X.media.press



Wilhelm Burger · Mark James Burge

Digitale Bildverarbeitung

Eine Einführung mit Java und ImageJ

2., überarbeitete Auflage
Mit 255 Abbildungen und 16 Tabellen

Springer

Wilhelm Burger
Medientechnik und -design / Digitale Medien
Fachhochschule Hagenberg
Hauptstr. 117
4232 Hagenberg, Österreich
wilbur@ieee.org

Mark James Burge
National Science Foundation
4201 Wilson Blvd., Suite 835
Arlington, VA 22230, USA
mburge@acm.org

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Ursprünglich erschienen in der Reihe **eXamen.press**

ISSN 1439-3107
ISBN-10 3-540-30940-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-30940-6 Springer Berlin Heidelberg New York
ISBN-10 3-540-21465-8 1. Aufl. Springer Berlin Heidelberg New York

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

© Springer-Verlag Berlin Heidelberg 2005, 2006
Printed in Germany

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften. Text und Abbildungen wurden mit größter Sorgfalt erarbeitet. Verlag und Autor können jedoch für eventuell verbliebene fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Satz: Druckfertige Daten der Autoren
Herstellung: LE-TEX, Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3100 YL – 5 4 3 2 1 0

Vorwort

Dieses Buch ist eine Einführung in die digitale Bildverarbeitung, die sowohl als Referenz für den Praktiker wie auch als Grundlagentext für die Ausbildung gedacht ist. Es bietet eine Übersicht über die wichtigsten klassischen Techniken in moderner Form und damit einen grundlegenden „Werkzeugkasten“ für dieses spannende Fachgebiet. Das Buch sollte daher insbesondere für folgende drei Einsatzbereiche gut geeignet sein:

- Für Wissenschaftler und Techniker, die digitale Bildverarbeitung als Hilfsmittel für die eigene Arbeit einsetzen oder künftig einsetzen möchten und Interesse an der Realisierung eigener, maßgeschneiderter Verfahren haben.
- Als umfassende Grundlage zum Selbststudium für ausgebildete IT-Experten, die sich erste Kenntnisse im Bereich der digitalen Bildverarbeitung und der zugehörigen Programmiertechnik aneignen oder eine bestehende Grundausbildung vertiefen möchten.
- Als einführendes Lehrbuch für eine ein- bis zweisemestrige Lehrveranstaltung im ersten Studienabschnitt, etwa ab dem 3. Semester. Die meisten Kapitel sind auf das Format einer wöchentlichen Vorlesung ausgelegt, ergänzt durch Einzelaufgaben für begleitende Übungen.

Inhaltlich steht die praktische Anwendbarkeit und konkrete Umsetzung im Vordergrund, ohne dass dabei auf die notwendigen formalen Details verzichtet wird. Allerdings ist dies kein Rezeptbuch, sondern Lösungsansätze werden schrittweise in drei unterschiedlichen Formen entwickelt: (a) in mathematischer Schreibweise, (b) als abstrakte Algorithmen und (c) als konkrete Java-Programme. Die drei Formen ergänzen sich und sollen in Summe ein Maximum an Verständlichkeit sicherstellen.

Voraussetzungen

Wir betrachten digitale Bildverarbeitung nicht vorrangig als mathematische Disziplin und haben daher die formalen Anforderungen in diesem Buch auf das Notwendigste reduziert – sie gehen über die im ersten Studienabschnitt üblichen Kenntnisse nicht hinaus. Als Einsteiger sollte man daher auch nicht beunruhigt sein, dass einige Kapitel auf den ersten Blick etwas mathematisch aussehen. Die durchgehende, einheitliche Notation und ergänzenden Informationen im Anhang tragen dazu bei, eventuell bestehende Schwierigkeiten leicht zu überwinden. Bezüglich der *Programmierung* setzt das Buch gewisse Grundkenntnisse voraus, idealerweise (aber nicht notwendigerweise) in **Java**. Elementare Datenstrukturen, prozedurale Konstrukte und die Grundkonzepte der objektorientierten Programmierung sollten dem Leser vertraut sein. Da Java mittlerweile in vielen Studienplänen als erste Programmiersprache unterrichtet wird, sollte der Einstieg in diesen Fällen problemlos sein. Aber auch Java-Neulinge mit etwas Programmiererfahrung in ähnlichen Sprachen (insbesondere C/C++) dürften sich rasch zurechtfinden.

Softwareseitig basiert dieses Buch auf **ImageJ**, einer komfortablen, frei verfügbaren Programmierumgebung, die von Wayne Rasband am U.S. National Institute of Health (NIH) entwickelt wird.[1] ImageJ ist vollständig in Java implementiert, läuft damit auf vielen Plattformen und kann durch eigene, kleine „Plugin"-Module leicht erweitert werden. Die meisten Programmbeispiele sind jedoch so gestaltet, dass sie problemlos in andere Umgebungen oder Programmiersprachen portiert werden können.

Einsatz in Forschung und Entwicklung

Dieses Buch ist einerseits für den Einsatz in der Lehre konzipiert, bietet andererseits jedoch an vielen Stellen grundlegende Informationen und Details, die in dieser Form nicht immer leicht zu finden sind. Es sollte daher für den interessierten Praktiker und Entwickler eine wertvolle Hilfe sein. Es ist aber nicht als umfassender Ausgangspunkt zur Forschung gedacht und erhebt vor allem auch keinen Anspruch auf wissenschaftliche Vollständigkeit. Im Gegenteil, es wurde versucht, die Fülle der möglichen Literaturangaben auf die wichtigsten und (auch für Studierende) leicht zugreifbaren Quellen zu beschränken. Darüber hinaus konnten einige weiterführende Techniken, wie etwa hierarchische Methoden, Wavelets, Eigenimages oder Bewegungsanalyse, aus Platzgründen nicht berücksichtigt werden. Auch Themenbereiche, die mit „Intelligenz" zu tun haben, wie Objekterkennung oder Bildverstehen, wurden bewusst ausgespart, und Gleiches gilt für alle dreidimensionalen Problemstellungen aus dem Bereich „Computer Vision". Die in diesem Buch gezeigten Verfahren sind durchweg „blind und dumm", wobei wir aber glauben,

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

dass gerade die technisch saubere Umsetzung dieser scheinbar einfachen Dinge eine essentielle Grundlage für den Erfolg aller weiterführenden (vielleicht sogar „intelligenteren“) Ansätze ist.

Man wird auch enttäuscht sein, falls man sich ein Programmierhandbuch für ImageJ oder Java erwartet – dafür gibt es wesentlich bessere Quellen. Die Programmiersprache selbst steht auch nie im Mittelpunkt, sondern dient uns vorrangig als Instrument zur Verdeutlichung, Präzisierung und – praktischerweise – auch zur Umsetzung der gezeigten Verfahren.

Einsatz in der Ausbildung

An vielen Ausbildungseinrichtungen ist der Themenbereich digitale Signal- und Bildverarbeitung seit Langem in den Studienplänen integriert, speziell im Bereich der Informatik und Kommunikationstechnik, aber auch in anderen technischen Studienrichtungen mit entsprechenden formalen Grundlagen und oft auch erst in höheren („graduate“) Studiensemestern.

Immer häufiger finden sich jedoch auch bereits in der Grundausbildung einführende Lehrveranstaltungen zu diesem Thema, vor allem in neueren Studienrichtungen der Informatik und Softwaretechnik, Mechatronik oder Medientechnik. Ein Problem dabei ist das weitgehende Fehlen von geeigneter Literatur, die bezüglich der Voraussetzungen und der Inhalte diesen Anforderungen entspricht. Die klassische Fachliteratur ist häufig zu formal für Anfänger, während oft gleichzeitig manche populäre, praktische Methode nicht ausreichend genau beschrieben ist. So ist es auch für die Lektoren schwierig, für eine solche Lehrveranstaltung ein einzelnes Textbuch oder zumindest eine kompakte Sammlung von Literatur zu finden und den Studierenden empfehlen zu können. Das Buch soll dazu beitragen, diese Lücke zu schließen.

Die Inhalte der nachfolgenden Kapitel sind für eine Lehrveranstaltung von 1–2 Semestern in einer Folge aufgebaut, die sich in der praktischen Ausbildung gut bewährt hat. Die Kapitel sind meist in sich so abgeschlossen, dass ihre Abfolge relativ flexibel gestaltet werden kann. Der inhaltliche Schwerpunkt liegt dabei auf den klassischen Techniken im Bildraum, wie sie in der heutigen Praxis im Vordergrund stehen. Die Kapitel 13–15 zum Thema Spektraltechniken sind hingegen als grundlegende Einführung gedacht und bewusst im hinteren Teil des Buchs platziert. Sie können bei Bedarf leicht reduziert oder überhaupt weggelassen werden. Die nachfolgende Übersicht (auf Seite VIII) zeigt mehrere Varianten zur Gliederung von Lehrveranstaltungen über ein oder zwei Semester.

1 Semester: Für einen Kurs über *ein* Semester könnte man je nach Zielsetzung zwischen den Themenschwerpunkten **Bildverarbeitung** und **Bildanalyse** wählen. Beide Lehrveranstaltungen passen gut in den ersten Abschnitt moderner Studienpläne im Bereich der

„Road Map“ für 1- und 2-Semester-Kurse	Bildverarb.	Bildanalyse	Grundlagen	Vertiefung
1. Crunching Pixels	☒	☒	☒	☐
2. Digitale Bilder	☒	☒	☒	☐
3. ImageJ	☒	☒	☒	☐
4. Histogramme	☒	☐	☒	☐
5. Punktoperationen	☒	☐	☒	☐
6. Filter	☒	☒	☒	☐
7. Kanten und Konturen	☒	☒	☒	☐
8. Auffinden von Eckpunkten	☐	☒	☐	☒
9. Detektion einfacher Kurven	☐	☒	☐	☒
10. Morphologische Filter	☒	☐	☒	☐
11. Regionen in Binärbildern	☐	☒	☒	☐
12. Farbbilder	☒	☐	☐	☒
13. Einführung in Spektraltechniken	☐	☒	☐	☒
14. Die diskrete Fouriertransformation in 2D	☐	☒	☐	☒
15. Die diskrete Kosinustransformation	☐	☐	☐	☒
16. Geometrische Bildoperationen	☒	☐	☐	☒
17. Bildvergleich	☐	☒	☐	☒
	1 Sem.		2 Sem.	

Informatik oder Informationstechnik. Der zweite Kurs eignet sich etwa auch als Grundlage für eine einführende Lehrveranstaltung in der medizinischen Ausbildung.

2 Semester: Stehen *zwei* Semester zur Vermittlung der Inhalte zur Verfügung, wäre eine Aufteilung in zwei aufbauende Kurse (*Grundlagen* und *Vertiefung*) möglich, wobei die Themen nach ihrem Schwierigkeitsgrad gruppiert sind.

Ergänzungen zur 2. Auflage

Die vorliegende zweite Auflage der deutschen Ausgabe enthält neben zahlreichen kleineren Korrekturen und Verbesserungen zwei ergänzende Abschnitte zu den Themen Histogrammanpassung (Kap. 5) und Lanczos-Interpolation (Kap. 16). Soweit sinnvoll wurden die Beispielprogamme auf Java 5 aktualisiert. Dank der Unterstützung von Seiten des Verlags erhielt diese Ausgabe auch ein moderneres Layout, hochwertigeres Papier und einige wichtige Seiten in Farbe.

Online-Materialien

Auf der Website zu diesem Buch

www.imagingbook.com

stehen zusätzliche Materialien in elektronischer Form frei zur Verfügung, u. a. Testbilder in Originalgröße und Farbe, Java-Quellcode für die angeführten Beispiele, aktuelle Ergänzungen und etwaige Korrekturen. Für Lehrende gibt es außerdem den vollständigen Satz von Abbildungen und Formeln zur Verwendung in eigenen Präsentationen. Kommentare, Fragen, Anregungen und Korrekturen sind willkommen und sollten adressiert werden an:

imagingbook@gmail.com

Ein Dankeschön

Dieses Buch wäre nicht entstanden ohne das Verständnis und die Unterstützung unsere Familien, die uns dankenswerterweise erlaubten, über mehr als ein Jahr hinweg ziemlich schlechte Väter und Ehepartner zu sein. Unser Dank geht auch an Wayne Rasband am NIH für die Entwicklung von ImageJ und sein hervorragendes Engagement innerhalb der Community, an die Kollegen Prof. Axel Pinz (TU Graz) und Prof. Vaclav Hlavac (TU Prag) für ihre sachkundigen Kommentare sowie an die aufmerksamen Leser der ersten Auflage für die zahlreichen Kommentare und Verbesserungsvorschläge. Respekt gebührt nicht zuletzt Ursula Zimpfer für ihr professionelles Copy-Editing sowie Jutta Maria Fleschutz vom Springer-Verlag für ihre Geduld und die gute Zusammenarbeit.

Hagenberg / Washington D.C.
März 2006

Inhaltsverzeichnis

1 Crunching Pixels 1
1.1 Programmieren mit Bildern 2
1.2 Bildanalyse und „intelligente“ Verfahren 3

2 Digitale Bilder 5
2.1 Arten von digitalen Bildern 5
2.2 Bildaufnahme 5
2.2.1 Das Modell der Lochkamera 5
2.2.2 Die „dünne“ Linse 8
2.2.3 Übergang zum Digitalbild 9
2.2.4 Bildgröße und Auflösung 10
2.2.5 Bildkoordinaten 11
2.2.6 Pixelwerte 12
2.3 Dateiformate für Bilder 14
2.3.1 Raster- vs. Vektordaten 15
2.3.2 Tagged Image File Format (TIFF) 15
2.3.3 Graphics Interchange Format (GIF) 16
2.3.4 Portable Network Graphics (PNG) 17
2.3.5 JPEG 17
2.3.6 Windows Bitmap (BMP) 21
2.3.7 Portable Bitmap Format (PBM) 21
2.3.8 Weitere Dateiformate 22
2.3.9 Bits und Bytes 22
2.4 Aufgaben 24

3 ImageJ 27
3.1 Software für digitale Bilder 28
3.1.1 Software zur Bildbearbeitung 28
3.1.2 Software zur Bildverarbeitung 28

3.2 Eigenschaften von ImageJ 28
3.2.1 Features 29
3.2.2 Fertige Werkzeuge 30
3.2.3 ImageJ-Plugins 31
3.2.4 Beispiel-Plugin: „inverter“ 32
3.3 Weitere Informationen zu ImageJ und Java 35
3.3.1 Ressourcen für ImageJ 35
3.3.2 Programmieren mit Java 35
3.4 Aufgaben 36

4 Histogramme 39
4.1 Was ist ein Histogramm? 39
4.2 Was ist aus Histogrammen abzulesen? 41
4.2.1 Eigenschaften der Bildaufnahme 41
4.2.2 Bildfehler 43
4.3 Berechnung von Histogrammen 46
4.4 Histogramme für Bilder mit mehr als 8 Bit 48
4.4.1 Binning 48
4.4.2 Beispiel 48
4.4.3 Implementierung 49
4.5 Histogramme von Farbbildern 49
4.5.1 Luminanzhistogramm 49
4.5.2 Histogramme der Farbkomponenten 50
4.5.3 Kombinierte Farbhistogramme 50
4.6 Das kumulative Histogramm 52
4.7 Aufgaben 52

5 Punktoperationen 55
5.1 Änderung der Bildintensität 56
5.1.1 Kontrast und Helligkeit 56
5.1.2 Beschränkung der Ergebniswerte (*clamping*) 56
5.1.3 Invertieren von Bildern 57
5.1.4 Schwellwertoperation (*tresholding*) 57
5.2 Punktoperationen und Histogramme 58
5.3 Automatische Kontrastanpassung 59
5.4 Linearer Histogrammausgleich 61
5.5 Histogrammanpassung 65
5.5.1 Häufigkeiten und Wahrscheinlichkeiten 65
5.5.2 Prinzip der Histogrammanpassung 66
5.5.3 Stückweise lineare Referenzverteilung 67
5.5.4 Anpassung an ein konkretes Histogamm 68
5.5.5 Beispiele 70
5.6 Gammakorrektur 74
5.6.1 Warum Gamma? 74
5.6.2 Die Gammafunktion 75
5.6.3 Reale Gammawerte 76
5.6.4 Anwendung der Gammakorrektur 77

5.6.5 Implementierung ... 78
5.6.6 Modifizierte Gammafunktion ... 78
5.7 Punktoperationen in ImageJ ... 81
5.7.1 Punktoperationen mit Lookup-Tabellen ... 81
5.7.2 Arithmetische Standardoperationen ... 82
5.7.3 Punktoperationen mit mehreren Bildern ... 83
5.7.4 ImageJ-Plugins für mehrere Bilder ... 84
5.8 Aufgaben ... 85

6 Filter ... 89
6.1 Was ist ein Filter? ... 89
6.2 Lineare Filter ... 91
6.2.1 Die Filtermatrix ... 91
6.2.2 Anwendung des Filters ... 92
6.2.3 Berechnung der Filteroperation ... 93
6.2.4 Beispiele für Filter-Plugins ... 94
6.2.5 Ganzzahlige Koeffizienten ... 95
6.2.6 Filter beliebiger Größe ... 97
6.2.7 Arten von linearen Filtern ... 98
6.3 Formale Eigenschaften linearer Filter ... 101
6.3.1 Lineare Faltung ... 101
6.3.2 Eigenschaften der linearen Faltung ... 102
6.3.3 Separierbarkeit von Filtern ... 103
6.3.4 Impulsantwort eines Filters ... 105
6.4 Nichtlineare Filter ... 106
6.4.1 Minimum- und Maximum-Filter ... 107
6.4.2 Medianfilter ... 108
6.4.3 Das gewichtete Medianfilter ... 109
6.4.4 Andere nichtlineare Filter ... 111
6.5 Implementierung von Filtern ... 112
6.5.1 Effizienz von Filterprogrammen ... 112
6.5.2 Behandlung der Bildränder ... 113
6.6 Filteroperationen in ImageJ ... 113
6.6.1 Lineare Filter ... 113
6.6.2 Gauß-Filter ... 115
6.6.3 Nichtlineare Filter ... 115
6.7 Aufgaben ... 115

7 Kanten und Konturen ... 117
7.1 Wie entsteht eine Kante? ... 117
7.2 Gradienten-basierte Kantendetektion ... 118
7.2.1 Partielle Ableitung und Gradient ... 119
7.2.2 Ableitungsfilter ... 120
7.3 Filter zur Kantendetektion ... 120
7.3.1 Prewitt- und Sobel-Operator ... 120
7.3.2 Roberts-Operator ... 123
7.3.3 Kompass-Operatoren ... 124

7.3.4 Kantenoperatoren in ImageJ 125
7.4 Weitere Kantenoperatoren 125
7.4.1 Kantendetektion mit zweiten Ableitungen 125
7.4.2 Kanten auf verschiedenen Skalenebenen 126
7.4.3 Canny-Filter 126
7.5 Von Kanten zu Konturen 128
7.5.1 Konturen verfolgen 128
7.5.2 Kantenbilder 129
7.6 Kantenschärfung 129
7.6.1 Kantenschärfung mit dem Laplace-Filter 130
7.6.2 Unscharfe Maskierung (*unsharp masking*) 132
7.7 Aufgaben 136

8 Auffinden von Eckpunkten 139
8.1 „Points of interest“ 139
8.2 Harris-Detektor 140
8.2.1 Lokale Strukturmatrix 140
8.2.2 *Corner Response Function* (CRF) 141
8.2.3 Bestimmung der Eckpunkte 142
8.2.4 Beispiele 142
8.3 Implementierung 142
8.3.1 Schritt 1 – Berechnung der *corner response function* 143
8.3.2 Schritt 2 – Bestimmung der Eckpunkte 148
8.3.3 Anzeigen der Eckpunkte 151
8.3.4 Zusammenfassung 152
8.4 Aufgaben 153

9 Detektion einfacher Kurven 155
9.1 Auffällige Strukturen 155
9.2 Hough-Transformation 156
9.2.1 Parameterraum 157
9.2.2 Akkumulator-Array 159
9.2.3 Eine bessere Geradenparametrisierung 159
9.3 Implementierung der Hough-Transformation 160
9.3.1 Füllen des Akkumulator-Arrays 161
9.3.2 Auswertung des Akkumulator-Arrays 163
9.3.3 Erweiterungen der Hough-Transformation 164
9.4 Hough-Transformation für Kreise und Ellipsen 167
9.4.1 Kreise und Kreisbögen 167
9.4.2 Ellipsen 168
9.5 Aufgaben 169

10 Morphologische Filter 171
10.1 Schrumpfen und wachsen lassen 172
10.1.1 Nachbarschaft von Bildelementen 173
10.2 Morphologische Grundoperationen 174
10.2.1 Das Strukturelement 174
10.2.2 Punktmengen 174
10.2.3 Dilation 175
10.2.4 Erosion 176
10.2.5 Eigenschaften von Dilation und Erosion 176
10.2.6 Design morphologischer Filter 177
10.2.7 Anwendungsbeispiel: *Outline* 178
10.3 Zusammengesetzte Operationen 179
10.3.1 Opening 179
10.3.2 Closing 182
10.3.3 Eigenschaften von Opening und Closing 182
10.4 Morphologische Filter für Grauwert- und Farbbilder 182
10.4.1 Strukturelemente 183
10.4.2 Grauwert-Dilation und -Erosion 184
10.4.3 Grauwert-Opening und -Closing 185
10.5 Implementierung morphologischer Filter 186
10.5.1 Binäre Bilder in ImageJ 186
10.5.2 Dilation und Erosion 187
10.5.3 Opening und Closing 189
10.5.4 Outline 190
10.5.5 Morphologische Operationen in ImageJ 190
10.6 Aufgaben 192

11 Regionen in Binärbildern 195
11.1 Auffinden von Bildregionen 196
11.1.1 Regionenmarkierung durch *Flood Filling* 196
11.1.2 Sequentielle Regionenmarkierung 200
11.1.3 Regionenmarkierung – Zusammenfassung 206
11.2 Konturen von Regionen 206
11.2.1 Äußere und innere Konturen 206
11.2.2 Kombinierte Regionenmarkierung und Konturfindung 208
11.2.3 Implementierung 209
11.2.4 Beispiele 212
11.3 Repräsentation von Bildregionen 214
11.3.1 Matrix-Repräsentation 214
11.3.2 Lauflängenkodierung 214
11.3.3 *Chain Codes* 215
11.4 Eigenschaften binärer Bildregionen 218
11.4.1 Formmerkmale (*Features*) 218
11.4.2 Geometrische Eigenschaften 219
11.4.3 Statistische Formeigenschaften 222
11.4.4 Momentenbasierte geometrische Merkmale 224

11.4.5 Projektionen 228
11.4.6 Topologische Merkmale 229
11.5 Aufgaben 229

12 Farbbilder 233
12.1 RGB-Farbbilder 233
12.1.1 Aufbau von Farbbildern 235
12.1.2 Farbbilder in ImageJ 237
12.2 Farbräume und Farbkonversion 248
12.2.1 Umwandlung in Grauwertbilder 249
12.2.2 Desaturierung von Farbbildern 251
12.2.3 HSV/HSB- und HLS-Farbraum 253
12.2.4 TV-Komponentenfarbräume – YUV, YIQ und YC_bC_r 262
12.2.5 Farbräume für den Druck – CMY und CMYK . . . 266
12.3 Colorimetrische Farbräume 270
12.3.1 CIE-Farbräume 271
12.3.2 CIE $L^*a^*b^*$ 276
12.3.3 sRGB 278
12.3.4 Adobe RGB 282
12.3.5 Farben und Farbräume in Java 283
12.4 Statistiken von Farbbildern 288
12.4.1 Wie viele Farben enthält ein Bild? 288
12.4.2 Histogramme 288
12.5 Farbquantisierung 289
12.5.1 Skalare Farbquantisierung 292
12.5.2 Vektorquantisierung 293
12.6 Aufgaben 297

13 Einführung in Spektraltechniken 299
13.1 Die Fouriertransformation 300
13.1.1 Sinus- und Kosinusfunktionen 300
13.1.2 Fourierreihen als Darstellung periodischer Funktionen 303
13.1.3 Fourierintegral 304
13.1.4 Fourierspektrum und -transformation 305
13.1.5 Fourier-Transformationspaare 306
13.1.6 Wichtige Eigenschaften der Fouriertransformation 307
13.2 Übergang zu diskreten Signalen 311
13.2.1 Abtastung 311
13.2.2 Diskrete und periodische Funktionen 317
13.3 Die diskrete Fouriertransformation (DFT) 317
13.3.1 Definition der DFT 319
13.3.2 Diskrete Basisfunktionen 320
13.3.3 Schon wieder Aliasing! 321
13.3.4 Einheiten im Orts- und Spektralraum 324
13.3.5 Das Leistungsspektrum 326

