetector(ImageProcessor ip,
40          float alpha, int threshold)
41   {
42     this.ipOrig = ip;
43     this.alpha = alpha;
44     this.threshold = threshold;
45   }
46
47   public void findCorners() {
48     makeDerivatives();
49     makeCrf(); //corner response function (CRF)
50     corners = collectCorners(border);
51     corners = cleanupCorners(corners);
52   }
53
54   void makeDerivatives() {
55     FloatProcessor Ix =
56            (FloatProcessor) ipOrig.convertToFloat();
```

```
57    FloatProcessor Iy =
58           (FloatProcessor) ipOrig.convertToFloat();
59
60    Ix = convolve1h(convolve1h(Ix,pfilt),dfilt);
61    Iy = convolve1v(convolve1v(Iy,pfilt),dfilt);
62
63    A = sqr((FloatProcessor) Ix.duplicate());
64    A = convolve2(A,bfilt);
65
66    B = sqr((FloatProcessor) Iy.duplicate());
67    B = convolve2(B,bfilt);
68
69    C = mult((FloatProcessor)Ix.duplicate(),Iy);
70    C = convolve2(C,bfilt);
71  }
72
73  void makeCrf() { // corner response function (CRF)
74    int w = ipOrig.getWidth();
75    int h = ipOrig.getHeight();
76    Q = new FloatProcessor(w,h);
77    float[] Apix = (float[]) A.getPixels();
78    float[] Bpix = (float[]) B.getPixels();
79    float[] Cpix = (float[]) C.getPixels();
80    float[] Qpix = (float[]) Q.getPixels();
81    for (int v=0; v<h; v++) {
82      for (int u=0; u<w; u++) {
83        int i = v*w+u;
84        float a = Apix[i], b = Bpix[i], c = Cpix[i];
85        float det = a*b-c*c;
86        float trace = a+b;
87        Qpix[i] = det - alpha * (trace * trace);
88      }
89    }
90  }
91
92  List<Corner> collectCorners(int border) {
93    List<Corner> cornerList = new Vector<Corner>(1000);
94    int w = Q.getWidth();
95    int h = Q.getHeight();
96    float[] Qpix = (float[]) Q.getPixels();
97    for (int v=border; v<h-border; v++){
98      for (int u=border; u<w-border; u++) {
99        float q = Qpix[v*w+u];
100       if (q>threshold && isLocalMax(Q,u,v)) {
101         Corner c = new Corner(u,v,q);
102         cornerList.add(c);
103       }
104     }
105   }
106   Collections.sort(cornerList);
107   return cornerList;
```

```
108  }
109
110  List<Corner> cleanupCorners(List<Corner> corners) {
111    double dmin2 = dmin*dmin;
112    Corner[] cornerArray = new Corner[corners.size()];
113    cornerArray = corners.toArray(cornerArray);
114    List<Corner> goodCorners =
115        new Vector<Corner>(corners.size());
116    for (int i=0; i<cornerArray.length; i++){
117      if (cornerArray[i] != null){
118        Corner c1 = cornerArray[i];
119        goodCorners.add(c1);
120        // delete all remaining corners close to c
121        for (int j=i+1; j<cornerArray.length; j++){
122          if (cornerArray[j] != null){
123            Corner c2 = cornerArray[j];
124            if (c1.dist2(c2)<dmin2)
125              cornerArray[j] = null; //delete corner
126          }
127        }
128      }
129    }
130    return goodCorners;
131  }
132
133  void printCornerPoints(List<Corner> crf) {
134    int i = 0;
135    for (Corner ipt: crf){
136      IJ.write((i++) + ": " + (int)ipt.q + " " + ipt.u + " " +
           ipt.v);
137    }
138  }
139
140  public ImageProcessor showCornerPoints(ImageProcessor ip) {
141    ByteProcessor ipResult = (ByteProcessor)ip.duplicate();
142    // change background image contrast and brightness
143    int[] lookupTable = new int[256];
144    for (int i=0; i<256; i++){
145      lookupTable[i] = 128 + (i/2);
146    }
147    ipResult.applyTable(lookupTable);
148    // draw corners:
149    for (Corner c: corners) {
150      c.draw(ipResult);
151    }
152    return ipResult;
153  }
154
155  void showProcessor(ImageProcessor ip, String title) {
156    ImagePlus win = new ImagePlus(title,ip);
157    win.show();
```

```
158  }
159
160  // utility methods for float processors —
161
162  static FloatProcessor convolve1h
163                (FloatProcessor p, float[] h) {
164    Convolver conv = new Convolver();
165    conv.setNormalize(false);
166    conv.convolve(p, h, 1, h.length);
167    return p;
168  }
169
170  static FloatProcessor convolve1v
171                (FloatProcessor p, float[] h) {
172    Convolver conv = new Convolver();
173    conv.setNormalize(false);
174    conv.convolve(p, h, h.length, 1);
175    return p;
176  }
177
178  static FloatProcessor convolve2
179                (FloatProcessor p, float[] h) {
180    convolve1h(p,h);
181    convolve1v(p,h);
182    return p;
183  }
184
185  static FloatProcessor sqr (FloatProcessor fp1) {
186    fp1.sqr();
187    return fp1;
188  }
189
190  static FloatProcessor mult (FloatProcessor fp1,
         FloatProcessor fp2) {
191    int mode = Blitter.MULTIPLY;
192    fp1.copyBits(fp2, 0, 0, mode);
193    return fp1;
194  }
195
196  static boolean isLocalMax (FloatProcessor fp,int u,int v) {
197    int w = fp.getWidth();
198    int h = fp.getHeight();
199    if (u<=0 || u>=w-1 || v<=0 || v>=h-1)
200      return false;
201    else {
202      float[] pix = (float[]) fp.getPixels();
203      int i0 = (v-1)*w+u, i1 = v*w+u, i2 = (v+1)*w+u;
204      float cp = pix[i1];
205      return
206        cp > pix[i0-1] && cp > pix[i0] && cp > pix[i0+1] &&
207        cp > pix[i1-1] &&              cp > pix[i1+1] &&
```

```
208         cp > pix[i2-1] && cp > pix[i2] && cp > pix[i2+1] ;
209     }
210   }
211
212 } // end of class HarrisCornerDetector
```

D.2 Combined Region Labeling and Contour Tracing

The following Java source code represents a complete implementation of the combined region labeling and contour tracing algorithm described in Sec. 11.2. It consists of the following classes (files):

- `Contour_Tracing_Plugin`: a sample ImageJ plugin that demonstrates the use of this region labeling implementation.
- `Contour` (p. 533): a class representing a contour object.
- `BinaryRegion` (p. 535): a class representing a binary region object.
- `ContourTracer` (p. 536): the actual region labeler and contour tracer. This class is instantiated to create a region labeler for a given image.
- `ContourOverlay` (p. 541): a class for displaying contours as vector graphics on top of images.

D.2.1 `Contour_Tracing_Plugin` (Class)

```
1  import java.util.List;
2  import regions.BinaryRegion;
3  import regions.RegionLabeling;
4  import contours.Contour;
5  import contours.ContourOverlay;
6  import contours.ContourTracer;
7
8  import ij.IJ;
9  import ij.ImagePlus;
10 import ij.gui.ImageWindow;
11 import ij.plugin.filter.PlugInFilter;
12 import ij.process.ImageProcessor;
13
14 // This plugin implements the combined contour tracing and
15 // component labeling algorithm as described in [22].
16 // It uses the ContourTracer class to create lists of points
17 // representing the internal and external contours of each region in
18 // the binary image. Instead of drawing directly into the image,
19 // we make use of ImageJ's ImageCanvas to draw the contours
20 // in a separate layer on top of the image. It illustrates how to use
21 // the Java2D API to draw the polygons and scale and transform
22 // them to match ImageJ's zooming.
23
24
```

```
25 public class Contour_Tracing_Plugin implements PlugInFilter
26 {
27   ImagePlus origImage = null;
28   String origTitle = null;
29   static boolean verbose = true;
30
31   public int setup(String arg, ImagePlus im) {
32     origImage = im;
33     origTitle = im.getTitle();
34     RegionLabeling.setVerbose(verbose);
35     return DOES_8G + NO_CHANGES;
36   }
37
38   public void run(ImageProcessor ip) {
39     ImageProcessor ip2 = ip.duplicate();
40
41     // label regions and trace contours
42     ContourTracer tracer = new ContourTracer(ip2);
43
44     // extract contours and regions
45     List<Contour> outerContours = tracer.getOuterContours();
46     List<Contour> innerContours = tracer.getInnerContours();
47     List<BinaryRegion> regions = tracer.getRegions();
48     if (verbose) printRegions(regions);
49
50     // change lookup table to show gray regions
51     ip2.setMinAndMax(0,512);
52     // create an image with overlay to show the contours
53     ImagePlus im2 = new ImagePlus("Contours of " + origTitle,
           ip2);
54     ContourOverlay cc = new ContourOverlay(im2, outerContours,
           innerContours);
55     new ImageWindow(im2, cc);
56   }
57
58   void printRegions(List<BinaryRegion> regions) {
59     for (BinaryRegion r: regions) {
60       IJ.write("" + r);
61     }
62   }
63
64 } // end of class Contour_Tracing_Plugin
```

D.2.2 Contour (Class)

```
1 package contours;
2 import ij.IJ;
3 import java.awt.Point;
4 import java.awt.Polygon;
5 import java.awt.Shape;
```

```
6  import java.awt.geom.Ellipse2D;
7  import java.util.ArrayList;
8  import java.util.Iterator;
9  import java.util.List;
10
11 public class Contour {
12   static int INITIAL_SIZE = 50;
13   int label;
14   List<Point> points;
15
16   Contour (int label, int size) {
17     this.label = label;
18     points = new ArrayList<Point>(size);
19   }
20
21   Contour (int label) {
22     this.label = label;
23     points = new ArrayList<Point>(INITIAL_SIZE);
24   }
25
26   void addPoint (Point n) {
27     points.add(n);
28   }
29
30   Shape makePolygon() {
31     int m = points.size();
32     if (m>1) {
33       int[] xPoints = new int[m];
34       int[] yPoints = new int[m];
35       int k = 0;
36       Iterator<Point> itr = points.iterator();
37       while (itr.hasNext() && k < m) {
38         Point cpt = itr.next();
39         xPoints[k] = cpt.x;
40         yPoints[k] = cpt.y;
41         k = k + 1;
42       }
43       return new Polygon(xPoints, yPoints, m);
44     }
45     else { // use circles for isolated pixels
46       Point cpt = points.get(0);
47       return new Ellipse2D.Double
48           (cpt.x-0.1, cpt.y-0.1, 0.2, 0.2);
49     }
50   }
51
52   static Shape[] makePolygons(List<Contour> contours) {
53     if (contours == null)
54       return null;
55     else {
56       Shape[] pa = new Shape[contours.size()];
```

