

1100101101010000101101010101010110110100101101000101010101010101010101
00000101010101010100111010100101101001010101010101010101010101010101
11011000010101010101110101001011010101010101010101010101010101010101
10010110110000101101010101010110110100100101010101010101010101010101
0010101101

Wilhelm Burger
Mark J. Burge

UNDERGRADUATE TOPICS
in COMPUTER SCIENCE

Principles of Digital Image Processing

Core Algorithms

 Springer

Undergraduate Topics in Computer Science

For further volumes:
<http://www.springer.com/series/7592>

Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

Wilhelm Burger · Mark J. Burge

Principles of Digital Image Processing

Core Algorithms

Wilhelm Burger
University of Applied Sciences
Hagenberg
Austria
wilbur@ieee.org

Mark J. Burge
noblis.org
Washington, D.C.
mburge@acm.org

Series editor

Ian Mackie, École Polytechnique, France and University of Sussex, UK

Advisory board

Samson Abramsky, University of Oxford, UK
Chris Hankin, Imperial College London, UK
Dexter Kozen, Cornell University, USA
Andrew Pitts, University of Cambridge, UK
Hanne Riis Nielson, Technical University of Denmark, Denmark
Steven Skiena, Stony Brook University, USA
Iain Stewart, University of Durham, UK
David Zhang, The Hong Kong Polytechnic University, Hong Kong

Undergraduate Topics in Computer Science ISSN 1863-7310
ISBN 978-1-84800-194-7 e-ISBN 978-1-84800-195-4
DOI 10.1007/978-1-84800-195-4

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2008942518

© Springer-Verlag London Limited 2009

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers. The use of registered names, trademarks, etc., in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Printed on acid-free paper

Springer Science+Business Media
springer.com

Preface

This is the second volume of a book series that provides a modern, algorithmic introduction to digital image processing. It is designed 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 updated and enhanced paperback edition of our comprehensive textbook *Digital Image Processing: An Algorithmic Approach Using Java* packages the original material into a series of compact volumes, thereby supporting a flexible sequence of courses in digital image processing. Tailoring the contents to the scope of individual semester courses is also an attempt to provide affordable (and “backpack-compatible”) textbooks without compromising the quality and depth of content.

This second volume, titled *Core Algorithms*, extends the introductory material presented in the first volume (*Fundamental Techniques*) with additional techniques that are, nevertheless, part of the standard image processing toolbox. A forthcoming third volume (*Advanced Techniques*) will extend this series and add important material beyond the elementary level, suitable for an advanced undergraduate or even graduate course.

Math, Algorithms, and “Real” Code

It has been our experience in teaching in this field that mastering the core takes more than just reading about the techniques—it requires active construction and experimentation with the *algorithms* to acquire a feeling for how to use these methods in the real world. Internet search engines have made finding *someone’s* code for almost any imaging problem as simple as coming up with a succinct enough set of keywords. However, the problem is not to find *a* solution, but developing one’s own and understanding how it works—or why it

eventually does not. Whereas we feel that the real value of this series is not in its code, but rather in the critical selection of algorithms, illustrated explanations, and concise mathematical derivations, we continue to augment our algorithms with complete implementations, as even the best description of a method often omits some essential element necessary for the actual implementation, which only the unambiguous semantics of a real programming language can provide.

Online Resources

The authors maintain a Website for this text that provides supplementary materials, including the complete Java source code for the examples, the test images used in the examples, and corrections. Visit this site at

www.imagingbook.com

Additional materials are available for educators, including a complete set of figures, tables, and mathematical elements shown in the text, in a format suitable for easy inclusion in presentations and course notes. Comments, questions, and corrections are welcome and should be addressed to

imagingbook@gmail.com

Acknowledgements

As with its predecessors, this book would not have been possible without the understanding and steady support of our families. Thanks go to Wayne Rasband (NIH) for developing and refining ImageJ and for his truly outstanding support of the community. We appreciate the contribution from many careful readers who have contacted us to suggest new topics, recommend alternative solutions, or to suggest corrections. Finally, we are grateful to Wayne Wheeler for initiating this book series and Catherine Brett and her colleagues at Springer's UK and New York offices for their professional support.

Hagenberg, Austria / Washington DC, USA

June 2008

Contents

Preface	v
1. Introduction	1
1.1 Programming with Images	2
1.2 Image Analysis	3
2. Regions in Binary Images.....	5
2.1 Finding Image Regions	6
2.1.1 Region Labeling with Flood Filling	6
2.1.2 Sequential Region Labeling	11
2.1.3 Region Labeling—Summary	17
2.2 Region Contours	17
2.2.1 External and Internal Contours	18
2.2.2 Combining Region Labeling and Contour Finding	20
2.2.3 Implementation	22
2.2.4 Example	25
2.3 Representing Image Regions	26
2.3.1 Matrix Representation	26
2.3.2 Run Length Encoding.....	27
2.3.3 Chain Codes	28
2.4 Properties of Binary Regions.....	32
2.4.1 Shape Features	32
2.4.2 Geometric Features	33
2.4.3 Statistical Shape Properties	36
2.4.4 Moment-Based Geometrical Properties	38
2.4.5 Projections	44

2.4.6	Topological Properties	45
2.5	Exercises	46
3.	Detecting Simple Curves	49
3.1	Salient Structures	49
3.2	Hough Transform	50
3.2.1	Parameter Space	51
3.2.2	Accumulator Array	54
3.2.3	A Better Line Representation	54
3.3	Implementing the Hough Transform	55
3.3.1	Filling the Accumulator Array	56
3.3.2	Analyzing the Accumulator Array	56
3.3.3	Hough Transform Extensions	60
3.4	Hough Transform for Circles and Ellipses	63
3.4.1	Circles and Arcs	64
3.4.2	Ellipses	66
3.5	Exercises	67
4.	Corner Detection	69
4.1	Points of Interest	69
4.2	Harris Corner Detector	70
4.2.1	Local Structure Matrix	70
4.2.2	Corner Response Function (CRF)	71
4.2.3	Determining Corner Points	72
4.2.4	Example	72
4.3	Implementation	72
4.3.1	Step 1: Computing the Corner Response Function	76
4.3.2	Step 2: Selecting “Good” Corner Points	79
4.3.3	Displaying the Corner Points	83
4.3.4	Summary	83
4.4	Exercises	84
5.	Color Quantization	85
5.1	Scalar Color Quantization	86
5.2	Vector Quantization	88
5.2.1	Populosity algorithm	88
5.2.2	Median-cut algorithm	88
5.2.3	Octree algorithm	89
5.2.4	Other methods for vector quantization	94
5.3	Exercises	95

6. Colorimetric Color Spaces	97
6.1 CIE Color Spaces	98
6.1.1 CIE XYZ color space	98
6.1.2 CIE x, y chromaticity	99
6.1.3 Standard illuminants	101
6.1.4 Gamut	102
6.1.5 Variants of the CIE color space	103
6.2 CIE L*a*b*	104
6.2.1 Transformation CIEXYZ → L*a*b*	104
6.2.2 Transformation L*a*b* → CIEXYZ	105
6.2.3 Measuring color differences	105
6.3 sRGB	106
6.3.1 Linear vs. nonlinear color components	107
6.3.2 Transformation CIEXYZ→sRGB	108
6.3.3 Transformation sRGB→CIE XYZ	108
6.3.4 Calculating with sRGB values	109
6.4 Adobe RGB	111
6.5 Chromatic Adaptation	111
6.5.1 XYZ scaling	112
6.5.2 Bradford adaptation	113
6.6 Colorimetric Support in Java	114
6.6.1 sRGB colors in Java	114
6.6.2 Profile connection space (PCS)	115
6.6.3 Color-related Java classes	118
6.6.4 A L*a*b* color space implementation	120
6.6.5 ICC profiles	121
6.7 Exercises	124
7. Introduction to Spectral Techniques	125
7.1 The Fourier Transform	126
7.1.1 Sine and Cosine Functions	126
7.1.2 Fourier Series of Periodic Functions	130
7.1.3 Fourier Integral	130
7.1.4 Fourier Spectrum and Transformation	131
7.1.5 Fourier Transform Pairs	132
7.1.6 Important Properties of the Fourier Transform	136
7.2 Working with Discrete Signals	137
7.2.1 Sampling	137
7.2.2 Discrete and Periodic Functions	144
7.3 The Discrete Fourier Transform (DFT)	144
7.3.1 Definition of the DFT	144

7.3.2	Discrete Basis Functions	147
7.3.3	Aliasing Again!	148
7.3.4	Units in Signal and Frequency Space	152
7.3.5	Power Spectrum	153
7.4	Implementing the DFT	154
7.4.1	Direct Implementation	154
7.4.2	Fast Fourier Transform (FFT)	155
7.5	Exercises	156
8.	The Discrete Fourier Transform in 2D	157
8.1	Definition of the 2D DFT	157
8.1.1	2D Basis Functions	158
8.1.2	Implementing the Two-Dimensional DFT	158
8.2	Visualizing the 2D Fourier Transform	162
8.2.1	Range of Spectral Values	162
8.2.2	Centered Representation	162
8.3	Frequencies and Orientation in 2D	164
8.3.1	Effective Frequency	164
8.3.2	Frequency Limits and Aliasing in 2D	164
8.3.3	Orientation	165
8.3.4	Normalizing the 2D Spectrum	166
8.3.5	Effects of Periodicity	167
8.3.6	Windowing	169
8.3.7	Windowing Functions	169
8.4	2D Fourier Transform Examples	171
8.5	Applications of the DFT	175
8.5.1	Linear Filter Operations in Frequency Space	175
8.5.2	Linear Convolution versus Correlation	177
8.5.3	Inverse Filters	178
8.6	Exercises	180
9.	The Discrete Cosine Transform (DCT)	183
9.1	One-Dimensional DCT	183
9.1.1	DCT Basis Functions	184
9.1.2	Implementing the One-Dimensional DCT	186
9.2	Two-Dimensional DCT	187
9.2.1	Separability	187
9.2.2	Examples	188
9.3	Other Spectral Transforms	188
9.4	Exercises	190

10. Geometric Operations	191
10.1 2D Mapping Function	193
10.1.1 Simple Mappings	193
10.1.2 Homogeneous Coordinates	194
10.1.3 Affine (Three-Point) Mapping	195
10.1.4 Projective (Four-Point) Mapping	197
10.1.5 Bilinear Mapping	203
10.1.6 Other Nonlinear Image Transformations	204
10.1.7 Local Image Transformations	207
10.2 Resampling the Image	209
10.2.1 Source-to-Target Mapping	209
10.2.2 Target-to-Source Mapping	210
10.3 Interpolation	210
10.3.1 Simple Interpolation Methods	211
10.3.2 Ideal Interpolation	213
10.3.3 Interpolation by Convolution	217
10.3.4 Cubic Interpolation	217
10.3.5 Spline Interpolation	219
10.3.6 Lanczos Interpolation	223
10.3.7 Interpolation in 2D	225
10.3.8 Aliasing	234
10.4 Java Implementation	238
10.4.1 Geometric Transformations	238
10.4.2 Pixel Interpolation	248
10.4.3 Sample Applications	251
10.5 Exercises	253
11. Comparing Images	255
11.1 Template Matching in Intensity Images	257
11.1.1 Distance between Image Patterns	258
11.1.2 Implementation	266
11.1.3 Matching under Rotation and Scaling	267
11.2 Matching Binary Images	269
11.2.1 Direct Comparison	269
11.2.2 The Distance Transform	270
11.2.3 Chamfer Matching	274
11.3 Exercises	278
A. Mathematical Notation	279
A.1 Symbols	279
A.2 Set Operators	281
A.3 Complex Numbers	282

B. Source Code	283
B.1 Combined Region Labeling and Contour Tracing	283
B.1.1 <code>Contour_Tracing_Plugin</code> (Class)	283
B.1.2 <code>Contour</code> (Class)	285
B.1.3 <code>BinaryRegion</code> (Class)	286
B.1.4 <code>ContourTracer</code> (Class)	287
B.1.5 <code>ContourOverlay</code> (Class)	292
B.2 Harris Corner Detector	294
B.2.1 <code>Harris_Corner_Plugin</code> (Class)	294
B.2.2 <code>File_Corner</code> (Class)	295
B.2.3 <code>File_HarrisCornerDetector</code> (Class)	296
B.3 Median-Cut Color Quantization	301
B.3.1 <code>ColorQuantizer</code> (Interface)	301
B.3.2 <code>MedianCutQuantizer</code> (Class)	301
B.3.3 <code>ColorHistogram</code> (Class)	309
B.3.4 <code>Median_Cut_Quantization</code> (Class)	310
Bibliography	313
Index	321

1

Introduction

Today, IT professionals must be more than simply familiar with digital image processing. They are expected to be able to knowledgeably manipulate images and related digital media and, 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.

In fact, it often 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 anticipate at first.

We expect technology to improve on what we as humans 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. Although it is possible that this will always remain true, we should not be discouraged, but instead consider this a creative engineering challenge. On the other hand, image processing technology has become a reliable and indispensable element in many everyday applications. As in every engineering discipline, sound knowledge of elementary concepts, careful design, and professional implementation are the essential keys to success.

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 non-specialists. 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 [22, 27, 77]. Although graphics professionals today tend to be interested in topics such as realism and, especially in terms of computer games, rendering speed, the field does draw on a number of methods that originate in image processing, such as image transformation (morphing), reconstruction of 3D models from image data, and specialized techniques such as image-based and nonphotorealistic rendering [57, 78]. 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 [79]. 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

Although image analysis is not the central theme of this book, most methods described here exhibit a certain “analytical flavor” that adds to the elementary “pixel crunching” techniques described in the preceding volume [14]. This intersection becomes evident in tasks like segmenting image regions (Ch. 2), detecting simple curves and corners (Chs. 3–4), or comparing images (Ch. 11) at the pixel level. All these methods work directly on the pixel data in a *bottom-up* way without recourse to any domain-specific or “semantic” knowledge. In some sense, one could describe all these methods as “dumb and blind”, which differentiates them from the approach pursued in *pattern recognition* and *computer vision*. Although these two disciplines are firmly grounded in, and rely heavily on, image processing, their ultimate goals are much loftier.

Pattern recognition is primarily a mathematical discipline and has been responsible for techniques such as probabilistic modeling, clustering, decision trees, or 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 for 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, [21, 55, 72]).

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, [19, 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 [2, 24, 35, 65, 69, 73].

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.

2

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 connected regions in images and how to isolate and describe such structures.

Let us assume that our task is to devise a procedure for finding the number and type of objects contained in a figure like Fig. 2.1. As long as we continue

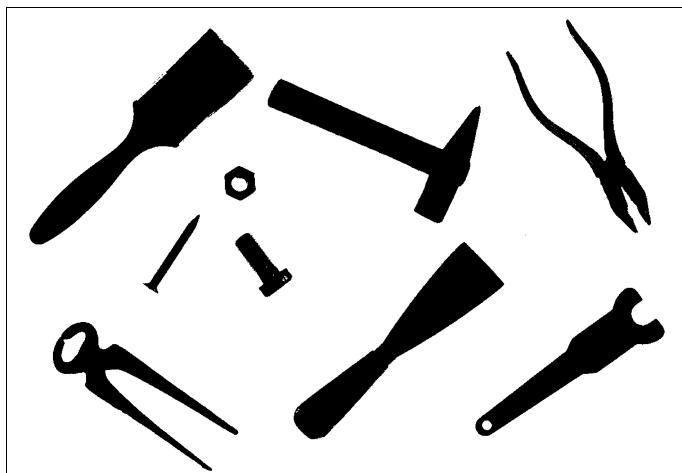


Figure 2.1 Binary image with nine objects. Each object corresponds to a connected region of related foreground pixels.

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*.

2.1 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. 2.2.2, 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 (see Vol. 1 [14, Fig. 7.5]) for determining when two pixels are “connected” to each other, since under each definition we can end up with different results. In the following region-marking algorithms, we use the following convention: the original binary image $I(u, v)$ contains the values 0 and 1 to mark the *background* and *foreground*, respectively; any other value is used for numbering (labeling) the regions, i. e., the pixel values are

$$I(u, v) = \begin{cases} 0 & \text{a } \textit{background} \text{ pixel} \\ 1 & \text{a } \textit{foreground} \text{ pixel} \\ 2, 3, \dots & \text{a region } \textit{label}. \end{cases}$$

2.1.1 Region Labeling with Flood Filling

The underlying algorithm for region marking by *flood filling* is simple: search for an unmarked foreground pixel and then fill (visit and mark) all the rest of the neighboring pixels in its region (Alg. 2.1). 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

Algorithm 2.1 Region marking using *flood filling* (Part 1). 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. The actual FLOODFILL() procedure is described in Alg. 2.2.

```

1: REGIONLABELING( $I$ )
    $I$ : binary image;  $I(u, v) = 0$ : background,  $I(u, v) = 1$ : foreground
   The image  $I$  is labeled (destructively modified) and returned.

2: Let  $m \leftarrow 2$                                  $\triangleright$  value of the next label to be assigned
3: for all image coordinates  $(u, v)$  do
4:   if  $I(u, v) = 1$  then
5:     FLOODFILL( $I, u, v, m$ )            $\triangleright$  use any version from Alg. 2.2
6:      $m \leftarrow m + 1$ .
7: return the labeled image  $I$ .

```

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. 2.2):

- (A) **Recursive Flood Filling:** The recursive version (Alg. 2.2, lines 1–8) does not make use of explicit data structures to keep track of the image coordinates but uses the local variables that are implicitly allocated by recursive procedure calls.¹ Within each region, a tree structure, rooted at the starting point, is defined by the neighborhood relation between pixels. The recursive step corresponds to a *depth-first traversal* [20] of this tree and results in very short and elegant program code. Unfortunately, since the maximum depth of the recursion—and thus the size of the required stack memory—is proportional to the size of the region, stack memory is quickly exhausted. Therefore this method is risky and really only practical for very small images.
- (B) **Iterative Flood Filling (*depth-first*):** Every recursive algorithm can also be reformulated as an iterative algorithm (Alg. 2.2, lines 9–20) by implementing and managing its own *stacks*. In this case, the stack records the “open” (that is, the adjacent but not yet visited) elements. As in the recursive version (A), the corresponding tree of pixels is traversed in *depth-first* order. By making use of its own dedicated stack (which is created in the much larger *heap* memory), the depth of the tree is no longer limited

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 2.2 Region marking using *flood filling* (Part 2). Three variations of the FLOODFILL() procedure: *recursive*, *depth-first*, and *breadth-first*.

```

1:  FLOODFILL( $I, u, v, \text{label}$ )                                 $\triangleright$  Recursive Version
2:  if ( $u, v$ ) is inside the image and  $I(u, v) = 1$  then
3:    Set  $I(u, v) \leftarrow \text{label}$ 
4:    FLOODFILL( $I, u+1, v, \text{label}$ )
5:    FLOODFILL( $I, u, v+1, \text{label}$ )
6:    FLOODFILL( $I, u, v-1, \text{label}$ )
7:    FLOODFILL( $I, u-1, v, \text{label}$ )
8:  return.

9: FLOODFILL( $I, u, v, \text{label}$ )                                 $\triangleright$  Depth-First Version
10: Create an empty stack  $S$ 
11: Put the seed coordinate  $(u, v)$  onto the stack: PUSH( $S, (u, v)$ )
12: while  $S$  is not empty do
13:   Get the next coordinate from the top of the stack:
14:      $(x, y) \leftarrow \text{POP}(S)$ 
15:   if  $(x, y)$  is inside the image and  $I(x, y) = 1$  then
16:     Set  $I(x, y) \leftarrow \text{label}$ 
17:     PUSH( $S, (x+1, y)$ )
18:     PUSH( $S, (x, y+1)$ )
19:     PUSH( $S, (x, y-1)$ )
20:     PUSH( $S, (x-1, y)$ )
21:   return.

22: FLOODFILL( $I, u, v, \text{label}$ )                                 $\triangleright$  Breadth-First Version
23: Create an empty queue  $Q$ 
24: Insert the seed coordinate  $(u, v)$  into the queue: ENQUEUE( $Q, (u, v)$ )
25: while  $Q$  is not empty do
26:   Get the next coordinate from the front of the queue:
27:      $(x, y) \leftarrow \text{DEQUEUE}(Q)$ 
28:   if  $(x, y)$  is inside the image and  $I(x, y) = 1$  then
29:     Set  $I(x, y) \leftarrow \text{label}$ 
30:     ENQUEUE( $Q, (x+1, y)$ )
31:     ENQUEUE( $Q, (x, y+1)$ )
32:     ENQUEUE( $Q, (x, y-1)$ )
33:     ENQUEUE( $Q, (x-1, y)$ )
34:   return.

```

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. 2.2, lines 21–32). 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 environment does not support more than about 10,000 recursive calls of the FLOODFILL() procedure (Alg. 2.2, line 1) 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 2.1 gives the complete Java implementation for both variants of the iterative FLOODFILL() procedure. 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. 2.1, line 1), 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. 2.1, line 18). We have specified `<Point>` as a type parameter for both generic container classes so they can only contain objects of type `Point`.³

Figure 2.2 illustrates the progress of the region marking in both variants within an example region, where the start point (i.e., seed point), which would normally lie on a contour edge, has been placed arbitrarily within the region in order to better illustrate the process. It is clearly visible that the *depth-first* method first explores *one* direction (in this case horizontally to the left) completely (that is, until it reaches the edge of the region) and only then examines the remaining directions. In contrast the *breadth-first* method markings proceed outward, layer by layer, equally in all directions.

Due to the way exploration takes place, the memory requirement of the *breadth-first* variant of the *flood-fill* version is generally much lower than that of the *depth-first* variant. For example, when flood filling the region in Fig. 2.2 (using the implementation given Prog. 2.1), the stack in the *depth-first* variant

² The class `LinkedList` is a part of the *Java Collection Framework* (see also Vol. 1 [14, Appendix B.2]).

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).

Depth-first variant (using a *stack*):

```

1 void floodFill(int x, int y, int label) {
2     Stack<Point> s = new Stack<Point>(); // stack
3     s.push(new Point(x,y));
4     while (!s.isEmpty()){
5         Point n = s.pop();
6         int u = n.x;
7         int v = n.y;
8         if ((u>=0) && (u<width) && (v>=0) && (v<height>)
9             && ip.getPixel(u,v)==1) {
10            ip.putPixel(u, v, label);
11            s.push(new Point(u+1, v));
12            s.push(new Point(u, v+1));
13            s.push(new Point(u, v-1));
14            s.push(new Point(u-1, v));
15        }
16    }
17 }
```

Breadth-first variant (using a *queue*):

```

18 void floodFill(int x, int y, int label) {
19     LinkedList<Point> q = new LinkedList<Point>();
20     q.addFirst(new Point(x, y));
21     while (!q.isEmpty()) {
22         Point n = q.removeLast();
23         int u = n.x;
24         int v = n.y;
25         if ((u>=0) && (u<width) && (v>=0) && (v<height>)
26             && ip.getPixel(u,v)==1) {
27            ip.putPixel(u, v, label);
28            q.addFirst(new Point(u+1, v));
29            q.addFirst(new Point(u, v+1));
30            q.addFirst(new Point(u, v-1));
31            q.addFirst(new Point(u-1, v));
32        }
33    }
34 }
```

Program 2.1 Flood filling (Java implementation). The standard class `Point` (defined in `java.awt`) represents a single pixel coordinate. The *depth-first* variant uses the standard 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.

grows to a maximum of 28,822 elements, while the queue used by the *breadth-first* variant never exceeds a maximum of 438 nodes.

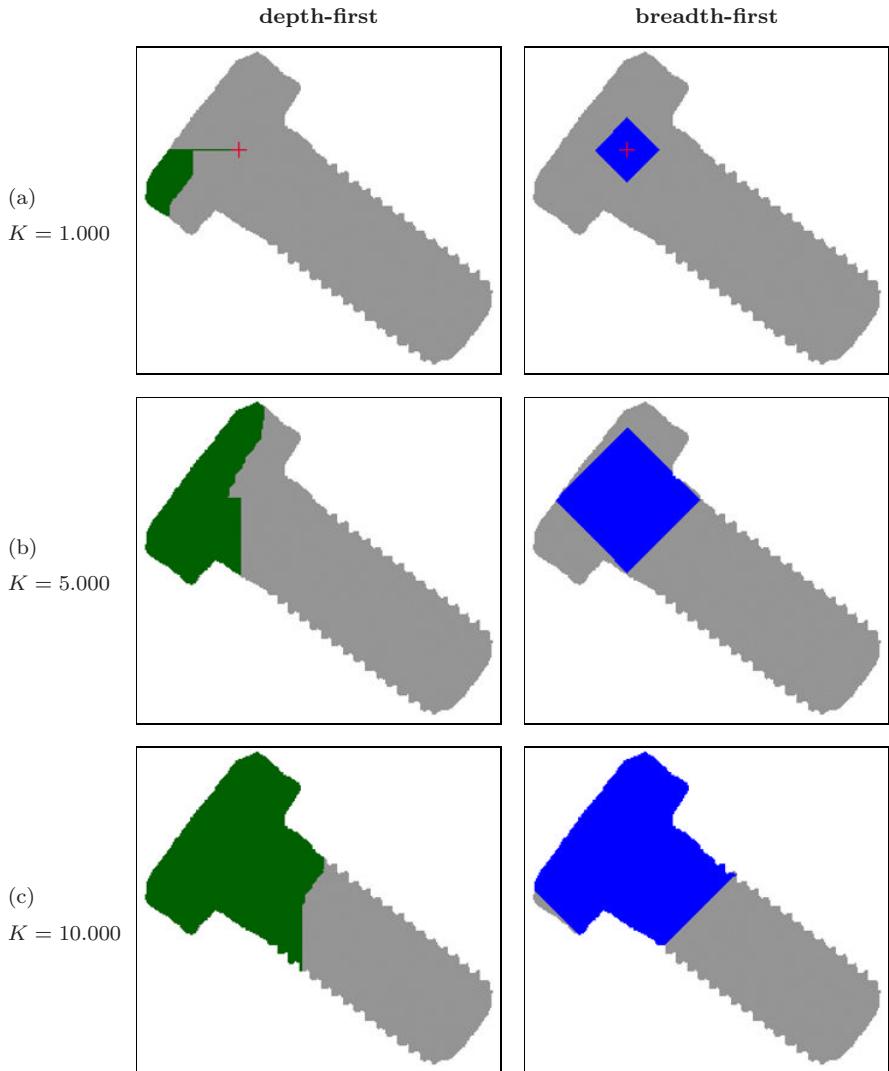


Figure 2.2 Iterative *flood filling*—comparison between the *depth-first* and *breadth-first* approach. The starting point, marked + in the top two image (a), was arbitrarily chosen. Intermediate results of the *flood fill* process after 1000 (a), 5000 (b), and 10,000 (c) marked pixels are shown. The image size is 250×242 pixels.

2.1.2 Sequential Region Labeling

Sequential region marking is a classical, nonrecursive technique that is known in the literature as “region labeling”. The algorithm consists of two steps: (1) 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 connected region. Even though this algorithm is relatively complex, especially its second stage, its moderate memory requirements make it a good choice under limited memory conditions. However, this is not a major issue on modern computers and thus, in terms of overall efficiency, sequential labeling offers no clear advantage over the simpler methods described earlier. The sequential technique is nevertheless interesting (not only from a historic perspective) and inspiring. The complete process is summarized in Alg. 2.3–2.4, with the following main steps:

Step 1: Initial labeling

In the first stage of region labeling, the image is traversed from top left to bottom right sequentially to assign a preliminary label to every foreground pixel. Depending on the definition of neighborhood (either 4- or 8-connected) used, the following neighbors in the direct vicinity of each pixel must be examined (\times marks the current pixel at the position (u, v)):

$$\mathcal{N}_4(u, v) = \begin{array}{|c|c|} \hline N_1 & \times \\ \hline \end{array} \quad \text{or} \quad \mathcal{N}_8(u, v) = \begin{array}{|c|c|c|} \hline N_2 & N_3 & N_4 \\ \hline N_1 & \times & \quad \\ \hline \end{array}$$

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 $N_1 \dots N_4$ must be examined.

Example

In the following example (Figs. 2.3–2.5), we use an 8-connected neighborhood and a very simple test image (Fig. 2.3 (a)) to demonstrate the sequential region labeling process.

Propagating labels. Again we assume that, in the image, the value $I(u, v) = 0$ represents background pixels and the value $I(u, v) = 1$ represents foreground pixels. We will also consider neighboring pixels that lie outside of the image matrix (e.g., on the array borders) to be part of the background. The neighborhood region $\mathcal{N}(u, v)$ is slid over the image horizontally and then vertically, starting from the top left corner. When the current image element $I(u, v)$ is a foreground pixel, it is either assigned a new region number or, in the case where one of its previously examined neighbors in $\mathcal{N}(u, v)$ was a foreground pixel, it takes on the region number of the neighbor. In this way, existing region numbers propagate in the image from the left to the right and from the top to the bottom, as shown in (Fig. 2.3 (b, c)).

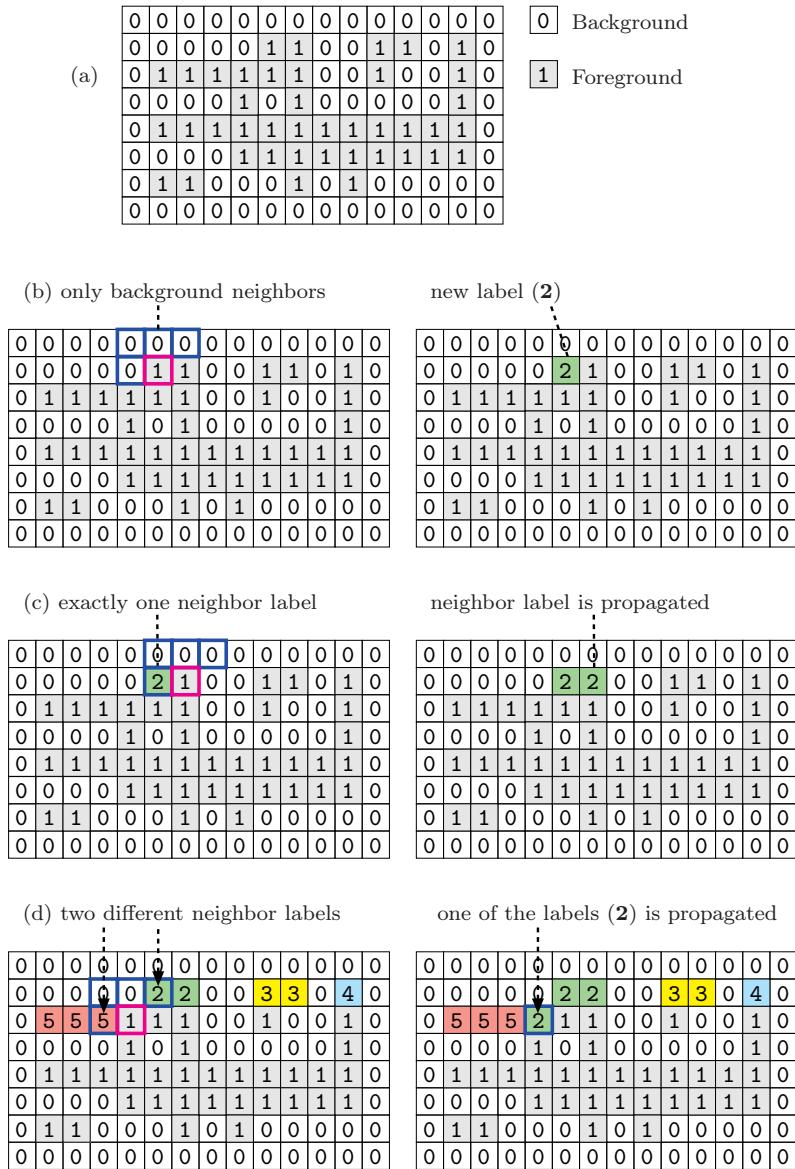


Figure 2.3 Sequential region labeling—label propagation. Original image (a). The first foreground pixel (marked 1) is found in (b): all neighbors are background pixels (marked 0), and the pixel is assigned the first label (2). In the next step (c), there is exactly one neighbor pixel marked with the label 2, so this value is propagated. In (d) there are two neighboring pixels, and they have differing labels (2 and 5); one of these values is propagated, and the collision (2, 5) is registered.

Algorithm 2.3 Sequential region labeling (Part 1). The binary input image I contains the values $I(u, v) = 0$ for background pixels and $I(u, v) = 1$ for foreground (region) pixels. The resulting region labels in I have the values $2 \dots m-1$.

```

1: SEQUENTIALLABELING( $I$ )
    $I$ : binary image;  $I(u, v) = 0$ : background,  $I(u, v) = 1$ : foreground
   The image  $I$  is labeled (destructively modified) and returned.
    $m$ : number of assigned labels;  $\mathcal{C}$ : set of label collisions.

2:  $(m, \mathcal{C}) \leftarrow \text{ASSIGNINITIALLABELS}(I)$ 
3:  $\mathcal{R} \leftarrow \text{RESOLVELABELCOLLISIONS}(m, \mathcal{C})$                                 ▷ see Alg. 2.4
4:  $\text{RELABELIMAGE}(I, \mathcal{R})$                                               ▷ see Alg. 2.4
5: return  $I$ .
```

```

6: ASSIGNINITIALLABELS( $I$ )
   Performs a preliminary labeling on image  $I$  (which is modified).
   Returns the number of assigned labels ( $m$ ) and
   the set of detected label collisions ( $\mathcal{C}$ ).

7: Initialize  $m \leftarrow 2$  (the value of the next label to be assigned).
8:  $\mathcal{C} \leftarrow \{\}$                                          ▷ empty set of collisions
9: for  $v \leftarrow 0 \dots H - 1$  do                               ▷  $H$  = height of image  $I$ 
10:   for  $u \leftarrow 0 \dots W - 1$  do                         ▷  $W$  = width of image  $I$ 
11:     if  $I(u, v) = 1$  then do one of:
12:       if all neighbors of  $(u, v)$  are background pixels (all  $n_i = 0$ )
           then
13:          $I(u, v) \leftarrow m$ 
14:          $m \leftarrow m + 1$ 
15:       else if exactly one of the neighbors has a label value  $n_k > 1$ 
           then
16:         set  $I(u, v) \leftarrow n_k$ 
17:       else if several neighbors of  $(u, v)$  have label values  $n_j > 1$ 
           then
18:         Select one of them as the new label:
            $I(u, v) \leftarrow k \in \{n_j\}$ .
19:         for all other neighbors of  $(u, v)$  with label values  $n_i > 1$ 
           and  $n_i \neq k$  do
20:           Create a new label collision:  $\mathbf{c}_i = \langle n_i, k \rangle$ .
21:           Record the collision:  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{c}_i\}$ 

22: Remark: The image  $I$  now contains label values  $0, 2, \dots, m-1$ .
return  $(m, \mathcal{C})$ .
```

continued in Alg. 2.4 ▷

Algorithm 2.4 Sequential region labeling (Part 2).

```

1: RESOLVELABELCOLLISIONS( $m, \mathcal{C}$ )
   Resolves the label collisions contained in the set  $\mathcal{C}$ .
   Returns  $\mathcal{R}$ , a vector of sets that represents a partitioning
   of the complete label set into equivalent labels.

2: Let  $\mathcal{L} = \{2, 3, \dots, m - 1\}$  be the set of preliminary region labels.
3: Create a partitioning of  $\mathcal{L}$  as a vector of sets, one set for each label
   value:
    $\mathcal{R} \leftarrow [\mathcal{R}_2, \mathcal{R}_3, \dots, \mathcal{R}_{m-1}] = [\{2\}, \{3\}, \{4\}, \dots, \{m - 1\}],$ 
   so  $\mathcal{R}_i = \{i\}$  for all  $i \in \mathcal{L}$ .
4: for all collisions  $\langle a, b \rangle \in \mathcal{C}$  do
5:   Find in  $\mathcal{R}$  the sets  $\mathcal{R}_a, \mathcal{R}_b$ :
    $\mathcal{R}_a \leftarrow$  the set that currently contains label  $a$ 
    $\mathcal{R}_b \leftarrow$  the set that currently contains label  $b$ 
6:   if  $\mathcal{R}_a \neq \mathcal{R}_b$  ( $a$  and  $b$  are contained in different sets) then
7:     Merge sets  $\mathcal{R}_a$  and  $\mathcal{R}_b$  by moving all elements of  $\mathcal{R}_b$  to  $\mathcal{R}_a$ :
      $\mathcal{R}_a \leftarrow \mathcal{R}_a \cup \mathcal{R}_b, \quad \mathcal{R}_b \leftarrow \{\}$ 
   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  $\mathcal{R}$ .
8:   return  $\mathcal{R}$ .
```

```

9: RELABELIMAGE( $I, \mathcal{R}$ )
   Relabels the image  $I$  using the label partitioning in  $\mathcal{R}$ .
   The image  $I$  is modified.

10: for all image locations  $(u, v)$  do
11:   if  $I(u, v) > 1$  then            $\triangleright I(u, v)$  contains a region label
12:     Find the set  $\mathcal{R}_i$  in  $\mathcal{R}$  that contains the label  $I(u, v)$ 
13:     Choose one unique representative element  $k$  from the set  $\mathcal{R}_i$ ,
        e.g., the minimum value:
         $k = \min(\mathcal{R}_i)$ 
14:     Replace the image label:
         $I(u, v) \leftarrow k$ 
15:   return.
```

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

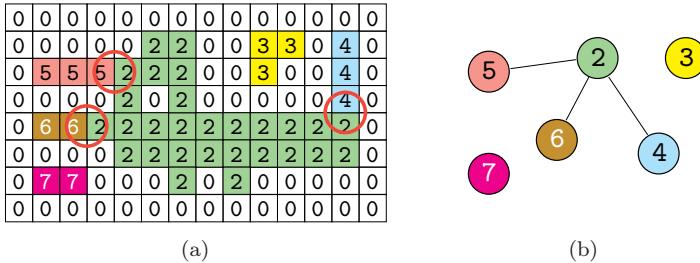


Figure 2.4 Sequential region labeling—intermediate result after Step 1. Label collisions indicated by circles (a); the nodes of the undirected graph (b) correspond to the labels, and its edges correspond to the collisions.

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. 2.3 (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 in the second step of the algorithm. The number of collisions depends on the content of the image. There can be only a few or very many collisions, and the exact number is only known at the end of the first step, once the whole image has been traversed. For this reason, collision management must make use of dynamic data structures such as lists or hash tables. Upon the completion of the first steps, all the original foreground pixels have been provisionally marked, and all the collisions between labels within the same regions have been registered for subsequent processing.

The example in Fig. 2.4 illustrates the state upon completion of step 1: all foreground pixels have been assigned preliminary labels (Fig. 2.4 (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. 2.4 (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 [20], where the labels \mathcal{L} determined in Step 1 constitute the “nodes” of the graph and the registered

collisions \mathcal{C} make up its “edges” (Fig. 2.4 (b)).

Step 3: Relabeling the image

Once all the distinct labels within a single region have been collected, the labels of all the pixels in the region are updated so they carry the same label (for example, choosing the smallest label number in the region), as shown in Fig. 2.5.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	2	2	0	0	3	3	0	2	0	
0	2	2	2	2	2	0	0	0	3	0	0	2	0	
0	0	0	0	2	0	2	0	0	0	0	0	2	0	
0	2	2	2	2	2	2	2	2	2	2	2	2	0	
0	0	0	0	2	2	2	2	2	2	2	2	2	0	
0	7	7	0	0	0	2	0	2	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	

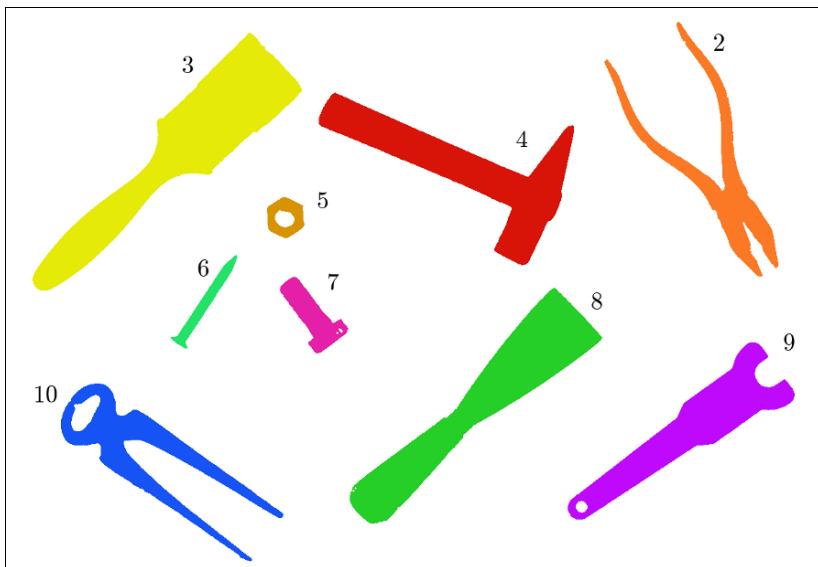
Figure 2.5 Sequential region labeling—final result after Step 3. All equivalent labels have been replaced by the smallest label within that region.

2.1.3 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 method (Sec. 2.1.1) was not useful because of practical limitations on the depth of recursion and the high memory costs associated with it. We also saw that classical sequential region labeling (Sec. 2.1.2) 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.

2.2 Region Contours

Once the regions in a binary image have been found, the next step is often to find the contours (that is, the outlines) of the regions. Like so many other tasks in image processing, at first glance this appears to be an easy one: simply follow along the edge of the region. We will see that, in actuality, describing this apparently simple process algorithmically requires careful thought, which has made contour finding one of the classic problems in image analysis.



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)

Figure 2.6 Example of a complete region labeling. The pixels within each region have been colored according to the consecutive label values 2, 3, … 10 they were assigned. The corresponding region statistics are shown in the table below (total image size is 1212×836).

2.2.1 External and Internal Contours

As we discussed in Vol. 1 [14, Sec. 7.2.7], 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, that this process only *marks* the pixels along the contour, which is useful, for instance, for display purposes. In this section, we will go one step further and develop an algorithm for obtaining an *ordered sequence* of border pixel coordinates for describing a region’s contour.

Note that connected image regions contain exactly one *outer* contour, yet, due to holes, they can contain arbitrarily many *inner* contours. Within such

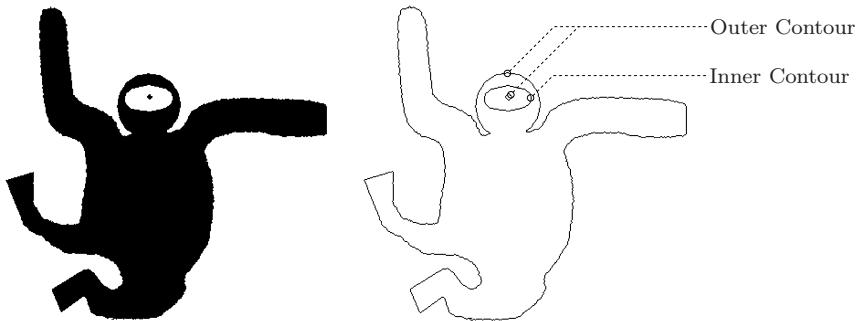


Figure 2.7 Binary image with outer and inner contours. The outer contour lies along the outside of the foreground region (dark). The inner contour surrounds the space within the region, which may contain further regions (holes), and so on.

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. 2.7).

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. 2.8). Therefore, when tracing a contour from a start point x_S , returning to the start point is *not* a sufficient condition for terminating the contour tracing process. Other factors, such as the current direction along which contour points are being traversed, must be taken into account.

One apparently simple way of determining a contour is to proceed in analogy to the two-stage process presented in the previous section (2.1); 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 the same way, an internal contour can be found by starting at a border pixel of a region's hole. A wide range of algorithms based on first finding the regions and then following along their contours have been published, including [61], [57, pp. 142–148], and [65, p. 296]. However, while the idea of contour tracing 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 a modern 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.

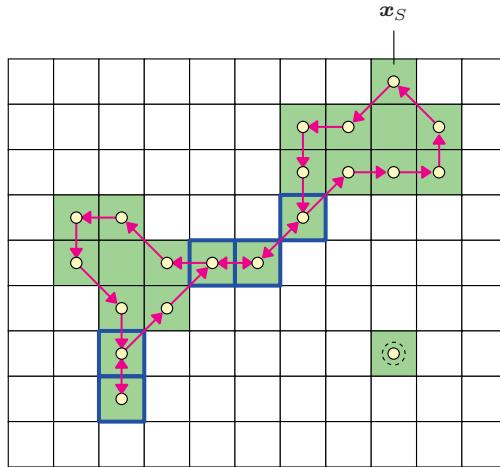


Figure 2.8 The path along a contour as an ordered sequence of pixel coordinates with a given start point x_S . Individual pixels may occur (be visited) more than once within the path, and a region consisting of a single isolated pixel will also have a contour (bottom right).

2.2.2 Combining Region Labeling and Contour Finding

This method, based on [18], combines the concepts of sequential region labeling (Sec. 2.1) and traditional contour tracing into a single algorithm able to perform both tasks simultaneously during a single pass through the image. It identifies and labels regions and at the same time traces both their inner and outer contours. The algorithm does not require any complicated data structures and is very efficient when compared with other methods with similar capabilities. The key steps of this method are described below and illustrated in Fig. 2.9:

1. As in the sequential region labeling (Alg. 2.3), 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 in Fig. 2.9 (a)) means that this pixel lies on the outer edge of a new region. A new *label* is assigned and the associated *outer* contour is traversed and marked by calling the method `TRACECONTOUR()` (see Fig. 2.9 (a) and Alg. 2.5 (line 19)). Furthermore, all background pixels directly bordering the region are marked with the special label -1 .

Case B: The transition from a foreground pixel (B in Fig. 2.9 (b)) to an

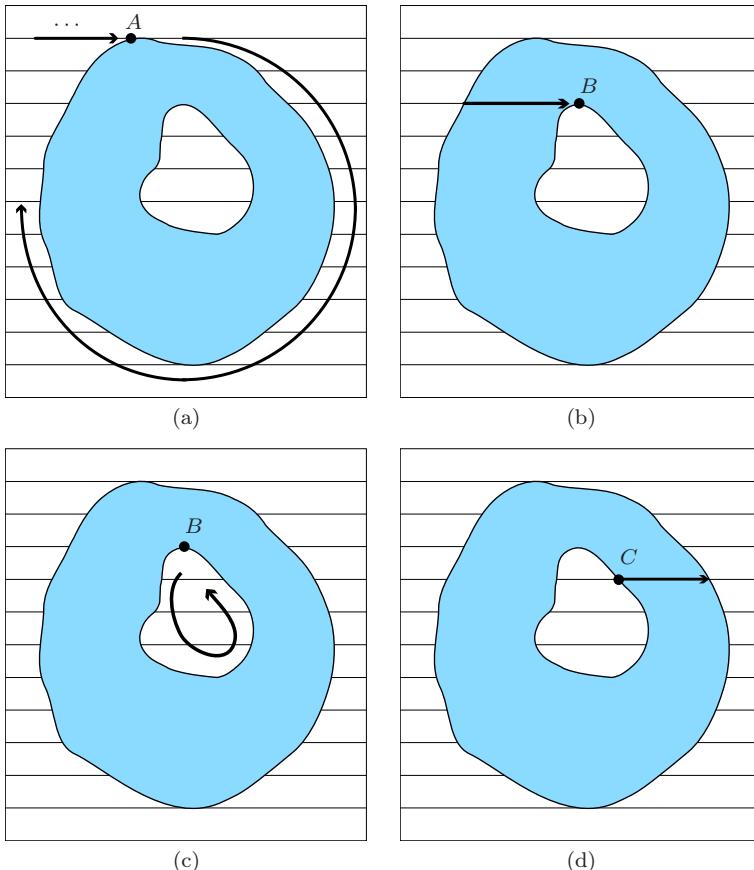


Figure 2.9 Combined region labeling and contour following (after [18]). The image is traversed from the top left to the lower right a row at a time. In (a), the first point A on the outer edge of the region is found. Starting from point A , the pixels on the edge along the outer contour are visited and 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.

unmarked background pixel means that this pixel lies on an *inner* contour. Starting from B , the inner contour is traversed and its pixels are marked with labels from the surrounding region (Fig. 2.9 (c)). Also, all bordering background pixels are again assigned the special label value -1 .

Case C: When a foreground pixel does not lie on a contour, then the neighboring pixel to the left has already been labeled (Fig. 2.9 (d)) and this label is propagated to the current pixel.

In Algorithms 2.5 and 2.6, 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” L (a rectangular array of the same size as the image) by the method `FINDNEXTPOINT()` in Alg. 2.6.

2.2.3 Implementation

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 B (pp. 283–293). The implementation closely follows the description in Algs. 2.5 and 2.6 but illustrates several additional details:⁴

- The task is performed by methods of the class `ContourTracer`. First the image I (`pixelArray`) and the associated label map L (`labelArray`) are enlarged by padding one layer of elements around their 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 turned into objects of class `Contour` and collected in two separate lists: `outerContours` and `innerContours`. Every contour consists of an ordered sequence of coordinate points of the standard class `Point` (defined in `java.awt`). The Java container class `ArrayList` (templated on the type `Point`) is used as a dynamic data structure for storing the point sequences of the outer and inner contours.
- The method `traceContour()` (see p. 289) traverses an outer or inner contour, beginning from the starting point x_S (xS , yS). It calls the method `findNextPoint()`, to determine the next contour point x_T (xT , yT) following x_S :
 - In the case that no following point is found, then $x_S = x_T$ 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 x_c (xC , yC) and the *previous* point x_p (xP , yP) are recorded. Only when *both* points correspond to the original starting

⁴ In the following description the names in parentheses after the algorithmic symbols denote the corresponding identifiers used in the Java implementation.

Algorithm 2.5 Combined contour tracing and region labeling (Part 1). 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 19 and line 26). TRACECONTOUR() itself is described in Alg. 2.6.

```

1: COMBINEDCONTOURLABELING ( $I$ )
    $I$ : binary image.
   Returns the sets of outer and inner contours and a label map.

2:  $\mathcal{C}_{\text{outer}} \leftarrow \{\}, \quad \mathcal{C}_{\text{inner}} \leftarrow \{\}$             $\triangleright$  create two empty sets of contours
3: Create a label map  $L$  of the same size as  $I$  and initialize:
4: for all image locations  $(u, v)$  do
5:    $L(u, v) \leftarrow 0$                                       $\triangleright$  label map  $L$ 
6:    $R \leftarrow 0$                                           $\triangleright$  region counter  $R$ 

   Scan the image from left to right and top to bottom:
7: for  $v \leftarrow 0 \dots N-1$  do
8:    $l \leftarrow 0$                                           $\triangleright$  set the current label  $l$  to “none”
9:   for  $u \leftarrow 0 \dots M-1$  do
10:    if  $I(u, v)$  is a foreground pixel then
11:      if  $(l \neq 0)$  then                                 $\triangleright$  continue inside region
12:         $L(u, v) \leftarrow l$ 
13:      else
14:         $l \leftarrow L(u, v)$ 
15:        if  $(l = 0)$  then                                 $\triangleright$  hit a new outer contour
16:           $R \leftarrow R + 1$ 
17:           $l \leftarrow R$ 
18:           $\mathbf{x}_S \leftarrow (u, v)$ 
19:           $\mathbf{c} \leftarrow \text{TRACECONTOUR}(\mathbf{x}_S, 0, l, I, L)$ 
20:           $\mathcal{C}_{\text{outer}} \leftarrow \mathcal{C}_{\text{outer}} \cup \{\mathbf{c}\}$         $\triangleright$  collect outer contour
21:           $L(u, v) \leftarrow l$ 
22:        else                                          $\triangleright I(u, v)$  is a background pixel
23:          if  $(l \neq 0)$  then
24:            if  $(L(u, v) = 0)$  then                       $\triangleright$  hit new inner contour
25:               $\mathbf{x}_S \leftarrow (u-1, v)$ 
26:               $\mathbf{c} \leftarrow \text{TRACECONTOUR}(\mathbf{x}_S, 1, l, I, L)$ 
27:               $\mathcal{C}_{\text{inner}} \leftarrow \mathcal{C}_{\text{inner}} \cup \{\mathbf{c}\}$         $\triangleright$  collect inner contour
28:               $l \leftarrow 0$ 
29:    return  $(\mathcal{C}_{\text{outer}}, \mathcal{C}_{\text{inner}}, L)$ .     $\triangleright$  return the contour sets and label map

```

continued in Alg. 2.6 \bowtie

Algorithm 2.6 Combined contour finding and region labeling (Part 2, continued from Alg. 2.5). Starting from \mathbf{x}_S , the procedure TRACECONTOUR traces along the contour in the direction $d_S = 0$ for outer contours or $d_S = 1$ for inner contours. During this process, all contour points as well as neighboring background points are marked in the label array L . Given a point \mathbf{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($\mathbf{x}_S, d_S, l, I, L$)																												
	\mathbf{x}_S : start position,																												
	d_S : initial search direction (0 for outer, 1 for inner contours),																												
	l : label for this contour, I : original image, L : label map.																												
	Traces and returns the contour starting at \mathbf{x}_S .																												
2:	$(\mathbf{x}_T, d_{\text{next}}) \leftarrow \text{FINDNEXTPOINT}(\mathbf{x}_S, d_S, I, L)$																												
3:	$\mathbf{c} \leftarrow [\mathbf{x}_T]$	▷ create a contour starting with \mathbf{x}_T																											
4:	$\mathbf{x}_p \leftarrow \mathbf{x}_S$	▷ previous position $\mathbf{x}_p = (u_p, v_p)$																											
5:	$\mathbf{x}_c \leftarrow \mathbf{x}_T$	▷ current position $\mathbf{x}_c = (u_c, v_c)$																											
6:	$done \leftarrow (\mathbf{x}_S \equiv \mathbf{x}_T)$	▷ isolated pixel?																											
7:	while ($\neg done$) do																												
8:	$L(u_c, v_c) \leftarrow l$																												
9:	$d_{\text{search}} \leftarrow (d_{\text{next}} + 6) \bmod 8$																												
10:	$(\mathbf{x}_n, d_{\text{next}}) \leftarrow \text{FINDNEXTPOINT}(\mathbf{x}_c, d_{\text{search}}, I, L)$																												
11:	$\mathbf{x}_p \leftarrow \mathbf{x}_c$																												
12:	$\mathbf{x}_c \leftarrow \mathbf{x}_n$																												
13:	$done \leftarrow (\mathbf{x}_p \equiv \mathbf{x}_S \wedge \mathbf{x}_c \equiv \mathbf{x}_T)$	▷ back at start point?																											
14:	if ($\neg done$) then																												
15:	APPEND(\mathbf{c}, \mathbf{x}_n)	▷ add point \mathbf{x}_n to contour \mathbf{c}																											
16:	return \mathbf{c} .	▷ return this contour																											
17:	FINDNEXTPOINT(\mathbf{x}_c, d, I, L)																												
	\mathbf{x}_c : start point, d : search direction,																												
	I : original image, L : label map.																												
18:	for $i \leftarrow 0 \dots 6$ do	▷ search in 7 directions																											
19:	$\mathbf{x}' \leftarrow \mathbf{x}_c + \text{DELTA}(d)$	▷ $\mathbf{x}' = (u', v')$																											
20:	if $I(u', v')$ is a background pixel then																												
21:	$L(u', v') \leftarrow -1$	▷ mark background as visited (-1)																											
22:	$d \leftarrow (d + 1) \bmod 8$																												
23:	else	▷ found a nonbackground pixel at \mathbf{x}'																											
24:	return (\mathbf{x}', d)																												
25:	return (\mathbf{x}_c, d) .	▷ found no next point, return start point																											
26:	$\text{DELTA}(d) = (\Delta x, \Delta y)$, with	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>d</th> <th>0</th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> <th>6</th> <th>7</th> </tr> </thead> <tbody> <tr> <td>Δx</td> <td>1</td> <td>1</td> <td>0</td> <td>-1</td> <td>-1</td> <td>-1</td> <td>0</td> <td>1</td> </tr> <tr> <td>Δy</td> <td>0</td> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>-1</td> <td>-1</td> <td>-1</td> </tr> </tbody> </table>	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
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																					

```

1 import java.util.List;
2 ...
3 public class Trace_Contours implements PlugInFilter {
4     public void run(ImageProcessor ip) {
5         ContourTracer tracer = new ContourTracer(ip);
6         // extract contours and regions
7         List<Contour> outerContours = tracer.getOuterContours();
8         List<Contour> innerContours = tracer.getInnerContours();
9         List<BinaryRegion> regions = tracer.getRegions();
10    ...
11 }
12 }
```

Program 2.2 Example of using the class `ContourTracer`. See Appendix B.1 for a listing of the complete implementation.

points on the contour, $\mathbf{x}_p = \mathbf{x}_S$ and $\mathbf{x}_c = \mathbf{x}_T$, we know that the contour has been completely traversed.

- The method `findNextPoint()` (see p. 290) determines which point on the contour follows the current point \mathbf{x}_c (x_C , y_C) 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 L* (`labelArray`) are marked with the value -1 to prevent them from being searched again. If no valid contour point is found among the seven possible neighbors, then `findNextPoint()` returns the original point \mathbf{x}_c (x_C , y_C).

In this implementation the core of the algorithm is contained in the class `ContourTracer` (pp. 287–292). Program 2.2 provides an example of its usage within the `run()` method of an ImageJ plugin. An interesting detail is the class `ContourOverlay` (pp. 292–293) 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).

2.2.4 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 2.10 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

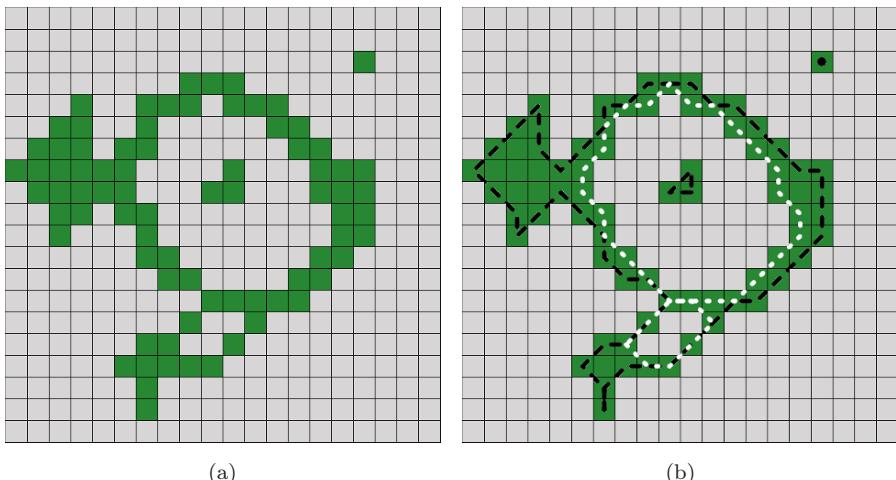


Figure 2.10 Combined contour and region marking: original image in gray (a), located contours (b) with black lines for out and white lines for inner contours. The contour consisting of single isolated pixels (for example, in the upper-right of (b)) are marked by a single circle in the appropriate color.

running through the centers of the contour pixels, and inner contours are drawn white. Contours of single-pixel regions are marked by small circles filled with the corresponding color. Figure 2.11 shows the results for a larger section taken from a real image (Vol. 1 [14, Fig. 7.12]).

2.3 Representing Image Regions

2.3.1 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 programming languages, to a simple and elegant mapping onto two-dimensional arrays, which makes possible a very natural way to work with raster images. One possible disadvantage with this representation is that it does not depend on the content of the image. In other words, it makes no difference whether the image contains only a pair of lines or is of a complex scene because the amount of memory required is constant and depends only on the dimensions of the image.

Regions in an image can be represented using a logical mask in which the area within the region is assigned the value *true* and the area without the value *false* (Fig. 2.12). Since Boolean values can be represented by a single bit, such

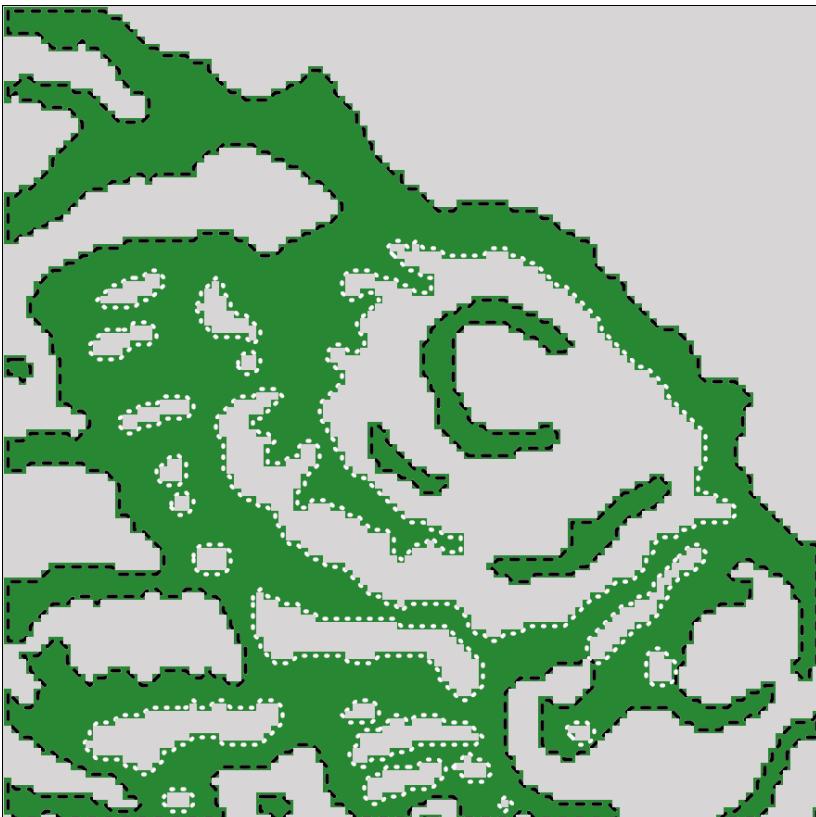


Figure 2.11 Example of a complex contour (in a section cut from Fig. 7.12 in Vol. 1 [14]). Outer contours are marked in black and inner contours in white.

a matrix is often referred to as a “bitmap”.⁵

2.3.2 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,

$$\text{Run}_i = \langle \text{row}_i, \text{column}_i, \text{length}_i \rangle,$$

⁵ In Java, variables of the type `boolean` are represented internally within the Java virtual machine (JVM) as 32-bit `ints`. There is currently no direct way to implement genuine bitmaps in Java.

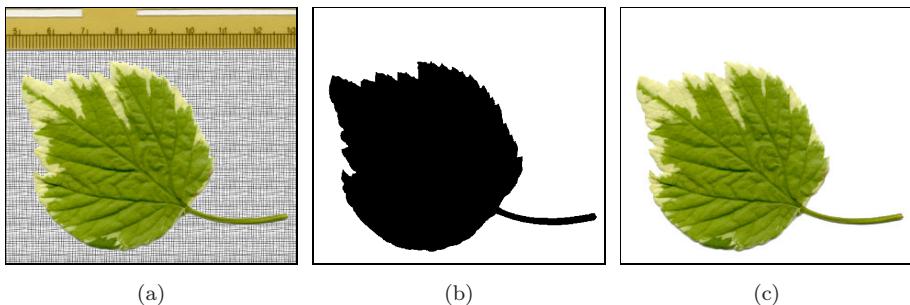


Figure 2.12 Use of a binary mask to specify a region of an image: original image (a), logical (bit) mask (b), and masked image (c).

Bitmap									RLE
	0	1	2	3	4	5	6	7	$\langle \text{row}, \text{column}, \text{length} \rangle$
0									
1			x	x	x	x	x	x	$\langle 1, 2, 6 \rangle$
2									$\langle 3, 4, 4 \rangle$
3				x	x	x	x		$\langle 4, 1, 3 \rangle$
4	x	x	x		x	x	x		$\langle 4, 5, 3 \rangle$
5	x	x	x	x	x	x	x	x	$\langle 5, 0, 9 \rangle$
6									

→

Figure 2.13 Run length encoding in row direction. A run of pixels can be represented by its starting point (1, 2) and its length (6).

two to represent the starting pixel (row, column) and a third for the length of the run as illustrated in Fig. 2.13. 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 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. 2.4.3).

2.3.3 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 [25], are a classical method of contour encoding. In this encoding, the contour beginning

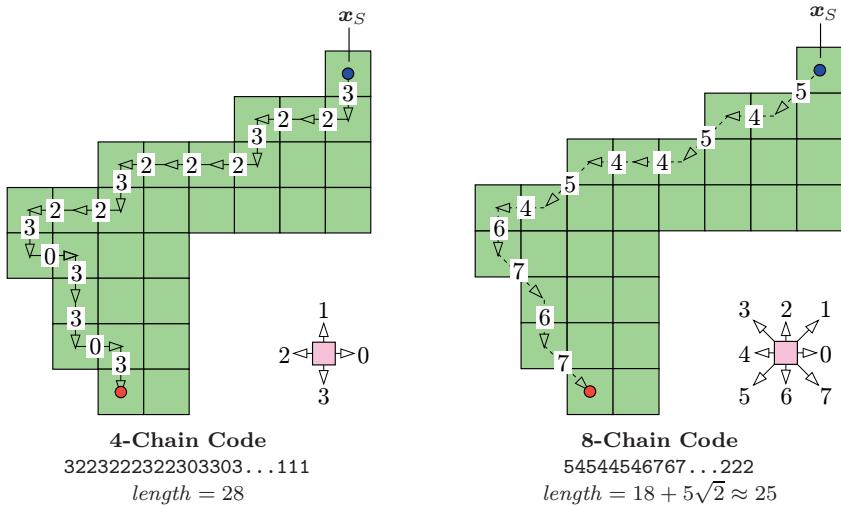


Figure 2.14 Chain codes with 4- and 8-connected neighborhoods. To compute a chain code, begin traversing the contour from a given starting point x_S . Encode the relative position between adjacent contour points using the directional code for either 4-connected (left) or 8-connected (right) neighborhoods. The length of the resulting path, calculated as the sum of the individual segments, can be used to approximate the true length of the contour.

at a given start point x_S is represented by the sequence of directional changes it describes on the discrete image raster (Fig. 2.14).

Absolute chain code

For a closed contour of a region \mathcal{R} , described by the sequence of points $\mathbf{c}_{\mathcal{R}} = [\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{M-1}]$ with $\mathbf{x}_i = \langle u_i, v_i \rangle$, we create the elements of its chain code sequence $\mathbf{c}'_{\mathcal{R}} = [c'_0, c'_1, \dots, c'_{M-1}]$ by

$$c'_i = \text{CODE}(\Delta u_i, \Delta v_i), \quad (2.1)$$

$$\text{where } (\Delta u_i, \Delta v_i) = \begin{cases} (u_{i+1} - u_i, v_{i+1} - v_i) & \text{for } 0 \leq i < M-1 \\ (u_0 - u_i, v_0 - v_i) & \text{for } i = M-1, \end{cases}$$

and $\text{CODE}(\Delta u, \Delta v)$ being defined by the following table:⁶

Δu	1	1	0	-1	-1	-1	0	1
Δv	0	1	1	1	0	-1	-1	-1
CODE($\Delta u, \Delta v$)	0	1	2	3	4	5	6	7

⁶ Assuming an 8-connected neighborhood.

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 \mathbf{x}_S , and for instance simply rotating the region by 90° results in a completely different chain code. When using a *differential* chain code, the situation improves slightly. Instead of encoding the difference in the *position* of the next contour point, the change in the *direction* along the discrete contour is encoded. A given *absolute* chain code $\mathbf{c}'_{\mathcal{R}} = [c'_0, c'_1, \dots, c'_{M-1}]$ can be converted element by element to a *differential* chain code $\mathbf{c}''_{\mathcal{R}} = [c''_0, c''_1, \dots, c''_{M-1}]$, with

$$c''_i = \begin{cases} (c'_{i+1} - c'_i) \bmod 8 & \text{for } 0 \leq i < M-1 \\ (c'_0 - c'_i) \bmod 8 & \text{for } i = M-1, \end{cases} \quad (2.2)$$

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 $\mathbf{c}'_{\mathcal{R}}$. For the contour in Fig. 2.14 (b), the results are

$$\begin{aligned} \mathbf{c}'_{\mathcal{R}} &= [5, 4, 5, 4, 4, 5, 4, 6, 7, 6, 7, \dots, 2, 2, 2], \\ \mathbf{c}''_{\mathcal{R}} &= [7, 1, 7, 0, 1, 7, 2, 1, 7, 1, 1, \dots, 0, 0, 3]. \end{aligned}$$

Given the starting point \mathbf{x}_S and the (absolute) initial direction c_0 , the original contour can be unambiguously reconstructed from the differential chain code.

Shape numbers

While the differential chain code remains the same when a region is rotated by 90° , the encoding is still dependent on the selected starting point. If we want to determine the similarity of two contours of the same length M using their differential chain codes $\mathbf{c}''_1, \mathbf{c}''_2$, we must first ensure that the same start point was used when computing the codes. A method that is often used [2, 28] is to interpret the elements c''_i in the differential chain code as the digits of

⁷ See Vol. 1 [14, Appendix B.1.2] for implementing the mod operator used in Eqn. (2.2).

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

$$\begin{aligned} \text{VAL}(\mathbf{c}_R'') &= c_0'' \cdot b^0 + c_1'' \cdot b^1 + \dots + c_{M-1}'' \cdot b^{M-1} \\ &= \sum_{i=0}^{M-1} c_i'' \cdot b^i. \end{aligned} \quad (2.3)$$

Then the sequence \mathbf{c}_R'' is shifted cyclically until the numeric value of the corresponding number reaches a maximum. We use the expression $\mathbf{c}_R'' \triangleright k$ to denote the sequence \mathbf{c}_R'' being cyclically shifted by k positions to the right,⁸ such as (for $k = 2$)

$$\begin{aligned} \mathbf{c}_R'' &= [0, 1, 3, 2, \dots, 9, 3, 7, 4] \\ \mathbf{c}_R'' \triangleright 2 &= [7, 4, 0, 1, 3, 2, \dots, 9, 3] \end{aligned}$$

and

$$k_{\max} = \arg \max_{0 \leq k < M} \text{VAL}(\mathbf{c}_R'' \triangleright k) \quad (2.4)$$

to denote the shift required to maximize the corresponding arithmetic value. The resulting code sequence or *shape number*,

$$\mathbf{s}_R = \mathbf{c}_R'' \triangleright k_{\max}, \quad (2.5)$$

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 $\text{VAL}()$ in Eqn. (2.3) produces values that are in general too large to be actually computed, in practice the relation

$$\text{VAL}(\mathbf{c}_1'') > \text{VAL}(\mathbf{c}_2'')$$

is determined by comparing the *lexicographic ordering* between the sequences \mathbf{c}_1'' and \mathbf{c}_2'' so that the arithmetic values need not be computed at all.

Unfortunately, comparisons based on chain codes are generally not very useful for determining the similarity between regions simply because rotations at arbitrary angles ($\neq 90^\circ$) have too great of an impact (change) on a region's code. In addition, chain codes are not capable of handling changes in size (scaling) or other distortions. Section 2.4 presents a number of tools that are more appropriate in these types of cases.

⁸ $(\mathbf{c}_R'' \triangleright k)[i] = \mathbf{c}_R''[(i - k) \bmod M]$.

Fourier descriptors

An elegant approach to describing contours are so-called Fourier descriptors, which interpret the two-dimensional contour $\mathbf{c}_{\mathcal{R}} = [\mathbf{x}_0, \mathbf{x}_1, \dots \mathbf{x}_{M-1}]$ with $\mathbf{x}_i = (u_i, v_i)$ as a sequence of values $[z_0, z_1 \dots z_{M-1}]$ in the complex plane, where

$$z_i = (u_i + i \cdot v_i) \in \mathbb{C}. \quad (2.6)$$

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. 7.3) 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 [28, 30, 46, 47, 69].

2.4 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 [21, 55, 72].

2.4.1 Shape Features

The comparison and classification of binary regions is widely used, for example, in optical character recognition (OCR) and for automating processes ranging from blood cell counting to quality control inspection of manufactured products on assembly lines. The analysis of binary regions turns out to be one of the simpler tasks for which many efficient algorithms have been developed and used to implement reliable applications that are in use every day.

By a *feature* of a region, we mean a specific numerical or qualitative measure that is computable from the values and coordinates of the pixels that make up

the region. As an example, one of the simplest features is its *size* or *area*; that is the number of pixels that make up a region. In order to describe a region in a compact form, different features are often combined into a *feature vector*. This vector is then used as a sort of “signature” for the region that can be used for classification or comparison with other regions. The best features are those that are simple to calculate and are not easily influenced (robust) by irrelevant changes, particularly translation, rotation, and scaling.

2.4.2 Geometric Features

A region \mathcal{R} of a binary image can be interpreted as a two-dimensional distribution of foreground points $\mathbf{x}_i = (u_i, v_i)$ on the discrete plane \mathbb{Z}^2 ,

$$\mathcal{R} = \{\mathbf{x}_0, \mathbf{x}_1 \dots \mathbf{x}_{N-1}\} = \{(u_0, v_0), (u_1, v_1) \dots (u_{N-1}, v_{N-1})\}.$$

Most geometric properties are defined in such a way that a region is considered to be a set of pixels that, in contrast to the definition in Sec. 2.1, 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. 2.14, the type of neighborhood relation must be taken into account for this calculation. When using a 4-neighborhood, the measured length of the contour (except when that length is 1) will be larger than its actual length. In the case of 8-neighborhoods, a good approximation is reached by weighing the horizontal and vertical segments with 1 and diagonal segments with $\sqrt{2}$. Given an 8-connected chain code $\mathbf{c}'_{\mathcal{R}} = [c'_0, c'_1, \dots c'_{M-1}]$, the perimeter of the region is arrived at by

$$\text{Perimeter}(\mathcal{R}) = \sum_{i=0}^{M-1} \text{length}(c'_i), \quad (2.7)$$

$$\text{with } \text{length}(c) = \begin{cases} 1 & \text{for } c = 0, 2, 4, 6, \\ \sqrt{2} & \text{for } c = 1, 3, 5, 7. \end{cases}$$

However, with this conventional method of calculation, the *real* perimeter ($P(\mathcal{R})$) is systematically overestimated. As a simple remedy, an empirical correction factor of 0.95 works satisfactorily even for relatively small regions:

$$P(\mathcal{R}) \approx \text{Perimeter}_{\text{corr}}(\mathcal{R}) = 0.95 \cdot \text{Perimeter}(\mathcal{R}). \quad (2.8)$$

Area

The area of a binary region \mathcal{R} can be found by simply counting the image pixels that make up the region,

$$A(\mathcal{R}) = |\mathcal{R}| = N. \quad (2.9)$$

The area of a connected region without holes can also be approximated from its closed contour, defined by M coordinate points $(\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{M-1})$, where $\mathbf{x}_i = (u_i, v_i)$, using the Gaussian area formula for polygons:

$$A(\mathcal{R}) \approx \frac{1}{2} \cdot \left| \sum_{i=0}^{M-1} (u_i \cdot v_{(i+1) \bmod M} - u_{(i+1) \bmod M} \cdot v_i) \right|. \quad (2.10)$$

When the contour is already encoded as a chain code $\mathbf{c}'_{\mathcal{R}} = [c'_0, c'_1, \dots, c'_{M-1}]$, then the region's area can be computed using Eqn. (2.10) by expanding $\mathbf{c}'_{\mathcal{R}}$ 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,

$$\text{Circularity}(\mathcal{R}) = 4\pi \cdot \frac{A(\mathcal{R})}{P^2(\mathcal{R})}, \quad (2.11)$$

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. 2.15). If an absolute value for a region's roundness is required, the corrected perimeter estimate (Eqn. (2.8)) should be employed:

$$\text{Circularity}(\mathcal{R}) \approx 4\pi \cdot \frac{A(\mathcal{R})}{\text{Perimeter}_{\text{corr}}^2(\mathcal{R})}. \quad (2.12)$$

Figure 2.15 shows the circularity values of different regions as computed with the formulation in Eqn. (2.12).