13.4 Implementierung der DFT 326
13.4.1 Direkte Implementierung 326
13.4.2 Fast Fourier Transform (FFT) 328
13.5 Aufgaben 329

14 Diskrete Fouriertransformation in 2D 331
14.1 Definition der 2D-DFT 331
14.1.1 2D-Basisfunktionen 332
14.1.2 Implementierung der zweidimensionalen DFT 332
14.2 Darstellung der Fouriertransformierten in 2D 333
14.2.1 Wertebereich 333
14.2.2 Zentrierte Darstellung 336
14.3 Frequenzen und Orientierung in 2D 337
14.3.1 Effektive Frequenz 337
14.3.2 Frequenzlimits und Aliasing in 2D 338
14.3.3 Orientierung 338
14.3.4 Geometrische Korrektur des 2D-Spektrums 339
14.3.5 Auswirkungen der Periodizität 340
14.3.6 *Windowing* 340
14.3.7 Fensterfunktionen 342
14.4 Beispiele für Fouriertransformierte in 2D 347
14.4.1 Skalierung 347
14.4.2 Periodische Bildmuster 347
14.4.3 Drehung 347
14.4.4 Gerichtete, längliche Strukturen 347
14.4.5 Natürliche Bilder 347
14.4.6 Druckraster 347
14.5 Anwendungen der DFT 351
14.5.1 Lineare Filteroperationen im Spektralraum 351
14.5.2 Lineare Faltung und Korrelation 352
14.5.3 Inverse Filter 353
14.6 Aufgaben 354

15 Die diskrete Kosinustransformation (DCT) 355
15.1 Eindimensionale DCT 355
15.1.1 Basisfunktionen der DCT 356
15.1.2 Implementierung der eindimensionalen DCT 356
15.2 Zweidimensionale DCT 358
15.2.1 Separierbarkeit 359
15.2.2 Beispiele 359
15.3 Andere Spektraltransformationen 359
15.4 Aufgaben 361

16 Geometrische Bildoperationen 363
16.1 2D-Koordinatentransformation 364
16.1.1 Einfache Abbildungen 365
16.1.2 Homogene Koordinaten 365
16.1.3 Affine Abbildung (Dreipunkt-Abbildung) 366
16.1.4 Projektive Abbildung (Vierpunkt-Abbildung) 367
16.1.5 Bilineare Abbildung 372
16.1.6 Weitere nichtlineare Bildverzerrungen 373
16.1.7 Lokale Transformationen 376
16.2 Resampling 377
16.2.1 *Source-to-Target Mapping* 378
16.2.2 *Target-to-Source Mapping* 378
16.3 Interpolation 379
16.3.1 Einfache Interpolationsverfahren 380
16.3.2 Ideale Interpolation 380
16.3.3 Interpolation durch Faltung 383
16.3.4 Kubische Interpolation 383
16.3.5 Lanczos-Interpolation 385
16.3.6 Interpolation in 2D 386
16.3.7 Aliasing 392
16.4 Java-Implementierung 395
16.4.1 Geometrische Abbildungen 396
16.4.2 Pixel-Interpolation 405
16.4.3 Anwendungsbeispiele 408
16.5 Aufgaben 410

17 Bildvergleich 411
17.1 Template Matching in Intensitätsbildern 412
17.1.1 Abstand zwischen Bildmustern 413
17.1.2 Umgang mit Drehungen und Größenänderungen . . 420
17.1.3 Implementierung 420
17.2 Vergleich von Binärbildern 420
17.2.1 Direkter Vergleich von Binärbildern 421
17.2.2 Die Distanztransformation 423
17.2.3 *Chamfer Matching* 426
17.3 Aufgaben 430

A Mathematische Notation 431
A.1 Häufig verwendete Symbole 431
A.2 Komplexe Zahlen $\mathbb{C}$ 433
A.3 Algorithmische Komplexität und $\mathcal{O}$-Notation 434

B Java-Notizen 435
B.1 Arithmetik 435
B.1.1 Ganzzahlige Division 435
B.1.2 Modulo-Operator 437
B.1.3 Unsigned Bytes 437

B.1.4 Mathematische Funktionen (`Math`-Klasse) 438
B.1.5 Runden 439
B.1.6 Inverse Tangensfunktion 439
B.1.7 `Float` und `Double` (Klassen) 439
B.2 Arrays in Java 440
B.2.1 Arrays erzeugen 440
B.2.2 Größe von Arrays 440
B.2.3 Zugriff auf Array-Elemente 441
B.2.4 Zweidimensionale Arrays 441

C ImageJ-Kurzreferenz 445
C.1 Installation und Setup 445
C.2 ImageJ-API 447
C.2.1 Bilder 447
C.2.2 Bildprozessoren 447
C.2.3 Plugins 448
C.2.4 GUI-Klassen 449
C.2.5 Window-Management 450
C.2.6 Utility-Klassen 450
C.2.7 Input-Output 450
C.3 Bilder und Bildfolgen erzeugen 450
C.3.1 `ImagePlus` (Klasse) 450
C.3.2 `ImageStack` (Klasse) 451
C.3.3 `NewImage` (Klasse) 451
C.3.4 `ImageProcessor` (Klasse) 452
C.4 Bildprozessoren erzeugen 452
C.4.1 `ImageProcessor` (Klasse) 452
C.4.2 `ByteProcessor` (Klasse) 452
C.4.3 `ColorProcessor` (Klasse) 452
C.4.4 `FloatProcessor` (Klasse) 453
C.4.5 `ShortProcessor` (Klasse) 453
C.5 Bildparameter 454
C.5.1 `ImageProcessor` (Klasse) 454
C.6 Zugriff auf Pixel 454
C.6.1 `ImageProcessor` (Klasse) 454
C.7 Konvertieren von Bildern 457
C.7.1 `ImageProcessor` (Klasse) 457
C.7.2 `ImagePlus`, `ImageConverter` (Klassen) 458
C.8 Histogramme und Bildstatistiken 458
C.8.1 `ImageProcessor` (Klasse) 458
C.9 Punktoperationen 459
C.9.1 `ImageProcessor` (Klasse) 459
C.9.2 `Blitter` (Interface) 460
C.10 Filter 461
C.10.1 `ImageProcessor` (Klasse) 461
C.11 Geometrische Operationen 461
C.11.1 `ImageProcessor` (Klasse) 461

C.12 Grafische Operationen in Bildern 462
C.12.1 `ImageProcessor` (Klasse) 462
C.13 Bilder darstellen 463
C.13.1 `ImagePlus` (Klasse) 463
C.14 Operationen auf Bildfolgen (Stacks) 464
C.14.1 `ImagePlus` (Klasse) 464
C.14.2 `ImageStack` (Klasse) 464
C.14.3 Stack-Beispiel 465
C.15 *Region of Interest* (ROI) 469
C.15.1 `ImageProcessor` (Klasse) 469
C.15.2 `ImageStack` (Klasse) 469
C.15.3 `ImagePlus` (Klasse) 470
C.15.4 `Roi`, `Line`, `OvalRoi`, `PolygonRoi` (Klassen) 470
C.16 *Image Properties* 471
C.16.1 `ImagePlus` (Klasse) 471
C.17 Interaktion 471
C.17.1 `IJ` (Klasse) 471
C.17.2 `ImageProcessor` (Klasse) 473
C.17.3 `GenericDialog` (Klasse) 473
C.18 Plugins 474
C.18.1 `PlugIn` (Interface) 474
C.18.2 `PlugInFilter` (Interface) 474
C.18.3 Plugins ausführen – `IJ` (Klasse) 476
C.19 Window-Management 476
C.19.1 `WindowManager` (Klasse) 476
C.20 Weitere Funktionen 477
C.20.1 `ImagePlus` (Klasse) 477
C.20.2 `IJ` (Klasse) 477

D Source Code 479
D.1 Harris Corner Detector 480
D.1.1 File `Corner.java` 480
D.1.2 File `HarrisCornerDetector.java` 481
D.1.3 File `HarrisCornerPlugin_.java` 485
D.2 Kombinierte Regionenmarkierung-Konturverfolgung 487
D.2.1 File `ContourTracingPlugin_.java` 487
D.2.2 File `Node.java` 488
D.2.3 File `Contour.java` 488
D.2.4 File `OuterContour.java` 489
D.2.5 File `InnerContour.java` 490
D.2.6 File `ContourSet.java` 490
D.2.7 File `ContourTracer.java` 492
D.2.8 File `ContourOverlay.java` 495

Literaturverzeichnis 497

Sachverzeichnis 503

1

Crunching Pixels

Lange Zeit war die digitale Verarbeitung von Bildern einer relativ kleinen Gruppe von Spezialisten mit teurer Ausstattung und einschlägigen Kenntnissen vorbehalten. Spätestens durch das Auftauchen von digitalen Kameras, Scannern und Multi-Media-PCs auf den Schreibtischen vieler Zeitgenossen wurde jedoch die Beschäftigung mit digitalen Bildern, bewusst oder unbewusst, zu einer Alltäglichkeit für viele Computerbenutzer. War es vor wenigen Jahren noch mit großem Aufwand verbunden, Bilder überhaupt zu digitalisieren und im Computer zu speichern, erlauben uns heute üppig dimensionierte Hauptspeicher, riesige Festplatten und Prozessoren mit Taktraten von mehreren Gigahertz digitale Bilder und Videos mühelos und schnell zu manipulieren. Dazu gibt es Tausende von Programmen, die dem Amateur genauso wie dem Fachmann die Bearbeitung von Bildern in bequemer Weise ermöglichen.

So gibt es heute eine riesige „Community“ von Personen, für die das Arbeiten mit digitalen Bildern auf dem Computer zur alltäglichen Selbstverständlichkeit geworden ist. Dabei überrascht es nicht, dass im Verhältnis dazu das Verständnis für die zugrunde liegenden Mechanismen meist über ein oberflächliches Niveau nicht hinausgeht. Für den typischen Konsumenten, der lediglich seine Urlaubsfotos digital archivieren möchte, ist das auch kein Problem, ähnlich wie ein tieferes Verständnis eines Verbrennungsmotors für das Fahren eines Autos weitgehend entbehrlich ist.

Immer häufiger stehen aber auch IT-Fachleute vor der Aufgabe, mit diesem Thema professionell umzugehen, schon allein deshalb, weil Bilder (und zunehmend auch andere Mediendaten) heute ein fester Bestandteil des digitalen Workflows in vielen Unternehmen und Institutionen sind, nicht nur in der Medizin oder in der Medienbranche. Genauso sind auch „gewöhnliche“ Softwaretechniker heute oft mit digitalen Bildern auf Programm-, Datei- oder Datenbankebene konfrontiert und Program-

mierumgebungen in sämtlichen modernen Betriebssystemen bieten dazu umfassende Möglichkeiten. Der einfache praktische Umgang mit dieser Materie führt jedoch, verbunden mit einem oft unklaren Verständnis der grundlegenden Zusammenhänge, häufig zur Unterschätzung der Probleme und nicht selten zu ineffizienten Lösungen, teuren Fehlern und persönlicher Frustration.

1.1 Programmieren mit Bildern

Bildverarbeitung wird im heutigen Sprachgebrauch häufig mit Bild*be*arbeitung verwechselt, also der Manipulation von Bildern mit fertiger Software, wie beispielsweise *Adobe Photoshop*, *Corel Paint* etc. In der Bild*ver*arbeitung geht es im Unterschied dazu um die Konzeption und Erstellung von Software, also um die Entwicklung (oder Erweiterung) dieser Programme selbst.

Moderne Programmierumgebungen machen auch dem Nicht-Spezialisten durch umfassende APIs (Application Programming Interfaces) praktisch jeden Bereich der Informationstechnik zugänglich: Netzwerke und Datenbanken, Computerspiele, Sound, Musik und natürlich auch Bilder. Die Möglichkeit, in eigenen Programmen auf die einzelnen Elemente eines Bilds zugreifen und diese beliebig manipulieren zu können, ist faszinierend und verführerisch zugleich. In der Programmierung sind Bilder nichts weiter als simple Zahlenfelder, also Arrays, deren Zellen man nach Belieben lesen und verändern kann. Alles, was man mit Bildern tun kann, ist somit grundsätzlich machbar und der Phantasie sind keine Grenzen gesetzt.

Im Unterschied zur digitalen Bildverarbeitung beschäftigt man sich in der *Computergrafik* mit der *Synthese* von Bildern aus geometrischen Beschreibungen bzw. dreidimensionalen Objektmodellen [23,29,87]. Realismus und Geschwindigkeit stehen – heute vor allem für Computerspiele – dabei im Vordergrund. Dennoch bestehen zahlreiche Berührungspunkte zur Bildverarbeitung, etwa die Transformation von Texturbildern, die Rekonstruktion von 3D-Modellen aus Bilddaten, oder spezielle Techniken wie „Image-Based Rendering“ und „Non-Photorealistic Rendering“ [64, 88]. In der Bildverarbeitung finden sich wiederum Methoden, die ursprünglich aus der Computergrafik stammen, wie volumetrische Modelle in der medizinischen Bildverarbeitung, Techniken der Farbdarstellung oder Computational-Geometry-Verfahren. Extrem eng ist das Zusammenspiel zwischen Bildverarbeitung und Grafik natürlich in der digitalen Post-Produktion für Film und Video, etwa zur Generierung von Spezialeffekten [89]. Die grundlegenden Verfahren in diesem Buch sind daher nicht nur für Einzelbilder, sondern auch für die Bearbeitung von Bildfolgen, d. h. Video- und Filmsequenzen, durchaus relevant.

1.2 Bildanalyse und „intelligente“ Verfahren

Viele Aufgaben in der Bildverarbeitung, die auf den ersten Blick einfach und vor allem dem menschlichen Auge so spielerisch leicht zu fallen scheinen, entpuppen sich in der Praxis als schwierig, unzuverlässig, zu langsam, oder gänzlich unmachbar. Besonders gilt dies für den Bereich der Bild*analyse*, bei der es darum geht, sinnvolle Informationen aus Bildern zu extrahieren, sei es etwa, um ein Objekt vom Hintergrund zu trennen, einer Straße auf einer Landkarte zu folgen oder den Strichcode auf einer Milchpackung zu finden – meistens ist das schwieriger, als es uns die eigenen Fähigkeiten erwarten lassen.

Dass die technische Realität heute von der beinahe unglaublichen Leistungsfähigkeit biologischer Systeme (und den Phantasien Hollywoods) noch weit entfernt ist, sollte uns zwar Respekt machen, aber nicht davon abhalten, diese Herausforderung unvoreingenommen und kreativ in Angriff zu nehmen. Vieles ist auch mit unseren heutigen Mitteln durchaus lösbar, erfordert aber – wie in jeder technischen Disziplin – sorgfältiges und rationales Vorgehen. Bildverarbeitung funktioniert nämlich in vielen, meist unspektakulären Anwendungen seit langem und sehr erfolgreich, zuverlässig und schnell, nicht zuletzt als Ergebnis fundierter Kenntnisse, präziser Planung und sauberer Umsetzung.

Die Analyse von Bildern ist in diesem Buch nur ein Randthema, mit dem wir aber doch an mehreren Stellen in Berührung kommen, etwa bei der Segmentierung von Bildregionen (Kap. 11), beim Auffinden von einfachen Kurven (Kap. 9) oder beim Vergleichen von Bildern (Kap. 17). Alle hier beschriebenen Verfahren arbeiten jedoch ausschließlich auf Basis der Pixeldaten, also „blind“ und „bottom-up“ und ohne zusätzliches Wissen oder „Intelligenz“. Darin liegt ein wesentlicher Unterschied zwischen digitaler Bildverarbeitung einerseits und „Mustererkennung“ (*Pattern Recognition*) bzw. *Computer Vision* andererseits. Diese Disziplinen greifen zwar häufig auf die Methoden der Bildverarbeitung zurück, ihre Zielsetzungen gehen aber weit über diese hinaus:

Pattern Recognition ist eine vorwiegend mathematische Disziplin, die sich allgemein mit dem Auffinden von „Mustern“ in Daten und Signalen beschäftigt. Typische Beispiele aus dem Bereich der Bildanalyse sind etwa die Unterscheidung von Texturen oder die optische Zeichenerkennung (OCR). Diese Methoden betreffen aber nicht nur Bilddaten, sondern auch Sprach- und Audiosignale, Texte, Börsenkurse, Verkehrsdaten, die Inhalte großer Datenbanken u.v.m. Statistische und syntaktische Methoden spielen in der Mustererkennung eine zentrale Rolle (s. beispielsweise [21, 62, 83]).

Computer Vision beschäftigt sich mit dem Problem, Sehvorgänge in der realen, dreidimensionalen Welt zu mechanisieren. Dazu gehört die räumliche Erfassung von Gegenständen und Szenen, das Erkennen von Objekten, die Interpretation von Bewegungen, autonome Navigation, das mechanische Aufgreifen von Dingen (durch Robo-

ter) usw. Computer Vision entwickelte sich ursprünglich als Teilgebiet der „Künstlichen Intelligenz“ (*Artificial Intelligence*, kurz „AI“) und die Entwicklung zahlreicher AI-Methoden wurde von visuellen Problemstellungen motiviert (s. beispielsweise [18, Kap. 13]). Auch heute bestehen viele Berührungspunkte, besonders aktuell im Zusammenhang mit adaptivem Verhalten und maschinellem Lernen. Einführende und vertiefende Literatur zum Thema Computer Vision findet man z. B. in [4, 26, 37, 76, 80, 84].

Interessant ist der Umstand, dass trotz der langjährigen Entwicklung in diesen Bereichen viele der ursprünglich als relativ einfach betrachteten Aufgaben weiterhin nicht oder nur unzureichend gelöst sind. Das macht die Arbeit an diesen Themen – trotz aller Schwierigkeiten – spannend. Wunderbares darf man sich von der digitalen Bildverarbeitung allein nicht erwarten, sie könnte aber durchaus die „Einstiegsdroge“ zu weiterführenden Unternehmungen sein.

2

Digitale Bilder

Zentrales Thema in diesem Buch sind digitale Bilder, und wir können davon ausgehen, dass man heute kaum einem Leser erklären muss, worum es sich dabei handelt. Genauer gesagt geht es um Rasterbilder, also Bilder, die aus regelmäßig angeordneten Elementen (*picture elements* oder *pixel*) bestehen, im Unterschied etwa zu Vektorgrafiken.

2.1 Arten von digitalen Bildern

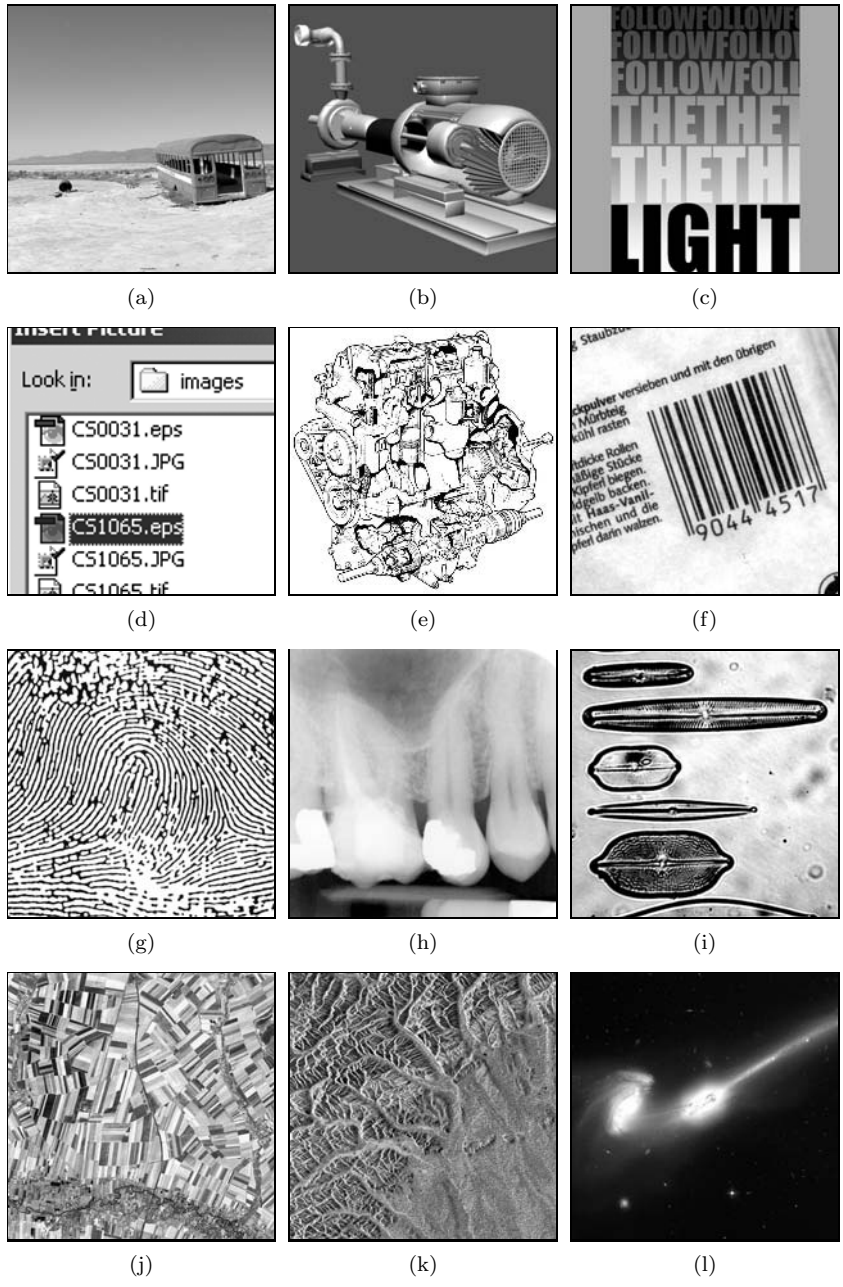
In der Praxis haben wir mit vielen Arten von digitalen Rasterbildern zu tun, wie Fotos von Personen oder Landschaften, Farb- und Grautonbilder, gescannte Druckvorlagen, Baupläne, Fax-Dokumente, Screenshots, Mikroskopaufnahmen, Röntgen- und Ultraschallbilder, Radaraufnahmen u. v. m. (Abb. 2.1). Auf welchem Weg diese Bilder auch entstehen, sie bestehen (fast) immer aus rechteckig angeordneten Bildelementen und unterscheiden sich – je nach Ursprung und Anwendungsbereich – vor allem durch die darin abgelegten Werte.

2.2 Bildaufnahme

Der eigentliche Prozess der Entstehung von Bildern ist oft kompliziert und meistens für die Bildverarbeitung auch unwesentlich. Dennoch wollen wir uns kurz ein Aufnahmeverfahren etwas genauer ansehen, mit dem die meisten von uns vertraut sind: eine optische Kamera.

2.2.1 Das Modell der Lochkamera

Das einfachste Prinzip einer Kamera, das wir uns überhaupt vorstellen können, ist die so genannte Lochkamera, die bereits im 13. Jahrhun-



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Abbildung 2.1
Digitale Bilder: natürliche Landschaftsszene (a), synthetisch generierte Szene (b), Poster-Grafik (c), Screenshot (d), Schwarz-Weiß-Illustration (e), Strichcode (f), Fingerabdruck (g), Röntgenaufnahme (h), Mikroskopbild (i), Satellitenbild (j), Radarbild (k), astronomische Aufnahme (l).

dert als „Camera obscura“ bekannt war. Sie hat zwar heute keinerlei praktische Bedeutung mehr (eventuell als Spielzeug), aber sie dient als brauchbares Modell, um die für uns wesentlichen Elemente der optischen Abbildung ausreichend zu beschreiben, zumindest soweit wir es im Rahmen dieses Buchs überhaupt benötigen.

Die Lochkamera besteht aus einer geschlossenen Box mit einer winzigen Öffnung an der Vorderseite und der Bildebene an der gegenüberliegenden Rückseite. Lichtstrahlen, die von einem Objektpunkt vor der Kamera ausgehend durch die Öffnung einfallen, werden geradlinig auf die Bildebene projiziert, wodurch ein verkleinertes und seitenverkehrtes Abbild der sichtbaren Szene entsteht (Abb. 2.2).