```
57      int i = 0;
58      for (Contour c: contours) {
59        pa[i] = c.makePolygon();
60        i = i + 1;
61      }
62      return pa;
63    }
64  }
65
66  void moveBy (int dx, int dy) {
67    for (Point pt: points) {
68      pt.translate(dx,dy);
69    }
70  }
71
72  static void moveContoursBy
73           (List<Contour> contours, int dx, int dy) {
74    for (Contour c: contours) {
75      c.moveBy(dx, dy);
76    }
77  }
78
79 } // end of class Contour
```

D.2.3 BinaryRegion (Class)

```
1 package regions;
2 import java.awt.Rectangle;
3 import java.awt.geom.Point2D;
4
5 public class BinaryRegion {
6   int label;
7   int numberOfPixels = 0;
8   double xc = Double.NaN;
9   double yc = Double.NaN;
10  int left = Integer.MAX_VALUE;
11  int right = -1;
12  int top = Integer.MAX_VALUE;
13  int bottom = -1;
14
15  int x_sum = 0;
16  int y_sum = 0;
17  int x2_sum = 0;
18  int y2_sum = 0;
19
20  public BinaryRegion(int id){
21    this.label = id;
22  }
23
24  public int getSize() {
```

```
25    return this.numberOfPixels;
26  }
27
28  public Rectangle getBoundingBox() {
29    if (left == Integer.MAX_VALUE)
30      return null;
31    else
32      return new Rectangle
33        (left, top, right-left+1, bottom-top+1);
34  }
35
36  public Point2D.Double getCenter(){
37    if (Double.isNaN(xc))
38      return null;
39    else
40      return new Point2D.Double(xc, yc);
41  }
42
43  public void addPixel(int x, int y){
44    numberOfPixels = numberOfPixels + 1;
45    x_sum = x_sum + x;
46    y_sum = y_sum + y;
47    x2_sum = x2_sum + x*x;
48    y2_sum = y2_sum + y*y;
49    if (x<left) left = x;
50    if (y<top) top = y;
51    if (x>right) right = x;
52    if (y>bottom) bottom = y;
53  }
54
55  public void update(){
56    if (numberOfPixels > 0){
57      xc = x_sum / numberOfPixels;
58      yc = y_sum / numberOfPixels;
59    }
60  }
61
62 } // end of class BinaryRegion
```

D.2.4 ContourTracer (Class)

```
1 package contours;
2 import java.awt.Point;
3 import java.util.ArrayList;
4 import java.util.LinkedList;
5 import java.util.List;
6 import regions.BinaryRegion;
7 import ij.IJ;
8 import ij.process.ImageProcessor;
9
```

```
10 public class ContourTracer {
11   static final byte FOREGROUND = 1;
12   static final byte BACKGROUND = 0;
13   static boolean beVerbose = true;
14
15   List<Contour> outerContours = null;
16   List<Contour> innerContours = null;
17   List<BinaryRegion> allRegions = null;
18   int regionId = 0;
19
20   ImageProcessor ip = null;
21   int width;
22   int height;
23   byte[][] pixelArray;
24   int[][] labelArray;
25
26   // label values in labelArray can be:
27   //   0 ... unlabeled
28   // -1 ... previously visited background pixel
29   // > 0 ... a valid label
30
31   // constructor method
32   public ContourTracer (ImageProcessor ip) {
33     this.ip = ip;
34     this.width = ip.getWidth();
35     this.height = ip.getHeight();
36     makeAuxArrays();
37     findAllContours();
38     collectRegions();
39   }
40
41   public static void setVerbose(boolean verbose) {
42     beVerbose = verbose;
43   }
44
45   public List<Contour> getOuterContours() {
46     return outerContours;
47   }
48
49   public List<Contour> getInnerContours() {
50     return innerContours;
51   }
52
53   public List<BinaryRegion> getRegions() {
54     return allRegions;
55   }
56
57   // nonpublic methods
58
59   void makeAuxArrays() {
60     int h = ip.getHeight();
```

```
61    int w = ip.getWidth();
62    pixelArray = new byte[h+2][w+2];
63    labelArray = new int[h+2][w+2];
64    // initialize auxiliary arrays
65    for (int v = 0; v < h+2; v++) {
66      for (int u = 0; u < w+2; u++) {
67        if (ip.get(u-1,v-1) == 0)
68          pixelArray[v][u] = BACKGROUND;
69        else
70          pixelArray[v][u] = FOREGROUND;
71      }
72    }
73  }
74
75  Contour traceOuterContour (int cx, int cy, int label) {
76    Contour cont = new Contour(label);
77    traceContour(cx, cy, label, 0, cont);
78    return cont;
79  }
80
81  Contour traceInnerContour(int cx, int cy, int label) {
82    Contour cont = new Contour(label);
83    traceContour(cx, cy, label, 1, cont);
84    return cont;
85  }
86
87  // trace one contour starting at (xS,yS) in direction dS
88  Contour traceContour (int xS, int yS, int label, int dS,
        Contour cont) {
89    int xT, yT; // T = successor of starting point (xS,yS)
90    int xP, yP; // P = previous contour point
91    int xC, yC; // C = current contour point
92    Point pt = new Point(xS, yS);
93    int dNext = findNextPoint(pt, dS);
94    cont.addPoint(pt);
95    xP = xS; yP = yS;
96    xC = xT = pt.x;
97    yC = yT = pt.y;
98
99    boolean done = (xS==xT && yS==yT); // true if isolated pixel
100
101   while (!done) {
102     labelArray[yC][xC] = label;
103     pt = new Point(xC, yC);
104     int dSearch = (dNext + 6) % 8;
105     dNext = findNextPoint(pt, dSearch);
106     xP = xC; yP = yC;
107     xC = pt.x; yC = pt.y;
108     // are we back at the starting position?
109     done = (xP==xS && yP==yS && xC==xT && yC==yT);
110     if (!done) {
```

```
111         cont.addPoint(pt);
112       }
113     }
114     return cont;
115   }
116
117   int findNextPoint (Point pt, int dir) {
118     // starts at Point pt in direction dir, returns the
119     // final tracing direction, and modifies pt
120     final int[][] delta = {
121       { 1,0}, { 1, 1}, {0, 1}, {-1, 1},
122       {-1,0}, {-1,-1}, {0,-1}, { 1,-1}};
123     for (int i = 0; i < 7; i++) {
124       int x = pt.x + delta[dir][0];
125       int y = pt.y + delta[dir][1];
126       if (pixelArray[y][x] == BACKGROUND) {
127         // mark surrounding background pixels
128         labelArray[y][x] = -1;
129         dir = (dir + 1) % 8;
130       }
131       else { // found a nonbackground pixel
132         pt.x = x; pt.y = y;
133         break;
134       }
135     }
136     return dir;
137   }
138
139   void findAllContours() {
140     outerContours = new ArrayList<Contour>(50);
141     innerContours = new ArrayList<Contour>(50);
142     int label = 0;   // current label
143
144     // scan top to bottom, left to right
145     for (int v = 1; v < pixelArray.length-1; v++) {
146       label = 0; // no label
147       for (int u = 1; u < pixelArray[v].length-1; u++) {
148
149         if (pixelArray[v][u] == FOREGROUND) {
150           if (label != 0) { // keep using the same label
151             labelArray[v][u] = label;
152           }
153           else {
154             label = labelArray[v][u];
155             if (label == 0) {
156               // unlabeled—new outer contour
157               regionId = regionId + 1;
158               label = regionId;
159               Contour oc = traceOuterContour(u, v, label);
160               outerContours.add(oc);
161               labelArray[v][u] = label;
```

```
          }
        }
      }
      else { // background pixel
        if (label != 0) {
          if (labelArray[v][u] == 0) {
            // unlabeled—new inner contour
            Contour ic = traceInnerContour(u-1, v, label);
            innerContours.add(ic);
          }
          label = 0;
        }
      }
    }
  }
  // shift back to original coordinates
  Contour.moveContoursBy (outerContours, -1, -1);
  Contour.moveContoursBy (innerContours, -1, -1);
}


// creates a container of BinaryRegion objects
// collects the region pixels from the label image
// and computes the statistics for each region
void collectRegions() {
  int maxLabel = this.regionId;
  int startLabel = 1;
  BinaryRegion[] regionArray =
       new BinaryRegion[maxLabel + 1];
  for (int i = startLabel; i <= maxLabel; i++) {
    regionArray[i] = new BinaryRegion(i);
  }
  for (int v = 0; v < height; v++) {
    for (int u = 0; u < width; u++) {
      int lb = labelArray[v][u];
      if (lb >= startLabel && lb <= maxLabel
          && regionArray[lb]!=null) {
        regionArray[lb].addPixel(u, v);
      }
    }
  }

  // create a list of regions to return, collect nonempty regions
  List<BinaryRegion> regionList =
           new LinkedList<BinaryRegion>();
    for (BinaryRegion r: regionArray) {
      if (r != null && r.getSize()>0) {
        r.update(); // compute the statistics for this region
        regionList.add(r);
      }
    }
```

```
213       allRegions = regionList;
214     }
215
216 } // end of class ContourTracer
```

D.2.5 ContourOverlay (Class)