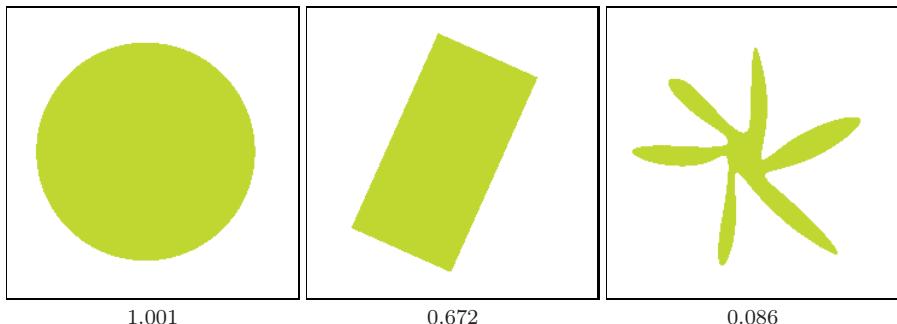


Figure 2.15 Circularity values for different shapes. Shown are the corresponding estimates for $\text{Circularity}(\mathcal{R})$ as defined in Eqn. (2.12).

Bounding box

The bounding box of a region \mathcal{R} is the minimal axis-parallel rectangle that encloses all points of \mathcal{R} ,

$$\text{BoundingBox}(\mathcal{R}) = \langle u_{\min}, u_{\max}, v_{\min}, v_{\max} \rangle, \quad (2.13)$$

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. 2.16 (a)).

Convex hull

The convex hull is the smallest convex polygon that contains all points of the region \mathcal{R} . A physical analogy is a board in which nails stick out in correspondence to each of the points in the region. If you were to place an elastic band around *all* the nails, then, when you release it, it will contract into a convex hull around the nails (Fig. 2.16 (b)). The convex hull can be computed for N contour points in time $\mathcal{O}(N \log V)$, where V is the number vertices in the polygon of the resulting convex hull [3].⁹

The convex hull is useful, for example, for determining the convexity or the *density* of a region. The *convexity* is defined as the relationship between the length of the convex hull and the original perimeter of the region. *Density* is then defined as the ratio between the area of the region and the area of its convex hull. The *diameter*, on the other hand, is the maximal distance between any two nodes on the convex hull.

⁹ For $\mathcal{O}()$ complexity notation, see Vol. 1 [14, Appendix A.3].

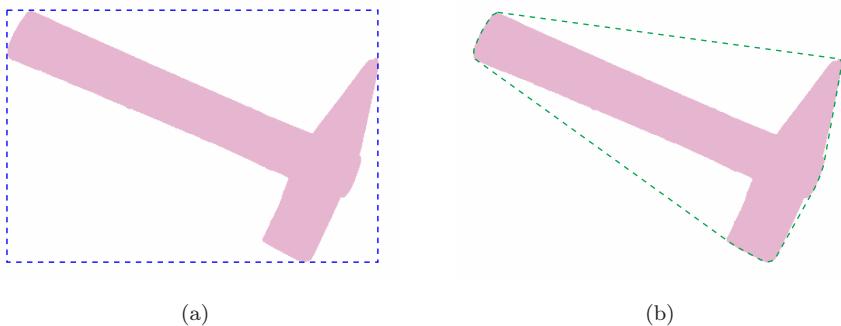


Figure 2.16 Example bounding box (a) and convex hull (b) of a binary image region.

2.4.3 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.¹⁰

The centroid $\bar{\mathbf{x}} = (\bar{x}, \bar{y})$ of a binary (not necessarily connected) region is the arithmetic mean of the coordinates in the x and y directions,

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v) \in \mathcal{R}} u \quad \text{and} \quad \bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v) \in \mathcal{R}} v . \quad (2.14)$$

¹⁰ 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.

Moments

The formulation of the region's centroid in Eqn. (2.14) is only a special case of the more general statistical concept of a *moment*. Specifically, the expression

$$m_{pq} = \sum_{(u,v) \in \mathcal{R}} I(u,v) \cdot u^p v^q \quad (2.15)$$

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 [64, 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. (2.15) can be simplified to

$$m_{pq} = \sum_{(u,v) \in \mathcal{R}} u^p v^q. \quad (2.16)$$

In this way, the *area* of a binary region can be expressed as its *zero-order* moment,

$$A(\mathcal{R}) = |\mathcal{R}| = \sum_{(u,v) \in \mathcal{R}} 1 = \sum_{(u,v) \in \mathcal{R}} u^0 v^0 = m_{00}(\mathcal{R}), \quad (2.17)$$

and similarly the *centroid* \bar{x} Eqn. (2.14) as

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v) \in \mathcal{R}} u^1 v^0 = \frac{m_{10}(\mathcal{R})}{m_{00}(\mathcal{R})}, \quad (2.18)$$

$$\bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v) \in \mathcal{R}} u^0 v^1 = \frac{m_{01}(\mathcal{R})}{m_{00}(\mathcal{R})}. \quad (2.19)$$

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 $\bar{x} = (\bar{x}, \bar{y})$ to obtain the *central* moments of order p, q :

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v) \in \mathcal{R}} I(u, v) \cdot (u - \bar{x})^p \cdot (v - \bar{y})^q. \quad (2.20)$$

For a binary image (with $I(u, v) = 1$ within the region \mathcal{R}), Eqn. (2.20) can be simplified to

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v) \in \mathcal{R}} (u - \bar{x})^p \cdot (v - \bar{y})^q. \quad (2.21)$$

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

$$s^{(p+q+2)}. \quad (2.22)$$

Thus size-invariant “normalized” moments are obtained by scaling with the reciprocal of the area $\mu_{00} = m_{00}$ raised to the required power in the form

$$\bar{\mu}_{pq}(\mathcal{R}) = \mu_{pq}(\mathcal{R}) \cdot \left(\frac{1}{\mu_{00}(\mathcal{R})} \right)^{(p+q+2)/2} \quad (2.23)$$

for $(p + q) \geq 2$ [46, p. 529].

Program 2.3 gives a direct (brute force) Java implementation for computing the ordinary, central, and normalized central moments for binary images (`BACKGROUND = 0`). This implementation is only meant to clarify the computation, and naturally much more efficient implementations are possible (see, for example, [48]).

2.4.4 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. 2.18(a)). Since rotating the region around the major axis requires less effort (smaller moment of inertia) than spinning it around any other axis, it is sometimes referred to as the major axis of rotation. As an example, when you hold a

```

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

```

Program 2.3 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. (2.16), (2.21), and (2.23)).

pencil between your hands and twist it around its major axis (that is, around the lead), the pencil exhibits the least mass inertia (Fig. 2.17). 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

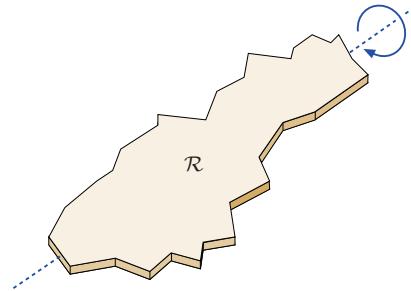


Figure 2.17 Major axis of a region. Rotating an elongated region \mathcal{R} , interpreted as a physical body, around its major axis requires less effort (least moment of inertia) than rotating it around any other axis.

$$\tan(2\theta_{\mathcal{R}}) = \frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})} \quad (2.24)$$

and therefore

$$\theta_{\mathcal{R}} = \frac{1}{2} \tan^{-1} \left(\frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})} \right) \quad (2.25)$$

$$= \frac{\text{Arctan}(2 \cdot \mu_{11}(\mathcal{R}), \mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R}))}{2}. \quad (2.26)$$

The resulting angle $\theta_{\mathcal{R}}$ is in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.¹¹ Orientation measurements based on region moments are very accurate in general.

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 $\bar{x} = (\bar{x}, \bar{y})$; for example, by a parametric line of the form

$$\mathbf{x} = \bar{x} + \lambda \cdot \mathbf{x}_d = \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} + \lambda \cdot \begin{pmatrix} \cos(\theta_{\mathcal{R}}) \\ \sin(\theta_{\mathcal{R}}) \end{pmatrix}, \quad (2.27)$$

for some length $\lambda > 0$. To find the unit orientation vector $\mathbf{x}_d = (\cos \theta, \sin \theta)^T$, we could first compute the inverse tangent to get 2θ (Eqn. (2.25)) and then compute the cosine and sine of θ . However, the vector \mathbf{x}_d can also be obtained without using trigonometric functions as follows. Rewriting Eqn. (2.24) as

$$\tan(2\theta_{\mathcal{R}}) = \frac{2 \cdot \mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})} = \frac{A}{B} = \frac{\sin(2\theta_{\mathcal{R}})}{\cos(2\theta_{\mathcal{R}})} \quad (2.28)$$

¹¹ See Appendix A.1 for the computation of angles with the `Arctan()` (inverse tangent) function and Vol. 1 [14, Appendix B.1.6] for the corresponding Java method `Math.atan2()`.

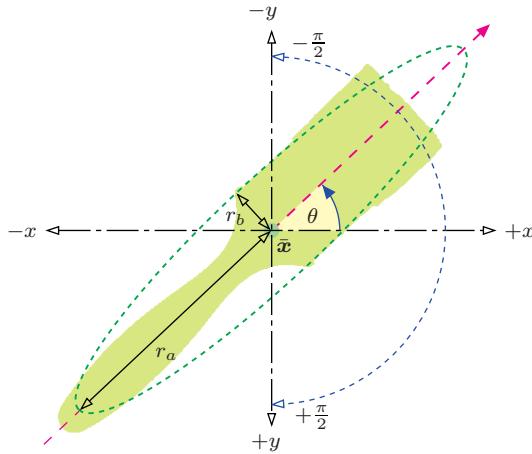


Figure 2.18 Region orientation and eccentricity. The major axis of the region extends through its center of gravity \bar{x} at the orientation θ . Note that angles are in the range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ and increment in the *clockwise* direction because the y axis of the image coordinate system points downward (in this example, $\theta \approx -0.759 \approx -43.5^\circ$). The eccentricity of the region is defined as the ratio between the lengths of the major axis (r_a) and the minor axis (r_b) of the “equivalent” ellipse.

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 $\mathbf{x}_d = (x_d, y_d)^T$ as

$$x_d = \cos(\theta_{\mathcal{R}}) = \begin{cases} 0 & \text{for } A = B = 0 \\ \left[\frac{1}{2} \left(1 + \frac{B}{\sqrt{A^2+B^2}} \right) \right]^{\frac{1}{2}} & \text{otherwise,} \end{cases} \quad (2.29)$$

$$y_d = \sin(\theta_{\mathcal{R}}) = \begin{cases} 0 & \text{for } A = B = 0 \\ \left[\frac{1}{2} \left(1 - \frac{B}{\sqrt{A^2+B^2}} \right) \right]^{\frac{1}{2}} & \text{for } A \geq 0 \\ -\left[\frac{1}{2} \left(1 - \frac{B}{\sqrt{A^2+B^2}} \right) \right]^{\frac{1}{2}} & \text{for } A < 0, \end{cases} \quad (2.30)$$

straight from the central region moments $\mu_{11}(\mathcal{R})$, $\mu_{20}(\mathcal{R})$, and $\mu_{02}(\mathcal{R})$, as defined in Eqn. (2.28). The horizontal component (x_d) in Eqn. (2.29) is always positive, while the case clause in Eqn. (2.30) corrects the sign of the vertical component (y_d) to map to the same angular range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ as Eqn. (2.25). The resulting vector \mathbf{x}_d is normalized (i.e., $\|(x_d, y_d)\| = 1$) and could be scaled

arbitrarily for display purposes by a suitable length λ , for example, using the region's eccentricity value described 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. (2.25)), 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. 2.4.2), and they can also handle nonconnected regions or point clouds. Several different formulations of region eccentricity can be found in the literature [2, 46, 47] (see also Exercise 2.11). We adopt the following definition because of its simple geometrical interpretation:

$$\text{Ecc}(\mathcal{R}) = \frac{a_1}{a_2} = \frac{\mu_{20} + \mu_{02} + \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}}{\mu_{20} + \mu_{02} - \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}}, \quad (2.31)$$

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

$$\mathbf{A} = \begin{pmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{pmatrix}$$

formed by the central moments μ_{pq} of the region \mathcal{R} . The values of Ecc are in the range $[1, \infty)$, where $\text{Ecc} = 1$ corresponds to a circular disk and elongated regions have values > 1 . 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. (2.25). The lengths of the ellipse’s major and minor axes, r_a and r_b , are

$$r_a = 2 \cdot \left(\frac{\lambda_1}{|\mathcal{R}|} \right)^{\frac{1}{2}} = \left(\frac{2a_1}{|\mathcal{R}|} \right)^{\frac{1}{2}}, \quad (2.32)$$

$$r_b = 2 \cdot \left(\frac{\lambda_2}{|\mathcal{R}|} \right)^{\frac{1}{2}} = \left(\frac{2a_2}{|\mathcal{R}|} \right)^{\frac{1}{2}}, \quad (2.33)$$

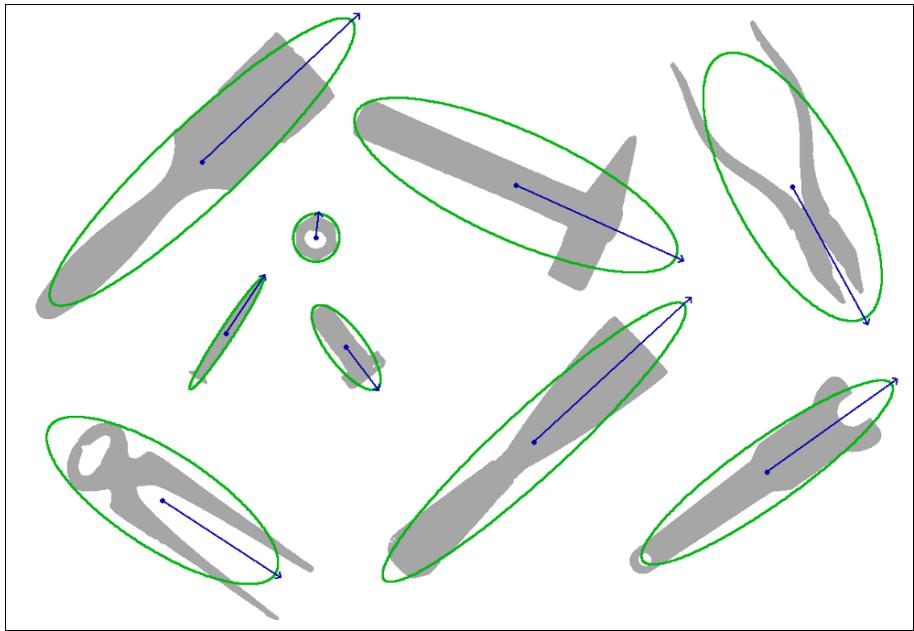


Figure 2.19 Orientation and eccentricity examples. The orientation θ (Eqn. (2.25)) is displayed for each connected region as a vector with the length proportional to the region's eccentricity value $Ecc(\mathcal{R})$ (Eqn. (2.31)). Also shown are the ellipses (Eqns. (2.32) and (2.33)) corresponding to the orientation and eccentricity parameters.

respectively, with a_1, a_2 as defined in Eqn. (2.31) and $|\mathcal{R}|$ being the number of pixels in the region. The resulting parametric equation of the equivalent ellipse is

$$\begin{aligned} \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} &= \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} + \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} r_a \cdot \cos(t) \\ r_b \cdot \sin(t) \end{pmatrix} \\ &= \begin{pmatrix} \bar{x} + \cos(\theta) \cdot r_a \cdot \cos(t) - \sin(\theta) \cdot r_b \cdot \sin(t) \\ \bar{y} + \sin(\theta) \cdot r_a \cdot \cos(t) + \cos(\theta) \cdot r_b \cdot \sin(t) \end{pmatrix} \end{aligned} \quad (2.34)$$

for $0 \leq t < 2\pi$. If entirely *filled*, the region described by this ellipse would have the same (first and second order) central moments as the original region \mathcal{R} . Figure 2.19 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” [37]:¹²

$$\begin{aligned}
 H_1 &= \bar{\mu}_{20} + \bar{\mu}_{02}, \\
 H_2 &= (\bar{\mu}_{20} - \bar{\mu}_{02})^2 + 4\bar{\mu}_{11}^2, \\
 H_3 &= (\bar{\mu}_{30} - 3\bar{\mu}_{12})^2 + (3\bar{\mu}_{21} - \bar{\mu}_{03})^2, \\
 H_4 &= (\bar{\mu}_{30} + \bar{\mu}_{12})^2 + (\bar{\mu}_{21} + \bar{\mu}_{03})^2, \\
 H_5 &= (\bar{\mu}_{30} - 3\bar{\mu}_{12}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot [(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3(\bar{\mu}_{21} + \bar{\mu}_{03})^2] \\
 &\quad + (3\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot [3(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2], \\
 H_6 &= (\bar{\mu}_{20} - \bar{\mu}_{02}) \cdot [(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2] \\
 &\quad + 4\bar{\mu}_{11} \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}), \\
 H_7 &= (3\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot [(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3(\bar{\mu}_{21} + \bar{\mu}_{03})^2] \\
 &\quad + (3\bar{\mu}_{12} - \bar{\mu}_{30}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot [3(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2].
 \end{aligned} \tag{2.35}$$

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 [28, p. 517].

2.4.5 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 vertical, projection of an image $I(u, v)$, with $0 \leq u < M$, $0 \leq v < N$, defined as

$$P_{\text{hor}}(v_0) = \sum_{u=0}^{M-1} I(u, v_0) \quad \text{for } 0 \leq v_0 < N, \tag{2.36}$$

$$P_{\text{ver}}(u_0) = \sum_{v=0}^{N-1} I(u_0, v) \quad \text{for } 0 \leq u_0 < M. \tag{2.37}$$

The *horizontal* projection $P_{\text{hor}}(v_0)$ (Eqn. (2.36)) is the sum of the pixel values in the image *row* v_0 and has length N corresponding to the height of the image. On the other hand, a *vertical* projection P_{ver} of length M is the sum of all the values in the image *column* u_0 (Eqn. (2.37)). 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.

¹² In order to improve the legibility of Eqn. (2.35) 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.

```

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 }
```

Program 2.4 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 single traversal of the image. The projections are represented by the one-dimensional arrays `horProj` and `verProj` with elements of type `int`.

Program Prog. 2.4 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. 2.20). In practice, especially to account for document skew, projections are often computed along the major axis of an image region Eqn. (2.25). 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.

2.4.6 Topological Properties

Topological features do not describe the shape of a region in continuous terms; instead, they capture its structural properties. They are typically invariant even under extreme image transformations. Two simple and robust topological features are the number of regions $N_R(\mathcal{R})$ and the number of holes $N_L(\mathcal{R})$ in those regions. $N_L(\mathcal{R})$ can be easily computed while finding the inner contours of a region, as described in Sec. 2.2.2.

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



Figure 2.20 Example of the horizontal projection $P_{\text{hor}}(v)$ (right) and vertical projection $P_{\text{ver}}(u)$ (bottom) of a binary image.

regions N_R and the number of their holes N_H ,

$$N_E(\mathcal{R}) = N_R(\mathcal{R}) - N_H(\mathcal{R}). \quad (2.38)$$

For a single connected region, the above formula simplifies to $1 - N_H$, so, for example, for an image of the number “8”, $N_E = 1 - 2 = -1$, while for an image of the letter “D”, $N_E = 1 - 1 = 0$.

Topological features are often used in combination with numerical features for classification, for example in optical character recognition (OCR) [12].

2.5 Exercises

Exercise 2.1

Trace, by hand, the execution of both variations (*depth-first* and *breadth-first*) of the flood-fill algorithm using the image shown in Fig. 2.21 and starting at coordinates $(5, 1)$.

Exercise 2.2

The implementation of the flood-fill algorithm in Prog. 2.1 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

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

Figure 2.21 Binary image for Exercise 2.1.

meet the two conditions given above. Modify the *depth-first* and *breadth-first* variants given in Prog. 2.1 accordingly and compare the new running times.

Exercise 2.3

Implement an ImageJ plugin that encodes a grayscale image using run length encoding (Sec. 2.3.2) and stores it in a file. Develop a second plugin that reads the file and reconstructs the image.

Exercise 2.4

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 2.5

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 2.6

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 2.7

Implement an algorithm for comparing contours using their shape numbers Eqn. (2.3). 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 2.8

Using Eqn. (2.10) 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 2.9

Sketch an example binary region where the centroid lies outside of the region.

Exercise 2.10

Implement the moment features developed by Hu (Eqn. (2.35)) and show that they are invariant under scaling and rotation for both binary and grayscale images.

Exercise 2.11

There are alternative definitions for the eccentricity of a region Eqn. (2.31); for example,

$$\begin{aligned} \text{Ecc}_2(\mathcal{R}) &= \frac{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}{(\mu_{20} + \mu_{02})^2} && [47, \text{ p. 394}], \\ \text{Ecc}_3(\mathcal{R}) &= \frac{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}}{m_{00}} && [46, \text{ p. 531}], \\ \text{Ecc}_4(\mathcal{R}) &= \frac{\sqrt{\mu_{20} - \mu_{02}} + 4 \cdot \mu_{11}}{m_{00}} && [2, \text{ p. 255}]. \end{aligned}$$

Implement all four variations (including the one in Eqn. (2.31)) 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 2.12

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. 2.19). Hint: Use Eqn. (2.34) to develop a method for drawing ellipses at arbitrary orientations (not available in ImageJ).

Exercise 2.13

The Java method in Prog. 2.4 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.

3

Detecting Simple Curves

In Volume 1 we demonstrated how to use appropriately designed filters to detect edges in images [14, Chap. 6]. 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 and so are missing from the map (false negatives).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.

3.1 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

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, fill in the missing elements eliminated during low-level processing.

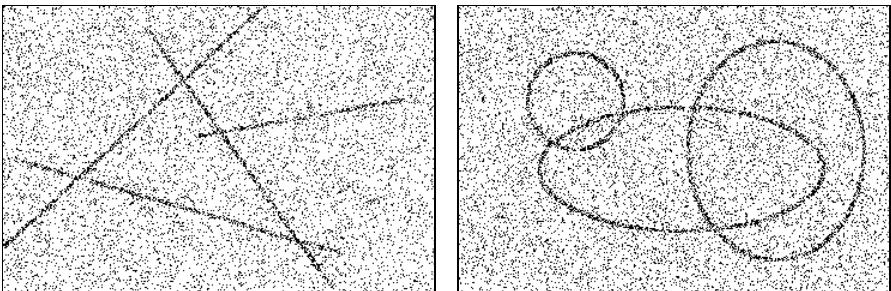


Figure 3.1 The human visual system is capable of instantly recognizing prominent image structures even under difficult conditions.

them if they belong to the object’s contour, and repeat. In principle, such an approach could be applied to either a continuous edge map consisting of edge strengths and orientations or a simple binary *edge map*. Unfortunately, with either input, such an approach is likely to fail due to image noise and ambiguities that arise when trying to follow the contours. Additional constraints and information about the type of object sought are needed in order to handle pixel-level problems such as branching, as well as interruptions. This type of local sequential *contour tracing* makes for an interesting optimization problem [47] (see also Sec. 2.2).

A completely different approach is to search for globally apparent structures that consist of certain simple shape features. As an example, Fig. 3.1 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. 3.1 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.

3.2 Hough Transform

The method from Paul Hough—originally published as a US Patent [36] 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 [21, 39]. 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. 3.2).

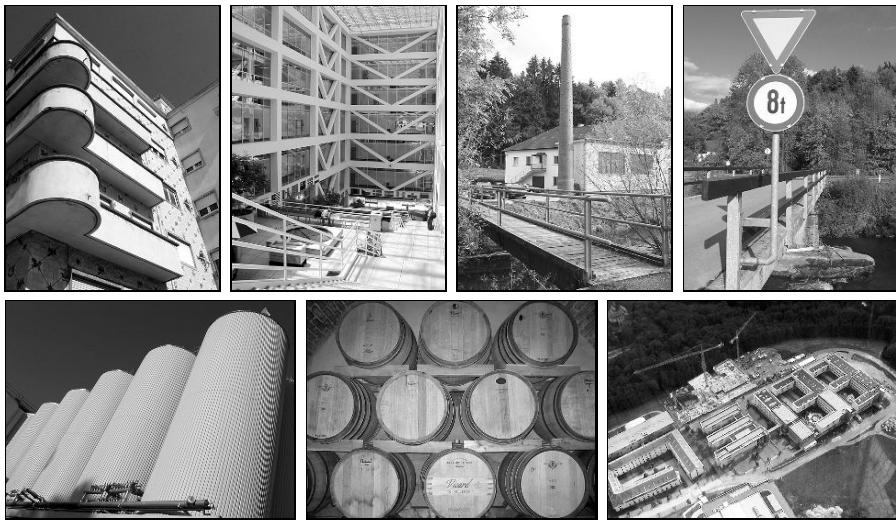


Figure 3.2 Simple geometrical forms such as sections of lines, circles, and ellipses are often found in man-made objects.

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

$$y = kx + d, \quad (3.1)$$

where k is the slope and d the intercept—that is, the height at which the line would intercept the y axis (Fig. 3.3). A line segment that passes through two given edge points $\mathbf{p}_1 = (x_1, y_1)$ and $\mathbf{p}_2 = (x_2, y_2)$ must satisfy the conditions

$$y_1 = kx_1 + d \quad \text{and} \quad y_2 = kx_2 + d \quad (3.2)$$

for $k, d \in \mathbb{R}$. The goal is to find values of k and d such that as many edge points as possible lie on the line they describe; in other words, the line that fits the most edge points. But how can you determine the number of edge points that lie on a given line segment? One possibility is to exhaustively “draw” every possible line segment into the image while counting the number of points that lie exactly on each of these. Even though the discrete nature of pixel images (with only a finite number of different lines) makes this approach possible in theory, generating such a large number of lines is infeasible in practice.

3.2.1 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

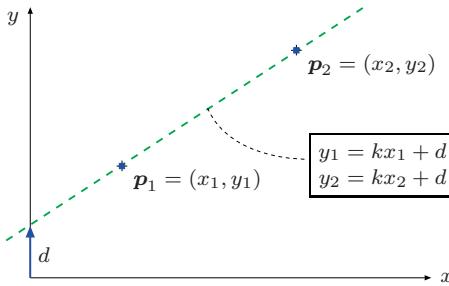


Figure 3.3 Two points, p_1 and p_2 , lie on the same line when $y_1 = kx_1 + d$ and $y_2 = kx_2 + d$ for a particular pair of parameters k and d .

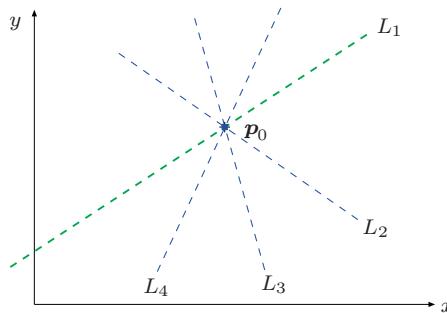


Figure 3.4 Set of lines passing through an image point. For all possible lines L_j passing through the point $p_0 = (x_0, y_0)$, the equation $y_0 = k_j x_0 + d_j$ holds for appropriate values of the parameters k_j, d_j .

the image. Every line $L_j = \langle k_j, d_j \rangle$ that runs through a point $p_0 = (x_0, y_0)$ must satisfy the condition

$$L_j : y_0 = k_j x_0 + d_j \quad (3.3)$$

for some suitable pair of values k_j, d_j . Equation 3.3 is underdetermined and the possible solutions for k_j, d_j correspond to an infinite set of lines passing through the given point p_0 (Fig. 3.4). Note that for a given k_j , the solution for d_j in Eqn. (3.3) is

$$d_j = -x_0 k_j + y_0, \quad (3.4)$$

which is another equation for a line, where now k_j, d_j are the *variables* and x_0, y_0 are the constant *parameters* of the equation. The solution set $\{(k_j, d_j)\}$ of Eqn. (3.4) describes the parameters of all possible lines L_j passing through the image point $p_0 = (x_0, y_0)$. For an arbitrary image point $p_i = (x_i, y_i)$, Eqn. (3.4) describes the line

$$M_i : d = -x_i k + y_i \quad (3.5)$$

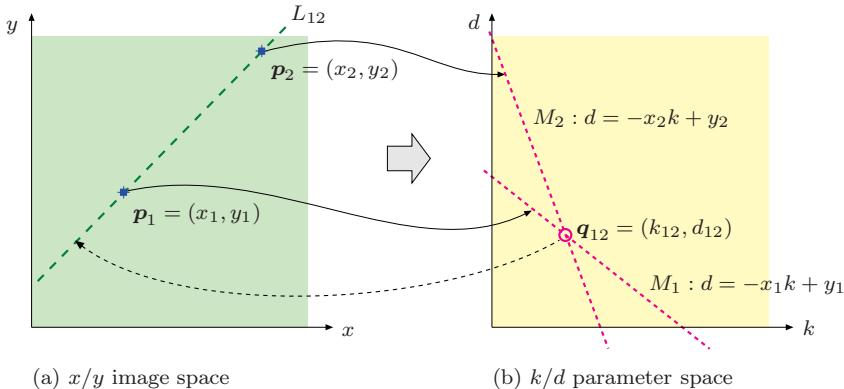


Figure 3.5 Relationship between image space and parameter space. The parameter values for all possible lines passing through the image point $\mathbf{p}_i = (x_i, y_i)$ in image space (a) lie on a single line M_i in parameter space (b). This means that each point $\mathbf{q}_j = (k_j, d_j)$ in parameter space corresponds to a single line L_j in image space. The intersection of the two lines M_1 , M_2 at the point $\mathbf{q}_{12} = (k_{12}, d_{12})$ in parameter space indicates that a line L_{12} through the two points k_{12} and d_{12} exists in the image space.

with the parameters $-x_i, y_i$ in the so-called *parameter* or *Hough space*, spanned by the coordinates k, d .

The relationship between (x, y) *image space* and (k, d) *parameter space* can be summarized as follows:

<i>Image Space</i> (x, y)		<i>Parameter Space</i> (k, d)	
Point	$\mathbf{p}_i = (x_i, y_i)$	$M_i : d = -x_i k + y_i$	Line
Line	$L_j : y = k_j x + d_j$	$\mathbf{q}_j = (k_j, d_j)$	Point

Each image point \mathbf{p}_i and its associated line bundle correspond to exactly one line M_i in parameter space. Therefore we are interested in those places in the parameter space where lines *intersect*. The example in Fig. 3.5 illustrates how the lines M_1 and M_2 intersect at the position $\mathbf{q}_{12} = (k_{12}, d_{12})$ in the parameter space, which means (k_{12}, d_{12}) are the parameters of the line in the image space that runs through both image points \mathbf{p}_1 and \mathbf{p}_2 . The more lines M_i that intersect at a single point in the parameter space, the more image space points lie on the corresponding line in the image! In general, we can state:

If N lines intersect at position (k', d') in *parameter space*, then N image points lie on the corresponding line $y = k'x + d'$ in *image space*.

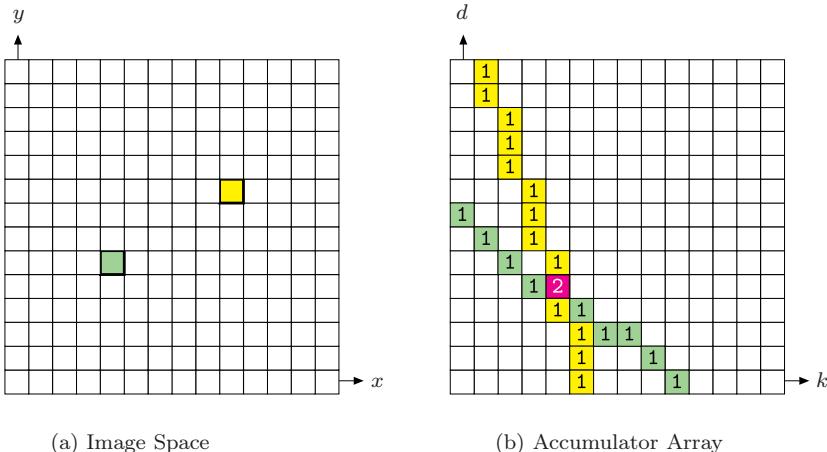


Figure 3.6 Main idea of the Hough transform. The accumulator array is a discrete representation of the parameter space (k, d). For each image point found (a), a discrete line in the parameter space (b) is drawn. This operation is performed *additively* so that the values of the array through which the 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).

3.2.2 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 a two-dimensional array. Since the array will be used to keep track of the number of times parameter space lines intersect, it is called an “accumulator” array. Each parameter space line is painted into the accumulator array and the cells through which it passes are incremented, so that ultimately each cell accumulates the total number of lines that intersect at that cell (Fig. 3.6).

3.2.3 A Better Line Representation

The line representation in Eqn. (3.1) is not used in practice because for vertical lines the slope is infinite, i.e., $k = \infty$. A more practical representation is the so-called *Hessian normal form* (HNF, [11, p. 195]) for representing lines,

$$x \cdot \cos(\theta) + y \cdot \sin(\theta) = r, \quad (3.6)$$

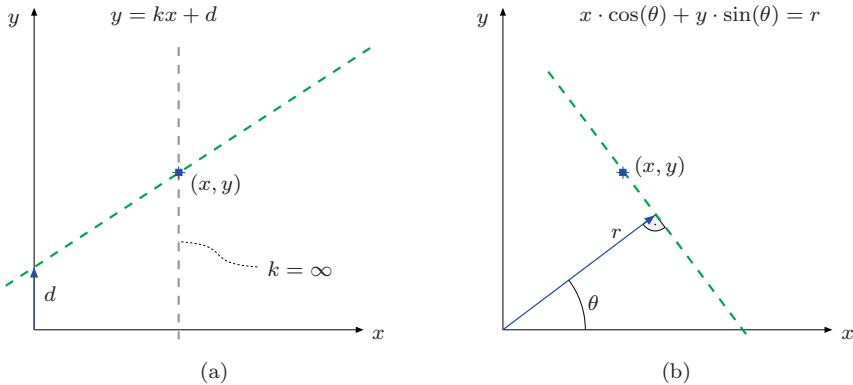


Figure 3.7 Representation of lines in 2D. In the normal k, d representation (a), vertical lines pose a problem because $k = \infty$. The Hessian normal form (b) avoids this problem by representing a line by its angle θ and distance r from the origin.

which does not exhibit such singularities and also provides a natural linear quantization for its parameters, the angle θ and the radius r (Fig. 3.7). With the HNF² representation, the parameter space is defined by the coordinates θ, r , and a point $p = (x, y)$ in image space corresponds to the function

$$r_{x,y}(\theta) = x \cdot \cos(\theta) + y \cdot \sin(\theta) \quad (3.7)$$

for angles in the range $0 \leq \theta < \pi$ (Fig. 3.8). 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,

$$-r_{\max} \leq r_{x,y}(\theta) \leq r_{\max}, \quad \text{where } r_{\max} = \frac{1}{2}\sqrt{M^2 + N^2}, \quad (3.8)$$

for an image of width M and height N .

3.3 Implementing the Hough Transform

The fundamental Hough algorithm using the HNF line representation (Eqn. (3.6)) is given in Alg. 3.1. 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 list of parameter pairs for the K strongest lines

$$\text{MaxLines} = [\langle \theta_1, r_1 \rangle, \langle \theta_2, r_2 \rangle, \dots, \langle \theta_K, r_K \rangle]$$

is computed. The next sections explain these two stages in detail.

² The Hessian normal form is a constrained variant of the general line equation $ax + by + c = 0$, with $a = \cos(\theta)$, $b = \sin(\theta)$, and $c = -r$ (see [11, p. 194]).

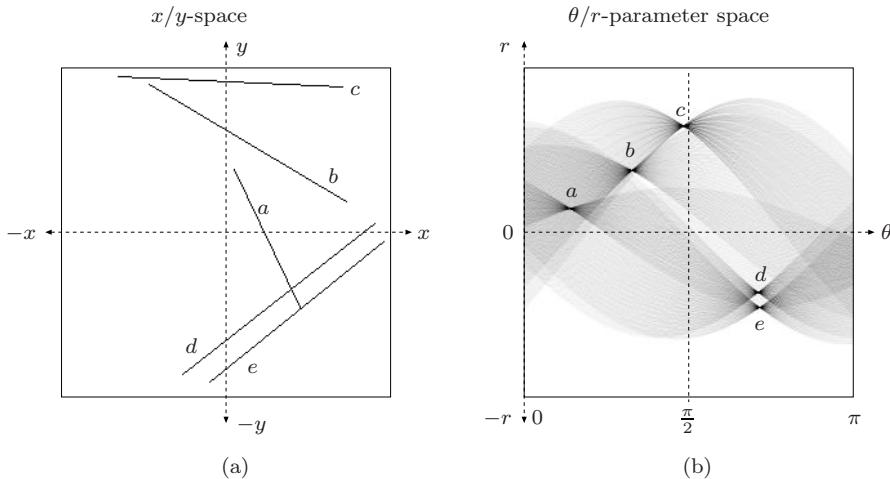


Figure 3.8 Image space and parameter space using the HNF representation.

3.3.1 Filling the Accumulator Array

A direct implementation of the first phase of Alg. 3.1 is given in the Java class `LinearHT` Prog. 3.1.³ 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`, for example,

```
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

$$\Delta_\theta = \frac{\pi}{N_\theta} \quad \text{and} \quad \Delta_r = \frac{2 \cdot r_{\max}}{N_r}$$

(see lines 17 and 21 in Prog. 3.1, respectively). The output of this program for a very noisy edge image is given in Fig. 3.9.

3.3.2 Analyzing the Accumulator Array

The second phase is localizing the maximum values in the accumulator array $Acc(i_\theta, i_r)$. As can readily be seen in Fig. 3.9 (b), even in the case where

³ The complete implementation of the Hough transform for straight lines can be found in the source code section of this book's Website.

Algorithm 3.1 Simple Hough algorithm for detecting straight lines. It returns a list containing the parameters $\langle \theta, r \rangle$ of the K strongest lines in the binary edge image I .

```

1: HOUGHLINES( $I, N_\theta, N_r, K$ )
   Computes the Hough transform to detect straight lines in the binary
   image  $I$  (of size  $M \times N$ ), using  $N_\theta, N_r$  discrete steps for the angle
   and radius, respectively. Returns the list of parameter pairs  $\langle \theta_i, r_i \rangle$ 
   for the  $K$  strongest lines found.

2:  $(u_c, v_c) \leftarrow (\frac{M}{2}, \frac{N}{2})$                                  $\triangleright$  image center
3:  $r_{\max} \leftarrow \sqrt{u_c^2 + v_c^2}$                                  $\triangleright$  max. radius is half the image diagonal
4:  $\Delta_\theta \leftarrow \frac{\pi}{N_\theta}$                                           $\triangleright$  angular increment
5:  $\Delta_r \leftarrow \frac{2 \cdot r_{\max}}{N_r}$                                       $\triangleright$  radial increment

6: Create the accumulator array  $Acc(i_\theta, i_r)$  of size  $N_\theta \times N_r$ 
7: for all accumulator cells  $(i_\theta, i_r)$  do
8:    $Acc(i_\theta, i_r) \leftarrow 0$                                           $\triangleright$  initialize the accumulator array

9: for all image coordinates  $(u, v)$  do                                $\triangleright$  scan the image
10:   if  $I(u, v)$  is an edge point then
11:      $(x, y) \leftarrow (u - u_c, v - v_c)$                                  $\triangleright$  coordinate relative to center
12:     for  $i_\theta \leftarrow 0 \dots N_\theta - 1$  do                                 $\triangleright$  angular index  $i_\theta$ 
13:        $\theta \leftarrow \Delta_\theta \cdot i_\theta$                                       $\triangleright$  real angle,  $0 \leq \theta < \pi$ 
14:        $r \leftarrow x \cdot \cos(\theta) + y \cdot \sin(\theta)$                        $\triangleright$  real radius (pos./neg.)
15:        $i_r \leftarrow \frac{N_r}{2} + \text{round}(\frac{r}{\Delta_r})$                           $\triangleright$  radial index  $i_r$ 
16:        $Acc(i_\theta, i_r) \leftarrow Acc(i_\theta, i_r) + 1$                        $\triangleright$  increment  $Acc(i_\theta, i_r)$ 

   Find the parameters pairs  $\langle \theta_j, r_j \rangle$  for the  $K$  strongest lines:
17:  $MaxLines \leftarrow \text{FINDMAXLINES}(Acc, K)$ 
18: return  $MaxLines$ .

```

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. 3.10) in the following.

```

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; //  $N_\theta$  steps for the angle ( $\theta = 0 \dots \pi$ )
5     int nRad; //  $N_r$  steps for the radius ( $r = -r_{\max} \dots r_{\max}$ )
6     int cRad; // center of radius axis ( $r = 0$ )
7     double dAng; // increment of angle  $\Delta_\theta$ 
8     double dRad; // increment of radius  $\Delta_r$ 
9     int[][] houghArray; // Hough accumulator  $Acc(i_\theta, i_r)$ 
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 ia = 0; ia < nAng; ia++) {
41            double theta = dAng * ia;
42            int ir = cRad + (int) Math.rint
43                ((x*Math.cos(theta) + y*Math.sin(theta)) / dRad);
44            if (ir >= 0 && ir < nRad) {
45                houghArray[ia][ir]++;
46            }
47        }
48    }
49}
50} // end of class LinearHT

```

Program 3.1 Hough transform for localizing straight lines (partial implementation). The complete Java implementation can be found in the source code section of the book's Website.

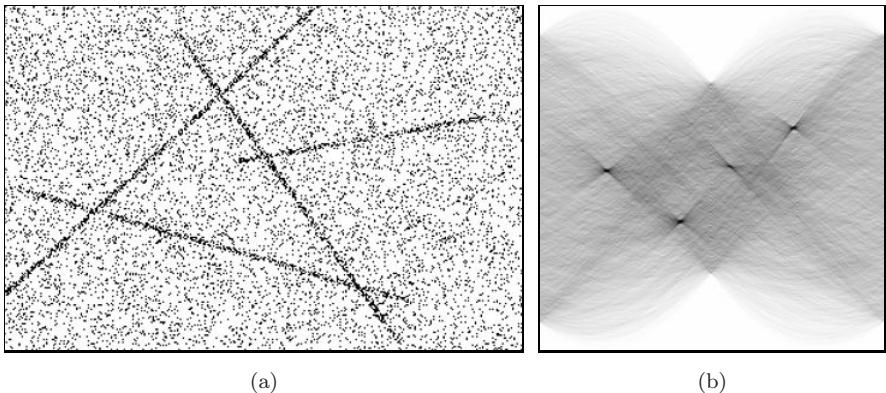


Figure 3.9 Hough transform for straight lines. The dimensions of the original image (a) are 360×240 pixels, so the maximal radius (measured from the image center (u_c, v_c)) is $r_{\max} \approx 216$. For the parameter space (b), a step size of 256 is used for both the angle $\theta = 0 \dots \pi$ (horizontal axis) and the radius $r = -r_{\max} \dots r_{\max}$ (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.

Approach A: Thresholding

First the accumulator is thresholded to the value of t_a by setting all accumulator values $Acc(i_\theta, i_r) < t_a$ to 0. The resulting scattering of points, or point clouds, are first coalesced into regions (Fig. 3.10(b)) using a technique such as a morphological *closing* operation (see Vol. 1 [14, Sec. 7.3.2]). Next the remaining regions must be localized, for instance using the region-finding technique from Sec. 2.1, and then each region's centroid (see Sec. 2.4.3) 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.

Approach B: Nonmaximum suppression

In this method, local maxima in the accumulator array are found by suppressing nonmaximal values.⁴ 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. 3.10(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

⁴ Nonmaximum suppression is also used in Sec. 4.2.3 for isolating corner points.

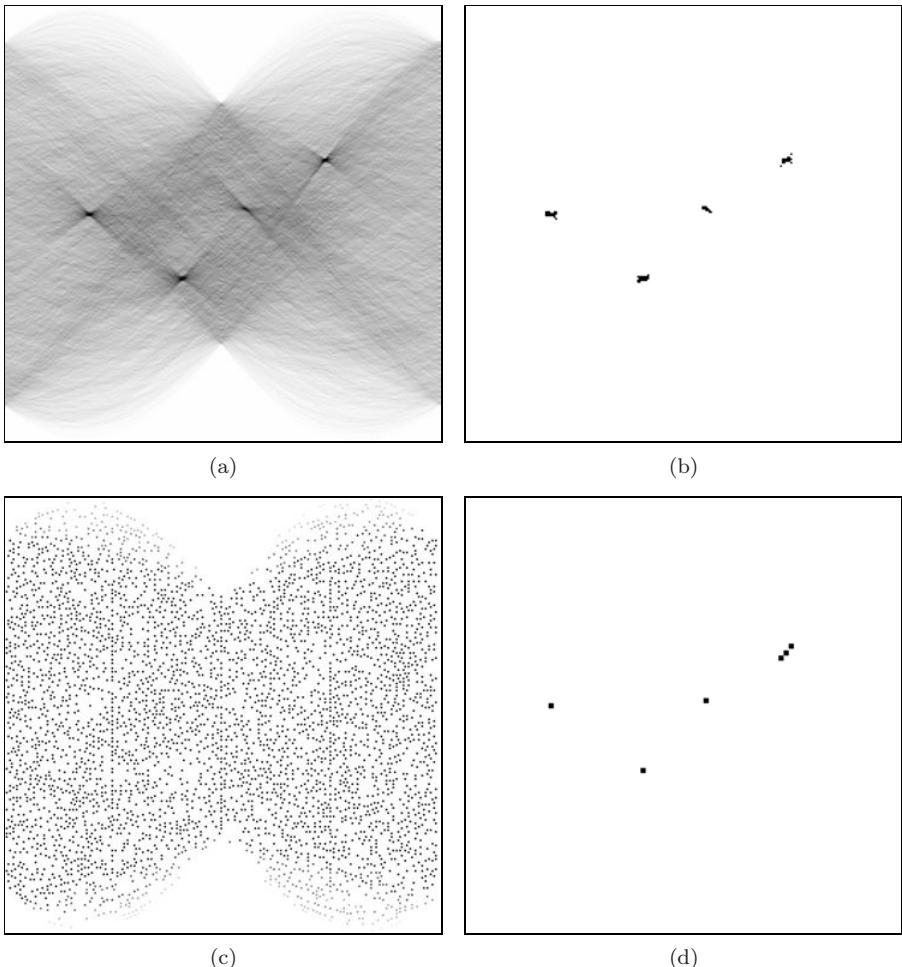


Figure 3.10 Determining the local maximum values in the accumulator array. Original distribution of the values in the Hough accumulator (a). **Variant A:** *Threshold operation* using 50% of the maximum value (b). The remaining regions represent the four dominant lines in the image, and the coordinates 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).

considered. The result for Fig. 3.9 (a) is shown in Fig. 3.10 (d).

3.3.3 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 $\theta = i_\theta \cdot \Delta_\theta$ (Alg. 3.1), to increment not only the corresponding accumulator cell $Acc(i_\theta, i_r)$ but also the *neighboring* cells $Acc(i_\theta, i_r - 1)$ and $Acc(i_\theta, i_r + 1)$. This makes the Hough transform more tolerant against inaccurate point coordinates and rounding errors.

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. 3.11).

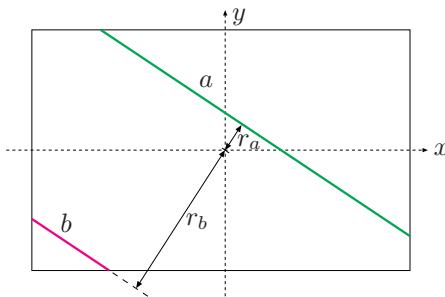


Figure 3.11 Bias problem. When an image represents only a finite section of an object, then those lines nearer the center (smaller r values) will have higher values than those farther away (larger r values). As an example, the maximum value of the accumulator for line a will be higher than that of line 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 $Acc(i_\theta, i_r)$ the maximum number of image points $MaxHits(i_\theta, i_r)$ possible for a line with the parameters θ, r and then normalize

$$Acc'(i_\theta, i_r) \leftarrow \frac{Acc(i_\theta, i_r)}{MaxHits(i_\theta, i_r)} \quad (3.9)$$

for $\text{MaxHits}(i_\theta, i_r) > 0$. The normalization term $\text{MaxHits}(i_\theta, i_r)$ can be determined, for example, by computing the Hough transform of an image with the same dimensions in which all pixels are edge pixels or by using a random image in which the pixels are uniformly distributed.

Line endpoints

Our simple version of the Hough transform determines the parameters of the line in the image but not their endpoints. These could be found in a subsequent step by determining which image points belong to any detected line (e.g., by applying a threshold to the perpendicular distance between the ideal line—defined by its parameters—and the actual image points). An alternative solution is to calculate the extreme point of the line during the computation of the accumulator array. For this, every cell of the accumulator array is supplemented with two additional coordinate pairs $\mathbf{x}_s = (x_s, y_s)$, $\mathbf{x}_e = (x_e, y_e)$, i.e.,

$$\text{Acc}(i_\theta, i_r) = \langle \text{count}, \mathbf{x}_s, \mathbf{x}_e \rangle.$$

Now the coordinates for the endpoints (\mathbf{x}_s , \mathbf{x}_e) 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 cells 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 $\mathbf{x}_0 = (x_0, y_0)^T$ for two lines

$$L_1 = \langle \theta_1, r_1 \rangle \quad \text{and} \quad L_2 = \langle \theta_2, r_2 \rangle,$$

we need to solve the system of linear equations

$$x_0 \cdot \cos(\theta_1) + y_0 \cdot \sin(\theta_1) = r_1, \tag{3.10}$$

$$x_0 \cdot \cos(\theta_2) + y_0 \cdot \sin(\theta_2) = r_2, \tag{3.11}$$

for the unknowns x_0, y_0 . The solution is

$$\begin{aligned} \mathbf{x}_0 &= \frac{1}{\cos(\theta_1)\sin(\theta_2) - \cos(\theta_2)\sin(\theta_1)} \cdot \begin{bmatrix} r_1 \sin(\theta_2) - r_2 \sin(\theta_1) \\ r_2 \cos(\theta_1) - r_1 \cos(\theta_2) \end{bmatrix} \\ &= \frac{1}{\sin(\theta_2 - \theta_1)} \cdot \begin{bmatrix} r_1 \sin(\theta_2) - r_2 \sin(\theta_1) \\ r_2 \cos(\theta_1) - r_1 \cos(\theta_2) \end{bmatrix} \end{aligned} \tag{3.12}$$

for $\sin(\theta_2 - \theta_1) \neq 0$. Obviously \mathbf{x}_0 is undefined (no intersection point exists) if the lines L_1, L_2 are parallel to each other (i.e., if $\theta_1 \equiv \theta_2$).

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 Vol. 1 [14, Sec. 6.3]), which can be used to improve the results of the HT.

The *edge strength* $E(u, v)$ is especially easy to take into consideration. Instead of incrementing visited accumulator cells by 1, add the strength of the respective edge:

$$Acc(i_\theta, i_r) \leftarrow Acc(i_\theta, i_r) + E(u, v).$$

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 to be considered along the θ axis. Since this also reduces the number of irrelevant “votes” in the accumulator, it increases the overall sensitivity of the Hough transform (see, for example, [45, 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 at every $\frac{\pi}{256} \approx 0.7^\circ$. While increasing the number of accumulator cells provides a finer result, bear in mind that it also increases the computation time and especially the amount of memory required.

Instead of increasing the resolution of the entire parameter space, the idea of the hierarchical HT is to gradually “zoom” in and refine the parameter space. First, the regions containing the most important lines are found using a relatively low-resolution parameter space, and then the parameter spaces of those regions are recursively passed to the HT and examined at a higher resolution. In this way, a relatively exact determination of the parameters can be found using a limited (in comparison) parameter space.

3.4 Hough Transform for Circles and Ellipses

3.4.1 Circles and Arcs

Since lines in 2D have two degrees of freedom, they could be completely specified using two real-valued parameters. In a similar fashion, representing a circle in 2D requires *three* parameters, for example

$$\text{Circle} = \langle \bar{x}, \bar{y}, \rho \rangle,$$

where \bar{x} , \bar{y} are the coordinates of the center and ρ is the radius of the circle (Fig. 3.12). A point $\mathbf{p} = (x, y)$ lies on this circle when the relation

$$(x - \bar{x})^2 + (y - \bar{y})^2 = \rho^2 \quad (3.13)$$

holds. Therefore the Hough transform requires a three-dimensional parameter space $\text{Acc}(\bar{x}, \bar{y}, \rho)$ 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}, ρ) that satisfies Eqn. (3.13) for a given image point (u, v) ? One solution is to apply a “brute force” method such as described in Alg. 3.2 that exhaustively tests each cell in the parameter space to see if the relation in Eqn. (3.13) holds.

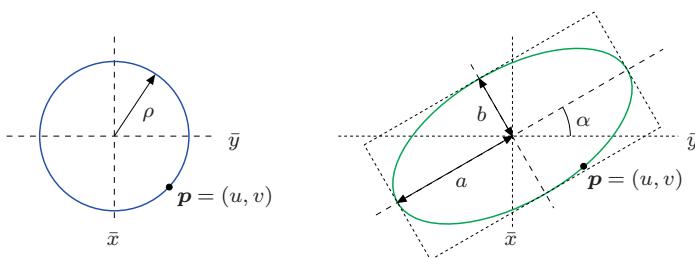


Figure 3.12 Representation of circles and ellipses in 2D.

If we examine Fig. 3.13, 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 $\mathbf{p} = (u, v)$. 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 [9] is particularly well-suited.

Figure 3.14 shows the spatial structure of the three-dimensional parameter space for circles. For a given image point $\mathbf{p}_k = (u_k, v_k)$, at each plane along

Algorithm 3.2 Exhaustive Hough algorithm for localizing circles.

```

1: HOUGH CIRCLES( $I$ )
   Returns the list of parameters  $\langle \bar{x}_i, \bar{y}_i, \rho_i \rangle$  corresponding to the
   strongest circles found in the binary image  $I$ .
2: Set up a three-dimensional array  $Acc(\bar{x}, \bar{y}, \rho)$  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  $(\bar{x}_i, \bar{y}_i, \rho_i)$  in the accumulator space do
6:       if  $(u - \bar{x}_i)^2 + (v - \bar{y}_i)^2 = \rho_i^2$  then
7:         Increment  $Acc(\bar{x}_i, \bar{y}_i, \rho_i)$ 
8: MaxCircles  $\leftarrow$  FINDMAXCIRCLES( $Acc$ )  $\triangleright$  a list of tuples  $\langle \bar{x}_j, \bar{y}_j, \rho_j \rangle$ 
9: return  $MaxCircles$ .

```

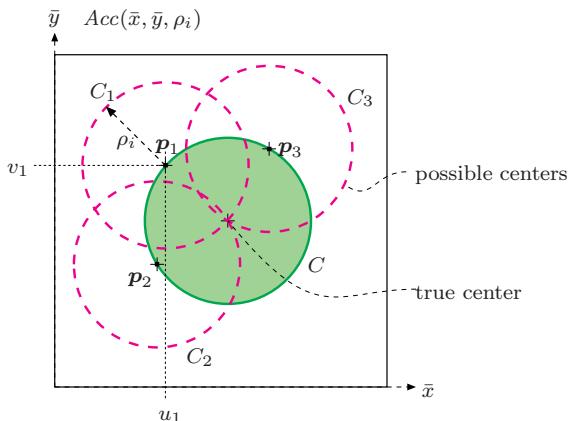


Figure 3.13 Hough transform for circles. The illustration depicts a slice of the three-dimensional accumulator array $Acc(\bar{x}, \bar{y}, \rho)$ at a given circle radius $\rho = \rho_i$. The center points of all the circles running through a given image point $p_1 = (u_1, v_1)$ form a circle C_1 with a radius of ρ_i centered around p_1 , just as the center points of the circles that pass through p_2 and p_3 lie on the circles C_2, C_3 . The cells along the edges of the three circles C_1, C_2, C_3 of radius 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 .

the ρ axis (for $\rho_i = \rho_{\min} \dots \rho_{\max}$), 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. 3.3.3) also occurs in

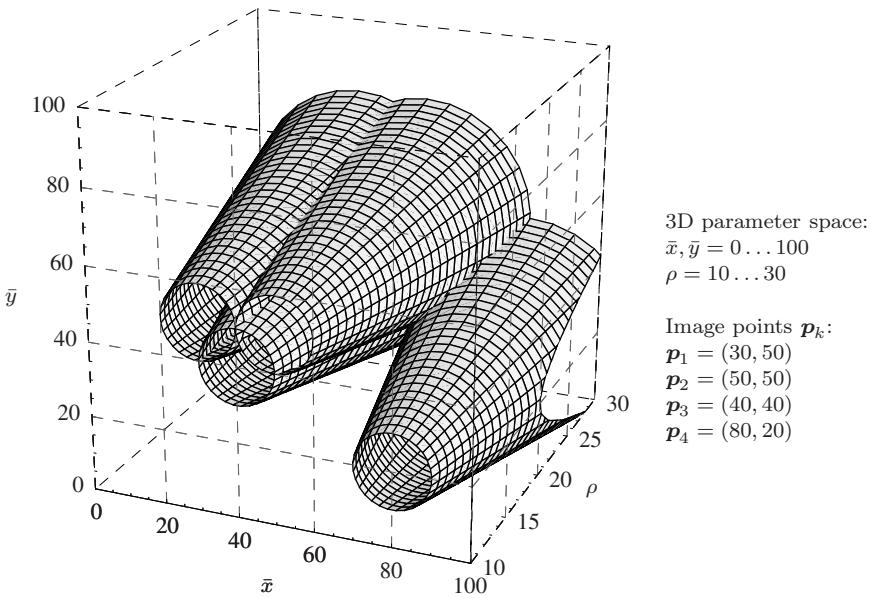


Figure 3.14 Three-dimensional parameter space for circles. For each image point $p_k = (u_k, v_k)$, the cells lying along a cone in the three-dimensional accumulator array $Acc(\bar{x}, \bar{y}, \rho)$ are incremented.

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.

3.4.2 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,

$$Ellipse = \langle \bar{x}, \bar{y}, r_a, r_b, \alpha \rangle,$$

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. 3.12).⁵ 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 every dimension results in 2^{35} accumulator cells, and implementing these using 4-byte `int` values thus requires 2^{37} bytes (128 gigabytes) of memory.

An interesting alternative in this case is the *generalized Hough transform*, which in principle can be used for detecting any arbitrary two-dimensional shape [2, 39]. 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.

3.5 Exercises

Exercise 3.1

Implement a version of the Hough transform for straight lines that incorporates the modified accumulator update, as suggested in Sec. 3.3.3. Analyze the extent to which the method improves the robustness with respect to inaccurate or noisy point positions.

Exercise 3.2

Implement a version of the Hough transform for finding lines that takes into account line endpoints as described in Sec. 3.3.3.

Exercise 3.3

Implement a *hierarchical* Hough transform for straight lines (see p. 63) capable of accurately determining line parameters.

Exercise 3.4

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. 3.4, in the accumulator array.

⁵ See Eqn. (2.34) on p. 43 for a parametric equation of this ellipse.

4

Corner Detection

Corners are prominent structural elements in an image and are therefore useful in a wide variety of applications, including following objects across related images (*tracking*), determining the correspondence between stereo images, serving as reference points for precise geometrical measurements, and calibrating camera systems for machine vision applications. Thus corner points are important not only in human vision but they are also “robust” in the sense that they do not arise accidentally in 3D scenes and furthermore can be located quite reliably under a wide range of viewing angles and lighting conditions.

4.1 Points of Interest

Despite being easily recognized by our visual system, accurately and precisely detecting corners automatically is not a trivial task. A good corner detector must satisfy a number of criteria, including distinguishing between true and accidental corners, reliably detecting corners in the presence of realistic image noise, and precisely and accurately determining the locations of corners, 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.

Numerous methods for finding corners or similar interest points have been proposed and most of them take advantage of the following basic principle. While an *edge* is usually defined as a location in the image at which the gradient is especially high in *one* direction and low in the direction normal to it, a *corner point* is defined as a location that exhibits a strong gradient value in *multiple* directions at the same time.

Most methods take advantage of this observation by examining the first or second derivative of the image in the x and y directions to find corners (e.g., [23, 31, 49, 51]). In the next section, we describe in detail the Harris detector, also known as the “Plessey feature point detector” [31], since it turns out that even though more efficient detectors are known (see, for example, [63, 68]), the Harris detector, and other detectors based on it, are the most widely used in practice.

4.2 Harris Corner Detector

This operator, developed by Harris and Stephens [31], is one of a group of related methods based on the same premise: a corner point exists where the gradient of the image is especially strong in more than one direction at the same time. In addition, locations along edges, where the gradient is strong in only one direction, should not be considered as corners, and the detector should be isotropic, i.e., independent of the orientation of the local gradients.

4.2.1 Local Structure Matrix

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

$$I_x(u, v) = \frac{\partial I}{\partial x}(u, v) \quad \text{and} \quad I_y(u, v) = \frac{\partial I}{\partial y}(u, v). \quad (4.1)$$

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

$$A(u, v) = I_x^2(u, v), \quad (4.2)$$

$$B(u, v) = I_y^2(u, v), \quad (4.3)$$

$$C(u, v) = I_x(u, v) \cdot I_y(u, v), \quad (4.4)$$

which will be interpreted as elements of the *local structural matrix* $\mathbf{M}(u, v)$:¹

$$\mathbf{M} = \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix} = \begin{pmatrix} A & C \\ C & B \end{pmatrix}. \quad (4.5)$$

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 Vol. 1 [14,

¹ For improved legibility, we simplify the notation used in the following by omitting the function coordinates (u, v) ; e.g., the function $I_x(u, v)$ is abbreviated as I_x or $A(u, v)$ is simply denoted A etc.

Sec. 5.2.7]),

$$\bar{\mathbf{M}} = \begin{pmatrix} A * H^{G,\sigma} & C * H^{G,\sigma} \\ C * H^{G,\sigma} & B * H^{G,\sigma} \end{pmatrix} = \begin{pmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{pmatrix}. \quad (4.6)$$

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

$$\bar{\mathbf{M}}' = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}, \quad (4.7)$$

where λ_1 and λ_2 are the *eigenvalues* of the matrix $\bar{\mathbf{M}}$, defined as²

$$\begin{aligned} \lambda_{1,2} &= \frac{\text{trace}(\bar{\mathbf{M}})}{2} \pm \sqrt{\left(\frac{\text{trace}(\bar{\mathbf{M}})}{2}\right)^2 - \det(\bar{\mathbf{M}})} \\ &= \frac{1}{2} \left(\bar{A} + \bar{B} \pm \sqrt{\bar{A}^2 - 2\bar{A}\bar{B} + \bar{B}^2 + 4\bar{C}^2} \right). \end{aligned} \quad (4.8)$$

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), $\bar{\mathbf{M}} = 0$ and therefore $\lambda_1 = \lambda_2 = 0$. On an ideal ramp, however, the eigenvalues are $\lambda_1 > 0$ and $\lambda_2 = 0$, independent of the orientation of the edge. The eigenvalues thus encode an edge's *strength*, and their associated *eigenvectors* 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 both eigenvalues must be significant. Since $\bar{A}, \bar{B} \geq 0$, we can assume that $\text{trace}(\bar{\mathbf{M}}) > 0$ and thus $|\lambda_1| \geq |\lambda_2|$. Therefore only the smaller of the two eigenvalues, $\lambda_2 = \text{trace}(\bar{\mathbf{M}})/2 - \sqrt{\dots}$, is relevant when determining a corner.

4.2.2 Corner Response Function (CRF)

As we can see from Eqn. (4.8), the difference between the two eigenvalues is

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

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

$$\begin{aligned} Q(u, v) &= \det(\bar{\mathbf{M}}) - \alpha \cdot (\text{trace}(\bar{\mathbf{M}}))^2 \\ &= (\bar{A}\bar{B} - \bar{C}^2) - \alpha \cdot (\bar{A} + \bar{B})^2 \end{aligned} \quad (4.9)$$

² Where $\det(\bar{\mathbf{M}})$ denotes the *determinant* and $\text{trace}(\bar{\mathbf{M}})$ denotes the *trace* of the matrix $\bar{\mathbf{M}}$ (see, for example, [11, pp. 252 and 259]).

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.

4.2.3 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 $\mathbf{c}_i = \langle u_i, v_i, q_i \rangle$ are inserted into the vector

$$\text{Corners} = [\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_N],$$

which is then sorted in descending order (i.e., $q_i \geq q_{i+1}$) according to *corner strength* $q_i = Q(u_i, v_i)$, as defined in Eqn. (4.9). To suppress the false corners that tend to arise in densely packed groups around true corners, all except the strongest corner in a specified vicinity 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 surrounding neighborhood of a stronger corner, are deleted.

The complete algorithm for the Harris detector is summarized again in Alg. 4.1, and the associated parameters are explained in Table 4.1.

4.2.4 Example

Figure 4.1 uses a simple synthetic image to illustrate the most important steps in corner detection using the Harris detector. The figure shows the result of the gradient computation, the three components of the structure matrix $\mathbf{M}(u, v) = \begin{pmatrix} A & C \\ C & B \end{pmatrix}$, and the values of the *corner response function* $Q(u, v)$ for each image position (u, v) . This example utilizes the standard settings as given in Table 4.1.

The second example (Fig. 4.2) illustrates the detection of corner points in a grayscale representation of a natural scene. It demonstrates how weak corners are eliminated in favor of the strongest corner in a region.

4.3 Implementation

Since the Harris detector algorithm is more complex than the algorithms we presented earlier, in the following sections, we explain its implementation in

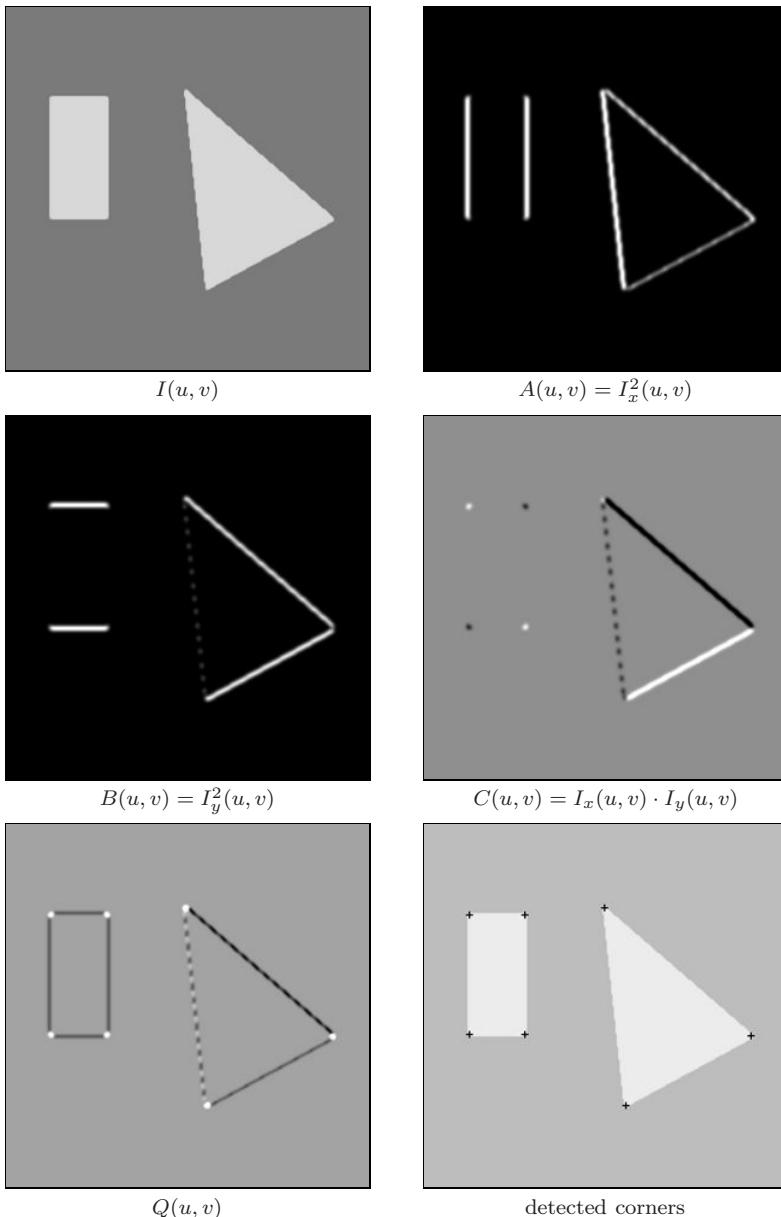


Figure 4.1 Harris corner detector—Example 1. Starting with the original image $I(u, v)$, the first derivative is computed, and then from it the components of the structure matrix $\mathbf{M}(u, v)$, with $A(u, v) = I_x^2(u, v)$, $B = I_y^2(u, v)$, $C = I_x(u, v) \cdot I_y(u, v)$. $A(u, v)$ and $B(u, v)$ represent, respectively, the strength of the horizontal and vertical edges. In $C(u, v)$, the values are strongly positive (white) or strongly negative (black) only where the edges are strong in both directions (null values are shown in gray). The corner response function, $Q(u, v)$, exhibits noticeable positive peaks at the corner positions.

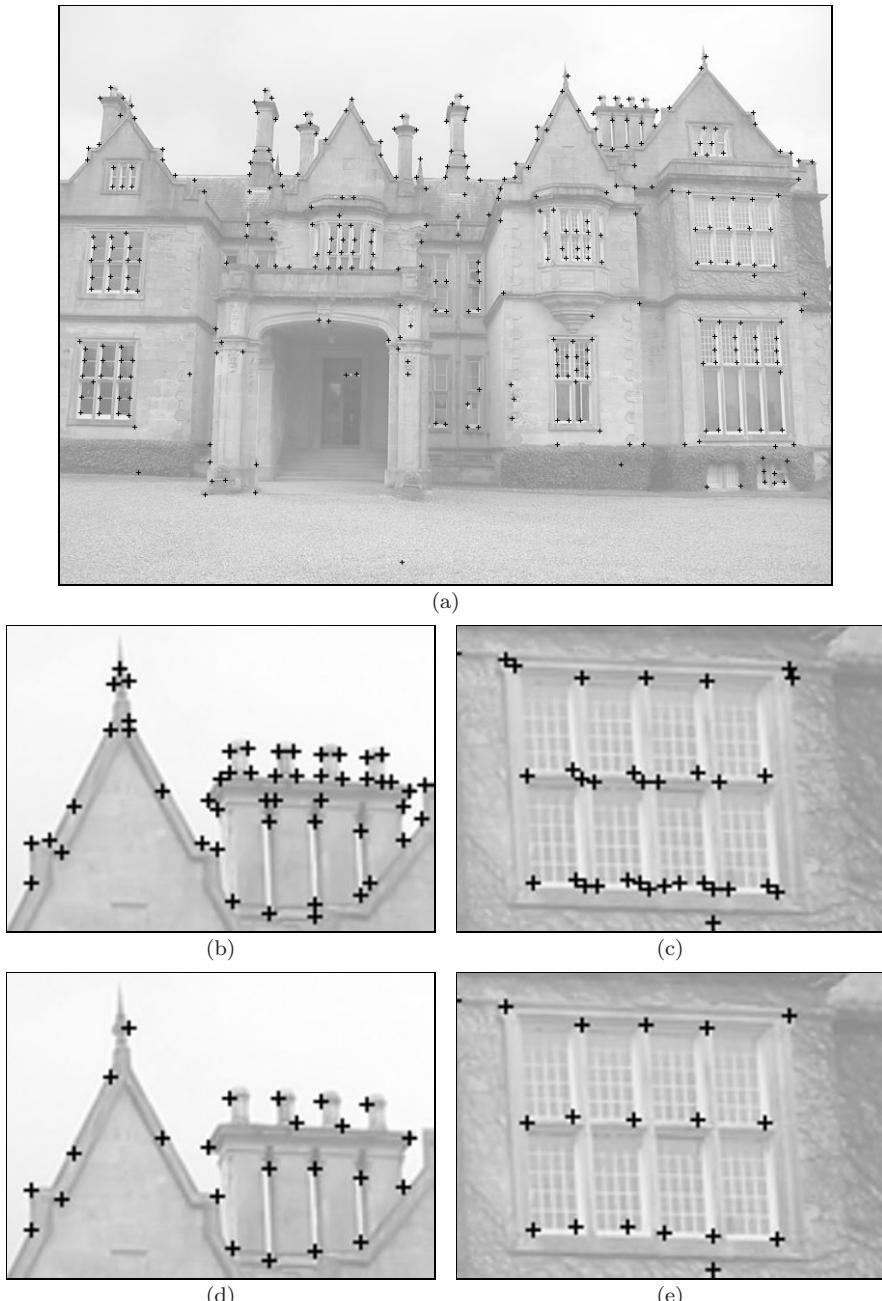


Figure 4.2 Harris corner detector—Example 2. A complete result with the final corner points marked (a). After selecting the strongest corner points within a 10-pixel radius, only 335 of the original 615 candidate corners remain. Details *before* (b, c) and *after* selection (d, e).

Algorithm 4.1 Harris corner detector (Part 1). This algorithm takes an intensity image I and creates a sorted list of detected corner points. $*$ is the convolution operator used for linear filter operations. Details for the parameters H_p , H_{dx} , H_{dy} , H_b , α , and t_H can be found in Table 4.1.

1: HARRISCORNERS(I)

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

STEP 1—COMPUTE THE CORNER RESPONSE FUNCTION:

2: $I' \leftarrow I * H_p$ \triangleright prefilter (smooth) the image

3: $I_x \leftarrow I' * H_{dx}$ \triangleright horizontal derivative

4: $I_y \leftarrow I' * H_{dy}$ \triangleright vertical derivative

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

 Compute elements of the local structure matrix $M = \begin{pmatrix} A & C \\ C & B \end{pmatrix}$:

6: $A(u, v) \leftarrow I_x^2(u, v)$

7: $B(u, v) \leftarrow I_y^2(u, v)$

8: $C(u, v) \leftarrow I_x(u, v) \cdot I_y(u, v)$

Blur each component of the structure matrix: $\bar{M} = \begin{pmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{pmatrix}$:

9: $\bar{A} \leftarrow A * H_b$

10: $\bar{B} \leftarrow B * H_b$

11: $\bar{C} \leftarrow C * H_b$

Compute the corner response function:

12: $Q(u, v) \leftarrow (\bar{A}(u, v) \cdot \bar{B}(u, v) - \bar{C}^2(u, v)) - \alpha \cdot (\bar{A}(u, v) + \bar{B}(u, v))^2$

STEP 2—COLLECT THE CORNER POINTS:

13: Create an empty list:

$Corners \leftarrow []$

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

15: **if** $Q(u, v) > t_H$ **and** ISLOCALMAX(Q, u, v) **then**

16: Create a new corner c_i :

$c_i \leftarrow \langle u_i, v_i, q_i \rangle = \langle u, v, Q(u, v) \rangle$

17: Add c_i to $Corners$

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

19: $GoodCorners \leftarrow \text{CLEANUPNEIGHBORS}(Corners)$

20: **return** $GoodCorners$.

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 B (pp. 294–300).

Algorithm 4.2 Harris corner detector (Part 2). Procedures for finding local maxima in the corner response function and cleaning up the list of detected corner points. Details for the parameter d_{\min} can be found in Table 4.1.