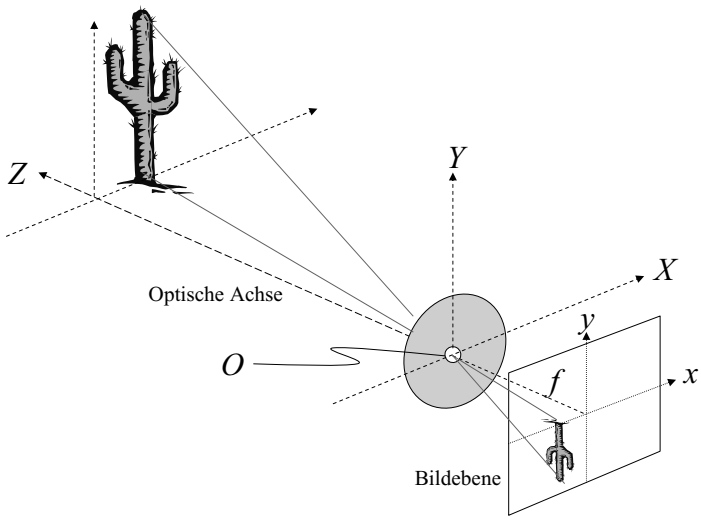


Abbildung 2.2
Geometrie der Lochkamera. Die Lochöffnung bildet den Ursprung des dreidimensionalen Koordinatensystems (X, Y, Z), in dem die Positionen der Objektpunkte in der Szene beschrieben werden. Die optische Achse, die durch die Lochöffnung verläuft, bildet die Z-Achse dieses Koordinatensystems. Ein eigenes, zweidimensionales Koordinatensystem (x, y) beschreibt die Projektionspunkte auf der Bildebene. Der Abstand f („Brennweite“) zwischen der Öffnung und der Bildebene bestimmt den Abbildungsmaßstab der Projektion.

Perspektivische Abbildung

Die geometrischen Verhältnisse der Lochkamera sind extrem einfach. Die so genannte „optische Achse“ läuft gerade durch die Lochöffnung und rechtwinkelig zur Bildebene. Nehmen wir an, ein sichtbarer Objektpunkt (in unserem Fall die Spitze des Kaktus) befindet sich in einer Distanz Z von der Lochebene und im vertikalen Abstand Y über der optischen Achse. Die Höhe der zugehörigen Projektion y wird durch zwei Parameter bestimmt: die (fixe) Tiefe der Kamerabox f und den Abstand Z des Objekts vom Koordinatenursprung. Durch Vergleich der ähnlichen Dreiecke ergibt sich der einfache Zusammenhang

$$y = -f\frac{Y}{Z} \quad \text{und genauso} \quad x = -f\frac{X}{Z}. \tag{2.1}$$

Proportional zur Tiefe der Box, also dem Abstand f, ändert sich auch der Maßstab der gewonnenen Abbildung analog zur Änderung der Brennweite in einer herkömmlichen Fotokamera. Ein kleines f (= kurze Brennweite) erzeugt eine kleine Abbildung bzw. – bei fixer Bildgröße – einen größeren Blickwinkel, genau wie bei einem Weitwinkelobjektiv. Verlängern wir die „Brennweite“ f, dann ergibt sich – wie bei einem Teleobjektiv – eine vergrößerte Abbildung verbunden mit einem entsprechend kleineren Blickwinkel. Das negative Vorzeichen in Gl. 2.1 zeigt lediglich an, dass die Projektion horizontal und vertikal gespiegelt, also um 180° gedreht, erscheint.

Gl. 2.1 beschreibt nichts anderes als die perspektivische Abbildung, wie wir sie heute als selbstverständlich kennen.[1] Wichtige Eigenschaften dieses theoretischen Modells sind u. a., dass Geraden im 3D-Raum immer auch als Geraden in der 2D-Projektion erscheinen und dass Kreise als Ellipsen abgebildet werden.

2.2.2 Die „dünne“ Linse

Während die einfache Geometrie der Lochkamera sehr anschaulich ist, hat die Kamera selbst in der Praxis keine Bedeutung. Um eine scharfe Projektion zu erzielen, benötigt man eine möglichst kleine Lochblende, die wiederum wenig Licht durchlässt und damit zu sehr langen Belichtungszeiten führt. In der Realität verwendet man optische Linsen und Linsensysteme, deren Abbildungsverhalten in vieler Hinsicht besser, aber auch wesentlich komplizierter ist. Häufig bedient man sich aber auch in diesem Fall zunächst eines einfachen Modells, das mit dem der Lochkamera praktisch identisch ist. Im Modell der „dünnen Linse“ ist lediglich die Lochblende durch eine Linse ersetzt (Abb. 2.3). Die Linse wird dabei als symmetrisch und unendlich dünn angenommen, d. h., jeder

[1] Es ist heute schwer vorstellbar, dass die Regeln der perspektivischen Geometrie zwar in der Antike bekannt waren, danach aber in Vergessenheit gerieten und erst in der Renaissance (um 1430 durch den Florentiner Maler Brunoleschi) wiederentdeckt wurden.

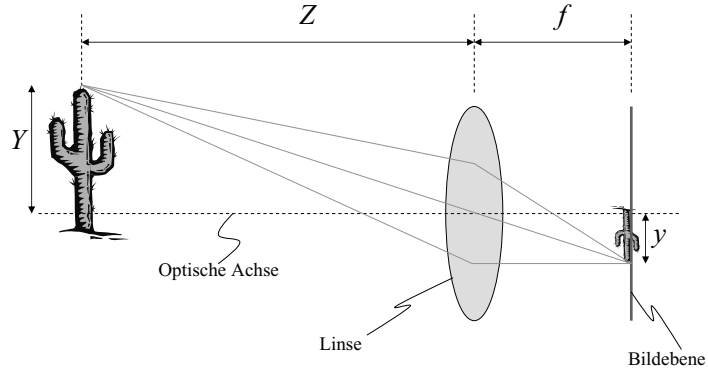


Abbildung 2.3
Modell der „dünnen Linse“.

Lichtstrahl, der in die Linse fällt, wird an einer virtuellen Ebene in der Linsenmitte gebrochen. Daraus ergibt sich die gleiche Abbildungsgeometrie wie bei einer Lochkamera. Für die Beschreibung echter Linsen und Linsensysteme ist dieses Modell natürlich völlig unzureichend, denn Details wie Schärfe, Blenden, geometrische Verzerrungen, unterschiedliche Brechung verschiedener Farben und andere reale Effekte sind darin überhaupt nicht berücksichtigt. Für unsere Zwecke reicht dieses primitive Modell zunächst aber aus und für Interessierte findet sich dazu eine Fülle an einführender Literatur (z. B. [48]).

2.2.3 Übergang zum Digitalbild

Das auf die Bildebene unserer Kamera projizierte Bild ist zunächst nichts weiter als eine zweidimensionale, zeitabhängige, kontinuierliche Verteilung von Lichtenergie. Um diesen kontinuierlichen „Lichtfilm“ als Schnappschuss in digitaler Form in unseren Computer zu bekommen, sind drei wesentliche Schritte erforderlich:

1. Die kontinuierliche Lichtverteilung muss räumlich abgetastet werden.
2. Die daraus resultierende Funktion muss zeitlich abgetastet werden, um ein einzelnes Bild zu erhalten.
3. Die einzelnen Werte müssen quantisiert werden in eine endliche Anzahl möglicher Zahlenwerte, damit sie am Computer darstellbar sind.

Schritt 1: Räumliche Abtastung (*spatial sampling*)

Die räumliche Abtastung, d. h. der Übergang von einer kontinuierlichen zu einer diskreten Lichtverteilung, erfolgt in der Regel direkt durch die Geometrie des Aufnahmesensors, z. B. in einer Digital- oder Viodeokamera. Die einzelnen Sensorelemente sind dabei fast immer regelmäßig und rechtwinklig zueinander auf der Sensorfläche angeordnet (Abb. 2.4). Es gibt allerdings auch Bildsensoren mit hexagonalen Elementen oder auch ringförmige Sensorstrukturen für spezielle Anwendungen.

Schritt 2: Zeitliche Abtastung (*temporal sampling*)

Die zeitliche Abtastung geschieht durch Steuerung der Zeit, über die die Messung der Lichtmenge durch die einzelnen Sensorelemente erfolgt. Auf dem CCD[2]-Chip einer Digitalkamera wird dies durch das Auslösen eines Ladevorgangs und die Messung der elektrischen Ladung nach einer vorgegebenen Belichtungszeit gesteuert.

[2] *Charge-Coupled Device*

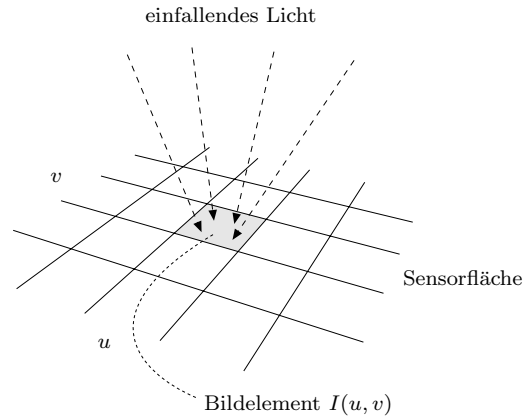


Abbildung 2.4
Die räumliche Abtastung der kontinuierlichen Lichtverteilung erfolgt normalerweise direkt durch die Sensorgeometrie, im einfachsten Fall durch eine ebene, regelmäßige Anordnung rechteckiger Sensorelemente, die jeweils die auf sie einfallende Lichtmenge messen.

Schritt 3: Quantisierung der Pixelwerte

Um die Bildwerte im Computer verarbeiten zu können, müssen diese abschließend auf eine endliche Menge von Zahlenwerten abgebildet werden, typischerweise auf ganzzahlige Werte (z. B. $256 = 2^8$ oder $4096 = 2^{12}$) oder auch auf Gleitkommawerte. Diese Quantisierung erfolgt durch Analog-Digital-Wandlung, entweder in der Sensorelektronik selbst oder durch eine spezielle Interface-Hardware.

Bilder als diskrete Funktionen

Das Endergebnis dieser drei Schritte ist eine Beschreibung des aufgenommenen Bilds als zweidimensionale, regelmäßige Matrix von Zahlen (Abb. 2.5). Etwas formaler ausgedrückt, ist ein digitales Bild I damit eine zweidimensionale Funktion von den ganzzahligen Koordinaten $\mathbb{N} \times \mathbb{N}$ auf eine Menge (bzw. ein Intervall) von Bildwerten $\mathbb{P}$, also

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

Damit sind wir bereits so weit, Bilder in unserem Computer darzustellen, sie zu übertragen, zu speichern, zu komprimieren oder in beliebiger Form zu bearbeiten. Ab diesem Punkt ist es uns zunächst egal, auf welchem Weg unsere Bilder entstanden sind, wir behandeln sie einfach nur als zweidimensionale, numerische Daten. Bevor wir aber mit der Verarbeitung von Bildern beginnen, noch einige wichtige Definitionen.

2.2.4 Bildgröße und Auflösung

Im Folgenden gehen wir davon aus, dass wir mit rechteckigen Bildern zu tun haben. Das ist zwar eine relativ sichere Annahme, es gibt aber auch Ausnahmen. Die *Größe* eines Bilds wird daher direkt bestimmt durch



$F(x,y)$

$$\longrightarrow$$

$$I(u,v) = \begin{bmatrix}
148 & 123 & 52 & 107 & 123 & 162 & 172 & 123 & 64 & 89 & \cdots \\
147 & 130 & 92 & 95 & 98 & 130 & 171 & 155 & 169 & 163 & \cdots \\
141 & 118 & 121 & 148 & 117 & 107 & 144 & 137 & 136 & 134 & \cdots \\
82 & 106 & 93 & 172 & 149 & 131 & 138 & 114 & 113 & 129 & \cdots \\
57 & 101 & 72 & 54 & 109 & 111 & 104 & 135 & 106 & 125 & \cdots \\
138 & 135 & 114 & 82 & 121 & 110 & 34 & 76 & 101 & 111 & \cdots \\
138 & 102 & 128 & 159 & 168 & 147 & 116 & 129 & 124 & 117 & \cdots \\
113 & 89 & 89 & 109 & 106 & 126 & 114 & 150 & 164 & 145 & \cdots \\
120 & 121 & 123 & 87 & 85 & 70 & 119 & 64 & 79 & 127 & \cdots \\
145 & 141 & 143 & 134 & 111 & 124 & 117 & 113 & 64 & 112 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots &
\end{bmatrix}$$

Abbildung 2.5
Übergang von einer kontinuierlichen Lichtverteilung $F(x,y)$ zum diskreten Digitalbild $I(u,v)$ (links), zugehöriger Bildausschnitt (unten).



die *Breite* M (Anzahl der Spalten) und die *Höhe* N (Anzahl der Zeilen) der zugehörigen Bildmatrix I.

Die *Auflösung* (*resolution*) eines Bilds spezifiziert seine räumliche Ausdehnung in der realen Welt und wird in der Anzahl der Bildelemente pro Längeneinheit angegeben, z. B. in „dots per inch“ (dpi) oder „lines per inch“ (lpi) bei Druckvorlagen oder etwa in Pixel pro Kilometer bei Satellitenfotos. Meistens geht man davon aus, dass die Auflösung eines Bilds in horizontaler und vertikaler Richtung identisch ist, die Bildelemente also quadratisch sind. Das ist aber nicht notwendigerweise so, z. B. weisen die meisten Videokameras nichtquadratische Bildelemente auf.

Die räumliche Auflösung eines Bilds ist in vielen Bildverarbeitungsschritten unwesentlich, solange es nicht um geometrische Operationen geht. Wenn aber etwa ein Bild gedreht werden muss, Distanzen zu messen sind oder ein präziser Kreis darin zu zeichnen ist, dann sind genaue Informationen über die Auflösung wichtig. Die meisten professionellen Bildformate und Softwaresysteme berücksichtigen daher diese Angaben sehr genau.

2.2.5 Bildkoordinaten

Um zu wissen, welche Bildposition zu welchem Bildelement gehört, benötigen wir ein Koordinatensystem. Entgegen der in der Mathematik üblichen Konvention ist das in der Bildverarbeitung übliche Koordinatensystem in der vertikalen Richtung umgedreht, die y-Koordinate läuft also von oben nach unten und der Koordinatenursprung liegt links oben (Abb. 2.6). Obwohl dieses System keinerlei praktische oder theoretische Vorteile hat (im Gegenteil, bei geometrischen Aufgaben häufig zu Verwirrung führt), wird es mit wenigen Ausnahmen in praktisch allen Softwaresystemen verwendet. Es dürfte ein Erbe der Fernsehtechnik sein, in der Bildzeilen traditionell entlang der Abtastrichtung des Elektronenstrahls, also von oben nach unten nummeriert werden. Aus praktischen Gründen starten wir die Nummerierung von Spalten und Zeilen bei 0, da auch Java-Arrays mit dem Index 0 beginnen.

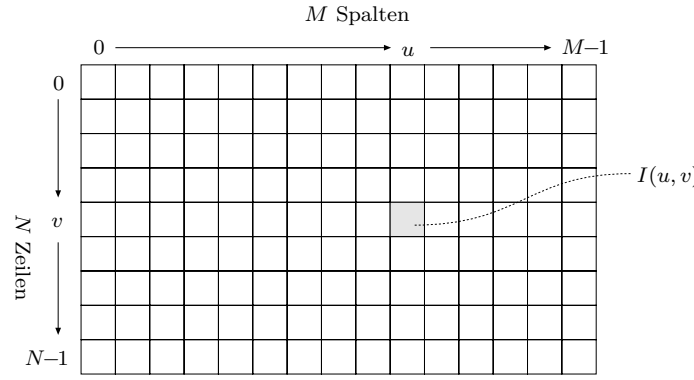


Abbildung 2.6
Bildkoordinaten. In der digitalen Bildverarbeitung wird traditionell ein Koordinatensystem verwendet, dessen Ursprung ($u = 0, v = 0$) links oben liegt. Die Koordinaten u, v bezeichnen die *Spalten* bzw. die *Zeilen* des Bilds. Für ein Bild der Größe $M \times N$ ist der maximale Spaltenindex $u_{\max} = M - 1$, der maximale Zeilenindex $v_{\max} = N - 1$.

2.2.6 Pixelwerte

Die Information innerhalb eines Bildelements ist von seinem Typ abhängig. Pixelwerte sind praktisch immer binäre Wörter der Länge k, sodass ein Pixel grundsätzlich 2^k unterschiedliche Werte annehmen kann. k wird auch häufig als die Bit-Tiefe (oder schlicht „Tiefe“) eines Bilds bezeichnet. Wie genau die einzelnen Pixelwerte in zugehörige Bitmuster kodiert sind, ist vor allem abhängig vom Bildtyp wie Binärbild, Grauwertbild, RGB-Farbbild und speziellen Bildtypen, die im Folgenden kurz zusammengefasst sind (Tabelle 2.1):

Grauwertbilder (Intensitätsbilder)

Die Bilddaten von Grauwertbildern bestehen aus nur einem Kanal, der die Intensität, Helligkeit oder Dichte des Bilds beschreibt. Da in den meisten Fällen nur positive Werte sinnvoll sind (schließlich entspricht Intensität der Lichtenergie, die nicht negativ sein kann), werden üblicherweise positive ganze Zahlen im Bereich $[0 \ldots 2^k - 1]$ zur Darstellung benutzt. Ein typisches Grauwertbild verwendet z. B. $k = 8$ Bits (1 Byte) pro Pixel und deckt damit die Intensitätswerte $[0 \ldots 255]$ ab, wobei der Wert 0 der minimalen Helligkeit (schwarz) und 255 der maximalen Helligkeit (weiß) entspricht.

Bei vielen professionellen Anwendungen für Fotografie und Druck, sowie in der Medizin und Astronomie reicht der mit 8 Bits/Pixel verfügbare Wertebereich allerdings nicht aus. Bildtiefen von 12, 14 und sogar 16 Bits sind daher nicht ungewöhnlich.

Binärbilder

Binärbilder sind spezielle Intensitätsbilder, die nur zwei Pixelwerte vorsehen – schwarz und weiß –, die mit einem einzigen Bit (0/1) pro Pixel kodiert werden. Binärbilder werden häufig verwendet zur Darstellung von Strichgrafiken, zur Archivierung von Dokumenten, für die Kodierung von Fax-Dokumenten, und natürlich im Druck.

Tabelle 2.1
Wertebereiche von Bildelementen und typische Einsatzbereiche.

Grauwertbilder (Intensitätsbilder):

Kanäle	*Bit/Pixel*	*Wertebereich*	*Anwendungen*
1	1	0...1	Binärbilder: Dokumente, Illustration, Fax
1	8	0...255	Universell: Foto, Scan, Druck
1	12	0...4095	Hochwertig: Foto, Scan, Druck
1	14	0...16383	Professionell: Foto, Scan, Druck
1	16	0...65535	Höchste Qualität: Medizin, Astronomie

Farbbilder:

Kanäle	*Bits/Pixel*	*Wertebereich*	*Anwendungen*
3	24	$[0...255]^3$	RGB, universell: Foto, Scan, Druck
3	36	$[0...4095]^3$	RGB, hochwertig: Foto, Scan, Druck
3	42	$[0...16383]^3$	RGB, professionell: Foto, Scan, Druck
4	32	$[0...255]^4$	CMYK, digitale Druckvorstufe

Spezialbilder:

Kanäle	*Bits/Pixel*	*Wertebereich*	*Anwendungen*
1	16	$-32768...32767$	Ganzzahlig pos./neg., hoher Wertebereich
1	32	$\pm 3.4 \cdot 10^{38}$	Gleitkomma: Medizin, Astronomie
1	64	$\pm 1.8 \cdot 10^{308}$	Gleitkomma: interne Verarbeitung

Farbbilder

Die meisten Farbbilder sind mit jeweils einer Komponente für die Primärfarben Rot, Grün und Blau (RGB) kodiert, typischerweise mit 8 Bits pro Komponente. Jedes Pixel eines solchen Farbbilds besteht daher aus $3 \times 8 = 24$ Bits und der Wertebereich jeder Farbkomponente ist wiederum $[0 \ldots 255]$. Ähnlich wie bei Intensitätsbildern sind Farbbilder mit Tiefen von 30, 36 und 42 Bits für professionelle Anwendungen durchaus üblich. Heute verfügen oft auch digitale Amateurkameras bereits über die Möglichkeit, z. B. 36 Bit tiefe Bilder aufzunehmen, allerdings fehlt dafür oft die Unterstützung in der zugehörigen Bildbearbeitungssoftware. In der digitalen Druckvorstufe werden üblicherweise subtraktive Farbmodelle mit 4 und mehr Farbkomponenten verwendet, z. B. das CMYK-(*C*yan-*M*agenta-*Y*ellow-Blac*k*-)Modell (s. auch Kap. 12).

Bei *Index-* oder *Palettenbildern* ist im Unterschied zu *Vollfarbenbildern* die Anzahl der unterschiedlichen Farben innerhalb eines Bilds auf eine Palette von Farb- oder Grauwerten beschränkt. Die Bildwerte selbst sind in diesem Fall nur Indizes (mit maximal 8 Bits) auf die Tabelle von Farbwerten (s. auch Abschn. 12.1.1).

Spezialbilder

Spezielle Bilddaten sind dann erforderlich, wenn die oben beschriebenen Standardformate für die Darstellung der Bildwerte nicht ausreichen.

Unter anderem werden häufig Bilder mit negativen Werten benötigt, die etwa als Zwischenergebnisse einzelner Verarbeitungsschritte (z. B. bei der Detektion von Kanten) auftreten. Des Weiteren werden auch Bilder mit Gleitkomma-Elementen (meist mit 32 oder 64 Bits/Pixel) verwendet, wenn ein großer Wertebereich bei gleichzeitig hoher Genauigkeit dargestellt werden muss, z. B. in der Medizin oder in der Astronomie. Die zugehörigen Dateiformate sind allerdings ausnahmslos anwendungsspezifisch und werden daher von üblicher Standardsoftware nicht unterstützt.

2.3 Dateiformate für Bilder

Während wir in diesem Buch fast immer davon ausgehen, dass Bilddaten bereits als zweidimensionale Arrays in einem Programm vorliegen, sind Bilder in der Praxis zunächst meist in Dateien gespeichert. Dateien sind daher eine essentielle Grundlage für die Speicherung, Archivierung und für den Austausch von Bilddaten, und die Wahl des richtigen Dateiformats ist eine wichtige Entscheidung. In der Frühzeit der digitalen Bildverarbeitung (bis etwa 1985) ging mit fast jeder neuen Softwareentwicklung auch die Entwicklung eines neuen Dateiformats einher, was zu einer Myriade verschiedenster Dateiformate und einer kombinatorischen Vielfalt an notwendigen Konvertierungsprogrammen führte.[3] Heute steht glücklicherweise eine Reihe standardisierter und für die meisten Einsatzzwecke passender Dateiformate zur Verfügung, was vor allem den Austausch von Bilddaten erleichtert und auch die langfristige Lesbarkeit fördert. Dennoch ist, vor allem bei umfangreichen Projekten, die Auswahl des richtigen Dateiformats nicht immer einfach und manchmal mit Kompromissen verbunden, wobei einige typische Kriterien etwa folgende sind:

- **Art der Bilder:** Schwarzweißbilder, Grauwertbilder, Scans von Dokumenten, Farbfotos, farbige Grafiken oder Spezialbilder (z. B. mit Gleitkommadaten). In manchen Anwendungen (z. B. bei Luft- oder Satellitenaufnahmen) ist auch die maximale Bildgröße wichtig.
- **Speicherbedarf und Kompression:** Ist die Dateigröße ein Problem und ist eine (insbesondere *verlustbehaftete*) Kompression der Bilddaten zulässig?
- **Kompatibilität:** Wie wichtig ist der Austausch von Bilddaten und eine langfristige Lesbarkeit (Archivierung) der Bilddaten?
- **Anwendungsbereich:** In welchem Bereich werden die Bilddaten hauptsächlich verwendet, etwa für den Druck, im Web, im Film, in der Computergrafik, Medizin oder Astronomie?

[3] Dieser historische Umstand behinderte lange Zeit nicht nur den konkreten Austausch von Bildern, sondern beanspruchte vielerorts auch wertvolle Entwicklungsressourcen.

2.3.1 Raster- vs. Vektordaten

Im Folgenden beschäftigen wir uns ausschließlich mit Dateiformaten zur Speicherung von *Rastbildern*, also Bildern, die durch eine regelmäßige Matrix (mit diskreten Koordinaten) von Pixelwerten beschrieben werden. Im Unterschied dazu wird bei *Vektorgrafiken* der Bildinhalt in Form von geometrischen Objekten mit kontinuierlichen Koordinaten repräsentiert und die Rasterung erfolgt erst bei der Darstellung auf einem konkreten Endgerät (z. B. einem Display oder Drucker).