```
 1 package contours;
 2 import ij.ImagePlus;
 3 import ij.gui.ImageCanvas;
 4 import java.awt.BasicStroke;
 5 import java.awt.Color;
 6 import java.awt.Graphics;
 7 import java.awt.Graphics2D;
 8 import java.awt.Polygon;
 9 import java.awt.RenderingHints;
10 import java.awt.Shape;
11 import java.awt.Stroke;
12 import java.util.List;
13
14 public class ContourOverlay extends ImageCanvas {
15   private static final long serialVersionUID = 1L;
16   static float strokeWidth = 0.5f;
17   static int capsstyle = BasicStroke.CAP_ROUND;
18   static int joinstyle = BasicStroke.JOIN_ROUND;
19   static Color outerColor = Color.black;
20   static Color innerColor = Color.white;
21   static float[] outerDashing = {strokeWidth * 2.0f,
         strokeWidth * 2.5f};
22   static float[] innerDashing = {strokeWidth * 0.5f,
         strokeWidth * 2.5f};
23   static boolean DRAW_CONTOURS = true;
24
25   Shape[] outerContourShapes = null;
26   Shape[] innerContourShapes = null;
27
28   public ContourOverlay(ImagePlus im,
29         List<Contour> outerCs, List<Contour> innerCs)
30   {
31     super(im);
32     if (outerCs != null)
33       outerContourShapes = Contour.makePolygons(outerCs);
34     if (innerCs != null)
35       innerContourShapes = Contour.makePolygons(innerCs);
36   }
37
38   public void paint(Graphics g) {
39     super.paint(g);
40     drawContours(g);
41   }
```