```

1:  IsLOCALMAX( $Q, u, v$ )            $\triangleright$  determine if  $Q(u, v)$  is a local maximum
2:  Let  $q_c \leftarrow Q(u, v)$  (center pixel)
3:  Let  $\mathcal{N} \leftarrow \text{Neighbors}(Q, u, v)$             $\triangleright$  values of all neighboring pixels
4:  if  $q_c \geq q_i$  for all  $q_i \in \mathcal{N}$  then
5:    return true
6:  else
7:    return false.

8:  CLEANUPNEIGHBORS( $Corners$ )    $\triangleright$   $Corners$  is sorted by descending  $q$ 
9:  Create an empty list:
    $GoodCorners \leftarrow []$ 
10:  while  $Corners$  is not empty do
11:     $c_i \leftarrow \text{REMOVEFIRST}(\mathit{Corners})$ 
12:    Add  $c_i$  to  $GoodCorners$ 
13:    for all  $c_j$  in  $Corners$  do
14:      if  $\text{Dist}(c_i, c_j) < d_{\min}$  then
15:        Delete  $c_j$  from  $Corners$ 
16:  return  $GoodCorners$ .

```

4.3.1 Step 1: Computing the Corner Response Function

In order to handle the range of the positive and negative values generated by the filters used in this step, we will need to use floating-point images to store the intermediate results, which also assures sufficient range and precision for small values. The kernels of the required filters, i. e., the presmoothing filter H_p , the gradient filters 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}; //  $H_p$ 
2 float[] dfilt = {0.453014f, 0.0f, -0.453014f}; //  $H_{dx}$ ,  $H_{dy}$ 
3 float[] bfilt = {0.01563f, 0.09375f, 0.234375f, 0.3125f,
4                      0.234375f, 0.09375f, 0.01563f}; //  $H_b$ 

```

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();

```

The first processing step is a presmoothing with the filter H_p (Alg. 4.1, line 2). Subsequently the gradient filters H_{dx} and H_{dy} are used to compute the horizontal and vertical derivatives (Alg. 4.1, line 4). Since one-dimensional

Table 4.1 Harris corner detector—actual parameter values.

Prefilter (Alg. 4.1, line 2): Smoothing with a small xy -separable filter

$$H_p = H_{px} * H_{py}, \text{ where}$$

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

Gradient filter (Alg. 4.1, line 4): Computing the first partial derivative in the x and y directions with

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

Blurfilter (Alg. 4.1, line 11): Smoothing the individual components of the structure matrix M with separable Gaussian filters

$$H_b = H_{bx} * H_{by} \text{ with}$$

$$H_{bx} = \frac{1}{64} [1 \ 6 \ 15 \ 20 \ 15 \ 6 \ 1], \quad H_{by} = H_{bx}^T = \frac{1}{64} \begin{bmatrix} 1 \\ 6 \\ 15 \\ 20 \\ 15 \\ 6 \\ 1 \end{bmatrix}.$$

Steering parameter (Alg. 4.1, line 12): $\alpha = 0.04$ to 0.06 (default 0.05)

Response threshold (Alg. 4.1, line 15): $t_H = 10,000$ to $1,000,000$ (default 25,000)

Neighborhood radius (Alg. 4.2, line 15): $d_{\min} = 10$ pixels

filters of the same direction are applied in each step, presmoothing 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 M 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  $H_b$ 
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 B, p. 299 for the complete source code).

Finally, the corner response function (Alg. 4.1, line 12) 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;           //  $\det(M)$ 
29             float trace = a+b;           //  $\text{trace}(M)$ 
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. B, p. 296) 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; }

```

4.3.2 Step 2: Selecting “Good” Corner Points

The result of the first stage of Alg. 4.1 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 $c_i = \langle u_i, v_i, q_i \rangle$ 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. 4.1, 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 Vol. 1 [14, Appendix 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. 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++) {
```

³ While a Java **Vector** container 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 (p. 80). Alternatively, we could have made use of the class **ArrayList**, which differs only slightly from the class **Vector**.

```

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. B, p. 300). 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. 4.1, lines 8–16). This process is outlined in Fig. 4.3 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:

```

85 List<Corner> cleanupCorners(List<Corner> corners) {
86     // corners is assumed to be sorted by descending q
87     double dmin2 = dmin*dmin; //  $d_{\min}^2$  ( $d_{\min}$  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  $c_1$ 
```

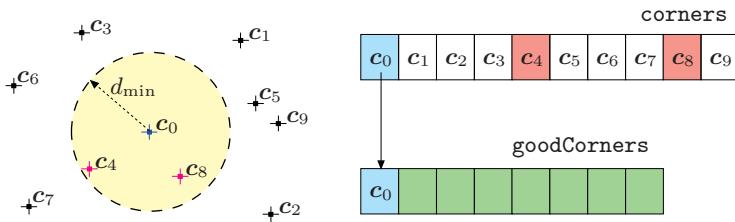


Figure 4.3 Selecting the strongest corners within a given spatial distance. The original list of corners (`corners`) is sorted by “corner strength” in descending order; i.e., c_0 is the strongest corner. First, corner c_0 is added to a new list `goodCorners`, while the weaker corners c_4 and c_8 (which are both within distance d_{\min} from c_0) are removed from the original `corners` list. The following corners c_1, c_2, \dots 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} .

```

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 $d^2(\mathbf{c}_1, \mathbf{c}_2) = (u_1 - u_2)^2 + (v_1 - v_2)^2$ between the corner points \mathbf{c}_1 and \mathbf{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.

4.3.3 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 }
```

4.3.4 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. B (pp. 294–300). 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.

4.4 Exercises

Exercise 4.1

Adapt the `draw()` method in the class `Corner` (p. 79) 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 4.2

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 4.3

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 4.4

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.

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 example, that a graphic artist has created an illustration with beautiful shades of color, for which he applied 150 different crayons. His editor likes the result but, for some technical reason, instructs the artist to draw the picture again, this time using only 10 different crayons. The artist now faces the problem of color quantization—his task is to select a “palette” of the 10 best suited from his 150 crayons and then choose the most similar color to redraw each stroke of his original picture.

In the general case, the original image I contains a set of m different colors $\mathcal{C} = \{\mathbf{C}_1, \mathbf{C}_2, \dots, \mathbf{C}_m\}$, where m could be only a few or several thousand, but at most 2^{24} for a 3×8 -bit color image. The goal is to replace the original colors by a (usually much smaller) set of colors $\mathcal{C}' = \{\mathbf{C}'_1, \mathbf{C}'_2, \dots, \mathbf{C}'_n\}$, with $n < m$. The difficulty lies in the proper choice of the reduced color palette \mathcal{C}' such that damage to the resulting image is minimized.

In practice, this problem is encountered, for example, when converting from full-color images to images with lower pixel depth or to index (“palette”) images, such as the conversion from 24-bit TIFF to 8-bit GIF images with only 256 (or fewer) colors. Until a few years ago, a similar problem had to be solved for displaying full-color images on computer screens because the available display memory was often limited to only 8 bits. Today, even the cheapest display hardware has at least 24-bit depth and therefore this particular need for (fast) color quantization no longer exists.

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 $[0 \dots m-1]$ is independently converted to the new range $[0 \dots n-1]$, in the simplest case by a linear quantization in the form

$$c'_i \leftarrow \left\lfloor c_i \cdot \frac{n}{m} \right\rfloor, \quad (5.1)$$

for all color components c_i . A typical example would be the conversion of a color image with 3×12 -bit components ($m = 4096$) to an RGB image with 3×8 -bit components ($n = 256$). In this case, each original component value is multiplied by $n/m = 256/4096 = 1/16 = 2^{-4}$ and subsequently truncated, which is equivalent to an integer division by 16 or simply ignoring the lower 4 bits of the corresponding binary values (Fig. 5.1 (a)).

m and n are usually the same for all color components but not always. An extreme (today rarely used) approach is to quantize 3×8 color vectors to single-byte (8-bit) colors, where 3 bits are used for red and green and only 2 bits for blue, as illustrated in Fig. 5.1 (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. 5.1. Naturally, due to the small number of colors available with this encoding (Fig. 5.2), 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. Scalar

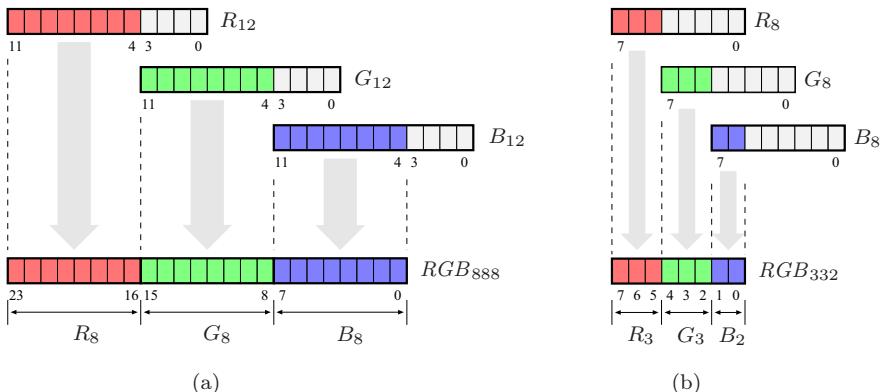


Figure 5.1 Scalar quantization of color components by truncating lower bits. Quantization of 3×12 -bit to 3×8 -bit colors (a). Quantization of 3×8 -bit to 3:3:2-packed 8-bit colors (b).

```

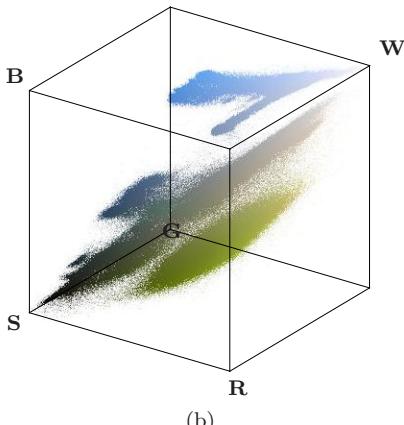
1 ColorProcessor cp = (ColorProcessor) ip;
2 int C = cp.getPixel(u, v); // 24-bit color pixel
3 int R = (C & 0x00FF0000) >> 16;
4 int G = (C & 0x0000FF00) >> 8;
5 int B = (C & 0x000000FF);
6 byte RGB = (byte) // 8-bit color pixel (3:3:2-packed)
7     (R & 0xE0 | (G & 0xE0) >> 3 | (B & 0xC0) >> 6);

```

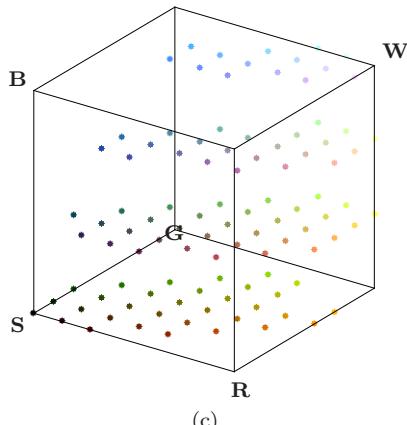
Program 5.1 3:3:2 quantization of a 24-bit RGB color pixel using bit operations (see also Fig. 5.1 (b) and Exercise 5.1).



(a)



(b)



(c)

Figure 5.2 Color distribution after a scalar 3:3:2 quantization. Original color image (a). Distribution of the original 226,321 colors (b) and the remaining $8 \times 8 \times 4 = 256$ colors after 3:3:2 quantization (c) in the RGB color cube.

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 quan-

tization 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.

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 $\mathcal{C} = \{\mathbf{C}_1, \mathbf{C}_2, \dots, \mathbf{C}_m\}$, the task of vector quantization is

- (a) to find a set of n representative color vectors $\mathcal{C}' = \{\mathbf{C}'_1, \mathbf{C}'_2, \dots, \mathbf{C}'_n\}$ 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.

5.2.1 Populosity algorithm

The populosity algorithm¹ [32] 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 Vol. 1 [14, Sec. 8.3.1], based on sorting the image pixels, 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.

5.2.2 Median-cut algorithm

The median-cut algorithm [32] 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

¹ Sometimes also called the “popularity” algorithm.

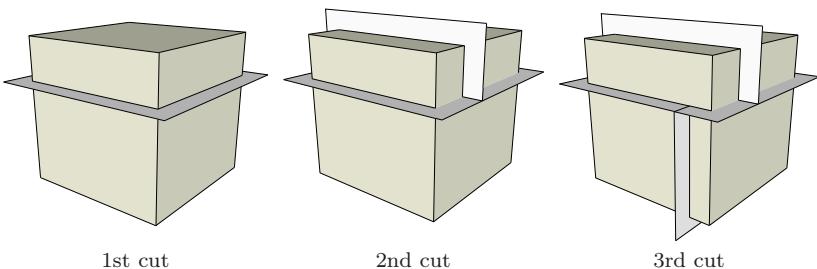


Figure 5.3 Median-cut algorithm. The RGB color space is recursively split into smaller cubes along one of the color axes.

as $32 \times 32 \times 32$) for efficiency reasons.² The initial histogram volume is then recursively split into smaller boxes until the desired number of representative colors is reached. In each recursive step, the color box representing the largest number of pixels is selected for splitting. A box is always split across the longest of its three axes at the median point, such that half of the contained pixels remain in each of the resulting subboxes (Fig. 5.3).

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 5.1–5.3 and a corresponding Java implementation can be found in the source code section of this book’s Website.

5.2.3 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 [26] 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 re-

² This corresponds to a scalar prequantization on the color components, which leads to additional quantization errors and thus produces suboptimal results. This step seems unnecessary on modern computers and should be avoided.

Algorithm 5.1 Median-cut color quantization (Part 1). 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.

```

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:  $\mathcal{C}_R \leftarrow \text{FINDREPRESENTATIVECOLORS}(I, K_{\max})$ 
3: return QUANTIZEIMAGE( $I, \mathcal{C}_R$ ) ▷ see Alg. 5.3

4: FINDREPRESENTATIVECOLORS( $I, K_{\max}$ )
   Returns a set of up to  $K_{\max}$  representative colors for the image  $I$ .
5: Let  $\mathcal{C} = \{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_K\}$  be the set of distinct colors in  $I$ . Each of
   the  $K$  color elements in  $\mathcal{C}$  is a tuple  $\mathbf{c}_i = (\text{red}_i, \text{grn}_i, \text{blu}_i, \text{cnt}_i)$ 
   consisting of the RGB color components (red, grn, blu) and the
   number of pixels (cnt) in  $I$  with that particular color.
6: if  $|\mathcal{C}| \leq K_{\max}$  then
7:   return  $\mathcal{C}$ .
8: else
   Create a color box  $\mathbf{b}_0$  at level 0 that contains all image colors  $\mathcal{C}$ 
   and make it the initial element in the set of color boxes  $\mathcal{B}$ :
9: Let  $\mathbf{b}_0 \leftarrow \text{CREATECOLORBOX}(\mathcal{C}, 0)$  ▷ see Alg. 5.2
10: Let  $\mathcal{B} \leftarrow \{\mathbf{b}_0\}$  ▷ initial set of color boxes
11: Let  $k \leftarrow 1$ 
12: Let  $done \leftarrow \text{false}$ 
13: while  $k < N_{\max}$  and not  $done$  do
14:    $\mathbf{b} \leftarrow \text{FINDBOXTOSPLIT}(\mathcal{B})$  ▷ see Alg. 5.2
15:   if  $\mathbf{b} \neq \text{nil}$  then
16:      $(\mathbf{b}_1, \mathbf{b}_2) \leftarrow \text{SPLITBOX}(\mathbf{b})$  ▷ see Alg. 5.2
17:      $\mathcal{B} \leftarrow \mathcal{B} - \{\mathbf{b}\}$  ▷ remove  $\mathbf{b}$  from  $\mathcal{B}$ 
18:      $\mathcal{B} \leftarrow \mathcal{B} \cup \{\mathbf{b}_1, \mathbf{b}_2\}$  ▷ insert  $\mathbf{b}_1, \mathbf{b}_2$  into  $\mathcal{B}$ 
19:      $k \leftarrow k + 1$ 
20:   else ▷ no more boxes to split
21:      $done \leftarrow \text{true}$ 
22:   Determine the average color inside each color box in set  $\mathcal{B}$ :
23:   Let  $\mathcal{C}_R \leftarrow \{\text{AVERAGECOLORS}(\mathbf{b}_j) \mid \mathbf{b}_j \in \mathcal{B}\}$  ▷ see Alg. 5.3
return  $\mathcal{C}_R$ .
```

Algorithm 5.2 Median-cut color quantization (Part 2).

-
- 1: **CREATECOLORBOX**(\mathcal{C}, m)

Creates and returns a new color box containing the colors \mathcal{C} . A color box b is a tuple $\langle \text{colors}, \text{level}, \text{rmin}, \text{rmax}, \text{gmin}, \text{gmax}, \text{bmin}, \text{bmax} \rangle$, where **colors** is the vector 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:

$$\left. \begin{array}{l} \text{Let } r_{\min} \leftarrow \min \text{red}(c) \\ \text{Let } r_{\max} \leftarrow \max \text{red}(c) \\ \text{Let } g_{\min} \leftarrow \min \text{grn}(c) \\ \text{Let } g_{\max} \leftarrow \max \text{grn}(c) \\ \text{Let } b_{\min} \leftarrow \min \text{blu}(c) \\ \text{Let } b_{\max} \leftarrow \max \text{blu}(c) \end{array} \right\} \text{for all colors } c \in \mathcal{C}$$
 - 3: Create a new color box: $b \leftarrow \langle \mathcal{C}, m, r_{\min}, r_{\max}, g_{\min}, g_{\max}, b_{\min}, b_{\max} \rangle$
 - 4: **return** 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.
 - 6: Let \mathcal{B}_s be the set of all color boxes that can be split (i.e., contain at least 2 different colors):

$$\mathcal{B}_s \leftarrow \{ b \mid b \in \mathcal{B} \wedge |\text{colors}(b)| \geq 2 \}$$
 - 7: **if** $\mathcal{B}_s = \{\}$ **then** ▷ no splittable box was found
 - 8: **return** **nil**.
 - 9: **else**
 - 10: Select a box $b_x \in \mathcal{B}_s$, such that **level**(b_x) is a minimum.
 - 11: **return** b_x .

 - 12: **SPLITBOX**(b)

Splits the color box b at the median plane perpendicular to its longest dimension and returns a pair of new color boxes.
 - 13: Let $m \leftarrow \text{level}(b)$
 - 14: Let $d \leftarrow \text{FINDMAXBOXDIMENSION}(b)$ ▷ see Alg. 5.3
 - 15: Let $\mathcal{C} \leftarrow \text{colors}(b)$ ▷ the set of colors in box 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: Let $b_1 \leftarrow \text{CREATECOLORBOX}(C_1, m + 1)$
 - 19: Let $b_2 \leftarrow \text{CREATECOLORBOX}(C_2, m + 1)$
 - 20: **return** (b_1, b_2) .

Algorithm 5.3 Median-cut color quantization (Part 3).

```

1: AVERAGECOLORS( $\mathbf{b}$ )
   Returns the average color  $\mathbf{c}_{\text{avg}}$  for the pixels represented by the color
   box  $\mathbf{b}$ .
2: Let  $\mathcal{C} \leftarrow \text{colors}(\mathbf{b})$                                  $\triangleright$  the set of colors in box  $\mathbf{b}$ 
3: Let  $n \leftarrow 0$ ,  $r_{\text{sum}} \leftarrow 0$ ,  $g_{\text{sum}} \leftarrow 0$ ,  $b_{\text{sum}} \leftarrow 0$ 
4: for all  $\mathbf{c} \in \mathcal{C}$  do
5:   Let  $k \leftarrow \text{cnt}(\mathbf{c})$ 
6:   Let  $n \leftarrow n + k$ 
7:   Let  $r_{\text{sum}} \leftarrow r_{\text{sum}} + k \cdot \text{red}(\mathbf{c})$ 
8:   Let  $g_{\text{sum}} \leftarrow g_{\text{sum}} + k \cdot \text{grn}(\mathbf{c})$ 
9:   Let  $b_{\text{sum}} \leftarrow b_{\text{sum}} + k \cdot \text{blu}(\mathbf{c})$ 
10:  Let  $r_{\text{avg}} \leftarrow \frac{1}{n} \cdot r_{\text{sum}}$ ,  $g_{\text{avg}} \leftarrow \frac{1}{n} \cdot g_{\text{sum}}$ ,  $b_{\text{avg}} \leftarrow \frac{1}{n} \cdot b_{\text{sum}}$ 
11:  Let  $\mathbf{c}_{\text{avg}} \leftarrow (r_{\text{avg}}, g_{\text{avg}}, b_{\text{avg}})$ 
12: return  $\mathbf{c}_{\text{avg}}$ .
```

```

13: FINDMAXBOXDIMENSION( $\mathbf{b}$ )
   Returns the largest dimension of the color box  $\mathbf{b}$  (Red, Green, or Blue).
14: Let  $\text{size}_r = \text{rmax}(\mathbf{b}) - \text{rmin}(\mathbf{b})$ 
15: Let  $\text{size}_g = \text{gmax}(\mathbf{b}) - \text{gmin}(\mathbf{b})$ 
16: Let  $\text{size}_b = \text{bmax}(\mathbf{b}) - \text{bmin}(\mathbf{b})$ 
17: Let  $\text{size}_{\text{max}} = \max(\text{size}_r, \text{size}_g, \text{size}_b)$ 
18: if  $\text{size}_{\text{max}} = \text{size}_r$  then
19:   return Red.
20: else if  $\text{size}_{\text{max}} = \text{size}_g$  then
21:   return Green.
22: else
23:   return Blue.
```

```

24: QUANTIZEIMAGE( $I, \mathcal{C}_R$ )
   Returns a new image with color pixels from  $I$  replaced by their closest
   representative colors in  $\mathcal{C}_R$ .
25: Create a new image  $I'$  the same size as  $I$ .
26: for all image coordinates  $(u, v)$  do
27:   Let  $\mathbf{c}$  be the color in  $\mathcal{C}_R$  that is “closest” to  $I(u, v)$  (e.g., using the
      Euclidean distance in RGB space).
28:    $I'(u, v) \leftarrow \mathbf{c}$ 
29: return  $I'$ .
```

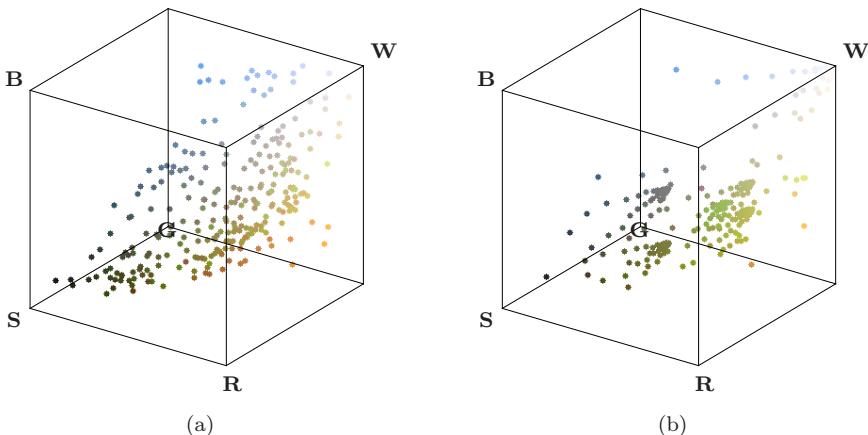


Figure 5.4 Color distribution after application of the median-cut (a) and octree (b) algorithms. In both cases, the set of 226,321 colors in the original image (Fig. 5.2(a)) was reduced to 256 representative colors.

duces 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. 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 required storage is small. The final replacement of the image pixels by the quantized color vectors can also be performed easily and efficiently with the octree structure because only up to eight comparisons (one at each tree layer) are necessary to locate the best-matching color for each pixel.

Figure 5.4 shows the resulting color distributions in RGB space after applying the median-cut and octree algorithms. In both cases, the original image (Fig. 5.2(a)) is quantized to 256 colors. Notice in particular the dense placement of quantized colors in certain regions of the green hues.

For both algorithms and the (scalar) 3:3:2 quantization, the resulting dis-

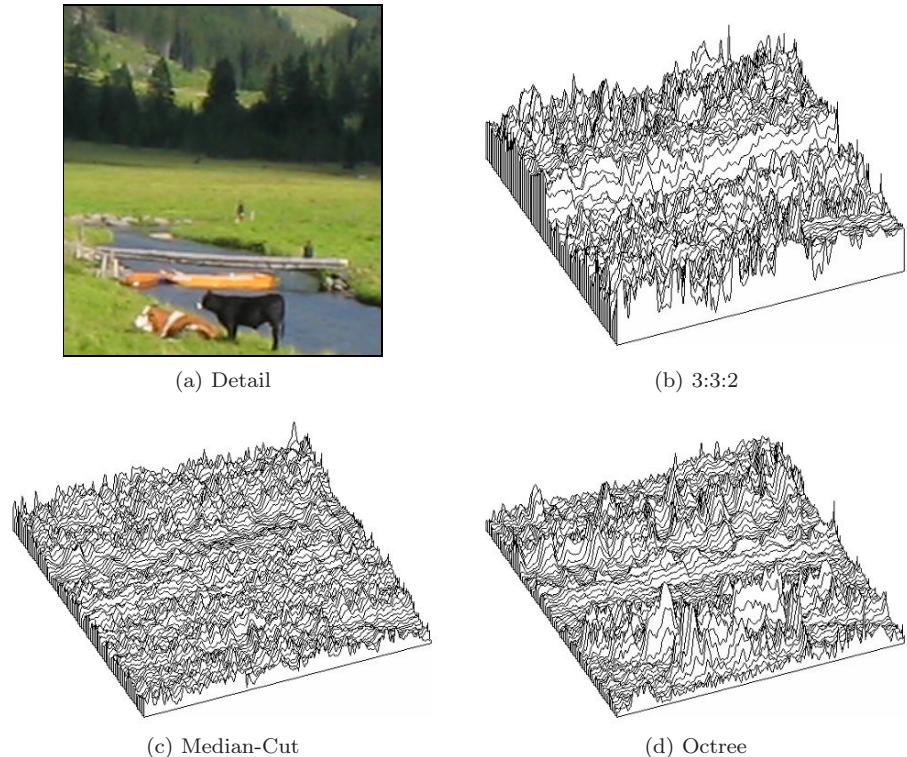


Figure 5.5 Quantization errors. Original image (a), distance between original and quantized color pixels for scalar 3:3:2 quantization (b), median-cut (c), and octree (d) algorithms.

tances between the original pixels and the quantized colors are shown in Fig. 5.5. The greatest error naturally results from 3:3:2 quantization, because this method does not consider the contents of the image at all. Compared with the median-cut method, the overall error for the octree algorithm is smaller, although the latter creates several large deviations, particularly inside the colored foreground regions and the forest region in the background. In general, however, the octree algorithm does not offer significant advantages in terms of the resulting image quality over the simpler median-cut algorithm.

5.2.4 Other methods for vector quantization

A suitable set of representative color vectors can usually be determined without inspecting all pixels in the original image. It is often sufficient to use only 10% of randomly selected pixels to obtain a high probability that none of the important colors is lost.

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 [67].

5.3 Exercises

Exercise 5.1

Simplify the 3:3:2 quantization given in Prog. 5.1 such that only a single bit mask/shift step is performed for each color component.

Exercise 5.2

The median-cut algorithm for color quantization (Sec. 5.2.2) is implemented in the *Independent JPEG Group's*³ `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.

³ www.ijg.org.

6

Colorimetric Color Spaces

In any application that requires precise, reproducible, and device-independent presentation of colors, the use of calibrated color systems is an absolute necessity. For example, color calibration is routinely used throughout the digital print work flow but also in digital film production, professional photography, image databases, etc. One may have experienced how difficult it is, for example, to render a good photograph on a color laser printer, and even the color reproduction on monitors largely depends on the particular manufacturer and computer system.

All the color spaces described in Vol. 1 [14, Sec. 8.2] 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., [80] or [66] for an overview). While several colorimetric standards exist, we focus on the most widely used CIE systems in the remaining part of this section.

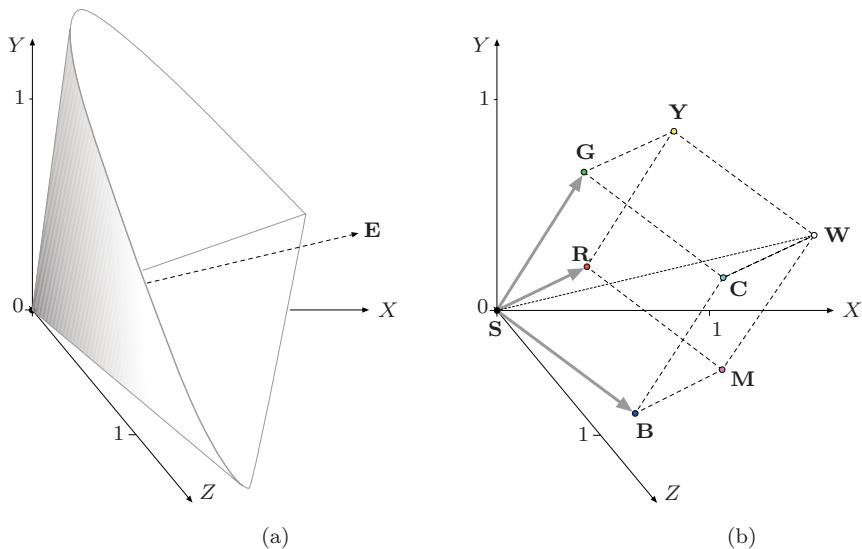


Figure 6.1 CIE XYZ color space. The XYZ color space is defined by the three imaginary primary colors X , Y , Z , where the Y dimension corresponds to the perceived luminance. All visible colors are contained inside an open, cone-shaped volume that originates at the black point S (a), where E denotes the axis of neutral (gray) colors. The RGB color space maps to the XYZ space as a linearly distorted cube (b).

6.1 CIE Color Spaces

The XYZ color system, developed by the CIE (Commission Internationale d’Éclairage)¹ in the 1920s and standardized in 1931, is the foundation of most colorimetric color systems that are in use today [60, p. 22].

6.1.1 CIE XYZ color space

The CIE XYZ color scheme was developed after extensive measurements of human visual perception under controlled conditions. It is based on three imaginary primary colors X , Y , Z , which are chosen such that all visible colors can be described as a summation of positive-only components, where the Y component corresponds to the perceived lightness or *luminosity* of a color. All visible colors lie inside a three-dimensional cone-shaped region (Fig. 6.1 (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. 6.3. Thus, as shown in Fig. 6.1 (b), the RGB color space is embedded in

¹ International Commission on Illumination (www.cie.co.at).

Table 6.1 Coordinates of the RGB color cube in CIE XYZ space. The X, Y, Z values refer to standard (ITU-R BT.709) primaries and white point D65 (see Table 6.2), 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

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 6.1.

6.1.2 CIE x, y chromaticity

As mentioned, the luminance in XYZ color space increases along the Y axis, starting at the black point **S** located at the coordinate origin ($X = Y = Z = 0$). The color hue is independent of the luminance and thus independent of the Y value. To describe the corresponding “pure” color hues and saturation in a convenient manner, the CIE system also defines the three *chromaticity* values

$$x = \frac{X}{X + Y + Z}, \quad y = \frac{Y}{X + Y + Z}, \quad z = \frac{Z}{X + Y + Z}, \quad (6.1)$$

where (obviously) $x + y + z = 1$ and thus one of the three values (e.g., z) is redundant. Equation (6.1) 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. 6.2). Thus, for an arbitrary XYZ color point $\mathbf{A} = (X_a, Y_a, Z_a)$, the corresponding chromaticity coordinates $\mathbf{a} = (x_a, y_a, z_a)$ are found by intersecting the line $\overline{\mathbf{SA}}$ with the $X + Y + Z = 1$ plane (Fig. 6.2 (a)). The final x, y coordinates are the result of projecting these intersection points onto the X/Y -plane (Fig. 6.2 (b)) by simply dropping the Z component z_a .

The result is the well-known horseshoe-shaped *CIE x, y chromaticity diagram*, which is shown in Fig. 6.2 (c). Any x, y point in this diagram defines the

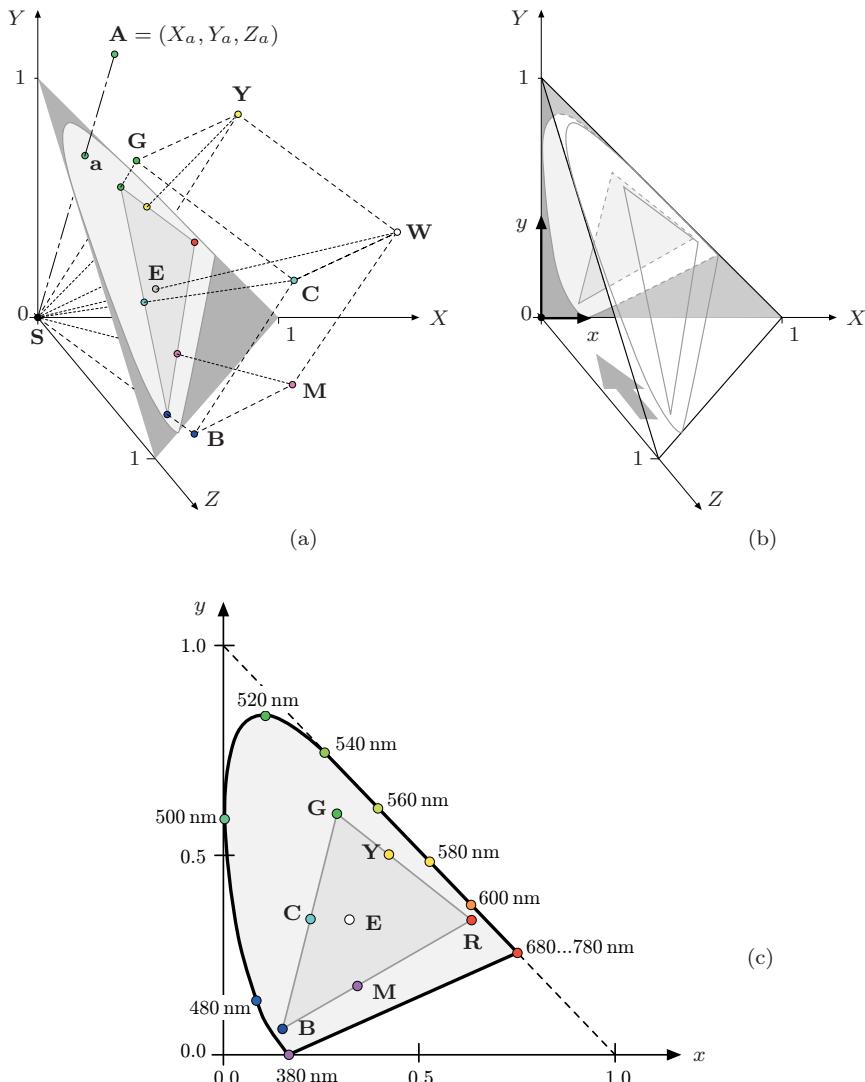


Figure 6.2 CIE x, y chromaticity diagram. For an arbitrary XYZ color point $\mathbf{A} = (X_a, Y_a, Z_a)$, the chromaticity values $\mathbf{a} = (x_a, y_a, z_a)$ are obtained by a central projection onto the 3D plane $X + Y + Z = 1$ (a). The corner points of the RGB cube map to a triangle, and its white point \mathbf{W} maps to the (colorless) neutral point \mathbf{E} . The intersection points are then projected onto the X/Y plane (b) by simply dropping the Z component, which produces the familiar CIE chromaticity diagram shown in (c). The CIE diagram contains all visible color tones (hues and saturations) but no luminance information, with wavelengths in the range 380–780 nanometers. A particular color space is specified by at least three primary colors (tristimulus values; e.g., \mathbf{R} , \mathbf{G} , \mathbf{B}), which define a triangle (linear hull) containing all representable colors.

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. (6.1)) as

$$X = x \cdot \frac{Y}{y}, \quad Z = z \cdot \frac{Y}{y} = (1 - x - y) \cdot \frac{Y}{y}. \quad (6.2)$$

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.

6.1.3 Standard illuminants

A central goal of colorimetry is the quantitative measurement of colors in physical reality, which strongly depends on the color properties of the illumination. The CIE system specifies a number of standard illuminants for a variety of real and hypothetical light sources, each specified by a spectral radiant power distribution and the “correlated color temperature” (expressed in degrees Kelvin) [80, Sec. 3.3.3]. The following daylight (D) illuminants are particularly important for the design of digital color spaces (Table 6.2):

D50 emulates the spectrum of natural (direct) sunlight with an equivalent color temperature of approximately 5000° K. D50 is the recommended

illuminant for viewing reflective images, such as paper prints. In practice, D50 lighting is commonly implemented with fluorescent lamps using multiple phosphors to approximate the specified color spectrum.

D65 has a correlated color temperature of approximately 6500°K and is designed to emulate the average (indirect) daylight observed under an overcast sky on the northern hemisphere. D65 is also used as the reference white for emissive devices, such as display screens.

The standard illuminants serve to specify the ambient viewing light but also to define the reference white points in various color spaces in the CIE color system. For example, the sRGB standard (see Sec. 6.3) 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 $\pm 2^{\circ}$).

Table 6.2 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

6.1.4 Gamut

The set of all colors that can be handled by a certain media device or can be represented by a particular color space is called “gamut”. This is usually a contiguous region in the 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 6.3 illustrates some typical gamut regions inside the CIE diagram.

The gamut of an output device mainly depends on the technology employed. For example, ordinary color monitors are typically not capable of displaying all colors of the gamut covered by the corresponding color space (usually sRGB). Conversely, it is also possible that devices would reproduce certain colors that cannot be represented in the utilized color space. Significant deviations exist, for example, between the RGB color space and the gamuts associated with CMYK-based printers. Also, media devices with very large gamuts exist, as demonstrated by the laser display system in Fig. 6.3. Representing such large gamuts and, in particular, transforming between different color representations

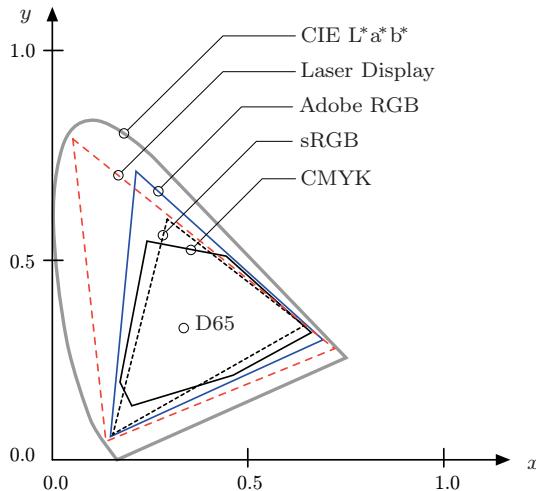


Figure 6.3 Gamut regions for different color spaces and output devices inside the CIE diagram.

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.

6.1.5 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_bC_r , and particularly $L^*a^*b^*$, which is described below.

In addition, CIE-compliant specifications exist for most common color spaces (see Vol. 1 [14, Sec. 8.2]), which allow more or less dependable conversions between almost any pair of color spaces.

6.2 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*² 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 non-linear correction function (similar to the modified gamma correction described in Vol. 1 [14, Sec. 4.7.6]) is applied to all three components, as detailed below.

6.2.1 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 [42] as follows:

$$\begin{aligned} L^* &= 116 \cdot Y' - 16, \\ a^* &= 500 \cdot (X' - Y'), \\ b^* &= 200 \cdot (Y' - Z'), \end{aligned} \tag{6.3}$$

$$\begin{aligned} \text{where } X' &= f_1\left(\frac{X}{X_{\text{ref}}}\right), & Y' &= f_1\left(\frac{Y}{Y_{\text{ref}}}\right), & Z' &= f_1\left(\frac{Z}{Z_{\text{ref}}}\right), \\ \text{and } f_1(c) &= \begin{cases} c^{\frac{1}{3}} & \text{for } c > 0.008856 \\ 7.787 \cdot c + \frac{16}{116} & \text{for } c \leq 0.008856. \end{cases} \end{aligned}$$

Usually D65 is specified as the reference white point $(X_{\text{ref}}, Y_{\text{ref}}, Z_{\text{ref}})$ (see Table 6.2). 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 $[-127, +127]$.

² Often L*a*b* is simply referred to as the “Lab” color space.

Table 6.3 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 6.1 and 6.2).

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

6.2.2 Transformation L*a*b* → CIE XYZ

The reverse transformation from L*a*b* space to XYZ coordinates is defined as follows:

$$\begin{aligned} X &= X_{\text{ref}} \cdot f_2\left(\frac{a^*}{500} + Y'\right), \\ Y &= Y_{\text{ref}} \cdot f_2(Y'), \\ Z &= Z_{\text{ref}} \cdot f_2\left(Y' - \frac{b^*}{200}\right), \end{aligned} \quad (6.4)$$

$$\begin{aligned} \text{where } Y' &= \frac{L^* + 16}{116} \\ \text{and } f_2(c) &= \begin{cases} c^3 & \text{for } c^3 > 0.008856 \\ \frac{c - 16/116}{7.787} & \text{for } c^3 \leq 0.008856. \end{cases} \end{aligned}$$

The complete Java code for the L*a*b*/XYZ conversion and the implementation of the associated `ColorSpace` class can be found in Progs. 6.1 and 6.2 (pp. 121–122).

Table 6.3 lists the relation between L*a*b* and XYZ coordinates for selected RGB colors. Figure 6.4 shows the separation of a color image into the corresponding L*a*b* components.

6.2.3 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) [29, p. 57]. The difference between two color points \mathbf{C}_1 and \mathbf{C}_2 can be found by simply measuring the *Euclidean*



Figure 6.4 $L^*a^*b^*$ components shown as grayscales images. The contrast of the a^* and b^* images has been increased by 40% for better viewing.

distance in $L^*a^*b^*$ space,

$$\begin{aligned} \text{ColorDist}_{\text{Lab}}(\mathbf{C}_1, \mathbf{C}_2) &= \|\mathbf{C}_1 - \mathbf{C}_2\| \\ &= \sqrt{(L_1^* - L_2^*)^2 + (a_1^* - a_2^*)^2 + (b_1^* - b_2^*)^2}, \end{aligned} \quad (6.5)$$

where $\mathbf{C}_1 = (L_1^*, a_1^*, b_1^*)$ and $\mathbf{C}_2 = (L_2^*, a_2^*, b_2^*)$.

6.3 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” [41]) 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. 6.3), which, however, includes most colors that can be reproduced by current computer and video monitors. Although sRGB was not designed as a universal color space, its CIE-based specification at least permits more or less exact conversions to and from other color spaces.

Table 6.4 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 non-linear 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

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 [34]. 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 [71]. 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 [70]. 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. 6.6 below).

Table 6.4 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 [44] (see Tables 6.1 and 6.2). Together, these values permit the unambiguous mapping of all other colors in the CIE diagram.

6.3.1 Linear vs. nonlinear color components

sRGB is a *nonlinear* color space with respect to the XYZ coordinate system, and it is important to carefully distinguish between the *linear* and *nonlinear* RGB component values. The nonlinear values (denoted R', G', B') represent the actual color tuples, the data values read from an image file or received from a digital camera. These values are 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,

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = M_{\text{RGB}} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = M_{\text{RGB}}^{-1} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}, \quad (6.6)$$

respectively, with

$$\mathbf{M}_{\text{RGB}} = \begin{pmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix}, \quad (6.7)$$

$$\mathbf{M}_{\text{RGB}}^{-1} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix}. \quad (6.8)$$

Notice that the three column vectors of $\mathbf{M}_{\text{RGB}}^{-1}$ (Eqn. (6.8)) are the coordinates of the primary colors **R**, **G**, **B** (tristimulus values) in XYZ space (cf. Table 6.4) and thus

$$\mathbf{R} = \mathbf{M}_{\text{RGB}}^{-1} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{G} = \mathbf{M}_{\text{RGB}}^{-1} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \mathbf{B} = \mathbf{M}_{\text{RGB}}^{-1} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}. \quad (6.9)$$

6.3.2 Transformation CIE XYZ→sRGB

To transform a given XYZ color to sRGB (Fig. 6.5), we first compute the *linear* R, G, B values by multiplying the (X, Y, Z) coordinate vector with the matrix \mathbf{M}_{RGB} (Eqn. (6.7)),

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \mathbf{M}_{\text{RGB}} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}. \quad (6.10)$$

Subsequently, a modified gamma correction (see Vol. 1 [14, Sec. 4.7.6]) with $\gamma = 2.4$ (which corresponds to an effective gamma value of ca. 2.2) is applied to the linear R, G, B values,

$$R' = f_\gamma(R), \quad G' = f_\gamma(G), \quad B' = f_\gamma(B),$$

$$\text{with } f_\gamma(c) = \begin{cases} 1.055 \cdot c^{\frac{1}{2.4}} - 0.055 & \text{for } c > 0.0031308 \\ 12.92 \cdot c & \text{for } c \leq 0.0031308. \end{cases} \quad (6.11)$$

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 $[0, 255]$.

6.3.3 Transformation sRGB→CIE XYZ

To compute the reverse transformation from sRGB to XYZ, the given (non-linear) R', G', B' values (in the range $[0, 1]$) are first linearized by inverting the

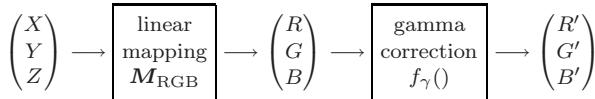


Figure 6.5 Color transformation from CIE XYZ to sRGB.

gamma correction³ (Eqn. (6.11)),

$$R = f_{\gamma}^{-1}(R'), \quad G = f_{\gamma}^{-1}(G'), \quad B = f_{\gamma}^{-1}(B'), \quad (6.12)$$

$$\text{with } f_{\gamma}^{-1}(c') = \begin{cases} \left(\frac{c'+0.055}{1.055}\right)^{2.4} & \text{for } c' > 0.03928 \\ \frac{c'}{12.92} & \text{for } c' \leq 0.03928. \end{cases} \quad (6.13)$$

Subsequently, the linearized (R, G, B) vector is transformed to XYZ coordinates by multiplication with the inverse of the matrix M_{RGB} (Eqn. (6.8)); i. e.,

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = M_{\text{RGB}}^{-1} \begin{pmatrix} R \\ G \\ B \end{pmatrix}. \quad (6.14)$$

Table 6.5 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.

6.3.4 Calculating with sRGB values

Due to the wide use of sRGB in digital photography, graphics, multimedia, Internet imaging, etc., there is a probability that a given image is encoded in sRGB colors. If, for example, a JPEG image is opened with ImageJ or Java, the pixel values in the resulting data array are media-oriented (i. e., nonlinear R', G', B' components of the sRGB color space). Unfortunately, this fact is often overlooked by programmers, with the consequence that colors are incorrectly manipulated and reproduced.

As a general rule, any arithmetic operation on color values should always be performed on the *linearized* R, G, B components, which are obtained from the nonlinear R', G', B' values through the inverse gamma function f_{γ}^{-1} (Eqn. (6.13)) and converted back again with f_{γ} (Eqn. (6.11)).

³ See Eqn. (4.35) in Vol. 1 [14, p. 86] for a general formulation of the inverse modified gamma function.

Table 6.5 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 **R₂₅**).

Pt.	Color	sRGB nonlinear			sRGB linearized			CIE XYZ		
		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
R₇₅	75% red	0.75	0.0	0.0	0.5225	0.0000	0.0000	0.2155	0.1111	0.0101
R₅₀	50% red	0.50	0.0	0.0	0.2140	0.0000	0.0000	0.0883	0.0455	0.0041
R₂₅	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

Example: color to grayscale conversion

The principle of converting RGB colors to grayscale values by computing a weighted sum of the color components was described already in Vol. 1 [14, Sec. 8.2.1], where we had simply ignored the issue of possible nonlinearities. As one may probably have guessed, however, the variables R , G , B , and Y in Eqn. (8.7) of Vol. 1 (p. 203),

$$Y = 0.2125 \cdot R + 0.7154 \cdot G + 0.0721 \cdot B, \quad (6.15)$$

implicitly refer to *linear* color and gray values, respectively, and not the raw sRGB values! Based on Eqn. (6.15), the *correct* grayscale conversion from raw (nonlinear) sRGB components R' , G' , B' is

$$Y' = f_\gamma \left[0.2125 \cdot f_\gamma^{-1}(R') + 0.7154 \cdot f_\gamma^{-1}(G') + 0.0721 \cdot f_\gamma^{-1}(B') \right], \quad (6.16)$$

with $f_\gamma()$ and $f_\gamma^{-1}()$ as defined in Eqns. (6.11) and (6.13). 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' ,

$$(R', G', B') \rightarrow (Y', Y', Y'),$$

always creates a gray (colorless) pixel, despite the nonlinearities of the sRGB space. This is due to the fact that the gamma correction (Eqns. (6.11) and (6.13)) 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. 6.1(b)).

For many applications, however, the following *approximation* to the exact grayscale conversion in Eqn. (6.16) is sufficient. It works without converting the sRGB values (i.e., directly on the nonlinear R', G', B' components) by computing a linear combination

$$Y' \approx w'_R \cdot R' + w'_G \cdot G' + w'_B \cdot B' \quad (6.17)$$

with a modified set of weights; e.g., $w'_R = 0.309$, $w'_G = 0.609$, $w'_B = 0.082$, as proposed in [58].

6.4 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. 6.3), which extends its use particularly to print applications. Figure 6.6 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.

6.5 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,

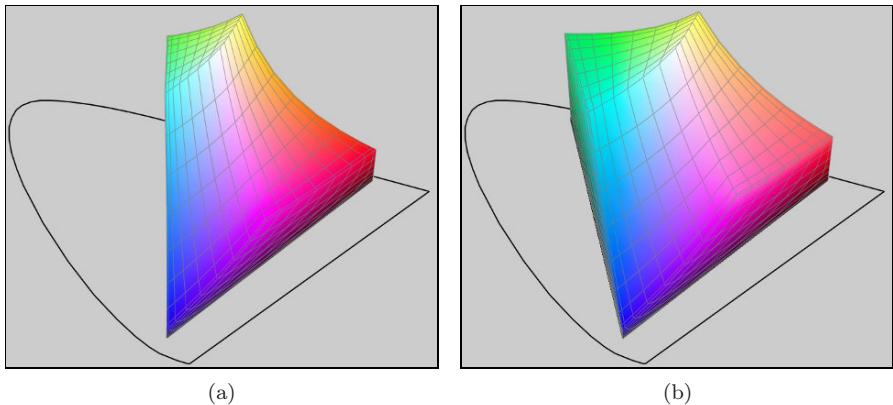


Figure 6.6 Gamuts for sRGB (a) and Adobe RGB (b) in the three-dimensional CIE XYZ color space.

although the spectral composition of the light that enters the eye is completely different in both situations. The CIE color system takes into account the color temperature of the ambient lighting because the exact interpretation of XYZ color values also requires knowledge of the corresponding reference white point. For example, a color value (X, Y, Z) specified with respect to the D50 reference white point is generally perceived differently when reproduced by a D65-based media device, although the absolute (i.e., measured) color is the same. Thus the actual meaning of XYZ values cannot be known without knowing the corresponding white point. This is known as *relative colorimetry*.

If colors are specified with respect to *different* white points, for example $\mathbf{W}_1 = (X_{W1}, Y_{W1}, Z_{W1})$ and $\mathbf{W}_2 = (X_{W2}, Y_{W2}, Z_{W2})$, they can be related by first applying a so-called *chromatic adaptation transformation* (CAT) [38, 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 .

6.5.1 XYZ scaling

The simplest chromatic adaptation method is XYZ scaling, where the individual color coordinates are individually multiplied by the ratios of the corresponding white point coordinates:

$$X_2 = X_1 \cdot \frac{X_{W2}}{X_{W1}}, \quad Y_2 = Y_1 \cdot \frac{Y_{W2}}{Y_{W1}}, \quad Z_2 = Z_1 \cdot \frac{Z_{W2}}{Z_{W1}}. \quad (6.18)$$

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 6.2), the result-

ing transformation is

$$\begin{aligned} X_{50} &= X_{65} \cdot \frac{X_{\text{D50}}}{X_{\text{D65}}} = X_{65} \cdot \frac{0.964296}{0.950456} = X_{65} \cdot 1.01456, \\ Y_{50} &= Y_{65} \cdot \frac{Y_{\text{D50}}}{Y_{\text{D65}}} = Y_{65} \cdot \frac{1.000000}{1.000000} = Y_{65}, \\ Z_{50} &= Z_{65} \cdot \frac{Z_{\text{D50}}}{Z_{\text{D65}}} = Z_{65} \cdot \frac{0.825105}{1.088754} = Z_{65} \cdot 0.757843. \end{aligned} \quad (6.19)$$

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.

6.5.2 Bradford adaptation

The most common chromatic adaptation models are based on scaling the color coordinates not directly in XYZ but in a “virtual” $R^*G^*B^*$ color space obtained from the XYZ values by a linear transformation

$$\begin{pmatrix} R^* \\ G^* \\ B^* \end{pmatrix} = \mathbf{M}_{\text{CAT}} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}, \quad (6.20)$$

where \mathbf{M}_{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

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \mathbf{M}_{\text{CAT}}^{-1} \cdot \begin{pmatrix} \frac{R_{\text{W2}}^*}{R_{\text{W1}}^*} & 0 & 0 \\ 0 & \frac{G_{\text{W2}}^*}{G_{\text{W1}}^*} & 0 \\ 0 & 0 & \frac{B_{\text{W2}}^*}{B_{\text{W1}}^*} \end{pmatrix} \cdot \mathbf{M}_{\text{CAT}} \cdot \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix}, \quad (6.21)$$

where $\frac{R_{\text{W2}}^*}{R_{\text{W1}}^*}$, $\frac{G_{\text{W2}}^*}{G_{\text{W1}}^*}$, $\frac{B_{\text{W2}}^*}{B_{\text{W1}}^*}$ are the (constant) ratios of the $R^*G^*B^*$ values of the white points \mathbf{W}_2 , \mathbf{W}_1 , respectively; i.e.,

$$\begin{pmatrix} R_{\text{W1}}^* \\ G_{\text{W1}}^* \\ B_{\text{W1}}^* \end{pmatrix} = \mathbf{M}_{\text{CAT}} \cdot \begin{pmatrix} X_{\text{W1}} \\ Y_{\text{W1}} \\ Z_{\text{W1}} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} R_{\text{W2}}^* \\ G_{\text{W2}}^* \\ B_{\text{W2}}^* \end{pmatrix} = \mathbf{M}_{\text{CAT}} \cdot \begin{pmatrix} X_{\text{W2}} \\ Y_{\text{W2}} \\ Z_{\text{W2}} \end{pmatrix}.$$

The popular “Bradford” model [38, p. 590] for chromatic adaptation specifies the transformation matrix

$$\mathbf{M}_{\text{CAT}} = \begin{pmatrix} 0.8951 & 0.2664 & -0.1614 \\ -0.7502 & 1.7135 & 0.0367 \\ 0.0389 & -0.0685 & 1.0296 \end{pmatrix}. \quad (6.22)$$

Inserting this particular M_{CAT} matrix in Eqn. (6.21) 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 6.2) is

$$\begin{aligned} \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} &= M_{50|65} \cdot \begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix} \\ &= \begin{pmatrix} 1.047884 & 0.022928 & -0.050149 \\ 0.029603 & 0.990437 & -0.017059 \\ -0.009235 & 0.015042 & 0.752085 \end{pmatrix} \cdot \begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix}, \end{aligned} \quad (6.23)$$

and conversely from D50-based to D65-based colors (i. e., $\mathbf{W}_1 = \mathbf{D50}$, $\mathbf{W}_2 = \mathbf{D65}$),

$$\begin{aligned} \begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix} &= M_{65|50} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} = M_{50|65}^{-1} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} \\ &= \begin{pmatrix} 0.955513 & -0.023079 & 0.063190 \\ -0.028348 & 1.009992 & 0.021019 \\ 0.012300 & -0.020484 & 1.329993 \end{pmatrix} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix}. \end{aligned} \quad (6.24)$$

Fig. 6.7 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 6.6.

The Bradford model is a widely used chromatic adaptation scheme but several similar procedures have been proposed (see also Exercise 6.1). Generally speaking, chromatic adaptation and related problems have a long history in color engineering and are still active fields of scientific research [80, Sec. 5.12].

6.6 Colorimetric Support in Java

6.6.1 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. 6.5). 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. (6.11) and (6.13)).

Table 6.6 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. (6.23).

Pt.	Color	sRGB			XYZ (D65)			XYZ (D50)		
		R'	G'	B'	X_{65}	Y_{65}	Z_{65}	X_{50}	Y_{50}	Z_{50}
S	black	0.00	0.0	0.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
R	red	1.00	0.0	0.0	0.4125	0.2127	0.0193	0.4361	0.2225	0.0139
Y	yellow	1.00	1.0	0.0	0.7700	0.9278	0.1385	0.8212	0.9394	0.1110
G	green	0.00	1.0	0.0	0.3576	0.7152	0.1192	0.3851	0.7169	0.0971
C	cyan	0.00	1.0	1.0	0.5380	0.7873	1.0694	0.5282	0.7775	0.8112
B	blue	0.00	0.0	1.0	0.1804	0.0722	0.9502	0.1431	0.0606	0.7141
M	magenta	1.00	0.0	1.0	0.5929	0.2848	0.9696	0.5792	0.2831	0.7280
W	white	1.00	1.0	1.0	0.9505	1.0000	1.0888	0.9643	1.0000	0.8251
K	50% gray	0.50	0.5	0.5	0.2034	0.2140	0.2330	0.2064	0.2140	0.1766
R₇₅	75% red	0.75	0.0	0.0	0.2155	0.1111	0.0101	0.2279	0.1163	0.0073
R₅₀	50% red	0.50	0.0	0.0	0.0883	0.0455	0.0041	0.0933	0.0476	0.0030
R₂₅	25% red	0.25	0.0	0.0	0.0210	0.0108	0.0010	0.0222	0.0113	0.0007
P	pink	1.00	0.5	0.5	0.5276	0.3812	0.2482	0.5492	0.3889	0.1876

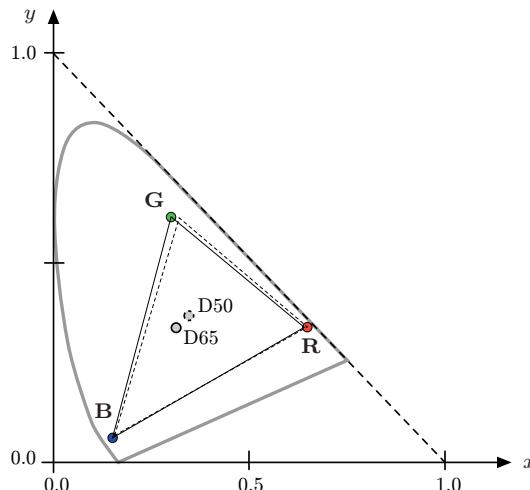


Figure 6.7 Bradford chromatic adaptation from white point D65 to D50. The solid triangle represents the original RGB gamut for white point D65, with the primaries (**R**, **G**, **B**) located at the corner points. The dashed triangle is the corresponding gamut after chromatic adaptation to white point D50.

6.6.2 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 sys-

Table 6.7 Color coordinates for sRGB primaries and the white point in Java’s default XYZ color space. The white point **W** is equal to D50.

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

tem is designed after the ICC⁴ “color management architecture”, which uses a CIE XYZ-based device-independent color space called the “profile connection space” (PCS) [40,43]. 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. 6.6.5) 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 6.8 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 6.2). The reason is that the ICC standard was developed primarily for color management in photography, graphics, and printing, where D50 is normally used as the reflective media white point. The Java methods `fromCIEXYZ()` and `toCIEXYZ()` thus take and return X , Y , Z color coordinates that are relative to the D50 white point. The resulting coordinates for the primary colors (listed in Table 6.7) are different from the ones given for white point D65 (see Table 6.4)! 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. 6.5) is used to convert between XYZ color coordinates that are measured with respect to different white points. The ICC specification [40] recommends a linear chromatic adaptation based on the Bradford model to convert between the D65-related XYZ coordinates (X_{65} , Y_{65} , Z_{65}) and D50-related values (X_{50} , Y_{50} , Z_{50}). This is also implemented by the Java API.

⁴ International Color Consortium (ICC, www.color.org).

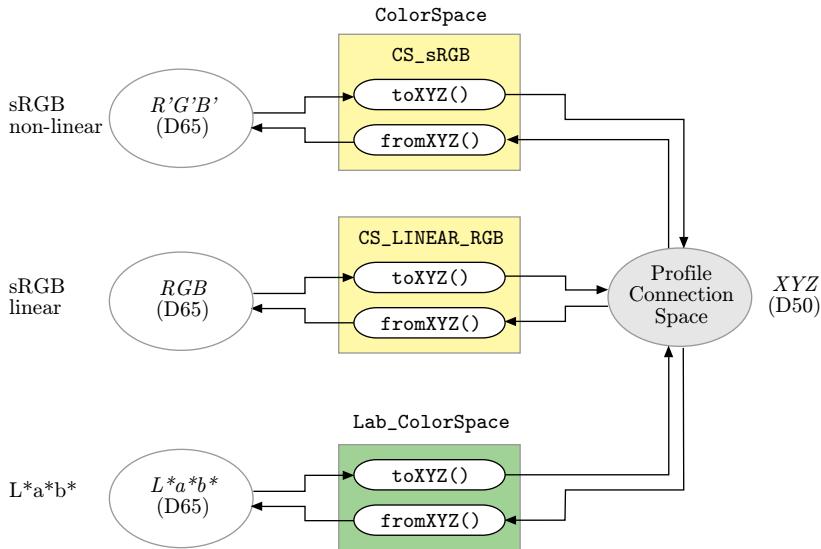


Figure 6.8 XYZ-based color conversion in Java. **ColorSpace** objects implement the methods **fromCIEXYZ()** and **toCIEXYZ()** to convert color vectors from and to the CIE XYZ color space, respectively. Colorimetric transformations between color spaces can be accomplished as a two-step process via the 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.

The complete mapping between the linearized sRGB color values (R, G, B) and the D50-based (X_{50}, Y_{50}, Z_{50}) coordinates can be expressed as a linear transformation composed of the $\text{RGB} \rightarrow \text{XYZ}_{65}$ transformation by matrix M_{RGB} (Eqns. (6.7) and (6.8)) and the chromatic adaptation transformation $\text{XYZ}_{65} \rightarrow \text{XYZ}_{50}$ defined by the matrix $M_{50|65}$ (Eqn. (6.23)),

$$\begin{aligned}
 \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} &= M_{50|65} \cdot M_{\text{RGB}}^{-1} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} \\
 &= (M_{\text{RGB}} \cdot M_{65|50})^{-1} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} \\
 &= \begin{pmatrix} 0.436131 & 0.385147 & 0.143033 \\ 0.222527 & 0.716878 & 0.060600 \\ 0.013926 & 0.097080 & 0.713871 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}, \tag{6.25}
 \end{aligned}$$

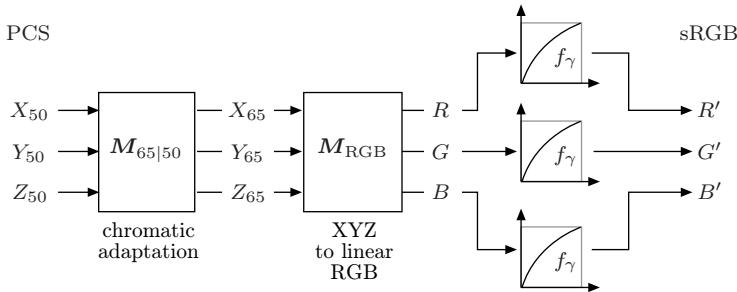


Figure 6.9 Transformation from D50-based PCS coordinates (X_{50}, Y_{50}, Z_{50}) to nonlinear sRGB values (R', G', B').

and, in the reverse direction,

$$\begin{aligned} \begin{pmatrix} R \\ G \\ B \end{pmatrix} &= M_{\text{RGB}} \cdot M_{65|50} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} \\ &= \begin{pmatrix} 3.133660 & -1.617140 & -0.490588 \\ -0.978808 & 1.916280 & 0.033444 \\ 0.071979 & -0.229051 & 1.405840 \end{pmatrix} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix}. \end{aligned} \quad (6.26)$$

Equations (6.25) and (6.26) 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 6.9 illustrates the complete transformation from D50-based PCS coordinates to nonlinear sRGB values.

6.6.3 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 Vol. 1 [14, Sec. 8.1.2] (see Prog. 8.3 on p. 196).

Class Color (java.awt.Color)

An object of class `Color` describes a particular color in the associated color space, which defines the number and type of the color components. `Color` objects are primarily used for graphic operations, such as to specify the color for drawing or filling graphic objects. Unless the color space is not explicitly specified, new `Color` objects are created as sRGB colors. The arguments passed to the `Color` constructor methods may be either `float` components in the range [0, 1] or integers in the range [0, 255], as demonstrated by the following example:

```
1 Color pink = new Color(1.0f, 0.5f, 0.5f);
2 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 HSV⁵ colors (see Sec. 8.2.3 of Vol. 1 [14, p. 209]).

Class ColorSpace (java.awt.color.ColorSpace)

An object of type `ColorSpace` represents an entire color space, such as sRGB or CMYK. Every subclass of `ColorSpace` (which itself is an abstract class) provides methods for converting its native colors to the 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:

```
1 // create an sRGB color space object:
2 ColorSpace sRGBcsp
3     = ColorSpace.getInstance(ColorSpace.CS_sRGB);
4 float[] pink_RGB = new float[] {1.0f, 0.5f, 0.5f};
5 // convert from sRGB to XYZ:
6 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

⁵ The HSV color space is referred to as “HSB” (hue, saturation, *brightness*) in the Java API.

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 [41],
- `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 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.