Für Vektorbilder sind übrigens standardisierte Austauschformate kaum vorhanden bzw. wenig verbreitet, wie beispielsweise das ANSI/ISO-Standardformat CGM („Computer Graphics Metafile“), SVG (Scalable Vector Graphics[4]) und einige proprietäre Formate wie DXF („Drawing Exchange Format“ von AutoDesk), AI („Adobe Illustrator“), PICT („QuickDraw Graphics Metafile“ von Apple) oder WMF/ EMF („Windows Metafile“ bzw. „Enhanced Metafile“ von Microsoft). Die meisten dieser Formate können Vektordaten und Rasterbilder *zusammen* in einer Datei kombinieren. Auch die Dateiformate PS („PostScript“) bzw. EPS („Encapsulated PostScript“) von Adobe und das daraus abgeleitete PDF („Portable Document Format“) bieten diese Möglichkeit, werden allerdings vorwiegend zur Druckausgabe und Archivierung verwendet.[5]

2.3.2 Tagged Image File Format (TIFF)

TIFF ist ein universelles und flexibles Dateiformat, das professionellen Ansprüchen in vielen Anwendungsbereichen gerecht wird. Es wurde ursprünglich von Aldus konzipiert, später von Microsoft und (derzeit) Adobe weiterentwickelt. Das Format unterstützt Grauwertbilder, Indexbilder und Vollfarbenbilder. TIFF-Dateien können mehrere Bilder mit unterschiedlichen Eigenschaften enthalten. TIFF spezifiziert zudem eine Reihe unterschiedlicher Kompressionsverfahren (u. a. LZW, ZIP, CCITT und JPEG) und Farbräume, sodass es beispielsweise möglich ist, mehrere Varianten eines Bilds in verschiedenen Größen und Darstellungsformen gemeinsam in einer TIFF-Datei abzulegen. TIFF findet eine breite Verwendung als universelles Austauschformat, zur Archivierung von Dokumenten, in wissenschaftlichen Anwendungen, in der Digitalfotografie oder in der digitalen Film- und Videoproduktion.

Die Stärke dieses Bildformats liegt in seiner Architektur (Abb. 2.7), die es erlaubt, neue Bildmodalitäten und Informationsblöcke durch Definition neuer „Tags“ zu definieren. So können etwa in ImageJ Bilder mit Gleitkommawerten (`float`) problemlos als TIFF-Bilder gespeichert und (allerdings nur mit ImageJ) wieder gelesen werden. In dieser Flexibilität

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

[5] Spezielle Varianten von PS-, EPS- und PDF-Dateien werden allerdings auch als (editierbare) Austauschformate für Raster- und Vektordaten verwendet, z. B. für Adobe *Photoshop* (Photoshop-EPS) oder *Illustrator* (AI).

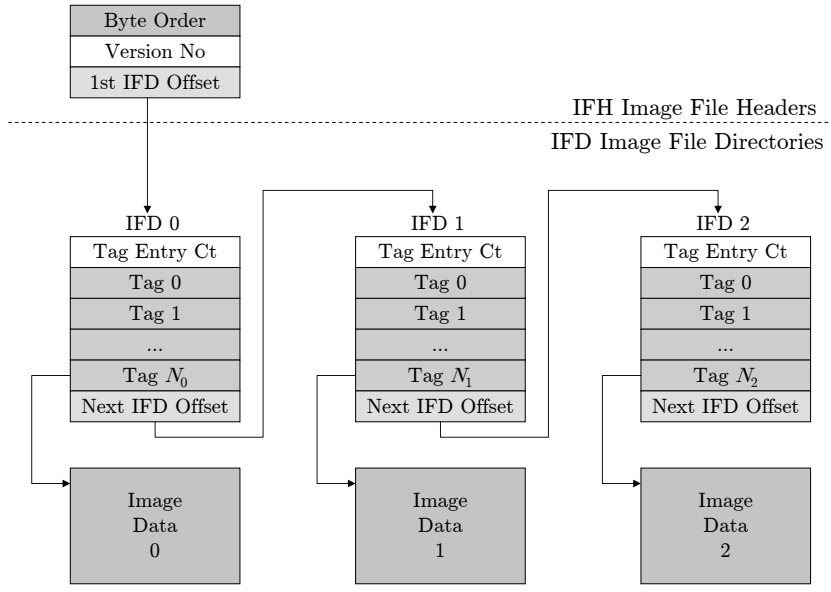


Abbildung 2.7
Struktur einer TIFF-Datei (Beispiel). Eine TIFF-Datei besteht aus dem Header und einer verketteten Folge von (in diesem Fall 3) Bildobjekten, die durch „Tags“ und zugehörige Parameter gekennzeichnet sind und wiederum Verweise auf die eigentlichen Bilddaten (Image Data) enthalten.

liegt aber auch ein Problem, nämlich dass proprietäre Tags nur vereinzelt unterstützt werden und daher „Unsupported Tag“-Fehler beim Öffnen von TIFF-Dateien nicht selten sind. Auch ImageJ kann nur einige wenige Varianten von (unkomprimierten) TIFF-Dateien lesen[6] und auch von den derzeit gängigen Web-Browsern wird TIFF nicht unterstützt.

2.3.3 Graphics Interchange Format (GIF)

GIF wurde ursprünglich (ca. 1986) von CompuServe für Internet-Anwendungen entwickelt und ist auch heute noch weit verbreitet. GIF ist ausschließlich für Indexbilder (Farb- und Grauwertbilder mit maximal 8-Bit-Indizes) konzipiert und ist damit kein Vollfarbenformat. Es werden Farbtabellen unterschiedlicher Größe mit $2 \ldots 256$ Einträgen unterstützt, wobei ein Farbwert als transparent markiert werden kann. Dateien können als „Animated GIFs“ auch mehrere Bilder gleicher Größe enthalten.

GIF verwendet (neben der verlustbehafteten Farbquantisierung – siehe Abschn. 12.5) das verlustfreie LZW-Kompressionsverfahren für die Bild- bzw. Indexdaten. Wegen offener Lizenzfragen bzgl. des LZW-Verfahrens stand die Weiterverwendung von GIF längere Zeit in Frage und es wurde deshalb sogar mit PNG (s. unten) ein Ersatzformat entwickelt. Mittlerweile sind die entsprechenden Patente jedoch abgelaufen und damit dürfte auch die Zukunft von GIF gesichert sein.

[6] Das `ImageIO`-Plugin bietet allerdings eine erweiterte Unterstützung für TIFF-Dateien (http://ij-plugins.sourceforge.net/plugins/imageio/).

Das GIF-Format eignet sich gut für „flache“ Farbgrafiken mit nur wenigen Farbwerten (z. B. typische Firmenlogos), Illustrationen und 8-Bit-Grauwertbilder. Bei neueren Entwicklungen sollte allerdings PNG als das modernere Format bevorzugt werden, zumal es GIF in jeder Hinsicht ersetzt oder übertrifft.

2.3.4 Portable Network Graphics (PNG)

PNG (ausgesprochen „ping“) wurde ursprünglich entwickelt, um (wegen der erwähnten Lizenzprobleme mit der LZW-Kompression) GIF zu ersetzen und gleichzeitig ein universelles Bildformat für Internet-Anwendungen zu schaffen. PNG unterstützt grundsätzlich drei Arten von Bildern:

- Vollfarbbilder (mit bis zu 3×16 Bits/Pixel)
- Grauwertbilder (mit bis zu 16 Bits/Pixel)
- Indexbilder (mit bis zu 256 Farben)

Ferner stellt PNG einen Alphakanal (Transparenzwert) mit maximal 16 Bit (im Unterschied zu GIF mit nur 1 Bit) zur Verfügung. Es wird nur ein Bild pro Datei gespeichert, dessen Größe allerdings Ausmaße bis $2^{30} \times 2^{30}$ Pixel annehmen kann. Als (verlustfreies) Kompressionsverfahren wird eine Variante von PKZIP („Phil Katz“ ZIP) verwendet. PNG sieht keine verlustbehaftete Kompression vor und kommt daher insbesondere nicht als Ersatz für JPEG in Frage. Es kann jedoch GIF in jeder Hinsicht (außer bei Animationen) ersetzen und ist auch das derzeit einzige unkomprimierte (verlustfreie) Vollfarbenformat für Web-Anwendungen.

2.3.5 JPEG

Der JPEG-Standard definiert ein Verfahren zur Kompression von kontinuierlichen Grauwert- und Farbbildern, wie sie vor allem bei natürlichen fotografischen Aufnahmen entstehen. Entwickelt von der „Joint Photographic Experts Group“ (JPEG)[7] mit dem Ziel einer durchschnittlichen Datenreduktion um den Faktor 1 : 16, wurde das Verfahren 1990 als ISO-Standard IS-10918 etabliert und ist heute das meistverwendete Darstellungsformat für Bilder überhaupt. In der Praxis erlaubt JPEG – je nach Anwendung – die Kompression von 24-Bit-Farbbildern bei akzeptabler Bildqualität im Bereich von 1 Bit pro Pixel, also mit einem Kompressionsfaktor von ca. 1 : 25. Der JPEG-Standard sieht Bilder mit bis zu 256 (Farb-)Komponenten vor, eignet sich also insbesondere auch zur Darstellung von CMYK-Bildern (siehe Abschn. 12.2.5).

Das JPEG-Kompressionsverfahren ist vergleichsweise aufwendig [58] und sieht neben dem „Baseline“-Algorithmus mehrere Varianten vor, u. a. auch eine unkomprimierte Version, die allerdings selten verwendet wird. Im Kern besteht es für RGB-Farbbilder aus folgenden drei Hauptschritten:

[7] www.jpeg.org

1. **Farbraumkonversion und Downsampling:** Zunächst werden durch eine Farbtransformation vom RGB- in den YC_bC_r-Raum (siehe Abschn. 12.2.4) die eigentlichen Farbkomponenten C_b, C_r von der Helligkeitsinformation Y getrennt. Die Unempfindlichkeit des menschlichen Auges gegenüber schnellen Farbänderungen erlaubt nachfolgend eine gröbere Abtastung der Farbkomponenten ohne subjektiven Qualitätsverlust, aber verbunden mit einer signifikanten Datenreduktion.
2. **Kosinustransformation und Quantisierung im Spektralraum:** Das Bild wird nun in regelmäßige 8×8-Blöcke aufgeteilt und für jeden der Blöcke wird unabhängig das Frequenzspektrum mithilfe der diskreten Kosinustransformation berechnet (siehe Kap. 15). Nun erfolgt eine Quantisierung der jeweils 64 Spektralkoeffizienten jedes Blocks anhand einer Quantisierungstabelle, die letztendlich die Qualität des komprimierten Bilds bestimmt. In der Regel werden vor allem die Koeffizienten der hohen Frequenzen stark quantisiert, die zwar für die „Schärfe“ des Bilds wesentlich sind, deren exakte Werte aber unkritisch sind.
3. **Verlustfreie Kompression:** Abschließend wird der aus den quantisierten Spektralkomponenten bestehende Datenstrom nochmals mit verlustfreien Methoden (Lauflängen- oder Huffman-Kodierung) komprimiert und damit gewissermaßen die letzte noch verbleibende Redundanz entfernt.

Das JPEG-Verfahren kombiniert also mehrere verschiedene, sich ergänzende Kompressionsmethoden. Die tatsächliche Umsetzung ist selbst für die „Baseline“-Version keineswegs trivial und wird durch die seit 1991 existierende Referenzimplementierung der *Independent JPEG Group* (IJG)[8] wesentlich erleichtert. Der Schwachpunkt der JPEG-Kompression, der vor allem bei der Verwendung an ungeeigneten Bilddaten deutlich wird, besteht im Verhalten bei abrupten Übergängen und dem Hervortreten der 8×8-Bildblöcke bei hohen Kompressionsraten. Abb. 2.9 zeigt dazu als Beispiel den Ausschnitt eines Grauwertbilds, das mit verschiedenen Qualitätsfaktoren (Photoshop $Q_{\text{JPG}} = 10, 5, 1$) komprimiert wurde.

JFIF-Fileformat

Entgegen der verbreiteten Meinung ist JPEG *kein* Dateiformat, sondern definiert „nur“ das Verfahren zur Kompression von Bilddaten[9] (Abb. 2.8). Was üblicherweise als JPEG-File bezeichnet wird, ist tatsächlich das „JPEG File Interchange Format“ (JFIF), das von Eric Hamilton und der IJG entwickelt wurde. Der eigentliche JPEG-Standard spezifiziert nur den JPEG-Kompressor und Dekompressor, alle übrigen Elemente sind durch JFIF definiert oder frei wählbar. Auch die als Schritt 1

[8] www.ijg.org

[9] Genau genommen wird im JPEG-Standard nur die Kompression der einzelnen Komponenten und die Struktur des JPEG-Streams definiert.

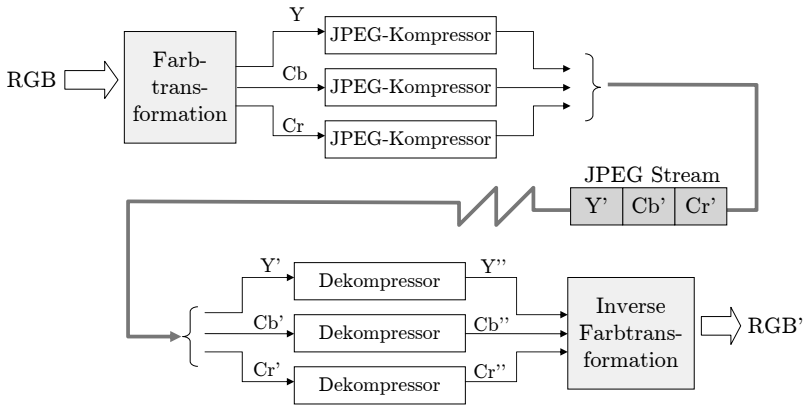


Abbildung 2.8
JPEG-Kompression eines RGB-Bilds. Zunächst werden durch die Farbraumtransformation die Farbkomponenten C_b, C_r von der Luminanzkomponente Y getrennt, wobei die Farbkomponenten gröber abgetastet werden als die Y-Komponente. Alle drei Komponenten laufen unabhängig durch einen JPEG-Kompressor, die Ergebnisse werden in einen gemeinsamen Datenstrom (JPEG Stream) zusammengefügt. Bei der Dekompression erfolgt derselbe Vorgang in umgekehrter Reihenfolge.

des JPEG-Verfahrens angeführte Farbraumtransformation und die Verwendung eines spezifischen Farbraums ist nicht Teil des eigentlichen JPEG-Standards, sondern erst durch JFIF spezifiziert. Die Verwendung unterschiedlicher Abtastraten für Farbe und Luminanz ist dabei lediglich eine praktische Konvention, grundsätzlich sind beliebige Abtastraten zulässig.

Exchangeable Image File Format (EXIF)

EXIF ist eine Variante des JPEG/JFIF-Formats zur Speicherung von Bilddaten aus Digitalkameras, das vor allem zusätzliche Metadaten über Kameratyp, Aufnahmeparameter usw. in standardisierter Form transportiert. EXIF wurde von der Japan Electronics and Information Technology Industries Association (JEITA) als Teil der DCF[10]-Richtlinie entwickelt und wird heute von praktisch allen Herstellern als Standardformat für die Speicherung von Digitalbildern auf Memory-Karten eingesetzt. Interessanterweise verwendet EXIF intern wiederum TIFF-kodierte Bilddaten für Vorschaubilder und ist so aufgebaut, dass Dateien auch von den meisten JPEG/JFIF-Readern problemlos gelesen werden können.

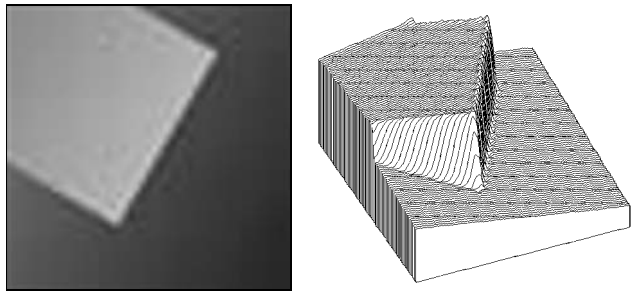
JPEG-2000

Dieses seit 1997 entwickelte und als ISO-ITU-Standard („Coding of Still Pictures“)[11] genormte Verfahren versucht, die bekannten Schwächen des traditionellen JPEG-Verfahrens zu beseitigen. Zum einen werden mit 64×64 deutlich größere Bildblöcke verwendet, zum anderen wird die Kosinustranformation durch eine diskrete *Wavelet*-Transformation ersetzt, die durch ihre lokale Begrenztheit vor allem bei raschen Bildübergängen Vorteile bietet. Das Verfahren erlaubt gegenüber JPEG deutlich höhere

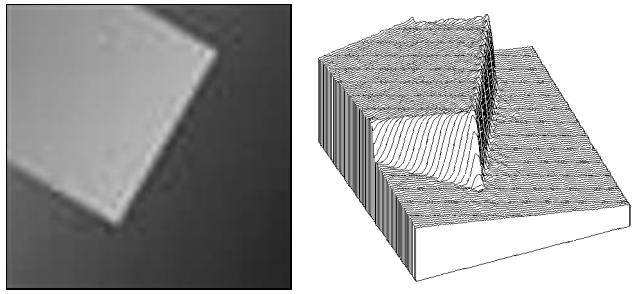
[10] Design Rule for Camera File System.
[11] www.jpeg.org/JPEG2000.htm

Abbildung 2.9
Artefakte durch JPEG-Kompression. Ausschnitt aus dem Originalbild (a) und JPEG-komprimierte Varianten mit Qualitätsfaktor $Q_{\mathrm{JPG}} = 10$ (b), $Q_{\mathrm{JPG}} = 5$ (c) und $Q_{\mathrm{JPG}} = 1$ (d). In Klammern angegeben sind die resultierenden Dateigrößen für das Gesamtbild (Größe 274×274).

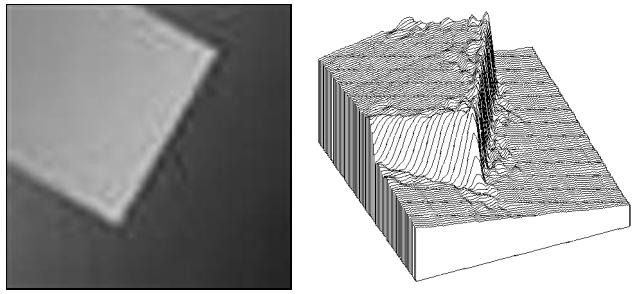
(a) Original (75.08 kB)



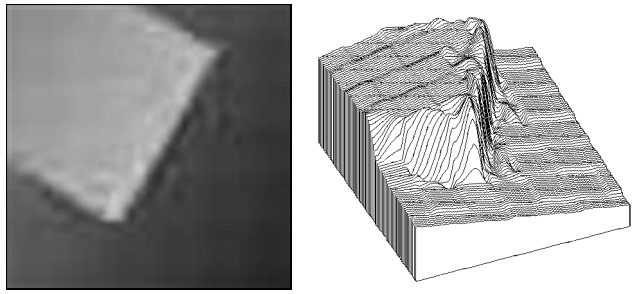
(b) $Q_{\mathrm{JPG}} = 10$ (11.40 kB)



(c) $Q_{\mathrm{JPG}} = 5$ (7.24 kB)



(d) $Q_{\mathrm{JPG}} = 1$ (5.52 kB)



Kompressionsraten von bis zu 0.25 Bit/Pixel bei RGB-Farbbildern. Bedauerlicherweise wird jedoch JPEG-2000 derzeit trotz seiner überlegenen Eigenschaften nur von wenigen Bildbearbeitungsprogrammen und Web-Browsern unterstützt.[12]

2.3.6 Windows Bitmap (BMP)

BMP ist ein einfaches und vor allem unter Windows weit verbreitetes Dateiformat für Grauwert-, Index- und Vollfarbenbilder. Auch Binärbilder werden unterstützt, wobei allerdings in weniger effizienter Weise jedes Pixel als ein ganzes Byte gespeichert wird. Zur Kompression wird optional eine einfache (und verlustfreie) Lauflängenkodierung verwendet. BMP ist bzgl. seiner Möglichkeiten ähnlich zu TIFF, allerdings deutlich weniger flexibel.

2.3.7 Portable Bitmap Format (PBM)

Die PBM-Familie[13] besteht aus einer Reihe sehr einfacher Dateiformate mit der Besonderheit, dass die Bildwerte optional in Textform gespeichert werden können, damit direkt lesbar und sehr leicht aus einem Programm oder mit einem Texteditor zu erzeugen sind. In Abb. 2.10 ist ein einfaches Beispiel gezeigt. Die Zeichen `P2` in der ersten Zeile markie-

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

Abbildung 2.10
Beispiel für eine PGM-Datei im Textformat (links) und das resultierende Grauwertbild (unten).



ren die Datei als PGM im („plain“) Textformat, anschließend folgt eine Kommentarzeile (`#`). In Zeile 3 ist die Bildgröße (Breite 17, Höhe 7) angegeben, Zeile 4 definiert den maximalen Pixelwert (`255`). Die übrigen Zeilen enthalten die tatsächlichen Pixelwerte. Rechts das entsprechende Grauwertbild.

Zusätzlich gibt es jeweils einen „RAW“-Modus, in dem die Pixelwerte als Binärdaten (Bytes) gespeichert sind. PBM ist vor allem unter Unix gebräuchlich und stellt folgende Formate zur Verfügung: PBM (*portable bit map*) für Binär- bzw. *Bitmap*-Bilder, PGM (*portable gray map*) für Grauwertbilder und PNM (*portable any map*) für Farbbilder. PGM-Bilder können auch mit ImageJ geöffnet werden.

[12] Auch in ImageJ wird JPEG-2000 derzeit nicht unterstützt.
[13] http://netpbm.sourceforge.net

2.3.8 Weitere Dateiformate

Für die meisten praktischen Anwendungen sind zwei Dateiformate ausreichend: TIFF als universelles Format für beliebige Arten von unkomprimierten Bildern und JPEG/JFIF für digitale Farbfotos, wenn der Speicherbedarf eine Rolle spielt. Für Web-Anwendungen ist zusätzlich noch PNG oder GIF erforderlich. Darüber hinaus existieren zahlreiche weitere Dateiformate, die zum Teil nur mehr in älteren Datenbeständen vorkommen oder aber in einzelnen Anwendungsbereichen traditionell in Verwendung sind:

- **RGB** ist ein einfaches Bildformat von Silicon Graphics.
- **RAS** (Sun Raster Format) ist ein einfaches Bildformat von Sun Microsystems.
- **TGA** (Truevision Targa File Format) war das erste 24-Bit-Dateiformat für PCs, bietet zahlreiche Bildformate mit 8–32 Bit und wird u. a. in der Medizin und Biologie immer noch häufig verwendet.
- **XBM/XPM** (X-Windows Bitmap/Pixmap) ist eine Familie von ASCII-kodierten Bildformaten unter X-Windows, ähnlich PBM/PGM (s. oben).

2.3.9 Bits und Bytes

Das Öffnen von Bilddateien sowie das Lesen und Schreiben von Bilddaten wird heute glücklicherweise meistens von fertigen Softwarebibliotheken erledigt. Dennoch kann es vorkommen, dass man sich mit der Struktur und dem Inhalt von Bilddateien bis hinunter auf die Byte-Ebene befassen muss, etwa wenn ein nicht unterstütztes Dateiformat zu lesen ist oder wenn die Art einer vorliegenden Datei unbekannt ist.

Big-Endian und *Little-Endian*

Das in der Computertechnik übliche Modell einer Datei besteht aus einer einfachen Folge von Bytes (= 8 Bits), wobei ein Byte auch die kleinste Einheit ist, die man aus einer Datei lesen oder in sie schreiben kann. Im Unterschied dazu sind die den Bildelementen entsprechenden Datenobjekte im Speicher meist größer als ein Byte, beispielsweise eine 32 Bit große `int`-Zahl (= 4 Bytes) für ein RGB-Farbpixel. Das Problem dabei ist, dass es für die *Anordnung* der 4 einzelnen Bytes in der zugehörigen Bilddatei verschiedene Möglichkeiten gibt. Um aber die ursprünglichen Farbpixel wieder korrekt herstellen zu können, muss natürlich bekannt sein, in welcher Reihenfolge die zugehörigen Bytes in der Datei gespeichert sind.

Angenommen wir hätten eine 32-Bit-`int`-Zahl z mit dem Binär- bzw. Hexadezimalwert[14]

[14] Der Dezimalwert von z ist 305419896.

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

dann ist $00010010_B = 12_H$ der Wert des *Most Significant Byte* (MSB) und $01111000_B = 78_H$ der Wert des *Least Significant Byte* (LSB). Sind die einzelnen Bytes innerhalb der Datei in der Reihenfolge von MSB nach LSB gespeichert, dann nennt man die Anordnung „Big Endian", im umgekehrten Fall „Little Endian". Für die Zahl z aus Gl. 2.2 heißt das konkret:

Anordnung	Bytefolge	1	2	3	4
Big Endian	MSB → LSB	12_H	34_H	56_H	78_H
Little Endian	LSB → MSB	78_H	56_H	34_H	12_H

Obwohl die richtige Anordnung der Bytes eigentlich eine Aufgabe des Betriebssystems (bzw. des Filesystems) sein sollte, ist sie in der Praxis hauptsächlich von der Prozessorarchitektur abhängig![15] So sind etwa Prozessoren aus der Intel-Familie (x86, Pentium) traditionell *little-endian* und Prozessoren anderer Hersteller (wie IBM, MIPS, Motorola, Sun) *big-endian*, was meistens auch für die zugeordneten Betriebs- und Filesysteme gilt.[16] *big-endian* wird auch als *Network Byte Order* bezeichnet, da im IP-Protokoll die Datenbytes in der Reihenfolge MSB nach LSB übertragen werden.