```
42
43   // nonpublic methods
44
45   private void drawContours(Graphics g) {
46     Graphics2D g2d = (Graphics2D) g;
47     g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
           RenderingHints.VALUE_ANTIALIAS_ON);
48
49     // scale and move overlay to the pixel centers
50     double mag = this.getMagnification();
51     g2d.scale(mag, mag);
52     g2d.translate(0.5-this.srcRect.x, 0.5-this.srcRect.y);
53
54     if (DRAW_CONTOURS) {
55       Stroke solidStroke = new BasicStroke
56           (strokeWidth, capsstyle, joinstyle);
57       Stroke dashedStrokeOuter = new BasicStroke
58           (strokeWidth, capsstyle, joinstyle, 1.0f,
               outerDashing, 0.0f);
59       Stroke dashedStrokeInner = new BasicStroke
60           (strokeWidth, capsstyle, joinstyle, 1.0f,
               innerDashing, 0.0f);
61
62       if (outerContourShapes != null)
63         drawShapes(outerContourShapes, g2d, solidStroke,
               dashedStrokeOuter, outerColor);
64       if (innerContourShapes != null)
65         drawShapes(innerContourShapes, g2d, solidStroke,
               dashedStrokeInner, innerColor);
66     }
67   }
68
69   void drawShapes(Shape[] shapes, Graphics2D g2d,
70         Stroke solidStrk, Stroke dashedStrk, Color col) {
71     g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
           RenderingHints.VALUE_ANTIALIAS_ON);
72     g2d.setColor(col);
73     for (int i = 0; i < shapes.length; i++) {
74       Shape s = shapes[i];
75       if (s instanceof Polygon)
76         g2d.setStroke(dashedStrk);
77       else
78         g2d.setStroke(solidStrk);
79       g2d.draw(s);
80     }
81   }
82
83 } // end of class ContourOverlay
```

References

1. Adobe Systems. "Adobe RGB (1998) Color Space Specification" (2005). http://www.adobe.com/digitalimag/pdfs/AdobeRGB1998.pdf.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. "The Design and Analysis of Computer Algorithms". Addison-Wesley, Reading, MA (1974).
3. K. Arnold, J. Gosling, and D. Holmes. "The Java Programming Language". Addison-Wesley, Reading, MA, fourth ed. (2005).
4. W. Bailer. "Writing ImageJ Plugins—A Tutorial" (2003). http://www.fh-hagenberg.at/mtd/depot/imaging/imagej/.
5. D. H. Ballard and C. M. Brown. "Computer Vision". Prentice Hall, Englewood Cliffs, NJ (1982).
6. C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software* **22**(4), 469–483 (1996).
7. H. G. Barrow, J. M. Tenenbaum, R. C. Bolles, and H. C. Wolf. Parametric correspondence and chamfer matching: two new techniques for image matching. In R. Reddy, editor, "Proceedings of the 5th International Joint Conference on Artificial Intelligence", pp. 659–663, Cambridge, MA (1977). William Kaufmann, Los Altos, CA.
8. R. E. Blahut. "Fast Algorithms for Digital Signal Processing". Addison-Wesley, Reading, MA (1985).
9. J. Bloch. "Effective Java Programming Language Guide". Addison-Wesley, Reading, MA (2001).
10. C. Boor. "A Practical Guide to Splines". Springer-Verlag, New York (2001).
11. G. Borgefors. Distance transformations in digital images. *Computer Vision, Graphics and Image Processing* **34**, 344–371 (1986).
12. G. Borgefors. Hierarchical chamfer matching: a parametric edge matching algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **10**(6), 849–865 (1988).
13. J. E. Bresenham. A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM* **20**(2), 100–106 (1977).
14. E. O. Brigham. "The Fast Fourier Transform and Its Applications". Prentice Hall, Englewood Cliffs, NJ (1988).
15. I. N. Bronshtein and K. A. Semendyayev. "Handbook of Mathematics". Springer-Verlag, Berlin, third ed. (2007).
16. H. Bunke and P. S. P. Wang, editors. "Handbook of Character Recognition and Document Image Analysis". World Scientific, Singapore (2000).
17. W. Burger and M. J. Burge. "Digitale Bildverarbeitung—Eine Einführung mit Java und ImageJ". Springer-Verlag, Berlin, second ed. (2006).

18. P. J. Burt and E. H. Adelson. The Laplacian pyramid as a compact image code. *IEEE Transactions on Communications* **31**(4), 532–540 (1983).
19. J. F. Canny. A computational approach to edge detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence* **8**(6), 679–698 (1986).
20. K. R. Castleman. "Digital Image Processing". Prentice Hall, Upper Saddle River, NJ (1995).
21. E. Catmull and R. Rom. A class of local interpolating splines. In R. E. Barnhill and R. F. Riesenfeld, editors, "Computer Aided Geometric Design", pp. 317–326. Academic Press, New York (1974).
22. F. Chang and C. J. Chen. A component-labeling algorithm using contour tracing technique. In "Proceedings of the Seventh International Conference on Document Analysis and Recognition ICDAR2003", pp. 741–745, Edinburgh (2003). IEEE Computer Society, Los Alamitos, CA.
23. F. Chang, C. J. Chen, and C. J. Lu. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision, Graphics, and Image Processing: Image Understanding* **93**(2), 206–220 (February 2004).
24. P. R. Cohen and E. A. Feigenbaum. "The Handbook of Artificial Intelligence". William Kaufmann, Los Altos, CA (1982).
25. T. H. Corman, C. E. Leiserson, R. L. Rivest, and C. Stein. "Introduction to Algorithms". MIT Press, Cambridge, MA, second ed. (2001).
26. L. S. Davis. A survey of edge detection techniques. *Computer Graphics and Image Processing* **4**, 248–270 (1975).
27. R. O. Duda, P. E. Hart, and D. G. Stork. "Pattern Classification". Wiley, New York (2001).
28. B. Eckel. "Thinking in Java". Prentice Hall, Englewood Cliffs, NJ, fourth ed. (2006). Earlier versions available online.
29. N. Efford. "Digital Image Processing—A Practical Introduction Using Java". Pearson Education, Upper Saddle River, NJ (2000).
30. D. Flanagan. "Java in a Nutshell". O'Reilly, Sebastopol, CA, fifth ed. (2005).
31. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. "Computer Graphics: Principles and Practice". Addison-Wesley, Reading, MA, second ed. (1996).
32. A. Ford and A. Roberts. "Colour Space Conversions" (1998). http://www.poynton.com/PDFs/coloureq.pdf.
33. W. Förstner and E. Gülch. A fast operator for detection and precise location of distinct points, corners and centres of circular features. In A. Grün and H. Beyer, editors, "Proceedings, International Society for Photogrammetry and Remote Sensing Intercommission Conference on the Fast Processing of Photogrammetric Data", pp. 281–305, Interlaken (June 1987).
34. D. A. Forsyth and J. Ponce. "Computer Vision—A Modern Approach". Prentice Hall, Englewood Cliffs, NJ (2003).
35. H. Freeman. Computer processing of line drawing images. *ACM Computing Surveys* **6**(1), 57–97 (March 1974).
36. M. Gervautz and W. Purgathofer. A simple method for color quantization: octree quantization. In A. Glassner, editor, "Graphics Gems I", pp. 287–293. Academic Press, New York (1990).

37. A. S. Glassner. "Principles of Digital Image Synthesis". Morgan Kaufmann Publishers, San Francisco (1995).
38. R. C. Gonzalez and R. E. Woods. "Digital Image Processing". Addison-Wesley, Reading, MA (1992).
39. R. L. Graham, D. E. Knuth, and O. Patashnik. "Concrete Mathematics: A Foundation for Computer Science". Addison-Wesley, Reading, MA, second ed. (1994).
40. P. Green. Colorimetry and colour differences. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 3, pp. 40–77. Wiley, New York (2002).
41. E. L. Hall. "Computer Image Processing and Recognition". Academic Press, New York (1979).
42. C. G. Harris and M. Stephens. A combined corner and edge detector. In C. J. Taylor, editor, "4th Alvey Vision Conference", pp. 147–151, Manchester (1988).
43. P. S. Heckbert. Color image quantization for frame buffer display. *Computer Graphics* **16**(3), 297–307 (1982).
44. P. S. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, University of California, Berkeley, Dept. of Electrical Engineering and Computer Science (1989). http://www.cs.cmu.edu/~ph/#papers.
45. J. Holm, I. Tastl, L. Hanlon, and P. Hubel. Color processing for digital photography. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 9, pp. 179–220. Wiley, New York (2002).
46. B. K. P. Horn. "Robot Vision". MIT-Press, Cambridge, MA (1982).
47. P. V. C. Hough. Method and means for recognizing complex patterns. US Patent 3,069,654 (1962).
48. M. K. Hu. Visual pattern recognition by moment invariants. *IEEE Transactions on Information Theory* **8**, 179–187 (1962).
49. R. W. G. Hunt. "The Reproduction of Colour". Wiley, New York, sixth ed. (2004).
50. J. Illingworth and J. Kittler. A survey of the Hough transform. *Computer Vision, Graphics and Image Processing* **44**, 87–116 (1988).
51. International Color Consortium. "Specification ICC.1:2004-10 (Profile Version 4.2.0.0): Image Technology Colour Management—Architecture, Profile Format, and Data Structure" (2004). http://www.color.org/documents/ICC1v42_2006-05.pdf.
52. International Electrotechnical Commission, IEC, Geneva. "IEC 61966-2-1: Multimedia Systems and Equipment—Colour Measurement and Management, Part 2-1: Colour Management—Default RGB Colour Space—sRGB" (1999). http://www.iec.ch.
53. International Organization for Standardization, ISO, Geneva. "ISO 13655:1996, Graphic Technology—Spectral Measurement and Colorimetric Computation for Graphic Arts Images" (1996).
54. International Organization for Standardization, ISO, Geneva. "ISO 15076-1:2005, Image Technology Colour Management—Architecture, Profile Format, and Data Structure: Part 1" (2005). Based on ICC.1:2004-10.
55. International Telecommunications Union, ITU, Geneva. "ITU-R Recommendation BT.709-3: Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange" (1998).

56. International Telecommunications Union, ITU, Geneva. "ITU-R Recommendation BT.601-5: Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios" (1999).
57. K. Jack. "Video Demystified—A Handbook for the Digital Engineer". LLH Publishing, Eagle Rock, VA, third ed. (2001).
58. B. Jähne. "Practical Handbook on Image Processing for Scientific Applications". CRC Press, Boca Raton, FL (1997).
59. B. Jähne. "Digitale Bildverarbeitung". Springer-Verlag, Berlin, fifth ed. (2002).
60. A. K. Jain. "Fundamentals of Digital Image Processing". Prentice Hall, Englewood Cliffs, NJ (1989).
61. X. Y. Jiang and H. Bunke. Simple and fast computation of moments. *Pattern Recognition* **24**(8), 801–806 (1991).
62. J. King. Engineering color at Adobe. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 15, pp. 341–369. Wiley, New York (2002).
63. R. A. Kirsch. Computer determination of the constituent structure of biological images. *Computers in Biomedical Research* **4**, 315–328 (1971).