6.6.4 A $L^*a^*b^*$ 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. 6.2. 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. 6.1 and 6.2). These conversions are performed in two steps via XYZ coordinates, where care must be taken regarding the right choice of the associated white point ($L^*a^*b^*$ 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
4 // sRGB→ $L^*a^*b^*$ :
5 float[] cyan_LAB = LABcsp.fromRGB(cyan_sRGB)
6
7 //  $L^*a^*b^*$ →XYZ:
8 float[] cyan_XYZ = LABcsp.toXYZ(cyan_LAB);

```

```

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 // continued...

```

Program 6.1 Java implementation of the L*a*b* color space (Part 1). `Lab_ColorSpace` is a subclass of the standard AWT class `ColorSpace`. The conversion from the profile connection space (XYZ) to L*a*b* (Eqn. (6.3)) 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 Alg. 6.2, line 52) performs the required gamma correction (lines 27–29). The definitions of the classes `Illuminant`, `ChromaticAdaptation`, and `BradfordAdaptation` can be found in the source code section of the book's Website.

6.6.5 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

```

38 // class Lab_ColorSpace (continued)
39
40 // CIELab→XYZ: returns D50-related XYZ values
41 // from D65-related L*a*b* values:
42 public float[] toCIEXYZ(float[] Lab) {
43     double yy = (Lab[0] + 16) / 116;
44     float X65 = (float)(Xref * f2(Lab[1] / 500 + yy));
45     float Y65 = (float)(Yref * f2(yy));
46     float Z65 = (float)(Zref * f2(yy - Lab[2] / 200));
47     float[] XYZ65 = new float[] {X65, Y65, Z65};
48     return catD65toD50.apply(XYZ65);
49 }
50
51 // Gamma correction (forward)
52 double f1(double c) {
53     if (c > 0.008856)
54         return Math.pow(c, 1.0 / 3);
55     else
56         return (7.787 * c) + (16.0 / 116);
57 }
58
59 // Gamma correction (inverse)
60 double f2(double c) {
61     double c3 = Math.pow(c, 3.0);
62     if (c3 > 0.008856)
63         return c3;
64     else
65         return (c - 16.0 / 116) / 7.787;
66 }
67
68 //sRGB→CIELab
69 public float[] fromRGB(float[] sRGB) {
70     float[] XYZ50 = sRGBcs.toCIEXYZ(sRGB);
71     return this.fromCIEXYZ(XYZ50);
72 }
73
74 //CIELab→sRGB
75 public float[] toRGB(float[] Lab) {
76     float[] XYZ50 = this.toCIEXYZ(Lab);
77     return sRGBcs.fromCIEXYZ(XYZ50);
78 }
79
80 } // end of class LabColorSpace

```

Program 6.2 Java implementation of the L*a*b* color space (Part 2). The method `toCIEXYZ()` implements the reverse transformation from L*a*b* to XYZ, where the method `f2()` (defined in line 60) does the inverse gamma correction (lines 44–46), followed by the chromatic adaptation from D65 to D50 in line 48. Auxiliary methods `f1()` and `f2()` implement the forward and inverse gamma corrections, respectively (as defined in Eqns. (6.3) and (6.4)). The methods `toRGB()` and `fromRGB()` perform the conversions to and from sRGB in two steps via XYZ coordinates.

accurately on different media. The contents and the format of ICC profile files is specified in [40], which is identical to ISO standard 15076 [43]. Profiles are thus a key element in the process of digital color management [76].

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.⁶

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.⁷ For standard color spaces, the associated ICC profiles are often available as part of the computer installation, such as `CIELAB.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
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.

⁶ 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., $col \neq csp.fromCIEXYZ(csp.toCIEXYZ(col))$ for a color space object `csp`. (This has been a documented Java problem for some time.) A “clean” implementation of the sRGB color space can be found in the source code section of this book’s Website.

⁷ ICC profile files may also come with extensions `.icm` or `.pf` (as in the Java distribution).

6.7 Exercises

Exercise 6.1

For chromatic adaptation (defined in Eqn. (6.21)), transformation matrices other than the Bradford model (Eqn. (6.22)) have been proposed; e.g. [71],

$$\begin{aligned} \mathbf{M}_{\text{Sharp}} &= \begin{pmatrix} 1.2694 & -0.0988 & -0.1706 \\ -0.8364 & 1.8006 & 0.0357 \\ 0.0297 & -0.0315 & 1.0018 \end{pmatrix} \text{ and} \\ \mathbf{M}_{\text{CMC}} &= \begin{pmatrix} 0.7982 & 0.3389 & -0.1371 \\ -0.5918 & 1.5512 & 0.0406 \\ 0.0008 & -0.0239 & 0.9753 \end{pmatrix}. \end{aligned}$$

Derive the complete chromatic adaptation transformations $\mathbf{M}_{50|65}$ and $\mathbf{M}_{65|50}$ for converting between D65 and D50 colors, analogous to Eqns. (6.23) and (6.24), for each of the transformation matrices above.

Exercise 6.2

Implement the conversion of an sRGB color image to a colorless (grayscale) sRGB image using the three methods in Eqns. (6.15) (incorrectly applying standard weights to nonlinear $R'G'B'$ components), (6.16) (exact computation), and (6.17) (approximation using nonlinear components and modified weights). Compare the results by computing difference images, and also determine the total errors.

Exercise 6.3

Write a program to evaluate the errors that are introduced by using *non-linear* instead of linear color components for grayscale conversion. To do this, compute the difference between the Y values obtained with the linear variant (Eqn. (6.16)) and the non-linear variant (Eqn. (6.17) with $w'_R = 0.309$, $w'_G = 0.609$, $w'_B = 0.082$) for all possible 2^24 RGB colors. Let your program return the maximum gray value difference and the sum of the absolute differences for all colors.

Exercise 6.4

Explain why—in contrast to Fig. 6.1 (b)—the edges of the 3D gamut volume for sRGB and Adobe RGB in Fig. 6.6 are not straight.

7

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* (see Vol. 1 [14, Sec. 5.2]). 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. 9 briefly explains the *discrete cosine transform*, a popular variant of the discrete Fourier transform that is frequently used in image compression.

7.1 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).

7.1.1 Sine and Cosine Functions

The well-known *cosine* function

$$f(x) = \cos(x) \quad (7.1)$$

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. 7.1 (a)). We say that the function is periodic with a cycle length (period) $T = 2\pi$; i.e.,

$$\cos(x) = \cos(x + 2\pi) = \cos(x + 4\pi) = \dots = \cos(x + k2\pi) \quad (7.2)$$

for any $k \in \mathbb{Z}$. The same is true for the corresponding *sine* function, except that its value is zero at the origin ($\sin(0) = 0$).

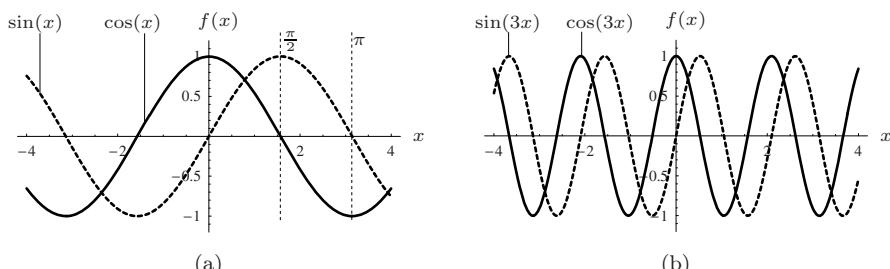


Figure 7.1 Cosine and sine functions. The expression $\cos(\omega x)$ describes a cosine function with angular frequency ω at position x . The angular frequency ω of this periodic function corresponds to a cycle length (period) $T = 2\pi/\omega$. For $\omega = 1$, the period is $T_1 = 2\pi$ (a), and 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*

$$\omega = \frac{2\pi}{T} = 1. \quad (7.3)$$

If we modify the function to

$$f(x) = \cos(3x), \quad (7.4)$$

we obtain a compressed cosine wave that oscillates three times faster than the original function $\cos(x)$ (Fig. 7.1 (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

$$T = \frac{2\pi}{\omega} \quad (7.5)$$

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

$$a \cdot \cos(\omega x) \quad \text{and} \quad a \cdot \sin(\omega x)$$

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

$$f = \frac{1}{T} = \frac{\omega}{2\pi} \quad \text{or} \quad \omega = 2\pi f, \quad (7.6)$$

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 ,

$$\cos(x) \rightarrow \cos(x - \varphi),$$

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.

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 right² by a quarter period ($\varphi = \frac{2\pi}{4} = \frac{\pi}{2}$), so

$$\sin(\omega x) = \cos\left(\omega x - \frac{\pi}{2}\right). \quad (7.7)$$

If we take the cosine function as the reference with phase $\varphi_{\text{cos}} = 0$, then the phase angle of the corresponding sine function is $\varphi_{\text{sin}} = \frac{\pi}{2} = 90^\circ$.

Cosine and sine functions are “orthogonal” in a sense and we can use this fact to create new “sinusoidal” functions with arbitrary frequency, 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:

$$A \cdot \cos(\omega x) + B \cdot \sin(\omega x) = C \cdot \cos(\omega x - \varphi). \quad (7.8)$$

The resulting amplitude C and the phase angle φ are defined only by the two original amplitudes A and B as

$$C = \sqrt{A^2 + B^2} \quad \text{and} \quad \varphi = \tan^{-1}\left(\frac{B}{A}\right). \quad (7.9)$$

Figure 7.2 (a) shows an example with amplitudes $A = B = 0.5$ and a resulting phase angle $\varphi = 45^\circ$.

Complex-valued sine functions—Euler’s notation

Figure 7.2 (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

$$z = a + i b$$

in the two-dimensional plane \mathbb{C} , where i is the imaginary unit ($i^2 = -1$). This association becomes even stronger if we look at Euler’s famous notation of complex numbers along the unit circle,

$$z = e^{i\theta} = \cos(\theta) + i \cdot \sin(\theta), \quad (7.10)$$

where $e \approx 2.71828$ is the Euler number. If we take the expression $e^{i\theta}$ 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,

$$\begin{aligned} \operatorname{Re}(e^{i\theta}) &= \cos(\theta), \\ \operatorname{Im}(e^{i\theta}) &= \sin(\theta). \end{aligned} \quad (7.11)$$

² In general, the function $f(x-d)$ is the original function $f(x)$ shifted to the right by a distance d .

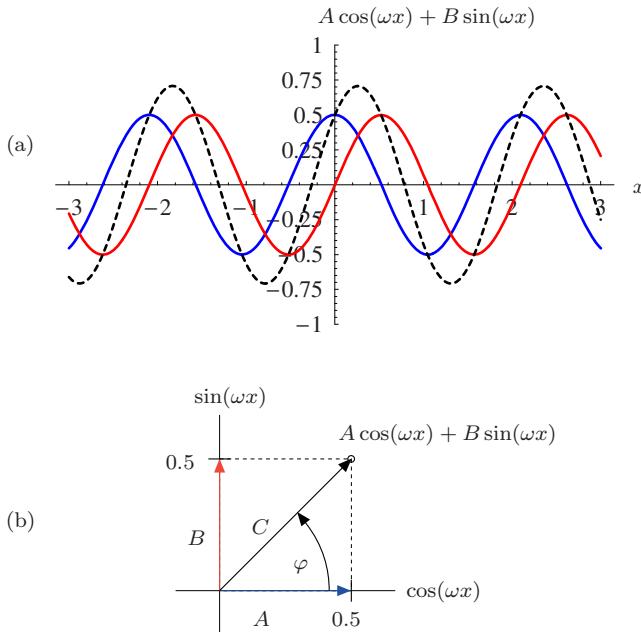


Figure 7.2 Adding cosine and sine functions with identical frequencies, $A \cdot \cos(\omega x) + B \cdot \sin(\omega x)$, with $\omega = 3$ and $A = B = 0.5$. The result is a phase-shifted cosine function (dotted curve) with amplitude $C = \sqrt{0.5^2 + 0.5^2} \approx 0.707$ and phase angle $\varphi = 45^\circ$ (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).

Since $z = e^{i\theta}$ 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 $a \geq 0$,

$$|a \cdot e^{i\theta}| = a \cdot |e^{i\theta}| = a. \quad (7.12)$$

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 $c = e^{i\varphi}$,

$$e^{i(\theta+\varphi)} = e^{i\theta} \cdot e^{i\varphi}. \quad (7.13)$$

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 A.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

$$e^{i\theta} = e^{i\omega x} = \cos(\omega x) + i \cdot \sin(\omega x). \quad (7.14)$$

We will make more use of this notation later in Sec. 7.1.4 to explain the Fourier transform.

7.1.2 Fourier Series of Periodic Functions

As we demonstrated in Eqn. (7.8), 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 Fourier³ 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.,

$$g(x) = \sum_{k=0}^{\infty} (A_k \cos(k\omega_0 x) + B_k \sin(k\omega_0 x)). \quad (7.15)$$

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. (7.15) 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*.

7.1.3 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

$$g(x) = \int_0^{\infty} A_{\omega} \cos(\omega x) + B_{\omega} \sin(\omega x) \, d\omega \quad (7.16)$$

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

³ Jean Baptiste Joseph de Fourier (1768–1830).

ω . The Fourier integral is the basis of the Fourier spectrum and the Fourier transform, as described below (for details, see e.g., [11, Sec. 15.3]).

In Eqn. (7.16), 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:

$$A_\omega = A(\omega) = \frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \cos(\omega x) \, dx, \quad (7.17)$$

$$B_\omega = B(\omega) = \frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \sin(\omega x) \, dx. \quad (7.18)$$

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. (7.16) 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. (7.17) how to compute the spectrum for a given function $g(x)$, and Eqn. (7.16) explains how to reconstruct the original function from its spectrum if it is ever needed.

7.1.4 Fourier Spectrum and Transformation

There is now only a small remaining step from the decomposition of a function $g(x)$, as shown in Eqn. (7.17), 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. (7.17)), the *Fourier spectrum* $G(\omega)$ of a function $g(x)$ is given as

$$\begin{aligned} G(\omega) &= \sqrt{\frac{\pi}{2}} \left[A(\omega) - i \cdot B(\omega) \right] \\ &= \sqrt{\frac{\pi}{2}} \left[\frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \cos(\omega x) \, dx - i \cdot \frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \sin(\omega x) \, dx \right] \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x) \cdot [\cos(\omega x) - i \cdot \sin(\omega x)] \, dx, \end{aligned} \quad (7.19)$$

with $g(x), G(\omega) \in \mathbb{C}$. Using Euler's notation of complex values (Eqn. (7.14)) yields the continuous Fourier spectrum from Eqn. (7.19) in its most popular form:

$$\boxed{\begin{aligned} G(\omega) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x) \cdot [\cos(\omega x) - i \cdot \sin(\omega x)] dx \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x) \cdot e^{-i\omega x} dx. \end{aligned}} \quad (7.20)$$

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

$$\boxed{\begin{aligned} g(x) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} G(\omega) \cdot [\cos(\omega x) + i \cdot \sin(\omega x)] d\omega \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} G(\omega) \cdot e^{i\omega x} d\omega. \end{aligned}} \quad (7.21)$$

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. (7.20)) and the inverse transformation \mathcal{F}^{-1} (Eqn. (7.21)) 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.

7.1.5 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”,

$$g(x) \circlearrowleft G(\omega).$$

Table 7.1 lists the transform pairs for some selected analytical functions, which are also shown graphically in Figs. 7.3 and 7.4.

⁴ Also called the “direct” or “forward” transformation.

⁵ Also called “backward” transformation.

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.

Table 7.1 Fourier transforms of selected analytical functions; $\delta()$ denotes the “impulse” or *Dirac* function (see Sec. 7.2.1).

Function	Transform Pair $g(x) \circ\bullet G(\omega)$	Figure
Cosine function with frequency ω_0	$g(x) = \cos(\omega_0 x)$ $G(\omega) = \sqrt{\frac{\pi}{2}} \cdot (\delta(\omega + \omega_0) + \delta(\omega - \omega_0))$	7.3 (a, c)
Sine function with frequency ω_0	$g(x) = \sin(\omega_0 x)$ $G(\omega) = i\sqrt{\frac{\pi}{2}} \cdot (\delta(\omega + \omega_0) - \delta(\omega - \omega_0))$	7.3 (b, d)
Gaussian function of width σ	$g(x) = \frac{1}{\sigma} \cdot e^{-\frac{x^2}{2\sigma^2}}$ $G(\omega) = e^{-\frac{\sigma^2 \omega^2}{2}}$	7.4 (a, b)
Rectangular pulse of width $2b$	$g(x) = \Pi_b(x) = \begin{cases} 1 & \text{for } x \leq b \\ 0 & \text{otherwise} \end{cases}$ $G(\omega) = \frac{2b \sin(b\omega)}{\sqrt{2\pi}\omega}$	7.4 (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. 7.3 (a, c)). Intuitively, this corresponds to our physical understanding of a spectrum (e.g., if we think of a pure 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. 7.3 (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. 7.4 (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. 7.4 (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.

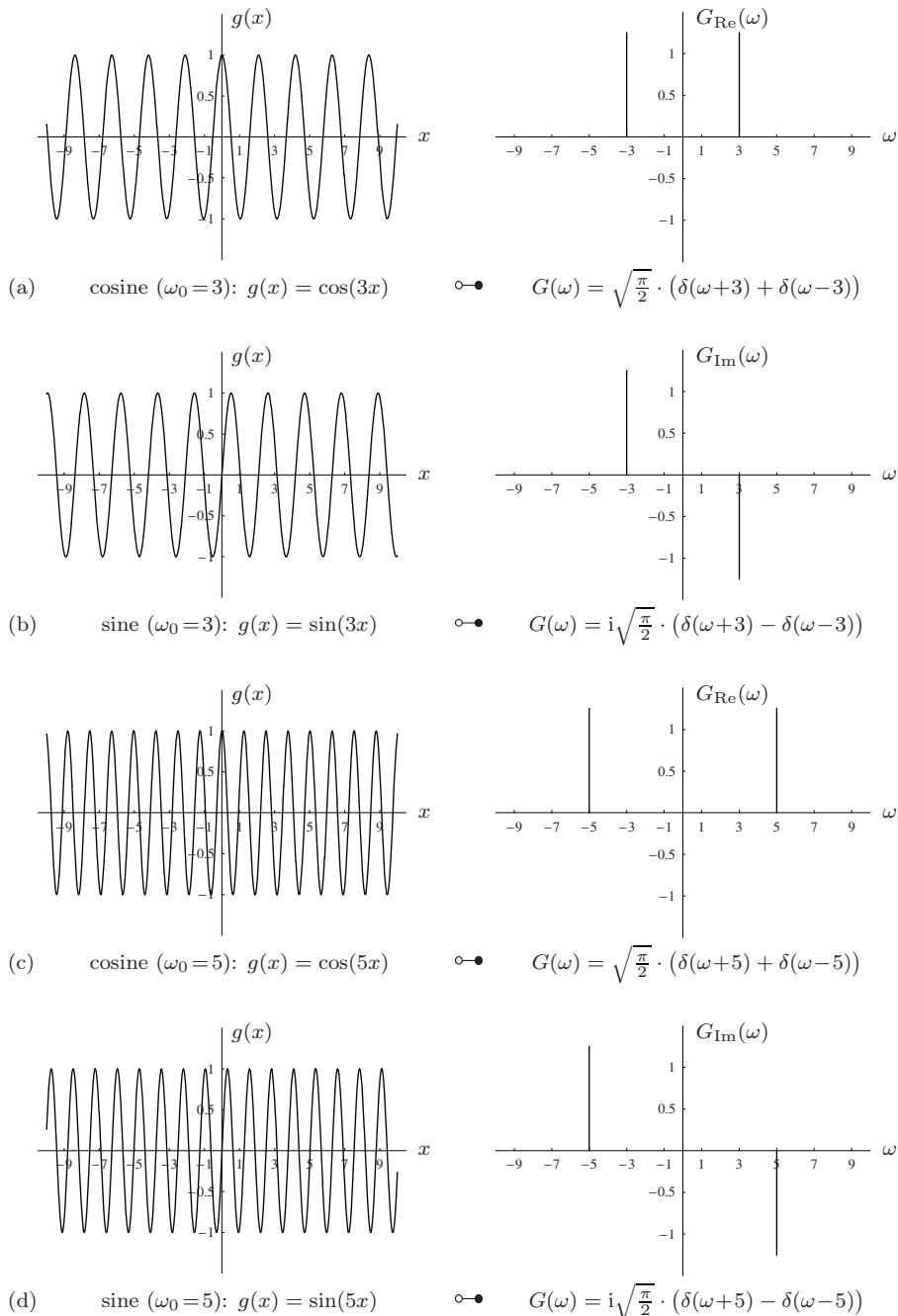


Figure 7.3 Fourier transform pairs—cosine and sine functions.

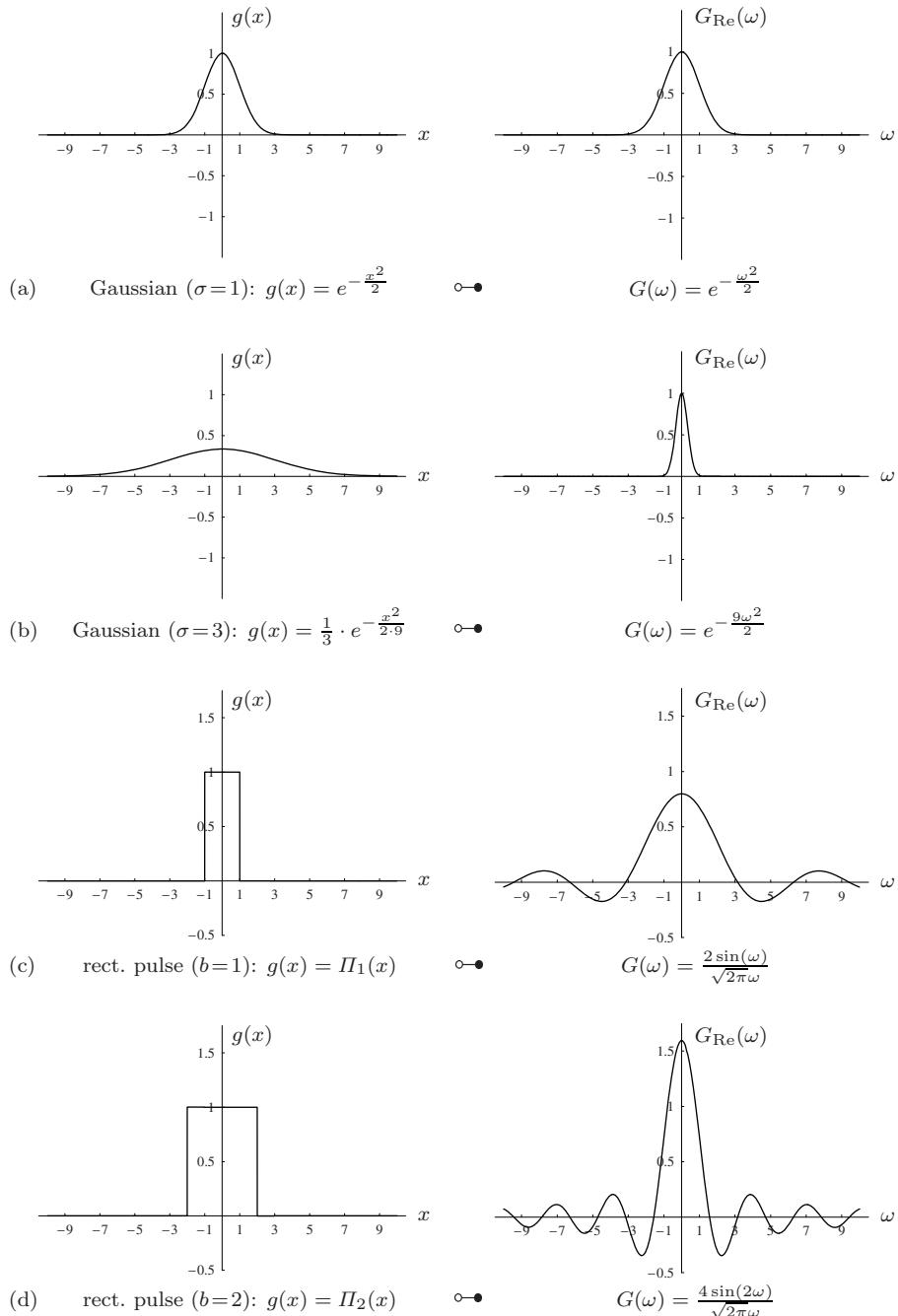


Figure 7.4 Fourier transform pairs—Gaussian functions and rectangular pulses.

7.1.6 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., [15, 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

$$G(\omega) = G^*(-\omega), \quad (7.22)$$

where G^* denotes the complex conjugate of G (see also Appendix A.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,

$$c \cdot g(x) \circledast c \cdot G(\omega). \quad (7.23)$$

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

$$g_1(x) + g_2(x) \circledast G_1(\omega) + G_2(\omega). \quad (7.24)$$

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. 7.4, *stretching* a signal by a factor s (i.e., $g(x) \rightarrow g(sx)$) leads to a *shortening* of the Fourier spectrum:

$$g(sx) \circledast \frac{1}{|s|} \cdot G\left(\frac{\omega}{s}\right). \quad (7.25)$$

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 $e^{-i\omega d}$ dependent on ω :

$$g(x-d) \circledast e^{-i\omega d} \cdot G(\omega). \quad (7.26)$$

Since $e^{-i\omega d}$ 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 Vol. 1 [14, Sec. 5.3.1]. 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 $H(\omega)$:

$$g(x) * h(x) \circlearrowright G(\omega) \cdot H(\omega). \quad (7.27)$$

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:

$$g(x) \cdot h(x) \circlearrowright G(\omega) * H(\omega). \quad (7.28)$$

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.

7.2 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.

7.2.1 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 (see

Vol. 1 [14, Sec. 5.3.4]) and in the Fourier transforms of the cosine and sine functions (Fig. 7.3). 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.,

$$\delta(x) = 0 \text{ for } x \neq 0 \quad \text{and} \quad \int_{-\infty}^{\infty} \delta(x) \, dx = 1. \quad (7.29)$$

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., $\delta(x) \rightarrow \delta(sx)$), with

$$\delta(sx) = \frac{1}{|s|} \cdot \delta(x) \quad \text{for } s \neq 0. \quad (7.30)$$

Despite the fact that $\delta(x)$ does not exist in physical reality and cannot be plotted (the corresponding plots in Fig. 7.3 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

$$\bar{g}(x) = g(x) \cdot \delta(x) = \begin{cases} g(0) & \text{for } x = 0 \\ 0 & \text{otherwise.} \end{cases} \quad (7.31)$$

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 $x = x_0$,

$$\bar{g}(x) = g(x) \cdot \delta(x - x_0) = \begin{cases} g(x_0) & \text{for } x = x_0 \\ 0 & \text{otherwise.} \end{cases} \quad (7.32)$$

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. 7.5 for the sampling position $x_0 = 3$.

⁷ The following description is intentionally casual and superficial in a mathematical sense. See, e. g., [15, 47] for more precise coverage of these topics.

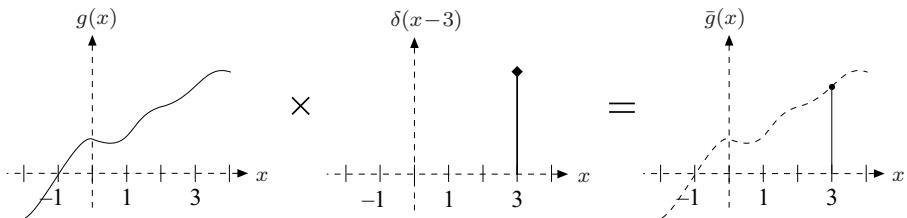


Figure 7.5 Sampling with the impulse function. The continuous signal $g(x)$ is sampled at position $x_0 = 3$ by multiplying $g(x)$ by a shifted impulse function $\delta(x-3)$.

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

$$\bar{g}(x) = g(x) \cdot \delta(x-x_1) + g(x) \cdot \delta(x-x_2) \quad (7.33)$$

$$= g(x) \cdot [\delta(x-x_1) + \delta(x-x_2)] \quad (7.34)$$

$$= \begin{cases} g(x_1) & \text{for } x = x_1 \\ g(x_2) & \text{for } x = x_2 \\ 0 & \text{otherwise.} \end{cases} \quad (7.35)$$

From Eqn. (7.34), sampling a continuous function $g(x)$ at N positions $x_i = 1, 2, \dots, N$ can thus be described as the sum of the N individual samples,

$$\begin{aligned} \bar{g}(x) &= g(x) \cdot [\delta(x-1) + \delta(x-2) + \dots + \delta(x-N)] \\ &= g(x) \cdot \sum_{i=1}^N \delta(x-i). \end{aligned} \quad (7.36)$$

The comb function

The sum of shifted impulses $\sum_{i=1}^N \delta(x-i)$ in Eqn. (7.36) is called a *pulse sequence* or *pulse train*. Extending this sequence to infinity in both directions, we obtain the “comb” or “Shah” function

$$\text{III}(x) = \sum_{i=-\infty}^{\infty} \delta(x-i). \quad (7.37)$$

The process of discretizing a continuous function by taking samples at regular integral intervals can thus be written simply as

$$\bar{g}(x) = g(x) \cdot \text{III}(x), \quad (7.38)$$

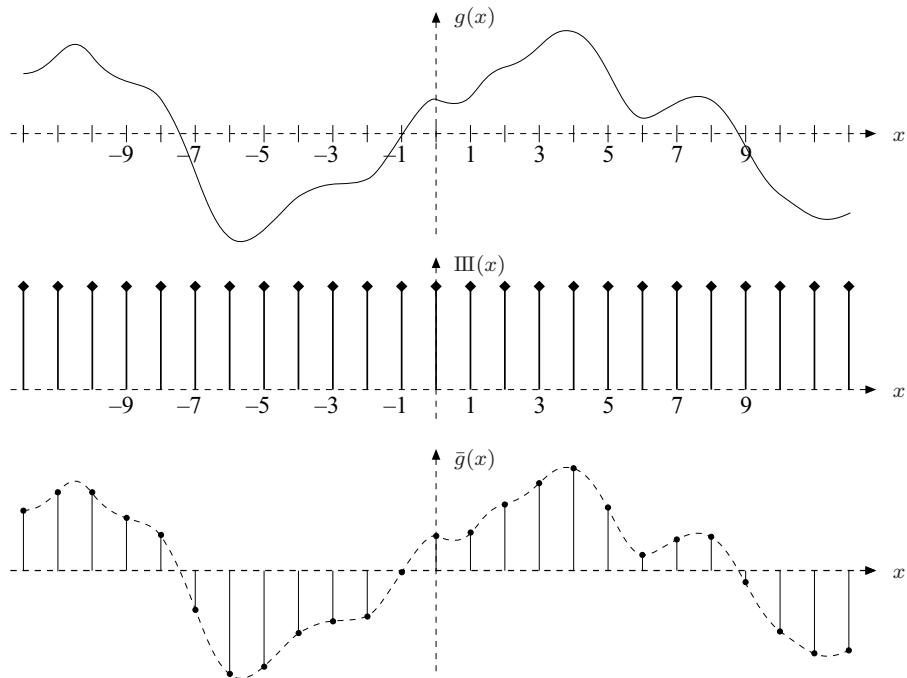


Figure 7.6 Sampling with the comb function. The original continuous signal $g(x)$ is multiplied by the comb function $\text{III}(x)$. The function value $g(x)$ is transferred to the resulting function $\bar{g}(x)$ only at integral positions $x = x_i \in \mathbb{Z}$ and ignored at all nonintegral positions.

i.e., as a pointwise multiplication of the original signal $g(x)$ with the comb function $\text{III}(x)$. As Fig. 7.6 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 $\text{III}(x)$ is simply scaled along the time or space axis; i.e.,

$$\bar{g}(x) = g(x) \cdot \text{III}\left(\frac{x}{\tau}\right) \quad \text{for } \tau > 0. \quad (7.39)$$

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. 7.1.5), the comb function features the rare property that its Fourier transform

$$\text{III}(x) \circ\bullet \text{III}\left(\frac{1}{2\pi}\omega\right) \quad (7.40)$$

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

$$\text{III}\left(\frac{x}{\tau}\right) \circ\bullet \tau \text{III}\left(\frac{\omega}{2\pi}\right) \quad (7.41)$$

due to the similarity property of the Fourier transform (Eqn. (7.25)). Figure 7.7 shows two examples of the comb function $\text{III}_\tau(x)$ 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 $\text{III}\left(\frac{x}{\tau}\right)$? We get the answer by recalling the convolution property of the Fourier transform (Eqn. (7.27)): 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

$$g(x) \cdot \text{III}\left(\frac{x}{\tau}\right) \circ\bullet G(\omega) * \tau \text{III}\left(\frac{\omega}{2\pi}\right). \quad (7.42)$$

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. 7.7). 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 Vol. 1 [14, Sec. 5.3.4]). Convolving with a *shifted* pulse $\delta(x-d)$ also reproduces the original function $f(x)$, though shifted by the same distance d ; i.e.,

$$f(x) * \delta(x-d) = f(x-d). \quad (7.43)$$

As a consequence, the spectrum $G(\omega)$ of the original continuous signal becomes *replicated* in the Fourier spectrum $\tilde{G}(\omega)$ of a sampled signal at every pulse of the sampling function's spectrum; i.e., infinitely many times (see Fig. 7.8 (a, b))! Thus the resulting Fourier spectrum is repetitive with a period $\frac{2\pi}{\tau}$, which corresponds to the sampling frequency ω_s .

Aliasing and the sampling theorem

As long as the spectral replicas in $\tilde{G}(\omega)$ 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 $\tilde{G}(\omega)$. As we can see in Fig. 7.8, this requires that

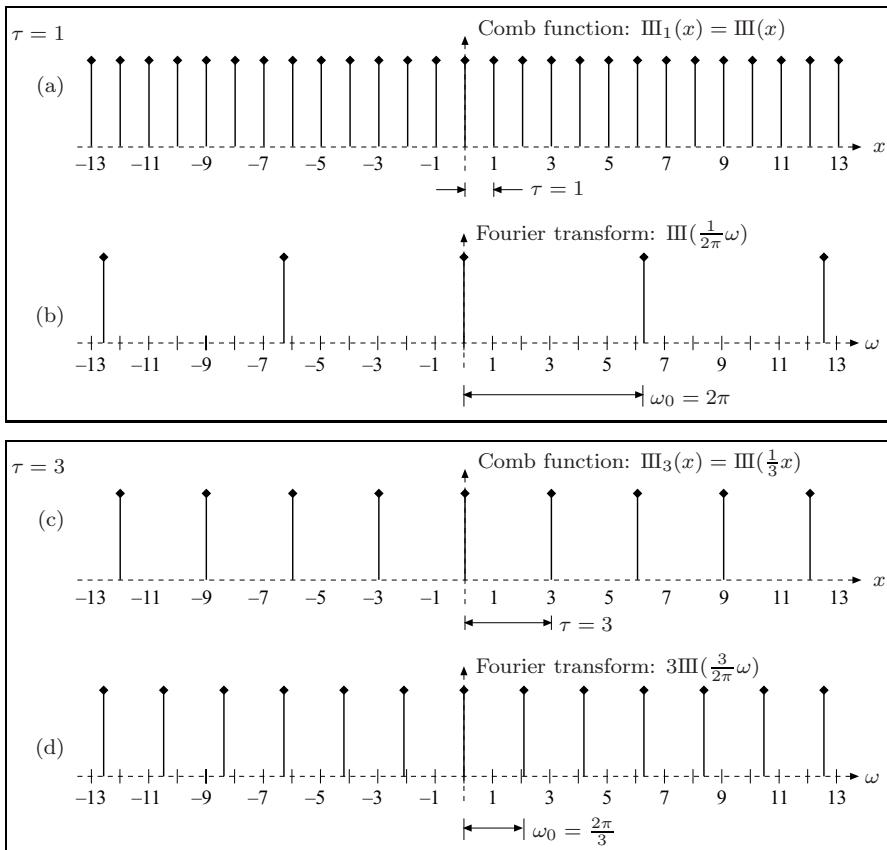


Figure 7.7 Comb function and its Fourier transform. Comb function $\text{III}_\tau(x)$ for the sampling interval $\tau = 1$ (a) and its Fourier transform. Comb function for $\tau = 3$ (c) and its Fourier transform (d). Note that the actual height of the δ -pulses is undefined and shown only for illustration.

the frequencies contained in the original signal $g(x)$ be within some upper limit ω_{\max} ; i.e., the signal contains no components with frequencies greater than ω_{\max} . The maximum allowed signal frequency ω_{\max} depends upon the sampling frequency ω_s used to discretize the signal, with the requirement

$$\omega_{\max} \leq \frac{1}{2}\omega_s \quad \text{or} \quad \omega_s \geq 2\omega_{\max}. \quad (7.44)$$

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 ω_{\max} . If this condition is not met, the replicas in the spectrum of the sampled signal overlap (Fig. 7.8 (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”.

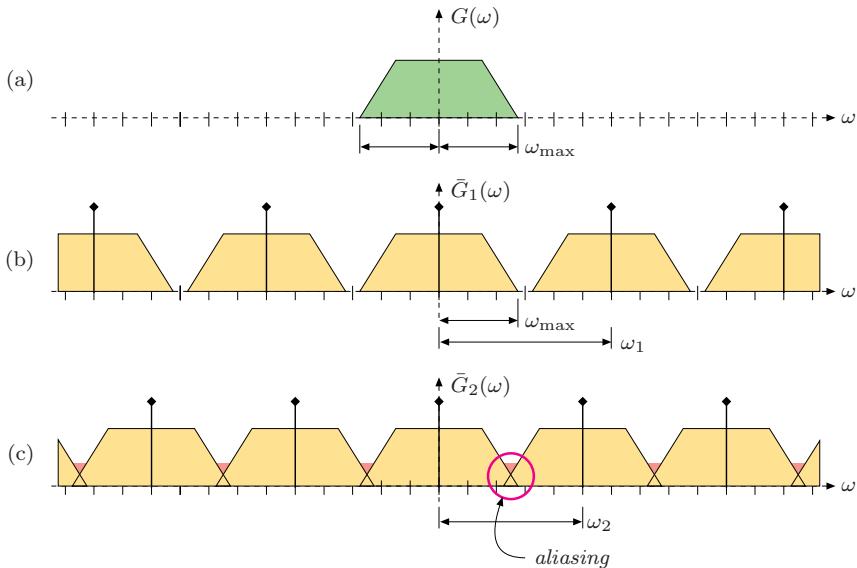


Figure 7.8 Spectral effects of sampling. The spectrum $G(\omega)$ of the original continuous signal is assumed to be band-limited within the range $\pm\omega_{\max}$ (a). Sampling the signal at a rate (sampling frequency) $\omega_s = \omega_1$ causes the signal's spectrum $G(\omega)$ to be replicated 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. [15, 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.

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.

7.2.2 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. 7.1.2, 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 7.9 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.

Thus, in general, the Fourier spectrum of a continuous, nonperiodic signal $g(x)$ is also continuous and nonperiodic (Fig. 7.9 (a, b)). However, if the signal $g(x)$ is *periodic*, then the corresponding spectrum is *discrete* (Fig. 7.9 (c,d)). Conversely, a discrete—but not necessarily periodic—signal leads to a periodic spectrum (Fig. 7.9 (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. 7.9 (g,h)). Note that the particular signals and spectra in Fig. 7.9 were chosen for illustration only and do not really correspond with each other.

7.3 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.

7.3.1 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 ($u = 0 \dots M-1$), the

⁹ Notation: we use $g(x)$, $G(\omega)$ for a *continuous* signal or spectrum, respectively, and $g(u)$, $G(m)$ for the *discrete* versions.

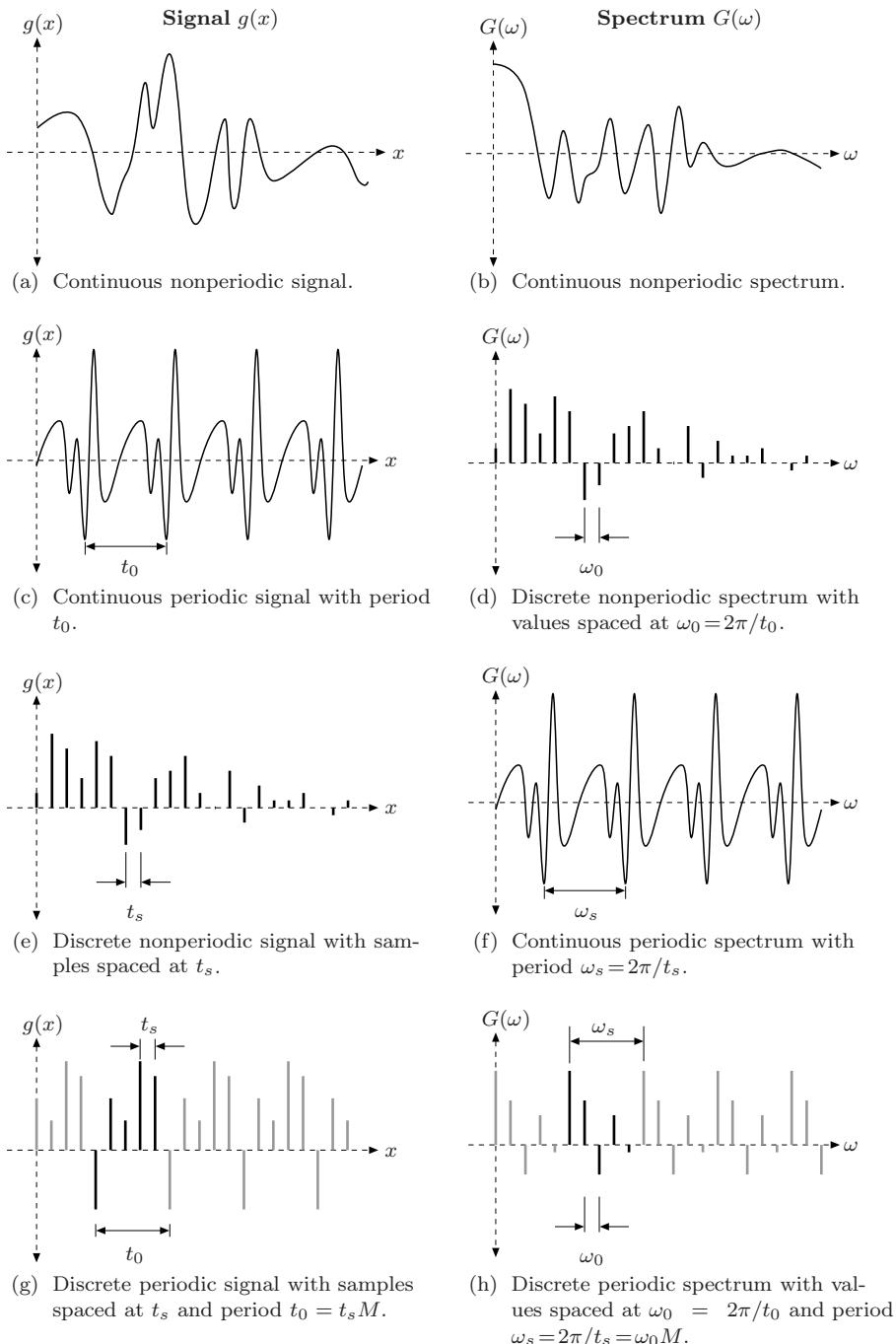


Figure 7.9 Transition from continuous to discrete periodic functions.

u	$g(u)$		$G(m)$	m
	Re	Im	Re	Im
0	1.0000	0.0000	14.2302	0.0000
1	3.0000	0.0000	-5.6745	-2.9198
2	5.0000	0.0000	*0.0000	*0.0000
3	7.0000	0.0000	-0.0176	-0.6893
4	9.0000	0.0000	*0.0000	*0.0000
5	8.0000	0.0000	0.3162	0.0000
6	6.0000	0.0000	*0.0000	*0.0000
7	4.0000	0.0000	-0.0176	0.6893
8	2.0000	0.0000	*0.0000	*0.0000
9	0.0000	0.0000	-5.6745	2.9198

Figure 7.10 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 $|G(m)| < 10^{-15}$.

forward transform (DFT) is defined as

$$\boxed{\begin{aligned} G(m) &= \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g(u) \cdot \left[\cos\left(2\pi \frac{mu}{M}\right) - i \cdot \sin\left(2\pi \frac{mu}{M}\right) \right] \\ &= \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g(u) \cdot e^{-i2\pi \frac{mu}{M}} \quad \text{for } 0 \leq m < M \end{aligned}} \quad (7.45)$$

and the *inverse* transform (DFT^{-1}) as

$$\boxed{\begin{aligned} g(u) &= \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G(m) \cdot \left[\cos\left(2\pi \frac{mu}{M}\right) + i \cdot \sin\left(2\pi \frac{mu}{M}\right) \right] \\ &= \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G(m) \cdot e^{i2\pi \frac{mu}{M}} \quad \text{for } 0 \leq u < M. \end{aligned}} \quad (7.46)$$

(Compare these definitions with the corresponding expressions for the *continuous* forward and inverse Fourier transforms in Eqns. (7.20) and (7.21), respectively.) Both the *signal* $g(u)$ and the discrete *spectrum* $G(m)$ are complex-valued vectors of length M ,

$$\begin{aligned} g(u) &= g_{\text{Re}}(u) + i \cdot g_{\text{Im}}(u), \\ G(m) &= G_{\text{Re}}(m) + i \cdot G_{\text{Im}}(m), \end{aligned} \quad (7.47)$$

for $u, m = 0 \dots M-1$ (Fig. 7.10). Expanding the first line of Eqn. (7.45), we

obtain the complex values of the Fourier spectrum in component notation as

$$G(m) = \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} \underbrace{[g_{\text{Re}}(u) + i \cdot g_{\text{Im}}(u)]}_{g(u)} \cdot \underbrace{[\cos(2\pi \frac{mu}{M}) - i \cdot \sin(2\pi \frac{mu}{M})]}_{C_m^M(u) - S_m^M(u)}, \quad (7.48)$$

where we denote as C_m^M and 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

$$G_{\text{Re}}(m) = \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g_{\text{Re}}(u) \cdot C_m^M(u) + g_{\text{Im}}(u) \cdot S_m^M(u), \quad (7.49)$$

$$G_{\text{Im}}(m) = \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g_{\text{Im}}(u) \cdot C_m^M(u) - g_{\text{Re}}(u) \cdot S_m^M(u), \quad (7.50)$$

for $m = 0 \dots M - 1$. Analogously, the *inverse* DFT in Eqn. (7.46) expands to

$$g(u) = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} \underbrace{[G_{\text{Re}}(m) + i \cdot G_{\text{Im}}(m)]}_{G(m)} \cdot \underbrace{[\cos(2\pi \frac{mu}{M}) + i \cdot \sin(2\pi \frac{mu}{M})]}_{C_m^M(u) + S_m^M(u)}, \quad (7.51)$$

and thus the real and imaginary parts of the reconstructed signal are

$$g_{\text{Re}}(u) = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G_{\text{Re}}(m) \cdot C_m^M(u) - G_{\text{Im}}(m) \cdot S_m^M(u), \quad (7.52)$$

$$g_{\text{Im}}(u) = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G_{\text{Im}}(m) \cdot C_m^M(u) + G_{\text{Re}}(m) \cdot S_m^M(u), \quad (7.53)$$

for $u = 0 \dots M - 1$.

7.3.2 Discrete Basis Functions

Equation (7.51) describes the decomposition of the discrete function $g(u)$ into a finite sum of M discrete cosine and sine functions (C_m^M , S_m^M) whose weights (or “amplitudes”) are determined by the DFT coefficients in $G(m)$. Each of these one-dimensional basis functions,

$$C_m^M(u) = C_u^M(m) = \cos(2\pi \frac{mu}{M}) = \cos(\omega_m u), \quad (7.54)$$

$$S_m^M(u) = S_u^M(m) = \sin(2\pi \frac{mu}{M}) = \sin(\omega_m u), \quad (7.55)$$

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. 7.11 and 7.12 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 $\mathbf{C}_0^M(u)$ (Eqn. (7.54)) has the constant value 1. The corresponding DFT coefficient $G_{\text{Re}}(0)$ —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. (7.52). In contrast, the zero-frequency sine function $\mathbf{S}_0^M(u)$ is zero for any value of u and thus cannot contribute anything to the signal. The corresponding DFT coefficients $G_{\text{Im}}(0)$ in Eqn. (7.52) and $G_{\text{Re}}(0)$ in Eqn. (7.53) are therefore of no relevance. For a real-valued signal (i.e., $g_{\text{Im}}(u) = 0$ for all u), the coefficient $G_{\text{Im}}(0)$ in the corresponding Fourier spectrum must also be zero.

As shown in Fig. 7.11, 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 $m = 2 \dots 7$ correspond to $2 \dots 7$ complete cycles over the signal length M (Figs. 7.11 and 7.12).

7.3.3 Aliasing Again!

A closer look at Figs. 7.11 and 7.12 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 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. 7.2.1) 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 ($m = 5 \dots 7$ 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. 7.8). 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. (7.22)), 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.,

$$\begin{aligned} |G(m)| &= |G(M-m)| = |G(M+m)| \\ &= |G(2M-m)| = |G(2M+m)| \\ &\quad \dots \\ &= |G(kM-m)| = |G(kM+m)| \end{aligned} \tag{7.56}$$

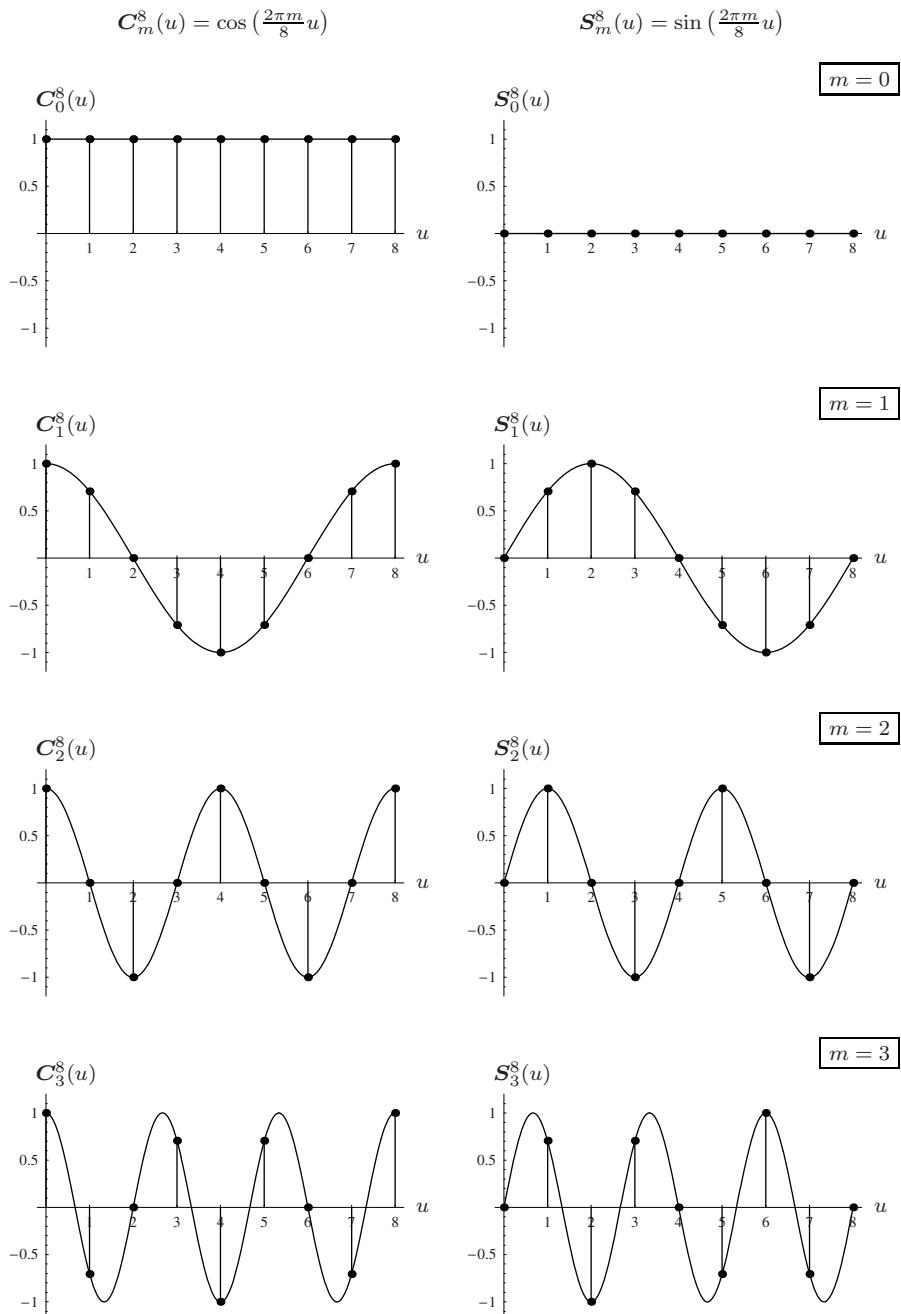


Figure 7.11 Discrete basis functions $C_m^M(u)$ and $S_m^M(u)$ for the signal length $M = 8$ and wave numbers $m = 0 \dots 3$. Each plot shows both the discrete function (round dots) and the corresponding continuous function.

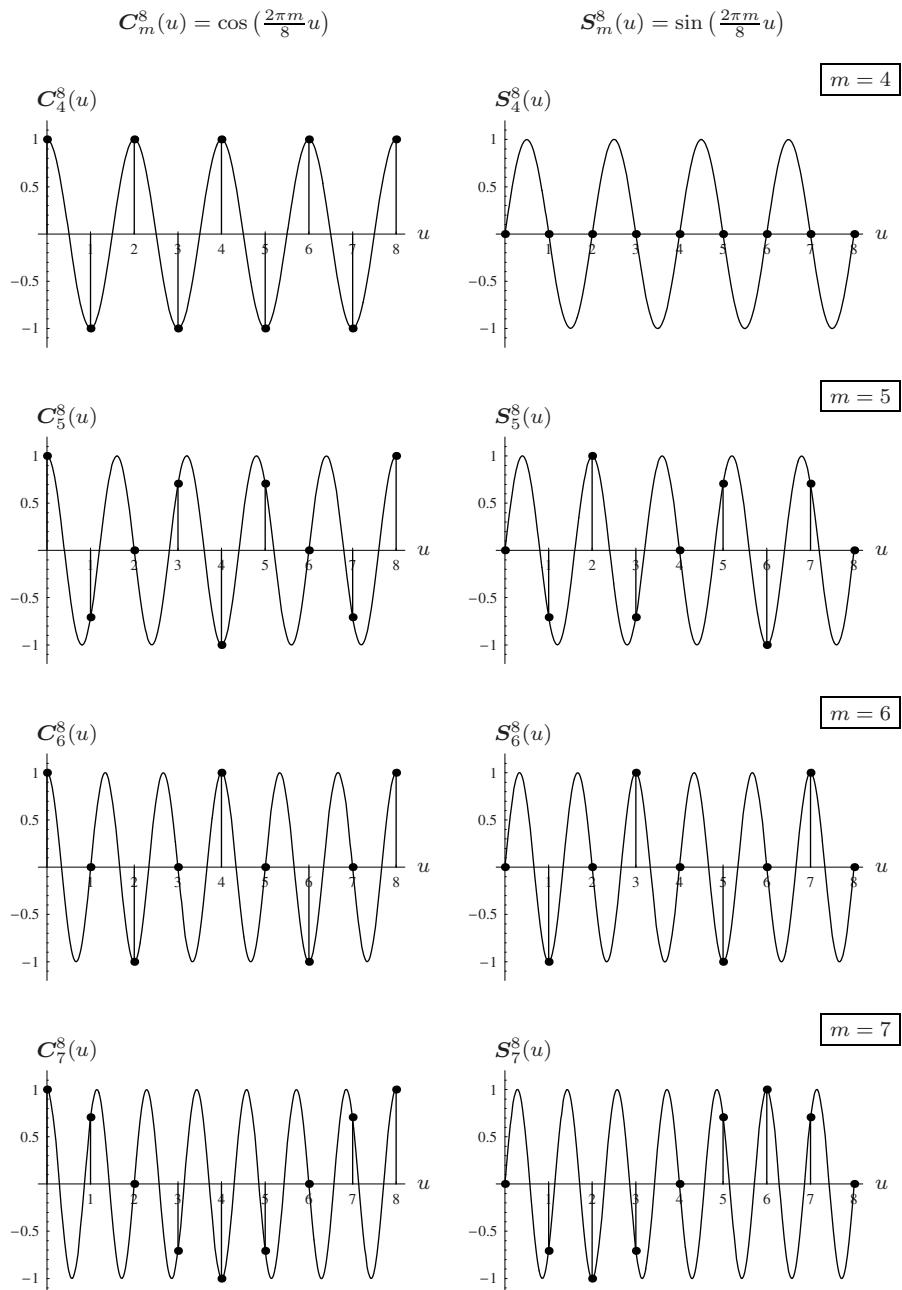


Figure 7.12 Discrete basis functions (continued). Signal length $M = 8$ and wave numbers $m = 4 \dots 7$. Notice that, for example, the discrete functions for $m = 5$ and $m = 3$ (Fig. 7.11) are identical because $m = 4$ is the maximum wave number that can be represented in a discrete spectrum of length $M = 8$.

$$C_m^8(u) = \cos\left(\frac{2\pi m}{8}u\right)$$

$$S_m^8(u) = \sin\left(\frac{2\pi m}{8}u\right)$$

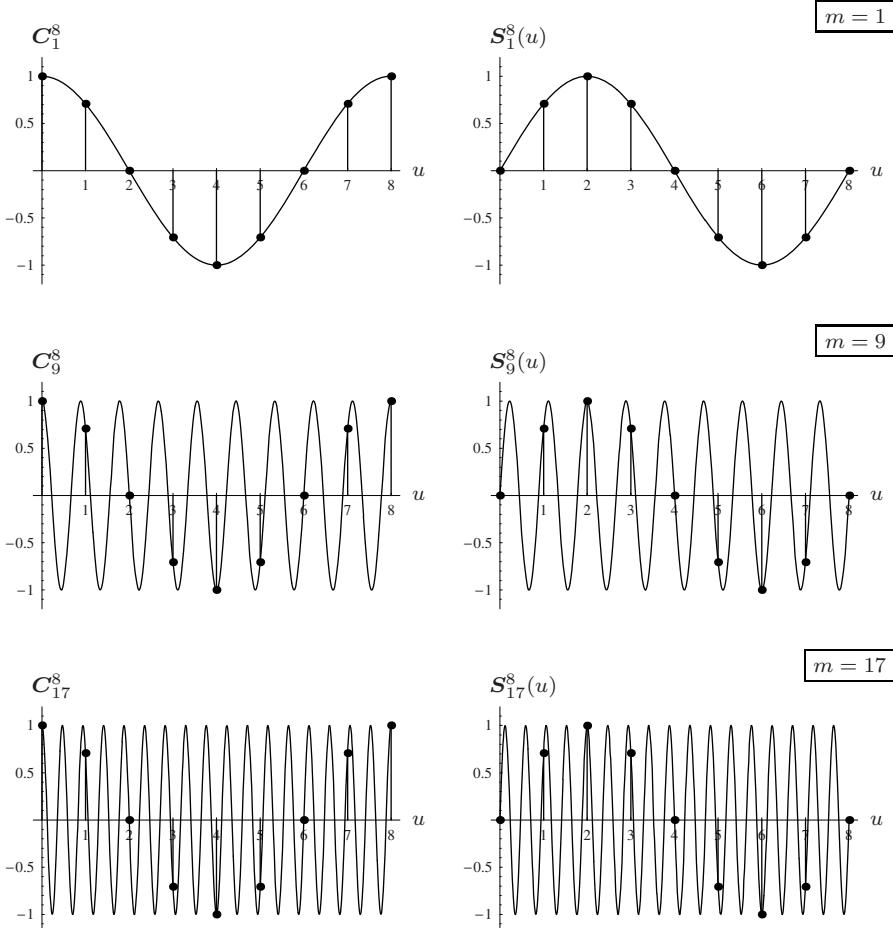


Figure 7.13 Aliasing in signal space. For the signal length $M = 8$, the discrete cosine and sine basis functions for the wave numbers $m = 1, 9, 17, \dots$ (round dots) are all identical. The sampling frequency itself corresponds to the wave number $m = 8$.

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.

7.3.4 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*

$$f_1 = \frac{1}{M\tau}. \quad (7.57)$$

In general, the wave number m of a discrete spectrum relates to the physical frequency as

$$f_m = m \frac{1}{M\tau} = m \cdot f_1 \quad (7.58)$$

for $0 \leq m < M$, which is equivalent to the angular frequency

$$\omega_m = 2\pi f_m = m \frac{2\pi}{M\tau} = m \cdot \omega_1. \quad (7.59)$$

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

$$m_{\max} = \frac{M}{2} = \frac{m_s}{2}, \quad (7.60)$$

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. (7.57) 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.

7.3.5 Power Spectrum

The *magnitude* of the complex-valued Fourier spectrum

$$|G(m)| = \sqrt{G_{\text{Re}}^2(m) + G_{\text{Im}}^2(m)} \quad (7.61)$$

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. 8.2).

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

$$g_2(u) = g_1(u-d), \quad (7.62)$$

the corresponding power spectra are the same,

$$|G_2(m)| = |G_1(m)|, \quad (7.63)$$

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

$$|G(m)| = |G(-m)| \quad (7.64)$$

for real-valued signals $g(u) \in \mathbb{R}$.

7.4 Implementing the DFT

7.4.1 Direct Implementation

Based on the definitions in Eqns. (7.49)–(7.50) the DFT can be directly implemented, as shown in Prog. 7.1. 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 $m = 0 \dots M - 1$ 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

$$\varphi_m \cdot u = 2\pi \frac{mu}{M} \equiv \underbrace{\frac{2\pi}{M} (mu \bmod M)}_{0 \leq k < M} = 2\pi \frac{k}{M} = \varphi_k, \quad (7.65)$$

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

$$\begin{aligned} \mathbf{W}_C[k] &\leftarrow \cos(\omega_k) = \cos\left(2\pi \frac{k}{M}\right), \\ \mathbf{W}_S[k] &\leftarrow \sin(\omega_k) = \sin\left(2\pi \frac{k}{M}\right), \end{aligned}$$

for $0 \leq k < M$. For computing the DFT, the necessary cosine and sine values (Eqn. (7.48)) can be read from these tables as

$$\mathbf{C}_k^M(u) = \cos\left(2\pi \frac{mu}{M}\right) = \mathbf{W}_C[mu \bmod M], \quad (7.66)$$

$$\mathbf{S}_k^M(u) = \sin\left(2\pi \frac{mu}{M}\right) = \mathbf{W}_S[mu \bmod M], \quad (7.67)$$

for arbitrary values of m and u , without any additional computation. The necessary modification of the `DFT()` method in Prog. 7.1 is left as an exercise (Exercise 7.5).

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.

¹⁰ See also Vol. 1 [14, Appendix B.1.2].

```

1 Complex[] DFT(Complex[] g, boolean forward) {
2     int M = g.length;
3     double s = 1 / Math.sqrt(M); //common scale factor
4     Complex[] G = new Complex[M];
5     for (int m = 0; m < M; m++) {
6         double sumRe = 0;
7         double sumIm = 0;
8         double phim = 2 * Math.PI * m / M;
9         for (int u = 0; u < M; u++) {
10            double gRe = g[u].re;
11            double gIm = g[u].im;
12            double cosw = Math.cos(phim * u);
13            double sinw = Math.sin(phim * u);
14            if (!forward) // inverse transform
15                sinw = -sinw;
16            //complex mult: [gRe + i gIm] · [cos(ω) + i sin(ω)]
17            sumRe += gRe * cosw + gIm * sinw;
18            sumIm += gIm * cosw - gRe * sinw;
19        }
20        G[m] = new Complex(s * sumRe, s * sumIm);
21    }
22    return G;
23 }

24 class Complex {
25     double re, im;
26
27     Complex(double re, double im) { //constructor method
28         this.re = re;
29         this.im = im;
30     }
31 }
```

Program 7.1 Direct implementation of the DFT based on the definition in Eqns. (7.49) and (7.50). 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` (bottom) defines the structure of the complex-valued vector elements.

7.4.2 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 [10].

Most FFT algorithms, including the one described in the famous publication by Cooley and Tukey in 1965 (see [28, 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 [5]. 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. (7.45)).

7.5 Exercises

Exercise 7.1

Compute the values of the cosine function $f(x) = \cos(\omega x)$ with angular frequency $\omega = 5$ for the positions $x = -3, -2, \dots, 2, 3$. What is the length of this function's period?

Exercise 7.2

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 7.3

Compute the real part, the imaginary part, and the magnitude of the complex value $z = 1.5 \cdot e^{-i2.5}$.

Exercise 7.4

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 7.5

Modify the direct implementation of the one-dimensional DFT given in Prog. 7.1 by using lookup tables for the cos and sin functions as described in Eqns. (7.66) and (7.67).

8

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.

8.1 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

$$\begin{aligned} G(m, n) &= \frac{1}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u, v) \cdot e^{-i2\pi \frac{mu}{M}} \cdot e^{-i2\pi \frac{nv}{N}} \\ &= \frac{1}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u, v) \cdot e^{-i2\pi(\frac{mu}{M} + \frac{nv}{N})} \end{aligned} \quad (8.1)$$

for the spectral coordinates $m = 0 \dots M-1$ and $n = 0 \dots N-1$. 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

$$\begin{aligned} g(u, v) &= \frac{1}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(m, n) \cdot e^{i2\pi \frac{mu}{M}} \cdot e^{i2\pi \frac{nv}{N}} \\ &= \frac{1}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(m, n) \cdot e^{i2\pi(\frac{mu}{M} + \frac{nv}{N})} \end{aligned} \quad (8.2)$$

for the image coordinates $u = 0 \dots M-1$ and $v = 0 \dots N-1$.

8.1.1 2D Basis Functions

Equation (8.2) 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

$$e^{i2\pi(\frac{mu}{M} + \frac{nv}{N})} = e^{i(\omega_m u + \omega_n v)} \quad (8.3)$$

$$= \underbrace{\cos \left[2\pi \left(\frac{mu}{M} + \frac{nv}{N} \right) \right]}_{C_{m,n}^{M,N}(u, v)} + i \cdot \underbrace{\sin \left[2\pi \left(\frac{mu}{M} + \frac{nv}{N} \right) \right]}_{S_{m,n}^{M,N}(u, v)}. \quad (8.4)$$

$C_{m,n}^{M,N}(u, v)$ and $S_{m,n}^{M,N}(u, v)$ 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 :

$$C_{m,n}^{M,N}(u, v) = \cos \left[2\pi \left(\frac{mu}{M} + \frac{nv}{N} \right) \right] = \cos(\omega_m u + \omega_n v), \quad (8.5)$$

$$S_{m,n}^{M,N}(u, v) = \sin \left[2\pi \left(\frac{mu}{M} + \frac{nv}{N} \right) \right] = \sin(\omega_m u + \omega_n v). \quad (8.6)$$

Each of these basis functions is periodic with M units in the horizontal direction and N units in the vertical direction.

Examples

Figures 8.1 and 8.2 show a set of 2D cosine functions $C_{m,n}^{M,N}$ of size $M \times N = 16 \times 16$ for various combinations of wave numbers $m, n = 0 \dots 3$. 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 $C_{2,2}^{M,N}(u, v)$ 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.

8.1.2 Implementing the Two-Dimensional DFT

As in the one-dimensional case, we could directly use the definition in Eqn. (8.1) to write a program or procedure that implements the 2D DFT. However, this is not even necessary. A minor rearrangement of Eqn. (8.1) into

$$G(m, n) = \frac{1}{\sqrt{N}} \sum_{v=0}^{N-1} \underbrace{\left[\frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g(u, v) \cdot e^{-i2\pi \frac{mu}{M}} \right]}_{\text{1-dim. DFT of row } g(\cdot, v)} \cdot e^{-i2\pi \frac{nv}{N}} \quad (8.7)$$

Algorithm 8.1 In-place computation of the two-dimensional DFT as a sequence of one-dimensional DFTs on row and column vectors.