Zur richtigen Interpretation einer Bilddatei ist daher die Kenntnis der für größere Speicherworte verwendeten Byte-Anordnung erforderlich. Diese ist meistens fix, bei einzelnen Dateiformaten (wie beispielsweise TIFF) jedoch variabel und als Parameter im Dateiheader angegeben (siehe Tabelle 2.2).

Dateiheader und Signaturen

Praktisch alle Bildformate sehen einen Dateiheader vor, der die wichtigsten Informationen über die nachfolgenden Bilddaten enthält, wie etwa den Elementtyp, die Bildgröße usw. Die Länge und Struktur dieses Headers ist meistens fix, in einer TIFF-Datei beispielsweise kann der Header aber wieder Verweise auf weitere Subheader enthalten.

Um die Information im Header überhaupt interpretieren zu können, muss zunächst der Dateityp festgestellt werden. In manchen Fällen ist dies auf Basis der *file name extension* (z. B. `.jpg` oder `.tif`) möglich, jedoch sind diese Abkürzungen nicht standardisiert, können vom Benutzer jederzeit geändert werden und sind in manchen Betriebssystemen (z. B.

[15] Das hat vermutlich historische Gründe. Wenigstens ist aber die Reihenfolge der *Bits* innerhalb eines Byte weitgehend einheitlich.

[16] In *Java* ist dies übrigens kein Problem, da intern in allen Implementierungen (der *Java Virtual Machine*) und auf allen Plattformen *big-endian* als einheitliche Anordnung verwendet wird.

MacOS) überhaupt nicht üblich. Stattdessen identifizieren sich viele Dateiformate durch eine eingebettete „Signatur“, die meist aus zwei Bytes am Beginn der Datei gebildet wird. Einige Beispiele für gängige Bildformate und zugehörige Signaturen sind in Tabelle 2.2 angeführt. Die meisten Bildformate können durch Inspektion der ersten Bytes der Datei identifiziert werden. Die Zeichenfolge ist jeweils hexadezimal (`0x..`) und als ASCII-Text dargestellt. So beginnt etwa eine PNG-Datei immer mit einer Folge aus den vier Byte-Werten `0x89`, `0x50`, `0x4e`, `0x47`, bestehend aus der „magic number“ `0x89` und der ASCII-Zeichenfolge „PNG“. Beim TIFF-Format geben hingegen die ersten beiden Zeichen (`II` für „Intel“ bzw. `MM` für „Motorola“) Auskunft über die Byte-Reihenfolge (*little-endian* bzw. *big-endian*) der nachfolgenden Daten.

Tabelle 2.2
Beispiele für Signaturen von Bilddateien. Die meisten Bildformate können durch Inspektion der ersten Bytes der Datei identifiziert werden. Die Zeichenfolge ist jeweils hexadezimal (`0x..`) und als ASCII-Text dargestellt (□ steht für ein nicht druckbares Zeichen).

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

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

2.4 Aufgaben

Aufg. 2.1. Ermitteln Sie die wirklichen Ausmaße (in mm) eines Bilds mit 1400×1050 quadratischen Pixel und einer Auflösung von 72 dpi.

Aufg. 2.2. Eine Kamera mit einer Brennweite von $f = 50$ mm macht eine Aufnahme eines senkrechten Mastes, der 12 m hoch ist und sich im Abstand von 95 m vor der Kamera befindet. Ermitteln Sie die Höhe der dabei entstehenden Abbildung (a) in mm und (b) in der Anzahl der Pixel unter der Annahme, dass der Kamerasensor eine Auflösung von 4000 dpi aufweist.

Aufg. 2.3. Der Bildsensor einer Digitalkamera besitzt 2016×3024 Pixel. Die Geometrie dieses Sensors ist identisch zu der einer herkömmlichen Kleinbildkamera (mit einer Bildgröße von 24×36 mm), allerdings um den Faktor 1.6 kleiner. Berechnen Sie die Auflösung dieses Sensors in dpi.

Aufg. 2.4. Überlegen Sie unter Annahme der Kamerageometrie aus Aufg. 2.3 und einer Objektivbrennweite von $f = 50$ mm, welche Verwischung (in Pixel) eine gleichförmige, horizontale Kameradrehung um 0.1 Grad innerhalb einer Belichtungszeit von $\frac{1}{30}$ s bewirkt. Berechnen Sie das Gleiche auch für $f = 300$ mm. Überlegen Sie, ob das Ausmaß der Verwischung auch von der Entfernung der Objekte abhängig ist.

Aufg. 2.5. Ermitteln Sie die Anzahl von Bytes, die erforderlich ist, um ein unkomprimiertes Binärbild mit 4000×3000 Pixel zu speichern.

Aufg. 2.6. Ermitteln Sie die Anzahl von Bytes, die erforderlich ist, um ein unkomprimiertes RGB-Farbbild der Größe 640×480 mit 8, 10, 12 bzw. 14 Bit pro Farbkanal zu speichern.

Aufg. 2.7. Nehmen wir an, ein Schwarz-Weiß-Fernseher hat eine Bildfläche von 625×512 Pixel mit jeweils 8 Bits und zeigt 25 Bilder pro Sekunde. (a) Wie viele verschiedene Bilder kann dieses Gerät grundsätzlich anzeigen und wie lange müsste man (ohne Schlafpausen) davor sitzen, um jedes mögliche Bild mindestens einmal gesehen zu haben? (b) Erstellen Sie dieselbe Berechnung für einen Farbfernseher mit jeweils 3×8 Bit pro Pixel.

Aufg. 2.8. Zeigen Sie, dass eine Gerade im dreidimensionalen Raum von einer Lochkamera (d. h. bei einer perspektivischen Projektion, Gl. 2.1) tatsächlich immer als Gerade abgebildet wird.

Aufg. 2.9. Erzeugen Sie mit einem Texteditor analog zu Abb. 2.10 eine PGM-Datei `disk.pgm`, die das Bild einer hellen, kreisförmigen Scheibe enthält, und öffnen Sie das Bild anschließend mit ImageJ. Versuchen Sie andere Programme zu finden, mit denen sich diese Datei öffnen und darstellen lässt.

3

ImageJ

Bis vor wenigen Jahren war die Bildverarbeitungs-“Community“ eine relativ kleine Gruppe von Personen, die entweder Zugang zu teuren Bildverarbeitungswerkzeugen hatte oder – aus Notwendigkeit – damit begann, eigene Softwarepakete für die digitale Bildverarbeitung zu programmieren. Meistens begannen solche „Eigenbau“-Umgebungen mit kleinen Programmkomponenten zum Laden und Speichern von Bildern, von und auf Dateien. Das war nicht immer einfach, denn oft hatte man es mit mangelhaft dokumentierten oder firmenspezifischen Dateiformaten zu tun. Die nahe liegendste Lösung war daher häufig, zunächst sein eigenes, für den jeweiligen Einsatzbereich „optimales“ Dateiformat zu entwerfen, was weltweit zu einer Vielzahl verschiedenster Dateiformate führte, von denen viele heute glücklicherweise wieder vergessen sind [61]. Das Schreiben von Programmen zur Konvertierung zwischen diesen Formaten war daher in den 1980ern und frühen 1990ern eine wichtige Angelegenheit. Die Darstellung von Bildern auf dem Bildschirm war ähnlich schwierig, da es dafür wenig Unterstützung vonseiten der Betriebssysteme und Systemschnittstellen gab. Es dauerte daher oft Wochen oder sogar Monate, bevor man am Computer auch nur elementare Dinge mit Bildern tun konnte und bevor man vor allem an die Entwicklung neuer Algorithmen für die Bildverarbeitung denken konnte.

Glücklicherweise ist heute vieles anders. Nur wenige, wichtige Bildformate haben überlebt (s. auch Abschn. 2.3) und sind meist über fertige Programmbibliotheken leicht zugreifbar. Die meisten Standard-APIs, z. B. für C++ und Java, beinhalten bereits eine Basisunterstützung für Bilder und andere digitale Mediendaten.

3.1 Software für digitale Bilder

Traditionell ist Software für digitale Bilder entweder zur Bearbeitung von Bildern oder zum Programmieren ausgelegt, also entweder für den Praktiker und Designer oder für den Programmentwickler.

3.1.1 Software zur Bildbearbeitung

Softwareanwendungen für die Manipulation von Bildern, wie z. B. Adobe Photoshop, Corel Paint u. v. a., bieten ein meist sehr komfortables User Interface und eine große Anzahl fertiger Funktionen und Werkzeuge, um Bilder interaktiv zu bearbeiten. Die Erweiterung der bestehenden Funktionalität durch *eigene* Programmkomponenten wird zwar teilweise unterstützt, z. B. können „Plugins“ für Photoshop[1] in C++ programmiert werden, doch ist dies eine meist aufwendige und jedenfalls für Programmieranfänger zu komplexe Aufgabe.

3.1.2 Software zur Bildverarbeitung

Im Gegensatz dazu unterstützt „echte“ Software für die digitale Bildverarbeitung primär die Erfordernisse von Algorithmenentwicklern und Programmierern und bietet dafür normalerweise weniger Komfort und interaktive Möglichkeiten für die Bildbearbeitung. Stattdessen bieten diese Umgebungen meist umfassende und gut dokumentierte Programmbibliotheken, aus denen relativ einfach und rasch neue Prototypen und Anwendungen erstellt werden können. Beispiele dafür sind etwa *Khoros/VisiQuest*[2], *IDL*[3], *MatLab*[4] und *ImageMagick*[5]. Neben der Möglichkeit zur konventionellen Programmierung (üblicherweise mit C/C++) werden häufig einfache Scriptsprachen und visuelle Programmierhilfen angeboten, mit denen auch komplizierte Abläufe auf einfache und sichere Weise konstruiert werden können.

3.2 Eigenschaften von ImageJ

ImageJ, das wir für dieses Buch verwenden, ist eine Mischung beider Welten. Es bietet einerseits bereits fertige Werkzeuge zur Darstellung und interaktiven Manipulation von Bildern, andererseits lässt es sich extrem einfach durch eigene Softwarekomponenten erweitern. ImageJ ist vollständig in Java implementiert, ist damit weitgehend plattformunabhängig und läuft unverändert u. a. unter Windows, MacOS und Linux. Die dynamische Struktur von Java ermöglicht es, eigene Module

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

– so genannte „Plugins“ – in Form eigenständiger Java-Codestücke zu erstellen und „on-the-fly“ im laufenden System zu übersetzen und auch sofort auszuführen, ohne ImageJ neu starten zu müssen. Dieser schnelle Ablauf macht ImageJ zu einer idealen Basis, um neue Bildverarbeitungsalgorithmen zu entwickeln und mit ihnen zu experimentieren. Da Java an vielen Ausbildungseinrichtungen immer häufiger als erste Programmiersprache unterrichtet wird, ist das Erlernen einer zusätzlichen Programmiersprache oft nicht notwendig und der Einstieg für viele Studierende sehr einfach. ImageJ ist zudem frei verfügbar, sodass Studierende und Lehrende die Software legal und ohne Lizenzkosten auf allen ihren Computern verwenden können. ImageJ ist daher eine ideale Basis für die Ausbildung in der digitalen Bildverarbeitung, es wird aber auch in vielen Labors, speziell in der Biologie und Medizin, für die tägliche Arbeit eingesetzt.

Entwickelt wurde (und wird) ImageJ von Wayne Rasband [66] am U.S. *National Institutes of Health* (NIH) als Nachfolgeprojekt der älteren Software *NIH-Image*, die allerdings nur auf MacIntosh verfügbar war. Die aktuelle Version von ImageJ, Updates, Dokumentation, Testbilder und eine ständig wachsende Sammlung beigestellter Plugins finden sich auf der ImageJ-Homepage.[6] Praktisch ist auch, dass der gesamte Quellcode von ImageJ online zur Verfügung steht. Die Installation von ImageJ ist einfach, Details dazu finden sich in der Online-Installationsanleitung, im IJ-Tutorial [3] und auch in Anhang C.

ImageJ ist allerdings nicht perfekt und weist softwaretechnisch sogar erhebliche Mängel auf, wohl aufgrund seiner Entstehungsgeschichte. Die Architektur ist unübersichtlich und speziell die Unterscheidung zwischen den häufig verwendeten `ImageProcessor`- und `ImagePlus`-Objekten bereitet nicht nur Anfängern erhebliche Schwierigkeiten. Die Implementierung einzelner Komponenten ist zum Teil nicht konsistent und unterschiedliche Funktionalitäten sind oft nicht sauber voneinander getrennt. Auch die fehlende Orthogonalität ist ein Problem, d. h., ein und dieselbe Funktionalität kann oft auf mehrfache Weise realisiert werden. Die Zusammenstellung im Anhang ist daher vorrangig nach Funktionen gruppiert, um die Übersicht zu erleichtern.

3.2.1 Features

Als reine Java-Anwendung läuft ImageJ auf praktisch jedem Computer, für den eine aktuelle Java-Laufzeitumgebung (Java *runtime environment*, „jre“) existiert. Bei der Installation von ImageJ wird ein eigenes Java-Runtime mitgeliefert, sodass Java selbst nicht separat installiert sein muss. ImageJ kann, unter den üblichen Einschränkungen, auch als Java-*Applet* innerhalb eines Web-Browsers betrieben werden, meistens wird es jedoch als selbstständige Java-Applikation verwendet.

[6] http://rsb.info.nih.gov/ij/. Die für dieses Buch verwendete Version ist 1.33h.

ImageJ kann sogar serverseitig, z. B. für Bildverarbeitungsoperationen in Online-Anwendungen eingesetzt werden [3]. Zusammengefasst sind die wichtigsten Eigenschaften von ImageJ:

- Ein Satz von fertigen Werkzeugen zum Erzeugen, Visualisieren, Editieren, Verarbeiten, Analysieren, Öffnen und Speichern von Bildern in mehreren Dateiformaten. ImageJ unterstützt auch „tiefe“ Integer-Bilder mit 16 und 32 Bits sowie Gleitkommabilder und Bildfolgen (sog. „stacks“).
- Ein einfacher Plugin-Mechanismus zur Erweiterung der Basisfunktionalität durch kleine Java-Codesegmente. Dieser ist die Grundlage aller Beispiele in diesem Buch.
- Eine Makro-Sprache[7] und ein zugehöriger Interpreter, die es erlauben, ohne Java-Kenntnisse bestehende Funktionen zu größeren Verarbeitungsfolgen zu verbinden. ImageJ-Makros werden in diesem Buch nicht eingesetzt.

3.2.2 Fertige Werkzeuge

Nach dem Start öffnet ImageJ zunächst sein Hauptfenster (Abb. 3.1), das mit folgenden Menü-Einträgen die eingebauten Werkzeuge zur Verfügung stellt:

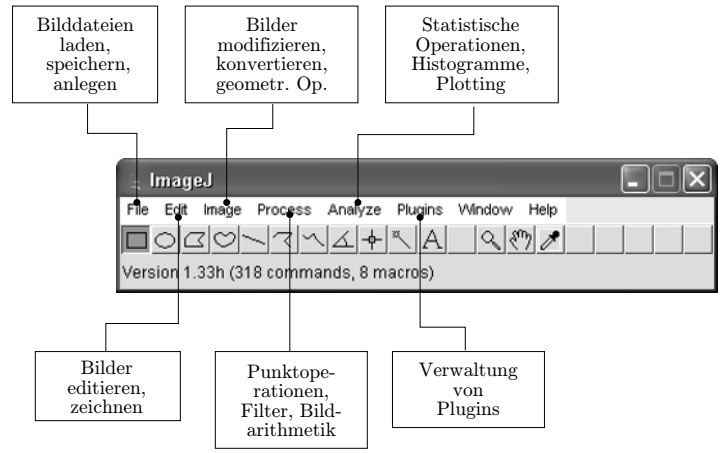


Abbildung 3.1
Hauptfenster von ImageJ (unter Windows-XP).

- File: Zum Laden und Speichern von Bildern sowie zum Erzeugen neuer Bilder.
- Edit: Zum Editieren und Zeichnen in Bildern.
- Image: Zur Modifikation und Umwandlung von Bildern sowie für geometrische Operationen.

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

- Process: Für typische Bildverarbeitungsoperationen, wie Punktoperationen, Filter und arithmetische Operationen auf Bilder.
- Analyze: Für die statistische Auswertung von Bilddaten, Anzeige von Histogrammen und spezielle Darstellungsformen.
- Plugin: Zum Bearbeiten, Übersetzen, Ausführen und Ordnen eigener Plugins.

ImageJ kann derzeit Bilddateien in mehreren Formaten öffnen, u. a. TIFF (nur unkomprimiert), JPEG, GIF, PNG und BMP, sowie die in der Medizin bzw. Astronomie gängigen Formate DICOM (Digital Imaging and Communications in Medicine) und FITS (Flexible Image Transport System). Wie in ähnlichen Programmen üblich, werden auch in ImageJ alle Operationen auf das aktuell selektierte Bild (current image) angewandt. ImageJ verfügt für die meisten eingebauten Operationen einen „Undo"-Mechanismus, der (auf einen Arbeitsschritt beschränkt) auch die selbst erzeugten Plugins unterstützt.

3.2.3 ImageJ-Plugins

Plugins sind kleine, in Java definierte Softwaremodule, die in einfacher, standardisierter Form in ImageJ eingebunden werden und damit seine Funktionalität erweitern (Abb. 3.2). Zur Verwaltung und Benutzung der

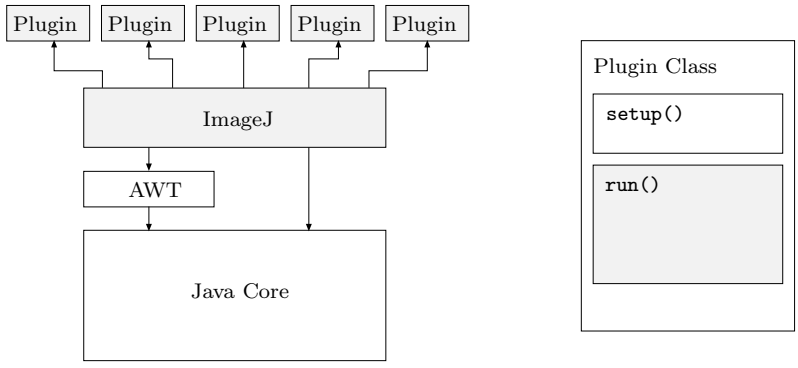


Abbildung 3.2
Software-Struktur von ImageJ (vereinfacht). ImageJ basiert auf dem Java-Kernsystem und verwendet insbesondere Javas AWT (Advanced Windowing Toolkit) als Grundlage für das User Interface und die Darstellung von Bilddaten. Plugins sind kleine Java-Klassen mit einer einfachen Schnittstelle zu ImageJ, mit denen sich die Funktionalität des Systems leicht erweitern lässt.

Plugins stellt ImageJ über das Hauptfenster (Abb. 3.1) ein eigenes Plugin-Menü zur Verfügung. ImageJ ist modular aufgebaut und tatsächlich sind zahlreiche eingebaute Funktionen wiederum selbst als Plugins implementiert. Als Plugins realisierte Funktionen können auch beliebig in einem der Hauptmenüs von ImageJ platziert werden.

Programmstruktur

Technisch betrachtet sind Plugins Java-Klassen, die eine durch ImageJ vorgegebene Interface-Spezifikation implementieren. Es gibt zwei verschiedene Arten von Plugins:

- `PlugIn` benötigt keinerlei Argumente, kann daher auch ohne Beteiligung eines Bilds ausgeführt werden,
- `PlugInFilter` wird beim Start immer ein Bild (das aktuelle Bild) übergeben.

Wir verwenden in diesem Buch fast ausschließlich den zweiten Typ – `PlugInFilter` – zur Realisierung von Bildverarbeitungsoperationen. Ein Plugin vom Typ `PlugInFilter` muss mindestens die folgenden zwei Methoden implementieren (Abb. 3.2) – `setup()` und `run()`:

`int setup (String arg, ImagePlus img)`
: Diese Methode wird bei der Ausführung eines Plugin von ImageJ als erste aufgerufen, vor allem um zu überprüfen, ob die Spezifikationen des Plugin mit dem übergebenen Bild zusammenpassen. Die Methode liefert einen Bitvektor (als `int`-Wert), der die Eigenschaften des Plugin beschreibt.

`void run (ImageProcessor ip)`
: Diese Methode erledigt die tatsächliche Arbeit des Plugin. Der einzige Parameter *ip* (ein Objekt vom Typ `ImageProcessor`) enthält das zu bearbeitende Bild und alle relevanten Informationen dazu. Die `run`-Methode liefert keinen Rückgabewert (`void`), kann aber das übergebene Bild verändern und auch neue Bilder erzeugen.

3.2.4 Beispiel-Plugin: „inverter“

Am besten wir sehen uns diese Sache an einem konkreten Beispiel an. Wir versuchen uns an einem einfachen Plugin, das ein 8-Bit-Grauwertbild invertieren, also ein Positiv in ein Negativ verwandeln soll. Das Invertieren der Intensität ist eine typische Punktoperation, wie wir sie in Kap. 5 im Detail behandeln. Unser Bild hat 8-Bit-Grauwerte im Bereich von 0 bis zum Maximalwert 255 sowie eine Breite und Höhe von M bzw. N Pixel. Die Operation ist sehr einfach: Der Wert jedes einzelnen Bildpixels $I(u, v)$ wird umgerechnet in einen neuen Pixelwert

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

der den ursprünglichen Pixelwert ersetzt, und das für alle Bildkoordinaten $u = 0 \ldots M-1$ und $v = 0 \ldots N-1$.

Plugin-Klasse `MyInverter_`

Die vollständige Auflistung des Java-Codes für dieses Plugin findet sich in Prog. 3.1. Das Programm enthält nach den Import-Anweisungen für die notwendigen Java-Packages die Definition einer einzigen Klasse `MyInverter_` in einer Datei mit (wie in Java üblich) demselben Namen (`MyInverter_.java`). Das Unterstreichungszeichen „_“ am Ende des Namens ist wichtig, da ImageJ nur so diese Klasse als Plugin akzeptiert.

Programm 3.1
ImageJ-Plugin zum Invertieren von 8-Bit-Grauwertbildern (File `MyInverter_.java`).

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

Die `setup()`-Methode

Vor der eigentlichen Ausführung des Plugin, also vor dem Aufruf der `run()`-Methode, wird die `setup()`-Methode vom ImageJ-Kernsystem aufgerufen, um Informationen über das Plugin zu erhalten. In unserem Beispiel wird nur der Wert `DOES_8G` (eine statische `int`-Konstante in der Klasse `PluginFilter`) zurückgegeben, was anzeigt, dass dieses Plugin 8-Bit-Grauwertbilder (`8G`) verarbeiten kann. Die Parameter *`arg`* und *`imp`* der `setup()`-Methode werden in diesem Beispiel nicht benutzt (s. auch Aufg. 3.4).

Die `run()`-Methode

Wie bereits erwähnt, wird der `run()`-Methode ein Objekt `ip` vom Typ `ImageProcessor` übergeben, in dem das zu bearbeitende Bild und zugehörige Informationen enthalten sind. Zunächst werden durch Anwendung der Methoden `getWidth()` und `getHeight()` auf `ip` die Dimensionen des Bilds abgefragt. Dann werden alle Bildkoordinaten in zwei geschachtelten `for`-Schleifen mit den Zählvariablen `u` und `v` horizontal bzw. vertikal durchlaufen. Für den eigentlichen Zugriff auf die Bilddaten werden zwei weitere Methoden der Klasse `ImageProcessor` verwendet:

`int getPixel (int` x`, int` y`)`
: Liefert den Wert des Bildelements an der Position (x, y).

`void putPixel (int x, int y, int a)`
Setzt das Bildelement an der Position (x, y) auf den neuen Wert a.

Editieren, Übersetzen und Ausführen des Plugins

Der Java-Quellcode des Plugins muss in einer Datei `MyInverter_.java` innerhalb des Verzeichnisses `<ij>/plugins/`[8] von ImageJ oder in einem Unterverzeichnis davon abgelegt werden. Neue Plugin-Dateien können über das Plugins→New... von ImageJ angelegt werden. Zum Editieren verfügt ImageJ über einen eingebauten Editor unter Plugins→Edit..., der jedoch für das ernsthafte Programmieren kaum Unterstützung bietet und daher wenig geeignet ist. Besser ist es, dafür einen modernen Editor oder gleich eine komplette Java-Programmierumgebung zu verwenden (unter Windows z. B. *Eclipse*[9], *NetBeans*[10] oder *JBuilder*[11]).