64. L. Kitchen and A. Rosenfeld. Gray-level corner detection. *Pattern Recognition Letters* **1**, 95–102 (1982).
65. T. Lindeberg. Feature detection with automatic scale selection. *International Journal of Computer Vision* **30**(2), 77–116 (1998).
66. D. G. Lowe. Object recognition from local scale-invariant features. In "Proceedings of the 7th IEEE International Conference on Computer Vision ICCV'99", vol. 2, pp. 1150–1157, Kerkyra, Corfu, Greece (1999). IEEE Computer Society, Los Alamitos, CA.
67. B. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In P. J. Hayes, editor, "Proceedings of the 7th International Joint Conference on Artificial Intelligence IJCAI'81", pp. 674–679, Vancouver, BC (1981). William Kaufmann, Los Altos, CA.
68. S. Mallat. "A Wavelet Tour of Signal Processing". Academic Press, New York (1999).
69. D. Marr and E. Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London, Series B* **207**, 187–217 (1980).
70. E. H. W. Meijering, W. J. Niessen, and M. A. Viergever. Quantitative evaluation of convolution-based methods for medical image interpolation. *Medical Image Analysis* **5**(2), 111–126 (2001). http://imagescience.bigr.nl/meijering/software/transformj/.
71. J. Miano. "Compressed Image File Formats". ACM Press, Addison-Wesley, Reading, MA (1999).
72. D. P. Mitchell and A. N. Netravali. Reconstruction filters in computer-graphics. In R. J. Beach, editor, "Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'88", pp. 221–228, Atlanta, GA (1988). ACM Press, New York.
73. P. A. Mlsna and J. J. Rodriguez. Gradient and laplacian-type edge detection. In A. Bovik, editor, "Handbook of Image and Video Processing", pp. 415–431. Academic Press, New York (2000).
74. J. D. Murray and W. VanRyper. "Encyclopedia of Graphics File Formats". O'Reilly, Sebastopol, CA, second ed. (1996).

75. M. Nadler and E. P. Smith. "Pattern Recognition Engineering". Wiley, New York (1993).
76. A. V. Oppenheim, R. W. Shafer, and J. R. Buck. "Discrete-Time Signal Processing". Prentice Hall, Englewood Cliffs, NJ, second ed. (1999).
77. T. Pavlidis. "Algorithms for Graphics and Image Processing". Computer Science Press / Springer-Verlag, New York (1982).
78. C. A. Poynton. "Digital Video and HDTV Algorithms and Interfaces". Morgan Kaufmann Publishers, San Francisco (2003).
79. W. S. Rasband. "ImageJ". U.S. National Institutes of Health, MD (1997–2007). http://rsb.info.nih.gov/ij/.
80. C. E. Reid and T. B. Passin. "Signal Processing in C". Wiley, New York (1992).
81. D. Rich. Instruments and methods for colour measurement. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 2, pp. 19–48. Wiley, New York (2002).
82. I. E. G. Richardson. "H.264 and MPEG-4 Video Compression". Wiley, New York (2003).
83. L. G. Roberts. Machine perception of three-dimensional solids. In J. T. Tippet, editor, "Optical and Electro-Optical Information Processing", pp. 159–197. MIT Press, Cambridge, MA (1965).
84. A. Rosenfeld and P. Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM* **12**, 471–494 (1966).
85. J. C. Russ. "The Image Processing Handbook". CRC Press, Boca Raton, FL, third ed. (1998).
86. C. Schmid and R. Mohr. Local grayvalue invariants for image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **19**(5), 530–535 (May 1997).
87. C. Schmid, R. Mohr, and C. Bauckhage. Evaluation of interest point detectors. *International Journal of Computer Vision* **37**(2), 151–172 (2000).
88. Y. Schwarzer, editor. "Die Farbenlehre Goethes". Westerweide Verlag, Witten (2004).
89. M. Seul, L. O'Gorman, and M. J. Sammon. "Practical Algorithms for Image Analysis". Cambridge University Press, Cambridge (2000).
90. L. G. Shapiro and G. C. Stockman. "Computer Vision". Prentice Hall, Englewood Cliffs, NJ (2001).
91. G. Sharma and H. J. Trussell. Digital color imaging. *IEEE Transactions on Image Processing* **6**(7), 901–932 (1997).
92. N. Silvestrini and E. P. Fischer. "Farbsysteme in Kunst und Wissenschaft". DuMont, Cologne (1998).
93. Y. Sirisathitkul, S. Auwatanamongkol, and B. Uyyanonvara. Color image quantization using distances between adjacent colors along the color axis with highest color variance. *Pattern Recognition Letters* **25**, 1025–1043 (2004).
94. S. M. Smith and J. M. Brady. SUSAN—a new approach to low level image processing. *International Journal of Computer Vision* **23**(1), 45–78 (1997).
95. M. Sonka, V. Hlavac, and R. Boyle. "Image Processing, Analysis and Machine Vision". PWS Publishing, Pacific Grove, CA, second ed. (1999).

96. M. Stokes and M. Anderson. "A Standard Default Color Space for the Internet—sRGB". Hewlett-Packard, Microsoft, www.w3.org/Graphics/Color/sRGB.html (1996).
97. S. Süsstrunk. Managing color in digital image libraries. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 17, pp. 385–419. Wiley, New York (2002).
98. S. Theodoridis and K. Koutroumbas. "Pattern Recognition". Academic Press, New York (1999).
99. E. Trucco and A. Verri. "Introductory Techniques for 3-D Computer Vision". Prentice Hall, Englewood Cliffs, NJ (1998).
100. K. Turkowski. Filters for common resampling tasks. In A. Glassner, editor, "Graphics Gems I", pp. 147–165. Academic Press, New York (1990).
101. T. Tuytelaars and L. J. Van Gool. Matching widely separated views based on affine invariant regions. *International Journal of Computer Vision* **59**(1), 61–85 (August 2004).
102. D. Wallner. Color management and transformation through ICC profiles. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 11, pp. 247–261. Wiley, New York (2002).
103. A. Watt. "3D Computer Graphics". Addison-Wesley, Reading, MA, third ed. (1999).
104. A. Watt and F. Policarpo. "The Computer Image". Addison-Wesley, Reading, MA (1999).
105. G. Wolberg. "Digital Image Warping". IEEE Computer Society Press, Los Alamitos, CA (1990).
106. G. Wyszecki and W. S. Stiles. "Color Science: Concepts and Methods, Quantitative Data and Formulae". Wiley–Interscience, New York, second ed. (2000).
107. T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Communications of the ACM* **27**(3), 236–239 (1984).
108. S. Zokai and G. Wolberg. Image registration using log-polar mappings for recovery of large-scale similarity and projective transformations. *IEEE Transactions on Image Processing* **14**(10), 1422–1434 (October 2005).

Index

Symbols

∗ (convolution operator), 99, 452
⊛ (correlation operator), 432, 452
⊕ (dilation operator), 177, 452
⊖ (erosion operator), 178, 452
`?` (operator), 267
⌊ ⌋, 47, 61, 452
∂, 119, 140, 452
∇, 119, 130, 452
`&` (operator), 244, 304, 460, 491
`&&` (operator), 150, 162, 203, 514
`|` (operator), 245, 304, 520
`||` (operator), 514
`~` (operator), 514
`»` (operator), 245, 304
`«` (operator), 245
`%` (operator), 459

A

`abs` (method), 81, 461, 495
absolute value, 495
accessing pixels, 485
accumulator array, 159
achromatic, 260
`acos` (method), 461
`ADD` (constant), 82, 85, 496
`add` (method), 81, 495, 529, 530, 534, 540
`addChoice` (method), 86, 507
`addNumericField` (method), 86, 507, 518, 526
`addSlice` (method), 505
`addStringField` (method), 518
adjoint matrix, 382
Adobe
  Illustrator, 15
  Photoshop, 59, 94, 115, 134
  RGB, 288
affine mapping, 378, 386
`AffineMapping` (class), 417, 418
`AffineTransform` (class), 413
aliasing, 329, 334, 337, 339, 350, 410
alpha
  blending, 83, 85, 506
  channel, 17, 244
  value, 83, 244
`altKeyDown` (method), 517
ambient lighting, 280
amplitude, 314–316
`AND` (constant), 82, 496
`and` (method), 495
angular frequency, 314, 315, 334, 338, 344
anti-aliasing, 500
`apply` (method), 297
`applyTable` (method), 68, 77, 80, 152, 495, 530
`applyTo` (method), 415, 416, 421, 423, 426, 427
approximation, 401, 402
ArcTan function, 122, 350, 388, 452, 462
area
  polygon, 224
  region, 224
arithmetic operation, 81, 82, 495
array, 462–467
  accessing elements, 463
  creation, 462
  duplication, 465
  size, 463
  sorting, 466
  two-dimensional, 464
`ArrayList` (class), 149, 468, 534, 539
`Arrays` (class), 300, 466
`Arrays.sort` (method), 466
`asin` (method), 461
associativity, 101, 178
`atan` (method), 461
`atan2` (method), 452, 461, 462
auto-contrast, 57
  modified, 58

autoThreshold (method), 495
AVERAGE (constant), 82, 496
AWT, 244, 294

B

background, 173
bandwidth, 330
Bartlett window, 355, 357, 358
BasicStroke (class), 541, 542
basis function, 333–337, 344, 349, 367, 368, 373
beep (method), 517
bias problem, 165
bicubic interpolation, 407
BicubicInterpolator (class), 425, 426
big endian, 22–24
bilinear
  interpolation, 405, 485
  mapping, 385, 386
BilinearInterpolator (class), 424, 426
BilinearMapping (class), 421
binarization, 55
binary
  image, 13, 129, 173, 189, 199, 472
  morphology, 173–186
BinaryProcessor (class), 55, 195, 472
binnedHistogram (method), 48
binning, 46–48, 51
bit
  mask, 244
  operation, 246
bitmap image, 13, 218
bitwise AND operator, 460
black (constant), 541
black box, 99
black-generation function, 273
Blitter (interface), 82, 85
BMP, 20, 23, 246
bounding box, 225
box filter, 92, 102, 121, 497
Bradford model, 289, 292
BradfordAdaptation (class), 297
breadth-first, 202
Bresenham algorithm, 168
brightness, 54, 490
byte, 22
byte (type), 459, 472
ByteBlitter (class), 508
ByteProcessor (class), 82, 252, 253, 472
ByteStatistics (class), 494

C

C2-continuous, 400
camera obscura, 7
Canny edge operator, 127, 129
Canvas (class), 474
card, 38, 452, 453
cardinal spline, 398, 400
cardinality, 452, 453
Catmull-Rom interpolation, 400
CCITT, 15
Cdf (method), 70
cdf, *see* cumulative distribution function
ceil (method), 461
CENTER_JUSTIFY (constant), 500
centralMoment (method), 229
centroid, 226
CGM format, 15
chain code, 219, 224
chamfer
  algorithm, 443
  matching, 446
chroma, 270
chromatic adaptation, 288
  Bradford model, 289, 292
  XYZ scaling, 289
ChromaticAdaptation (class), 297
CIE, 276
  chromaticity diagram, 277, 280
  L*a*b*, 275, 281, 282
  standard illuminant, 279
  XYZ, 276, 282, 285, 286, 295, 298
circle, 167, 380
circularity, 224
circumference, 223
city block distance, 443
clamping, 54, 92
clipboard, 522
clone (method), 299, 300, 416, 465
Cloneable (interface), 465
cloning arrays, 465
close (method), 193, 194
closeAllWindows (method), 521
closing, 185, 188, 193
clutter, 447
CMYK, 271–275
collectCorners (method), 149

Collections (class), 150
collision, 207
Color (class), 261–263, 294, 541, 542
color
  count, 299
  difference, 283
  image, 13, 239–312, 473
  keying, 266
  management, 296
  pixel, 242, 244, 245