```

1: SEPARABLE-2D-DFT ( $g$ )            $\triangleright g(u, v) \in \mathbb{C}, 0 \leq u < M, 0 \leq v < N$ 
   Returns the DFT for the two-dimensional function  $g(u, v)$ . The resulting spectrum  $G(m, n)$  has the same dimensions as  $g$ . The algorithm works “in place”; i. e.,  $g$  is modified.
2: for  $v \leftarrow 0 \dots N-1$  do
3:   Let  $g(\cdot, v)$  be the  $v$ th row vector of  $g$ :
      Replace  $g(\cdot, v)$  by  $\text{DFT}(g(\cdot, v))$ 
4: for  $u \leftarrow 0 \dots M-1$  do
5:   Let  $g(u, \cdot)$  be the  $u$ th column vector of  $g$ :
      Replace  $g(u, \cdot)$  by  $\text{DFT}(g(u, \cdot))$ 
   Remark:  $g(u, v) \equiv G(m, n) \in \mathbb{C}$  now contains the discrete 2D Fourier spectrum.
6:    $G \leftarrow g$ 
7: return  $G$ 
```

shows that its core contains a *one-dimensional* DFT (see Eqn. (7.45)) of the v th 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. (8.7)). If, in a first step, we *replace* each *row* vector $g(\cdot, v)$ of the original image by its one-dimensional Fourier transform,

$$g'(\cdot, v) \leftarrow \text{DFT}(g(\cdot, v)) \quad \text{for } 0 \leq v < N,$$

then we only need to replace each resulting *column* vector by its one-dimensional DFT in a second step:

$$g''(u, \cdot) \leftarrow \text{DFT}(g'(u, \cdot)) \quad \text{for } 0 \leq u < M.$$

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. 8.1. 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. 7.4.2) is needed to implement any multidimensional DFT.

As we can see from Eqn. (8.7), 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. 8.1 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

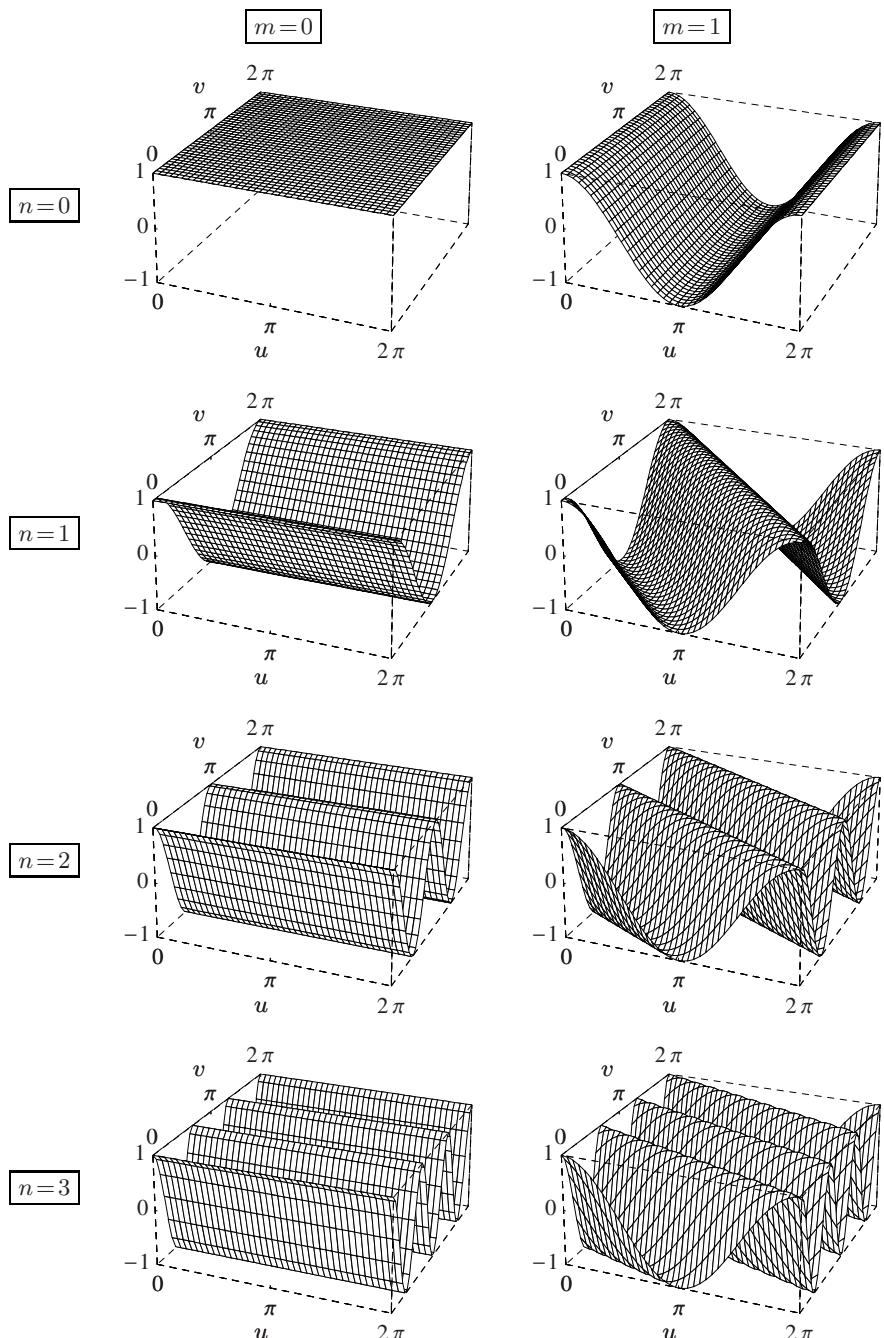


Figure 8.1 Two-dimensional cosine functions. $C_{m,n}^{M,N}(u, v) = \cos \left[2\pi \left(\frac{mu}{M} + \frac{nv}{N} \right) \right]$ for $M = N = 16$, $n = 0 \dots 3$, $m = 0, 1$.

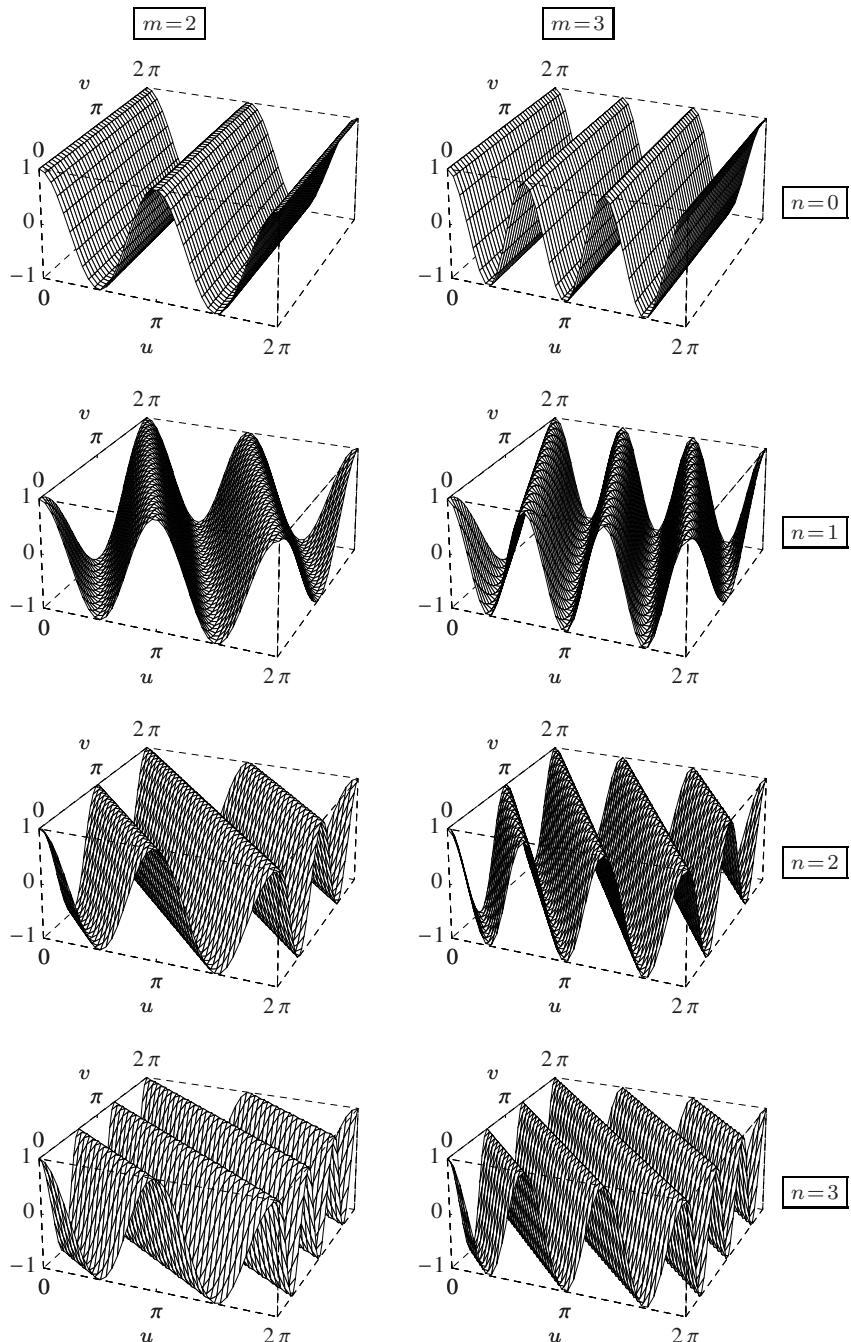


Figure 8.2 Two-dimensional cosine functions (*continued*). $M = N = 16$, $n = 0 \dots 3$, $m = 2, 3$.

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”.

8.2 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 is displayed, which in the case of the Fourier transform is $|G(m, n)|$, the *power spectrum* (see Sec. 7.3.5).

8.2.1 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. 8.4). 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.

8.2.2 Centered Representation

Analogous to the one-dimensional case, the 2D spectrum is a periodic function in both dimensions,

$$G(m, n) = G(m + pM, n + qN), \quad (8.8)$$

for arbitrary $p, q \in \mathbb{Z}$. In the case of a real-valued 2D signal $(g(u, v) \in \mathbb{R}$, see Eqn. (7.56)), the power spectrum is also *symmetric* about the origin,

$$|G(m, n)| = |G(-m, -n)|. \quad (8.9)$$

It is thus common to use a centered representation of the spectrum with coordinates m, n in the ranges

$$-\left\lfloor \frac{M}{2} \right\rfloor \leq m \leq \left\lfloor \frac{M-1}{2} \right\rfloor \quad \text{and} \quad -\left\lfloor \frac{N}{2} \right\rfloor \leq n \leq \left\lfloor \frac{N-1}{2} \right\rfloor,$$

respectively. This can be easily accomplished by swapping the four quadrants of the Fourier transform, as illustrated in Fig. 8.3. In the resulting representation, the low-frequency coefficients are found at the center and the high-frequency entries along the outer boundaries. Figure 8.4 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 ($\log_{10} |G(m, n)|$).

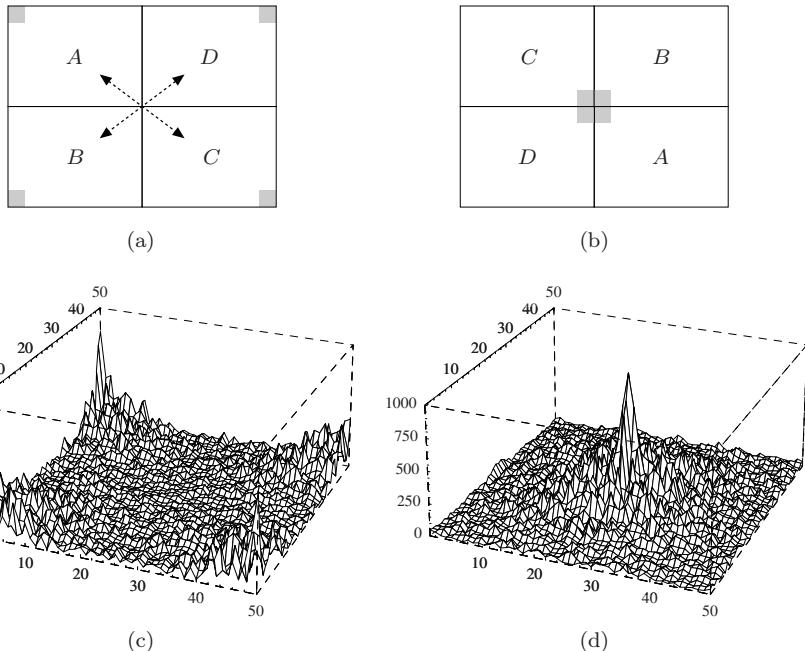


Figure 8.3 Centering the 2D Fourier spectrum. In the original (noncentered) spectrum, the coordinate center (i.e., the region of low frequencies) is located in the upper left corner and, due to the periodicity of the spectrum, also at all other corners (a). In this case, the coefficients for the highest 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).

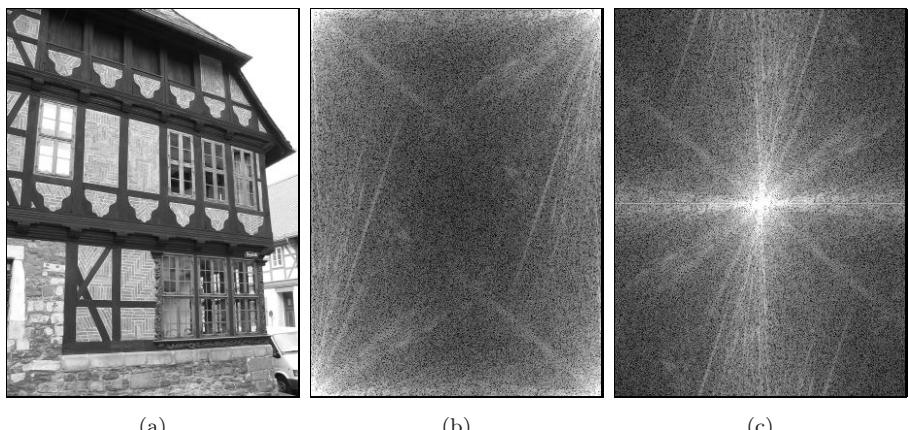


Figure 8.4 Intensity plot of a 2D power spectrum: original image (a), noncentered spectrum (b), and centered spectrum (c).

8.3 Frequencies and Orientation in 2D

As we could see in Figs. 8.1 and 8.2, 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. 8.5).

8.3.1 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. (7.57)) as

$$\hat{f}_{(m,n)} = \frac{1}{\tau} \sqrt{\left(\frac{m}{M}\right)^2 + \left(\frac{n}{N}\right)^2}, \quad (8.10)$$

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

$$\hat{f}_{(\pm \frac{M}{2}, 0)} = \hat{f}_{(0, \pm \frac{N}{2})} = \frac{1}{\tau} \sqrt{\left(\frac{1}{2}\right)^2} = \frac{1}{2\tau} = \frac{1}{2} f_s, \quad (8.11)$$

where $f_s = \frac{1}{\tau}$ denotes the sampling frequency. Notice that the effective signal frequency at the corners of the spectrum is

$$\hat{f}_{(\pm \frac{M}{2}, \pm \frac{N}{2})} = \frac{1}{\tau} \sqrt{\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2} = \frac{1}{\sqrt{2}\tau} = \frac{1}{\sqrt{2}} f_s, \quad (8.12)$$

which is a factor $\sqrt{2}$ higher than along the coordinate axes (Eqn. (8.11)).

8.3.2 Frequency Limits and Aliasing in 2D

Figure 8.6 illustrates the relationship described in Eqns. (8.11) and (8.12). 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.

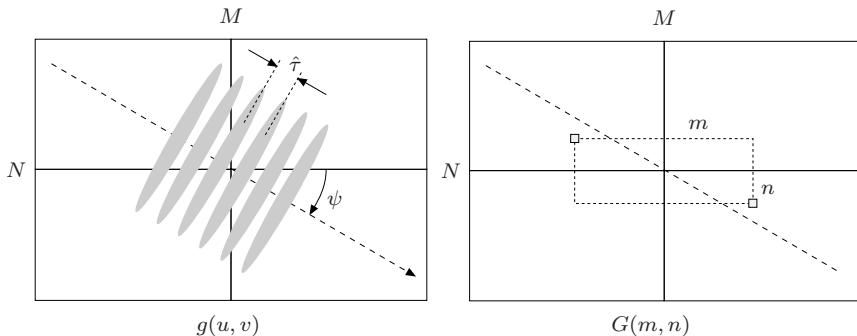


Figure 8.5 Frequency and orientation in 2D. The image (left) contains some periodic pattern with effective frequency $\hat{f} = 1/\hat{r}$ and orientation ψ . The frequency coefficient corresponding to this pattern is located at position $(m, n) = \pm\hat{f} \cdot (M \cos \psi, N \sin \psi)$ in the 2D 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.

Apparently the lowest effective sampling frequency (Eqn. (8.10)) 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.

8.3.3 Orientation

The spatial orientation of a two-dimensional cosine or sine wave with spectral coordinates m, n (wave numbers $0 \leq m < M, 0 \leq n < N$) is

$$\psi_{(m,n)} = \text{Arctan}\left(\frac{n}{N}, \frac{m}{M}\right) = \text{Arctan}(nM, mN), \quad (8.13)$$

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

$$(m, n) = \pm\hat{f} \cdot (M \cos \psi, N \sin \psi), \quad (8.14)$$

as shown before in Fig. 8.5.

¹ Arctan(y, x) in Eqn. (8.13) denotes the inverse tangent function $\tan^{-1}(y/x)$ (also see Vol. 1 [14, Appendix B.1.6]).

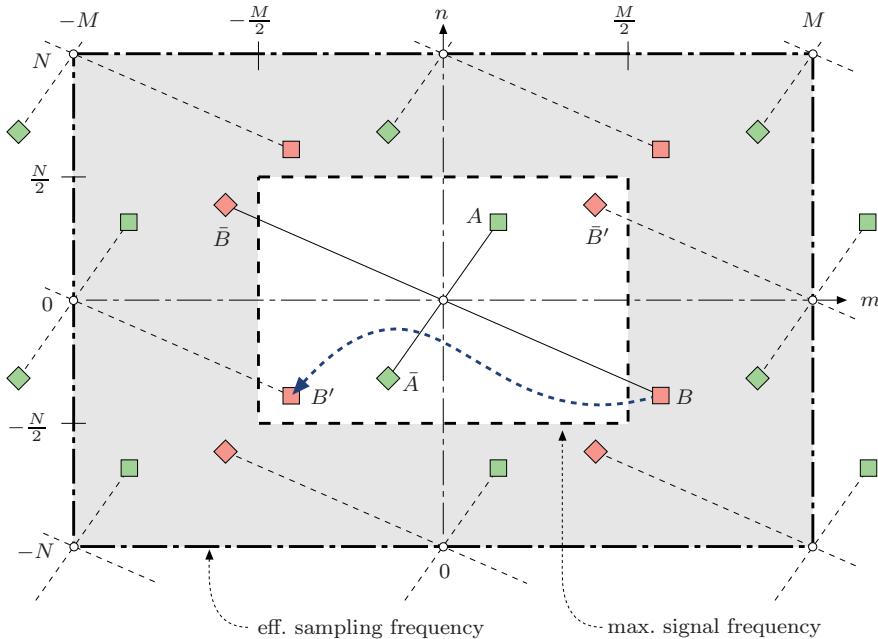


Figure 8.6 Maximum signal frequencies and aliasing in 2D. The boundary of the $M \times N$ 2D spectrum (inner rectangle) marks the region of permissible signal frequencies along any direction. The outer rectangle corresponds to the effective sampling frequency, which is twice the maximum signal frequency 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.

8.3.4 Normalizing the 2D Spectrum

From Eqn. (8.14) we can derive that in the special case of a sinusoid with spatial orientation $\psi = 45^\circ$ the corresponding spectral coefficients are found at the frequency coordinates

$$(m, n) = \pm(\lambda M, \lambda N) \quad \text{for } -\frac{1}{2} \leq \lambda \leq +\frac{1}{2}; \quad (8.15)$$

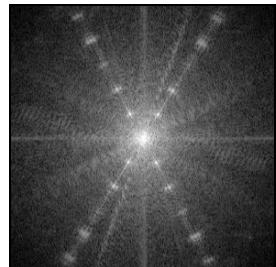
i.e., at the diagonals of the spectrum (see also Eqn. (8.12)). Unless the image (and thus the spectrum) is quadratic ($M = N$), the angle of orientation in the image and in the spectrum are not the same, though they coincide along the directions of the coordinate axes. This means that rotating an image by some



(a)



(b)



(c)

Figure 8.7 Normalizing 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 normalized to square proportions (c), we can clearly observe that the cylinders of this (Harley-Davidson V-Rod) engine are really arranged at a 60° angle.

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. 8.7).

8.3.5 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

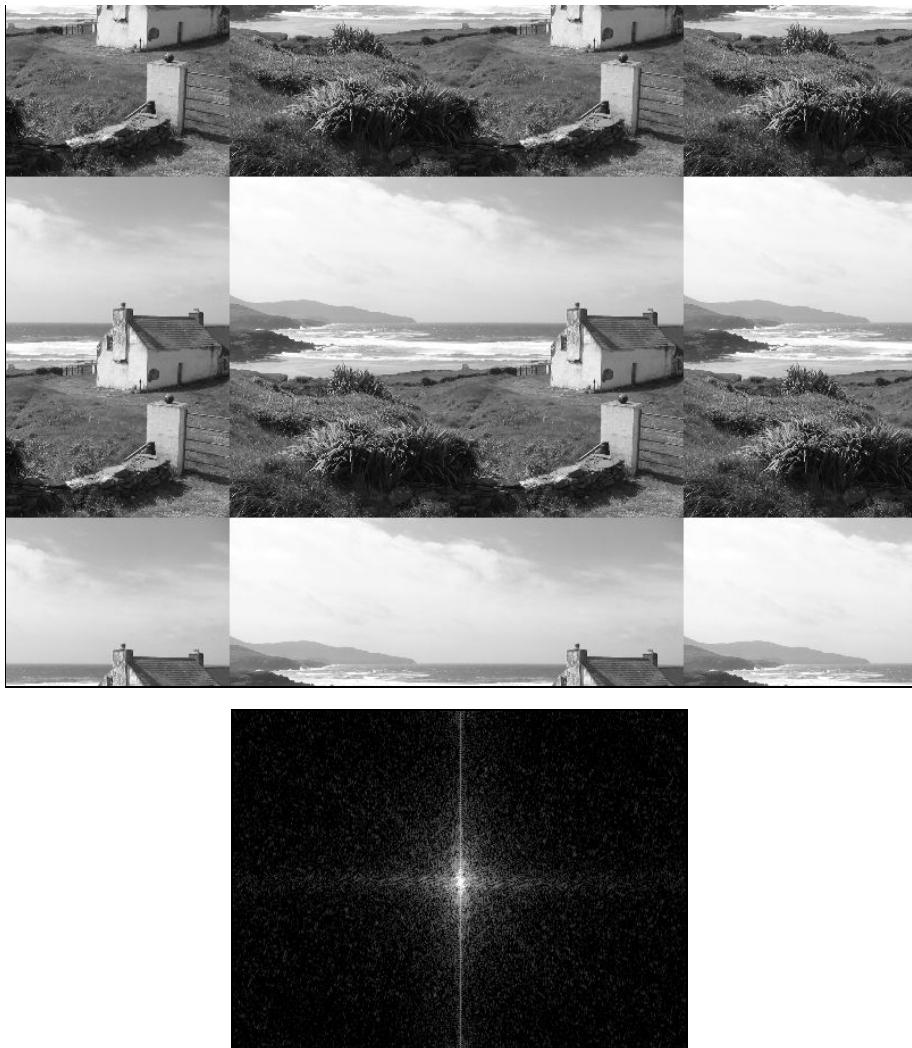


Figure 8.8 Effects of periodicity in the 2D spectrum. The discrete Fourier transform is computed under the implicit assumption that the image signal is periodic along both dimensions (top). Large differences in intensity at opposite image borders—here most notably in the vertical direction—lead to broad-band signal components that in this case appear as a bright line along the spectrum’s vertical axis (bottom).

grid; see Fig. 8.8). 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.

8.3.6 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 upon the spectrum. As we already know (from Eqn. (7.27)), 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. 8.9 (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.

8.3.7 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., [10, Sec. 9.3], [59, Ch. 10]). Table 8.1 lists the definitions of several popular windowing functions; the corresponding 2D (logarithmic) power spectra are displayed in Figs. 8.9 and 8.10.

The spectrum of the *rectangular pulse* function (Fig. 8.9 (a)), which assigns identical weights to all image elements, exhibits a relatively narrow peak at the

Table 8.1 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.

Definitions:	
$r_u = \frac{u-M/2}{M/2} = \frac{2u}{M} - 1$	$r_v = \frac{v-N/2}{N/2} = \frac{2v}{N} - 1$
$r_{u,v} = \sqrt{r_u^2 + r_v^2}$	
Elliptical window:	$w(u, v) = \begin{cases} 1 & \text{for } 0 \leq r_{u,v} \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Gaussian window:	$w(u, v) = e\left(\frac{-r_{u,v}^2}{2\sigma^2}\right), \quad \sigma = 0.3 \dots 0.4$
Super-Gaussian window:	$w(u, v) = e\left(\frac{-r_{u,v}^n}{\kappa}\right), \quad n = 6, \kappa = 0.3 \dots 0.4$
Cosine² window:	$w(u, v) = \begin{cases} \cos\left(\frac{\pi}{2}r_u\right) \cdot \cos\left(\frac{\pi}{2}r_v\right) & \text{for } 0 \leq r_u, r_v \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Bartlett window:	$w(u, v) = \begin{cases} 1 - r_{u,v} & \text{for } 0 \leq r_{u,v} \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Hanning window:	$w(u, v) = \begin{cases} 0.5 \cdot \cos(\pi r_{u,v} + 1) & \text{for } 0 \leq r_{u,v} \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Parzen window:	$w(u, v) = \begin{cases} 1 - 6r_{u,v}^2 + 6r_{u,v}^3 & \text{for } 0 \leq r_{u,v} < 0.5 \\ 2 \cdot (1 - r_{u,v})^3 & \text{for } 0.5 \leq r_{u,v} < 1 \\ 0 & \text{otherwise} \end{cases}$

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. 8.9 (b) is quite similar. The *Gaussian* window in Fig. 8.9 (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. 8.9 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. 8.10 (c)) or the *Parzen* window (Fig. 8.10 (d)), which are both easy to compute and frequently used in practice.

Figure 8.11 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.

8.4 2D Fourier Transform Examples

The following examples demonstrate some basic properties of the two-dimensional DFT on real intensity images. All examples in Figs. 8.12–8.18 show a centered and squared spectrum with logarithmic intensity values (see Sec. 8.2).

Scaling. Figure 8.12 shows that scaling the image in signal space has the opposite effect in frequency space, analogous to the one-dimensional case (Fig. 7.4).

Periodic image patterns. The images in Fig. 8.13 contain repetitive periodic patterns at various orientations and scales. They appear as distinct peaks at the corresponding positions (see Eqn. (8.14)) in the spectrum.

Rotation. Figure 8.14 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. 8.15 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. 8.16 and 8.17.

Print pattern. The regular patterns generated by the common raster print techniques (Fig. 8.18) are typical examples for periodic multidirectional structures, which stand out clearly in the corresponding Fourier spectrum.

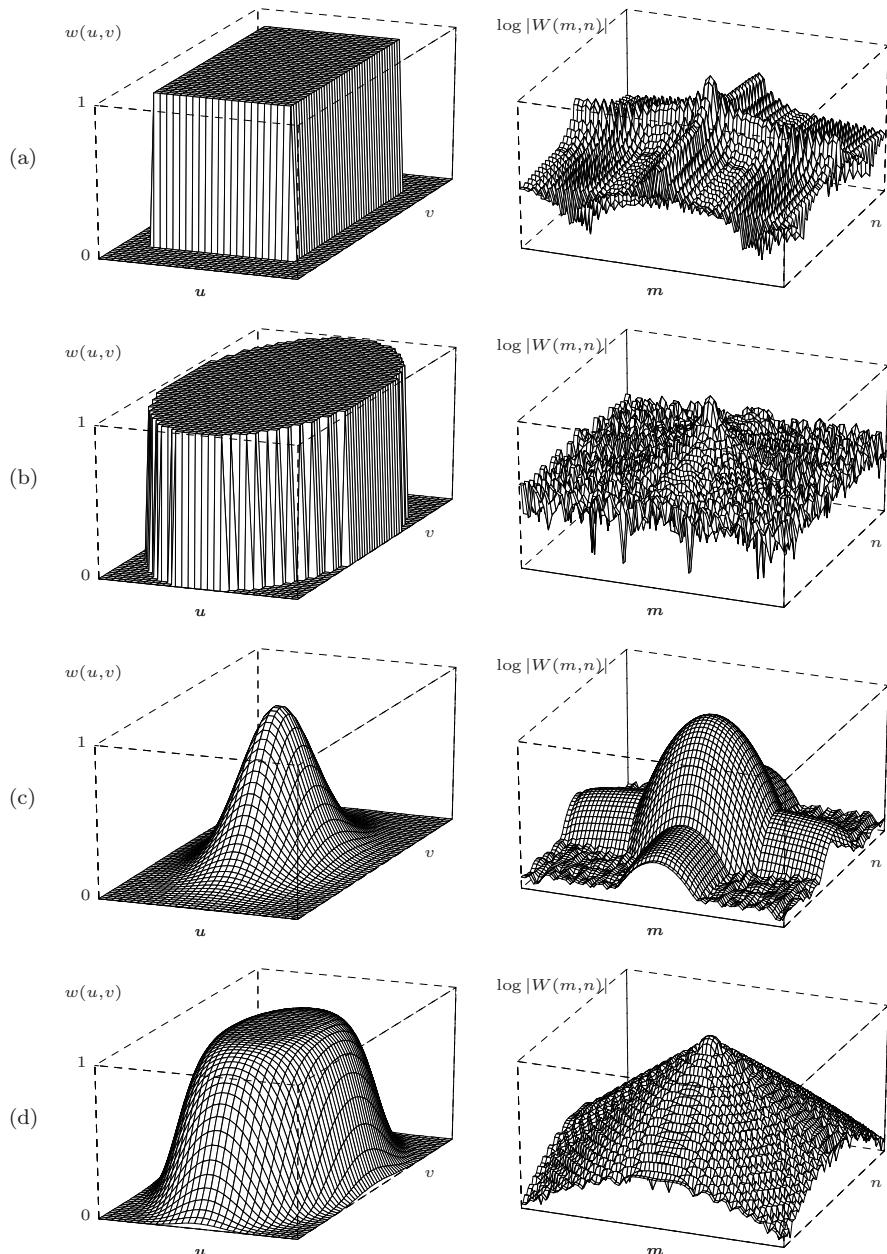


Figure 8.9 Windowing functions and their logarithmic power spectra. Rectangular pulse (a), elliptical window (b), Gaussian window with $\sigma = 0.3$ (c), and super-Gaussian window of order $n = 6$ and $\kappa = 0.3$ (d). The windowing functions are deliberately of nonsquare size ($M : N = 1 : 2$).

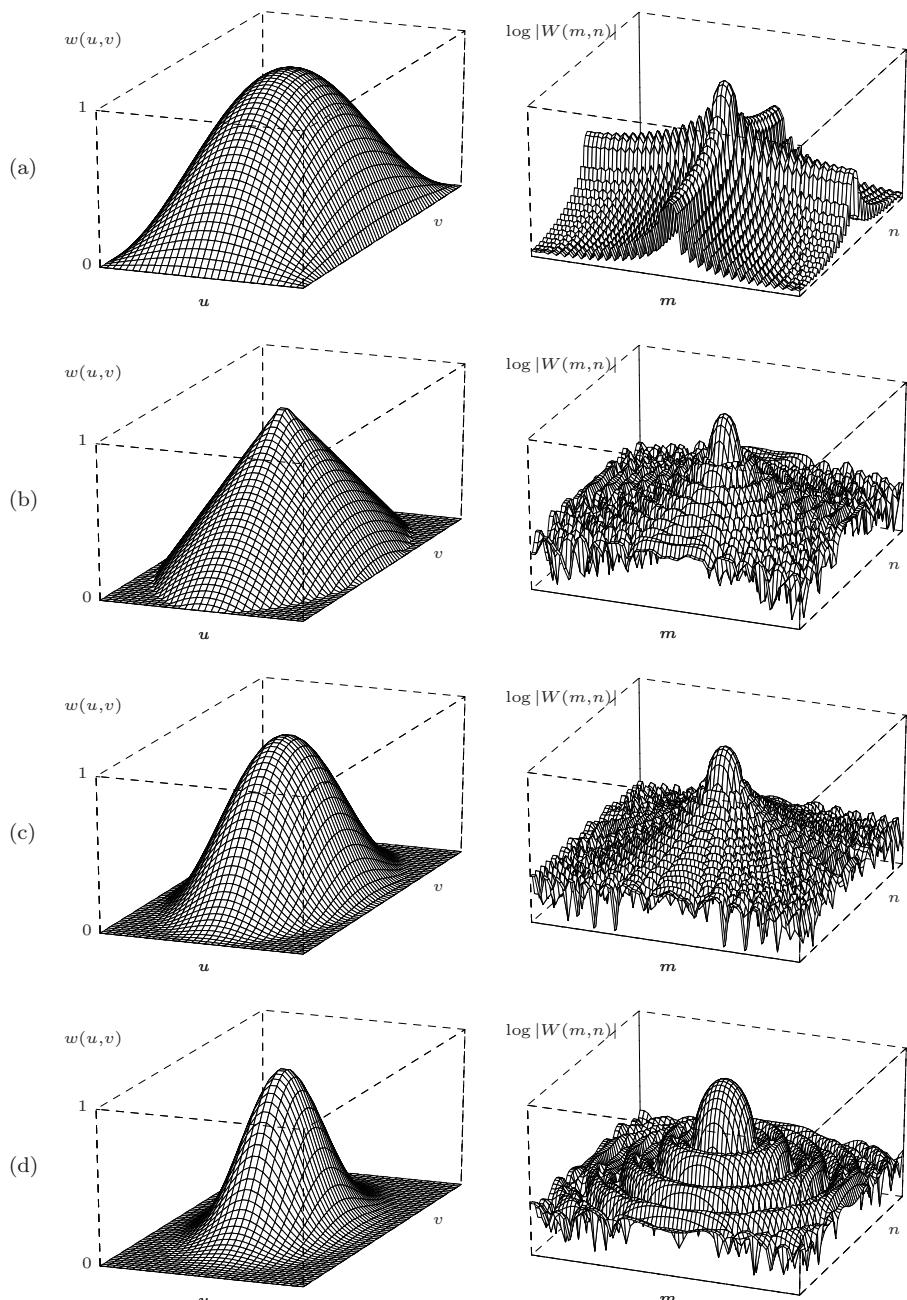


Figure 8.10 Windowing functions and their logarithmic power spectra (*continued*). Cosine² window (a), Bartlett window (b), Hanning window (c), and Parzen window (d).

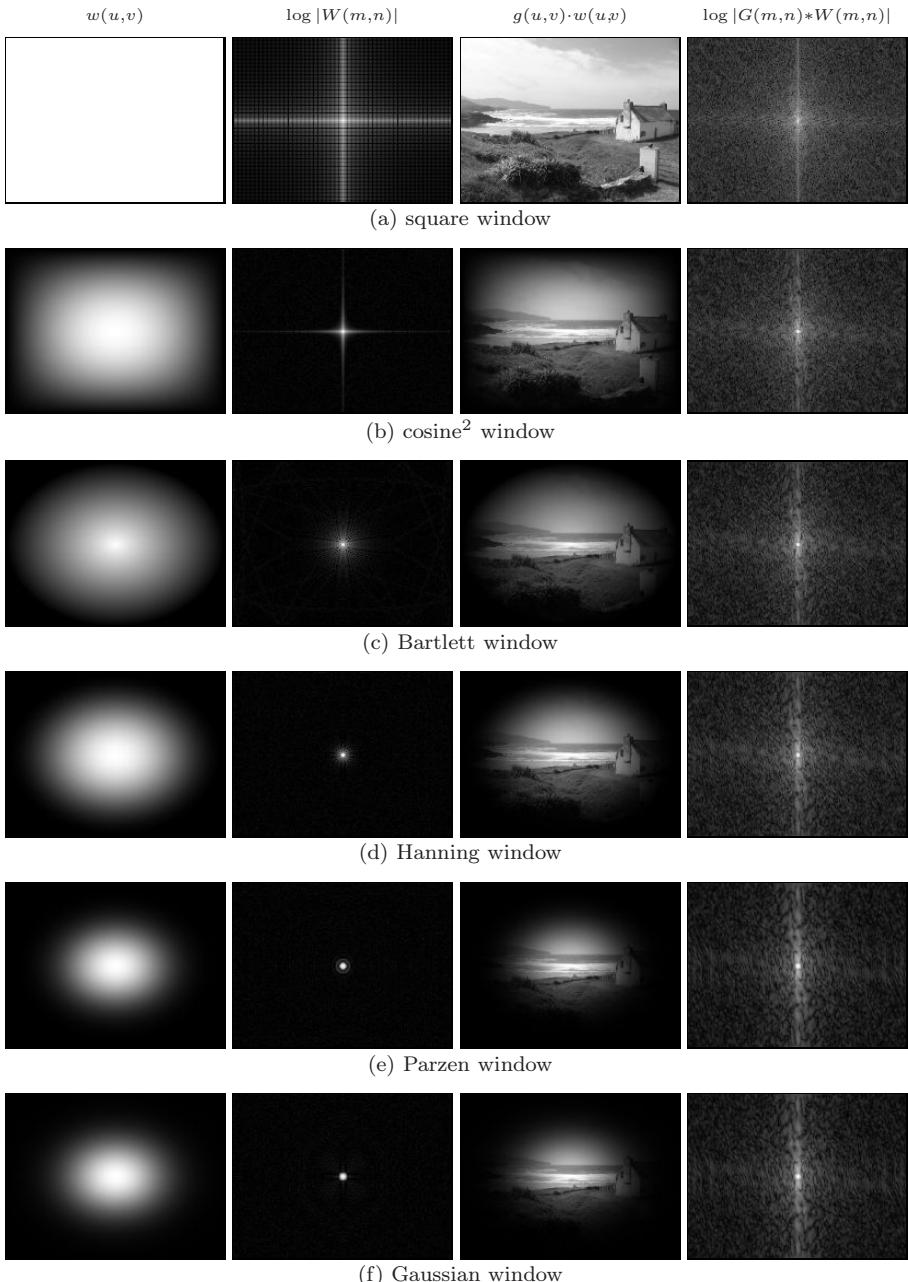


Figure 8.11 Application of windowing functions on images. The plots show the windowing function $w(u,v)$, the logarithmic power spectrum of the windowing function $\log |W(m,n)|$, the windowed image $g(u,v) \cdot w(u,v)$, and the power spectrum of the windowed image $\log |G(m,n)*W(m,n)|$.

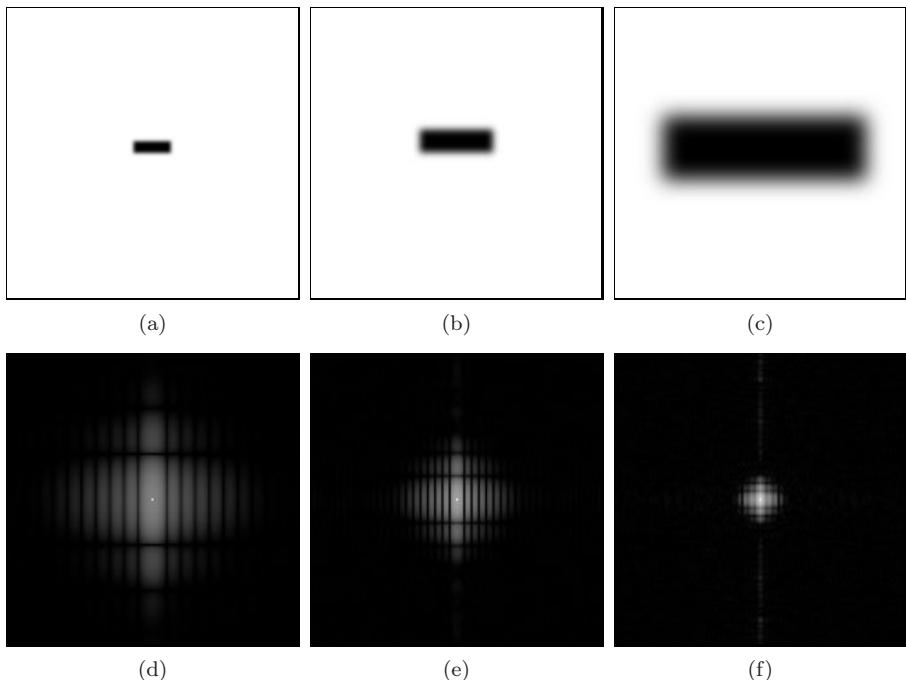


Figure 8.12 DFT under image scaling. The rectangular pulse in the image function (a–c) creates a strongly oscillating power spectrum (d–f), as in the one-dimensional case. Stretching the image causes the spectrum to contract and vice versa.

8.5 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.

8.5.1 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. 7.1.6), 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

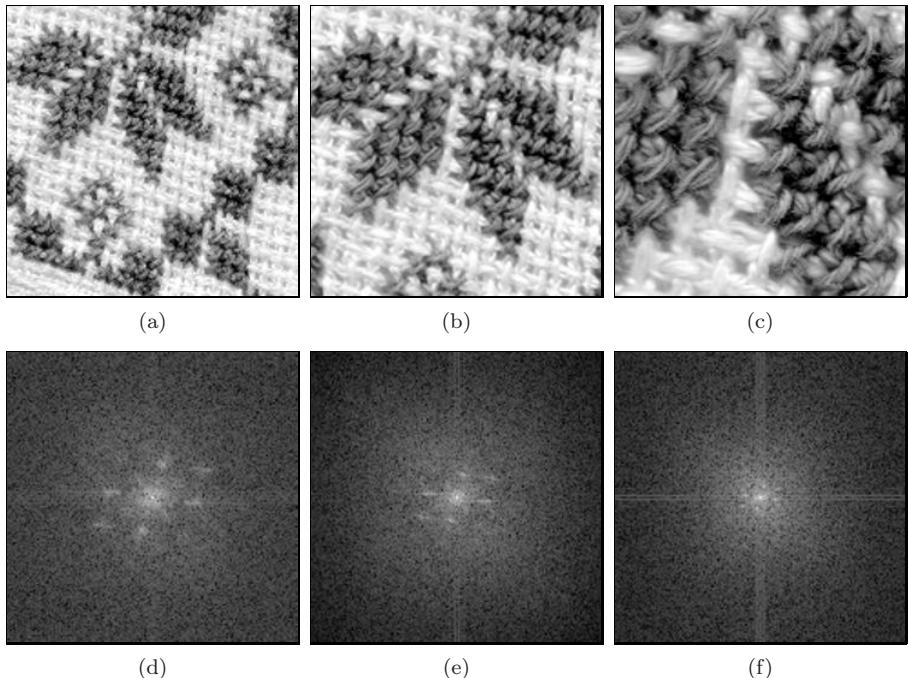


Figure 8.13 DFT of oriented, repetitive patterns. The image function (a–c) contains patterns with three dominant orientations, which appear as pairs of corresponding frequency spots in the spectrum (c–f). Enlarging the image causes the spectrum to contract.

by the following steps:

$$\begin{array}{ccc}
 \text{Image space: } g(u, v) * h(u, v) & = & g'(u, v) \\
 \downarrow & & \uparrow \\
 \text{DFT} & & \text{DFT}^{-1} \\
 \downarrow & & \uparrow \\
 \text{Frequency space: } G(m, n) \cdot H(m, n) & \longrightarrow & G'(m, n)
 \end{array} \quad (8.16)$$

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

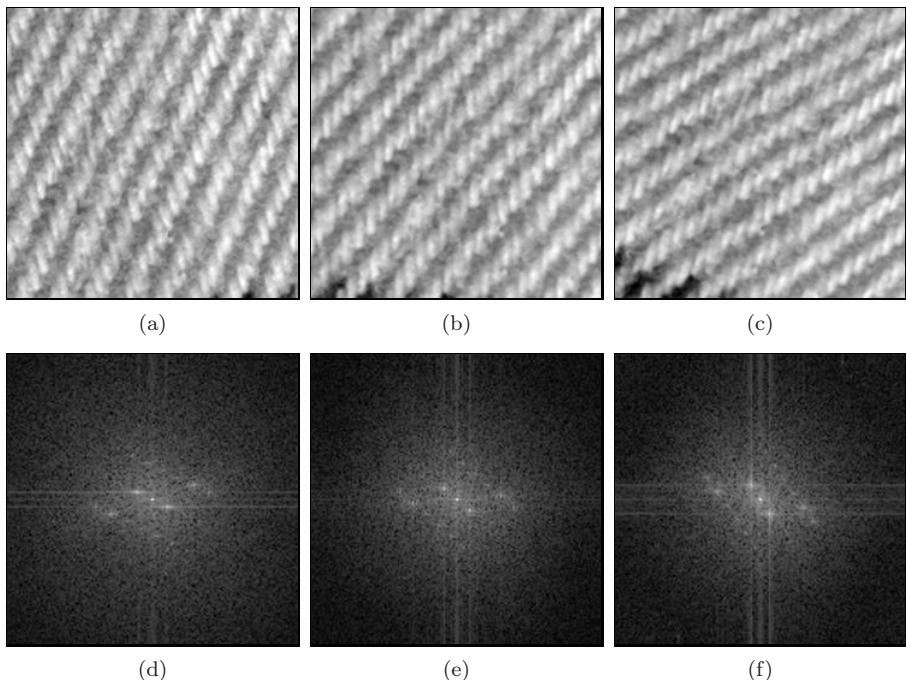


Figure 8.14 DFT—image rotation. The original image (a) is rotated by 15° (b) and 30° (c). The corresponding (squared) spectrum turns in the same direction and by exactly the same amount (d–f).

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 [28, Sec. 4.4].

8.5.2 Linear Convolution versus Correlation

As described already in Vol. 1 [14, Sec. 5.3], 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. (8.16). This could be advantageous for comparing images using correlation methods (see Sec. 11.1.1) 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-

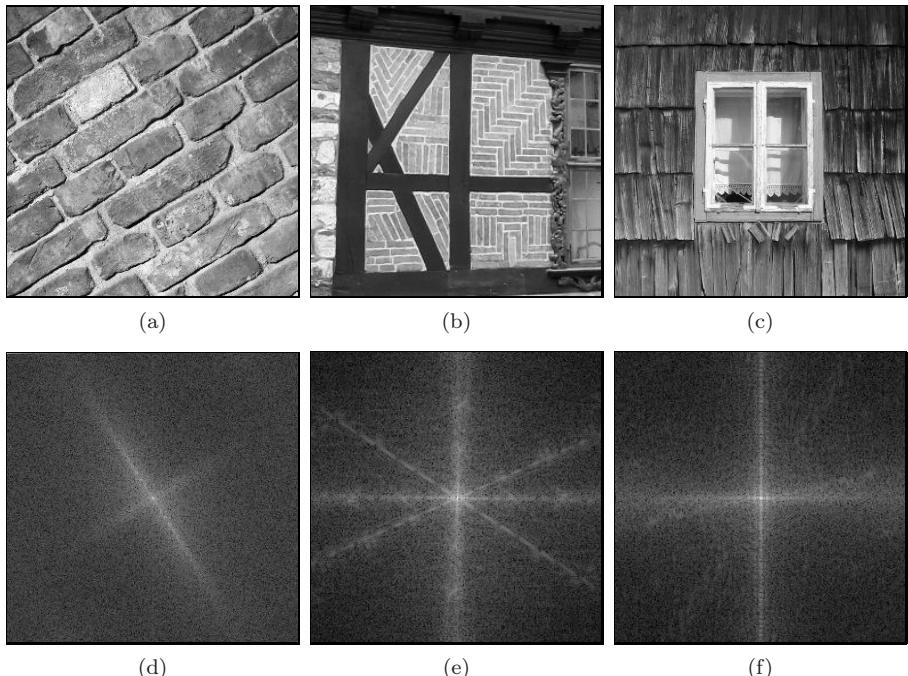


Figure 8.15 DFT—superposition of image patterns. Strong, oriented subpatterns (a–c) are easy to identify in the corresponding spectrum (d–f). Notice the broadband effects caused by straight structures, such as the dark beam on the wall in (b,e).

dimensional DFT. They can be invoked through the menu Process→FFT→FD Math.

8.5.3 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,

$$G_{\text{blur}} = G_{\text{orig}} \cdot H_{\text{blur}},$$

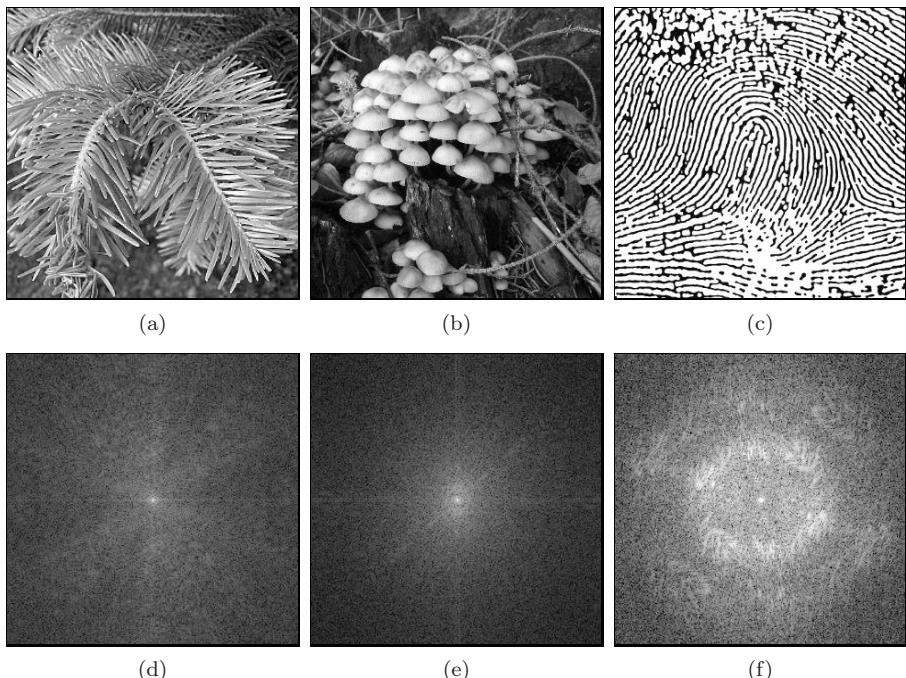


Figure 8.16 DFT—natural image patterns. Examples of repetitive structures in natural images (a–c) that are also visible in the corresponding spectrum (d–f).

it should be possible to reconstruct the original (nonblurred) image by computing the inverse Fourier transform of the expression

$$G_{\text{orig}}(m, n) = \frac{G_{\text{blur}}(m, n)}{H_{\text{blur}}(m, n)}.$$

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. 8.19 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. 8.19 (c)). However, as shown in Fig. 8.19 (d), large errors occur if the inverse filter deviates only marginally from the real filter, which quickly renders the method useless.

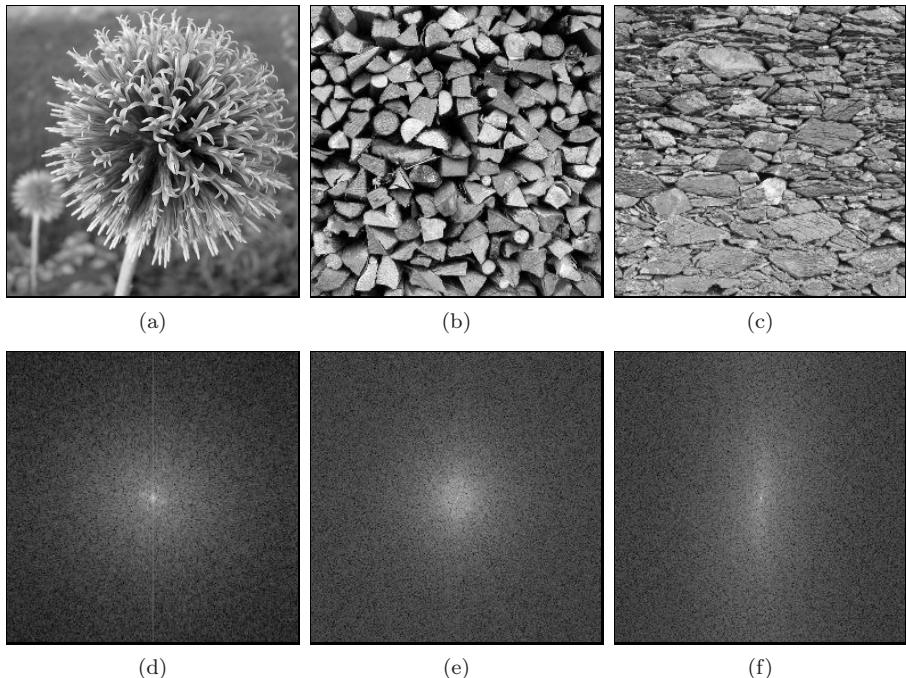


Figure 8.17 DFT—natural image patterns with no dominant orientation. The repetitive patterns contained in these images (a–c) have no common orientation or sufficiently regular spacing to stand out locally in the corresponding Fourier spectra (d–f).

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., [28, Sec. 5.4], [47, Sec. 8.3], [46, Sec. 17.8], [15, Ch. 16]).

8.6 Exercises

Exercise 8.1

Implement the two-dimensional DFT using the one-dimensional DFT, as described in Sec. 8.1.2. Apply the 2D DFT to real intensity images of arbitrary size and display the results (by converting to ImageJ `FloatProcessor` images). Implement the inverse transform and verify that the back-transformed result is identical to the original image.

Exercise 8.2

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.

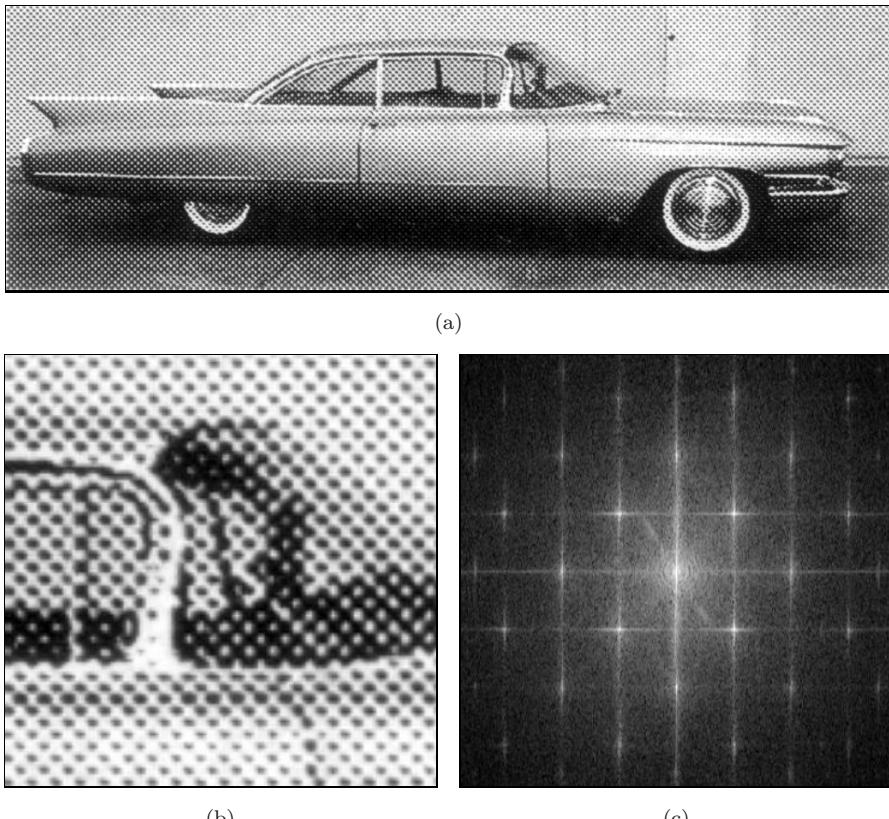


Figure 8.18 DFT of a print pattern. The regular diagonally oriented raster pattern (a, b) is clearly visible in the corresponding power spectrum (c). It is possible (at least in principle) to remove such patterns by erasing these peaks in the Fourier spectrum and reconstructing the smoothed image from the modified spectrum using the inverse DFT.

Exercise 8.3

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 8.4

Generalize Eqn. (8.10) and Eqns. (8.12)–(8.14) 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 8.5

Implement the *elliptical* and the *super-Gauss* windowing functions (Table 8.1) 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.



(a)



(b)



(c)



(d)

Figure 8.19 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).

9

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. (7.22)) 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 [47].

9.1 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

$$G(m) = \sqrt{\frac{2}{M}} \cdot \sum_{u=0}^{M-1} \left[g(u) \cdot c_m \cdot \cos\left(\pi \frac{m(2u+1)}{2M}\right) \right] \quad (9.1)$$

for $0 \leq m < M$, and the *inverse* transform is

$$g(u) = \sqrt{\frac{2}{M}} \cdot \sum_{m=0}^{M-1} \left[G(m) \cdot c_m \cdot \cos\left(\pi \frac{m(2u+1)}{2M}\right) \right] \quad (9.2)$$

for $0 \leq u < M$, respectively, with

$$c_m = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } m = 0 \\ 1 & \text{otherwise.} \end{cases} \quad (9.3)$$

Note that the index variables (u, m) are used differently in the forward transform (Eqn. (9.1)) and the inverse transform (Eqn. (9.2)), so the two transforms are—in contrast to the DFT—*not* symmetric.

9.1.1 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. (7.48)) and those of the DCT (Eqn. (9.1)),

$$\begin{aligned} \text{DFT: } C_m^M(u) &= \cos\left(2\pi \frac{mu}{M}\right), \\ \text{DCT: } D_m^M(u) &= \cos\left(\pi \frac{m(2u+1)}{2M}\right) = \cos\left(2\pi \frac{m(u+0.5)}{2M}\right), \end{aligned} \quad (9.4)$$

one can see that, for a given wave number m , the period ($\tau_m = 2\frac{M}{m}$) of the corresponding DCT basis function is double the period of the DFT basis functions ($\tau_m = \frac{M}{m}$). Notice that the DCT basis functions are also phase-shifted by 0.5 units.

Figure 9.1 shows the DCT basis functions $D_m^M(u)$ for the signal length $M = 8$ and wave numbers $m = 0 \dots 7$. For example, the basis function $D_7^8(u)$ for wave number $m = 7$ performs seven full cycles over a length of $2M = 16$ units and therefore has the radial frequency $\omega = m/2 = 3.5$.

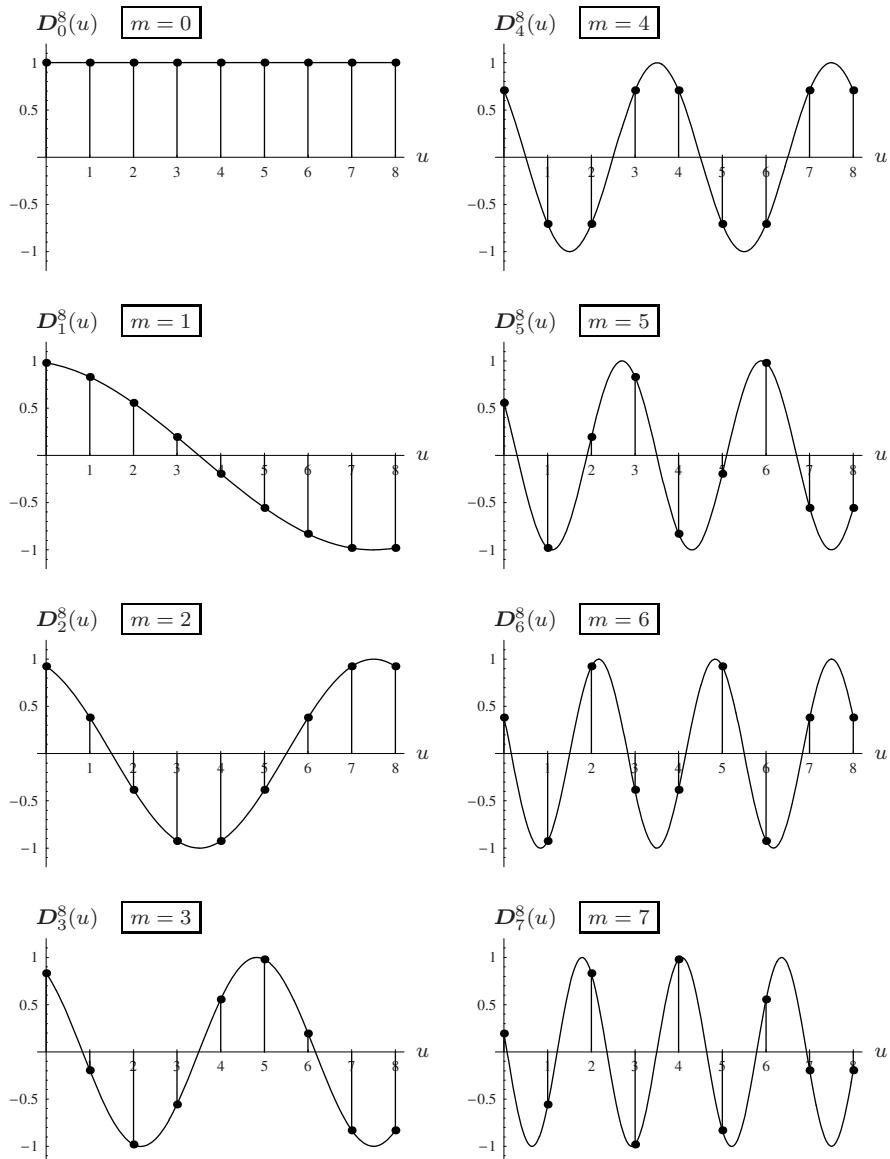


Figure 9.1 DCT basis functions $D_0^M(u) \dots D_7^M(u)$ for $M = 8$. Each plot shows both the discrete function (round dots) and the corresponding continuous function. Compared with the basis functions of the DFT (Figs. 7.11 and 7.12), all frequencies are divided in half and the DCT basis functions 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).

```

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 * 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 * m * (2 * u + 1)) / (2 * M);
27             double cosPhi = Math.cos(Phi);
28             sum += G[m] * cm * cosPhi;
29         }
30         g[u] = s * sum;
31     }
32     return g;
33 }
```

Program 9.1 One-dimensional DCT (Java implementation). The method `DCT()` computes the forward transform for a real-valued signal vector \mathbf{g} of arbitrary length according to the definition in Eqn. (9.1). The method returns the DCT spectrum as a real-valued vector of the same length as the input vector \mathbf{g} . The inverse transform `iDCT()` computes the inverse DCT for the real-valued cosine spectrum \mathbf{G} .

9.1.2 Implementing the One-Dimensional DCT

Since the DCT does not create any complex values and the forward and inverse transforms (Eqns. (9.1) and (9.2)) are almost identical, the whole procedure is fairly easy to implement in Java, as shown in Prog. 9.1. The only notable detail is that the factor c_m in Eqn. (9.1) is independent of the iteration variable u and can thus be computed outside the inner summation loop (see Prog. 9.1, line 7).

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 [47, p. 152].¹ The DCT is often used for image compression, in particular for JPEG compression, where the size of the transformed sub-images is fixed at 8×8 and the processing can be highly optimized.

9.2 Two-Dimensional DCT

The two-dimensional form of the DCT follows immediately from the one-dimensional definition (Eqns. (9.1) and (9.2)), resulting in the 2D forward transform

$$\begin{aligned} G(m, n) &= \frac{2}{\sqrt{MN}} \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} \left[g(u, v) \cdot c_m \cdot \cos\left(\frac{\pi(2u+1)m}{2M}\right) \cdot c_n \cdot \cos\left(\frac{\pi(2v+1)n}{2N}\right) \right] \\ &= \frac{2c_m c_n}{\sqrt{MN}} \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} \left[g(u, v) \cdot \mathbf{D}_m^M(u) \cdot \mathbf{D}_n^N(v) \right] \end{aligned} \quad (9.5)$$

for $0 \leq m < M$, $0 \leq n < N$, and the inverse transform

$$\begin{aligned} g(u, v) &= \frac{2}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \left[G(m, n) \cdot c_m \cdot \cos\left(\frac{\pi(2u+1)m}{2M}\right) \cdot c_n \cdot \cos\left(\frac{\pi(2v+1)n}{2N}\right) \right] \\ &= \frac{2}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \left[G(m, n) \cdot c_m \cdot \mathbf{D}_m^M(u) \cdot c_n \cdot \mathbf{D}_n^N(v) \right] \end{aligned} \quad (9.6)$$

for $0 \leq u < M$, $0 \leq v < N$. The coefficients c_m and c_n in Eqns. (9.5) and (9.6) are the same as in the one-dimensional case (Eqn. (9.3)). 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. (9.5)).

9.2.1 Separability

Similar to the DFT (see Eqn. (8.7)), the two-dimensional DCT can also be separated into two successive one-dimensional transforms. To make this fact clear, the forward DCT (Eqn. (9.5)) can be expressed in the following way:

$$G(m, n) = \sqrt{\frac{2}{N}} \cdot \sum_{v=0}^{N-1} \underbrace{\left[\left(\sqrt{\frac{2}{M}} \cdot \sum_{u=0}^{M-1} g(u, v) \cdot c_m \cdot \mathbf{D}_m^M(u) \right) \cdot c_n \cdot \mathbf{D}_n^N(v) \right]}_{\text{one-dimensional DCT of } g(\cdot, v)} \quad (9.7)$$

¹ See Vol. 1 [14, Appendix A.3] for a brief explanation of the $\mathcal{O}()$ notation.

The inner expression in Eqn. (9.7) corresponds to a one-dimensional DCT of the v th 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.

9.2.2 Examples

Figure 9.2 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.

9.3 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. 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 [15, 46].

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 [52].

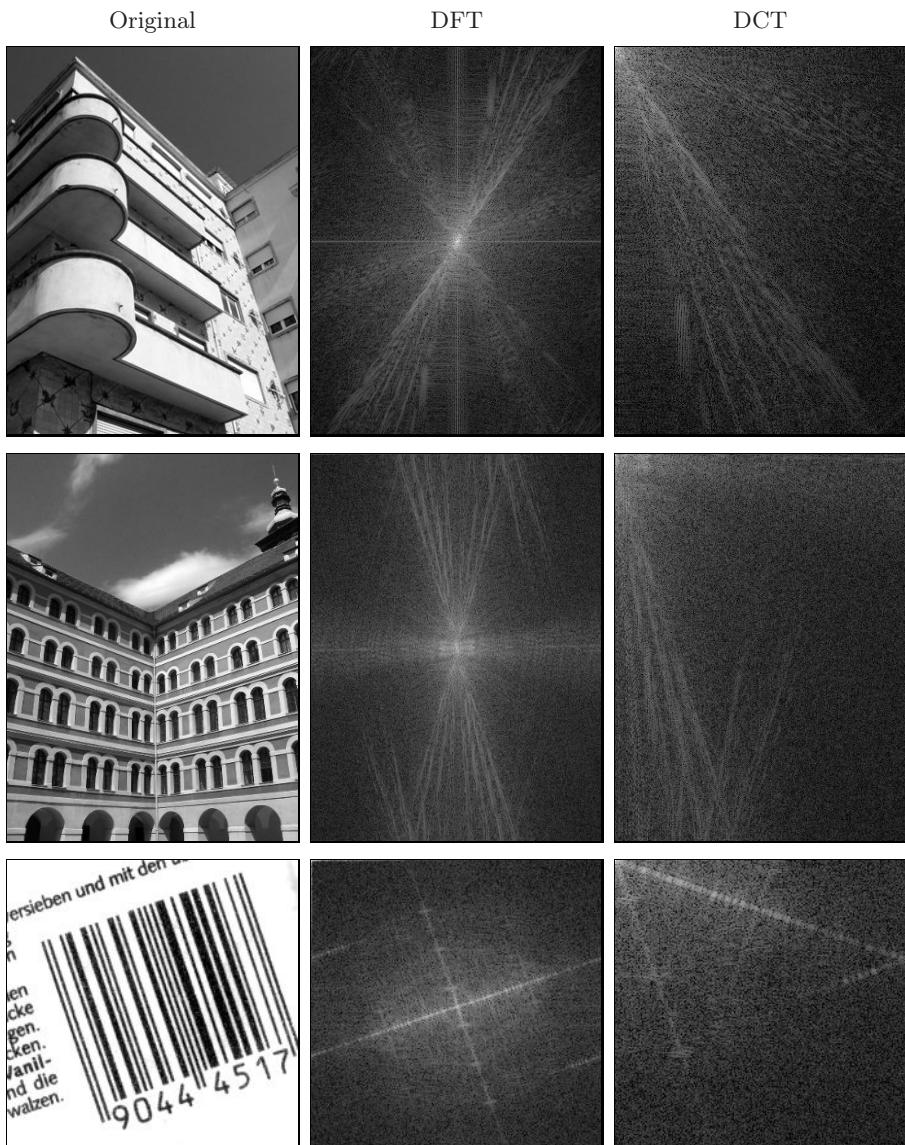


Figure 9.2 Two-dimensional DFT versus DCT. Both transforms show the frequency effects of image structures in a similar fashion. In the real-valued DCT spectrum (right), all coefficients are contained in a single quadrant and the frequency resolution is doubled compared with the DFT power spectrum (center). The DFT spectrum is centered as usual, while the origin of the DCT spectrum is located at the upper left corner. Both spectral plots display logarithmic intensity values.

9.4 Exercises

Exercise 9.1

Implement the two-dimensional DCT (Sec. 9.2) as an ImageJ plugin for images of arbitrary size. Make use of the fact that the 2D DCT can be performed as a sequence of one-dimensional transforms (see Eqn. (9.7)).

Exercise 9.2

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 9.3

Verify by numerical computation that the DCT basis functions $\mathbf{D}_m^M(u)$ for $0 \leq m, u < M$ (Eqn. (9.4)) are pairwise orthogonal; i. e., the inner product of the vectors $\mathbf{D}_m^M \cdot \mathbf{D}_n^M$ is zero for any pair $m \neq n$.

10

Geometric Operations

Common to all 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. 10.1. 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,

$$I(x, y) \rightarrow I'(x', y'); \quad (10.1)$$

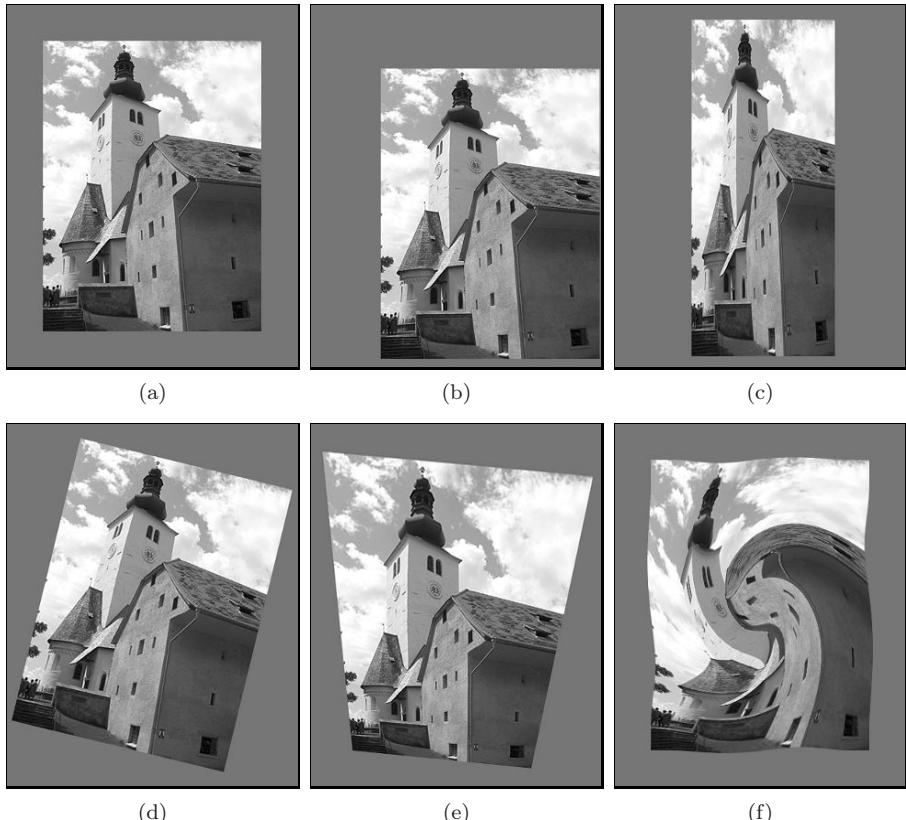


Figure 10.1 Typical examples for geometric operations: original image (a), translation (b), scaling (contracting or stretching) in x and y directions (c), rotation about the center (d), projective transformation (e), and nonlinear distortion (f).

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*

$$T : \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

that specifies for each original 2D coordinate point $\mathbf{x} = (x, y)$ the corresponding target point $\mathbf{x}' = (x', y')$ in the new image I' ,

$$\mathbf{x}' = T(\mathbf{x}). \quad (10.2)$$

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. 10.3.

10.1 2D Mapping Function

The mapping function $T()$ in Eqn. (10.2) is an arbitrary continuous function that for reasons of simplicity is often specified as two separate functions,

$$x' = T_x(x, y) \quad \text{and} \quad y' = T_y(x, y), \quad (10.3)$$

for the x and y components.

10.1.1 Simple Mappings

The simple mapping functions include translation, scaling, shearing, and rotation, defined as follows:

Translation (shift) by a vector (d_x, d_y) :

$$\begin{aligned} T_x : x' &= x + d_x \\ T_y : y' &= y + d_y \end{aligned} \quad \text{or} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix}. \quad (10.4)$$

Scaling (contracting or stretching) along the x or y axis by the factor s_x or s_y , respectively:

$$\begin{aligned} T_x : x' &= s_x \cdot x \\ T_y : y' &= s_y \cdot y \end{aligned} \quad \text{or} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \quad (10.5)$$

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):

$$\begin{aligned} T_x : x' &= x + b_x \cdot y \\ T_y : y' &= y + b_y \cdot x \end{aligned} \quad \text{or} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & b_x \\ b_y & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \quad (10.6)$$

Rotation by an angle α , with the coordinate origin being the center of rotation:

$$\begin{aligned} T_x : x' &= x \cdot \cos \alpha - y \cdot \sin \alpha \\ T_y : y' &= x \cdot \sin \alpha + y \cdot \cos \alpha \end{aligned} \quad \text{or}$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \quad (10.7)$$

Rotating the image by an angle α around an *arbitrary center point* $\mathbf{x}_c = (x_c, y_c)$ is accomplished by first translating the image by $(-x_c, -y_c)$, such that \mathbf{x}_c coincides with the origin, then rotating the image about the origin (as in Eqn. (10.7)), and finally shifting the image back by (x_c, y_c) :

$$\begin{aligned} T_x : x' &= x_c + (x - x_c) \cdot \cos \alpha - (y - y_c) \cdot \sin \alpha \\ T_y : y' &= y_c + (x - x_c) \cdot \sin \alpha + (y - y_c) \cdot \cos \alpha \end{aligned} \quad \text{or}$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x_c \\ y_c \end{pmatrix} + \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \cdot \begin{pmatrix} x - x_c \\ y - y_c \end{pmatrix}. \quad (10.8)$$

10.1.2 Homogeneous Coordinates

The operations listed in Eqns. (10.4)–(10.7) constitute the important class of “affine” mapping functions or *affine transformations* (see also Sec. 10.1.3). To simplify the concatenation of mappings, it is advantageous to specify all operations in the form of vector-matrix multiplications, as in Eqns. (10.5)–(10.7). Notice that the translation Eqn. (10.4), 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., [22, 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,

$$\mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix} \quad \text{converts to} \quad \hat{\mathbf{x}} = \begin{pmatrix} \hat{x} \\ \hat{y} \\ h \end{pmatrix} = \begin{pmatrix} h x \\ h y \\ h \end{pmatrix}. \quad (10.9)$$

Thus every ordinary 2D (Cartesian) coordinate pair $\mathbf{x} = (x, y)^T$ is replaced by a three-dimensional homogeneous coordinate vector $\hat{\mathbf{x}} = (\hat{x}, \hat{y}, h)^T$ with arbitrary $h \neq 0$. If the last component (h) of the homogeneous vector $\hat{\mathbf{x}}$ is nonzero, the components of the corresponding Cartesian vector $(x, y)^T$ are found to be

$$x = \frac{\hat{x}}{h} \quad \text{and} \quad y = \frac{\hat{y}}{h}. \quad (10.10)$$

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 $\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2$ represent the same Cartesian point \mathbf{x} if they are multiples of each other; i. e.,

$$\text{if } \hat{\mathbf{x}}_1 = s \cdot \hat{\mathbf{x}}_2, \quad \text{then} \quad \mathbf{x}_1 = \mathbf{x}_2 = \mathbf{x} \quad (10.11)$$

for $s \neq 0$. For example, the homogeneous coordinates $\hat{\mathbf{x}}_1 = (3, 2, 1)^T$, $\hat{\mathbf{x}}_2 = (-6, -4, -2)^T$, and $\hat{\mathbf{x}}_3 = (30, 20, 10)^T$ are all equivalent to the Cartesian coordinate vector $\mathbf{x} = (3, 2)^T$.

10.1.3 Affine (Three-Point) Mapping

Using homogeneous coordinates, we can rewrite the 2D translation (Eqn. (10.4)) as

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x+d_x \\ y+d_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad (10.12)$$

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 $\hat{x}' = \mathbf{A} \cdot \hat{x}$ or

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \quad (10.13)$$

This 2D coordinate transformation is called an “affine mapping” with the six parameters $a_{11} \dots a_{23}$, where a_{13} , a_{23} specify the translation (equivalent to d_x, d_y in Eqn. (10.4)) and $a_{11}, a_{12}, a_{21}, a_{22}$ aggregate the scaling, shearing, and rotation terms (Eqns. (10.5)–(10.7)). For example, the affine transformation matrix for a rotation about the origin by an angle α is

$$\mathbf{A}_{\text{rot}} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}. \quad (10.14)$$

Compound transformations can be obtained by consecutive matrix multiplications (from right to left). For example, the transformation matrix for a rotation by α about a given center point $\mathbf{x}_c = (-x_c, -y_c)$, consisting of a translation to the origin followed by a rotation and another translation (see Eqn. (10.8)) is

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & x_c \\ 0 & 1 & y_c \\ 0 & 0 & 1 \end{pmatrix}}_{\text{translation by } (x_c, y_c)} \cdot \underbrace{\begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\text{rotation by } \alpha} \cdot \underbrace{\begin{pmatrix} 1 & 0 & -x_c \\ 0 & 1 & -y_c \\ 0 & 0 & 1 \end{pmatrix}}_{\text{translation by } (-x_c, -y_c)} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (10.15)$$

$$= \begin{pmatrix} \cos \alpha & -\sin \alpha & x_c \cdot (1 - \cos \alpha) + y_c \cdot \sin \alpha \\ \sin \alpha & \cos \alpha & y_c \cdot (1 - \cos \alpha) - x_c \cdot \sin \alpha \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (10.16)$$

$$= \begin{pmatrix} x_c + (x - x_c) \cdot \cos \alpha - (y - y_c) \cdot \sin \alpha \\ y_c + (x - x_c) \cdot \sin \alpha + (y - y_c) \cdot \cos \alpha \\ 1 \end{pmatrix}. \quad (10.17)$$

This is of course the same result for (x', y') as in Eqn. (10.8).

Note that multiplying two affine transformation matrices always gives another affine transformation. Also, an affine mapping transforms straight lines to straight lines, triangles to triangles, and rectangles to parallelograms, as illustrated in Fig. 10.2. The distance ratio between points on a straight line remains unchanged by this type of mapping function.

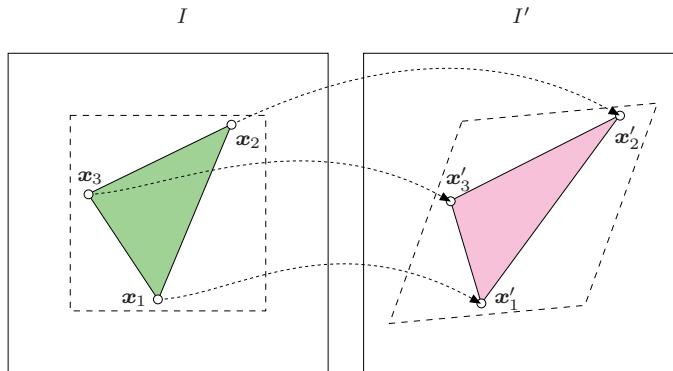


Figure 10.2 Affine mapping. This mapping transforms straight lines to straight lines, triangles to triangles, and rectangles to parallelograms. Parallel lines remain parallel, and the distance ratio between points on a straight line does not change. An affine 2D transformation is uniquely specified by three pairs of corresponding points; e.g., $(\mathbf{x}_1, \mathbf{x}'_1)$, $(\mathbf{x}_2, \mathbf{x}'_2)$, and $(\mathbf{x}_3, \mathbf{x}'_3)$.

Affine transformation parameters from three point pairs

The six parameters of the 2D affine mapping (Eqn. (10.13)) are uniquely determined by three pairs of corresponding points $(\mathbf{x}_1, \mathbf{x}'_1)$, $(\mathbf{x}_2, \mathbf{x}'_2)$, $(\mathbf{x}_3, \mathbf{x}'_3)$, with the first point $\mathbf{x}_i = (x_i, y_i)$ of each pair located in the original image and the corresponding point $\mathbf{x}'_i = (x'_i, y'_i)$ located in the target image. From these six coordinate values, the six transformation parameters $a_{11} \dots a_{23}$ are derived by solving the system of linear equations

$$\begin{aligned} x'_1 &= a_{11} \cdot x_1 + a_{12} \cdot y_1 + a_{13}, & y'_1 &= a_{21} \cdot x_1 + a_{22} \cdot y_1 + a_{23}, \\ x'_2 &= a_{11} \cdot x_2 + a_{12} \cdot y_2 + a_{13}, & y'_2 &= a_{21} \cdot x_2 + a_{22} \cdot y_2 + a_{23}, \\ x'_3 &= a_{11} \cdot x_3 + a_{12} \cdot y_3 + a_{13}, & y'_3 &= a_{21} \cdot x_3 + a_{22} \cdot y_3 + a_{23}, \end{aligned} \quad (10.18)$$

provided that the points (vectors) \mathbf{x}_1 , \mathbf{x}_2 , \mathbf{x}_3 are linearly independent (i.e., that they do not lie on a common straight line). Since Eqn. (10.18) 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

$$\begin{aligned} a_{11} &= \frac{1}{d} \cdot [y_1(x'_2 - x'_3) + y_2(x'_3 - x'_1) + y_3(x'_1 - x'_2)], \\ a_{12} &= \frac{1}{d} \cdot [x_1(x'_3 - x'_2) + x_2(x'_1 - x'_3) + x_3(x'_2 - x'_1)], \\ a_{21} &= \frac{1}{d} \cdot [y_1(y'_2 - y'_3) + y_2(y'_3 - y'_1) + y_3(y'_1 - y'_2)], \\ a_{22} &= \frac{1}{d} \cdot [x_1(y'_3 - y'_2) + x_2(y'_1 - y'_3) + x_3(y'_2 - y'_1)], \\ a_{13} &= \frac{1}{d} \cdot [x_1(y_3x'_2 - y_2x'_3) + x_2(y_1x'_3 - y_3x'_1) + x_3(y_2x'_1 - y_1x'_2)], \\ a_{23} &= \frac{1}{d} \cdot [x_1(y_3y'_2 - y_2y'_3) + x_2(y_1y'_3 - y_3y'_1) + x_3(y_2y'_1 - y_1y'_2)], \end{aligned} \quad (10.19)$$

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. 10.2.2), can be found by computing the inverse of the transformation matrix (Eqn. (10.13)),

$$\begin{aligned} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \\ &= \frac{1}{a_{11}a_{22} - a_{12}a_{21}} \begin{pmatrix} a_{22} & -a_{12} & a_{12}a_{23} - a_{13}a_{22} \\ -a_{21} & a_{11} & a_{13}a_{21} - a_{11}a_{23} \\ 0 & 0 & a_{11}a_{22} - a_{12}a_{21} \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}. \end{aligned} \quad (10.20)$$

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 $(\mathbf{x}_i, \mathbf{x}'_i)$ using Eqn. (10.19) with source and target coordinates interchanged.

10.1.4 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. 10.3). This is particularly useful for deforming images controlled by mesh partitioning, as described in Sec. 10.1.7. To map from an arbitrary sequence of four 2D points $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4)$ to a set of corresponding points $(\mathbf{x}'_1, \mathbf{x}'_2, \mathbf{x}'_3, \mathbf{x}'_4)$, 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}) :

$$\begin{pmatrix} \hat{x}' \\ \hat{y}' \\ h' \end{pmatrix} = \begin{pmatrix} h'x' \\ h'y' \\ h' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \quad (10.21)$$

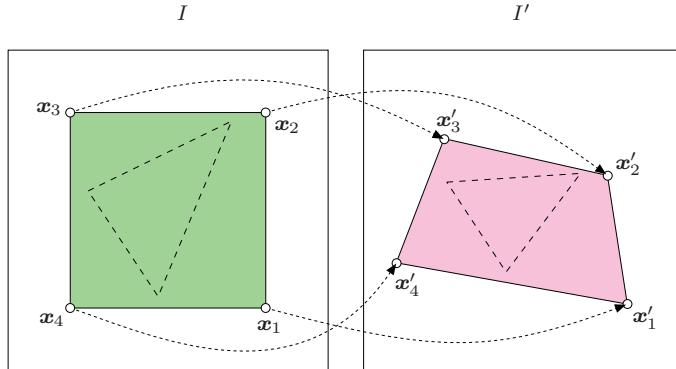


Figure 10.3 Projective mapping. Four pairs of corresponding 2D points uniquely specify a projective transformation. Straight lines are again mapped to straight lines, and a rectangle is mapped to some quadrilateral. In general, neither parallelism between straight lines nor distance ratios are preserved.

In Cartesian coordinates, the resulting mapping functions

$$x' = \frac{1}{h'} \cdot (a_{11} x + a_{12} y + a_{13}) = \frac{a_{11} x + a_{12} y + a_{13}}{a_{31} x + a_{32} y + 1}, \quad (10.22)$$

$$y' = \frac{1}{h'} \cdot (a_{21} x + a_{22} y + a_{23}) = \frac{a_{21} x + a_{22} y + a_{23}}{a_{31} x + a_{32} y + 1}, \quad (10.23)$$

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 N th-order algebraic curve onto another N th-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. 10.3) 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”.

Projective transformation parameters from four point pairs

Given four pairs of corresponding 2D points, $(x_1, x'_1), \dots, (x_4, x'_4)$, with one point $x_i = (x_i, y_i)$ in the source image and the second point $x'_i = (x'_i, y'_i)$ 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. (10.22), we get for each point pair $i = 1 \dots 4$ a pair

of linear equations

$$\begin{aligned}x'_i &= a_{11}x_i + a_{12}y_i + a_{13} - a_{31}x_i x'_i - a_{32}y_i x'_i, \\y'_i &= a_{21}x_i + a_{22}y_i + a_{23} - a_{31}x_i y'_i - a_{32}y_i y'_i,\end{aligned}\quad (10.24)$$

for the eight unknowns $a_{11} \dots a_{32}$. Combining the resulting eight equations in matrix notation gives

$$\begin{pmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1 x'_1 & -y_1 x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1 y'_1 & -y_1 y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2 x'_2 & -y_2 x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2 y'_2 & -y_2 y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3 x'_3 & -y_3 x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3 y'_3 & -y_3 y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4 x'_4 & -y_4 x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4 y'_4 & -y_4 y'_4 \end{pmatrix} \cdot \begin{pmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \end{pmatrix}. \quad (10.25)$$

or

$$\mathbf{x}' = \mathbf{M} \cdot \mathbf{a}. \quad (10.26)$$

The unknown parameters $\mathbf{a} = (a_{11}, a_{12}, \dots, a_{32})^T$ can be found by solving the system of linear equations above using standard numerical methods such as the Gauss algorithm [11, p. 276].¹

Inverse mapping

In general, a *linear* transformation of the form $\mathbf{x}' = \mathbf{A} \cdot \mathbf{x}$ can be inverted by computing the inverse of the matrix \mathbf{A} (i.e., $\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{x}'$), provided that \mathbf{A} is nonsingular ($\text{Det}(\mathbf{A}) \neq 0$). The inverse of a 3×3 matrix \mathbf{A} is relatively easy to find using the relation

$$\mathbf{A}^{-1} = \frac{1}{\text{Det}(\mathbf{A})} \cdot \mathbf{A}_{\text{adj}} \quad (10.27)$$

between the determinant $\text{Det}(\mathbf{A})$ and the corresponding adjoint matrix \mathbf{A}_{adj} [11, pp. 251, 260]. For an arbitrary 3×3 matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix},$$

¹ We recommend relying on existing numerical software libraries for this purpose. Several free libraries 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.

the determinant is

$$\text{Det}(\mathbf{A}) = a_{11} a_{22} a_{33} + a_{12} a_{23} a_{31} + a_{13} a_{21} a_{32} - a_{11} a_{23} a_{32} - a_{12} a_{21} a_{33} - a_{13} a_{22} a_{31} \quad (10.28)$$

and its adjoint matrix is

$$\mathbf{A}_{\text{adj}} = \begin{pmatrix} a_{22} a_{33} - a_{23} a_{32} & a_{13} a_{32} - a_{12} a_{33} & a_{12} a_{23} - a_{13} a_{22} \\ a_{23} a_{31} - a_{21} a_{33} & a_{11} a_{33} - a_{13} a_{31} & a_{13} a_{21} - a_{11} a_{23} \\ a_{21} a_{32} - a_{22} a_{31} & a_{12} a_{31} - a_{11} a_{32} & a_{11} a_{22} - a_{12} a_{21} \end{pmatrix}. \quad (10.29)$$

In the special case of a projective mapping, the coefficient $a_{33} = 1$ (Eqn. (10.21)), which slightly simplifies the computation. Since scalar multiples of homogeneous vectors are all equivalent in Cartesian space, the multiplication by the factor $1/\text{Det}(\mathbf{A})$ in Eqn. (10.27) can be omitted. Thus, to invert the linear transformation, we only need to multiply the homogeneous coordinate vector with the adjoint matrix \mathbf{A}_{adj} and (if needed) “homogenize” the resulting vector,

$$\begin{pmatrix} \hat{x} \\ \hat{y} \\ h \end{pmatrix} = \mathbf{A}_{\text{adj}} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \quad \text{and subsequently} \quad \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \frac{1}{h} \begin{pmatrix} \hat{x} \\ \hat{y} \\ h \end{pmatrix}. \quad (10.30)$$

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. (10.20)) 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 the unit square, which avoids iteratively solving a system of equations. The projective mapping, shown in Fig. 10.4, from the corner points of the unit square \mathcal{S}_1 to an arbitrary quadrilateral $\mathcal{Q} = (\mathbf{x}'_1, \dots, \mathbf{x}'_4)$ with

$$\begin{array}{ll} (0, 0) \rightarrow \mathbf{x}'_1 & (1, 1) \rightarrow \mathbf{x}'_3 \\ (1, 0) \rightarrow \mathbf{x}'_2 & (0, 1) \rightarrow \mathbf{x}'_4 \end{array}$$

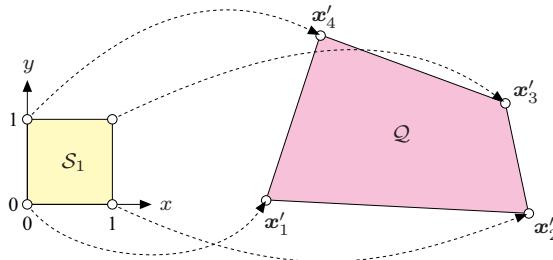


Figure 10.4 Projective mapping from the unit square S_1 to an arbitrary quadrilateral $\mathcal{Q} = (\mathbf{x}'_1, \dots, \mathbf{x}'_4)$.

reduces the system of equations in Eqn. (10.25) to

$$\begin{aligned} x'_1 &= a_{13}, \\ y'_1 &= a_{23}, \\ x'_2 &= a_{11} + a_{13} - a_{31} \cdot x'_2, \\ y'_2 &= a_{21} + a_{23} - a_{31} \cdot y'_2, \\ x'_3 &= a_{11} + a_{12} + a_{13} - a_{31} \cdot x'_3 - a_{32} \cdot x'_3, \\ y'_3 &= a_{21} + a_{22} + a_{23} - a_{31} \cdot y'_3 - a_{32} \cdot y'_3, \\ x'_4 &= a_{12} + a_{13} - a_{32} \cdot x'_4, \\ y'_4 &= a_{22} + a_{23} - a_{32} \cdot y'_4. \end{aligned} \quad (10.31)$$

This set of equations has the following closed-form solution for the eight unknown transformation parameters $a_{11}, a_{12}, \dots, a_{32}$:

$$a_{31} = \frac{(x'_1 - x'_2 + x'_3 - x'_4) \cdot (y'_4 - y'_3) - (y'_1 - y'_2 + y'_3 - y'_4) \cdot (x'_4 - x'_3)}{(x'_2 - x'_3) \cdot (y'_4 - y'_3) - (x'_4 - x'_3) \cdot (y'_2 - y'_3)}, \quad (10.32)$$

$$a_{32} = \frac{(y'_1 - y'_2 + y'_3 - y'_4) \cdot (x'_2 - x'_3) - (x'_1 - x'_2 + x'_3 - x'_4) \cdot (y'_2 - y'_3)}{(x'_2 - x'_3) \cdot (y'_4 - y'_3) - (x'_4 - x'_3) \cdot (y'_2 - y'_3)}, \quad (10.33)$$

$$a_{11} = x'_2 - x'_1 + a_{31} x'_2, \quad a_{12} = x'_4 - x'_1 + a_{32} x'_4, \quad a_{13} = x'_1, \quad (10.34)$$

$$a_{21} = y'_2 - y'_1 + a_{31} y'_2, \quad a_{22} = y'_4 - y'_1 + a_{32} y'_4, \quad a_{23} = y'_1. \quad (10.35)$$

By computing the inverse of the corresponding 3×3 transformation matrix (Eqn. (10.27)), the mapping may be *reversed* to transform an arbitrary quadrilateral to the unit square. A mapping T between two arbitrary quadrilaterals

$$\mathcal{Q}_1 \xrightarrow{T} \mathcal{Q}_2$$

can thus be implemented by combining a reversed mapping and a forward mapping via the unit square [79, p. 55] [33]. As illustrated in Fig. 10.5, the

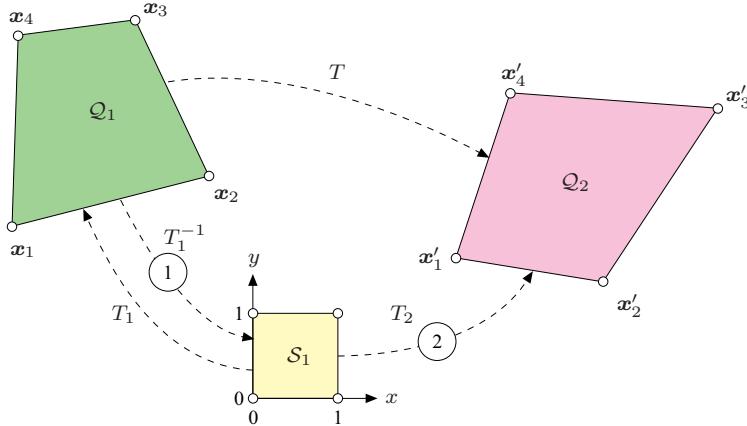


Figure 10.5 Two-step projective transformation between arbitrary quadrilaterals. In the first step, quadrilateral \mathcal{Q}_1 is transformed to the unit square \mathcal{S}_1 by the inverse mapping function T_1^{-1} . In the second step, T_2 transforms the square \mathcal{S}_1 to the target quadrilateral \mathcal{Q}_2 . The complete mapping T results from the concatenation of the mappings T_1^{-1} and T_2 .

transformation of the first quadrilateral $\mathcal{Q}_1 = (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4)$ to the second quadrilateral $\mathcal{Q}_2 = (\mathbf{x}'_1, \mathbf{x}'_2, \mathbf{x}'_3, \mathbf{x}'_4)$ 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 :

$$\mathcal{Q}_1 \xrightarrow{T_1^{-1}} \mathcal{S}_1 \xrightarrow{T_2} \mathcal{Q}_2. \quad (10.36)$$

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 (\mathbf{x}_i and \mathbf{x}'_i , respectively) into Eqns. (10.32)–(10.35). The complete transformation T is then the concatenation of the two transformations T_1^{-1} and T_2 ,

$$\mathbf{x}' = T(\mathbf{x}) = T_2(T_1^{-1}(\mathbf{x})), \quad (10.37)$$

or, expressed in matrix notation,

$$\mathbf{x}' = \mathbf{A} \cdot \mathbf{x} = \mathbf{A}_2 \cdot \mathbf{A}_1^{-1} \cdot \mathbf{x}. \quad (10.38)$$

Of course, the matrix $\mathbf{A} = \mathbf{A}_2 \cdot \mathbf{A}_1^{-1}$ 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:

$$\begin{aligned} \mathcal{Q}_1 : \quad \mathbf{x}_1 &= (2, 5) & \mathbf{x}_2 &= (4, 6) & \mathbf{x}_3 &= (7, 9) & \mathbf{x}_4 &= (5, 9) \\ \mathcal{Q}_2 : \quad \mathbf{x}'_1 &= (4, 3) & \mathbf{x}'_2 &= (5, 2) & \mathbf{x}'_3 &= (9, 3) & \mathbf{x}'_4 &= (7, 5) \end{aligned}$$

Using Eqns. (10.32)–(10.35), the transformation parameters (matrices) for the projective mappings from the unit \mathcal{S}_1 square to the quadrilaterals $\mathbf{A}_1 : \mathcal{S}_1 \rightarrow \mathcal{Q}_1$ and $\mathbf{A}_2 : \mathcal{S}_1 \rightarrow \mathcal{Q}_2$ are obtained as

$$\mathbf{A}_1 = \begin{pmatrix} 3.3\dot{3} & 0.50 & 2.00 \\ 3.00 & -0.50 & 5.00 \\ 0.3\dot{3} & -0.50 & 1.00 \end{pmatrix} \quad \text{and} \quad \mathbf{A}_2 = \begin{pmatrix} 1.00 & -0.50 & 4.00 \\ -1.00 & -0.50 & 3.00 \\ 0.00 & -0.50 & 1.00 \end{pmatrix}.$$

Concatenating the inverse mapping \mathbf{A}_1^{-1} with \mathbf{A}_2 (by matrix multiplication), we get the complete mapping $\mathbf{A} = \mathbf{A}_2 \cdot \mathbf{A}_1^{-1}$ with

$$\mathbf{A}_1^{-1} = \begin{pmatrix} 0.60 & -0.45 & 1.05 \\ -0.40 & 0.80 & -3.20 \\ -0.40 & 0.55 & -0.95 \end{pmatrix} \quad \text{and} \quad \mathbf{A} = \begin{pmatrix} -0.80 & 1.35 & -1.15 \\ -1.60 & 1.70 & -2.30 \\ -0.20 & 0.15 & 0.65 \end{pmatrix}.$$

The Java method `makeMapping()` in class `ProjectiveMapping` (p. 245) shows an implementation of this two-step technique.

10.1.5 Bilinear Mapping

Similar to the projective transformation (Eqn. (10.21)), the bilinear mapping function

$$\begin{aligned} T_x : x' &= a_1x + a_2y + a_3xy + a_4, \\ T_y : y' &= b_1x + b_2y + b_3xy + b_4, \end{aligned} \tag{10.39}$$

is specified with four pairs of corresponding points and has eight parameters $(a_1 \dots a_4, b_1 \dots b_4)$. 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 $(\mathbf{x}_1, \mathbf{x}'_1) \dots (\mathbf{x}_4, \mathbf{x}'_4)$. In the general case, for a bilinear mapping between arbitrary quadrilaterals, the coefficients $a_1 \dots a_4, b_1 \dots b_4$ (Eqn. (10.39)) are found as the solution of two separate systems of equations, each with four unknowns:

$$\begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & x_1 \cdot y_1 & 1 \\ x_2 & y_2 & x_2 \cdot y_2 & 1 \\ x_3 & y_3 & x_3 \cdot y_3 & 1 \\ x_4 & y_4 & x_4 \cdot y_4 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \quad \text{or} \quad \mathbf{x} = \mathbf{M} \cdot \mathbf{a}, \tag{10.40}$$

$$\begin{pmatrix} y'_1 \\ y'_2 \\ y'_3 \\ y'_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & x_1 \cdot y_1 & 1 \\ x_2 & y_2 & x_2 \cdot y_2 & 1 \\ x_3 & y_3 & x_3 \cdot y_3 & 1 \\ x_4 & y_4 & x_4 \cdot y_4 & 1 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \quad \text{or} \quad \mathbf{y} = \mathbf{M} \cdot \mathbf{b}. \tag{10.41}$$

These equations can again be solved using standard numerical techniques, as described on page 199. A sample implementation of this computation is shown by the Java method `makeInverseMapping()` inside the class `BilinearMapping` on page 247.

In the special case of bilinearly mapping the unit square \mathcal{S}_1 to an arbitrary quadrilateral $\mathcal{Q} = (\mathbf{x}'_1, \dots, \mathbf{x}'_4)$, the parameters $a_1 \dots a_4$ and $b_1 \dots b_4$ 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 10.6 shows results of the affine, projective, and bilinear transformations applied to a simple test pattern. The affine transformation (Fig. 10.6 (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. 10.6 (c, d)).

10.1.6 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 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. 10.2.2)—is not always easy or may even be impossible. In the following three examples, we therefore look straight at the inverse maps

$$\mathbf{x} = T^{-1}(\mathbf{x}')$$

without really bothering about the corresponding forward transformations.

“Twirl” transformation

The twirl mapping causes the image to be rotated around a given anchor point $\mathbf{x}_c = (x_c, y_c)$ with a space-variant rotation angle, which has a fixed value α at the center \mathbf{x}_c and decreases linearly with the radial distance from the center.

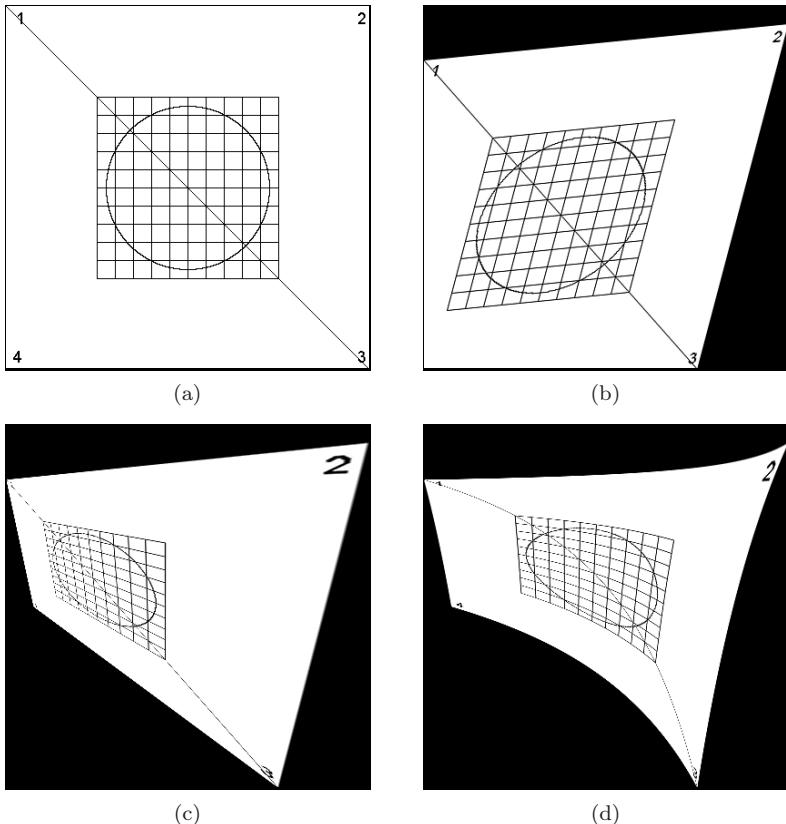


Figure 10.6 Geometric transformations compared: original image (a), affine transformation with respect to the triangle 1-2-3 (b), projective transformation (c), and bilinear transformation (d).

The image remains unchanged outside the limiting radius r_{\max} . The corresponding (inverse) mapping function is defined as

$$T_x^{-1} : \quad x = \begin{cases} x_c + r \cdot \cos(\beta) & \text{for } r \leq r_{\max} \\ x' & \text{for } r > r_{\max}, \end{cases} \quad (10.42)$$

$$T_y^{-1} : \quad y = \begin{cases} y_c + r \cdot \sin(\beta) & \text{for } r \leq r_{\max} \\ y' & \text{for } r > r_{\max}, \end{cases} \quad (10.43)$$

with

$$d_x = x' - x_c, \quad r = \sqrt{d_x^2 + d_y^2},$$

$$d_y = y' - y_c, \quad \beta = \text{Arctan}(d_y, d_x) + \alpha \cdot \left(\frac{r_{\max} - r}{r_{\max}} \right).$$

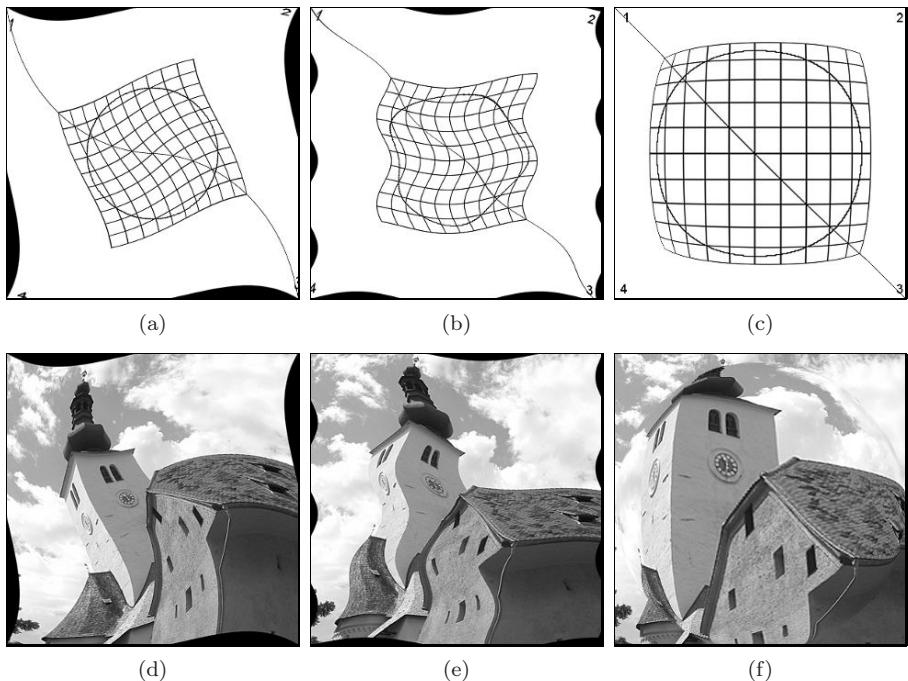


Figure 10.7 Various nonlinear image deformations: *twirl* (a, d), *ripple* (b, e), and *sphere* (c, f) transformations. The original (source) images are shown in Fig. 10.6(a) and Fig. 10.1 (a), respectively.

Figure 10.7 (a, d) shows a twirl mapping with the anchor point \mathbf{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 $\alpha = 43^\circ$ at the center. A Java implementation of this transformation is shown in the class `TwirlMapping` on page 247.

“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:

$$T_x^{-1} : x = x' + a_x \cdot \sin\left(\frac{2\pi \cdot y'}{\tau_x}\right), \quad (10.44)$$

$$T_y^{-1} : y = y' + a_y \cdot \sin\left(\frac{2\pi \cdot x'}{\tau_y}\right). \quad (10.45)$$

An example for the ripple mapping with $\tau_x = 120$, $\tau_y = 250$, $a_x = 10$, and $a_y = 15$ is shown in Fig. 10.7 (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 $\mathbf{x}_c = (x_c, y_c)$ of the lens center, the radius of the lens r_{\max} and its refraction index ρ . The corresponding mapping functions are defined as

$$T_x^{-1} : x = x' - \begin{cases} z \cdot \tan(\beta_x) & \text{for } r \leq r_{\max} \\ 0 & \text{for } r > r_{\max}, \end{cases} \quad (10.46)$$

$$T_y^{-1} : y = y' - \begin{cases} z \cdot \tan(\beta_y) & \text{for } r \leq r_{\max} \\ 0 & \text{for } r > r_{\max}, \end{cases} \quad (10.47)$$

with

$$\begin{aligned} d_x &= x' - x_c, & r &= \sqrt{d_x^2 + d_y^2}, & \beta_x &= \left(1 - \frac{1}{\rho}\right) \cdot \sin^{-1}\left(\frac{d_x}{\sqrt{(d_x^2 + z^2)}}\right), \\ d_y &= y' - y_c, & z &= \sqrt{r_{\max}^2 - r^2}, & \beta_y &= \left(1 - \frac{1}{\rho}\right) \cdot \sin^{-1}\left(\frac{d_y}{\sqrt{(d_y^2 + z^2)}}\right). \end{aligned}$$

Figure 10.7 (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 $\rho = 1.8$.

10.1.7 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 $\mathbf{x}_1 \dots \mathbf{x}_n$ are precisely mapped onto a given set of target points $\mathbf{x}'_1 \dots \mathbf{x}'_n$. For $n = 3$, this problem can be solved with an affine mapping (see Sec. 10.1.3), and for $n = 4$ we could use a projective or bilinear mapping (see Secs. 10.1.4 and 10.1.5). A precise global mapping of $n > 4$ points requires a more complicated function $T(\mathbf{x})$ (e.g., a two-dimensional n th-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. 10.8.

For a *triangular* mesh partitioning (Fig. 10.8 (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

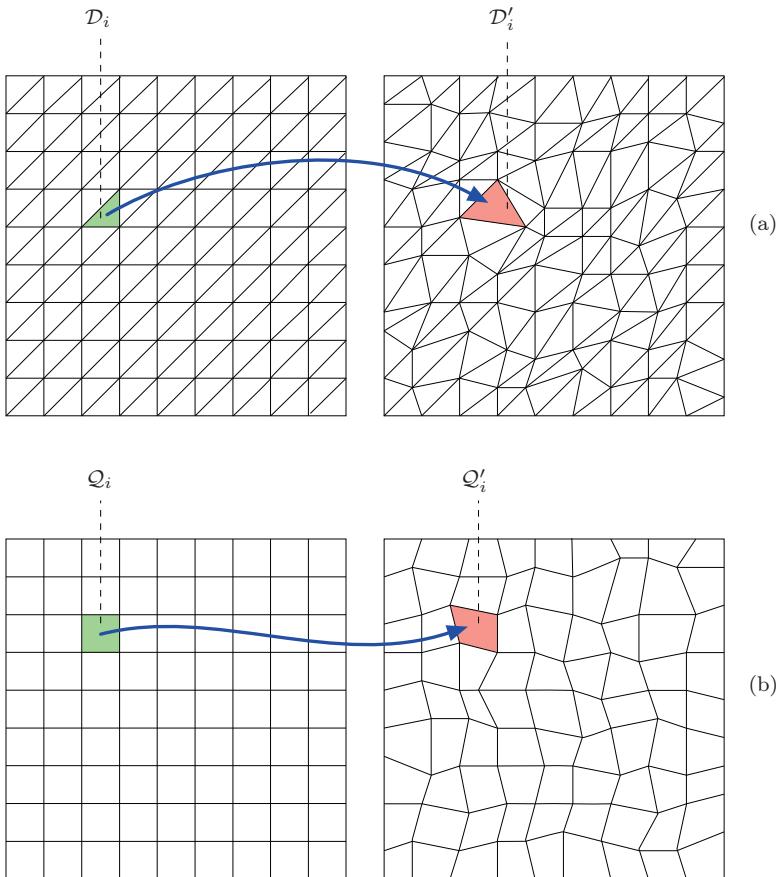


Figure 10.8 Mesh partitioning. Almost arbitrary image deformations can be implemented by partitioning the image plane into nonoverlapping triangles $\mathcal{D}_i, \mathcal{D}'_i$ (a) or quadrilaterals $\mathcal{Q}_i, \mathcal{Q}'_i$ (b) and applying simple local transformations. Every patch in the resulting mesh is transformed separately with the required transformation parameters derived from the corresponding three or four corner points, respectively.

patch in a mesh partitioning composed of quadrilaterals \mathcal{Q}_i (Fig. 10.8 (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” [79], which performs a stepwise geometric

transformation from one image to another while simultaneously blending their intensity (or color) values.²

10.2 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.

10.2.1 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. 10.9. 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' 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

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.

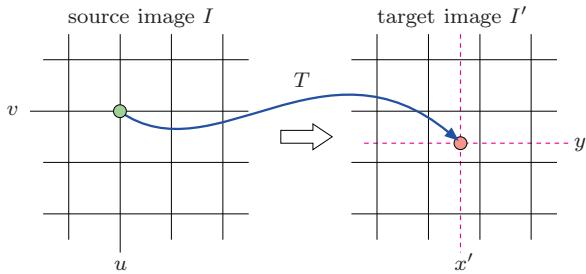


Figure 10.9 Source-to-target mapping. For each discrete pixel position (u, v) in the source image I , the corresponding (continuous) target position (x', y') is found by applying the geometric transformation $T(u, v)$. In general, the target position (x', y') does not coincide with any discrete raster point. The source pixel value $I(u, v)$ is subsequently transferred to one of the adjacent target pixels.

of all these complications, source-to-target mapping is not really the method of choice.

10.2.2 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. 10.3.

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 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. 10.1, *target-to-source* mapping is the common method for geometrically transforming 2D images.

10.3 Interpolation

Interpolation is the process of estimating the intermediate values of a sampled function or signal at continuous positions or the attempt to reconstruct

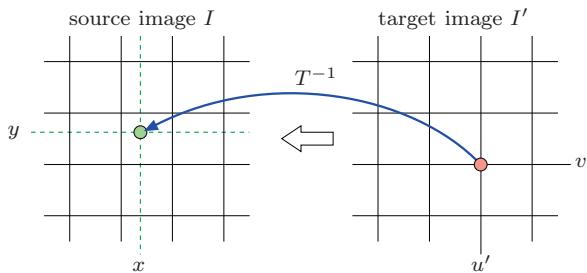


Figure 10.10 Target-to-source mapping. For each discrete pixel position (u', v') in the target image I' , the corresponding continuous source position (x, y) is found by applying the inverse mapping function $T^{-1}(u', v')$. The new pixel value $I'(u', v')$ is determined by interpolating the pixel values in the source image within some neighborhood of (x, y) .

Algorithm 10.1 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
    $T$ : continuous coordinate transform function ( $\mathbb{R}^2 \rightarrow \mathbb{R}^2$ )
   Returns the transformed image.
2: Create the target image  $I'$ .
3: for all target image coordinates  $(u', v')$  do
4:   Let  $(x, y) \leftarrow T^{-1}(u', v')$ 
5:    $I'(u', v') \leftarrow \text{GETINTERPOLATEDVALUE}(I, x, y)$ 
6: return  $I'$ .