Für die Übersetzung von Plugins (in Java-Bytecode) ist in ImageJ ein eigener Java-Compiler als Teil des Runtime Environments verfügbar.[12] Zur Übersetzung und nachfolgenden Ausführung verwendet man einfach das Menü

Plugins→Compile and Run...,

wobei etwaige Fehlermeldungen über ein eigenes Textfenster angezeigt werden. Sobald das Plugin in den entsprechenden `.class`-File übersetzt ist, wird es auf das aktuelle Bild angewandt. Eine Fehlermeldung zeigt an, falls keine Bilder geöffnet sind oder das aktuelle Bild nicht den Möglichkeiten des Plugins entspricht.

Im Verzeichnis `<ij>/plugins/` angelegte, korrekt benannte Plugins werden beim Starten von ImageJ automatisch als Eintrag im Plugins-Menü installiert und brauchen dann vor der Ausführung natürlich nicht mehr übersetzt zu werden. Plugin-Einträge können manuell mit

Plugins→Shortcuts→Install Plugin..

auch an anderen Stellen des Menübaums platziert werden. Folgen von Plugin-Aufrufen und anderen ImageJ-Kommandos können über

Plugins→Macros→Record

auch automatisch als nachfolgend abrufbare Makros aufgezeichnet werden.

[8] `<ij>` ist das Verzeichnis, in dem ImageJ selbst installiert ist.
[9] www.eclipse.org
[10] www.netbeans.org
[11] www.borland.com
[12] Derzeit nur unter Windows. Angaben zu MacOS und Linux finden sich im ImageJ Installation Manual.

Anzeigen der Ergebnisse und „undo“

Unser Plugin erzeugt kein neues Bild, sondern verändert das ihm übergebene Bild in „destruktiver“ Weise. Das muss nicht immer so sein, denn Plugins können auch neue Bilder erzeugen oder nur z. B. Statistiken berechnen, ohne das übergebene Bild dabei zu modifizieren. Es mag überraschen, dass unser Plugin keinerlei Anweisungen für das neuerliche Anzeigen des Bilds enthält – das erledigt ImageJ automatisch, sobald es annehmen muss, dass ein Plugin das übergebene Bild verändert hat. Außerdem legt ImageJ vor jedem Aufruf der `run()`-Methode eines Plugins automatisch eine Kopie („Snapshot“) des übergebenen Bilds an. Dadurch ist es nachfolgend möglich, über Edit→Undo den ursprünglichen Zustand wieder herzustellen, ohne dass wir in unserem Programm dafür explizite Vorkehrungen treffen müssen.

3.3 Weitere Informationen zu ImageJ und Java

In den nachfolgenden Kapiteln verwenden wir in Beispielen meist konkrete Plugins und Java-Code zur Erläuterung von Algorithmen und Verfahren. Dadurch sind die Beispiele nicht nur direkt anwendbar, sondern sie sollen auch schrittweise zusätzliche Techniken in der Umsetzung mit ImageJ vermitteln. Aus Platzgründen wird allerdings oft nur die `run()`-Methode eines Plugins angegeben und eventuell zusätzliche Klassen- und Methodendefinitionen, sofern sie im Kontext wichtig sind. Der vollständige Quellcode zu den Beispielen ist natürlich auch auf der Website zu diesem Buch[13] zu finden.

3.3.1 Ressourcen für ImageJ

Anhang C enthält eine Übersicht der wichtigsten Möglichkeiten des ImageJ-API. Die vollständige und aktuellste API-Referenz einschließlich Quellcode, Tutorials und vielen Beispielen in Form konkreter Plugins sind auf der offiziellen ImageJ-Homepage verfügbar. Zu empfehlen ist auch das Tutorial von W. Bailer [3], das besonders für das Programmieren von ImageJ-Plugins nützlich ist.

3.3.2 Programmieren mit Java

Die Anforderungen dieses Buchs an die Java-Kenntnisse der Leser sind nicht hoch, jedoch sind elementare Grundlagen erforderlich, um die Beispiele zu verstehen und erweitern zu können. Einführende Bücher sind in großer Zahl auf dem Markt verfügbar, empfehlenswert ist z. B. [60]. Lesern, die bereits Programmiererfahrung besitzen, aber bisher nicht mit Java gearbeitet haben, empfehlen wir u. a. die einführenden Tutorials auf

[13] www.imagingbook.com

der Java-Homepage von Sun Microsystems.[14] Zusätzlich sind in Anhang B einige spezifische Java-Themen zusammengestellt, die in der Praxis häufig Fragen oder Probleme aufwerfen.

3.4 Aufgaben

Aufg. 3.1. Installieren Sie die aktuelle Version von ImageJ auf Ihrem Computer und machen Sie sich mit den eingebauten Funktionen vertraut.

Aufg. 3.2. Verwenden Sie `MyInverter_.java` (Prog. 3.1) als Vorlage, um ein eigenes Plugin zu programmieren, das ein Grauwertbild horizontal (oder vertikal) spiegelt. Testen Sie das neue Plugin anhand geeigneter (auch sehr kleiner) Bilder und überprüfen Sie die Ergebnisse genau.

Aufg. 3.3. Erstellen Sie ein neues Plugin für 8-Bit-Grauwertbilder, das um (d. h. *in*) das übergebene Bild (beliebiger Größe) einen weißen Rahmen (Pixelwert = 255) mit 10 Pixel Breite malt.

Aufg. 3.4. Erstellen Sie ein Plugin, das ein 8-Bit-Grauwertbild horizontal und zyklisch verschiebt, bis der ursprüngliche Zustand wiederhergestellt ist. Um das modifizierte Bild nach jeder Verschiebung am Bildschirm anzeigen zu können, benötigt man eine Referenz auf das zugehörige Bild (`ImagePlus`, nicht `ImageProcessor`), die nur über die `setup()`-Methode zugänglich ist (`setup()` wird immer vor der `run`-Methode aufgerufen). Dazu können wir die Plugin-Definition aus Prog. 3.1 folgendermaßen ändern:

```
1  public class XY_ implements PlugInFilter {
2
3    ImagePlus myimage;    // new instance variable
4
5    public int setup(String arg, ImagePlus img) {
6      myimage = img;    // keep reference to image (img)
7      return DOES_8G;
8    }
9
10   public void run(ImageProcessor ip) {
11     ...
12     myimage.updateAndDraw();  // redraw image
13     ...
14   }
15
16 }
```

[14] http://java.sun.com/j2se/

Aufg. 3.5. Erstellen Sie ein ImageJ-Plugin, das ein Grauwertbild als PGM-Datei („raw“, d. h. in Textform) abspeichert (siehe auch Abb. 2.10). Überprüfen Sie das Resultat mit einem Texteditor und versuchen Sie, Ihre Datei mit ImageJ auch wieder zu öffnen.

4

Histogramme

Histogramme sind Bildstatistiken und ein häufig verwendetes Hilfsmittel, um wichtige Eigenschaften von Bildern rasch zu beurteilen. Insbesondere sind Belichtungsfehler, die bei der Aufnahme von Bildern entstehen, im Histogramm sehr leicht zu erkennen. Moderne Digitalkameras bieten oft die Möglichkeit, das Histogramm eines gerade aufgenommenen Bilds sofort anzuzeigen (Abb. 4.1), da eventuelle Belichtungsfehler



Abbildung 4.1
Digitalkamera mit Histogrammanzeige für das aktuelle Bild.

durch nachfolgende Bearbeitungsschritte nicht mehr korrigiert werden können. Neben Aufnahmefehlern können aus Histogrammen aber auch viele Rückschlüsse auf einzelne Verarbeitungsschritte gezogen werden, denen ein Digitalbild im Laufe seines „Lebens“ unterzogen wurde.

4.1 Was ist ein Histogramm?

Histogramme sind Häufigkeitsverteilungen und Histogramme von Bildern beschreiben die Häufigkeit der einzelnen Intensitätswerte. Am einfachsten ist dies anhand altmodischer Grauwertbilder zu verstehen, ein



Abbildung 4.2
8-Bit-Grauwertbild mit Histogramm, das die Häufigkeitsverteilung der 256 Intensitätswerte anzeigt.

Beispiel dazu zeigt Abb. 4.2. Für ein Grauwertbild I mit möglichen Intensitätswerten im Bereich $I(u,v) \in [0, K-1]$ enthält das zugehörige Histogramm H genau K Einträge, wobei für ein typisches 8-Bit-Grauwertbild $K = 2^8 = 256$ ist. Jeder Histogrammeintrag $H(i)$ ist definiert als

$$\mathsf{h}(i) = \text{ die } \textit{Anzahl} \text{ der Pixel von } I \text{ mit dem Intensitätswert } i$$

für alle $0 \le i < K$. Etwas formaler ausgedrückt ist das

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

$\mathsf{h}(0)$ ist also die Anzahl der Pixel mit dem Wert 0, $\mathsf{h}(1)$ die Anzahl der Pixel mit Wert 1 usw. $\mathsf{h}(255)$ ist schließlich die Anzahl aller weißen Pixel mit dem maximalen Intensitätswert $255 = K-1$. Das Ergebnis der Histogrammberechnung ist ein eindimensionaler Vektor h der Länge K, wie Abb. 4.3 für ein Bild mit $K = 16$ möglichen Intensitätswerten zeigt.

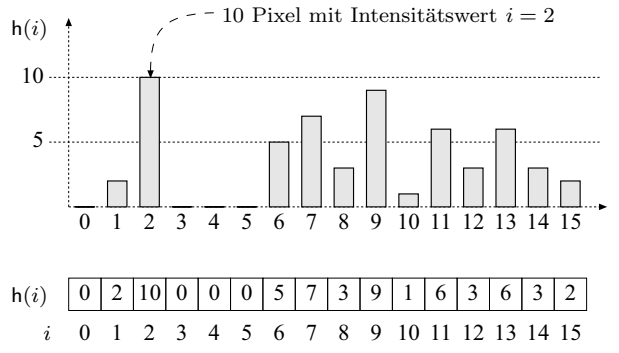


Abbildung 4.3
Histogrammvektor für ein Bild mit $K = 16$ möglichen Intensitätswerten. Der Index der Vektorelemente $i = 0 \ldots 15$ ist der Intensitätswert. Ein Wert von 10 in Zelle 2 bedeutet, dass das zugehörige Bild 10 Pixel mit dem Intensitätswert 2 aufweist.

Offensichtlich enthält ein Histogramm keinerlei Informationen darüber, *woher* die einzelnen Einträge ursprünglich stammen, d. h., jede räumliche Information über das zugehörige Bild geht im Histogramm verloren. Das ist durchaus beabsichtigt, denn die Hauptaufgabe eines

[1] card{...} bezeichnet die Anzahl der Elemente („Kardinalität“) einer Menge (s. auch S. 431).

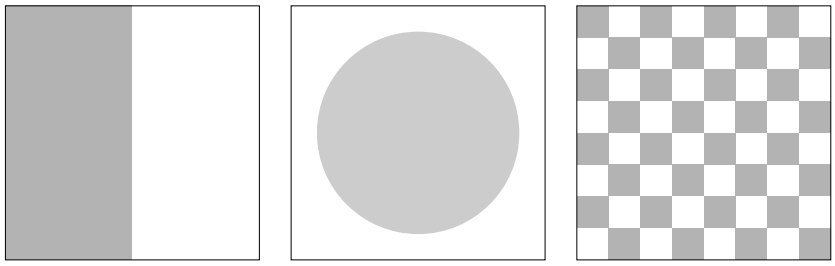


Abbildung 4.4
Drei recht unterschiedliche Bilder mit identischen Histogrammen.

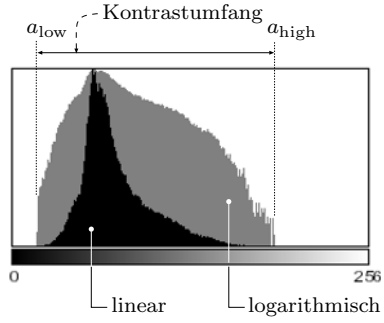


Abbildung 4.5
Effektiv genutzter Bereich von Intensitätswerten. Die Grafik zeigt die Häufigkeiten der Pixelwerte in linearer Darstellung (schwarze Balken) und logarithmischer Darstellung (graue Balken). In der logarithmischen Form werden auch relativ kleine Häufigkeiten, die im Bild sehr bedeutend sein können, deutlich sichtbar.

Histogramms ist es, bestimmte Informationen über ein Bild in kompakter Weise sichtbar zu machen. Gibt es also irgendeine Möglichkeit, das Originalbild aus dem Histogramm allein zu rekonstruieren, d. h., kann man ein Histogramm irgendwie „invertieren“? Natürlich (im Allgemeinen) nicht, schon allein deshalb, weil viele unterschiedliche Bilder – jede unterschiedliche Anordnung einer bestimmten Menge von Pixelwerten – genau dasselbe Histogramm aufweisen (Abb. 4.4).

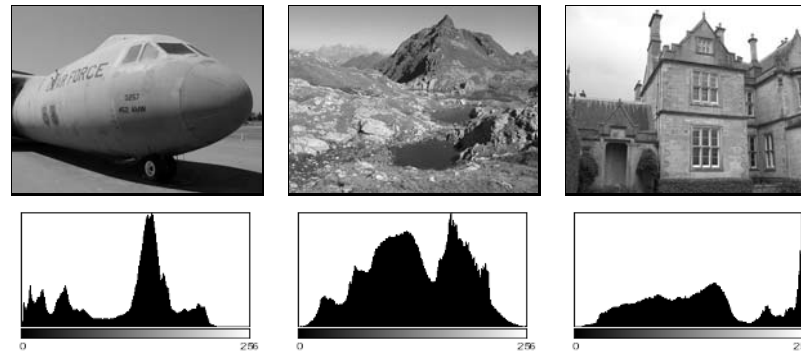
4.2 Was ist aus Histogrammen abzulesen?

Das Histogramm zeigt wichtige Eigenschaften eines Bilds, wie z. B. den Kontrast und die Dynamik, Probleme bei der Bildaufnahme und eventuelle Folgen von anschließenden Verarbeitungsschritten. Das Hauptaugenmerk gilt dabei der Größe des effektiv genutzten Intensitätsbereichs (Abb. 4.5) und der Gleichmäßigkeit der Häufigkeitsverteilung.

4.2.1 Eigenschaften der Bildaufnahme

Belichtung

Belichtungsfehler sind im Histogramm daran zu erkennen, dass größere Intensitätsbereiche an einem Ende der Intensitätsskala ungenutzt sind, während am gegenüberliegenden Ende eine Häufung von Pixelwerten auftritt (Abb. 4.6).



(a) (b) (c)

Abbildung 4.6
Belichtungsfehler sind am Histogramm leicht ablesbar. Unterbelichtete Aufnahme (a), korrekte Belichtung (b), überbelichtete Aufnahme (c).

Kontrast

Als Kontrast bezeichnet man den Bereich von Intensitätsstufen, die in einem konkreten Bild effektiv genutzt werden, also die Differenz zwischen dem maximalen und minimalen Pixelwert. Ein Bild mit vollem Kontrast nützt den gesamten Bereich von Intensitätswerten von $a = a_{\min} \ldots a_{\max} = 0 \ldots K-1$ (schwarz bis weiß). Der Bildkontrast ist daher aus dem Histogramm leicht abzulesen. Abb. 4.7 zeigt ein Beispiel mit unterschiedlichen Kontrasteinstellungen und die Auswirkungen auf das Histogramm.

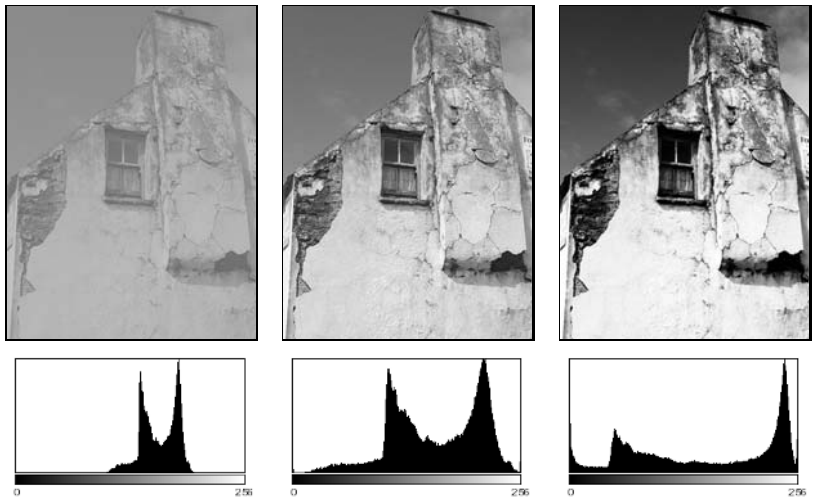
Dynamik

Unter Dynamik versteht man die Anzahl *verschiedener* Pixelwerte in einem Bild. Im Idealfall entspricht die Dynamik der insgesamt verfügbaren Anzahl von Pixelwerten K – in diesem Fall wird der Wertebereich voll ausgeschöpft. Bei einem Bild mit eingeschränktem Kontrastumfang $a = a_{\text{low}} \ldots a_{\text{high}}$, mit

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

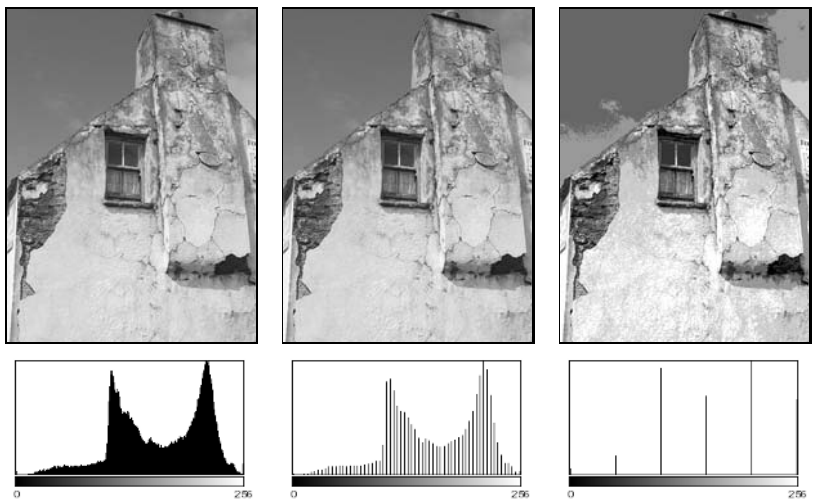
wird die maximal mögliche Dynamik dann erreicht, wenn alle dazwischen liegenden Intensitätswerte ebenfalls im Bild vorkommen (Abb. 4.8).

Während der Kontrast eines Bilds immer erhöht werden kann, solange der maximale Wertebereich nicht ausgeschöpft ist, kann die Dynamik eines Bilds nicht erhöht werden (außer durch Interpolation von Pixelwerten, siehe Abschn. 16.3). Eine hohe Dynamik ist immer ein Vorteil, denn sie verringert die Gefahr von Qualitätsverlusten durch nachfolgende Verarbeitungsschritte. Aus genau diesem Grund arbeiten professionelle Kameras und Scanner mit Tiefen von mehr als 8 Bits, meist 12–14 Bits pro Kanal (Grauwert oder Farbe), obwohl die meisten Ausgabegeräte wie Monitore und Drucker nicht mehr als 256 Abstufungen differenzieren können.



(a) (b) (c)

Abbildung 4.7
Unterschiedlicher Kontrast und Auswirkungen im Histogramm: niedriger Kontrast (a), normaler Kontrast (b), hoher Kontrast (c).



(a) (b) (c)

Abbildung 4.8
Unterschiedliche Dynamik und Auswirkungen im Histogramm. Hohe Dynamik (a), niedrige Dynamik mit 64 Intensitätswerten (b), extrem niedrige Dynamik mit nur 6 Intensitätswerten (c).

4.2.2 Bildfehler

Histogramme können verschiedene Arten von Bildfehlern anzeigen, die entweder auf die Bildaufnahme oder nachfolgende Bearbeitungsschritte zurückzuführen sind. Da ein Histogramm aber immer von der abgebildeten Szene abhängt, gibt es grundsätzlich kein „ideales“ Histogramm. Ein Histogramm kann für eine bestimmte Szene perfekt sein, aber unakzeptabel für eine andere. So wird man von astronomischen Aufnahmen grundsätzlich andere Histogramme erwarten als von guten Landschaftsaufnahmen oder Portraitfotos. Dennoch gibt es einige universelle Regeln. Zum Beispiel kann man bei Aufnahmen von natürlichen Szenen, etwa mit

einer Digitalkamera, mit einer weitgehend glatten Verteilung der Intensitätswerte ohne einzelne, isolierte Spitzen rechnen.

Sättigung

Idealerweise sollte der Kontrastbereich eines Sensorsystems (z. B. einer Kamera) größer sein als der Umfang der Lichtintensität, die von einer Szene empfangen wird. In diesem Fall würde das Histogramm nach der Aufnahme an beiden Seiten glatt auslaufen, da sowohl sehr helle als auch sehr dunkle Intensitätswerte zunehmend seltener werden und alle vorkommenden Lichtintensitäten entsprechenden Bildwerten zugeordnet werden. In der Realität ist dies oft nicht der Fall, und Helligkeitswerte außerhalb des vom Sensor abgedeckten Kontrastbereichs, wie Glanzlichter oder besonders dunkle Bildpartien, werden abgeschnitten. Die Folge ist eine Sättigung des Histogramms an einem Ende oder an beiden Enden des Wertebereichs, da die außerhalb liegenden Intensitäten auf den Minimal- bzw. den Maximalwert abgebildet werden, was im Histogramm durch markante Spitzen an den Enden des Intensitätsbereichs deutlich wird. Typisch ist dieser Effekt bei Über- oder Unterbelichtung während der Bildaufnahme und generell dann nicht vermeidbar, wenn der Kontrastumfang der Szene den des Sensors übersteigt (Abb. 4.9 (a)).

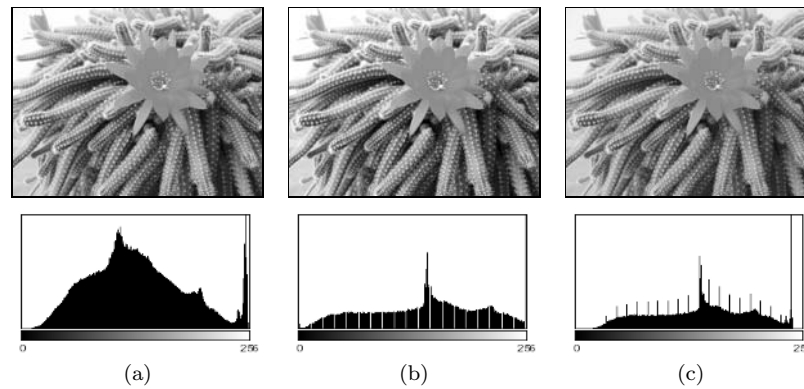


Abbildung 4.9
Auswirkungen von Bildfehlern im Histogramm: Sättigungseffekt im Bereich der hohen Intensitäten (a), Histogrammlöcher verursacht durch eine geringfügige Kontrasterhöhung (b) und Histogrammspitzen aufgrund einer Kontrastreduktion (c).

Spitzen und Löcher

Wie bereits erwähnt ist die Verteilung der Helligkeitswerte in einer unbearbeiteten Aufnahme in der Regel glatt, d. h., es ist wenig wahrscheinlich, dass im Histogramm (abgesehen von Sättigungseffekten an den Rändern) isolierte Spitzen auftreten oder einzelne Löcher, die lokale Häufigkeit eines Intensitätswerts sich also sehr stark von seinen Nachbarn unterscheidet. Beide Effekte sind jedoch häufig als Folge von Bildmanipulationen zu beobachten, etwa nach Kontraständerungen. Insbesondere führt eine

Erhöhung des Kontrasts (s. Kap. 5) dazu, dass Histogrammlinien auseinander gezogen werden und – aufgrund des diskreten Wertebereichs – Fehlstellen (Löcher) im Histogramm entstehen (Abb. 4.9 (b)). Umgekehrt können durch eine Kontrastverminderung aufgrund des diskreten Wertebereichs bisher unterschiedliche Pixelwerte zusammenfallen und die zugehörigen Histogrammeinträge erhöhen, was wiederum zu deutlich sichtbaren Spitzen im Histogramm führt (Abb. 4.9 (c)).[2]

Auswirkungen von Bildkompression

Bildveränderungen aufgrund von Bildkompression hinterlassen bisweilen markante Spuren im Histogramm. Deutlich wird das z. B. bei der GIF-Kompression, bei der der Wertebereich des Bilds auf nur wenige Intensitäten oder Farben reduziert wird. Der Effekt ist im Histogramm als Linienstruktur deutlich sichtbar und kann durch nachfolgende Verarbeitung im Allgemeinen nicht mehr eliminiert werden (Abb. 4.10). Es

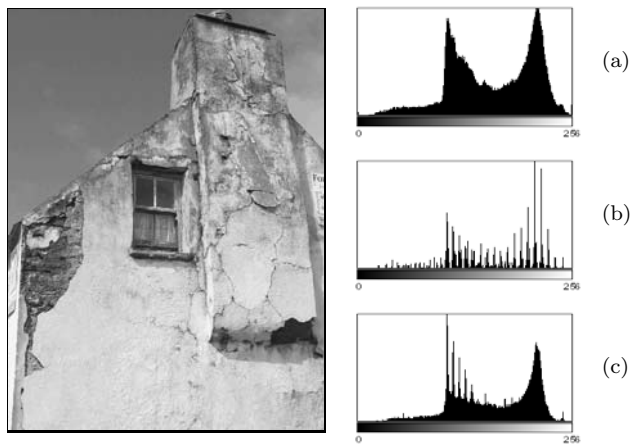


Abbildung 4.10
Auswirkungen einer Farbquantisierung durch GIF-Konvertierung. Das Originalbild wurde auf ein GIF-Bild mit 256 Farben konvertiert (links). Original-Histogramm (a) und Histogramm nach der GIF-Konvertierung (b). Bei der nachfolgenden Skalierung des RGB-Farbbilds auf 50% seiner Größe entstehen durch Interpolation wieder Zwischenwerte, doch bleiben die Folgen der ursprünglichen Konvertierung deutlich sichtbar (c).

ist also über das Histogramm in der Regel leicht festzustellen, ob ein Bild jemals einer Farbquantisierung (wie etwa bei Umwandlung in eine GIF-Datei) unterzogen wurde, auch wenn das Bild (z. B. als TIFF- oder JPEG-Datei) vorgibt, ein echtes Vollfarbenbild zu sein.