  saturation, 257
  table, 243, 248, 250, 311
  temperature, 279
color quantization, 43, 244, 250, 254, 301–310
  3:3:2, 303
  median-cut, 305
  octree, 306
  populosity, 305
color space, 253–299
  CMYK, 271
  colorimetric, 275–299
  HLS, 258
  HSB, 258, 295
  HSV, 258, 295
  in Java, 292–299
  L*a*b*, 281
  RGB, 240
  sRGB, 283
  XYZ, 276
  YC_bC_r, 270
  YIQ, 269
  YUV, 268
color system
  additive, 239
  subtractive, 272
COLOR_RGB (constant), 248
ColorChooser (class), 474
ColorModel (class), 250, 294, 502, 503
ColorProcessor (class), 194, 245, 247, 251, 253, 257, 300, 473, 485
ColorSpace (class), 293–295, 297
ColorStatistics (class), 494
comb function, 327
commutativity, 100, 178, 179
compactness, 224
Comparable (interface), 467
compareTo (method), 150, 467, 527
comparing images, 429–450
complementary set, 176
Complex (class), 341
complex number, 316, 453
complexity, 454
component
  histogram, 48
  ordering, 242
computeMatch (method), 438, 441
computer
  graphics, 2
  vision, 3
concat (method), 417
concolve (method), 137
Concolver (class), 137
conic section, 380
connected components problem, 207
container, 149
contains (method), 511
contour, 127, 209–216
ContourOverlay (class), 216
contrast, 40, 54
  automatic adjustment, 57
convertHSBToRGB (method), 254, 493
convertRGBStackToRGB (method), 493
convertRGBtoIndexedColor (method), 254, 494
convertToByte (method), 85, 137, 196, 253, 257, 485, 492, 508
convertToFloat (method), 137, 253, 492, 528
convertToGray16 (method), 254, 493
convertToGray32 (method), 254, 493
convertToGray8 (method), 254, 493
convertToHSB (method), 254, 493
convertToRGB (method), 253, 254, 492, 493
convertToRGBStack (method), 494
convertToShort (method), 253, 492
convex hull, 225, 236
convexity, 225, 234
convolution, 99, 364, 433, 455
  property, 324, 363
convolve (method), 114, 148, 497
convolve3x3 (method), 497
Convolver (class), 114, 148, 531
coordinate
  Cartesian, 378
  homogeneous, 377, 416
COPY (constant), 496
copy (method), 522

COPY_INVERTED (constant), 496
copyBits (method), 82, 85, 137, 193, 194, 496, 508, 531
Corner (class), 148, 150
corner, 139
  detection, 139–153
  point, 153
  response function, 141, 145
  strength, 141
CorrCoeffMatcher (class), 438, 440
correlation, 99, 364, 432
  coefficient, 434
cos (method), 461
cosine function, 322
  one-dimensional, 314
  two-dimensional, 346, 347
cosine transform, 18, 367
cosine^2 window, 357, 358
countColors (method), 300
counting colors, 299
createByteImage (method), 477, 508, 518
createEmptyStack (method), 476, 504
createFloatImage (method), 477
createImage (method), 477, 478
createProcessor (method), 192, 478
createRGBImage (method), 478
createShortImage (method), 477
creating
  image processors, 478–480
  new images, 52, 476–478
crop (method), 497
cross correlation, 432–435
CRT, 240
CS_CIEXYZ (constant), 295
CS_GRAY (constant), 295
CS_LINEAR_RGB (constant), 295
CS_PYCC (constant), 295
CS_sRGB (constant), 295
CS_sRGBt (constant), 297
cubic
  B-spline interpolation, 401
  interpolation, 397
  spline, 400
cumulative
  distribution function, 64
  histogram, 50, 58, 62, 64
currentMemory (method), 523
cycle length, 314

D

D50, 279, 280, 295
D65, 280, 282, 284
DCT, 367–373
  one-dimensional, 367, 368
  two-dimensional, 370
DCT (method), 370
debugging, 112
deconvolution, 365
deleteLastSlice (method), 505
deleteSlice (method), 505
delta function, 325
depth-first, 202
derivative
  estimation, 119
  first, 118, 144
  partial, 119
  second, 126, 130
desaturation, 257
determinant, 382
DFT, 332–366, 452
  one-dimensional, 332–341
  two-dimensional, 343–366
DFT (method), 341
diameter, 226
DICOM, 31
DIFFERENCE (constant), 82, 194, 496
difference filter, 98
digital images, 8
dilate (method), 191, 193, 194, 497
dilation, 177, 187, 191, 497
Dirac function, 103, 179, 320, 325
DirectColorModel (class), 502
directory information, 523
discrete
  cosine transform, 367–373
  Fourier transform, 332–366, 452
  sine transform, 367
displaying images, 500, 503
distance, 151, 431
  city block, 443
  Euclidean, 432, 443
  Manhattan, 443
  mask, 443
  maximum difference, 431
  sum of differences, 431
  sum of squared differences, 432
  transform, 442
DIVIDE (constant), 82, 496
DOES_16 (constant), 519
DOES_32 (constant), 519

DOES_8C (constant), 248, 249, 251, 519
DOES_8G (constant), 32, 45, 519
DOES_ALL (constant), 493, 519
DOES_RGB (constant), 246, 247, 519
DOES_STACKS (constant), 519
DONE (constant), 493, 503, 519
dots per inch (dpi), 11, 339
Double (class), 462
double (type), 92, 458
dpi, 339
draw (method), 152, 500, 501, 542
drawDot (method), 499, 500
drawLine (method), 152, 499, 527
drawOval (method), 499
drawPixel (method), 499
drawPolygon (method), 499
drawRect (method), 499
drawString (method), 499, 500
DST, 367
duplicate (method), 85, 93, 94, 109, 137, 152, 194, 416, 478, 508, 529, 533
duplicateArray (method), 466
DXF format, 15
dynamic range, 40

E

E (constant), 461
eccentricity, 231, 236
Eclipse, 34, 470
edge
  filter, 497
  map, 129, 155
  sharpening, 130–137
  strength, 141
edge operator, 120–127
  Canny, 127, 129
  compass, 123
  in ImageJ, 125
  Kirsch, 123
  LoG, 126, 129
  Prewitt, 120, 129
  Roberts, 123, 129
  Sobel, 120, 125, 129
effective gamma value, 79
eigenvalue, 141, 231
eigenvector, 141
ellipse, 170, 232, 380, 499
Ellipse2D (class), 534
elliptical window, 356
elongatedness, 231
EMF format, 15
Encapsulated PostScript (EPS), 15
erode (method), 193, 194, 497
erosion, 178, 187, 191, 497
error (method), 515
escapePressed (method), 517
Euclidean distance, 151, 438, 443
Euler number, 235
Euler's notation, 316
executing plugins, 520
EXIF, 19, 284
exp (method), 461
exposure, 40

F

fast Fourier transform, 342, 345, 359, 364, 455
fax encoding, 219
feature, 222
FFT, *see* fast Fourier transform, *see* fast Fourier transform
file format, 24
  BMP, 20
  EXIF, 19
  GIF, 15
  JFIF, 19
  JPEG-2000, 19
  magic number, 23
  PBM, 21
  Photoshop, 23
  PNG, 16
  RAS, 22
  RGB, 22
  TGA, 22
  TIFF, 15
  XBM/XPM, 22
file information, 522
FileInfo (class), 484, 522, 523
FileOpener (class), 484
FileSaver (class), 482
FileSaver (method), 482
fill (method), 52, 499
fillOval (method), 499
fillPolygon (method), 499
filter, 87–116, 497
  average, 497
  border handling, 91, 111
  box, 92, 97, 102, 121, 497
  color image, 136
  computation, 91

debugging, 112
derivative, 119
difference, 98
edge, 120–125, 497
efficiency, 111
Gaussian, 97, 102, 114, 133, 140, 144
ImageJ, 113–115
impulse response, 103
in frequency space, 363
indexed image, 248
inverse, 364
kernel, 99
Laplace, 98, 131, 136
Laplacian, 116
linear, 89–104, 113, 497
low-pass, 97
mask, 89
matrix, 89
maximum, 105, 115, 197, 497
median, 106, 115, 173, 497
minimum, 105, 115, 197, 497
morphological, 173–197
nonlinear, 104–111, 115
normalized, 93
separable, 101, 102, 131
smoothing, 92, 94, 96, 134, 497
unsharp masking, 133
weighted median, 107
`Find_Corners` (plugin), 153
`findCorners` (method), 152
`findEdges` (method), 125, 497
FITS, 31
`flipHorizontal` (method), 497
`flipVertical` (method), 498
`Float` (class), 462
`float` (type), 473
floating-point image, 13, 473
`FloatProcessor` (class), 253, 438, 473
`FloatStatistics` (class), 494
`floatToIntBits` (method), 486
flood filling, 200–204
`floor` (method), 461
floor function, 453
font, 500
for-loop, 488
foreground, 173
four-point mapping, 380
Fourier, 317
analysis, 318
coefficients, 318
descriptor, 221
integral, 318
series, 317
spectrum, 222, 319, 330
transform, 314–452
transform pair, 320, 322, 323
`Frame` (class), 474, 501
`FreehandRoi` (class), 475, 506
`freeMemory` (method), 523
frequency, 314, 338
angular, 314, 315, 334, 344
common, 315
directional, 348
distribution, 63
effective, 348, 349
fundamental, 317, 318, 339
maximum, 329, 350
space, 320, 338, 363
two-dimensional, 348
`fromCIEXYZ` (method), 292–294, 297
function
basis, 333–337, 344
cosine, 314
delta, 325
Dirac, 320, 325
impulse, 320, 325
periodic, 314
sine, 314
fundamental
frequency, 317, 318, 339
period, 338

G

`gamma` (method), 81, 495
gamma correction, 72–80, 256, 292, 295, 298, 495
applications, 75
inverse, 79
modified, 76–80, 285
gamut, 273, 280, 283, 288
Adobe RGB, 288
sRGB, 288
garbage, 463
Gaussian
area formula, 224
distribution, 51
filter, 97, 102, 114, 133, 140, 144
filter size, 102
function, 321, 323
separable, 102

window, 355, 356, 358
GaussKernel1d (class), 137
GenericDialog (class), 84, 86, 474, 507, 517, 518, 526
geometric operation, 375–428, 497
get (method), 34, 54, 63, 111, 259, 487, 490, 534, 538
get2dHistogram (method), 301
getBitDepth (method), 248
getBlues (method), 249, 251
getBrightness (method), 490
getClipboard (method), 522
getColorModel (method), 249–251, 502
getColumn (method), 489
getComponents (method), 295
getCurrentColorModel (method), 503
getCurrentImage (method), 249, 521
getCurrentSlice (method), 501
getCurrentWindow (method), 521
getDirectory (method), 523
getDoScaling (method), 494
getf (method), 440, 441, 487
getFileInfo (method), 522
getFloatArray (method), 490
getGreens (method), 249, 251
getHeight (method), 33, 93, 485, 491, 505
getHistogram (method), 45, 52, 63, 68, 299, 485, 494
getHistogramMax (method), 494
getHistogramMin (method), 494
getHistogramSize (method), 494
getHSB (method), 490
getID (method), 501
getIDList (method), 86, 507, 521
getImage (method), 86, 484, 501, 507, 517, 521
getImageArray (method), 505
getImageCount (method), 521
getImageStack (method), 504
getIntArray (method), 489
getInterpolate (method), 485, 498
getInterpolatedPixel (method), 424, 425, 498
getInterpolatedRGBPixel (method), 498
getInverse (method), 415
getLine (method), 485, 489
getMagnification (method), 542
getMapSize (method), 249–251
getMask (method), 510, 511, 514
getMaskArray (method), 512, 513
getMatchValue (method), 441
getMax (method), 503
getMin (method), 503
getNextChoiceIndex (method), 86, 507
getNextNumber (method), 86, 507, 518, 526
getNextString (method), 518
getNumber (method), 515
getOriginalFileInfo (method), 484, 523
getPixel (method), 33, 93, 109, 111, 245, 304, 460, 486, 490
getPixelCount (method), 487
getPixels (method), 463, 488, 491, 505, 529
getPixelsCopy (method), 489
getPixelSize (method), 249
getPixelValue (method), 485, 486
getProcessor (method), 85, 478, 484, 493, 501, 502, 505, 508
getProperties (method), 514
getProperty (method), 514, 516
getReds (method), 249, 251
getRGB (method), 490
getRoi (method), 509, 512, 514
getRow (method), 489
getShortSliceLabel (method), 505
getShortTitle (method), 86, 501, 507
getSize (method), 505