```

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.

10.3.1 Simple Interpolation Methods

To illustrate the problem, we first attend to the one-dimensional case (Fig. 10.11). Several simple methods exist for interpolating the values of a

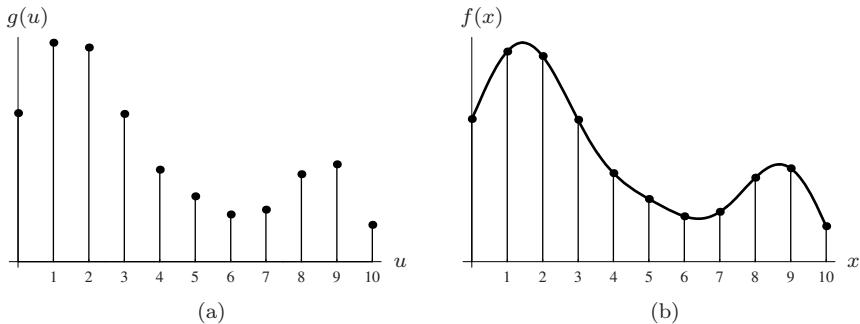


Figure 10.11 Interpolating a discrete function in 1D. Given the discrete function values $g(u)$ (a), the goal is to estimate the original function $f(x)$ at arbitrary continuous positions $x \in \mathbb{R}$ (b).

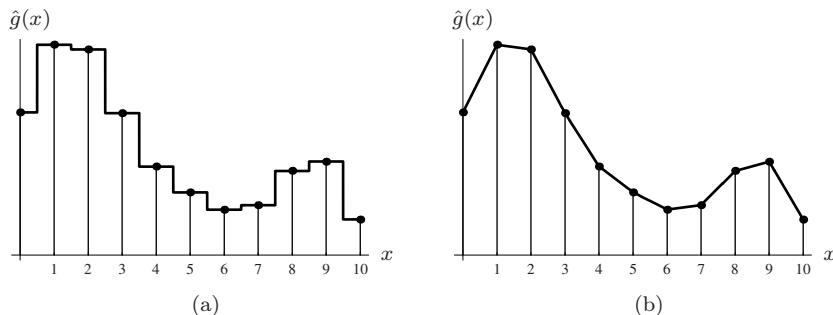


Figure 10.12 Simple interpolation methods. The *nearest-neighbor interpolation* (a) simply selects the discrete sample $g(u)$ closest to the given continuous coordinate x as the interpolating value $\hat{g}(x)$. Under *linear interpolation* (b), the result is a piecewise linear function connecting adjacent samples $g(u)$ and $g(u + 1)$.

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 $\hat{g}(x)$,

$$\hat{g}(x) = g(u_0), \quad (10.48)$$

$$\text{where } u_0 = \text{round}(x) = \lfloor x + 0.5 \rfloor. \quad (10.49)$$

A typical result of this so-called *nearest-neighbor interpolation* is shown in Fig. 10.12 (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 ,

$$\begin{aligned}\hat{g}(x) &= g(u_0) + (x - u_0) \cdot (g(u_0 + 1) - g(u_0)) \\ &= g(u_0) \cdot (1 - (x - u_0)) + g(u_0 + 1) \cdot (x - u_0).\end{aligned}\quad (10.50)$$

As shown in Fig. 10.12 (b), the result is a piecewise linear function made up of straight line segments between consecutive sample values.

10.3.2 Ideal Interpolation

Obviously the results of these simple interpolation methods do not well approximate the original continuous function (Fig. 10.11). 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. 7.2.1), 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 $\Pi_\pi(\omega)$ of width $\pm\omega_{\max} = \pm\pi$,

$$\hat{G}(\omega) = G(\omega) \cdot \Pi_\pi(\omega) = G(\omega) \cdot \begin{cases} 1 & \text{for } -\pi \leq \omega \leq \pi \\ 0 & \text{otherwise.} \end{cases}$$

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. (10.51) corresponds (see

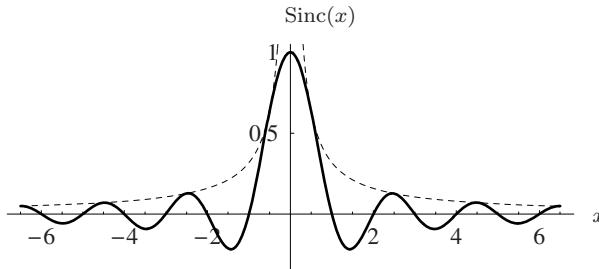


Figure 10.13 Sinc function in 1D. The function $\text{Sinc}(x)$ has the value 1 at the origin and zero values at all integer positions. The dashed line plots the amplitude $|\frac{1}{\pi x}|$ of the underlying sine function.

Eqn. (7.28)) to a *linear convolution* with the inverse Fourier transform of the windowing function $\Pi_\pi(\omega)$, which is the *Sinc* function, defined as

$$\text{Sinc}(x) = \begin{cases} 1 & \text{for } |x| = 0 \\ \frac{\sin(\pi x)}{\pi x} & \text{for } |x| > 0 \end{cases} \quad (10.51)$$

and shown in Fig. 10.13 (see also Table 7.1). This correspondence, which was already discussed in Sec. 7.1.6, between convolution in the signal domain and simple multiplication in the frequency domain is summarized in Fig. 10.14.

So theoretically 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 Sinc(x) are *convolved*. The reconstructed value of the continuous function at position x_0 is thus

$$\hat{g}(x_0) = [\text{Sinc} * g](x_0) = \sum_{u=-\infty}^{\infty} \text{Sinc}(x_0 - u) \cdot g(u), \quad (10.52)$$

where $*$ is the linear convolution operator (see Vol. 1 [14, Sec. 5.3.1]). 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. (10.52) modifies to

$$\hat{g}(x_0) = \sum_{u=-\infty}^{\infty} \text{Sinc}(x_0 - u) \cdot g(u \bmod N). \quad (10.53)$$

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

⁴ This assumption is explained by the fact that a discrete Fourier spectrum implicitly corresponds to a periodic signal (also see Sec. 7.2.2).

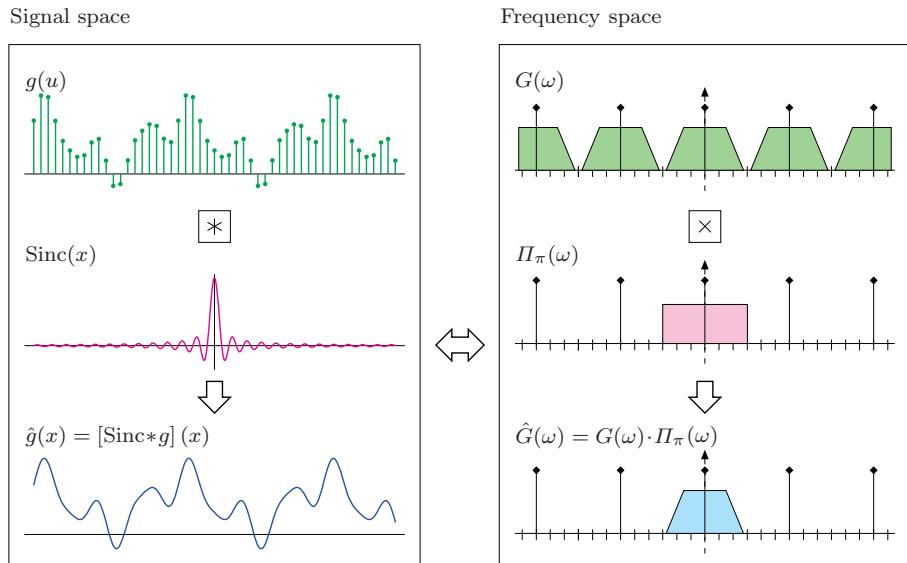


Figure 10.14 Interpolation of a discrete signal—relation between signal and frequency space. The discrete signal $g(u)$ in signal space (left) corresponds to the periodic Fourier spectrum $G(\omega)$ in frequency space (right). The spectrum $\hat{G}(\omega)$ of the continuous signal is isolated from $G(\omega)$ 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 $\text{Sinc}(x)$.

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 $|\frac{1}{\pi(x_0-u)}|$).

Figure 10.15 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. 10.16 (a) demonstrates. Problems occur, however, around local high-frequency signal events, such as rapid transitions or pulses, as shown in Fig. 10.16 (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).

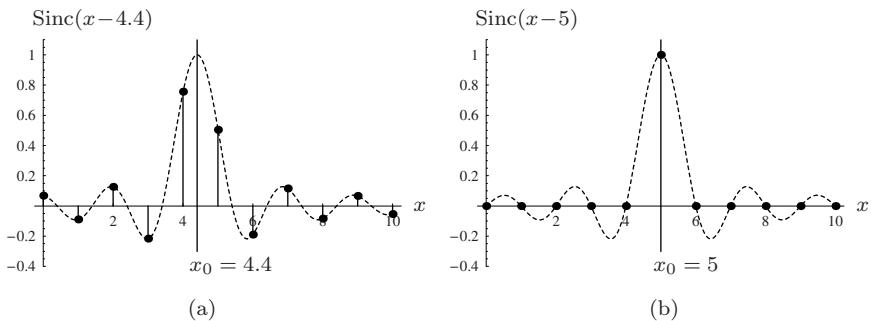


Figure 10.15 Interpolation by convolving with the Sinc function. The Sinc function is shifted by aligning its origin with the interpolation points $x_0 = 4.4$ (a) and $x_0 = 5$ (b). The values of the shifted Sinc function (dashed curve) at the integral positions are the weights (coefficients) for the corresponding sample values $g(u)$.

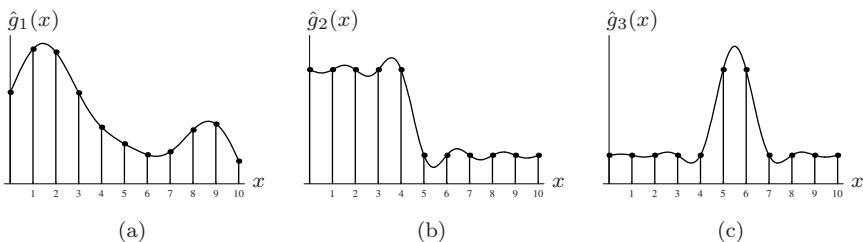


Figure 10.16 Sinc interpolation on various signal types. The reconstructed function in (a) is identical to the continuous, band-limited original. The results for the step function (b) and the pulse function (c) show the strong ringing caused by Sinc (ideal low-pass) interpolation.

A useful interpolation function implements a low-pass filter that on the one hand introduces minimal blurring by maintaining the maximum 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. 10.12) 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.

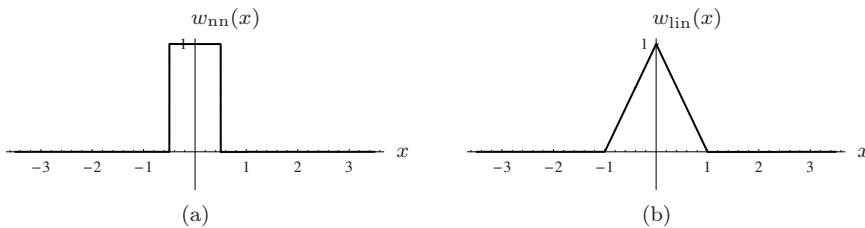


Figure 10.17 Convolution kernels for the nearest-neighbor interpolation $w_{nn}(x)$ and the linear interpolation $w_{lin}(x)$.

10.3.3 Interpolation by Convolution

As we saw earlier in the context of Sinc interpolation (Eqn. (10.51)), 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

$$\hat{g}(x_0) = [w * g](x_0) = \sum_{u=-\infty}^{\infty} w(x_0 - u) \cdot g(u). \quad (10.54)$$

The Sinc interpolation in Eqn. (10.52) is obviously only a special case with $w(x) = \text{Sinc}(x)$. Similarly, the one-dimensional *nearest-neighbor interpolation* (Eqn. (10.49), Fig. 10.12 (a)) can be expressed as a linear convolution with the kernel

$$w_{nn}(x) = \begin{cases} 1 & \text{for } -0.5 \leq x < 0.5 \\ 0 & \text{otherwise} \end{cases} \quad (10.55)$$

and the *linear interpolation* (Eqn. (10.50), Fig. 10.12 (b)) with the kernel

$$w_{lin}(x) = \begin{cases} 1 - |x| & \text{for } |x| < 1 \\ 0 & \text{for } |x| \geq 1. \end{cases} \quad (10.56)$$

The interpolation kernels $w_{nn}(x)$ and $w_{lin}(x)$ are both shown in Fig. 10.17, and sample results for various function types are plotted in Fig. 10.18.

10.3.4 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 version of the Sinc function or an approximation of it, thereby making the convolution

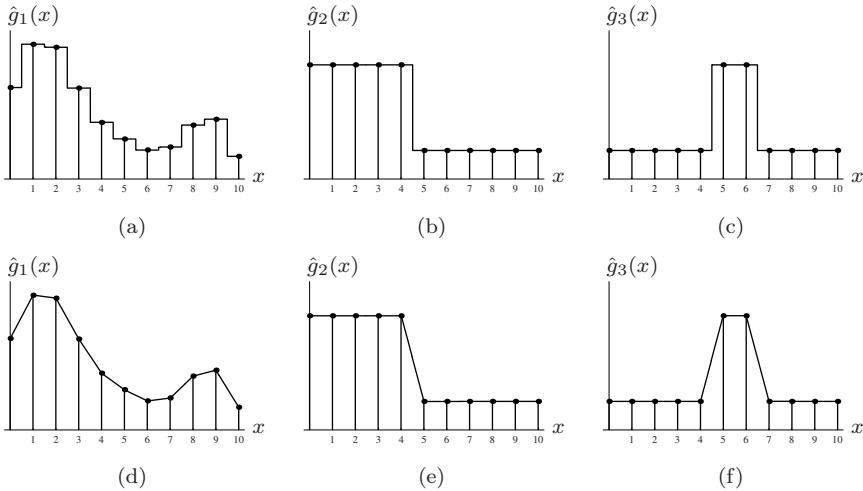


Figure 10.18 Interpolation examples: nearest-neighbor interpolation (a–c), linear interpolation (d–f).

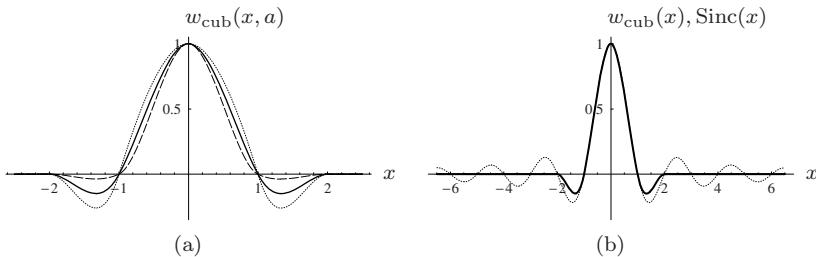


Figure 10.19 Cubic interpolation kernel. Function $w_{\text{cub}}(x, a)$ with control parameter a set to $a = 0.25$ (dashed curve), $a = 1$ (continuous curve), and $a = 1.75$ (dotted curve) (a). Cubic function $w_{\text{cub}}(x)$ and Sinc function compared (b).

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

$$w_{\text{cub}}(x, a) = \begin{cases} (-a + 2) \cdot |x|^3 + (a - 3) \cdot |x|^2 + 1 & \text{for } 0 \leq |x| < 1 \\ -a \cdot |x|^3 + 5a \cdot |x|^2 - 8a \cdot |x| + 4a & \text{for } 1 \leq |x| < 2 \\ 0 & \text{for } |x| \geq 2. \end{cases} \quad (10.57)$$

The single control parameter a can be used to adjust the slope of this spline⁵ function (Fig. 10.19), which affects the amount of overshoot and thus the per-

⁵ The family of functions described by Eqn. (10.57) are commonly referred to as *cardinal splines* [6] (see also Sec. 10.3.5).

ceived “sharpness” of the interpolated signal. For $a = 1$, which is often recommended as a standard setting, Eqn. (10.57) simplifies to

$$w_{\text{cub}}(x) = \begin{cases} |x|^3 - 2 \cdot |x|^2 + 1 & \text{for } 0 \leq |x| < 1 \\ -|x|^3 + 5 \cdot |x|^2 - 8 \cdot |x| + 4 & \text{for } 1 \leq |x| < 2 \\ 0 & \text{for } |x| \geq 2. \end{cases} \quad (10.58)$$

Figure 10.20 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. 10.20 (d)). With $a = 0.5$, the expression in Eqn. (10.57) corresponds to a *Catmull-Rom spline* [16] (see also Sec. 10.3.5), which produces significantly better results than the standard setup (with $a = 1$), particularly in smooth signal regions (see Fig. 10.22 (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. 10.19 (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. (10.54)) 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 the expression

$$\hat{g}(x_0) = \sum_{u=\lfloor x_0 \rfloor - 1}^{\lfloor x_0 \rfloor + 2} w_{\text{cub}}(x_0 - u, a) \cdot g(u). \quad (10.59)$$

10.3.5 Spline Interpolation

The cubic interpolation kernel (Eqn. (10.57)) described in the previous section is a piecewise cubic polynomial function, also known as a *cubic spline* in computer graphics. In its general form, this function takes not only one but *two*

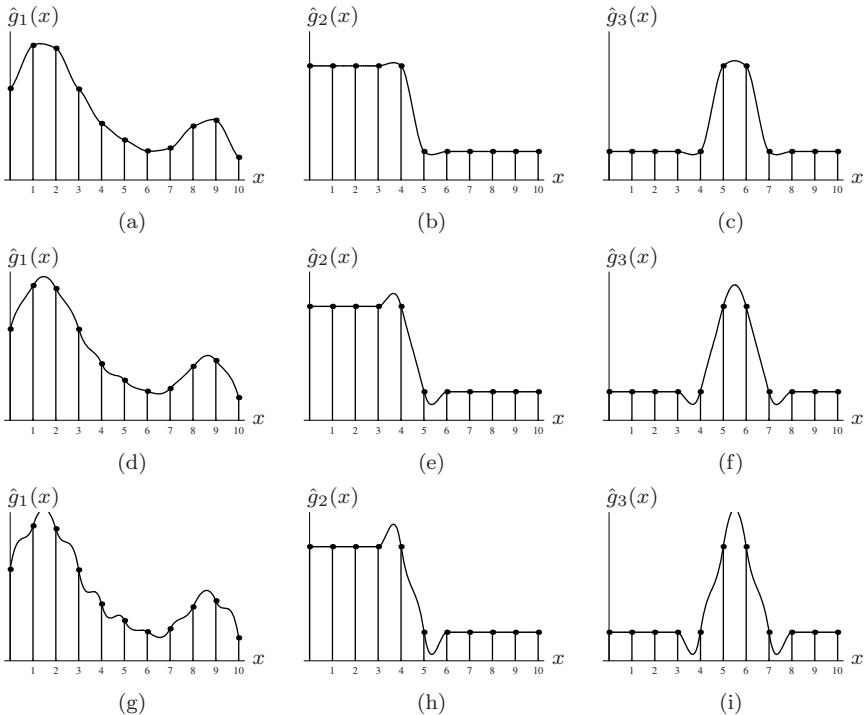


Figure 10.20 Cubic interpolation examples. Parameter a in Eqn. (10.57) controls the amount of signal overshoot or perceived sharpness: $a = 0.25$ (a–c), standard setting $a = 1$ (d–f), $a = 1.75$ (g–i). Notice in (d) the ripple effects incurred by interpolating with the standard settings in smooth signal regions.

control parameters (a, b) [54],⁶

$$w_{\text{cs}}(x, a, b) =$$

$$\frac{1}{6} \cdot \begin{cases} (-6a - 9b + 12) \cdot |x|^3 \\ \quad + (6a + 12b - 18) \cdot |x|^2 - 2b + 6 & \text{for } 0 \leq |x| < 1 \\ (-6a - b) \cdot |x|^3 + (30a + 6b) \cdot |x|^2 \\ \quad + (-48a - 12b) \cdot |x| + 24a + 8b & \text{for } 1 \leq |x| < 2 \\ 0 & \text{for } |x| \geq 2. \end{cases} \quad (10.60)$$

Equation (10.60) 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 $w_{\text{cs}}(x, a, b)$ specifies a one-parameter family of so-called *cardinal*

⁶ In [54], the parameters a and b were originally named C and B , respectively, with $B \equiv b$ and $C \equiv a$.

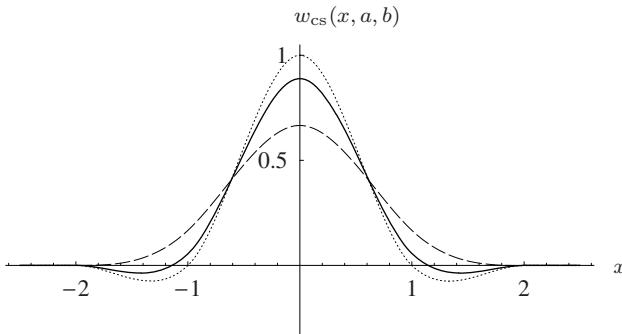


Figure 10.21 Examples of cardinal spline functions $w_{cs}(x, a, b)$ as specified by Eqn. (10.60): *Catmull-Rom* spline $w_{cs}(x, 0.5, 0)$ (dotted line), *cubic B-spline* $w_{cs}(x, 0, 1)$ (dashed line), and *Mitchell-Netravali* function $w_{cs}(x, \frac{1}{3}, \frac{1}{3})$ (solid line).

splines equivalent to the cubic interpolation function $w_{cub}(x, a)$ in Eqn. (10.57),

$$w_{cs}(x, a, 0) = w_{cub}(x, a),$$

and for the standard setting $a = 1$ (Eqn. (10.58)) in particular

$$w_{cs}(x, 1, 0) = w_{cub}(x, 1) = w_{cub}(x).$$

Figure 10.21 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. (10.59).

Catmull-Rom interpolation

With the control parameters set to $a = 0.5$ and $b = 0$, the function in Eqn. (10.60) is a *Catmull-Rom spline* [16], as already mentioned in Sec. 10.3.4:

$$w_{crm}(x) = w_{cs}(x, 0.5, 0) = \frac{1}{2} \cdot \begin{cases} 3 \cdot |x|^3 - 5 \cdot |x|^2 + 2 & \text{for } 0 \leq |x| < 1 \\ -|x|^3 + 5 \cdot |x|^2 - 8 \cdot |x| + 4 & \text{for } 1 \leq |x| < 2 \\ 0 & \text{for } |x| \geq 2. \end{cases} \quad (10.61)$$

Examples of signals interpolated with this kernel are shown in Fig. 10.22 (a–c). The results are similar to ones produced by cubic interpolation (with $a = 1$, see Fig. 10.20) with regard to sharpness, but the Catmull-Rom reconstruction is clearly superior in smooth signal regions (compare, e.g., Fig. 10.20 (d) vs. Fig. 10.22 (a)).

Cubic B-spline approximation

With parameters set to $a = 0$ and $b = 1$, Eqn. (10.60) corresponds to a cubic B-spline function [6] of the form

$$w_{\text{cbs}}(x) = w_{\text{cs}}(x, 0, 1)$$

$$= \frac{1}{6} \cdot \begin{cases} 3 \cdot |x|^3 - 6 \cdot |x|^2 - 4 & \text{for } 0 \leq |x| < 1 \\ -|x|^3 + 6 \cdot |x|^2 - 12 \cdot |x| + 8 & \text{for } 1 \leq |x| < 2 \\ 0 & \text{for } |x| \geq 2. \end{cases} \quad (10.62)$$

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. 10.22 (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 [54] proposed a cubic interpolation kernel as described in Eqn. (10.60) with parameter settings $a = \frac{1}{3}$ and $b = \frac{1}{3}$, and the resulting interpolation function

$$w_{\text{mn}}(x) = w_{\text{cs}}(x, \frac{1}{3}, \frac{1}{3})$$

$$= \frac{1}{18} \cdot \begin{cases} 21 \cdot |x|^3 - 36 \cdot |x|^2 + 16 & \text{for } 0 \leq |x| < 1 \\ -7 \cdot |x|^3 + 36 \cdot |x|^2 - 60 \cdot |x| + 32 & \text{for } 1 \leq |x| < 2 \\ 0 & \text{for } |x| \geq 2. \end{cases} \quad (10.63)$$

This function is the weighted sum of a Catmull-Rom spline (Eqn. (10.61)) and a cubic B-spline (Eqn. (10.62)), as is apparent in Fig. 10.21.⁷ The examples in Fig. 10.22 (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*.

⁷ See also Exercise 10.5.

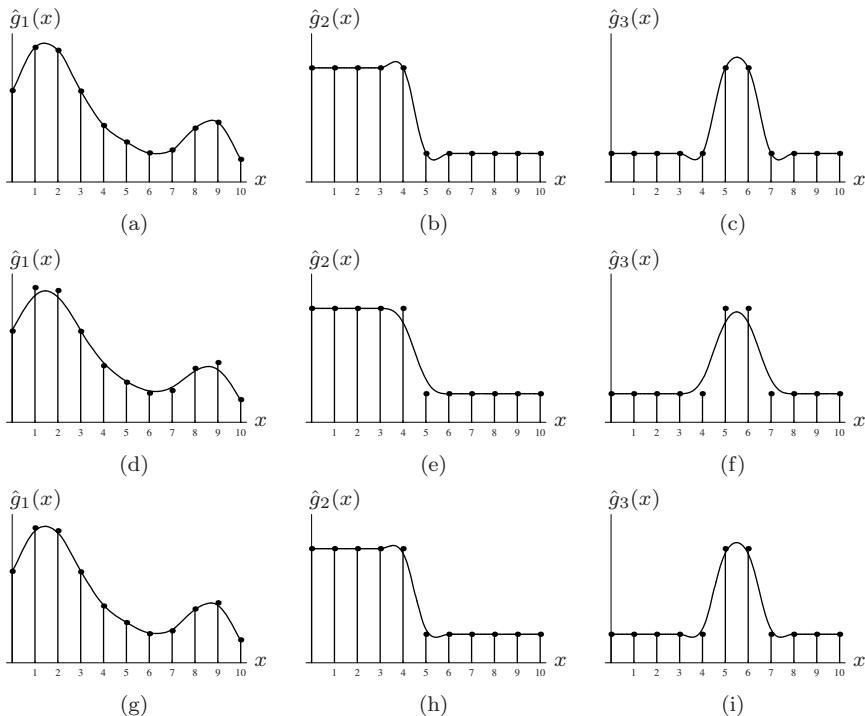


Figure 10.22 Cardinal spline reconstruction examples: *Catmull-Rom* interpolation (a–c), *cubic B-spline* approximation (d–f), and *Mitchell-Netravali* approximation (g–i).

10.3.6 Lanczos Interpolation

The Lanczos⁸ 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

$$w(x) = \psi(x) \cdot \text{Sinc}(x). \quad (10.64)$$

The particular window functions for the Lanczos interpolation are defined as

$$\psi_{Ln}(x) = \begin{cases} 1 & \text{for } |x| = 0 \\ \frac{\sin(\pi \frac{x}{n})}{\pi \frac{x}{n}} & \text{for } 0 < |x| < n \\ 0 & \text{for } |x| \geq n, \end{cases} \quad (10.65)$$

where $n \in \mathbb{N}$ denotes the *order* of the filter [56, 74]. Notice that the window function is again a truncated Sinc function! For the Lanczos filters of order $n =$

⁸ Cornelius Lanczos (1893–1974).

2, 3, which are the most commonly used in image processing, the corresponding window functions are

$$\psi_{L2}(x) = \begin{cases} 1 & \text{for } |x| = 0 \\ \frac{\sin(\pi \frac{x}{2})}{\pi \frac{x}{2}} & \text{for } 0 < |x| < 2 \\ 0 & \text{for } |x| \geq 2, \end{cases} \quad (10.66)$$

$$\psi_{L3}(x) = \begin{cases} 1 & \text{for } |x| = 0 \\ \frac{\sin(\pi \frac{x}{3})}{\pi \frac{x}{3}} & \text{for } 0 < |x| < 3 \\ 0 & \text{for } |x| \geq 3. \end{cases} \quad (10.67)$$

Both window functions are shown in Fig. 10.23 (a, b). From Eqns. (10.64) and (10.65), the general one-dimensional Lanczos interpolation kernel of order $n \geq 1$ is then defined as

$$\begin{aligned} w_{Ln}(x) &= \psi_{Ln}(x) \cdot \text{Sinc}(x) \\ &= \begin{cases} 1 & \text{for } |x| = 0 \\ n \cdot \frac{\sin(\pi \frac{x}{n}) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 < |x| < n \\ 0 & \text{for } |x| \geq n, \end{cases} \end{aligned} \quad (10.68)$$

and thus the 1D Lanczos kernels of orders $n = 2$ and $n = 3$ are

$$w_{L2}(x) = \begin{cases} 1 & \text{for } |x| = 0 \\ 2 \cdot \frac{\sin(\pi \frac{x}{2}) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 < |x| < 2 \\ 0 & \text{for } |x| \geq 2 \end{cases} \quad (10.69)$$

and

$$w_{L3}(x) = \begin{cases} 1 & \text{for } |x| = 0 \\ 3 \cdot \frac{\sin(\pi \frac{x}{3}) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 < |x| < 3 \\ 0 & \text{for } |x| \geq 3. \end{cases} \quad (10.70)$$

Figure 10.23 (c, d) shows the resulting interpolation kernels together with the original Sinc function. The function $w_{L2}(x)$ is quite similar to the Catmull-Rom kernel $w_{\text{crm}}(x)$ (Eqn. (10.61), Fig. 10.21), so the results can be expected to be similar as well, as shown in Fig. 10.24 (a–c) (cf. Fig. 10.22 (a–c)). Notice, however, the relatively poor reconstruction in the smooth signal regions (Fig. 10.24 (a)) and the strong ringing introduced in the constant high-amplitude regions (Fig. 10.24 (b)). The “3-tap” kernel $w_{L3}(x)$ reduces these artifacts and produces steeper edges, at the cost of increased overshoot (Fig. 10.22 (d–f)).

In summary, although Lanczos interpolators 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

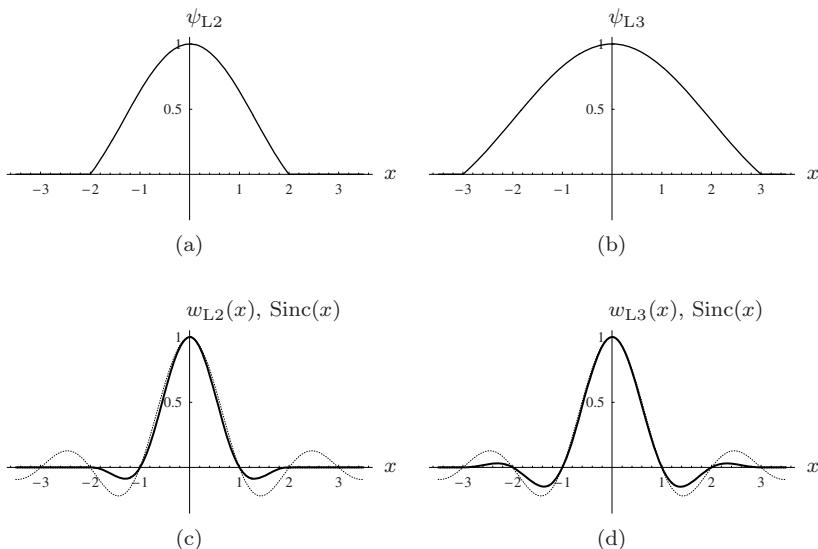


Figure 10.23 One-dimensional Lanczos interpolation kernels. Lanczos window functions ψ_{L2} (a), ψ_{L3} (b), and the corresponding interpolation kernels w_{L2} (c), w_{L3} (d). The original Sinc function (dotted curve) is shown for comparison.

Mitchell-Netravali interpolations. While these are based on efficiently computable polynomial functions, Lanczos interpolation requires trigonometric functions which are relatively costly to compute, unless some form of tabulation is used.

10.3.7 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

$$\text{SINC}(x, y) = \text{Sinc}(x) \cdot \text{Sinc}(y) = \frac{\sin(\pi x)}{\pi x} \cdot \frac{\sin(\pi y)}{\pi y}, \quad (10.71)$$

which is shown in Fig. 10.25 (a). Just as in 1D, the 2D Sinc function is not a practical interpolation function for various reasons. In the following, we 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.

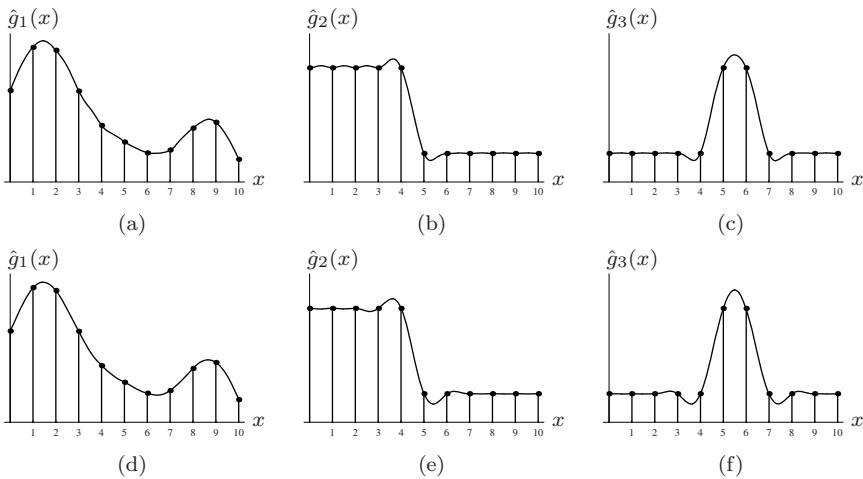


Figure 10.24 Lanczos interpolation examples: Lanczos-2 (a–c), Lanczos-3 (d–f). Note the ringing in the flat (constant) regions caused by Lanczos-2 interpolation in the left part of (b). The Lanczos-3 interpolator shows less ringing (e) but produces steeper edges at the cost of increased overshoot (e, f).

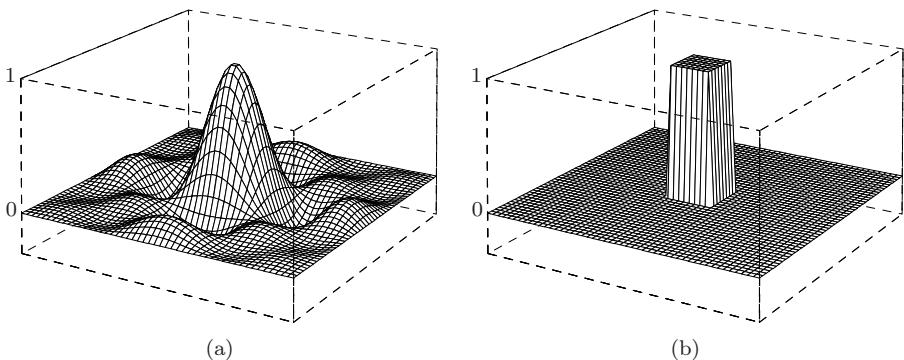


Figure 10.25 Interpolation kernels in 2D: Sinc kernel $SINC(x, y)$ (a) and nearest-neighbor kernel $W_{nn}(x, y)$ (b) for $-3 \leq x, y \leq 3$.

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,

$$\hat{I}(x_0, y_0) = I(u_0, v_0),$$

$$\text{with } u_0 = \text{round}(x_0) = \lfloor x_0 + 0.5 \rfloor, \quad (10.72)$$

$$v_0 = \text{round}(y_0) = \lfloor y_0 + 0.5 \rfloor.$$

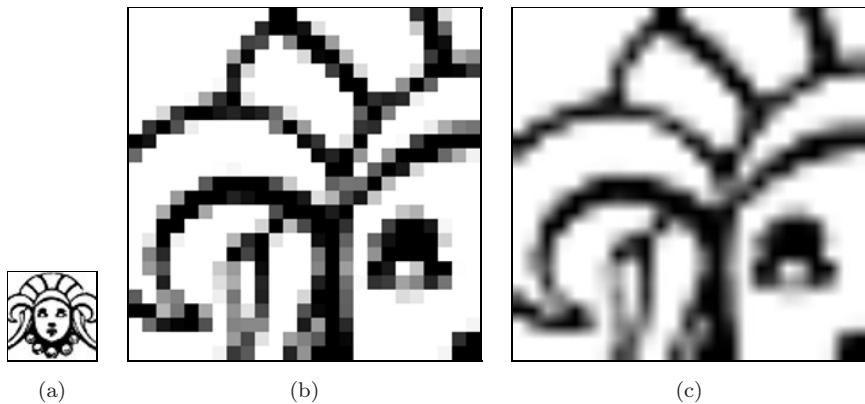


Figure 10.26 Image enlargement ($8\times$): original (a), nearest-neighbor interpolation (b), and bilinear interpolation (c).

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. (10.55), defined as

$$W_{\text{nn}}(x, y) = \begin{cases} 1 & \text{for } -0.5 \leq x, y < 0.5 \\ 0 & \text{otherwise.} \end{cases} \quad (10.73)$$

This function is shown in Fig. 10.25 (b). Nearest-neighbor interpolation is known for its strong blocking effects (Fig. 10.26 (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. 10.3.1) is the so-called *bilinear* interpolation,⁹ whose operation is illustrated in Fig. 10.27. 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

$$\begin{aligned} A &= I(u_0, v_0), & B &= I(u_0+1, v_0), \\ C &= I(u_0, v_0+1), & D &= I(u_0+1, v_0+1), \end{aligned} \quad (10.74)$$

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

⁹ Not to be confused with the bilinear *mapping* (transformation) described in Sec. 10.1.5.

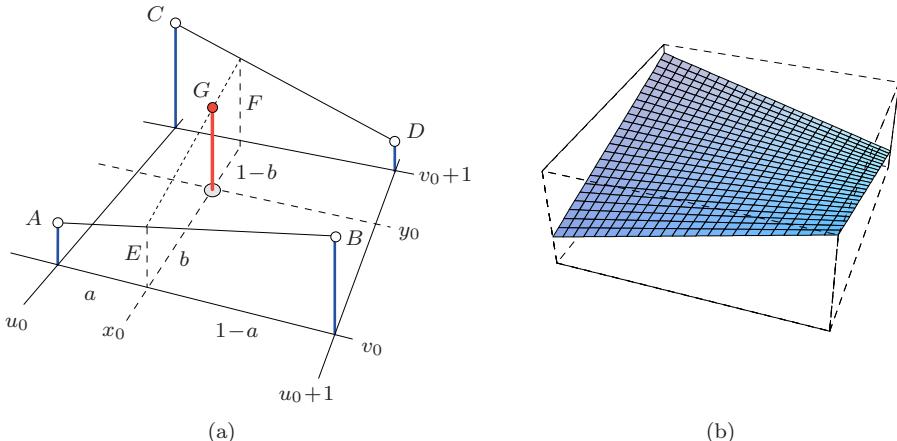


Figure 10.27 Bilinear interpolation. For a given position (x_0, y_0) , the interpolated value is computed from the values A, B, C, D of the four closest pixels in two steps (a). First the intermediate values E and F are computed by linear interpolation in the horizontal direction between A, B and C, 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).

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

$$\begin{aligned} \hat{I}(x_0, y_0) &= G = E + (y_0 - v_0) \cdot (F - E) = E + b \cdot (F - E) \\ &= (a-1)(b-1)A + a(1-b)B + (1-a)bC + abD. \end{aligned} \quad (10.75)$$

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 $w_{\text{lin}}(y)$ (Eqn. (10.56)),

$$\begin{aligned} W_{\text{bil}}(x, y) &= w_{\text{lin}}(x) \cdot w_{\text{lin}}(y) \\ &= \begin{cases} 1 - x - y - x \cdot y & \text{for } 0 \leq |x|, |y| < 1 \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \quad (10.76)$$

In this function (plotted in Fig. 10.28 (a)), we can recognize the bilinear term that gives this method its name.

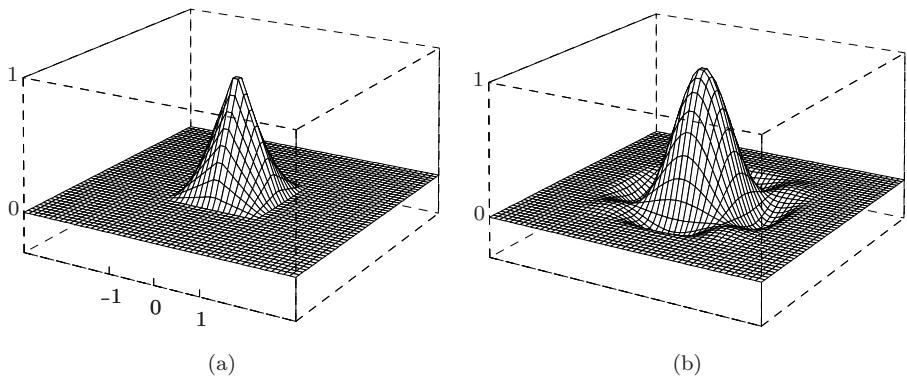


Figure 10.28 2D interpolation kernels: bilinear kernel $W_{\text{bil}}(x, y)$ (a) and bicubic kernel $W_{\text{bic}}(x, y)$ (b) for $-3 \leq x, y \leq 3$.

Bicubic and spline interpolation

The convolution kernel for the two-dimensional cubic interpolation is also defined as the product of the corresponding one-dimensional kernels (Eqn. (10.58)),

$$W_{\text{bic}}(x, y) = w_{\text{cub}}(x) \cdot w_{\text{cub}}(y). \quad (10.77)$$

The resulting kernel is plotted in Fig. 10.28 (b). Due to the decomposition into one-dimensional kernels (Eqn. (10.77)), the computation of the bicubic interpolation is *separable* in x, y and can thus be expressed as

$$\begin{aligned} \hat{I}(x_0, y_0) &= \sum_{\substack{v=\lfloor y_0 \rfloor - 1 \\ \lfloor y_0 \rfloor + 2}}^{\lfloor y_0 \rfloor + 2} \left[\sum_{\substack{u=\lfloor x_0 \rfloor - 1 \\ \lfloor x_0 \rfloor - 1}}^{\lfloor x_0 \rfloor + 2} [I(u, v) \cdot W_{\text{bic}}(x_0 - u, y_0 - v)] \right] \\ &= \sum_{j=0}^3 \left[w_{\text{cub}}(y_0 - v_j) \cdot \underbrace{\sum_{i=0}^3 [I(u_i, v_j) \cdot w_{\text{cub}}(x_0 - u_i)]}_{p_j} \right], \end{aligned} \quad (10.78)$$

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. 10.29. Equation (10.78) 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.

This method, which is summarized in Alg. 10.2, 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. (10.61)) with

$$W_{\text{crm}}(x, y) = w_{\text{crm}}(x) \cdot w_{\text{crm}}(y) \quad (10.79)$$

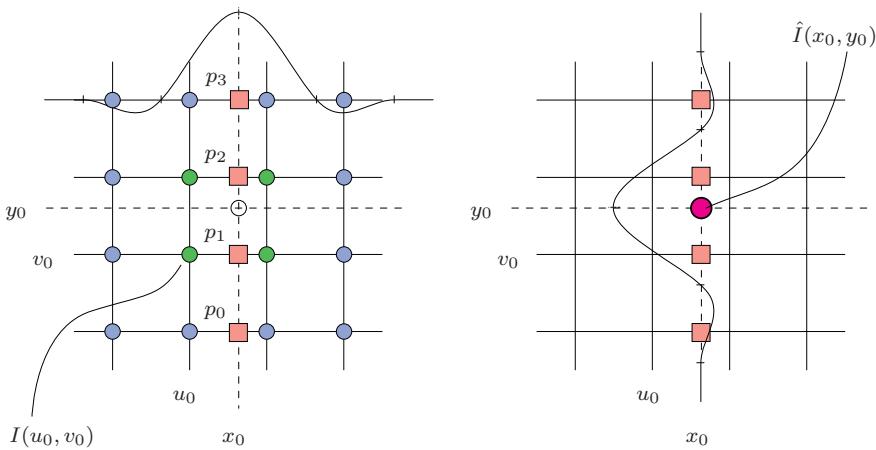


Figure 10.29 Bicubic interpolation in two steps. The discrete image I (pixels are marked \circ) is to be interpolated at some continuous position (x_0, y_0) . In step 1 (left), a one-dimensional interpolation is performed in the horizontal direction with $w_{\text{cub}}(x)$ over four pixels $I(u_i, v_j)$ in four lines. One intermediate result p_j (marked \square) 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 $p_0 \dots p_3$.

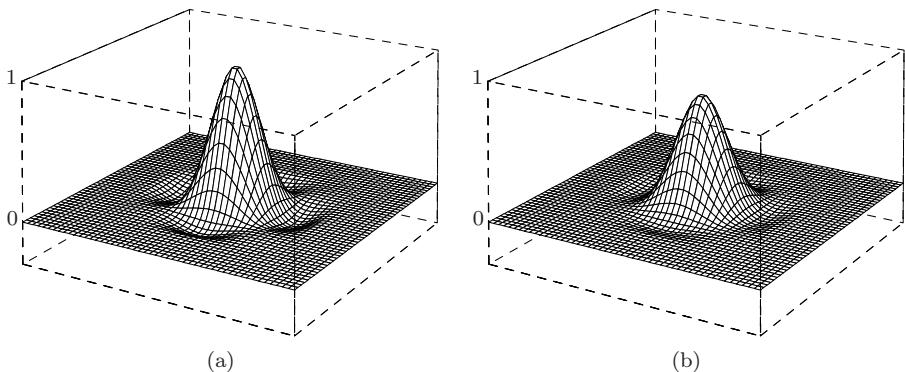


Figure 10.30 Two-dimensional spline interpolation kernels: Catmull-Rom kernel $W_{\text{crm}}(x, y)$ (a), Mitchell-Netravali kernel $W_{\text{mn}}(x, y)$ (b), for $-3 \leq x, y \leq 3$.

or the *Mitchell-Netravali* interpolation (Eqn. (10.63)) with

$$W_{\text{mn}}(x, y) = w_{\text{mn}}(x) \cdot w_{\text{mn}}(y). \quad (10.80)$$

The corresponding 2D kernels are shown in Fig. 10.30. For interpolation with separable kernels of larger size see the general procedure in Alg. 10.3.

Algorithm 10.2 Bicubic interpolation of image I at position (x_0, y_0) . The one-dimensional cubic function $w_{\text{cub}}(\cdot)$ (Eqn. (10.57)) is used for the separate interpolation in the x and y directions based on a neighborhood of 4×4 pixels.

```

1: BICUBICINTERPOLATION ( $I, x_0, y_0$ ) ▷  $(x_0, y_0) \in \mathbb{R}^2$ 
   Returns the interpolated value of the image  $I$  at the continuous position  $(x_0, y_0)$ .
2: Let  $q \leftarrow 0$ 
3: for  $j \leftarrow 0 \dots 3$  do ▷ iterate over 4 lines
4:   Let  $v \leftarrow \lfloor y_0 \rfloor + j - 1$ 
5:   Let  $p \leftarrow 0$ 
6:   for  $i \leftarrow 0 \dots 3$  do ▷ iterate over 4 columns
7:     Let  $u \leftarrow \lfloor x_0 \rfloor + i - 1$ 
8:     Let  $p \leftarrow p + I(u, v) \cdot w_{\text{cub}}(x_0 - u)$ 
9:    $q \leftarrow q + p \cdot w_{\text{cub}}(y_0 - v)$ 
10:  return  $q$ .
```

Lanczos interpolation

The kernels for the 2D Lanczos interpolation are also x/y -separable into one-dimensional kernels (Eqns. (10.69) and (10.70), respectively),

$$W_{\text{Ln}}(x, y) = w_{\text{Ln}}(x) \cdot w_{\text{Ln}}(y). \quad (10.81)$$

The resulting kernels for orders $n = 2$ and $n = 3$ are shown in Fig. 10.31. Because of the separability the 2D Lanczos interpolation can be computed, similar to the bicubic interpolation, separately in the x and y directions. Like the bicubic kernel, the 2-tap Lanczos kernel W_{L2} (Eqn. (10.69)) is zero outside the interval $-2 \leq x, y \leq 2$, and thus the procedure described in Eqn. (10.78) and Alg. 10.2 can be used with only a small modification (replace w_{cub} by w_{L2}).