Einen anderen Fall zeigt Abb. 4.11, wo eine einfache, „flache“ Grafik mit nur zwei Grauwerten (128, 255) einer JPEG-Kompression unterzogen wird, die für diesen Zweck eigentlich nicht geeignet ist. Das resultierende Bild ist durch eine große Anzahl neuer, bisher nicht enthaltener

[2] Leider erzeugen auch manche Aufnahmegeräte (vor allem einfache Scanner) derartige Fehler durch interne Kontrastanpassung („Optimierung“) der Bildqualität.

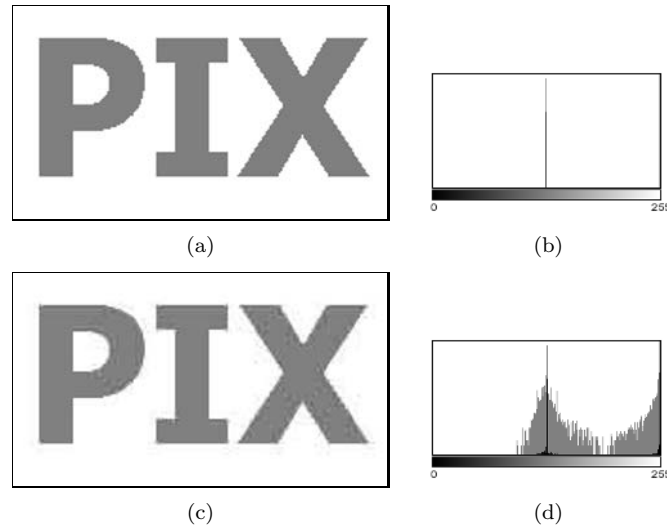


Abbildung 4.11
Auswirkungen der JPEG-Kompression. Das Originalbild (a) enthält nur zwei verschiedene Grauwerte, wie im zugehörigen Histogramm (b) leicht zu erkennen ist. Durch die JPEG-Kompression entstehen zahlreiche zusätzliche Grauwerte, die im resultierenden Bild (c) genauso wie im Histogramm (d) sichtbar sind. In beiden Histogrammen sind die Häufigkeiten linear (schwarze Balken) bzw. logarithmisch (graue Balken) dargestellt.

Grauwerte „verschmutzt“, wie man vor allem im Histogramm deutlich feststellen kann.[3]

4.3 Berechnung von Histogrammen

Die Berechnung eines Histogramms für ein 8-Bit-Grauwertbild (mit Intensitätswerten zwischen 0 und 255) ist eine einfache Angelegenheit. Alles, was wir dazu brauchen, sind 256 einzelne Zähler, einer für jeden möglichen Intensitätswert. Zunächst setzen wir alle diese Zähler auf Null. Dann durchlaufen wir alle Bildelemente $I(u, v)$, ermitteln den zugehörigen Pixelwert p und erhöhen den entsprechenden Zähler um eins. Am Ende sollte jeder Zähler die Anzahl der gefundenen Pixel des zugehörigen Intensitätswerts beinhalten.

Wir benötigen also für K mögliche Intensitätswerte genauso viele verschiedene Zählervariablen, z. B. 256 für ein 8-Bit-Grauwertbild. Natürlich realisieren wir diese Zähler nicht als einzelne Variablen, sondern als Array mit K ganzen Zahlen (`int[]` in Java). Angenehmerweise sind in diesem Fall die Intensitätswerte alle positiv und beginnen bei 0, sodass wir sie in Java direkt als Indizes $i \in [0, N-1]$ für das Histogramm-Array verwenden können. Prog. 4.1 zeigt den fertigen Java-Quellcode für die Berechnung des Histogramms, eingebaut in die `run()`-Methode eines entsprechenden ImageJ-Plugins.

Das Histogramm-Array `H` vom Typ `int[]` wird in Prog. 4.1 gleich zu Beginn (Zeile 8) angelegt und automatisch auf Null initialisiert, an-

[3] Der undifferenzierte Einsatz der JPEG-Kompression für solche Arten von Bildern ist ein häufiger Fehler. JPEG ist für natürliche Bilder mit weichen Übergängen konzipiert und verursacht bei Grafiken u. Ä. starke Artefakte (siehe beispielsweise Abb. 2.9 auf S. 20).

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

Programm 4.1
ImageJ-Plugin zur Berechnung des Histogramms für 8-Bit-Grauwertbilder. Die `setup()`-Methode liefert `DOES_8G + NO_CHANGES` und zeigt damit an, dass das Plugin auf 8-Bit-Grauwertbilder angewandt werden kann und diese nicht verändert werden (Zeile 4). Man beachte, dass Java im neu angelegten Histogramm-Array (Zeile 8) automatisch alle Elemente auf Null initialisiert.

```
1     public void run(ImageProcessor ip) {
2         int[] H = ip.getHistogram();
3         ... // histogram H[] can now be used
4     }
```

Programm 4.2
Verwendung der vordefinierten ImageJ-Methode `getHistogram()` in der `run()`-Methode eines Plugin.

schließend werden alle Bildelemente durchlaufen. Dabei ist es grundsätzlich nicht relevant, in welcher Reihenfolge Zeilen und Spalten durchlaufen werden, solange alle Bildpunkte genau einmal besucht werden. In diesem Beispiel haben wir (im Unterschied zu Prog. 3.1) die Standard-Reihenfolge gewählt, in der die äußere `for`-Schleife über die vertikale Koordinate v und die innere Schleife über die horizontale Koordinate u iteriert.[4] Am Ende ist das Histogramm berechnet und steht für weitere Schritte (z. B. zur Anzeige) zur Verfügung.

Die Histogrammberechnung gibt es in ImageJ allerdings auch bereits fertig, und zwar in Form der Methode `getHistogram()` für Objekte der Klasse `ImageProcessor`. Damit lässt sich die `run()`-Methode in Prog. 4.1 natürlich deutlich einfacher gestalten (Prog. 4.2):

[4] In dieser Form werden die Bildelemente in genau der Reihenfolge gelesen, in der sie auch hintereinander im Hauptspeicher liegen, was zumindest bei großen Bildern wegen des effizienteren Speicherzugriffs einen gewissen Geschwindigkeitsvorteil verspricht (siehe auch Anhang 2.2).

4.4 Histogramme für Bilder mit mehr als 8 Bit

Meistens werden Histogramme berechnet, um die zugehörige Verteilung auf dem Bildschirm zu visualisieren. Das ist zwar bei Histogrammen für Bilder mit $2^8 = 256$ Einträgen problemlos, für Bilder mit größeren Wertebereichen, wie 16 und 32 Bit oder Gleitkommawerten (s. Tabelle 2.1), ist die Darstellung in dieser Form aber nicht ohne weiteres möglich.

4.4.1 Binning

Die Lösung dafür besteht darin, jeweils mehrere Intensitätswerte bzw. ein *Intervall* von Intensitätswerten zu einem Eintrag zusammenzufassen, anstatt für jeden möglichen Wert eine eigene Zählerzelle vorzusehen. Man kann sich diese Zählerzelle als Eimer (engl. *bin*) vorstellen, in dem Pixelwerte gesammelt werden, daher wird die Methode häufig auch „Binning“ genannt.

In einem solchen Histogramm der Größe B enthält jede Zelle $\mathsf{h}(j)$ die Anzahl aller Bildelemente mit Werten aus einem zugeordneten Intensitätsintervall $a_j \le a < a_{j+1}$, d. h. (analog zu Gl. 4.1)

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

Üblicherweise wird dabei der verfügbare Wertebereich in B gleich große Bins der Intervalllänge $k_B = K/B$ geteilt, d. h. der Startwert des Intervalls j ist

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

4.4.2 Beispiel

Um für ein 14-Bit-Bild ein Histogramm mit $B = 256$ Einträgen zu erhalten, teilen wir den verfügbaren Wertebereich von $j = 0 \ldots 2^{14}-1$ in 256 gleiche Intervalle der Länge $k_B = 2^{14}/256 = 64$, sodass $a_0 = 0$, $a_1 = 64$, $a_2 = 128$, ... $a_{255} = 16320$ und $a_{256} = a_B = 2^{14} = 16384 = K$. Damit ergibt sich folgende Zuordnung der Intervalle zu den Histogrammzellen $\mathsf{h}(0) \ldots \mathsf{h}(255)$:

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

4.4.3 Implementierung

Falls, wie im diesem Beispiel, der Wertebereich $0 \ldots K-1$ in gleiche Intervalle der Länge $k_B = K/B$ aufgeteilt ist, benötigt man natürlich keine Tabelle der Werte a_j, um für einen gegebenen Pixelwert $a = I(u,v)$ das zugehörige Histogrammelement j zu bestimmen. Dazu genügt es, den Pixelwert $I(u,v)$ durch die Intervalllänge k_B zu dividieren, d. h.

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

Als Index für die zugehörige Histogrammzelle $\mathsf{h}(j)$ benötigen wir allerdings einen ganzzahligen Wert und verwenden dazu

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

wobei $\lfloor \cdot \rfloor$ die *floor*-Funktion[5] bezeichnet. Eine Java-Methode zur Histogrammberechnung mit „linearem Binning“ ist in Prog. 4.3 gezeigt. Man beachte, dass die gesamte Berechnung des Ausdrucks in Gl. 4.4 ganzzahlig durchgeführt wird, ohne den Einsatz von Gleitkomma-Operationen. Auch ist keine explizite Anwendung der *floor*-Funktion notwendig, weil der Ausdruck

```
a * B / K
```

in Zeile 11 ohnehin einen ganzzahligen Wert liefert.[6] Die Binning-Methode ist in gleicher Weise natürlich auch für Bilder mit Gleitkommawerten anwendbar.

4.5 Histogramme von Farbbildern

Mit Histogrammen von Farbbildern sind meistens Histogramme der zugehörigen Bildintensität (Luminanz) gemeint oder die Histogramme der einzelnen Farbkanäle. Beide Varianten werden von praktisch jeder gängigen Bildbearbeitungssoftware unterstützt und dienen genauso wie bei Grauwertbildern zur objektiven Beurteilung der Bildqualität, insbesondere nach der Aufnahme.

4.5.1 Luminanzhistogramm

Das Luminanzhistogramm $\mathsf{h}_{\mathrm{Lum}}$ eines Farbbilds ist nichts anderes als das Histogramm des entsprechenden Grauwertbilds, für das natürlich alle bereits oben angeführten Aspekte ohne Einschränkung gelten. Das einem Farbbild entsprechende Grauwertbild erhält man durch die Berechnung der zugehörigen Luminanz aus den einzelnen Farbkomponenten. Dazu werden allerdings die Werte der Farbkomponenten nicht einfach addiert, sondern üblicherweise in Form einer gewichteten Summe verknüpft (s. auch Kap. 12).

[5] $\lfloor x \rfloor$ rundet x auf die nächstliegende ganze Zahl ab (siehe Anhang 1.1).
[6] Siehe auch die Anmerkungen zur Integer-Division in Java in Anhang B.1.1.

Programm 4.3
Histogrammberechnung durch „Binning“ (Java-Methode). Beispiel für ein 8-Bit-Grauwertbild mit $K = 256$ Intensitätsstufen und ein Histogramm der Größe $B = 32$. Die Methode `binnedHistogram()` liefert das Histogramm des übergebenen Bildobjekts `ip` als `int`-Array der Größe B.

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

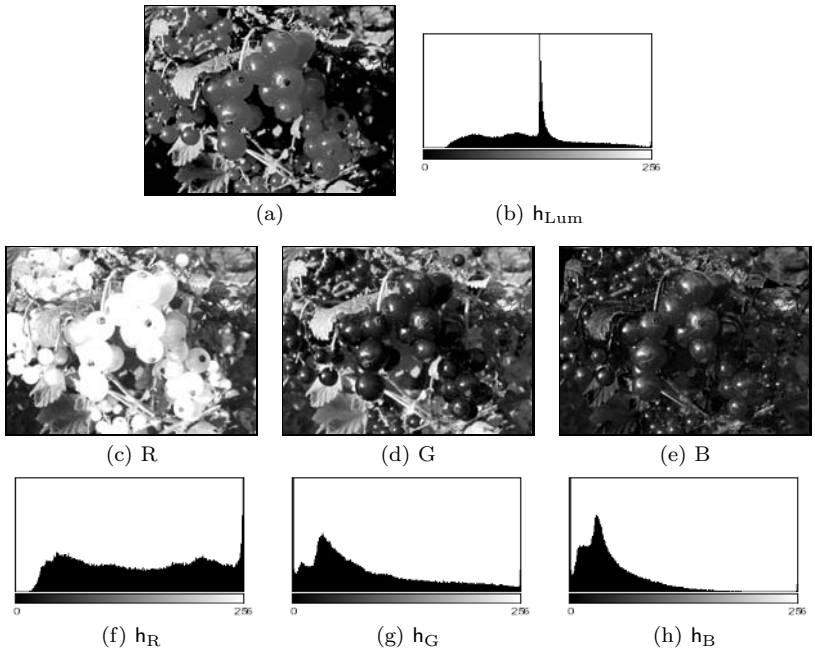
4.5.2 Histogramme der Farbkomponenten

Obwohl das Luminanzhistogramm alle Farbkomponenten berücksichtigt, können darin einzelne Bildfehler dennoch unentdeckt bleiben. Zum Beispiel ist es möglich, dass das Luminanzhistogramm durchaus sauber aussieht, obwohl einer der Farbkanäle bereits gesättigt ist. In RGB-Bildern trägt insbesondere der Blau-Kanal nur wenig zur Gesamthelligkeit bei und ist damit besonders anfällig für dieses Problem.

Komponentenhistogramme geben zusätzliche Aufschlüsse über die Intensitätsverteilung innerhalb der einzelnen Farbkanäle. Jede Farbkomponente wird als unabhängiges Intensitätsbild betrachtet und die zugehörigen Einzelhistogramme werden getrennt berechnet und angezeigt. Abb. 4.12 zeigt das Luminanzhistogramm $\mathsf{h}_{\mathrm{Lum}}$ und die drei Komponentenhistogramme h_{R}, h_{G} und h_{B} für ein typisches RGB-Farbbild. Man beachte, dass in diesem Beispiel die Sättigung aller drei Farbkanäle (rot im oberen Intensitätsbereich, grün und blau im unteren Bereich) nur in den Komponentenhistogrammen, nicht aber im Luminanzhistogramm deutlich wird. Auffallend (aber nicht untypisch) ist in diesem Fall auch das gegenüber den drei Komponentenhistogrammen völlig unterschiedliche Aussehen des Luminanzhistogramms $\mathsf{h}_{\mathrm{Lum}}$ (Abb. 4.12 (b)).

4.5.3 Kombinierte Farbhistogramme

Luminanzhistogramme und Komponentenhistogramme liefern nützliche Informationen über Belichtung, Kontrast, Dynamik und Sättigungseffekte bezogen auf die einzelnen Farbkomponenten. Sie geben jedoch keine Informationen über die Verteilung der tatsächlichen *Farben* in einem Bild, denn das räumliche Zusammentreffen der Farbkomponenten innerhalb eines Bildelements wird dabei nicht berücksichtigt. Wenn z. B. h_{R},



(a) (b) $\mathsf{h}_{\mathrm{Lum}}$

(c) R (d) G (e) B

(f) h_{R} (g) h_{G} (h) h_{B}

Abbildung 4.12
Histogramme für ein RGB-Farbbild: Originalbild (a), Luminanzhistogramm $\mathsf{h}_{\mathrm{Lum}}$ (b), RGB-Farbkomponenten als Intensitätsbilder (c–e) und die zugehörigen Komponentenhistogramme h_{R}, h_{G}, h_{B} (f–h). Die Tatsache, dass alle drei Farbkanäle in Sättigung sind, wird nur in den einzelnen Komponentenhistogrammen deutlich. Die dadurch verursachte Verteilungsspitze befindet sich in der Mitte des Luminanzhistogramms (b).

das Komponentenhistogramm für den Rot-Kanal, einen Eintrag

$$\mathsf{h}_{\mathrm{R}}(200) = 24$$

hat, dann wissen wir nur, dass das Bild 24 Pixel mit einer Rot-Intensität von 200 aufweist, aber mit beliebigen ($*$) Grün- und Blauwerten, also

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

Nehmen wir weiter an, die drei Komponentenhistogramme hätten die Einträge

$$\mathsf{h}_{\mathrm{R}}(50) = 100, \quad \mathsf{h}_{\mathrm{G}}(50) = 100, \quad \mathsf{h}_{\mathrm{B}}(50) = 100.$$

Können wir daraus schließen, dass in diesem Bild ein Pixel mit der Kombination

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

als Farbwert 100 mal oder überhaupt vorkommt? Im Allgemeinen natürlich nicht, denn es ist offen, ob alle drei Komponenten gemeinsam mit dem Wert von jeweils 50 in irgend einem Pixel zusammen auftreten. Man kann mit Bestimmtheit nur sagen, das der Farbwert $(50, 50, 50)$ in diesem Bild höchstens 100 mal vorkommen kann.

Während also konventionelle Histogramme von Farbbildern zwar einiges an Information liefern können, geben sie nicht wirklich Auskunft über die Zusammensetzung der tatsächlichen Farben in einem Bild. So können verschiedene Farbbilder sehr ähnliche Komponentenhistogramme aufweisen, obwohl keinerlei farbliche Ähnlichkeit zwischen den Bildern

besteht. Ein interessantes Thema sind daher *kombinierte* Histogramme, die das Zusammentreffen von mehreren Farbkomponenten statistisch erfassen und damit u. a. auch eine grobe Ähnlichkeit zwischen Bildern ausdrücken können. Auf diesen Aspekt kommen wir im Zusammenhang mit Farbbildern in Kap. 12 nochmals zurück.

4.6 Das kumulative Histogramm

Das kumulative Histogramm ist eine Variante des gewöhnlichen Histogramms, das für die Berechnung bei Bildoperationen mit Histogrammen nützlich ist, z. B. im Zusammenhang mit dem Histogrammausgleich (Abschn. 5.4). Das kumulative Histogramm $\mathsf{H}(i)$ ist definiert als

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

Der Wert von $\mathsf{H}(i)$ ist also die Summe aller darunter liegenden Werte des ursprünglichen Histogramms $\mathsf{h}(j)$, mit $j = 0 \ldots i$. Oder, in rekursiver Form definiert (umgesetzt in Prog. 5.2 auf S. 64):

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

Der Funktionsverlauf eines kumulativen Histogramms ist daher immer monoton steigend, mit dem Maximalwert

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

also der Gesamtzahl der Pixel im Bild mit der Breite M und der Höhe N. Abb. 4.13 zeigt ein konkretes Beispiel für ein kumulatives Histogramm.

4.7 Aufgaben

Aufg. 4.1. In Prog. 4.3 sind `B` und `K` konstant. Überlegen Sie, warum es dennoch nicht sinnvoll ist, den Wert von `B/K` außerhalb der Schleifen im Voraus zu berechnen.

Aufg. 4.2. Erstellen Sie ein ImageJ-Plugin, das von einem 8-Bit-Grauwertbild das kumulative Histogramm berechnet und in Form eines neuen Bilds darstellt, ähnlich wie die in ImageJ eingebaute Histogramm-Funktion (unter Analyze→Histogram).

Aufg. 4.3. Entwickeln Sie ein Verfahren für nichtlineares Binning mithilfe einer Tabelle der Intervallgrenzen a_i (Gl. 4.2).

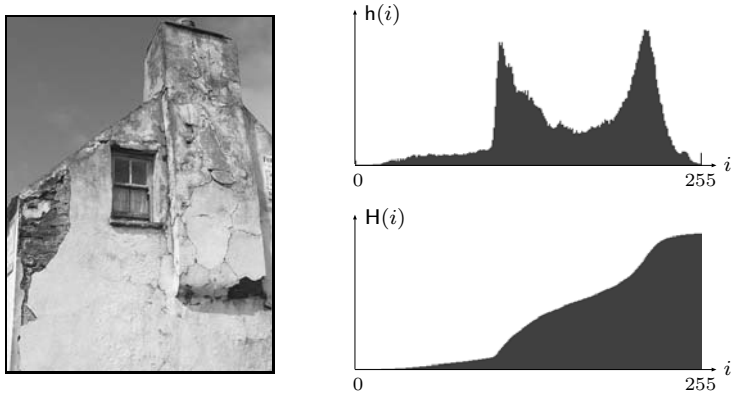


Abbildung 4.13
Gewöhnliches Histogramm $\mathsf{h}(i)$ und das zugehörige kumulative Histogramm $\mathsf{H}(i)$.

Aufg. 4.4. Erstellen Sie ein Plugin, das ein zufälliges Bild mit gleichverteilten Pixelwerten im Bereich $[0, 255]$ erzeugt. Verwenden Sie dazu die Java-Methode `Math.random()` und überprüfen Sie mittels des Histogramms, wie weit die Pixelwerte tatsächlich gleichverteilt sind.

5

Punktoperationen

Als Punktoperationen bezeichnet man Operationen auf Bilder, die nur die Werte der einzelnen Bildelemente betreffen und keine Änderungen der Größe, Geometrie oder der lokalen Bildstruktur nach sich ziehen. Jeder neue Pixelwert $a' = I'(u, v)$ ist ausschließlich abhängig vom ursprünglichen Pixelwert $a = I(u, v)$ an der selben Position und damit *unabhängig* von den Werten anderer, insbesondere benachbarter Pixel. Der neue Pixelwert wird durch eine Funktion $f(a)$ bestimmt, d. h.

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

Wenn – wie in diesem Fall – die Funktion $f()$ auch unabhängig von den Bildkoordinaten ist, also für jede Bildposition gleich ist, dann bezeichnet man die Operation als *homogen*. Typische Beispiele für homogene Punktoperationen sind

- Änderungen von Kontrast und Helligkeit,
- Anwendung von beliebigen Helligkeitskurven,
- das Invertieren von Bildern,
- das Quantisieren der Bildhelligkeit in grobe Stufen (Poster-Effekt),
- eine Schwellwertbildung,
- Gammakorrektur,
- Farbtransformationen,
- usw.

Wir betrachten nachfolgend einige dieser Beispiele im Detail. Eine *nicht*-homogene Punktoperation würde demgegenüber zusätzlich die Bildkoordinaten (u, v) berücksichtigen, d. h.

$$\begin{aligned} I'(u,v) &\leftarrow g\,(I(u,v), u, v)\;, \quad \text{bzw.} \\ a' &\leftarrow g(a, u, v)\,. \end{aligned} \tag{5.2}$$

Programm 5.1
ImageJ-Plugin-Code für eine Punktoperation zur Kontrasterhöhung um 50%. Man beachte, dass in Zeile 7 die Multiplikation eines ganzzahligen Pixelwerts (vom Typ `int`) mit der Konstante 1.5 (implizit vom Typ `double`) ein Ergebnis vom Typ `double` erzeugt. Daher ist ein expliziter *Typecast* `(int)` für die Zuweisung auf die Variable `a` notwendig.

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

Eine häufige Anwendung nichthomogener Operationen ist z. B. die selektive Kontrast- oder Helligkeitsanpassung, etwa um eine ungleichmäßige Beleuchtung bei der Bildaufnahme auszugleichen.