getSliceLabel (method), 505
getSliceLabels (method), 505
getStack (method), 476, 502, 504, 508
getStackSize (method), 504
getString (method), 515
getStringWidth (method), 499
getTempCurrentImage (method), 521
getTitle (method), 501, 526
getType (method), 248
getVersion (method), 523
getWeightingFactors (method), 257, 485
getWidth (method), 33, 93, 485, 491, 506
getWindow (method), 501
getWindowCount (method), 521

GIF, 15, 23, 31, 43, 219, 243, 248
gradient, 118, 119, 140, 144
graph, 207
Graphics (class), 541
graphics overlay, 216
Graphics2D (class), 542
grayscale
  conversion, 256, 287
  image, 12, 17, 472
  morphology, 187–188

H

Hadamard transform, 372
HandleExtraFileTypes (class), 481
Hanning window, 354, 355, 357, 358
Harris corner detector, 140
HarrisCornerDetector (class), 148, 153
HashSet (class), 468
hasNext (method), 534
HDTV, 271
Hertz, 315, 339
Hessian normal form, 159, 166
hexadecimal, 245, 460
hide (method), 501
hierarchical techniques, 127
histogram, 37–51, 299–301, 452, 494
  binning, 46
  channel, 48
  color image, 47
  component, 48
  computing, 44
  cumulative, 50, 58, 64
  equalization, 59
  matching, 67
  normalized, 63
  specification, 62–72
HLS, 258, 264–266, 270
HLStoRGB (method), 268
homogeneous
  coordinate, 377, 416
  point operation, 53, 60, 63
hot spot, 89, 176
Hough transform, 130, 156–171
  bias problem, 165
  edge strength, 167
  for circles, 167–169
  for ellipses, 170
  for lines, 156–167
  generalized, 170
  hierarchical, 167
HSB, *see* HSV
HSBtoRGB (method), 263, 264, 295
HSV, 254, 255, 258, 261, 266, 270, 295, 490
hue, 490
Huffman code, 18

I

i (imaginary unit), 316, 452, 453
ICC, 292
  profile, 296
ICC_ColorSpace (class), 295, 296
ICC_Profile (class), 296
iDCT (method), 370
idempotent, 186
IJ (class), 475, 477, 480, 523
ij (package), 471, 475
ij.gui (package), 474, 509
ij.io (package), 475
ij.plugin (package), 473
ij.plugin.filter (package), 473
ij.process (package), 472
Illuminant (class), 297
illuminant, 279
Image (class), 476, 478
image
  acquisition, 5
  analysis, 3
  binary, 13, 199, 472
  bitmap, 13
  color, 13, 473
  compression and histogram, 43
  coordinates, 11, 452
  creating new, 52
  defects, 42
  depth, 12
  digital, 8
  display, 52
  file, 480
  file format, 5, 13
  floating-point, 13, 473
  grayscale, 12, 17, 472
  height, 485
  indexed color, 13, 17, 472
  intensity, 12
  loading, 480
  locking, 522
  palette, 13
  parameter, 485
  plane, 7
  properties, 513–515

raster, 14
RGB, 473
size, 10
space, 100, 363
special, 13
statistics, 494
storing, 480
true color, 17
warping, 387
width, 485
ImageCanvas (class), 474, 541
ImageConverter (class), 253, 254, 492
ImageJ, 27–36
accessing pixels, 485
animation, 503
API, 471–475
directory, 523
displaying images, 500
filter, 113–115, 497
geometric operation, 413, 497
graphic operation, 499
GUI, 474
histogram, 494
image conversion, 492
installation, 469
loading images, 480
locking, 522
macro, 30, 34
main window, 30
memory, 523
open, 480
plugin, 31–35, 518
point operation, 80–86, 494
region of interest, 506
revert, 480
save, 480
snapshot, 35
stack, 30, 504, 505
storing images, 480
system information, 523
tutorial, 35
undo, 31, 35
Website, 35
ImagePlus (class), 153, 247, 251, 252, 471, 481, 509, 517, 522, 541
ImageProcessor (class), 32, 194, 246, 247, 249–253, 259, 463, 472, 485
ImageStack (class), 472, 505, 508, 512
ImageStatistics (class), 494
ImageWindow (class), 474, 533
impulse
function, 103, 320, 325
response, 103, 183
in place, 345
IndexColorModel (class), 249, 251, 252, 502
indexed color image, 13, 17, 243, 248, 254, 472
insert (method), 137, 489, 500, 508
instanceof (operator), 488, 491
intBitsToFloat (method), 486
Integer.MAX_VALUE (constant), 535
intensity
histogram, 48
image, 12
interest point, 139
interpolation, 392–410, 423–425, 485
B-spline, 400, 401
bicubic, 407, 409, 425
bilinear, 405, 409, 424, 498
by convolution, 397
Catmull-Rom, 399, 400, 425, 426
color, 498
cubic, 397
ideal, 393
kernel, 397
Lanczos, 402, 408, 428
linear, 397
Mitchell-Netravali, 400, 401, 428
nearest-neighbor, 397, 405, 409, 412, 424, 498
spline, 399
two-dimensional, 404–410
invalidNumber (method), 526
invariance, 224, 227, 228, 233, 234, 430
inverse
filter, 364
gamma function, 74
tangent function, 462
inversion, 55
invert (method), 55, 81, 193, 415, 417
invertLookupTable (class), 501
invertLut (method), 190, 501, 503
isInvertedLut (method), 501, 504
isJava14 (method), 523

isJava2 (method), 523
isLocalMax (method), 150
isMacintosh (method), 523
isMacOSX (method), 523
isNaN (method), 536
isotropic, 89, 119, 131, 133, 140, 153, 181
isWindows (method), 523
Iterator (class), 534
iterator (method), 534
ITU601, 271
ITU709, 75, 79, 256, 271, 284, 485

J

Jama (package), 381, 421, 422
Java
  applet, 29
  arithmetic, 457
  array, 462–467
  AWT, 31
  class file, 34
  collection, 462
  compiler, 34
  editor, 470
  integer division, 63, 457
  JVM, 23, 469, 523
  mathematical functions, 460
  rounding, 461
  runtime environment, 29, 469, 470
  virtual machine, 23, 469
JBuilder, 34, 470
JFIF, 19, 21, 23
JPEG, 15, 17–19, 21, 23, 31, 44, 219, 243, 284, 287, 312, 369
JPEG-2000, 19
JVM, 523

K

kernel, 99
killRoi (method), 510
Kirsch operator, 123

L

L*a*b*, 281
Lab_ColorSpace (class), 293, 297, 298
label, 200
Lanczos interpolation, 402, 408, 428
Laplace
  filter, 98, 131, 132, 136
  operator, 130
Laplacian of Gaussian (LoG), 116
LEFT_JUSTIFY (constant), 500
lens, 8
Line (class), 475, 506, 510
line, 499, 500
  endpoints, 166
  equation, 156, 159
  Hessian normal form, 159
  intercept/slope form, 156
  intersection, 166
linear
  convolution, 99, 497
  correlation, 99
  interpolation, 397
  transformation, 382
linearity, 100, 321
LinearMapping (class), 416, 419
lines per inch (lpi), 11
lineTo (method), 500
LinkedList (class), 202, 540
List (interface), 149, 150, 468, 528, 534, 537, 540, 541
list, 451
little endian, 22–24
loading images, 480
local mapping, 389
lock (method), 522
locking images, 522
lockSilently (method), 522
LoG
  filter, 116
  operator, 129
log (method), 81, 461, 495, 515
logic operation, 495
lookup table, 80, 152, 190, 530
LSB, 22
luminance, 256, 270
LZW, 15

M

magic number, 23
major axis, 228
makeGaussKernel1d (method), 103, 137
makeIndexColorImage (method), 252
makeInverseMapping (method), 423
makeLine (method), 513
makeMapping (method), 418, 420
makeOval (method), 513
makeRectangle (method), 513
managing windows, 521

Manhattan distance, 443
Mapping (class), 414
mapping
  affine, 378, 386
  bilinear, 385, 386
  four-point, 380
  function, 376
  linear, 382
  local, 389
  nonlinear, 386
  perspective, 380
  projective, 380–386
  ripple, 388
  spherical, 388
  three-point, 378
  twirl, 387
mask, 134, 218
  image, 511, 514
matchHistograms (method), 68
Math (class), 460, 461
Matrix (class), 421
MAX (constant), 82, 115, 496
max (method), 81, 461, 495
maximum
  filter, 105, 197
  frequency, 329, 350
maxMemory (method), 523
media-oriented color, 287
MEDIAN (constant), 115
median filter, 106, 115, 173, 497
  cross-shaped, 110
  weighted, 107
median-cut algorithm, 305
medianFilter (method), 497
memory management, 523
mesh partitioning, 389
MIN (constant), 82, 115, 496
min (method), 81, 461, 495
minimum filter, 105, 197
Mitchell-Netravali interpolation, 401, 428
mod operator, 340, 395, 453
modified auto-contrast, 58
modulus, *see* mod operator
moment, 219, 226–233
  central, 227
  Hu's, 233, 236
  invariant, 233
  least inertia, 228
moment (method), 229
morphing, 390
morphological filter, 173–197
  binary, 173–186
  closing, 185, 188, 193
  color, 187
  dilation, 177, 187, 191
  erosion, 178, 187, 191
  grayscale, 187–188
  opening, 185, 188, 193
  outline, 181, 194
moveTo (method), 500
MSB, 22
multi-resolution techniques, 127
MULTIPLY (constant), 82, 497, 531
multiply (method), 81, 85, 137, 495, 508
My_Inverter (plugin), 33

N

NaN (constant), 462, 535
nearest-neighbor interpolation, 397
NearestNeighborInterpolator (class), 424
NEGATIVE_INFINITY (constant), 462
neighborhood, 175, 200, 223
NetBeans, 34, 470
neutral
  point, 279
neutral element, 179
NewImage (class), 474, 477, 508, 518
newImage (method), 477
next (method), 534
nextGaussian (method), 51
nextInt (method), 51
NIH-Image, 29
NO_CHANGES (constant), 35, 45, 251, 519
NO_IMAGE_REQUIRED (constant), 519
NO_UNDO (constant), 520
Node (class), 202
noImage (method), 36, 86, 493, 503, 507, 515
noise (method), 495
nominal gamma value, 79
nonhomogeneous operation, 53
nonmaximum suppression, 164
normal distribution, 51
normalCentralMoment (method), 229
normalization, 93
normalized histogram, 63
NTSC, 75, 267, 269
null (constant), 463

Nyquist, 330, 350

O

$\mathcal{O}$ notation, 454
object, 451
OCR, 222, 235
octree algorithm, 306
`open` (method), 193, 194, 480, 481, 484
`Opener` (class), 481
`openImage` (method), 481, 482
opening, 185, 188, 193
opening images, 480–482, 484
`openMultiple` (method), 482
`openTiff` (method), 482
`openURL` (method), 482
optical axis, 7
`OR` (constant), 82, 497
`or` (method), 495
orientation, 228, 348, 350
orthogonal, 373
oscillation, 314, 315
outer product, 102
outline, 181, 194
`outline` (method), 194, 195
`OvalRoi` (class), 475, 506, 510
overlay, 533

P

packed ordering, 242–244
PAL, 75, 267
palette, 243, 248, 250
  image, *see* indexed color image
parameter space, 157
partial derivative, 119
Parzen window, 354, 355, 357, 358
`paste` (method), 522
pattern recognition, 3, 222
PDF, 15
pdf, *see* probability density function
perimeter, 223
period, 314
periodicity, 314, 344, 349, 352
perspective
  image, 170
  mapping, 380
  transformation, 7
phase, 315, 339
  angle, 316
Photoshop, 23
`PI` (constant), 461
PICT format, 15
piecewise linear function, 65
pinhole camera, 7
pixel, 5
  value, 11
`PixelInterpolator` (class), 423
planar ordering, 242
Plessey detector, 140
`PlugIn` (interface), 31, 473, 518
`PlugInFilter` (interface), 31, 246, 473, 519
PNG, 16, 23, 31, 247, 248, 284
`Pnt2d` (class), 414
`Point` (class), 534, 535, 538
point operation, 53–86, 494
  arithmetic, 80
  effects on histogram, 55
  gamma correction, 72
  histogram equalization, 59
  homogeneous, 80
  in ImageJ, 80–86
  inversion, 55
  thresholding, 55
point set, 176
point spread function, 104
`Point2D.Double` (class), 536
`PointRoi` (class), 506
`Polygon` (class), 511, 534, 542
polygon, 499
  area, 224
`PolygonRoi` (class), 475, 506, 510
`pop` (method), 203
populosity algorithm, 305
`POSITIVE_INFINITY` (constant), 462
PostScript, 15
`pow` (method), 77, 461
power spectrum, 339, 348
Prewitt operator, 120, 129