The 3-tap Lanczos kernel W_{L3} (Eqn. (10.70)) requires two additional rows and columns, and therefore the 2D interpolation changes to

$$\begin{aligned} \hat{I}(x_0, y_0) &= \sum_{\substack{v=\\ \lfloor y_0 \rfloor - 2}}^{\lfloor y_0 \rfloor + 3} \left[\sum_{\substack{u=\\ \lfloor x_0 \rfloor - 2}}^{\lfloor x_0 \rfloor + 3} [I(u, v) \cdot W_{\text{L3}}(x_0 - u, y_0 - v)] \right] \\ &= \sum_{j=0}^5 [w_{\text{L3}}(y_0 - v_j) \cdot \sum_{i=0}^5 [I(u_i, v_j) \cdot w_{\text{L3}}(x_0 - u_i)]], \end{aligned} \quad (10.82)$$

$$\text{with } u_i = \lfloor x_0 \rfloor + i - 2 \quad \text{and} \quad v_j = \lfloor y_0 \rfloor + j - 2.$$

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.

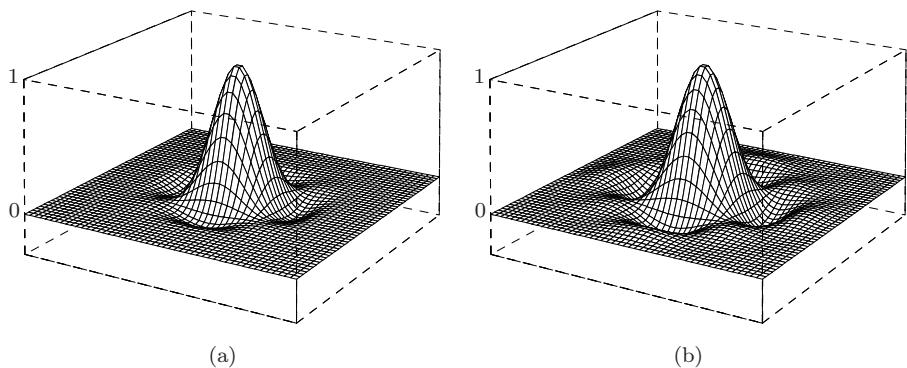


Figure 10.31 Two-dimensional Lanczos kernels for $n = 2$ and $n = 3$: kernels $W_{L2}(x, y)$ (a) and $W_{L3}(x, y)$ (b), with $-3 \leq x, y \leq 3$.

In general, the expression for a 2D Lanczos interpolator L_n of arbitrary order $n \geq 1$ is

$$\begin{aligned} \hat{I}(x_0, y_0) &= \sum_{v=\lfloor y_0 \rfloor - n + 1}^{\lfloor y_0 \rfloor + n} \left[\sum_{u=\lfloor x_0 \rfloor - n + 1}^{\lfloor x_0 \rfloor + n} [I(u, v) \cdot W_{Ln}(x_0 - u, y_0 - v)] \right] \\ &= \sum_{j=0}^{2n-1} [w_{Ln}(y_0 - v_j) \cdot \sum_{i=0}^{2n-1} [I(u_i, v_j) \cdot w_{Ln}(x_0 - u_i)]], \end{aligned} \quad (10.83)$$

with $u_i = \lfloor x_0 \rfloor + i - n + 1$ and $v_j = \lfloor y_0 \rfloor + j - n + 1$.

The size of this interpolator's support region is $2n \times 2n$ pixels. How the expression in Eqn. (10.83) could be computed is shown in Alg. 10.3, which actually describes a general interpolation procedure that can be used with any separable interpolation kernel $W(x, y) = w_n(x) \cdot w_n(y)$ of extent $\pm n$.

Examples and discussion

Figures 10.32 and 10.33 compare the interpolation methods described above: nearest-neighbor, bilinear, bicubic Catmull-Rom, cubic B-spline, Mitchell-Netravali, and Lanczos interpolation. In both figures, the original images are rotated counter-clockwise by 15° . A gray background is used to visualize the edge overshoot produced by some of the interpolators.

Nearest-neighbor interpolation (Fig. 10.32 (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. 10.32 (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

Algorithm 10.3 General interpolation with a separable interpolation kernel $W(x, y) = w_n(x) \cdot w_n(y)$ of extent $\pm n$ (i.e., the 1D kernel $w_n(x)$ is zero for $x < -n$ and $x > n$, with $n \in \mathbb{N}$). Note that the BICUBICINTERPOLATION procedure in Alg. 10.2 is a special instance of this algorithm (with $n = 2$).

```

1: SEPARABLEINTERPOLATION ( $I, x_0, y_0, w_n$ )  $\triangleright (x_0, y_0) \in \mathbb{R}^2$ 
   Returns the interpolated value of the image  $I$  at the continuous position  $(x_0, y_0)$ , using the interpolation kernel  $W(x, y) = w_n(x) \cdot w_n(y)$ .
2: Let  $n$  be the extent of the kernel  $w_n$  ( $n \geq 1$ )
3: Let  $q \leftarrow 0$ 
4: for  $j \leftarrow 0 \dots 2n-1$  do  $\triangleright$  iterate over  $2n$  lines
5:   Let  $v \leftarrow v_j = \lfloor y_0 \rfloor + j - n + 1$ 
6:   Let  $p \leftarrow 0$ 
7:   for  $i \leftarrow 0 \dots 2n-1$  do  $\triangleright$  iterate over  $2n$  columns
8:     Let  $u \leftarrow u_i = \lfloor x_0 \rfloor + i - n + 1$ 
9:     Let  $p \leftarrow p + I(u, v) \cdot w_n(x_0 - u)$ 
10:     $q \leftarrow q + p \cdot w_n(y_0 - v)$ 
11: return  $q$ .
```

cannot create any over- or undershoot at edges. This is not the case for the *bicubic* interpolation (Fig. 10.32 (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. (10.57)), the bicubic kernel can be easily tuned to fit the need of particular applications. For example, the *Catmull-Rom* method (Fig. 10.32 (e)) can be implemented with the bicubic interpolation by setting $a = 0.5$ (Eqns. (10.61) and (10.79)).

Results from the 2D *Lanczos* interpolation (Fig. 10.32 (h)) using the 2-tap kernel W_{L2} cannot be much better than from the bicubic interpolation, which can be adjusted to give similar results without causing any ringing in flat regions, as visible in Fig. 10.24. The 3-tap Lanczos kernel W_{L3} (Fig. 10.32 (i)) on the other hand should produce slightly sharper edges at the cost of increased overshoot (see also Exercise 10.7).

In summary, for high-quality applications one should consider the *Catmull-Rom* (Eqns. (10.61) and (10.79)) or the *Mitchell-Netravali* (Eqns. (10.63) and (10.80)) methods, which offer good reconstruction at the same computational cost as the bicubic interpolation.

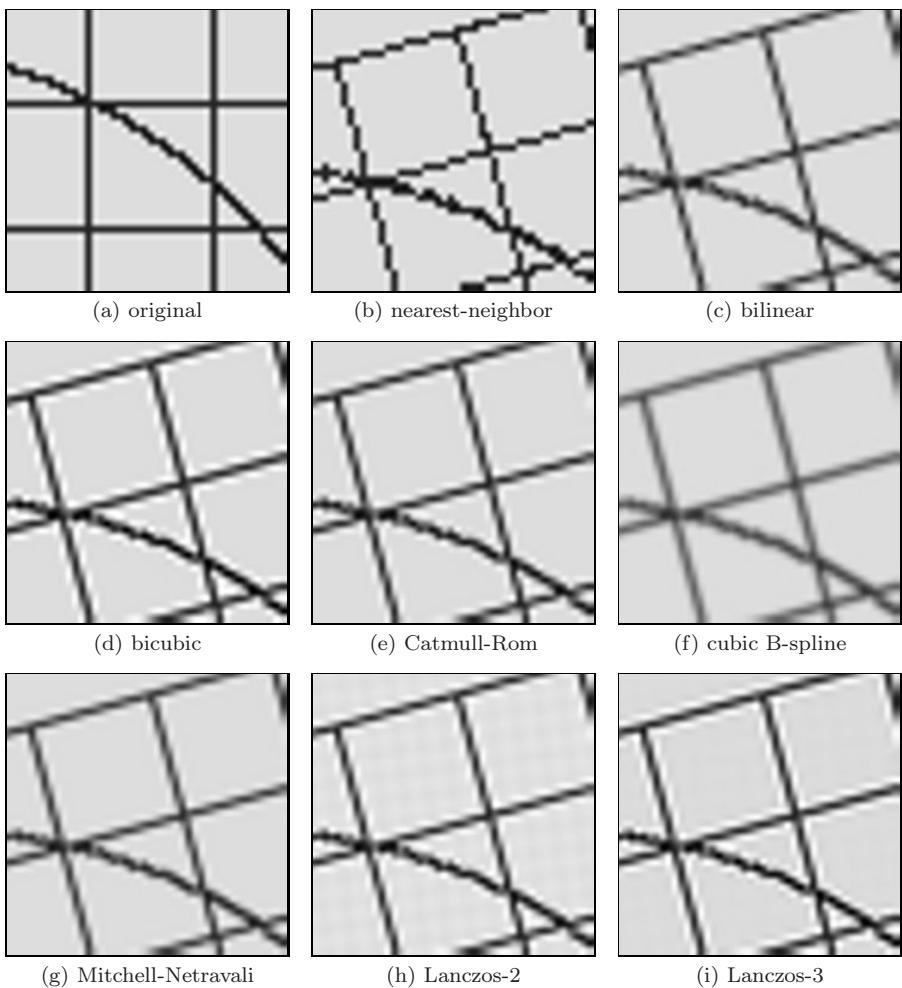


Figure 10.32 Image interpolation methods compared (line art): part of the original image (a), which is subsequently rotated by 15° . Nearest-neighbor (b), bilinear (c), bicubic (d), Catmull-Rom (e), cubic B-spline (f), Mitchell-Netravali (g), Lanczos-2 (h), Lanczos-3 (i) interpolation. The image was given a gray background to visualize overshooting, particularly noticeable with bicubic (d) and Lanczos-3 (i) interpolation. Notice the ringing in the flat image regions produced by the Lanczos-2 interpolation (h).

10.3.8 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. 10.34):

1. Each discrete image point (u'_0, v'_0) of the *target* image is projected by the



Figure 10.33 Image interpolation methods compared (text image): original image (a), which is subsequently rotated by 15° . Nearest-neighbor (b), bilinear (c), bicubic (d), Catmull-Rom (e), cubic B-spline (f), Mitchell-Netravali (g), Lanczos-2 (h), Lanczos-3 (i) interpolation.

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 $I'(u'_0, v'_0)$.

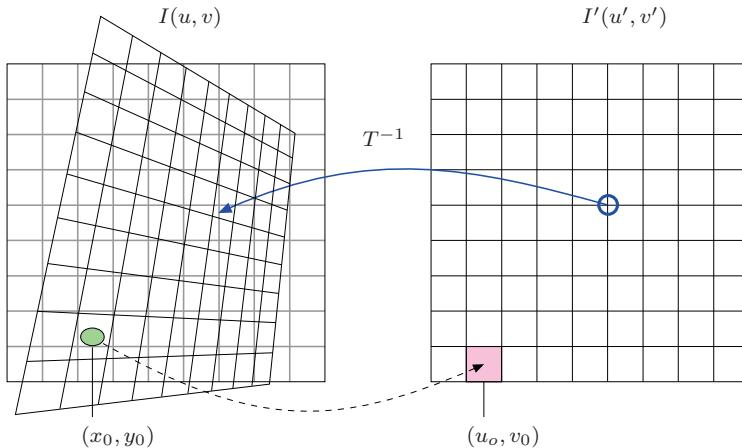


Figure 10.34 Sampling errors in geometric operations. If the geometric transformation T leads to a local contraction of the image (which corresponds to a local enlargement by T^{-1}), the distance between adjacent sample points in I is increased. This reduces the local sampling frequency and thus the maximum signal frequency allowed in the source image, which eventually leads to aliasing.

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 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. 10.35. The effect is most noticeable under nearest-neighbor interpolation in Fig. 10.35 (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. 10.35 (c, d) have wider interpolation kernels but still cannot avoid the aliasing effect. The problem of course gets worse with increasing reduction factors.

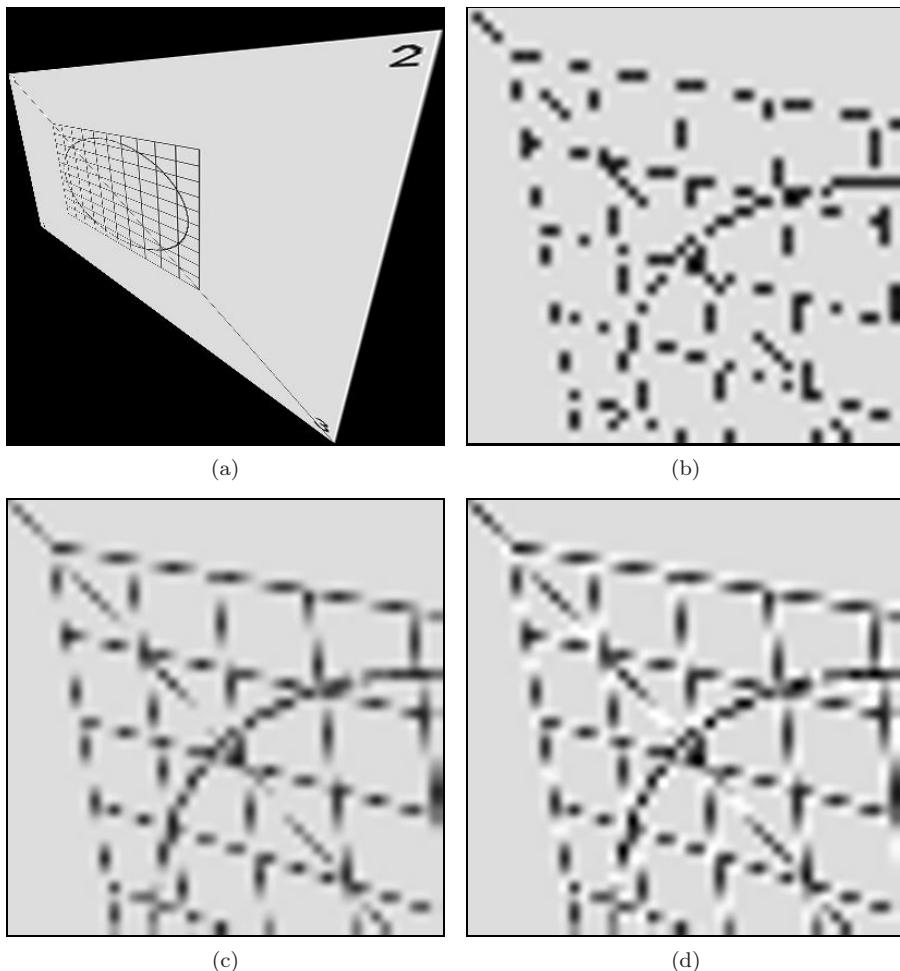


Figure 10.35 Aliasing caused by local image contraction. Aliasing is caused by a violation of the sampling criterion and is largely unaffected by the interpolation method used: complete transformed image (a), detail using nearest-neighbor interpolation (b), bilinear interpolation (c), and bicubic interpolation (d).

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. This can be accomplished with a suitable low-pass filter, as illustrated in Fig. 10.36.

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

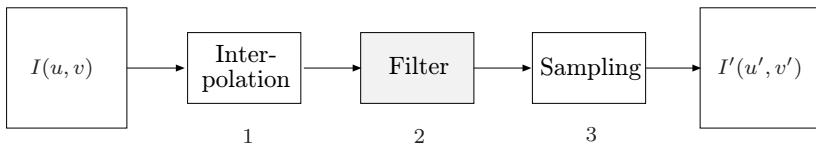


Figure 10.36 Low-pass filtering to avoid aliasing in geometric operations. After interpolation (step 1), the reconstructed image function is subjected to low-pass filtering (step 2) before being resampled (step 3).

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 [22, 33, 79].

10.4 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 [53]. In the following, we develop a rudimentary Java implementation for a set of geometric operations with the class structure summarized in Fig. 10.37. The Java classes form two groups: the first group implements the geometric transformations discussed in Sec. 10.1,¹⁰ while the second group implements the most important interpolation methods described in Sec. 10.3. Finally, we show sample ImageJ plugins to demonstrate the use of this implementation.

10.4.1 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`).

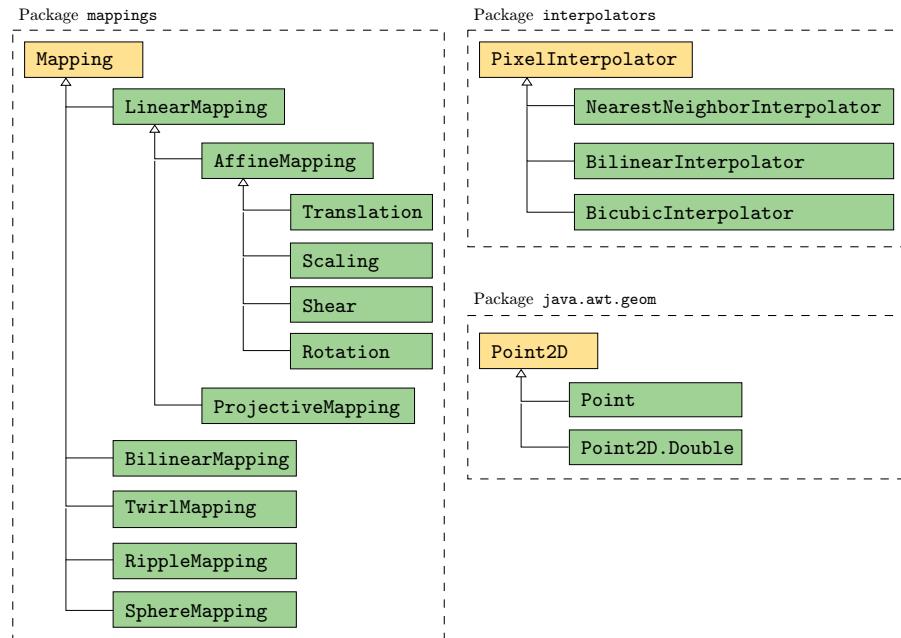


Figure 10.37 Package and class structure for the Java implementation of geometric operations. The class `Mapping` and its subclasses implement the geometric transformations, and `PixelInterpolator` implements various interpolation methods. The standard (abstract) Java AWT class `Point2D` (and its concrete subclasses `Point` and `Point2D.Double`) are used to represent individual points in 2D.

Class Point2D

Two-dimensional coordinates are represented by the (abstract) class `Point2D`, defined in the standard Java package `java.awt.geom`. Its subclasses `Point` and `Point2D.Double` are used to specify coordinate points with integer and floating-point coordinates, respectively.

Class Mapping

The abstract class `Mapping` is the superclass for all subsequent transformations. All subclasses of `Mapping` are required to implement the method

```
Point2D applyTo(Point2D pnt)
```

which applies the corresponding transformation to a given coordinate point `pnt` and returns the transformed point. The method

```
void applyTo(ImageProcessor ip, PixelInterpolator intPol)
```

on the other hand applies this geometric mapping to a whole image (`ip`) using a specified pixel interpolator (`intPol`). This method is implemented by the class `Mapping` itself and is not supposed to be overwritten by subclasses (see line 28 in the code segment below).

The actual image transformation is performed using the *target-to-source* method (Sec. 10.2.2) and thus requires the *inverse* coordinate transform T^{-1} , which can be obtained via the method `getInverse()` (see lines 19 and 33 below). 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 // file Mapping.java
2
3 package mappings;
4 import ij.process.ImageProcessor;
5 import interpolators.PixelInterpolator;
6
7 public abstract class Mapping implements Cloneable {
8
9     boolean isInverse = false;
10
11    // subclasses must implement this method:
12    abstract Point2D applyTo(Point2D pnt);
13
14    Mapping invert() {
15        throw new
16            IllegalArgumentException("cannot invert mapping");
17    }
18
19    Mapping getInverse() {
20        if (isInverse)
21            return this;
22        else
23            return this.invert(); // only linear mappings invert
24    }
25
26    // transforms the image ip using this geometric mapping
27    // and the specified pixel interpolator intPol
28    public void applyTo(ImageProcessor ip, PixelInterpolator intPol)
29    {
30        ImageProcessor targetIp = ip;
31        // make a temporary copy of the image:
32        ImageProcessor sourceIp = ip.duplicate();
33        Mapping invMap = this.getInverse(); // get inverse mapping
34        intPol.setImageProcessor(sourceIp);
35
36        int w = targetIp.getWidth();

```

```
37     int h = targetIp.getHeight();
38
39     Point2D pt = new Point2D.Double();
40     for (int v=0; v<h; v++){
41         for (int u=0; u<w; u++){
42             pt.setLocation(u,v);
43             invMap.applyTo(pt);
44             int p = (int) Math.rint(intPol.getInterpolatedPixel(pt));
45             targetIp.putPixel(u,v,p);
46         }
47     }
48 }
49
50 Mapping duplicate() { //clones any mapping object
51     Mapping newMap = null;
52     try {
53         newMap = (Mapping) this.clone();
54     }
55     catch (CloneNotSupportedException e){};
56     return newMap;
57 }
58
59 } // end of class Mapping
```

Class LinearMapping

LinearMapping is a subclass of Mapping that implements an arbitrary linear transformation in 2D using homogeneous coordinates. The nine elements $a_{11}, a_{12}, \dots, a_{33}$ 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(Point2D pnt)`), inversion (method `invert()`), and concatenation with another linear mapping (method `concat(LinearMapping B)`):

```
1 // file LinearMapping.java
2
3 package mappings;
4 import java.awt.geom.Point2D;
5
6 public class LinearMapping extends Mapping {
7     double
8         a11 = 1, a12 = 0, a13 = 0, // transformation matrix
9         a21 = 0, a22 = 1, a23 = 0,
10        a31 = 0, a32 = 0, a33 = 1;
11
12     LinearMapping() {}
13
14     LinearMapping ( // constructor method
15         double a11, double a12, double a13,
```

```

16     double a21, double a22, double a23,
17     double a31, double a32, double a33, boolean inv) {
18
19     this.a11 = a11; this.a12 = a12; this.a13 = a13;
20     this.a21 = a21; this.a22 = a22; this.a23 = a23;
21     this.a31 = a31; this.a32 = a32; this.a33 = a33;
22     isInverse = inv;
23 }
24
25 Point2D applyTo (Point2D pnt) {          // see Eqn. (10.21)
26     double x0 = pnt.getX();
27     double y0 = pnt.getY();
28     double h = (a31*x0 + a32*y0 + a33);
29     double x1 = (a11*x0 + a12*y0 + a13) / h;
30     double y1 = (a21*x0 + a22*y0 + a23) / h;
31     pnt.setLocation(x1,y1);
32     return pnt;
33 }
34
35 Mapping invert() {                      // see Eqn. (10.27)
36     LinearMapping lm = (LinearMapping) duplicate();
37     double det =
38         a11*a22*a33 + a12*a23*a31 + a13*a21*a32 -
39         a11*a23*a32 - a12*a21*a33 - a13*a22*a31;
40     lm.a11 = (a22*a33 - a23*a32) / det;
41     lm.a12 = (a13*a32 - a12*a33) / det;
42     lm.a13 = (a12*a23 - a13*a22) / det;
43     lm.a21 = (a23*a31 - a21*a33) / det;
44     lm.a22 = (a11*a33 - a13*a31) / det;
45     lm.a23 = (a13*a21 - a11*a23) / det;
46     lm.a31 = (a21*a32 - a22*a31) / det;
47     lm.a32 = (a12*a31 - a11*a32) / det;
48     lm.a33 = (a11*a22 - a12*a21) / det;
49     lm.isInverse = !isInverse;
50     return lm;
51 }
52
53 // concatenates this transform matrix A with B: C ← B · A
54 LinearMapping concat(LinearMapping B) {
55     LinearMapping lm = (LinearMapping) duplicate();
56     lm.a11 = B.a11*a11 + B.a12*a21 + B.a13*a31;
57     lm.a12 = B.a11*a12 + B.a12*a22 + B.a13*a32;
58     lm.a13 = B.a11*a13 + B.a12*a23 + B.a13*a33;
59     lm.a21 = B.a21*a11 + B.a22*a21 + B.a23*a31;
60     lm.a22 = B.a21*a12 + B.a22*a22 + B.a23*a32;
61     lm.a23 = B.a21*a13 + B.a22*a23 + B.a23*a33;
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
68 } // end of class LinearMapping

```

Class AffineMapping

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 (A_i, B_i) as described in Eqn. (10.19):

```
1 // file AffineMapping.java
2
3 package mappings;
4 import java.awt.geom.Point2D;
5
6 public class AffineMapping extends LinearMapping {
7
8     AffineMapping ( // constructor method
9         double a11, double a12, double a13,
10        double a21, double a22, double a23,
11        boolean inv) {
12     super(a11,a12,a13,a21,a22,a23,0,0,1,inv);
13 }
14
15 // create the affine transform between
16 // arbitrary triangles (A1..A3) and (B1..B3)
17 public static AffineMapping makeMapping (
18     Point2D A1, Point2D A2, Point2D A3,
19     Point2D B1, Point2D B2, Point2D B3) {
20
21     double ax1 = A1.getX(), ax2 = A2.getX(), ax3 = A3.getX();
22     double ay1 = A1.getY(), ay2 = A2.getY(), ay3 = A3.getY();
23     double bx1 = B1.getX(), bx2 = B2.getX(), bx3 = B3.getX();
24     double by1 = B1.getY(), by2 = B2.getY(), by3 = B3.getY();
25
26     double S = ax1*(ay3-ay2) + ax2*(ay1-ay3) + ax3*(ay2-ay1);
27     double a11 = (ay1*(bx2-bx3)+ay2*(bx3-bx1)+ay3*(bx1-bx2)) / S;
28     double a12 = (ax1*(bx3-bx2)+ax2*(bx1-bx3)+ax3*(bx2-bx1)) / S;
29     double a21 = (ay1*(by2-by3)+ay2*(by3-by1)+ay3*(by1-by2)) / S;
30     double a22 = (ax1*(by3-by2)+ax2*(by1-by3)+ax3*(by2-by1)) / S;
31     double a13 = (ax1*(ay3*bx2-ay2*bx3) + ax2*(ay1*bx3-ay3*bx1)
32                  + ax3*(ay2*bx1-ay1*bx2)) / S;
33     double a23 = (ax1*(ay3*by2-ay2*by3) + ax2*(ay1*by3-ay3*by1)
34                  + ax3*(ay2*by1-ay1*by2)) / S;
35
36     return new AffineMapping(a11,a12,a13,a21,a22,a23,false);
37 }
38
39 } // end of class AffineMapping
```

Classes Translation, Scaling, Shear, Rotation

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. (10.4)
2   public Translation (double dx, double dy) {
3     super(
4       1, 0, dx,
5       0, 1, dy,
6       false );
7   }
8 }
```

```

1 class Scaling extends AffineMapping { // see Eqn. (10.5)
2   public Scaling(double sx, double sy) {
3     super(
4       sx, 0, 0,
5       0, sy, 0,
6       false );
7   }
8 }
```

```

1 class Shear extends AffineMapping { // see Eqn. (10.6)
2   public Shear(double bx, double by) {
3     super(
4       1, bx, 0,
5       by, 1, 0,
6       false );
7   }
8 }
```

```

1 class Rotation extends AffineMapping { // see Eqn. (10.7)
2   public 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 }
```

Class ProjectiveMapping

The class `ProjectiveMapping` implements a linear projective transformation as defined in Eqn. (10.21). The class provides a constructor method for initial-

izing the eight transformation parameters $a_{11}, a_{12}, \dots, a_{32}$ ($a_{33} = 1$) and two different (overloaded) methods to compute the transformation for given pairs of quadrilaterals.

The first version of `makeMapping()`, in line 17, creates a projective mapping from the unit square S_1 to an arbitrary quadrilateral Q , defined by the coordinate points $P1\dots P4$ (see Eqns. (10.32)–(10.35)). The second version of `makeMapping()`, in line 40, computes the projective mapping between two arbitrary quadrilaterals $A1\dots A4$ and $B1\dots B4$ in two steps via the unit square (see Eqn. (10.37)). This method makes use of the methods `invert()` and `concat()` defined earlier in the class `LinearMapping`:

```

1 // file ProjectiveMapping.java
2
3 package mappings;
4 import java.awt.geom.Point2D;
5
6 class ProjectiveMapping extends LinearMapping {
7
8     ProjectiveMapping(
9         double a11, double a12, double a13,
10        double a21, double a22, double a23,
11        double a31, double a32, boolean inv) {
12     super(a11,a12,a13,a21,a22,a23,a31,a32,1,inv);
13 }
14
15 // creates the projective mapping from the unit square S1 to
16 // the arbitrary quadrilateral Q given by points A1...A4:
17 public static ProjectiveMapping makeMapping(
18     Point2D A1, Point2D A2, Point2D A3, Point2D A4) {
19     double x1=A1.getX(), x2=A2.getX(), x3=A3.getX(), x4=A4.getX();
20     double y1=A1.getY(), y2=A2.getY(), y3=A3.getY(), y4=A4.getY();
21     double S = (x2-x3)*(y4-y3) - (x4-x3)*(y2-y3);
22
23     double a31 = ((x1-x2+x3-x4)*(y4-y3)-(y1-y2+y3-y4)*(x4-x3))/S;
24     double a32 = ((y1-y2+y3-y4)*(x2-x3)-(x1-x2+x3-x4)*(y2-y3))/S;
25
26     double a11 = x2 - x1 + a31*x2;
27     double a12 = x4 - x1 + a32*x4;
28     double a13 = x1;
29
30     double a21 = y2 - y1 + a31*y2;
31     double a22 = y4 - y1 + a32*y4;
32     double a23 = y1;
33
34     return new
35         ProjectiveMapping(a11,a12,a13,a21,a22,a23,a31,a32,false);
36 }
37
38 // creates the projective mapping between arbitrary
39 // quadrilaterals Qa, Qb via the unit square S1: Qa → S1 → Qb
40 public static ProjectiveMapping makeMapping (

```

```

41     Point2D A1, Point2D A2, Point2D A3, Point2D A4,
42     Point2D B1, Point2D B2, Point2D B3, Point2D B4) {
43     ProjectiveMapping T1 = makeMapping(A1, A2, A3, A4);
44     ProjectiveMapping T2 = makeMapping(B1, B2, B3, B4);
45     LinearMapping T1i = (LinearMapping) T1.invert();
46     LinearMapping T = T1i.concat(T2);
47     T.isInverse = false;
48     return (ProjectiveMapping) T;
49 }
50
51 } // end of class ProjectiveMapping

```

Class BilinearMapping

This class implements the bilinear transformation described in Sec. 10.1.5. As a nonlinear mapping, it is a direct subclass of `Mapping`, it has eight transformation parameters ($a_1 \dots b_4$), and (since it does not inherit the corresponding method from class `LinearTransformation`) defines its own `applyTo(Point2D pnt)` method for transforming individual coordinate points (see Eqn. (10.39)):

```

1 // file BilinearMapping.java
2
3 package mappings;
4 import java.awt.geom.Point2D;
5 import Jama.Matrix; // use the JAMA linear algebra package
6
7 class BilinearMapping extends Mapping {
8     double a1, a2, a3, a4;
9     double b1, b2, b3, b4;
10
11    BilinearMapping( // constructor method
12        double a1, double a2, double a3, double a4,
13        double b1, double b2, double b3, double b4,
14        boolean inv) {
15        this.a1 = a1; this.a2 = a2; this.a3 = a3; this.a4 = a4;
16        this.b1 = b1; this.b2 = b2; this.b3 = b3; this.b4 = b4;
17        isInverse = inv;
18    }
19
20    Point2D applyTo (Point2D pnt){
21        double x0 = pnt.getX();
22        double y0 = pnt.getY();
23        double x1 = a1 * x0 + a2 * y0 + a3 * x0 * y0 + a4;
24        double y1 = b1 * x0 + b2 * y0 + b3 * x0 * y0 + b4;
25        pnt.setLocation(x1, y1);
26        return pnt;
27    }
28
29    // (continued below)

```

To avoid the task of inverting the forward 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 $P_1 \dots P_4$) to another quadrilateral \mathcal{Q} ($Q_1 \dots Q_4$):

```

30  // map between arbitrary quadrilaterals  $\mathcal{P} \rightarrow \mathcal{Q}$ 
31  public static BilinearMapping makeInverseMapping(
32      Point2D P1, Point2D P2, Point2D P3, Point2D P4, // source quad  $\mathcal{P}$ 
33      Point2D Q1, Point2D Q2, Point2D Q3, Point2D Q4) // target quad  $\mathcal{Q}$ 
34  {
35
36      //set up the column vectors  $x, y$ 
37      Matrix X = new Matrix(new double[][] [
38          {Q1.getX()}, {Q2.getX()}, {Q3.getX()}, {Q4.getX()}]);
39      Matrix Y = new Matrix(new double[][] [
40          {Q1.getY()}, {Q2.getY()}, {Q3.getY()}, {Q4.getY()}]);
41
42      //set up the matrix  $M$ 
43      Matrix M = new Matrix(new double[][] [
44          {P1.getX(), P1.getY(), P1.getX() * P1.getY(), 1},
45          {P2.getX(), P2.getY(), P2.getX() * P2.getY(), 1},
46          {P3.getX(), P3.getY(), P3.getX() * P3.getY(), 1},
47          {P4.getX(), P4.getY(), P4.getX() * P4.getY(), 1}
48      );
49
50      Matrix A = M.solve(X); // solve  $x = M \cdot a$  (Eqn. (10.40))
51      Matrix B = M.solve(Y); // solve  $y = M \cdot b$  (Eqn. (10.41))
52
53      double a1 = A.get(0,0); double b1 = B.get(0,0);
54      double a2 = A.get(1,0); double b2 = B.get(1,0);
55      double a3 = A.get(2,0); double b3 = B.get(2,0);
56      double a4 = A.get(3,0); double b4 = B.get(3,0);
57
58      return new BilinearMapping(a1,a2,a3,a4,b1,b2,b3,b4,true);
59  }
60
61 } // end of class BilinearMapping

```

In the method above, two 4×4 systems of linear equations are solved in lines 50–51 using the method `solve()` of the `Matrix` class in the JAMA¹¹ numerical library.

Class TwirlMapping

This class implements the twirl mapping (Eqns. (10.42) and (10.43)) 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

¹¹ <http://math.nist.gov/javanumerics/jama/>.

`makeInverseMapping():`

```

1 // file TwirlMapping.java
2
3 package mappings;
4 import java.awt.geom.Point2D;
5
6 public class TwirlMapping extends Mapping {
7     double xc, yc, angle, rad;
8
9     TwirlMapping (double xc, double yc, double angle,
10                  double rad, boolean inv) {
11         this.xc = xc;  this.yc = yc;
12         this.angle = angle;
13         this.rad = rad;
14         this.isInverse = inv;
15     }
16
17     public static TwirlMapping makeInverseMapping (
18         double xc, double yc, double angle, double rad) {
19         return new TwirlMapping(xc, yc, angle, rad, true);
20     }
21
22     Point2D applyTo (Point2D pnt){
23         double x = pnt.getX();
24         double y = pnt.getY();
25         double dx = x - xc;
26         double dy = y - yc;
27         double d = Math.sqrt(dx*dx + dy*dy);
28         if (d < rad) {
29             double a = Math.atan2(dy,dx) + angle * (rad-d) / rad;
30             double x1 = xc + d * Math.cos(a);
31             double y1 = yc + d * Math.sin(a);
32             pnt.setLocation(x1,y1);
33         }
34         return pnt;
35     }
36 }
37 } // end of class TwirlMapping

```

Similar classes could be defined to implement the ripple transformation and the spherical distortion described in Sec. 10.1.6 (see Exercise 10.3).

10.4.2 Pixel Interpolation

The following class definitions implement three of the interpolation methods described in Sec. 10.3. Each class provides its own version of the method `getInterpolatedPixel(Point2D 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.

Class PixelInterpolator

PixelInterpolator is the (abstract) superclass for the actual interpolator classes. In particular, it specifies the method `getInterpolatedPixel(Point2D pnt)`, which must be implemented by all subclasses and is invoked by the method `applyTo()` in class `Mapping` (p. 241):

```
1 // file PixelInterpolator.java
2
3 package interpolators;
4 import java.awt.geom.Point2D;
5 import ij.process.ImageProcessor;
6
7 public abstract class PixelInterpolator {
8     ImageProcessor ip;
9
10    PixelInterpolator() {}
11
12    public void setImageProcessor(ImageProcessor ip) {
13        this.ip = ip;
14    }
15
16    public abstract double getInterpolatedPixel(Point2D pnt);
17
18 } // end of class PixelInterpolator
```

Class NearestNeighborInterpolator

This class implements the two-dimensional *nearest-neighbor* interpolation (see Eqn. (10.72)):

```
1 // file NearestNeighborInterpolator.java
2
3 package interpolators;
4 import java.awt.geom.Point2D;
5
6 public class NearestNeighborInterpolator extends PixelInterpolator {
7
8     public double getInterpolatedPixel(Point2D pnt) {
9         int u = (int) Math.rint(pnt.getX());
10        int v = (int) Math.rint(pnt.getY());
11        return ip.getPixel(u,v);
12    }
13
14 } // end of class NearestNeighborInterpolator
```

Class BilinearInterpolator

This class implements the *bilinear* interpolation method (see Eqn. (10.75)):

```

1 // file BilinearInterpolator.java
2
3 package interpolators;
4 import java.awt.geom.Point2D;
5
6 public class BilinearInterpolator extends PixelInterpolator {
7
8     public double getInterpolatedPixel(Point2D pnt) {
9         double x = pnt.getX();
10        double y = pnt.getY();
11        int u = (int) Math.floor(x);
12        int v = (int) Math.floor(y);
13        double a = x - u;
14        double b = y - v;
15        int A = ip.getPixel(u,v);
16        int B = ip.getPixel(u+1,v);
17        int C = ip.getPixel(u,v+1);
18        int D = ip.getPixel(u+1,v+1);
19        double E = A + a*(B-A);
20        double F = C + a*(D-C);
21        double G = E + b*(F-E);
22        return G;
23    }
24 } // 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`.

Class BicubicInterpolator

This class implements the bicubic interpolation method described in Eqn. (10.78) and Alg. 10.2. The control parameter a (with $a = 1$ as the default value) can be modified by the corresponding parameter in the second constructor method (line 11). The one-dimensional cubic interpolation (Eqn. (10.59)) is implemented by the method `double cubic (double x)`:

```

1 // file BicubicInterpolator.java
2
3 package interpolators;
4 import java.awt.geom.Point2D;
5
6 public class BicubicInterpolator extends PixelInterpolator {
7     double a = 1; // control parameter a (default setting)
8
9     public BicubicInterpolator() {}
10
11    public BicubicInterpolator(double a) {

```

```

12     this.a = a;
13 }
14
15 public double getInterpolatedPixel(Point2D pnt) {
16     double x0 = pnt.getX();
17     double y0 = pnt.getY();
18     // use floor to correctly handle negative coordinates:
19     int u0 = (int) Math.floor(x0);
20     int v0 = (int) Math.floor(y0);
21
22     double q = 0;
23     for (int j = 0; j <= 3; j++) {
24         int v = v0 + j - 1;
25         double p = 0;
26         for (int i = 0; i <= 3; i++) {
27             int u = u0 + i - 1;
28             p = p + ip.getPixel(u,v) * cubic(x0 - u);
29         }
30         q = q + p * cubic(y0 - v);
31     }
32     return q;
33 }
34
35 double cubic(double x) {
36     if (x < 0) x = -x;
37     double z = 0;
38     if (x < 1)
39         z = (-a+2)*x*x*x + (a-3)*x*x + 1;
40     else if (x < 2)
41         z = -a*x*x*x + 5*a*x*x - 8*a*x + 4*a;
42     return z;
43 }
44
45 } // end of class BicubicInterpolator

```

Setting $a = 0.5$, this class implements a *Catmull-Rom* interpolation (see Sec. 10.3.5).

10.4.3 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 17). Subsequently, the interpolator object `ipol` is created (line 18). 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 19:

```

1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4 import interpolators.BicubicInterpolator;
5 import interpolators.PixelInterpolator;
6 import mappings.Rotation;
7
8 public class Geometry_Rotate implements PlugInFilter {
9
10    double angle = 15; // rotation angle (in degrees)
11
12    public int setup(String arg, ImagePlus imp) {
13        return DOES_8G;
14    }
15
16    public void run(ImageProcessor ip) {
17        Rotation map = new Rotation((2 * Math.PI * angle) / 360);
18        PixelInterpolator ipol = new BicubicInterpolator(0.5);
19        map.applyTo(ip, ipol);
20    }
21
22 } // 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 $\mathcal{P} = p_1 \dots p_4$ and $\mathcal{Q} = q_1 \dots q_4$. 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 27. In this case, we used a *bilinear* interpolator (line 29), which is applied as in the previous example (line 30):

```

1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4 import interpolators.BicubicInterpolator;
5 import interpolators.PixelInterpolator;
6 import java.awt.Point;
7 import java.awt.geom.Point2D;
8 import mappings.ProjectiveMapping;
9
10 public class Geometry_ProjectiveMapping implements PlugInFilter {
11
12    public int setup(String arg, ImagePlus imp) {
13        return DOES_8G;
14    }
15
16    public void run(ImageProcessor ip) {
17        Point p1 = new Point(100, 100);
18        Point p2 = new Point(200, 100);
19        Point p3 = new Point(100, 200);
20        Point p4 = new Point(200, 200);
21
22        Point q1 = new Point(150, 150);
23        Point q2 = new Point(250, 150);
24        Point q3 = new Point(150, 250);
25        Point q4 = new Point(250, 250);
26
27        ProjectiveMapping map = ProjectiveMapping.makeMapping(p1, p2, p3, p4, q1, q2, q3, q4);
28
29        BilinearInterpolator ipol = new BilinearInterpolator(0.5);
30        map.applyTo(ip, ipol);
31    }
32
33 } // end of class Geometry_ProjectiveMapping

```

```
14 }
15
16 public void run(ImageProcessor ip) {
17     Point2D p1 = new Point(0,0);
18     Point2D p2 = new Point(400,0);
19     Point2D p3 = new Point(400,400);
20     Point2D p4 = new Point(0,400);
21
22     Point2D q1 = new Point(0,60);
23     Point2D q2 = new Point(400,20);
24     Point2D q3 = new Point(300,400);
25     Point2D q4 = new Point(30,200);
26
27     ProjectiveMapping map =
28         ProjectiveMapping.makeMapping(p1,p2,p3,p4,q1,q2,q3,q4);
29     PixelInterpolator ipol = new BilinearInterpolator();
30     map.applyTo(ip, ipol);
31 }
32
33 } // end of class Geometry_ProjectiveMapping
```

10.5 Exercises

Exercise 10.1

Show that a straight line $y = kx + d$ in 2D is mapped to another straight line under a projective transformation (Eqn. (10.21)).

Exercise 10.2

Show that parallel lines remain parallel under affine transformation (Eqn. (10.13)).

Exercise 10.3

Implement the nonlinear geometric transformations `RippleMapping` (Eqns. (10.44) and (10.45)) and `SphereMapping` (Eqns. (10.46) and (10.47)) as Java classes analogous to the implementation of `TwirlMapping` (p. 247). Also create suitable ImageJ plugins and use them to test these mappings.

Exercise 10.4

Design a nonlinear geometric transformation similar to the ripple transformation (Eqns. (10.44) and (10.45)) that uses a *sawtooth* function instead of a sinusoid for the distortions in the horizontal and vertical directions. Use the class `TwirlMapping` (p. 247) as a template for your implementation.

Exercise 10.5

The one-dimensional interpolation function by Mitchell and Natravali $w_{mn}(x)$ is defined as a general spline function $w_{cs}(x, a, b)$ (Eqn. (10.63)). Show that this function can be expressed as the weighted sum of a Catmull-Rom function $w_{crm}(x)$ (Eqn. (10.61)) and a cubic B-spline $w_{cbs}(x)$ (Eqn.

(10.62)) in the form

$$\begin{aligned} w_{mn}(x) &= w_{cs}\left(x, \frac{1}{3}, \frac{1}{3}\right) \\ &= \frac{1}{3} \cdot [2 \cdot w_{cs}(x, 0.5, 0) + w_{cs}(x, 0, 1)] \\ &= \frac{1}{3} \cdot [2 \cdot w_{crm}(x) + w_{cbs}(x)]. \end{aligned}$$

Exercise 10.6

Implement the two-dimensional *Mitchell-Netravali* interpolation as defined by Eqn. (10.63) and Eqn. (10.80) as a Java class analogous to the class **BicubicInterpolator** (p. 250). Compare the results with those of the bicubic interpolation.

Exercise 10.7

Implement the two-dimensional *Lanczos* interpolation with a W_{L3} kernel as defined in Eqn. (10.82) as a Java class analogous to the class **Bicubic-Interpolator** (p. 250). Compare the results to the bicubic interpolation.

Exercise 10.8

The one-dimensional Lanczos interpolation kernel of order $n = 4$ is (analogous to Eqn. (10.70)) defined as

$$w_{L4} = \begin{cases} 1 & \text{for } |x| = 0 \\ 4 \cdot \frac{\sin(\pi \frac{x}{4}) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{for } 0 \leq |x| < 4 \\ 0 & \text{for } |x| \geq 4. \end{cases} \quad (10.84)$$

Generalize the two-dimensional L3 kernel in Eqn. (10.82) to L_n , where n (the number of “taps”) can be chosen arbitrarily, and implement this interpolator as a Java class analogous to **BicubicInterpolator** (p. 250). Perform tests on suitable images to see how the interpolation performs when n is increased.

11

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

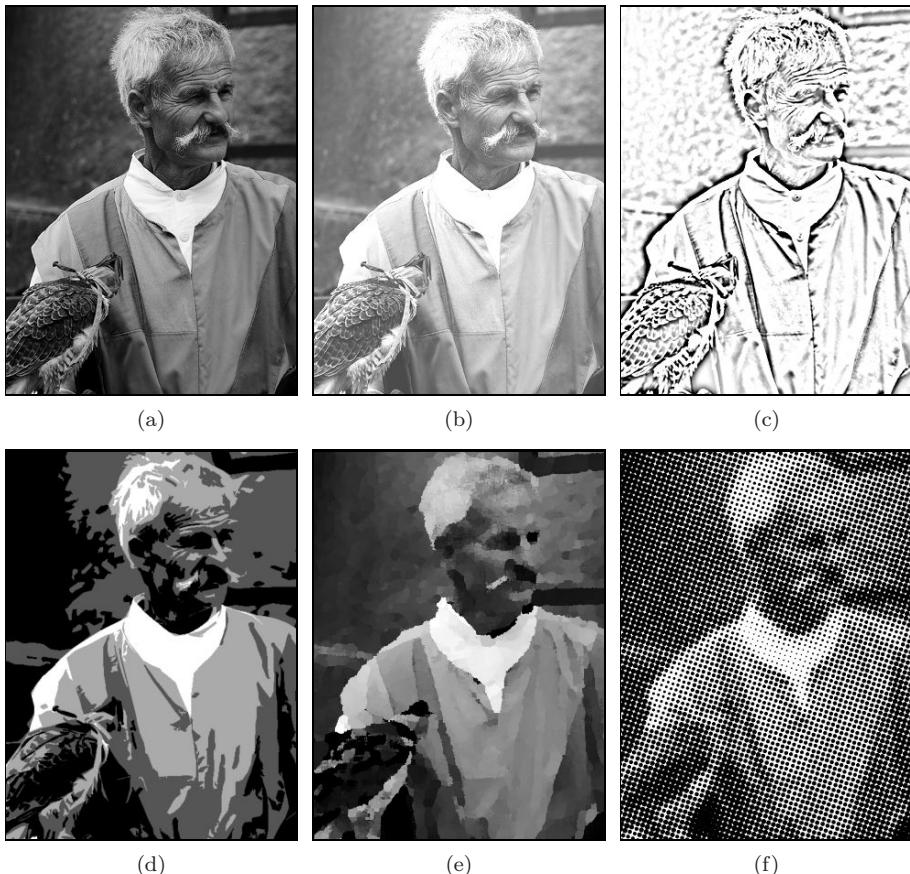


Figure 11.1 Are these images the “same”? Simply measuring the difference between pixel values will return a large distance between the original image (a) and any of the five other images (b–f).

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, and what happens when brightness or contrast changes (Fig. 11.1)? We already touched on this problem of invariance under geometric transformations when we discussed the shape properties of segmented regions in Sec. 2.4.2. 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.

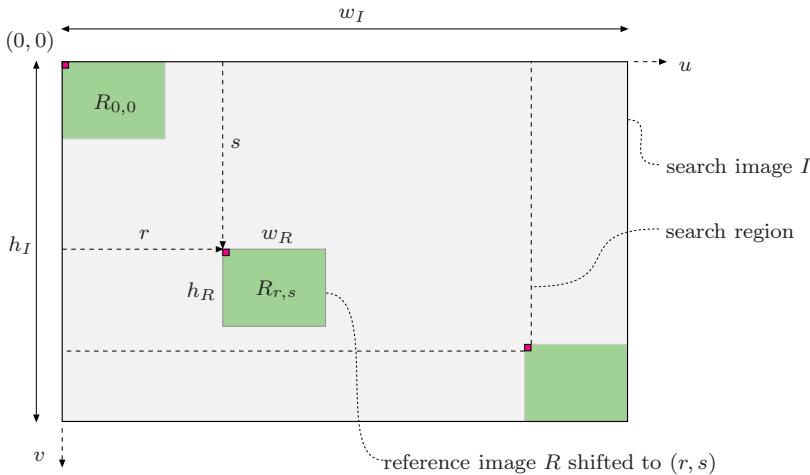


Figure 11.2 Geometry of template matching. The reference image R is shifted across the search image I by an offset (r, s) using the origins of the two images as the reference points. The dimensions of the search image ($w_I \times h_I$) and the reference image ($w_R \times h_R$) determine the maximal search region for this comparison.

11.1 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

$$R_{r,s}(u, v) = R(u - r, v - s) \quad (11.1)$$

the reference image R shifted by the distance (r, s) in the horizontal and vertical directions, respectively, then the matching problem (illustrated in Fig. 11.2) can be summarized as

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 sub-images must be found that is reasonably tolerant against intensity and contrast variations.

11.1.1 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. 11.3). Several distance measures have been proposed for two-dimensional intensity images, including the following three basic definitions:¹

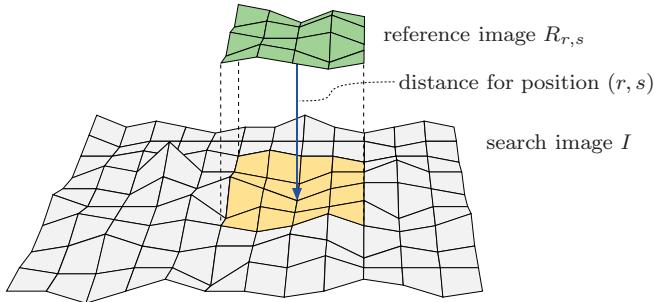


Figure 11.3 Measuring the distance between two-dimensional image functions. The reference image R is positioned at offset (r, s) on top of the search image I .

Sum of absolute differences:

$$d_A(r, s) = \sum_{(i,j) \in R} |I(r+i, s+j) - R(i, j)|; \quad (11.2)$$

Maximum difference:

$$d_M(r, s) = \max_{(i,j) \in R} |I(r+i, s+j) - R(i, j)|; \quad (11.3)$$

Sum of squared differences:

$$d_E(r, s) = \left[\sum_{(i,j) \in R} (I(r+i, s+j) - R(i, j))^2 \right]^{1/2}. \quad (11.4)$$

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.

¹ We use the short notation $(i, j) \in R$ to specify the set of all possible template coordinates $\{(i, j) \mid 0 \leq i < w_R, 0 \leq j < h_R\}$.

Distance and correlation

Because of its formal properties, the N -dimensional distance d_E (Eqn. (11.4)) 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

$$\begin{aligned} d_E^2(r, s) &= \sum_{(i,j) \in R} (I(r+i, s+j) - R(i, j))^2 \\ &= \underbrace{\sum_{(i,j) \in R} I^2(r+i, s+j)}_{A(r, s)} + \underbrace{\sum_{(i,j) \in R} R^2(i, j)}_B - 2 \cdot \underbrace{\sum_{(i,j) \in R} I(r+i, s+j) \cdot R(i, j)}_{C(r, s)}. \end{aligned} \quad (11.5)$$

Notice that the term B in Eqn. (11.5) 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

$$(I \circledast R)(r, s) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(r+i, s+j) \cdot R(i, j), \quad (11.6)$$

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. (11.5). As we can see in Eqn. (11.6), correlation is in principle the same operation as linear *convolution* (see Vol. 1, Eqn. (5.14) in Sec. 5.3.1 [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. (11.5) 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. (11.5)) 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. 8.5).

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:

$$\begin{aligned} C_N(r, s) &= \frac{C(r, s)}{\sqrt{A(r, s) \cdot B}} = \frac{C(r, s)}{\sqrt{A(r, s)} \cdot \sqrt{B}} \\ &= \frac{\sum_{(i,j) \in R} I(r+i, s+j) \cdot R(i, j)}{\left[\sum_{(i,j) \in R} I^2(r+i, s+j) \right]^{1/2} \cdot \left[\sum_{(i,j) \in R} R^2(i, j) \right]^{1/2}}. \end{aligned} \quad (11.7)$$

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. (11.6), the expression in Eqn. (11.7) 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. (11.7) into

$$C_L(r, s) = \frac{\sum_{(i,j) \in R} (I(r+i, s+j) - \bar{I}(r, s)) \cdot (R(i, j) - \bar{R})}{\left[\sum_{(i,j) \in R} (I(r+i, s+j) - \bar{I}_{r,s})^2 \right]^{1/2} \cdot \underbrace{\left[\sum_{(i,j) \in R} (R(i, j) - \bar{R})^2 \right]^{1/2}}_{S_R^2 = K \cdot \sigma_R^2}}, \quad (11.8)$$

where the average values $\bar{I}_{r,s}$ and \bar{R} are defined as

$$\bar{I}_{r,s} = \frac{1}{K} \cdot \sum_{(i,j) \in R} I(r+i, s+j) \quad \text{and} \quad \bar{R} = \frac{1}{K} \cdot \sum_{(i,j) \in R} R(i, j), \quad (11.9)$$

respectively ($K = |R|$ being the number of pixels in the reference image R). In statistics, the expression in Eqn. (11.8) 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

$$S_R^2 = K \cdot \sigma_R^2 = \sum_{(i,j) \in R} (R(i, j) - \bar{R})^2 \quad (11.10)$$

in the denominator of Eqn. (11.8) 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. (11.10) can be reformulated as

$$\begin{aligned} S_R^2 &= \sum_{(i,j) \in R} R^2(i, j) - K \cdot \bar{R}^2 \\ &= \sum_{(i,j) \in R} R^2(i, j) - \frac{1}{K} \cdot \left(\sum_{(i,j) \in R} R(i, j) \right)^2. \end{aligned} \quad (11.11)$$

By inserting the results from Eqns. (11.9) and (11.11) we can rewrite Eqn. (11.8) as

$$C_L(r, s) = \frac{\sum_{(i,j) \in R} (I(r+i, s+j) \cdot R(i, j)) - K \cdot \bar{I}_{r,s} \cdot \bar{R}}{\left[\sum_{(i,j) \in R} I^2(r+i, s+j) - K \cdot \bar{I}_{r,s}^2 \right]^{1/2} \cdot S_R} \quad (11.12)$$

and thereby obtain an efficient way to compute the local correlation coefficient. Since \bar{R} and $S_R = \sqrt{S_R^2}$ 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. (11.12) can be computed in one common iteration, as described in Alg. 11.1. A sample Java implementation of this procedure is given by the class `CorrCoeffMatcher` in Progs. 11.1 and 11.2 (Sec. 11.1.2).

Algorithm 11.1 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. (11.12). 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: Let  $K \leftarrow w_R \cdot h_R$ 
3: Let  $\Sigma_R \leftarrow 0$ ,  $\Sigma_{R2} \leftarrow 0$ 
4: for  $i \leftarrow 0 \dots (w_R - 1)$  do
5:   for  $j \leftarrow 0 \dots (h_R - 1)$  do
6:      $\Sigma_R \leftarrow \Sigma_R + R(i, j)$ 
7:      $\Sigma_{R2} \leftarrow \Sigma_{R2} + (R(i, j))^2$ 
8:   Let  $\bar{R} \leftarrow \Sigma_R / K$                                  $\triangleright$  Eqn. (11.9)
9:   Let  $S_R \leftarrow \sqrt{\Sigma_{R2} - K \cdot \bar{R}^2} = \sqrt{\Sigma_{R2} - \Sigma_R^2 / K}$        $\triangleright$  Eqn. (11.11)

   STEP 2—COMPUTE THE CORRELATION MAP:
10:  Let  $C \leftarrow$  new map of size  $(w_I - w_R + 1) \times (h_I - h_R + 1)$ ,  $C(r, s) \in \mathbb{R}$ 
11:  for  $r \leftarrow 0 \dots (w_I - w_R)$  do                       $\triangleright$  place  $R$  at position  $(r, s)$ 
12:    for  $s \leftarrow 0 \dots (h_I - h_R)$  do
13:      Compute the correlation coefficient for position  $(r, s)$ :
14:      Let  $\Sigma_I \leftarrow 0$ ,  $\Sigma_{I2} \leftarrow 0$ ,  $\Sigma_{IR} \leftarrow 0$ 
15:      for  $i \leftarrow 0 \dots (w_R - 1)$  do
16:        for  $j \leftarrow 0 \dots (h_R - 1)$  do
17:          Let  $a_I \leftarrow I(r+i, s+j)$ 
18:          Let  $a_R \leftarrow R(i, j)$ 
19:          Let  $\Sigma_I \leftarrow \Sigma_I + a_I$ 
20:          Let  $\Sigma_{I2} \leftarrow \Sigma_{I2} + a_I^2$ 
21:          Let  $\Sigma_{IR} \leftarrow \Sigma_{IR} + a_I \cdot a_R$ 
22:          Let  $\bar{I}_{r,s} \leftarrow \Sigma_I / K$                                  $\triangleright$  Eqn. (11.9)
23:           $C(r, s) \leftarrow \frac{\Sigma_{IR} - K \cdot \bar{I}_{r,s} \cdot \bar{R}}{\sqrt{\Sigma_{I2} - K \cdot \bar{I}_{r,s}^2} \cdot S_R} = \frac{\Sigma_{IR} - \Sigma_I \cdot \bar{R}}{\sqrt{\Sigma_{I2} - \Sigma_I^2 / K} \cdot S_R}$ 
24:        return  $C$ .                                          $\triangleright C(r, s) \in [-1, 1]$ 
```

Examples and discussion

Figure 11.4 compares the performance of the described distance functions in a typical example. The original image (Fig. 11.4(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. 11.4(b)).

- The *sum of absolute differences* (Eqn. (11.2)) in Fig. 11.4(c) shows a distinct peak value at the original template position, as does the *Euclidean distance* (Eqn. (11.4)) in Fig. 11.4(e). Both measures work satisfactorily in this regard but are strongly affected by global intensity changes, as demonstrated in Figs. 11.5 and 11.6.
- The *maximum difference* (Eqn. (11.3)) in Fig. 11.4(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. 11.4(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. 11.4(g) appears naturally very similar to the Euclidean distance (Fig. 11.4(e)) because in principle it is the same measure. As expected, the *correlation coefficient* (Eqn. (11.8)) in Fig. 11.4(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.

Figure 11.5 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. 11.5(c)), while the correlation coefficient (Fig. 11.5(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

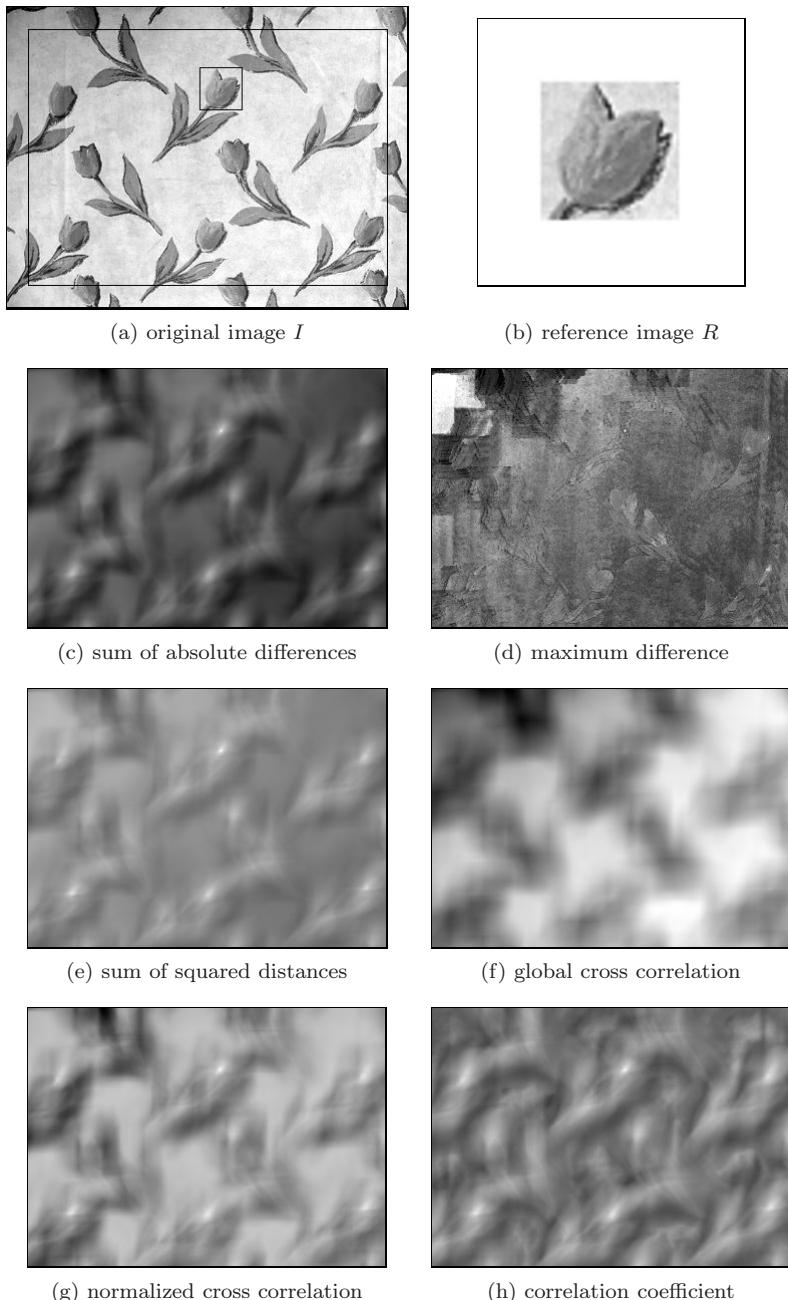


Figure 11.4 Comparison of various distance functions. From the original image (a), the marked section is used as the reference image R , shown enlarged in (b). In the resulting difference images (c–h), brightness corresponds to the amount of agreement.

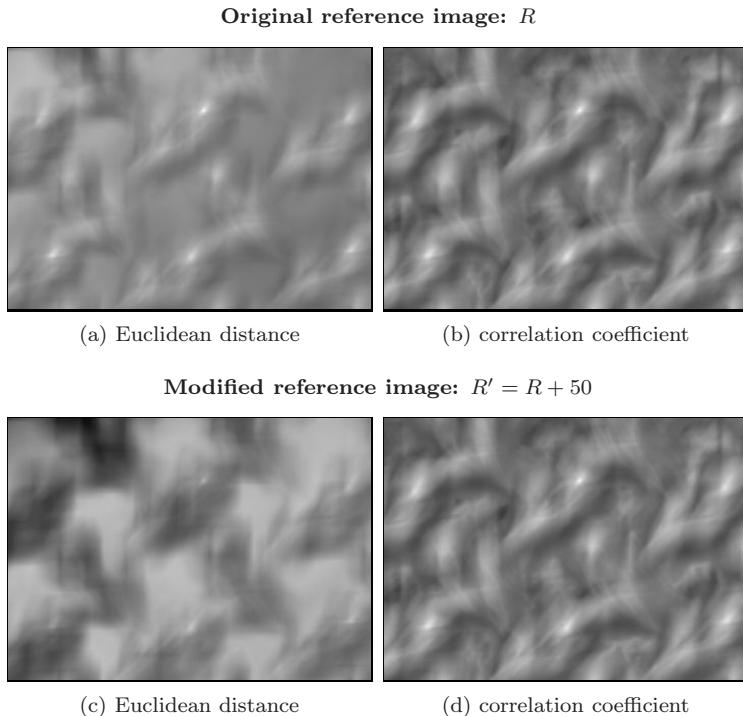


Figure 11.5 Effects of changing global intensity. Original reference image R : the results from both the Euclidean distance (a) and the correlation coefficient (b) show distinct peaks at the positions of maximum agreement. Modified reference image $R' = R + 50$: the peak values disappear in the Euclidean distance (c), while the correlation coefficient (d) remains unaffected.

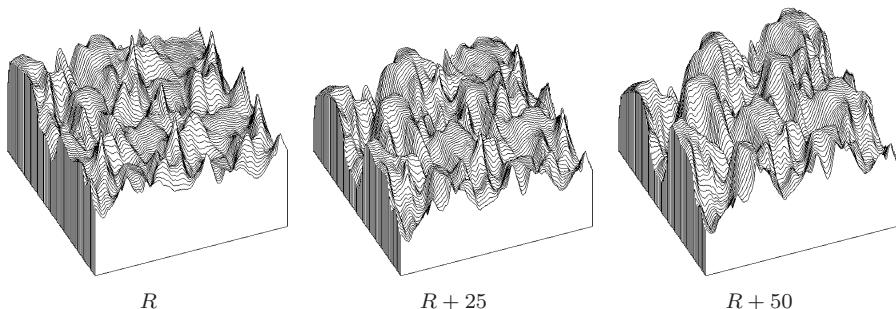


Figure 11.6 Euclidean distance under global intensity changes. Distance function for the original template R (left), with the template intensity increased by 25 units (center) and 50 units (right). Notice that the local peaks disappear as the template intensity (and thus the total distance between the image and the template) is increased.

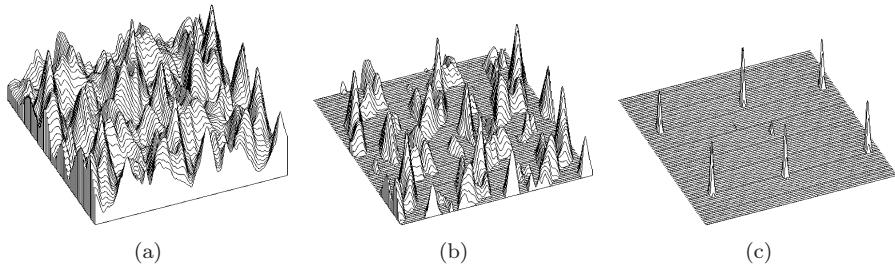


Figure 11.7 Detection of match points by simple thresholding: correlation coefficient (a), positive values only (b), and values greater than 0.5 (c). The remaining peaks indicate the positions of the six similar (but not identical) tulip patterns in the original image (Fig. 11.4 (a)).

the resulting values are in the fixed range of $[-1, 1]$, a simple threshold operation can be used to localize the best match points (Fig. 11.7).

11.1.2 Implementation

Programs 11.1 and 11.2 lists a Java implementation of template matching based on the local correlation coefficient (Eqn. (11.8)). 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:

```
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 (`matchP`) of type `FloatProcessor`. The performance can be improved by using direct access to the pixel arrays (instead of the access methods `getf()` and `setf()`).²

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

² See the ImageJ Short Reference [13, Chap. 7] for details.

```

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 ( $\bar{R}$ )
9     float varR;     // square root of template variance ( $\sigma_R$ )
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 ( $\bar{R}$ ) and variance term ( $S_R$ ) of the template:
24        float sumR = 0;      //  $\sum_R = \sum R(i,j)$ 
25        float sumR2 = 0;     //  $\sum_{R^2} = \sum R^2(i,j)$ 
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;   //  $\bar{R} = [\sum R(i,j)]/K$ 
34        varR =             //  $S_R = [\sum R^2(i,j) - K \cdot \bar{R}^2]^{1/2}$ 
35        (float) Math.sqrt(sumR2 - K * meanR * meanR);
36    }
37
38    // continued...

```

Program 11.1 Class `CorrCoeffMatcher`. This is a direct implementation of Alg. 11.1. The constructor method (lines 11–36) computes the mean $\bar{R} = \text{meanR}$ (Eqn. (11.9)) and the variance term $S_R = \text{varR}$ (Eqn. (11.11)) of the reference image R .

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.

11.1.3 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

```

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; //  $\Sigma_I = \sum I(r+i, s+j)$ 
53         float sumI2 = 0; //  $\Sigma_{I^2} = \sum (I(r+i, s+j))^2$ 
54         float sumIR = 0; //  $\Sigma_{IR} = \sum I(r+i, s+j) \cdot R(i, j)$ 
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; //  $\bar{I}_{r,s} = \Sigma_I / K$ 
66         return (sumIR - K * meanI * meanR) /
67             ((float)Math.sqrt(sumI2 - K * meanI * meanI) * varR);
68     }
69
70 } // end of class CorrCoeffMatcher

```

Program 11.2 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. (11.12)).

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.

Solutions to the rotation and scaling problems (such as matching in *logarithmic-polar space* [81]) 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) [50, 62, 75].

11.2 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.

11.2.1 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. 11.8).

The problem with directly comparing binary images is that even the smallest deviations between image patterns, such as those caused by a 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

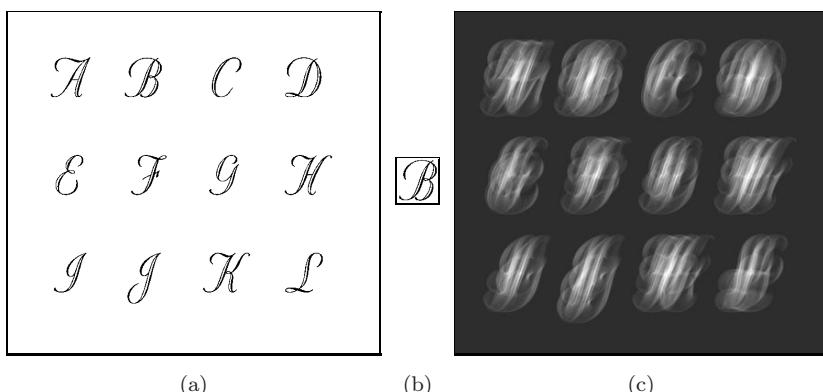


Figure 11.8 Direct comparison of binary images. Given are a binary search image (a) and a binary reference image (b). The local similarity value for any template position corresponds to the relative number of matching (black) foreground pixels. High similarity values are shown as bright spots in the 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.

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.

11.2.2 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 $I(u, v) = I(\mathbf{p})$, we denote

$$FG(I) = \{\mathbf{p} \mid I(\mathbf{p}) = 1\}, \quad (11.13)$$

$$BG(I) = \{\mathbf{p} \mid I(\mathbf{p}) = 0\}, \quad (11.14)$$

as the set of coordinates of the foreground and background pixels, respectively. The so-called distance transform of I , $D(\mathbf{p}) \in \mathbb{R}$, is defined as

$$D(\mathbf{p}) = \min_{\mathbf{p}' \in FG(I)} \text{dist}(\mathbf{p}, \mathbf{p}') \quad (11.15)$$

for all $\mathbf{p} = (u, v)$, where $u = 0 \dots M-1$, $v = 0 \dots N-1$ (for image size $M \times N$). If $I(\mathbf{p})$ is a foreground pixel itself (i.e., $\mathbf{p} \in FG$), then the distance $D(\mathbf{p}) = 0$ since no shift is necessary for moving this pixel onto a foreground pixel.

The function $\text{dist}(\mathbf{p}, \mathbf{p}')$ in Eqn. (11.15) measures the geometric distance between the two coordinate points $\mathbf{p} = (u, v)$ and $\mathbf{p}' = (u', v')$. Examples of suitable distance functions are the Euclidean distance

$$d_E(\mathbf{p}, \mathbf{p}') = \|\mathbf{p} - \mathbf{p}'\| = \sqrt{(u - u')^2 + (v - v')^2} \in \mathbb{R}^+ \quad (11.16)$$

or the *Manhattan distance*³

$$d_M(\mathbf{p}, \mathbf{p}') = |u - u'| + |v - v'| \in \mathbb{N}_0. \quad (11.17)$$

Figure 11.9 shows a simple example of a distance transform using the Manhattan distance $d_M()$.

The direct implementation of the distance transform (following the definition in Eqn. (11.15)) is computationally expensive because the closest foreground pixel must be found for each pixel position \mathbf{p} (unless $I(\mathbf{p})$ is a foreground pixel itself).⁴

³ Also called “city block distance”.

⁴ 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 $N \times N$.

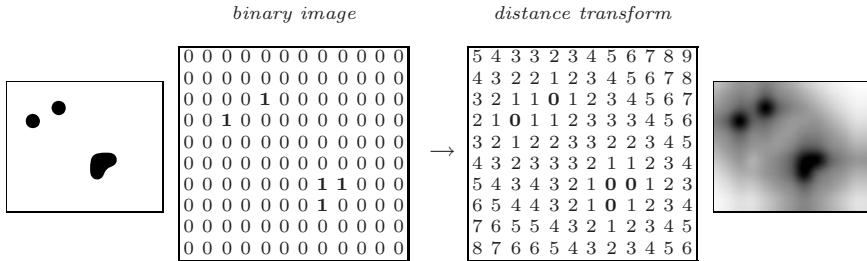


Figure 11.9 Example of a distance transform of a binary image using the Manhattan distance $d_M()$.

Chamfer algorithm

The so-called *chamfer* algorithm [7] is an efficient method for computing the distance transform. Similar to the sequential region labeling algorithm (Alg. 2.3 in Sec. 2.1.2), 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,

$$M^L = \begin{bmatrix} m_2^L & m_3^L & m_4^L \\ m_1^L & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \text{and} \quad M^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & m_1^R \\ m_4^R & m_3^R & m_2^R \end{bmatrix} \quad (11.18)$$

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 function $\text{dist}(\mathbf{p}, \mathbf{p}')$ used. In particular, the distance masks for the Manhattan distance (Eqn. (11.17)) are

$$M_M^L = \begin{bmatrix} 2 & 1 & 2 \\ 1 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}, \quad M_M^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & 1 \\ 2 & 1 & 2 \end{bmatrix}, \quad (11.19)$$

and similarly for the Euclidean distance (Eqn. (11.16))

$$M_E^L = \begin{bmatrix} \sqrt{2} & 1 & \sqrt{2} \\ 1 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}, \quad M_E^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & 1 \\ \sqrt{2} & 1 & \sqrt{2} \end{bmatrix}. \quad (11.20)$$

Algorithm 11.2 Chamfer algorithm for computing the distance transform. From the binary image I , the distance transform D (Eqn. (11.15)) is computed using a pair of distance masks (Eqn. (11.18)) for the first and second passes. Notice that the image borders require special treatment.

```

1: DISTANCETRANSFORM ( $I$ )
    $I$ : binary image of size  $M \times N$ .
   Returns the distance transform of image  $I$ .

   STEP 1—INITIALIZE:
2:  $D \leftarrow$  new distance map of size  $M \times N$ ,  $D(u, v) \in \mathbb{R}$ 
3: for all image coordinates  $(u, v)$  do
4:   if  $I(u, v) = 1$  then
5:      $D(u, v) \leftarrow 0$                                  $\triangleright$  foreground pixel (zero distance)
6:   else
7:      $D(u, v) \leftarrow \infty$                           $\triangleright$  background pixel (infinite distance)

   STEP 2—L $\rightarrow$ R PASS (using distance mask  $M^L = m_i^L$ ):
8: for  $v \leftarrow 1, 2, \dots, N-1$  do                       $\triangleright$  top  $\rightarrow$  bottom
9:   for  $u \leftarrow 1, 2, \dots, M-2$  do                   $\triangleright$  left  $\rightarrow$  right
10:    if  $D(u, v) > 0$  then
11:      Let  $d_1 \leftarrow m_1^L + D(u-1, v)$ 
12:      Let  $d_2 \leftarrow m_2^L + D(u-1, v-1)$ 
13:      Let  $d_3 \leftarrow m_3^L + D(u, v-1)$ 
14:      Let  $d_4 \leftarrow m_4^L + D(u+1, v-1)$ 
15:       $D(u, v) \leftarrow \min(d_1, d_2, d_3, d_4)$ 

   STEP 3—R $\rightarrow$ L PASS (using distance mask  $M^R = m_i^R$ ):
16: for  $v \leftarrow N-2, \dots, 1, 0$  do                       $\triangleright$  bottom  $\rightarrow$  top
17:   for  $u \leftarrow M-2, \dots, 2, 1$  do                   $\triangleright$  right  $\rightarrow$  left
18:     if  $D(u, v) > 0$  then
19:       Let  $d_1 \leftarrow m_1^R + D(u+1, v)$ 
20:       Let  $d_2 \leftarrow m_2^R + D(u+1, v+1)$ 
21:       Let  $d_3 \leftarrow m_3^R + D(u, v+1)$ 
22:       Let  $d_4 \leftarrow m_4^R + D(u-1, v+1)$ 
23:        $D(u, v) \leftarrow \min(D(u, v), d_1, d_2, d_3, d_4)$ 

24: return  $D$ .
```

Algorithm 11.2 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. (11.19). The result obtained with the mask for the Euclidean distance (Eqn. (11.20)) is only an approximation to the actual distance to the nearest foreground pixel, which is nevertheless

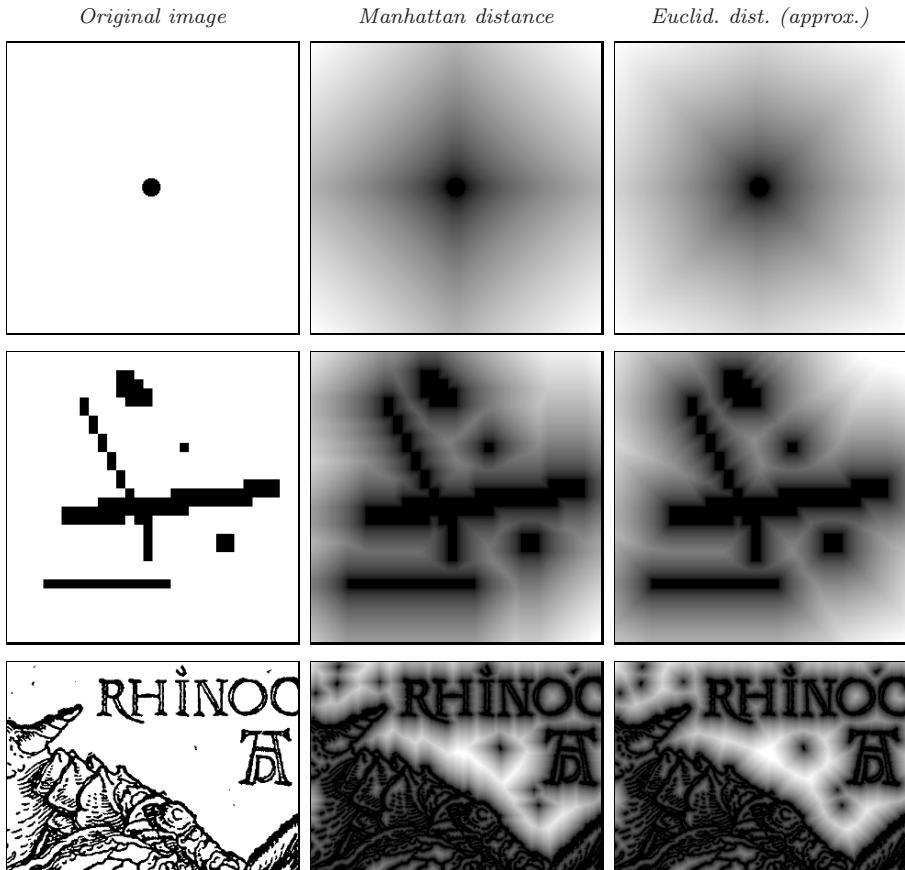


Figure 11.10 Distance transform with the chamfer algorithm: original image with black foreground pixels (left), and results of distance transforms using the Manhattan distance (center) and the Euclidean distance (right). The brightness (scaled to maximum contrast) corresponds to the estimated distance to the nearest foreground pixel.

more accurate than the estimate produced by the Manhattan distance. As demonstrated by the examples in Fig. 11.10, 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 11.3), which include the exact distances to pixels in a larger neighborhood [7]. Furthermore, floating point-operations can be avoided by using distance masks with scaled integer values,

such as the masks

$$M_{E'}^L = \begin{bmatrix} 4 & 3 & 4 \\ 3 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \text{and} \quad M_{E'}^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & 3 \\ 4 & 3 & 4 \end{bmatrix} \quad (11.21)$$

for the Euclidean distance. Compared with the original masks (Eqn. (11.20)), the resulting distance values are scaled by about a factor of 3.

11.2.3 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 [4]) 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. 11.2.1), 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,

$$Q(r, s) = \frac{1}{K} \cdot \sum_{(i,j) \in FG(R)} D(r + i, s + j), \quad (11.22)$$

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. 11.3. 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 ,

$$\mathbf{p}_{\text{opt}} = (r_{\text{opt}}, s_{\text{opt}}) = \underset{(r,s)}{\operatorname{argmin}} Q(r, s). \quad (11.23)$$

The example in Fig. 11.11 demonstrates the difference between direct pixel comparison and chamfer matching using the binary image shown in Fig. 11.8. 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 11.12 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

Algorithm 11.3 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. 11.2). 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 Q that is returned.

1: CHAMFERMATCH (I, R)

I : binary search image of size $w_I \times h_I$

R : binary reference image of size $w_R \times h_R$

Returns a two-dimensional map of match scores.

STEP 1—INITIALIZE:

- 2: Let $D \leftarrow \text{DISTANCETRANSFORM}(I)$ ▷ see Alg. 11.2
- 3: Let $K \leftarrow$ number of foreground pixels in R
- 4: Let $Q \leftarrow$ new *match map* of size $(w_I - w_R + 1) \times (h_I - h_R + 1)$, $Q(r, s) \in \mathbb{R}$

STEP 2—COMPUTE THE MATCH SCORE:

- 5: **for** $r \leftarrow 0 \dots (w_I - w_R)$ **do** ▷ place R at (r, s)
- 6: **for** $s \leftarrow 0 \dots (h_I - h_R)$ **do**
- 7: Get match score for template placed at (r, s) :
- 8: Let $q \leftarrow 0$
- 9: **for** $i \leftarrow 0 \dots (w_R - 1)$ **do**
- 10: **for** $j \leftarrow 0 \dots (h_R - 1)$ **do**
- 11: **if** $R(i, j) = 1$ **then** ▷ foreground pixel in template
- 12: $q \leftarrow q + D(r+i, s+j)$
- 13: $Q(r, s) \leftarrow q/K$

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 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.

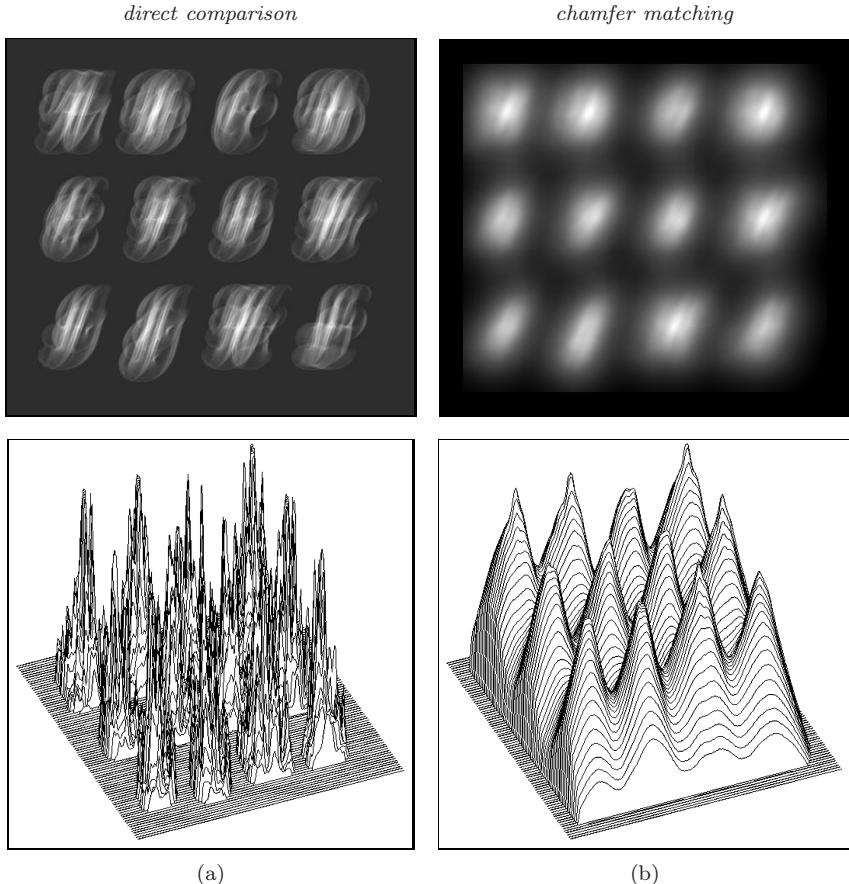


Figure 11.11 Direct pixel comparison vs. chamfer matching (see original images in Fig. 11.8). Unlike the results of the direct pixel comparison (a), the chamfer match score Q (b) is much smoother. It shows distinct peak values in places of high agreement that are easy to track down with local search methods. The match score Q (Eqn. (11.22)) in (b) is shown inverted for easy comparison.

(11.22)) but add up the *squared* distances,

$$Q_{rms}(r, s) = \sqrt{\frac{1}{K} \cdot \sum_{(i,j) \in FG(R)} D^2(r + i, s + i)} \quad (11.24)$$

(“root mean square” of the distances) as the match score between the template R and the current subimage, as suggested in [7]. Also, hierarchical variants of the chamfer method have been proposed [8] to reduce the search effort as well as to increase robustness.

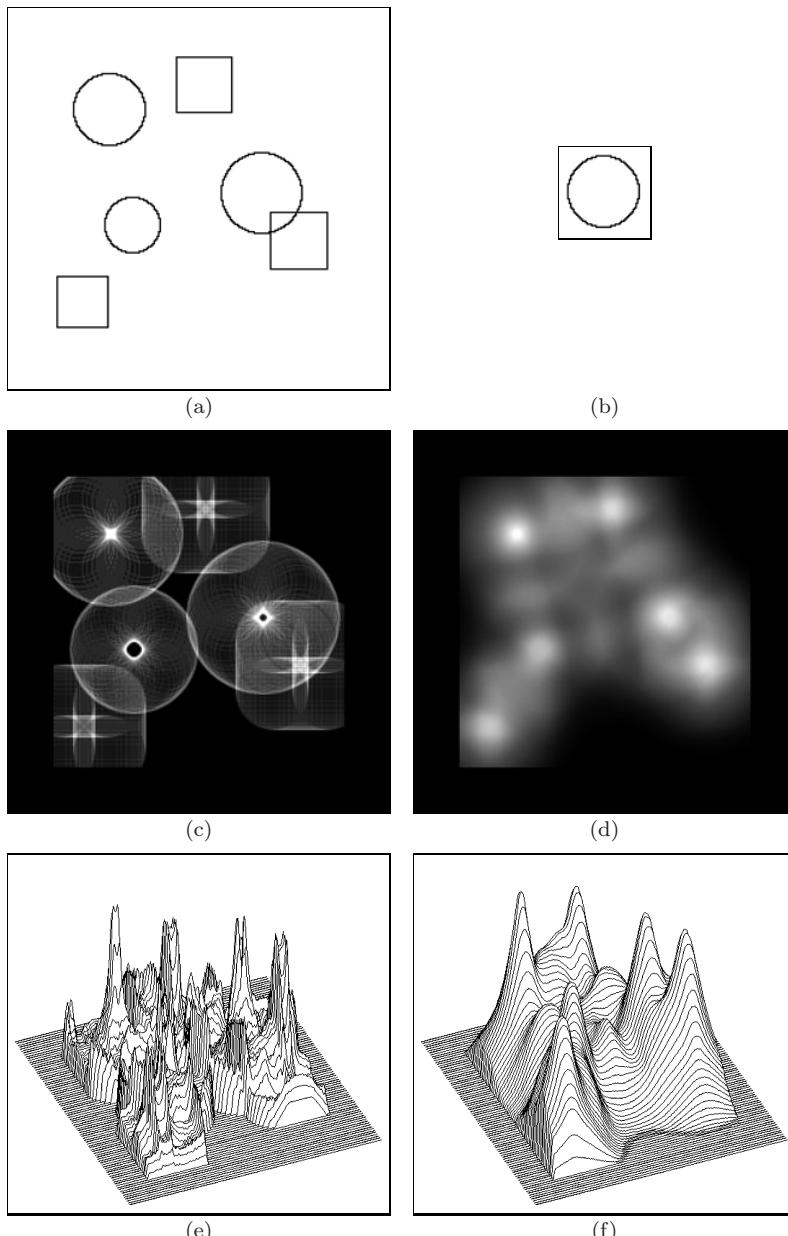


Figure 11.12 Chamfer matching under varying scales. Binary search image with three circles of different diameters and three identical squares (a). The medium-sized circle at the top is used as the template (b). The result from a direct pixel comparison (c, e) and the result from chamfer matching (d, 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).

11.3 Exercises

Exercise 11.1

Implement the chamfer-matching method (Alg. 11.2) for binary images using the Euclidean distance and the Manhattan distance.

Exercise 11.2

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. 11.2), and compute the maximum deviation (in percent).

Exercise 11.3

Modify the chamfer algorithm for computing the distance transform (Alg. 11.2) by replacing the 3×3 pixel Euclidean distance masks (Eqn. (11.20)) with the following masks of size 5×5 pixels:

$$M^L = \begin{bmatrix} \cdot & 2.236 & \cdot & 2.236 & \cdot \\ 2.236 & 1.414 & 1.000 & 1.414 & 2.236 \\ \cdot & 1.000 & \times & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix},$$

$$M^R = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \times & 1.000 & \cdot \\ 2.236 & 1.414 & 1.000 & 1.414 & 2.236 \\ \cdot & 2.236 & \cdot & 2.236 & \cdot \end{bmatrix}.$$

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 11.4

Implement the chamfer-matching technique using (a) the linear summation of distances (Eqn. (11.22)) and (b) the summation of squared distances (Eqn. (11.24)) 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 11.5

Adapt the template-matching method described in Sec. 11.1 for the comparison of RGB color images.

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.

$\{a, b, c, d, \dots\}$	A <i>set</i> ; 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 ($\{\} \}$).
$(a_1, a_2, \dots a_n)$	A <i>vector</i> ; i. e., a fixed-size collection of elements of the same type. $(a_1, a_2, \dots a_n)^T$ denotes the <i>transposed</i> (i. e., column) vector. In programming, vectors are usually implemented as one-dimensional arrays, with elements being referred to by position (index).
$[c_1, c_2, \dots c_m]$	A <i>sequence</i> or <i>list</i> ; 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 <i>Collections</i> framework (see also Vol. 1 [14, Appendix B.2.7]) provides numerous ready-to-use implementations.

$\langle \alpha_1, \alpha_2, \dots, \alpha_k \rangle$	A <i>tuple</i> ; i.e., an ordered list of elements, each possibly of a different type. Tuples are typically implemented as <i>objects</i> (in Java or C++) or <i>structures</i> (in C) with elements being referred to by name.
\neg	Logical “not” operator.
\wedge	Logical “and” operator.
$*$	Linear convolution operator.
\circledast	Linear correlation operator (Sec. 11.1.1).
\oplus	Morphological dilation operator (see Vol. 1 [14, Sec. 7.2.3]).
\ominus	Morphological erosion operator (see Vol. 1 [14, Sec. 7.2.4]).
∂	Partial derivative operator (see Vol. 1 [14, Sec. 6.2.1]). For example, $\frac{\partial f}{\partial x}(x, y)$ denotes the <i>first</i> 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 <i>second</i> derivative, etc.
∇	Gradient. ∇f is the vector of partial derivatives of a multidimensional function f (see Vol. 1 [14, Sec. 6.2.1]).
$[x]$	“Floor” of x , the largest integer $z \in \mathbb{Z}$ smaller than $x \in \mathbb{R}$ (i.e., $z = [x] \leq x$). For example, $[3.141] = 3$, $[-1.2] = -2$.
a	Pixel value (usually $0 \leq a < K$).
$\text{Arctan}(y, x)$	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 Java method <code>Math.atan2(y, x)</code> .
$\text{card}\{\dots\}$	Cardinality (size) of a set, $\text{card } A \equiv A $.
DFT	Discrete Fourier transform (Sec. 7.3).
\mathcal{F}	Continuous Fourier transform (Sec. 7.1.4).
$g(x), g(x, y)$	One- and two-dimensional <i>continuous</i> functions ($x, y \in \mathbb{R}$).
$g(u), g(u, v)$	One- and two-dimensional <i>discrete</i> functions ($u, v \in \mathbb{Z}$).
$G(m), G(m, n)$	One- and two-dimensional discrete Fourier spectra ($m, n \in \mathbb{Z}$).

$h(i)$	Histogram of an image at pixel value (or bin) i (see Vol. 1 [14, Sec. 3.1]).
$H(i)$	Cumulative histogram of an image at pixel value (or bin) i (see Vol. 1 [14, Sec. 3.6]).
$I(u, v)$	Intensity value of the image I at (integer) position (u, v) .
i	Imaginary unit, $i^2 = -1$ (see Sec. A.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: $(a \text{ mod } b)$ is the remainder of the integer division a/b .
$p(i)$	Probability density function (see Vol. 1 [14, Sec. 4.6.1]).
$P(i)$	Probability distribution function or cumulative probability density (see Vol. 1 [14, Sec. 4.6.1]).
Q	Quadrilateral (Sec. 10.1.4).
$\text{round}(x)$	Rounding function: rounds x to the nearest integer. $\text{round}(x) = \lfloor x + 0.5 \rfloor$ (Sec. 10.3.1).
$\text{truncate}(x)$	Truncation function: truncates x toward zero to the closest integer. For example, $\text{truncate}(3.141) = 3$, $\text{truncate}(-2.5) = -2$.
S_1	Unit square (Sec. 10.1.4).

A.2 Set Operators

$ A $	The size (number of elements) of the set A (equivalent to $\text{card } A$).
$\forall_x \dots$	“All” quantifier (for all x, \dots).
$\exists_x \dots$	“Exists” quantifier (there is some x for which \dots).
\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:

$$z = a + ib, \quad z, i \in \mathbb{C}, \quad a, b \in \mathbb{R}, \quad i^2 = -1, \quad (\text{A.1})$$

$$z^* = a - ib \quad (\text{conjugate complex}), \quad (\text{A.2})$$

$$sz = sa + isb, \quad s \in \mathbb{R}, \quad (\text{A.3})$$

$$|z| = \sqrt{a^2 + b^2}, \quad |sz| = s|z|, \quad (\text{A.4})$$

$$z = a + ib$$

$$= |z| \cdot (\cos \psi + i \sin \psi) \quad (\text{A.5})$$

$$= |z| \cdot e^{i\psi}, \quad \text{where } \psi = \tan^{-1}(b/a), \quad (\text{A.6})$$

$$\operatorname{Re}(a + ib) = a, \quad \operatorname{Re}(e^{i\varphi}) = \cos \varphi, \quad (\text{A.7})$$

$$\operatorname{Im}(a + ib) = b, \quad \operatorname{Im}(e^{i\varphi}) = \sin \varphi, \quad (\text{A.8})$$

$$e^{i\varphi} = \cos \varphi + i \cdot \sin \varphi, \quad (\text{A.9})$$

$$e^{-i\varphi} = \cos \varphi - i \cdot \sin \varphi, \quad (\text{A.10})$$

$$\cos(\varphi) = \frac{1}{2} \cdot (e^{i\varphi} + e^{-i\varphi}), \quad (\text{A.11})$$

$$\sin(\varphi) = \frac{1}{2i} \cdot (e^{i\varphi} - e^{-i\varphi}), \quad (\text{A.12})$$

Arithmetic operations:

$$z_1 = (a_1 + ib_1) = |z_1| e^{i\varphi_1},$$

$$z_2 = (a_2 + ib_2) = |z_2| e^{i\varphi_2},$$

$$z_1 + z_2 = (a_1 + a_2) + i(b_1 + b_2), \quad (\text{A.13})$$

$$z_1 \cdot z_2 = (a_1 a_2 - b_1 b_2) + i(a_1 b_2 + b_1 a_2) \quad (\text{A.14})$$

$$= |z_1| \cdot |z_2| \cdot e^{i(\varphi_1 + \varphi_2)}, \quad (\text{A.15})$$

$$\frac{z_1}{z_2} = \frac{a_1 a_2 + b_1 b_2}{a_2^2 + b_2^2} + i \frac{a_2 b_1 - a_1 b_2}{a_2^2 + b_2^2} \quad (\text{A.16})$$

$$= \frac{|z_1|}{|z_2|} \cdot e^{i(\varphi_1 - \varphi_2)}. \quad (\text{A.17})$$

B

Source Code

B.1 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. 2.2. 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. 285): a class representing a contour object.
- `BinaryRegion` (p. 286): a class representing a binary region object.
- `ContourTracer` (p. 287): the actual region labeler and contour tracer. This class is instantiated to create a region labeler for a given image.
- `ContourOverlay` (p. 292): a class for displaying contours as vector graphics on top of images.

B.1.1 `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 [17].
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,
55             innerContours);
56         new ImageWindow(im2, cc);
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
```

B.1.2 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         }
44     }
45 }
```

```

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

```

B.1.3 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 numberOfWorks = 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.numberOfWorkPixels;
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         numberOfWorkPixels = numberOfWorkPixels + 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 (numberOfWorkPixels > 0){
57             xc = x_sum / numberOfWorkPixels;
58             yc = y_sum / numberOfWorkPixels;
59         }
60     }
61
62 } // end of class BinaryRegion
```

B.1.4 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 int findNextPoint (Point pt, int dir) {
117     // starts at Point pt in direction dir, returns the
118     // final tracing direction, and modifies pt
119     final int[][] delta = {
120         { 1,0}, { 1, 1}, {0, 1}, {-1, 1},
121         {-1,0}, {-1,-1}, {0,-1}, { 1,-1}};
122     for (int i = 0; i < 7; i++) {
123         int x = pt.x + delta[dir][0];
124         int y = pt.y + delta[dir][1];
125         if (pixelArray[y][x] == BACKGROUND) {
126             // mark surrounding background pixels
127             labelArray[y][x] = -1;
128             dir = (dir + 1) % 8;
129         }
130         else { // found a nonbackground pixel
131             pt.x = x; pt.y = y;
132             break;
133         }
134     }
135     return dir;
136 }
137 }

138 void findAllContours() {
139     outerContours = new ArrayList<Contour>(50);
140     innerContours = new ArrayList<Contour>(50);
141     int label = 0; // current label
142
143     // scan top to bottom, left to right
144     for (int v = 1; v < pixelArray.length-1; v++) {
145         label = 0; // no label
146         for (int u = 1; u < pixelArray[v].length-1; u++) {
147
148             if (pixelArray[v][u] == FOREGROUND) {
149                 if (label != 0) { // keep using the same label
150                     labelArray[v][u] = label;
151                 }
152             else {
153                 label = labelArray[v][u];
154                 if (label == 0) {
155                     // unlabeled—new outer contour
156                     regionId = regionId + 1;
157                     label = regionId;
158                 }
159             }
160         }
161     }
162 }
```

```
159         Contour oc = traceOuterContour(u, v, label);
160         outerContours.add(oc);
161         labelArray[v][u] = label;
162     }
163 }
164 }
165 else { // background pixel
166     if (label != 0) {
167         if (labelArray[v][u] == 0) {
168             // unlabeled—new inner contour
169             Contour ic = traceInnerContour(u-1, v, label);
170             innerContours.add(ic);
171         }
172         label = 0;
173     }
174 }
175 }
176 }
177 // shift back to original coordinates
178 Contour.moveContoursBy (outerContours, -1, -1);
179 Contour.moveContoursBy (innerContours, -1, -1);
180 }
181
182
183 // creates a container of BinaryRegion objects
184 // collects the region pixels from the label image
185 // and computes the statistics for each region
186 void collectRegions() {
187     int maxLabel = this.regionId;
188     int startLabel = 1;
189     BinaryRegion[] regionArray =
190         new BinaryRegion[maxLabel + 1];
191     for (int i = startLabel; i <= maxLabel; i++) {
192         regionArray[i] = new BinaryRegion(i);
193     }
194     for (int v = 0; v < height; v++) {
195         for (int u = 0; u < width; u++) {
196             int lb = labelArray[v][u];
197             if (lb >= startLabel && lb <= maxLabel
198                 && regionArray[lb] != null) {
199                 regionArray[lb].addPixel(u, v);
200             }
201         }
202     }
203
204     // create a list of regions to return, collect nonempty regions
205     List<BinaryRegion> regionList =
206         new LinkedList<BinaryRegion>();
207     for (BinaryRegion r: regionArray) {
208         if (r != null && r.getSize() > 0) {
209             r.update(); // compute the statistics for this region
210             regionList.add(r);
211         }
212 }
```

```
212     }
213     allRegions = regionList;
214 }
215
216 } // end of class ContourTracer
```

B.1.5 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 *
22         2.5f};
22     static float[] innerDashing = {strokeWidth * 0.5f, strokeWidth *
23         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,
48                         RenderingHints.VALUE_ANTIALIAS_ON);
49
50     // scale and move overlay to the pixel centers
51     double mag = this.getMagnification();
52     g2d.scale(mag, mag);
53     g2d.translate(0.5-this/srcRect.x, 0.5-this/srcRect.y);
54
55     if (DRAW_CONTOURS) {
56         Stroke solidStroke = new BasicStroke
57             (strokeWidth, capsstyle, joinstyle);
58         Stroke dashedStrokeOuter = new BasicStroke
59             (strokeWidth, capsstyle, joinstyle, 1.0f, outerDashing, 0.0
60              f);
61         Stroke dashedStrokeInner = new BasicStroke
62             (strokeWidth, capsstyle, joinstyle, 1.0f, innerDashing, 0.0
63              f);
64
65         if (outerContourShapes != null)
66             drawShapes(outerContourShapes, g2d, solidStroke,
67                         dashedStrokeOuter, outerColor);
68         if (innerContourShapes != null)
69             drawShapes(innerContourShapes, g2d, solidStroke,
70                         dashedStrokeInner, innerColor);
71     }
72 }
73
74 void drawShapes(Shape[] shapes, Graphics2D g2d,
75                 Stroke solidStrk, Stroke dashedStrk, Color col) {
76     g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
77                         RenderingHints.VALUE_ANTIALIAS_ON);
78     g2d.setColor(col);
79     for (int i = 0; i < shapes.length; i++) {
80         Shape s = shapes[i];
81         if (s instanceof Polygon)
82             g2d.setStroke(dashedStrk);
83         else
84             g2d.setStroke(solidStrk);
85         g2d.draw(s);
86     }
87 }
88
89 } // end of class ContourOverlay
```

B.2 Harris Corner Detector

The following Java source code represents a complete implementation of the Harris corner detector, as described in Ch. 4. It consists of the following classes (files):

- `Harris_Corner_Plugin`: a sample ImageJ plugin that demonstrates the use of the corner detector.
- `Corner` (p. 295): a class representing an individual corner object.
- `HarrisCornerDetector` (p. 296): the actual corner detector. This class is instantiated to create a corner detector for a given image.

B.2.1 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",
43             IJ.getInstance());
44     float def_alpha = HarrisCornerDetector.DEFAULT_ALPHA;
45     dlg.addNumericField("Alpha (default: "+def_alpha+")", alpha, 3)
46         ;
47     int def_threshold = HarrisCornerDetector.DEFAULT_THRESHOLD;
48     dlg.addNumericField("Threshold (default: "+def_threshold+")",
49             threshold, 0);
50     dlg.addNumericField("Max. points (0 = show all)", nmax, 0);
51     dlg.showDialog();
52     if(dlg.wasCanceled())
53         return false;
54     if(dlg.invalidNumber()) {
55         IJ.showMessage("Error", "Invalid input number");
56         return false;
57     }
58     alpha = (float) dlg.getNextNumber();
59     threshold = (int) dlg.getNextNumber();
60     nmax = (int) dlg.getNextNumber();
61     return true;
62 }
63 } // end of class Harris_Corner_Plugin
```

B.2.2 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 }
```

```

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

```

B.2.3 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
27         ,0.234375f,0.09375f,0.01563f};
28     //=[1,6,15,20,15,6,1]/64
29     ImageProcessor ipOrig;
30     FloatProcessor A;
31     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 ) {
192     int mode = Blitter.MULTIPLY;
193     fp1.copyBits(fp2, 0, 0, mode);
194     return fp1;
195 }
196
197 static boolean isLocalMax (FloatProcessor fp,int u,int v) {
198     int w = fp.getWidth();
199     int h = fp.getHeight();
200     if (u<=0 || u>=w-1 || v<=0 || v>=h-1)
201         return false;
202     else {
203         float[] pix = (float[]) fp.getPixels();
204         int i0 = (v-1)*w+u, i1 = v*w+u, i2 = (v+1)*w+u;
205         float cp = pix[i1];
206         return
207             cp > pix[i0-1] && cp > pix[i0] && cp > pix[i0+1] &&
208             cp > pix[i1-1] && cp > pix[i1+1] &&
209             cp > pix[i2-1] && cp > pix[i2] && cp > pix[i2+1] ;
210     }
211 }
212 } // end of class HarrisCornerDetector
```

B.3 Median-Cut Color Quantization

This is an implementation of Heckbert’s median-cut color quantization algorithm [32], as described in Sec. 5.2 (Alg. 5.1–5.3). Unlike in the original algorithm, no initial uniform (scalar) quantization is used for reducing the number of image colors. Instead, all colors contained in the original image are considered in the quantization process. After the set of representative colors has been found, each image color is mapped to the closest representative in RGB color space using the Euclidean distance.

B.3.1 ColorQuantizer (Interface)

This is a general interface for all color quantizers.

```
1 package color;
2
3 import ij.process.ByteProcessor;
4 import ij.process.ColorProcessor;
5
6 public interface ColorQuantizer {
7     public abstract ByteProcessor quantizeImage(ColorProcessor cp);
8     public abstract int[] quantizeImage(int[] origPixels);
9     public abstract int countQuantizedColors();
10
11 }
```

B.3.2 MedianCutQuantizer (Class)

This class contains the main functionality of the median-cut quantizer. Figure B.1 illustrates the key data structures involved and their relationships. The classes `ColorNode` and `ColorBox` are implemented as nested classes inside `MedianCutQuantizer`. Also, notice the use of the nested enumeration class `ColorDimension` for implementing the constants `RED`, `GREEN`, and `BLUE` and the associated comparator methods.

```
1 package color;
2
3 import ij.process.ByteProcessor;
4 import ij.process.ColorProcessor;
5
6 import java.awt.image.IndexColorModel;
7 import java.util.ArrayList;
8 import java.util.Arrays;
9 import java.util.Comparator;
10 import java.util.List;
11
12 public class MedianCutQuantizer implements ColorQuantizer {
13
```

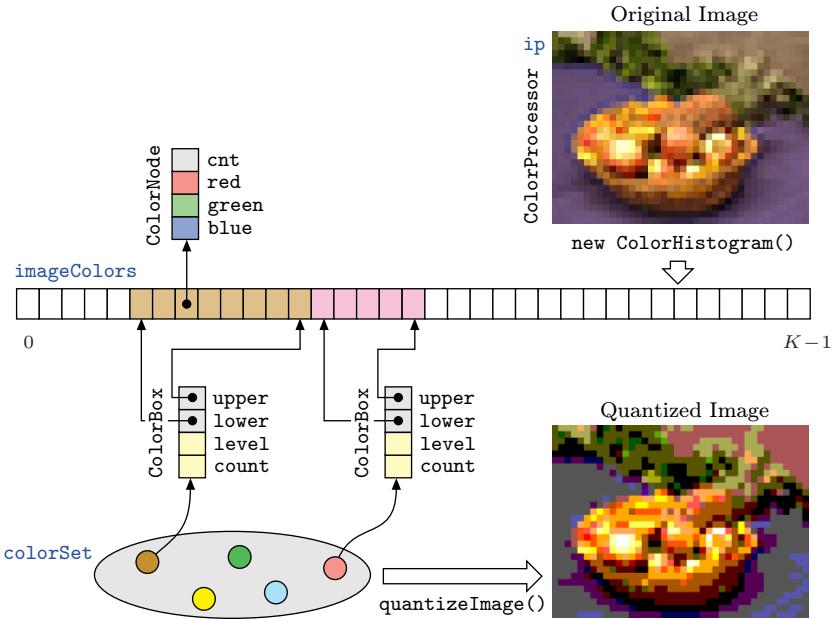


Figure B.1 Median-cut data structures. Initially, a new `ColorHistogram` is computed for the original color image (`ip` of type `ColorProcessor`). The resulting array `imageColors` of size K corresponds to the unique colors ($C = \{c_1, c_2, \dots, c_K\}$ in Alg. 5.1) contained in the original RGB image. Each cell of `imageColors` refers to a `ColorNode` object (c_i) that holds the associated color (red, green, blue) and its frequency (`cnt`) in the image. Each `colorBox` object (corresponding to a color box b in Alg. 5.1) selects a contiguous range of image colors, bounded by the indices `lower` and `upper`. The ranges of elements in `imageColors`, indexed by different `colorBox` objects, never overlap. Each element in `imageColors` is contained in exactly one `colorBox`; i.e., the color boxes held in `colorSet` (\mathcal{B} in Alg. 5.1) form a partitioning of `imageColors` (`colorSet` is implemented as a list of `ColorBox` objects). To split a particular `colorBox` along a color dimension $d = \text{Red}, \text{Green}, \text{or Blue}$, the corresponding subrange of elements in `imageColors` is *sorted* with the property `red`, `green`, or `blue`, respectively, as the sorting key. In Java, this is quite easy to implement using the standard `Arrays.sort()` utility method and a dedicated `Comparator` object for each color dimension. Finally, the method `quantizeImage()` replaces each pixel in `ip` by the closest color in `colorSet`.

```

14  private ColorNode[] imageColors = null; // original image colors
15  private ColorNode[] quantColors = null; // quantized colors
16
17  public MedianCutQuantizer(ColorProcessor ip, int Kmax) {
18      this((int[]) ip.getPixels(), Kmax);
19  }
20
21  public MedianCutQuantizer(int[] pixels, int Kmax) {
22      quantColors = findRepresentativeColors(pixels, Kmax);
23  }
24
25  public int countQuantizedColors() {

```

```
26     return quantColors.length;
27 }
28
29 public ColorNode[] getQuantizedColors() {
30     return quantColors;
31 }
32
33 ColorNode[] findRepresentativeColors(int[] pixels, int Kmax) {
34     ColorHistogram colorHist = new ColorHistogram(pixels);
35     int K = colorHist.getNumberOfColors();
36     ColorNode[] rCols = null;
37
38     imageColors = new ColorNode[K];
39     for (int i = 0; i < K; i++) {
40         int rgb = colorHist.getColor(i);
41         int cnt = colorHist.getCount(i);
42         imageColors[i] = new ColorNode(rgb, cnt);
43     }
44
45     if (K <= Kmax) // image has fewer colors than Kmax
46         rCols = imageColors;
47     else {
48         ColorBox initialBox = new ColorBox(0, K-1, 0);
49         List<ColorBox> colorSet = new ArrayList<ColorBox>();
50         colorSet.add(initialBox);
51         int k = 1;
52         boolean done = false;
53         while (k < Kmax && !done) {
54             ColorBox nextBox = findBoxToSplit(colorSet);
55             if (nextBox != null) {
56                 ColorBox newX = nextBox.splitBox();
57                 colorSet.add(newX);
58                 k = k + 1;
59             } else {
60                 done = true;
61             }
62         }
63         rCols = averageColors(colorSet);
64     }
65     return rCols;
66 }
67
68 public int[] quantizeImage(int[] origPixels) {
69     int[] qantPixels = origPixels.clone();
70     for (int i = 0; i < origPixels.length; i++) {
71         ColorNode color = findClosestColor(origPixels[i]);
72         qantPixels[i] = color.rgb;
73     }
74     return qantPixels;
75 }
76
77 public ByteProcessor quantizeImage(ColorProcessor cp) {
78     if (countQuantizedColors() > 256)
```

```
79      throw new Error("cannot index to more than 256 colors");
80      int w = cp.getWidth();
81      int h = cp.getHeight();
82      int[] origPixels = (int[]) cp.getPixels();
83      byte[] idxPixels = new byte[origPixels.length];
84
85      for (int i = 0; i < origPixels.length; i++) {
86          idxPixels[i] = (byte) findClosestColorIndex(origPixels[i]);
87      }
88
89      IndexColorModel idxCm = makeIndexColorModel();
90      return new ByteProcessor(w, h, idxPixels, idxCm);
91  }
92
93  IndexColorModel makeIndexColorModel() {
94      int nColors = countQuantizedColors();
95      byte[] rMap = new byte[nColors];
96      byte[] gMap = new byte[nColors];
97      byte[] bMap = new byte[nColors];
98      for (int i=0; i<nColors; i++) {
99          rMap[i] = (byte) quantColors[i].red;
100         gMap[i] = (byte) quantColors[i].grn;
101         bMap[i] = (byte) quantColors[i].blu;
102     }
103     return new IndexColorModel(8, nColors, rMap, gMap, bMap);
104  }
105
106  ColorNode findClosestColor (int rgb) {
107      int idx = findClosestColorIndex(rgb);
108      return quantColors[idx];
109  }
110
111  int findClosestColorIndex (int rgb) {
112      int red = ((rgb & 0xFF0000) >> 16);
113      int grn = ((rgb & 0xFF00) >> 8);
114      int blu = (rgb & 0xFF);
115      int minIdx = 0;
116      int minDistance = Integer.MAX_VALUE;
117      for (int i=0; i<quantColors.length; i++) {
118          ColorNode color = quantColors[i];
119          int d2 = color.distance2(red, grn, blu);
120          if (d2 < minDistance) {
121              minDistance = d2;
122              minIdx = i;
123          }
124      }
125      return minIdx;
126  }
127
128  private ColorBox findBoxToSplit(List<ColorBox> colorBoxes) {
129      ColorBox boxToSplit = null;
130      // from the set of splittable color boxes
131      // select the one with the minimum level
```

```
132     int minLevel = Integer.MAX_VALUE;
133     for (ColorBox box : colorBoxes) {
134         if (box.colorCount() >= 2) { // box can be split
135             if (box.level < minLevel) {
136                 boxToSplit = box;
137                 minLevel = box.level;
138             }
139         }
140     }
141     return boxToSplit;
142 }
143
144 private ColorNode[] averageColors(List<ColorBox> colorBoxes) {
145     int n = colorBoxes.size();
146     ColorNode[] avgColors = new ColorNode[n];
147     int i = 0;
148     for (ColorBox box : colorBoxes) {
149         avgColors[i] = box.getAverageColor();
150         i = i + 1;
151     }
152     return avgColors;
153 }
154
155 // ----- class ColorNode -----
156
157 class ColorNode {
158     private int rgb;
159     private int red, grn, blu;
160     private int cnt;
161
162     ColorNode (int rgb, int cnt) {
163         this.rgb = (rgb & 0xFFFFFFFF);
164         this.red = (rgb & 0xFF0000) >> 16;
165         this.grn = (rgb & 0xFF00) >> 8;
166         this.blu = (rgb & 0xFF);
167         this.cnt = cnt;
168     }
169
170     ColorNode (int red, int grn, int blu, int cnt) {
171         this.rgb = ((red & 0xff) << 16) | ((grn & 0xff) << 8) | blu & 0
172             xff;
173         this.red = red;
174         this.grn = grn;
175         this.blu = blu;
176         this.cnt = cnt;
177     }
178
179     int distance2 (int red, int grn, int blu) {
180         // returns the squared distance between (red, grn, blu)
181         // and this color
182         int dr = this.red - red;
183         int dg = this.grn - grn;
184         int db = this.blu - blu;
```

```
184     return dr*dr + dg*dg + db*db;
185 }
186
187 public String toString() {
188     String s = this.getClass().getSimpleName();
189     s = s + " red=" + red + " green=" + grn + " blue=" + blu + "
190         count=" + cnt;
191     return s;
192 }
193
194 // ----- class ColorBox -----
195
196 class ColorBox {
197     int lower = 0; // lower index into 'imageColors'
198     int upper = -1; // upper index into 'imageColors'
199     int level; // split level of this color box
200     int count = 0; // number of pixels represented by thos color box
201     int rmin, rmax; // range of contained colors in red dimension
202     int gmin, gmax; // range of contained colors in green dimension
203     int bmin, bmax; // range of contained colors in blue dimension
204
205     ColorBox(int lower, int upper, int level) {
206         this.lower = lower;
207         this.upper = upper;
208         this.level = level;
209         this.trim();
210     }
211
212     int colorCount() {
213         return upper - lower;
214     }
215
216     void trim() {
217         // recompute the boundaries of this color box
218         rmin = 255; rmax = 0;
219         gmin = 255; gmax = 0;
220         bmin = 255; bmax = 0;
221         count = 0;
222         for (int i = lower; i <= upper; i++) {
223             ColorNode color = imageColors[i];
224             count = count + color.cnt;
225             int r = color.red;
226             int g = color.grn;
227             int b = color.blu;
228             if (r > rmax) rmax = r;
229             if (r < rmin) rmin = r;
230             if (g > gmax) gmax = g;
231             if (g < gmin) gmin = g;
232             if (b > bmax) bmax = b;
233             if (b < bmin) bmin = b;
234         }
235     }
}
```

```
236 // Split this color box at the median point along its
237 // longest color dimension
238 ColorBox splitBox() {
239     if (this.colorCount() < 2) // this box cannot be split
240         return null;
241     else {
242         // find longest dimension of this box:
243         ColorDimension dim = getLongestColorDimension();
244
245         // find median along dim
246         int med = findMedian(dim);
247
248         // now split this box at the median return the resulting new
249         // box.
250         int nextLevel = level + 1;
251         ColorBox newBox = new ColorBox(med + 1, upper, nextLevel);
252         this.upper = med;
253         this.level = nextLevel;
254         this.trim();
255         return newBox;
256     }
257 }
258 }
259
260 // Find longest dimension of this color box (RED, GREEN, or BLUE)
261 ColorDimension getLongestColorDimension() {
262     int rLength = rmax - rmin;
263     int gLength = gmax - gmin;
264     int bLength = bmax - bmin;
265     if (bLength >= rLength && bLength >= gLength)
266         return ColorDimension.BLUE;
267     else if (gLength >= rLength && gLength >= bLength)
268         return ColorDimension.GREEN;
269     else return ColorDimension.RED;
270 }
271
272 // Find the position of the median in RGB space along
273 // the red, green or blue dimension, respectively.
274 int findMedian(ColorDimension dim) {
275     // sort color in this box along dimension dim:
276     Arrays.sort(imageColors, lower, upper+1, dim.comparator);
277     // find the median point:
278     int half = count / 2;
279     int nPixels, median;
280     for (median = lower, nPixels = 0; median < upper; median++) {
281         nPixels = nPixels + imageColors[median].cnt;
282         if (nPixels >= half)
283             break;
284     }
285     return median;
286 }
287
288 ColorNode getAverageColor() {
```

```
289     int rSum = 0;
290     int gSum = 0;
291     int bSum = 0;
292     int n = 0;
293     for (int i = lower; i <= upper; i++) {
294         ColorNode ci = imageColors[i];
295         int cnt = ci.cnt;
296         rSum = rSum + cnt * ci.red;
297         gSum = gSum + cnt * ci.grn;
298         bSum = bSum + cnt * ci.blu;
299         n = n + cnt;
300     }
301     double nd = n;
302     int avgRed = (int) (0.5 + rSum / nd);
303     int avgGrn = (int) (0.5 + gSum / nd);
304     int avgBla = (int) (0.5 + bSum / nd);
305     return new ColorNode(avgRed, avgGrn, avgBla, n);
306 }
307
308 public String toString() {
309     String s = this.getClass().getSimpleName();
310     s = s + " lower=" + lower + " upper=" + upper;
311     s = s + " count=" + count + " level=" + level;
312     s = s + " rmin=" + rmin + " rmax=" + rmax;
313     s = s + " gmin=" + gmin + " gmax=" + gmax;
314     s = s + " bmin=" + bmin + " bmax=" + bmax;
315     s = s + " bmin=" + bmin + " bmax=" + bmax;
316     return s;
317 }
318 }
319
320 // — color dimensions -----
321 // The main purpose of this enumeration class is to
322 // associate the color dimensions RED, GREEN, BLUE
323 // with the corresponding comparators.
324
325 enum ColorDimension {
326     RED (new Comparator<ColorNode>() {
327         public int compare(ColorNode colA, ColorNode colB) {
328             return colA.red - colB.red;
329         }),
330         GREEN (new Comparator<ColorNode>() {
331             public int compare(ColorNode colA, ColorNode colB) {
332                 return colA.grn - colB.grn;
333             }),
334             BLUE (new Comparator<ColorNode>() {
335                 public int compare(ColorNode colA, ColorNode colB) {
336                     return colA.blu - colB.blu;
337                 });
338
339     public final Comparator<ColorNode> comparator;
340
341     ColorDimension(Comparator<ColorNode> cmp) {
```

```
342     this.comparator = cmp;
343   }
344 }
345
346 //---- utility methods ----
347
348 void listColorNodes(){
349   int i = 0;
350   for (ColorNode color : quantColors) {
351     IJ.write(" color " + i + ": " + color.toString());
352     i++;
353   }
354 }
355
356
357 } //class MedianCut
```

B.3.3 ColorHistogram (Class)

This utility class is used to compute the histogram of (unique) image colors.

```
1 package color;
2
3 import ij.process.ColorProcessor;
4 import java.util.Arrays;
5
6 public class ColorHistogram {
7   int colorArray[] = null;
8   int countArray[] = null;
9
10  public ColorHistogram(ColorProcessor ip) {
11    this((int[]) ip.getPixels());
12  }
13
14  public ColorHistogram(int[] pixelsOrig) {
15    int N = pixelsOrig.length;
16    int[] pixelsCpy = new int[N];
17    for (int i = 0; i < N; i++) {
18      // remove possible alpha components
19      pixelsCpy[i] = 0xFFFFFFFF & pixelsOrig[i];
20    }
21    Arrays.sort(pixelsCpy);
22
23    // count unique colors:
24    int k = -1; // current color index
25    int curColor = -1;
26    for (int i = 0; i < pixelsCpy.length; i++) {
27      if (pixelsCpy[i] != curColor) {
28        k++;
29        curColor = pixelsCpy[i];
30      }
31    }
32    int nColors = k+1;
```

```

33
34     // tabulate and count unique colors:
35     colorArray = new int[nColors];
36     countArray = new int[nColors];
37     k = -1; // current color index
38     curColor = -1;
39     for (int i = 0; i < pixelsCopy.length; i++) {
40         if (pixelsCopy[i] != curColor) { // new color
41             k++;
42             curColor = pixelsCopy[i];
43             colorArray[k] = curColor;
44             countArray[k] = 1;
45         }
46         else {
47             countArray[k]++;
48         }
49     }
50 }
51
52 public int[] getColorArray() {
53     return colorArray;
54 }
55
56 public int[] getCountArray() {
57     return countArray;
58 }
59
60 public int getNumberOfColors() {
61     if (colorArray == null)
62         return 0;
63     else
64         return colorArray.length;
65 }
66
67 public int getColor(int index) {
68     return this.colorArray[index];
69 }
70
71 public int getCount(int index) {
72     return this.countArray[index];
73 }
74 }

```

B.3.4 Median_Cut_Quantization (Class)

This simple ImageJ plugin demonstrates the use of the `MedianCutQuantizer` class to quantize color images. The quantization process has two steps:

- First, a `ColorQuantizer` object is created from a given image using one of the constructor methods provided.
- Then this `ColorQuantizer` can be used to quantize the original image or

any other image using the same set of representative colors (color table).

```
1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ByteProcessor;
4 import ij.process.ColorProcessor;
5 import ij.process.ImageProcessor;
6 import color.ColorQuantizer;
7 import color.MedianCutQuantizer;
8
9 public class Median_Cut_Quantization implements PlugInFilter {
10    // specify the desired number of colors:
11    static int NCOLORS = 32;
12
13    public int setup(String arg, ImagePlus imp) {
14        return DOES_RGB + NO_CHANGES;
15    }
16
17    public void run(ImageProcessor ip) {
18        ColorProcessor cp = (ColorProcessor) ip.convertToRGB();
19
20        // create a quantizer object
21        ColorQuantizer quantizer = new MedianCutQuantizer(cp, NCOLORS);
22        int qColors = quantizer.countQuantizedColors();
23
24        // quantize to an indexed image
25        ByteProcessor idxIp = quantizer.quantizeImage(cp);
26        ImagePlus idxIm = new ImagePlus("Quantized Index Image (" +
27            qColors + " colors)", idxIp);
28        idxIm.show();
29
30        // quantize to an RGB image
31        int[] rgbPixels = quantizer.quantizeImage((int[]) cp.getPixels())
32            ;
33        ImageProcessor rgbIp =
34            new ColorProcessor(cp.getWidth(), cp.getHeight(), rgbPixels);
35        ImagePlus rgbIm =
36            new ImagePlus("Quantized RGB Image (" + qColors + " colors)" ,
37                rgbIp);
38        rgbIm.show();
39    }
40}
```


Bibliography

- [1] Adobe Systems. “Adobe RGB (1998) Color Space Specification” (2005). <http://www.adobe.com/digitalimag/pdfs/AdobeRGB1998.pdf>.
- [2] D. H. BALLARD AND C. M. BROWN. “Computer Vision”. Prentice Hall, Englewood Cliffs, NJ (1982).
- [3] 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).
- [4] 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.
- [5] R. E. BLAHUT. “Fast Algorithms for Digital Signal Processing”. Addison-Wesley, Reading, MA (1985).
- [6] C. BOOR. “A Practical Guide to Splines”. Springer-Verlag, New York (2001).
- [7] G. BORGEFORS. Distance transformations in digital images. *Computer Vision, Graphics and Image Processing* **34**, 344–371 (1986).
- [8] G. BORGEFORS. Hierarchical chamfer matching: a parametric edge matching algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **10**(6), 849–865 (1988).
- [9] J. E. BRESENHAM. A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM* **20**(2), 100–106 (1977).

- [10] E. O. BRIGHAM. “The Fast Fourier Transform and Its Applications”. Prentice Hall, Englewood Cliffs, NJ (1988).
- [11] I. N. BRONSSTEIN AND K. A. SEMENDYAYEV. “Handbook of Mathematics”. Springer-Verlag, Berlin, third ed. (2007).
- [12] H. BUNKE AND P. S. P. WANG, editors. “Handbook of Character Recognition and Document Image Analysis”. World Scientific, Singapore (2000).
- [13] W. BURGER AND M. J. BURGE. “ImageJ Short Reference for Java Developers” (2008). <http://www.imagingbook.com>.
- [14] W. BURGER AND M. J. BURGE. “Principles of Image Processing—Fundamental Techniques”. Springer, New York (2009).
- [15] K. R. CASTLEMAN. “Digital Image Processing”. Prentice Hall, Upper Saddle River, NJ (1995).
- [16] E. CATMULL AND R. ROM. A class of local interpolating splines. In R. E. BARNHILL AND R. F. RIESENFIELD, editors, “Computer Aided Geometric Design”, pp. 317–326. Academic Press, New York (1974).
- [17] 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.
- [18] 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).
- [19] P. R. COHEN AND E. A. FEIGENBAUM. “The Handbook of Artificial Intelligence”. William Kaufmann, Los Altos, CA (1982).
- [20] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN. “Introduction to Algorithms”. MIT Press, Cambridge, MA, second ed. (2001).
- [21] R. O. DUDA, P. E. HART, AND D. G. STORK. “Pattern Classification”. Wiley, New York (2001).
- [22] 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).
- [23] 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).
- [24] D. A. FORSYTH AND J. PONCE. “Computer Vision—A Modern Approach”. Prentice Hall, Englewood Cliffs, NJ (2003).
 - [25] H. FREEMAN. Computer processing of line drawing images. *ACM Computing Surveys* **6**(1), 57–97 (March 1974).
 - [26] 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).
 - [27] A. S. GLASSNER. “Principles of Digital Image Synthesis”. Morgan Kaufmann Publishers, San Francisco (1995).
 - [28] R. C. GONZALEZ AND R. E. WOODS. “Digital Image Processing”. Addison-Wesley, Reading, MA (1992).
 - [29] P. GREEN. Colorimetry and colour differences. In P. GREEN AND L. MACDONALD, editors, “Colour Engineering”, ch. 3, pp. 40–77. Wiley, New York (2002).
 - [30] E. L. HALL. “Computer Image Processing and Recognition”. Academic Press, New York (1979).
 - [31] 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).
 - [32] P. S. HECKBERT. Color image quantization for frame buffer display. *Computer Graphics* **16**(3), 297–307 (1982).
 - [33] 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>.
 - [34] 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).
 - [35] B. K. P. HORN. “Robot Vision”. MIT-Press, Cambridge, MA (1982).
 - [36] P. V. C. HOUGH. Method and means for recognizing complex patterns. US Patent 3,069,654 (1962).
 - [37] M. K. HU. Visual pattern recognition by moment invariants. *IEEE Transactions on Information Theory* **8**, 179–187 (1962).

- [38] R. W. G. HUNT. “The Reproduction of Colour”. Wiley, New York, sixth ed. (2004).
- [39] J. ILLINGWORTH AND J. KITTLER. A survey of the Hough transform. *Computer Vision, Graphics and Image Processing* **44**, 87–116 (1988).
- [40] 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.
- [41] 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>.
- [42] International Organization for Standardization, ISO, Geneva. “ISO 13655: 1996, Graphic Technology—Spectral Measurement and Colorimetric Computation for Graphic Arts Images” (1996).
- [43] 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.
- [44] 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).
- [45] B. JÄHNE. “Practical Handbook on Image Processing for Scientific Applications”. CRC Press, Boca Raton, FL (1997).
- [46] B. JÄHNE. “Digitale Bildverarbeitung”. Springer-Verlag, Berlin, fifth ed. (2002).
- [47] A. K. JAIN. “Fundamentals of Digital Image Processing”. Prentice Hall, Englewood Cliffs, NJ (1989).
- [48] X. Y. JIANG AND H. BUNKE. Simple and fast computation of moments. *Pattern Recognition* **24**(8), 801–806 (1991).
- [49] L. KITCHEN AND A. ROSENFELD. Gray-level corner detection. *Pattern Recognition Letters* **1**, 95–102 (1982).
- [50] 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.

- [51] 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.
- [52] S. MALLAT. “A Wavelet Tour of Signal Processing”. Academic Press, New York (1999).
- [53] E. H. W. MEIJERING, W. J. NIJSEN, 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/>.
- [54] 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.
- [55] M. NADLER AND E. P. SMITH. “Pattern Recognition Engineering”. Wiley, New York (1993).
- [56] A. V. OPPENHEIM, R. W. SHAFER, AND J. R. BUCK. “Discrete-Time Signal Processing”. Prentice Hall, Englewood Cliffs, NJ, second ed. (1999).
- [57] T. PAVLIDIS. “Algorithms for Graphics and Image Processing”. Computer Science Press / Springer-Verlag, New York (1982).
- [58] C. A. POYNTON. “Digital Video and HDTV Algorithms and Interfaces”. Morgan Kaufmann Publishers, San Francisco (2003).
- [59] C. E. REID AND T. B. PASSIN. “Signal Processing in C”. Wiley, New York (1992).
- [60] 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).
- [61] A. ROSENFELD AND P. PFALTZ. Sequential operations in digital picture processing. *Journal of the ACM* **12**, 471–494 (1966).
- [62] 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).
- [63] C. SCHMID, R. MOHR, AND C. BAUCKHAGE. Evaluation of interest point detectors. *International Journal of Computer Vision* **37**(2), 151–172 (2000).

- [64] M. SEUL, L. O'GORMAN, AND M. J. SAMMON. "Practical Algorithms for Image Analysis". Cambridge University Press, Cambridge (2000).
- [65] L. G. SHAPIRO AND G. C. STOCKMAN. "Computer Vision". Prentice Hall, Englewood Cliffs, NJ (2001).
- [66] G. SHARMA AND H. J. TRUSSELL. Digital color imaging. *IEEE Transactions on Image Processing* **6**(7), 901–932 (1997).
- [67] 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).
- [68] 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).
- [69] M. SONKA, V. HLAVAC, AND R. BOYLE. "Image Processing, Analysis and Machine Vision". PWS Publishing, Pacific Grove, CA, second ed. (1999).
- [70] 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).
- [71] 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).
- [72] S. THEODORIDIS AND K. KOUTROUMBAS. "Pattern Recognition". Academic Press, New York (1999).
- [73] E. TRUCCO AND A. VERRI. "Introductory Techniques for 3-D Computer Vision". Prentice Hall, Englewood Cliffs, NJ (1998).
- [74] K. TURKOWSKI. Filters for common resampling tasks. In A. GLASSNER, editor, "Graphics Gems I", pp. 147–165. Academic Press, New York (1990).
- [75] 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).
- [76] 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).
- [77] A. WATT. "3D Computer Graphics". Addison-Wesley, Reading, MA, third ed. (1999).

- [78] A. WATT AND F. POLICARPO. “The Computer Image”. Addison-Wesley, Reading, MA (1999).
- [79] G. WOLBERG. “Digital Image Warping”. IEEE Computer Society Press, Los Alamitos, CA (1990).
- [80] G. WYSZECKI AND W. S. STILES. “Color Science: Concepts and Methods, Quantitative Data and Formulae”. Wiley–Interscience, New York, second ed. (2000).
- [81] 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

\circledast (correlation operator) 259, 280
 \oplus (dilation operator) 280
 \ominus (erosion operator) 280
 $*$ (convolution operator) 75, 214, 280
 \neg (logic operator) 24, 280
 \wedge (logic operator) 24, 280
 \sqcup 280
 ∂ 70, 280
 ∇ 280
 $\&$ (operator) 87
 $\&\&$ (operator) 10, 58, 81
 $|$ (operator) 87
 $>>$ (operator) 87

A

accumulator array 54
`add` (method) 285, 291, 298
`addNumericField` (method) 295
adjoint matrix 200
Adobe
– RGB 111, 112
affine mapping 195, 205
`AffineMapping` (class) 243, 244
`AffineTransform` (class) 238
aliasing 141, 142, 148, 151, 152, 164, 234
ambient lighting 102
amplitude 127, 128
angular frequency 126, 127, 147, 152, 158
anonymous class 308
`apply` (method) 121, 122
`applyTable` (method) 83, 299

`applyTo` (method) 240, 242, 246, 252, 253
approximation 222
Arctan function 40, 165, 205, 280
area
– polygon 34
– region 34
`ArrayList` (class) 80, 285, 290
`Arrays.sort` (method) 309
`atan2` (method) 280
AWT 118

B

bandwidth 143
Bartlett window 170, 173, 174
`BasicStroke` (class) 292, 293
basis function 147, 149–151, 158, 164, 183, 184, 190
bias problem 61
bicubic interpolation 229
`BicubicInterpolator` (class) 250, 252
bilinear
– interpolation 227
– mapping 203, 205
`BilinearInterpolator` (class) 249, 252
`BilinearMapping` (class) 246
binary
– image 5
bitmap image 27
`black` (constant) 292
`BLUE` (constant) 308
bounding box 35
Bradford model 113, 116
`BradfordAdaptation` (class) 121

breadth-first 9
Bresenham algorithm 64

C

C2-continuous 220
card 280, 281
cardinal spline 218, 221
cardinality 280, 281
Catmull-Rom interpolation 221
centralMoment(method) 39
centroid 36
chain code 28, 34
chamfer
– algorithm 271
– matching 274
chromatic adaptation 111
– Bradford model 113, 116
– XYZ scaling 112
ChromaticAdaptation(class) 121
CIE 98
– chromaticity diagram 99, 102
– $L^*a^*b^*$ 104, 105
– standard illuminant 101
– XYZ 98, 104, 105, 108, 119, 123
circle 64, 198
circularity 34
circumference 33
city block distance 270
clone(method) 241
Cloneable(interface) 240
clutter 275
collectCorners(method) 80
Collections(class) 81
collision 15
Color(class) 118, 119, 292, 293
color
– difference 105
– image 85–124
– management 123
– temperature 101
color quantization 85–95, 301
– 3:3:2 86
– median-cut 88, 301–311
– octree 89
– populosity 88
color space 123
– colorimetric 97–123
– HSB 119
– HSV 119
– in Java 114
– $L^*a^*b^*$ 104
– sRGB 106
– XYZ 98

ColorBox(class) 304, 306
ColorDimension(class) 308
ColorModel(class) 118
ColorNode(class) 304, 305
ColorProcessor(class) 309
ColorQuantizer(interface) 301, 311
ColorSpace(class) 117–119, 121
comb function 139
compactness 34
compareTo(method) 81, 295
comparing images 255–278
Complex(class) 155
complex number 128, 282
computeMatch(method) 266, 268
computer
– graphics 2
– vision 3
concat(method) 242, 246
conic section 198
connected components problem 16
container 79
contour 17–26
ContourOverlay(class) 25
ContourTracer(class) 22
convertToFloat(method) 297
convertToRGB(method) 311
convex hull 35, 47
convexity 35
convolution 177, 259
– property 137, 175
convolve(method) 78
Convolver(class) 78, 299
coordinate
– Cartesian 194
– homogeneous 194, 241
copyBits(method) 300
Corner(class) 79, 81
corner 69
– detection 69–84
– point 84
– response function 71, 73
– strength 72
CorrCoeffMatcher(class) 266, 267
correlation 177, 259
– coefficient 260
cosine function 134
– one-dimensional 126
– two-dimensional 160, 161
cosine transform 183
 \cos^2 window 173, 174
cross correlation 259–261
CS_CIEXYZ(constant) 120
CS_GRAY(constant) 120

- CS_LINEAR_RGB** (constant) 120
CS_PYCC (constant) 120
CS_sRGB (constant) 119, 120
CS_sRGBbt (constant) 121
cubic
– B-spline interpolation 222
– interpolation 217
– spline 219
cycle length 126
- D**
- D50 101, 102, 119
D65 102, 104, 107
DCT 183–190
– one-dimensional 183, 186
– two-dimensional 187
DCT (method) 186
deconvolution 180
delta function 137
depth-first 7
derivative
– first 77
determinant 200
DFT 144–183, 280
– one-dimensional 144–154
– two-dimensional 157–183
DFT (method) 155
diameter 35
Dirac function 133, 137
discrete
– cosine transform 183–190
– Fourier transform 144–183, 280
– sine transform 183
distance 82, 258
– city block 270
– Euclidean 258, 270
– Manhattan 270
– mask 271
– maximum difference 258
– sum of differences 258
– sum of squared differences 258
– transform 270
DOES_RGB (constant) 311
dots per inch (dpi) 153
dpi 153
draw (method) 83, 293
drawLine (method) 83, 296
DST 183
duplicate (method) 83, 241, 284, 297
- E**
- eccentricity 42, 48
edge
- map 49
– strength 71
eigenvalue 42, 71
eigenvector 71
ellipse 42, 66, 198
Ellipse2D (class) 286
elliptical window 172
elongatedness 42
enum type 308
Euclidean distance 82, 265, 270
Euler number 45
Euler’s notation 128
EXIF 107
- F**
- fast Fourier transform 155, 162, 175, 177
fax encoding 28
feature 32
FFT *see* fast Fourier transform
filter
– Gaussian 70, 77
– in frequency space 175
– inverse 178
– linear 75
Find_Corners (plugin) 84
findCorners (method) 83
FloatProcessor (class) 266
flood filling 6–10
floor function 281
four-point mapping 197
Fourier 130
– analysis 130
– coefficients 130
– descriptor 32
– integral 130
– series 130
– spectrum 32, 131, 144
– transform 126–280
– transform pair 132, 134, 135
frequency 127, 152
– angular 126, 127, 147, 158
– common 127
– directional 164
– effective 164
– fundamental 130, 152, 153
– maximum 142, 164
– space 132, 152, 175
– two-dimensional 164
fromCIEXYZ (method) 116–118, 121
function
– basis 147, 149–151, 158
– cosine 126
– delta 137

- Dirac 133, 137
- impulse 133, 137
- periodic 126
- sine 126
- fundamental
 - frequency 130, 152, 153
 - period 152

G

- gamma correction 114, 120, 122
- modified 108
- gamut 102, 106, 111
 - Adobe RGB 112
 - sRGB 112
- Gaussian
 - area formula 34
 - filter 70, 77
 - function 133, 135
 - window 170, 172, 174
- GenericDialog**(class) 295
- geometric operation 191–254
- get**(method) 286, 289
- getComponents**(method) 119
- getf**(method) 267, 268
- getInterpolatedPixel**(method) 249–251
- getInverse**(method) 240
- getMagnification**(method) 293
- getMatchValue**(method) 268
- getNextNumber**(method) 295
- getPixel**(method) 87
- getPixels**(method) 297
- getTitle**(method) 294
- GIF 28
- gradient 70, 77
- graph 16
- Graphics**(class) 292
- graphics overlay 25
- Graphics2D**(class) 293
- grayscale
 - conversion 110
- GREEN (constant) 308

H

- Hadamard transform 188
- Hanning window 170, 171, 173, 174
- Harris corner detector 70
- HarrisCornerDetector**(class) 78, 84
- hasNext**(method) 285
- Hertz 127, 152
- Hessian normal form 54, 62
- histogram 281
- homogeneous

- coordinate 194, 241
- Hough transform 50–67
 - bias problem 61
 - edge strength 63
 - for circles 64–66
 - for ellipses 66–67
 - for lines 50–63
 - generalized 67
 - hierarchical 63
- HSBtoRGB**(method) 119
- HSV 119

I

- i (imaginary unit) 128, 281, 282
- ICC 116
 - profile 121
- ICC_ColorSpace**(class) 120, 123
- ICC_Profile**(class) 123
- iDCT**(method) 186
- Illuminant**(class) 121
- illuminant 101
- image
 - binary 5
 - coordinates 281
 - space 175
 - warping 204
- ImageCanvas**(class) 292
- ImageJ**
 - geometric operation 238
- ImagePlus**(class) 84, 292, 311
- ImageWindow**(class) 284
- impulse
 - function 133, 137
- in place 159
- IndexColorModel**(class) 304
- Integer.MAX_VALUE**(constant) 286
- interest point 69
- interpolation 210–233, 248–251
 - B-spline 221, 222
 - bicubic 229, 233, 250
 - bilinear 227, 232, 249
 - by convolution 217
 - Catmull-Rom 219, 221, 251
 - cubic 217
 - ideal 213
 - kernel 217
 - Lanczos 223, 231, 254
 - linear 217
 - Mitchell-Netravali 221, 222, 254
 - nearest-neighbor 217, 226, 232, 236, 249
 - spline 219
 - two-dimensional 225–233
- invalidNumber**(method) 295

invariance 34, 37, 38, 43, 45, 256
inverse
– filter 178
`invert`(method) 240, 242, 246
`isLocalMax`(method) 81
`isNaN`(method) 287
isotropic 70, 84
`Iterator`(class) 285
`iterator`(method) 285
ITU709 107

J

`Jama`(package) 199, 246, 247
JPEG 28, 95, 107, 109, 187

L

$L^*a^*b^*$ 104
`Lab_ColorSpace`(class) 117, 121, 122
label 6
Lanczos interpolation 223, 231, 254
line
– endpoints 62
– equation 51, 54
– Hessian normal form 54
– intercept/slope form 51
– intersection 62
linear
– convolution 75
– interpolation 217
– transformation 199
linearity 136
`LinearMapping`(class) 241, 244
`LinkedList`(class) 9, 291
`List`(interface) 80, 285, 288, 291, 292, 297
list 279
local mapping 207
lookup table 83, 299

M

major axis 38
`makeInverseMapping`(method) 248
`makeMapping`(method) 243, 245
Manhattan distance 270
`Mapping`(class) 239
mapping
– affine 195, 205
– bilinear 203, 205
– four-point 197
– function 193
– linear 199
– local 207
– nonlinear 204

– perspective 198
– projective 197–203, 205
– ripple 206
– spherical 207
– three-point 195
– twirl 204
mask 26
`Matrix`(class) 247
maximum
– frequency 142, 164
media-oriented color 109
median-cut algorithm 88, 301
`MedianCutQuantizer`(class) 301, 311
mesh partitioning 207
Mitchell-Netravali interpolation 222, 254
mod operator 154, 214, 281
moment 28, 37–44
– central 37
– Hu’s 43, 48
– invariant 43
– least inertia 38
`moment`(method) 39
morphing 208
`MULTIPLY`(constant) 300

N

`NaN`(constant) 286
nearest-neighbor interpolation 217
`NearestNeighborInterpolator`(class) 249
neighborhood 6, 33
neutral
– point 101
`next`(method) 285
`NO_CHANGES`(constant) 311
nonmaximum suppression 59
`normalCentralMoment`(method) 39
Nyquist 143, 164

O

object 280
OCR 32, 46
octree algorithm 89
orientation 38, 164, 165
orthogonal 190
oscillation 126, 127
overlay 284

P

parameter space 51
Parzen window 170, 171, 173, 174
pattern recognition 3, 32

- perimeter 33
 period 126
 periodicity 126, 158, 163, 167
 perspective
 – image 66
 – mapping 198
 phase 127, 153
 – angle 128
PixelInterpolator(class) 249
 Plessey detector 70
 PNG 107
Point(class) 253, 285, 286, 289
point(class) 22
Point2D(class) 239, 241, 253
Point2D.Double(class) 287
Polygon(class) 286, 293
 polygon
 – area 34
pop(method) 10
 populosity algorithm 88
 power spectrum 153, 162
 print pattern 181
 profile connection space 115, 119
 projection 44, 48
 projective mapping 197–203, 205
ProjectiveMapping(class) 244, 252
 pseudo-perspective mapping 198
push(method) 10
- Q**
- quadrilateral 197
 quantization 85–95
 – linear 86
 – scalar 86
 – vector 88
quantizeImage(method) 311
- R**
- Rectangle**(class) 287
 rectangular pulse 133, 135
 – window 172
RED(constant) 308
 refraction index 207
 region 5–48
 – area 34, 38, 48
 – centroid 36, 48
 – convex hull 35
 – diameter 35
 – eccentricity 42
 – labeling 6–17
 – major axis 38
 – matrix representation 26
 – moment 37
- orientation 38
 – perimeter 33
 – projection 44
 – run length encoding 27
 – topology 45
 relative colorimetry 112
RenderingHints(class) 293
 resampling 209–210
RGBtoHSB(method) 119
 ripple mapping 206
Rotation(class) 244, 252
 rotation 43, 177, 191, 193, 251
 round function 281
 roundness 34
 run length encoding 27
- S**
- sampling 137–143
 – frequency 164
 – interval 140, 141
 – theorem 141, 143, 148, 151, 164, 213
scale(method) 293
Scaling(class) 244
 scaling 43, 191, 193
 separability 187
 sequence 279
 set 279
setColor(method) 293
setf(method) 268
setMinAndMax(method) 284
setNormalize(method) 78, 299
setRenderingHint(method) 293
setStroke(method) 293
setup(method) 284, 294
setValue(method) 83, 296
 Shah function 139
 Shannon 143
Shape(class) 286, 292, 293
 shape
 – feature 32
 – number 30, 31, 47
Shear(class) 244
 shearing 193
 shift property 136
show(method) 84, 299, 311
showDialog(method) 295
showMessage(method) 295
showProcessor(method) 299
 signal space 132, 152
 similarity 136
 Sinc function 133, 214, 225
 sine function 134
 – one-dimensional 126

- sine transform 183
size(method) 285
solve(method) 247
sort(method) 81, 298, 309
source-to-target mapping 209
spectrum 125–190
spherical mapping 207
spline
– cardinal 218, 221
– Catmull-Rom 219, 221, 222
– cubic 219, 222
– cubic B- 221, 222, 253
– interpolation 219
sqr(method) 299
square window 174
sRGB 106, 108, 112, 114
– ambient lighting 102
– grayscale conversion 110
– white point 102
Stack(class) 9, 10
stack 7
standard illuminant 101, 111
Stroke(class) 293
structure 280
super(method) 292
super-Gaussian window 170, 172
- T**
target-to-source mapping 204, 210, 240
template matching 255, 257, 267
three-point mapping 195
threshold 59
TIFF 28
time unit 127
toArray(method) 81, 298
toCIEXYZ(method) 116–119, 122
topological property 45
tracking 69
transform pair 132
TransformJ(package) 238
translate(method) 286, 293
Translation(class) 244
- translation 43, 193
tree 7
truncate function 281
tuple 280
twirl mapping 204
TwirlMapping(class) 247
TYPE_Lab(constant) 121
- U**
unit square 204
- V**
variance 261
Vector(class) 80, 298
vector 279
– graphics 25
viewing angle 102
- W**
Walsh transform 188
warping 204
wasCanceled(method) 295
wave number 147, 158, 164, 184
wavelet 188
white(constant) 292
white point 101, 104
– D50 101, 116
– D65 102, 107
windowed matching 267
windowing 169
windowing function 169–171
– Bartlett 170, 173, 174
– cosine² 173, 174
– elliptical 170, 172
– Gaussian 170, 172, 174
– Hanning 170, 173, 174
– Parzen 170, 173, 174
– rectangular pulse 172
– super-Gaussian 170, 172
- X**
XYZ scaling 112

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 Series

The complete manuscript for this book was prepared by the authors "camera-ready" in L^AT_EX using Donald Knuth's Computer Modern fonts. The additional packages *algorithmicx* (by Szász János) for presenting algorithms, *listings* (by Carsten Heinz) for listing program 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.