primary color, 240
print pattern, 363
probability, 64
  density function, 63
  distribution, 63
profile connection space, 292, 295
projection, 233, 237, 301
projective mapping, 380–386
`ProjectiveMapping` (class), 419, 426
pseudo-perspective mapping, 380
pseudocolor, 311
`push` (method), 203
`putBehind` (method), 521

putColumn (method), 489
putPixel (method), 33, 93, 94, 109, 111, 245, 460, 486, 490
putPixelValue (method), 487
putRow (method), 489
pyramid techniques, 127

Q

quadrilateral, 380
quantization, 10, 55, 301–310
  linear, 303
  scalar, 303
  vector, 305

R

Random (package), 51
random
  process, 63
  variable, 64
random (method), 51, 461
random image, 51
rank (method), 115
RankFilters (class), 115
RAS format, 22
raster image, 14
RAW format, 247
reading and writing pixels, 485
Rectangle (class), 536
rectangular pulse, 321, 323
  window, 356
reflect (method), 193
reflection, 177, 179, 180
refraction index, 388
region, 199–237
  area, 224, 228, 236
  centroid, 226, 236
  convex hull, 225
  diameter, 226
  eccentricity, 231
  labeling, 200–209
  major axis, 228
  matrix representation, 216
  moment, 226
  orientation, 228
  perimeter, 223
  projection, 233
  run length encoding, 218
  topology, 234
region of interest, 475, 481, 494, 495, 497, 501, 506–513, 520, 522
relative colorimetry, 289
remainder operator, 459
RenderingHints (class), 542
repaintImageWindows (method), 521
repaintWindow (method), 501
resampling, 390–392
resetEscape (method), 517
resetMinAndMax (method), 504
resetRoi (method), 512
resize (method), 485, 498
resolution, 10
reverting, 480, 484
revertToSaved (method), 484
RGB
  color image, 239–253
  color space, 240, 270
  format, 22
  image, 473
RGBtoHLS (method), 267
RGBtoHSB (method), 261–263, 295
RIGHT_JUSTIFY (constant), 500
rint (method), 461
ripple mapping, 388
Roberts operator, 123, 129
ROI, *see* region of interest
Roi (class), 475, 506
ROI_REQUIRED (constant), 520
rotate (method), 485, 498
rotateLeft (method), 498
rotateRight (method), 498
Rotation (class), 418, 426
rotation, 233, 361, 375, 377, 426
round (method), 77, 93, 94, 461
round function, 81, 453
rounding, 54, 82, 458, 461
roundness, 224
rubber banding, 510
run (method), 32, 480, 518, 519
run length encoding, 218
runPlugIn (method), 520

S

sampling, 325–330
  frequency, 350
  interval, 327, 328
  spatial, 9
  theorem, 329, 330, 337, 338, 350, 394
  time, 9
saturation, 42, 257, 490
save (method), 481, 482
saveAs (method), 481

saveAsBmp (method), 482
saveAsGif (method), 482
saveAsJpeg (method), 482, 483
saveAsLut (method), 483
saveAsPng (method), 483
saveAsRaw (method), 483
saveAsRawStack (method), 483
saveAsText (method), 483
saveAsTiff (method), 483
saveAsTiffStack (method), 484
saveAsZip (method), 484
saving images, 480–484, 521
scale (method), 485, 498, 542
Scaling (class), 418
scaling, 233, 375, 377
separability, 101, 115, 181, 371
separable filter, 131
sequence, 451
Set (interface), 468
set, 176, 451
set (method), 34, 54, 63, 111, 259, 487, 490
setAntialiasedText (method), 500
setBackgroundValue (method), 498
setBrightness (method), 490
setClipRect (method), 500
setColor (method), 500, 542
setColorModel (method), 249, 250, 252, 504
setCurrentWindow (method), 521
setDoScaling (method), 494
setf (method), 441, 487
setFloatArray (method), 490
setFont (method), 500
setHistogramRange (method), 494
setHistogramSize (method), 494
setHSB (method), 490
setIntArray (method), 489
setInterpolate (method), 485, 489, 498
setJustification (method), 500
setLineWidth (method), 500
setMask (method), 512
setMinAndMax (method), 502–504, 533
setNormalize (method), 114, 137, 148, 531
setPixels (method), 488, 506
setProcessor (method), 478
setProperty (method), 515, 516
setRenderingHint (method), 542
setRGB (method), 490
setRoi (method), 510, 512
setSilentMode (method), 482
setSlice (method), 501
setSliceLabel (method), 506
setStack (method), 504
setStroke (method), 542
setTempCurrentImage (method), 521
setThreshold (method), 502, 503
setTitle (method), 501
setup (method), 32, 35, 85, 246, 248, 493, 519, 520, 525, 533
setValue (method), 52, 152, 500, 527
setWeightingFactors (method), 257, 485
Shah function, 327
Shannon, 330
Shape (class), 534, 541, 542
shape
  feature, 222
  number, 220, 221, 236
sharpen (method), 497
Shear (class), 418
shearing, 377
shift property, 324
shiftKeyDown (method), 517
short (type), 491
ShortProcessor (class), 253, 472, 491
ShortStatistics (class), 494
show (method), 52, 153, 247, 501, 502, 508, 514, 518, 530
showDialog (method), 86, 507, 518, 526
showMessage (method), 515, 526
showMessageWithCancel (method), 515
showProcessor (method), 530
showProgress (method), 517
showStatus (method), 517
signal space, 100, 320, 338
similarity, 324
sin (method), 461
Sinc function, 321, 394, 404
sine function, 322
  one-dimensional, 314
sine transform, 367
size (method), 534
skeletonization, 195
skeletonize (method), 195

slice, 506
smooth (method), 497
smoothing filter, 89, 92
snapshot array, 489
Sobel operator, 120, 129, 497
software, 28
solve (method), 422
sort (method), 109, 150, 300, 466, 529
sorting arrays, 466
source-to-target mapping, 390
spaceBarDown (method), 517
spatial sampling, 9
special image, 13
spectrum, 313–373
spherical mapping, 388
spline
  cardinal, 398, 400
  Catmull-Rom, 399, 400, 402
  cubic, 400, 401
  cubic B-, 400–402, 428
  interpolation, 399
sqr (method), 81, 495, 531
sqrt (method), 81, 461, 495
square window, 358
sRGB, 79, 256, 257, 283, 285, 286, 288, 292
  ambient lighting, 280
  grayscale conversion, 287
  white point, 280
Stack (class), 202, 203
stack, 200, 246, 504, 505, 508, 512, 520
STACK_REQUIRED (constant), 520
StackStatistics (class), 494
standard deviation, 51
standard illuminant, 279, 288
storing images, 480
Stroke (class), 542
structure, 451
structuring element, 175, 176, 180, 181, 187, 191
SUBTRACT (constant), 82, 497
super (method), 541
super-Gaussian window, 355, 356
SUPPORTS_MASKING (constant), 520

T

tan (method), 461
tangent function, 462
target-to-source mapping, 387, 391, 415
template matching, 429–431, 440
temporal sampling, 9
TextRoi (class), 475, 510
TGA format, 22
thin lens, 8
thinning, 195
thread, 522
three-point mapping, 378
threshold, 55, 129, 163
threshold (method), 55, 496
TIFF, 15, 19, 21, 23, 31, 219, 246, 248
time unit, 315
toArray (method), 151, 530
toCIEXYZ (method), 292–295, 297
toDegrees (method), 461
topological property, 234
toRadians (method), 461
tracking, 139
transform pair, 320
TransformJ (package), 413
translate (method), 535, 542
Translation (class), 418
translation, 233, 377
transparency, 83, 244, 252
tree, 202
true color image, 13, 17, 241, 243, 244
truncate function, 453, 459
truncation, 82
tuple, 451
twirl mapping, 387
TwirlMapping (class), 422
type cast, 54, 458, 488
TYPE_Lab (constant), 297
TypeConverter (class), 252

U

undercolor-removal function, 273
undo array, 489
uniform distribution, 51
unit square, 386
unlock (method), 522
unsharp masking, 133–137
UnsharpMask (class), 136
unsharpMask (method), 137
unsigned byte (type), 459
updateAndDraw (method), 36, 52, 249, 484, 502

`updateAndRepaintWindow` (method), 502
user interaction, 515–517

V

variance, 434
`Vector` (class), 149, 463, 468, 529
vector, 451
  graphic, 14
  graphics, 216
`versionLessThan` (method), 523
viewing angle, 280

W

`wait` (method), 517
Walsh transform, 372
warping, 387
`wasCanceled` (method), 86, 507, 518, 526
wave number, 334, 344, 349, 368
wavelet, 373
Website for this book, 35
`white` (constant), 541
white point, 258, 279, 282
  D50, 279, 292
  D65, 280, 284
window management, 521
windowed matching, 439
windowing, 352
windowing function, 354–357
  Bartlett, 355, 357, 358
  cosine^2, 357, 358
  elliptical, 355, 356
  Gaussian, 355, 356, 358
  Hanning, 355, 357, 358
  Parzen, 355, 357, 358
  rectangular pulse, 356
  super-Gaussian, 355, 356
`WindowManager` (class), 86, 249, 475, 507, 521
WMF format, 15
`write` (method), 515

X

XBM/XPM format, 22
`XOR` (constant), 497
`xor` (method), 496
XYZ scaling, 289

Y

YC_bC_r, 272
YC_bC_r, 270
YIQ, 269, 272
YUV, 255, 268, 270, 272

Z

ZIP, 15
zoom, 498

About the Authors

Wilhelm Burger received a Master's degree in Computer Science from the University of Utah (Salt Lake City) and a doctorate in Systems Science from Johannes Kepler University in Linz, Austria. As a post-graduate researcher at the Honeywell Systems & Research Center in Minneapolis and the University of California at Riverside, he worked mainly in the areas of visual motion analysis and autonomous navigation. In the Austrian research initiative on digital imaging, he was engaged in projects on generic object recognition and biometric identification. Since 1996, he has been the director of the Digital Media degree programs at the Upper Austria University of Applied Sciences at Hagenberg. Personally the author appreciates large-engine vehicles and (occasionally) a glass of dry "Veltliner".

Mark J. Burge received a BA degree from Ohio Wesleyan University, a MSc in Computer Science from the Ohio State University, and a doctorate from Johannes Kepler University in Linz, Austria. He spent several years as a researcher in Zürich, Switzerland at the Swiss Federal Institute of Technology (ETH), where he worked in computer vision and pattern recognition. As a post-graduate researcher at the Ohio State University, he was involved in the "Image Understanding and Interpretation Project" sponsored by the NASA Commercial Space Center. He earned tenure within the University System of Georgia as an associate professor in computer science and served as a Program Director at the National Science Foundation. Currently he is a Principal at Noblis (Mitretek) in Washington D.C. Personally, he is an expert on classic Italian espresso machines.

About this Book

The complete manuscript for this book was prepared by the authors "camera-ready" in LaTeX using Donald Knuth's Computer Modern fonts. The additional packages algorithmicx (by Szász János) for presenting algorithms, listings (by Carsten Heinz) for listing progam code, and psfrag (by Michael C. Grant and David Carlisle) for replacing text in graphics were particularly helpful in this task. Most illustrations were produced with Macromedia Freehand (now part of Adobe), function plots with Mathematica, and images with ImageJ or Adobe Photoshop. All book figures, test images in color and full resolution, as well as the Java source code for all examples are available at the book's support site: www.imagingbook.com.