5.1 Änderung der Bildintensität

5.1.1 Kontrast und Helligkeit

Dazu gleich ein Beispiel: Die Erhöhung des Bildkontrasts um 50% (d. h. um den Faktor 1.5) oder das Anheben der Helligkeit um 10 Stufen entspricht einer homogenen Punktoperation mit der Funktion

$$f_{\mathrm{c}}(a) = a \cdot 1.5 \qquad \text{bzw.} \qquad f_{\mathrm{b}}(a) = a + 10\,. \tag{5.3}$$

Die Umsetzung der Kontrasterhöhung als ImageJ-Plugin ist in Prog. 5.1 gezeigt, wobei dieser Code natürlich leicht für beliebige Punktoperationen angepasst werden kann.

5.1.2 Beschränkung der Ergebniswerte (*clamping*)

Bei der Umsetzung von Bildoperationen muss berücksichtigt werden, dass der vorgegebene Wertebereich für Bildpixel beschränkt ist (bei 8-Bit-Grauwertbildern auf $[0 \ldots 255]$) und die berechneten Ergebnisse möglicherweise außerhalb dieses Wertebereichs liegen. Um das zu vermeiden, ist in Prog. 5.1 (Zeile 9) die Anweisung

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

vorgesehen, die alle höheren Ergebniswerte auf den Maximalwert 255 begrenzt. Dieser Vorgang wird häufig als „Clamping“ bezeichnet. Genauso sollte man i. Allg. auch die Ergebnisse nach „unten“ auf den Minimalwert 0 begrenzen und damit verhindern, dass Pixelwerte negativ werden, etwa durch die Anweisung

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

In Prog. 5.1 war dieser zweite Schritt allerdings nicht notwendig, da ohnehin nur positive Ergebniswerte entstehen können.

5.1.3 Invertieren von Bildern

Bilder zu invertieren ist eine einfache Punktoperation, die einerseits die Ordnung der Pixelwerte (durch Multiplikation mit -1) umkehrt und andererseits durch Addition eines konstanten Intensitätswerts dafür sorgt, dass das Ergebnis innerhalb des erlaubten Wertebereichs bleibt. Für ein Bildelement $a = I(u, v)$ mit dem Wertebereich $[0, a_{\max}]$ ist die zugehörige Operation daher

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

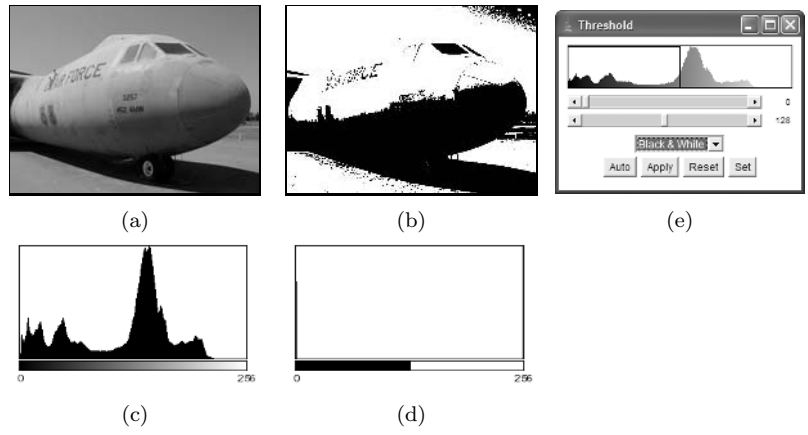
Die Inversion eines 8-Bit-Grauwertbilds mit $a_{\max} = 255$ war Aufgabe unseres ersten Plugin-Beispiels in Abschn. 3.2.4 (Prog. 3.1). Ein „clamping“ ist in diesem Fall übrigens nicht notwendig, da sichergestellt ist, dass der erlaubte Wertebereich nicht verlassen wird. In ImageJ ist diese Operation unter Edit→Invert zu finden. Das Histogramm wird beim Invertieren eines Bilds gespiegelt, wie Abb. 5.5 (c) zeigt.

5.1.4 Schwellwertoperation (*tresholding*)

Eine Schwellwertoperation ist eine spezielle Form der Quantisierung, bei der die Bildwerte in zwei Klassen getrennt werden, abhängig von einem vorgegebenen Schwellwert („threshold value“) a_{th}. Alle Pixel werden in dieser Punktoperation einem von zwei fixen Intensitätswerten a_0 oder a_1 zugeordnet, d. h.

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

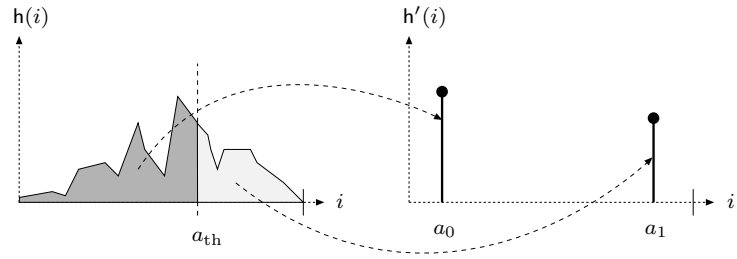
wobei $0 < a_{\text{th}} \leq a_{\max}$. Eine häufige Anwendung ist die Binarisierung von Grauwertbildern mit $a_0 = 0$ und $a_1 = 1$. In ImageJ gibt es zwar einen eigenen Datentyp für Binärbilder (`BinaryProcessor`), diese sind aber intern – wie gewöhnliche Grauwertbilder – als 8-Bit-Bilder mit den Pixelwerten 0 und 255 implementiert. Für die Binarisierung in ein derartiges Bild mit einer Schwellwertoperation wäre daher $a_0 = 0$ und $a_1 = 255$ zu setzen. Ein entsprechendes Beispiel ist in Abb. 5.1 gezeigt, wie auch das in ImageJ unter Image→Adjust→Threshold verfügbare Tool. Die Auswirkung einer Schwellwertoperation auf das Histogramm ist klarerweise, dass die gesamte Verteilung in zwei Einträge an den Stellen a_0 und a_1 aufgeteilt wird, wie in Abb. 5.2 dargestellt.



(a) (b) (e)

(c) (d)

Abbildung 5.1
Schwellwertoperation. Originalbild (a) und zugehöriges Histogramm (c), Ergebnis nach der Schwellwertoperation mit $a_{\text{th}} = 128$, $a_0 = 0$ und $a_1 = 255$ (b) und resultierendes Histogramm (d). Interaktives Threshold-Menü in ImageJ (e).



(a) (b)

Abbildung 5.2
Auswirkung der Schwellwertoperation im Histogramm. Der Schwellwert ist a_{th}. Die ursprüngliche Verteilung (a) wird im resultierenden Histogramm (b) in zwei isolierten Einträgen bei a_0 und a_1 konzentriert.

5.2 Punktoperationen und Histogramme

Wir haben bereits in einigen Fällen gesehen, dass die Auswirkungen von Punktoperationen auf das Histogramm relativ einfach vorherzusehen sind. Eine Erhöhung der Bildhelligkeit verschiebt beispielsweise das gesamte Histogramm nach rechts, durch eine Kontrasterhöhung wird das Histogramm breiter, das Invertieren des Bilds bewirkt eine Spiegelung des Histogramms usw. Obwohl diese Vorgänge so einfach (vielleicht sogar trivial?) erscheinen, mag es nützlich sein, sich den Zusammenhang zwischen Punktoperationen und den dadurch verursachten Veränderungen im Histogramm nochmals zu verdeutlichen.

Wie die Grafik in Abb. 5.3 zeigt, gehört zu jedem Eintrag (Balken) im Histogramm an der Stelle i die Menge all jener Bildelemente, die genau den Pixelwert i aufweisen.[1]

Wird infolge einer Operation eine bestimmte Histogrammlinie verschoben, dann verändern sich natürlich auch alle Elemente der zugehörigen Pixelmenge, bzw. umgekehrt. Was passiert daher, wenn aufgrund einer Operation zwei bisher getrennte Histogrammlinien zusammenfallen?

[1] Das gilt in der Form natürlich nur für Histogramme, in denen jeder mögliche Intensitätswert einen Eintrag hat, d. h. nicht für Histogramme, die durch *Binning* (Abschn. 4.4.1) berechnet wurden.

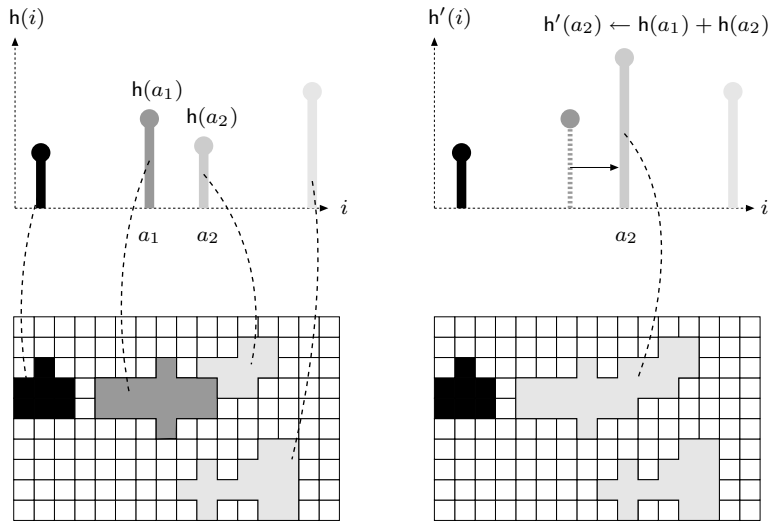


Abbildung 5.3
Histogrammeinträge entsprechen *Mengen* von Bildelementen. Wenn eine Histogrammlinie sich aufgrund einer Punktoperation verschiebt, dann werden alle Pixel der entsprechenden Menge in gleicher Weise modifiziert (a). Sobald dabei zwei Histogrammlinien $\mathsf{h}(a_1)$, $\mathsf{h}(a_2)$ zusammenfallen, vereinigen sich die zugehörigen Pixelmengen und werden ununterscheidbar (b).

– die beiden zugehörigen Pixelmengen *vereinigen* sich und der gemeinsame Eintrag im Histogramm ist die Summe der beiden bisher getrennten Einträge. Die Elemente in der vereinigten Menge sind ab diesem Punkt nicht mehr voneinander unterscheidbar oder trennbar, was uns zeigt, dass mit diesem Vorgang ein (möglicherweise unbeabsichtigter) Verlust von Dynamik und Bildinformation verbunden ist.

5.3 Automatische Kontrastanpassung

Ziel der automatischen Kontrastanpassung ist es, die Pixelwerte eines Bilds so zu verändern, dass der gesamte verfügbare Wertebereich abgedeckt wird. Dazu wird das aktuell dunkelste Pixel auf den niedrigsten, das hellste Pixel auf den höchsten Intensitätswert abgebildet und alle dazwischenliegenden Pixelwerte linear verteilt.

Nehmen wir an, a_{low} und a_{high} ist der aktuell kleinste bzw. größte Pixelwert in einem Bild I, das über einen maximalen Intensitätsbereich $[a_{\min}, a_{\max}]$ verfügt. Um den gesamten Intensitätsbereich abzudecken, wird zunächst der kleinste Pixelwert a_{low} auf den Minimalwert abgebildet und nachfolgend der Bildkontrast um den Faktor $(a_{\max} - a_{\min})/(a_{\text{high}} - a_{\text{low}})$ erhöht (siehe Abb. 5.4). Die einfache Auto-Kontrast-Funktion ist daher definiert als

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

vorausgesetzt natürlich $a_{\text{high}} \neq a_{\text{low}}$, d. h., das Bild weist mindestens zwei unterschiedliche Pixelwerte auf. Für ein 8-Bit-Grauwertbild mit $a_{\min} = 0$ und $a_{\max} = 255$ vereinfacht sich die Funktion in Gl. 5.6 zu

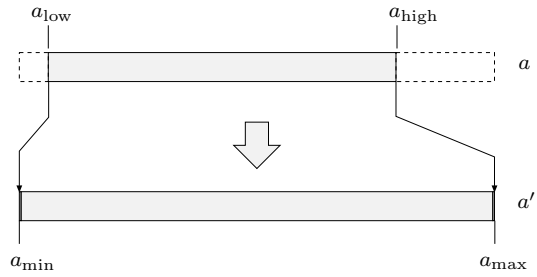


Abbildung 5.4
Auto-Kontrast-Operation nach Gl. 5.6. Pixelwerte a im Intervall $[a_{\text{low}}, a_{\text{high}}]$ werden linear auf Werte a' im Intervall $[a_{\text{min}}, a_{\text{max}}]$ abgebildet.

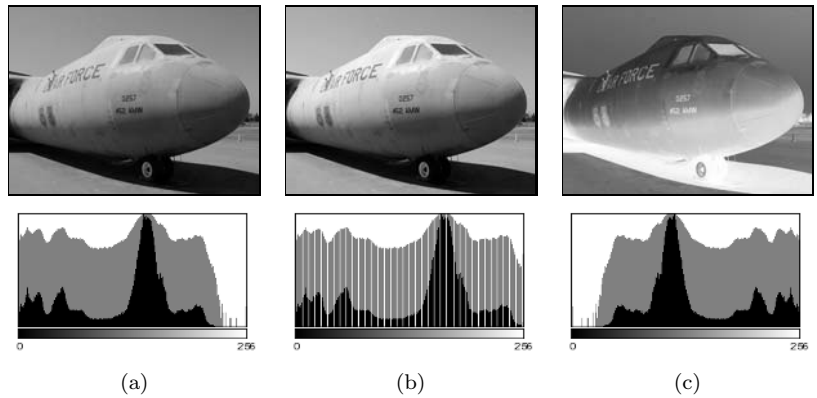


Abbildung 5.5
Auswirkung der Auto-Kontrast-Operation und Inversion auf das Histogramm. Originalbild und zugehöriges Histogramm (a), Ergebnis nach Anwendung der Auto-Kontrast-Operation (b) und der Inversion des Bilds (c). Die Histogramme sind linear (schwarz) und logarithmisch (grau) dargestellt.

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

Der Bereich $[a_{\text{min}}, a_{\text{max}}]$ muss nicht dem maximalen Wertebereich entsprechen, sondern kann grundsätzlich ein beliebiges Intervall sein, den das Ergebnisbild abdecken soll. Natürlich funktioniert die Methode auch dann, wenn der Kontrast auf einen kleineren Bereich zu reduzieren ist. Abb. 5.5 (b) zeigt die Auswirkungen einer Auto-Kontrast-Operation auf das zugehörige Histogramm, in dem die lineare Streckung des ursprünglichen Wertebereichs durch die regelmäßig angeordneten Lücken deutlich wird.

In der Praxis kann die Formulierung in Gl. 5.6 dazu führen, dass durch einzelne Pixel mit extremen Werten die gesamte Intensitätsverteilung stark verändert wird. Das lässt sich weitgehend vermeiden, indem man einen fixen Prozentsatz (s_{low}, s_{high}) der Pixel am oberen bzw. unteren Ende des Wertebereichs in „Sättigung“ gehen lässt, d. h. auf die beiden Extremwerte abbildet. Dazu bestimmen wir zwei Grenzwerte $\hat{a}_{\text{low}}$, $\hat{a}_{\text{high}}$ so, dass im gegebenen Bild I ein bestimmter Anteil von s_{low} aller Pixel kleiner als $\hat{a}_{\text{low}}$ und ein Anteil s_{high} der Pixel größer als $\hat{a}_{\text{high}}$ ist (Abb. 5.6). Die Werte für $\hat{a}_{\text{low}}$, $\hat{a}_{\text{high}}$ sind vom gegebenen Bildinhalt abhängig und können auf einfache Weise aus dem kumulativen Histogramm[2] $\mathsf{H}(i)$ des Ausgangsbilds I berechnet werden:

[2] Siehe Abschn. 4.6.

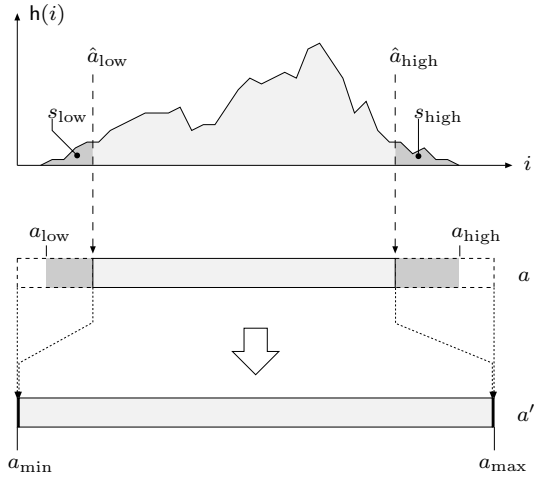


Abbildung 5.6
Modifizierte Auto-Kontrast Operation (Gl. 5.10). Ein gewisser Prozentsatz (s_{low}, s_{high}) der Bildpixel – dargestellt als entsprechende Flächen am linken bzw. rechten Rand des Histogramms $\mathsf{h}(i)$ – wird auf die Extremwerte abgebildet („gesättigt“), die dazwischenliegenden Werte ($a = \hat{a}_{\text{low}} \ldots \hat{a}_{\text{high}}$) werden linear auf das Intervall $a_{\text{min}} \ldots a_{\text{max}}$ verteilt.

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

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

($M \cdot N$ ist die Anzahl der Bildelemente im Ausgangsbild I). Alle Pixelwerte *außerhalb* von $\hat{a}_{\text{low}}$ und $\hat{a}_{\text{high}}$ werden auf die Extremwerte a_{min} bzw. a_{max} abgebildet, während die dazwischen liegenden Werte von a linear auf das Intervall $[a_{\text{min}}, a_{\text{max}}]$ skaliert werden. Dadurch wird erreicht, dass sich die Abbildung auf die Schwarz- und Weißwerte nicht nur auf einzelne, extreme Pixelwerte stützt, sondern eine repräsentative Zahl von Bildelementen berücksichtigt. Die Punktoperation für die modifizierte Auto-Kontrast-Operation ist daher

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

In der Praxis wird meist $s_{\text{low}} = s_{\text{high}} = s$ angesetzt, mit üblichen Werten für s im Bereich $0.5 \ldots 1.5$ Prozent. Bei der Auto-Kontrast-Operation in *Adobe Photoshop* werden beispielsweise $s = 0.5$ Prozent der Pixel an beiden Enden des Intensitätsbereichs gesättigt. Die Auto-Kontrast-Operation ist ein häufig verwendetes Werkzeug und deshalb in praktisch jeder Bildverarbeitungssoftware verfügbar, u. a. auch in ImageJ (Abb. 5.7). Dabei ist, wie auch in anderen Anwendungen üblich, die in Gl. 5.10 gezeigte Variante implementiert, wie u. a. im logarithmischen Histogramm in Abb. 5.5 (b) deutlich zu erkennen ist.

5.4 Linearer Histogrammausgleich

Ein häufiges Problem ist die Anpassung unterschiedlicher Bilder auf eine (annähernd) übereinstimmende Intensitätsverteilung, etwa für die ge-

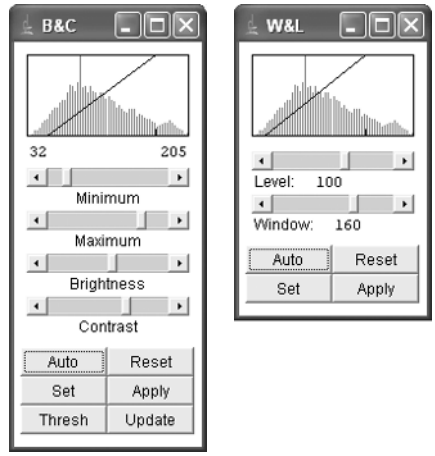


Abbildung 5.7
Interaktive Menüs zur Kontrast- und Helligkeitsanpassung in ImageJ. Das Brightness/Contrast-Tool (links) und das Window/Level-Tool (rechts) sind über das Image→Adjust-Menü erreichbar.

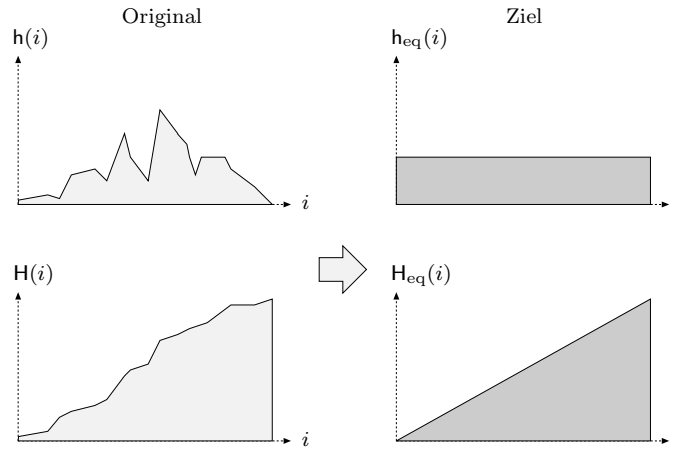


Abbildung 5.8
Idee des Histogrammausgleichs. Durch eine Punktoperation auf ein Bild mit dem ursprünglichen Histogramm $\mathsf{h}(i)$ soll erreicht werden, dass das Ergebnis ein gleichverteiltes Histogramm $\mathsf{h}_{\mathrm{eq}}(i)$ aufweist (oben). Das zugehörige kumulative Histogramm $\mathsf{H}_{\mathrm{eq}}(i)$ wird dadurch keilförmig (unten).

meinsame Verwendung in einem Druckwerk oder um sie leichter miteinander vergleichen zu können. Ziel des Histogrammausgleichs ist es, ein Bild durch eine homogene Punktoperation so zu verändern, dass das Ergebnisbild ein gleichförmig verteiltes Histogramm aufweist (Abb. 5.8). Das kann bei diskreten Verteilungen natürlich nur angenähert werden, denn homogene Punktoperationen können (wie im vorigen Abschnitt diskutiert) Histogrammeinträge nur verschieben oder zusammenfügen, nicht aber *trennen*. Insbesondere können dadurch einzelne Spitzen im Histogramm nicht entfernt werden und daher ist eine *echte* Gleichverteilung nicht zu erzielen. Man kann daher das Bild nur so weit verändern, dass das Ergebnis ein *annähernd gleichverteiltes* Histogramm aufweist. Die Frage ist, was eine gute Näherung bedeutet und *welche* Punktoperation – die klarerweise vom Bildinhalt abhängt – dazu führt.

Eine Lösungsidee gibt uns das kumulative Histogramm (Abschn. 4.6), das bekanntlich für eine gleichförmige Verteilung die Form eines linearen Keils aufweist (Abb. 5.8). Auch das geht natürlich nicht exakt, jedoch

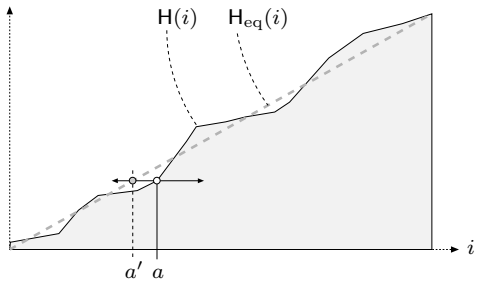


Abbildung 5.9
Durch Anwendung einer geeigneten Punktoperation $a' \leftarrow f_{\text{eq}}(a)$ wird die Histogrammlinie von a so weit (nach links oder rechts) verschoben, dass sich ein annähernd keilförmiges kumulatives Histogramm H_{eq} ergibt.

lassen sich durch eine entsprechende Punktoperation die Histogrammlinien so verschieben, dass sie im kumulativen Histogramm zumindest näherungsweise eine linear ansteigende Funktion bilden (Abb. 5.9).

Die gesuchte Punktoperation $f_{\text{eq}}(a)$ ist auf einfache Weise aus dem kumulativen Histogramm H des ursprünglichen Bilds wie folgt zu berechnen (eine Herleitung findet sich z. B. in [30, p. 173]). Für ein Bild mit $M \times N$ Pixel im Wertebereich $[0, K-1]$ ist die Funktion

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

Die Funktion $f_{\text{eq}}(a)$ in Gl. 5.11 ist monoton steigend, da auch $\mathsf{H}(a)$ monoton ist und K, M und N positive Konstanten sind. Ein Ausgangsbild, das bereits eine Gleichverteilung aufweist, sollte durch einen Histogrammausgleich natürlich nicht verändert werden. Auch eine wiederholte Anwendung des Histogrammausgleichs sollte nach der ersten Anwendung keine weiteren Änderungen im Ergebnis verursachen. Beides trifft für die Formulierung in Gl. 5.11 zu. Der Java-Code für den Histogrammausgleich ist in Prog. 5.2 aufgelistet, ein Beispiel für dessen Anwendung zeigt Abb. 5.10.

Man beachte, dass für „inaktive“ Pixelwerte i, d. h. solche, die im ursprünglichen Bild nicht vorkommen ($\mathsf{h}(i) = 0$), die Einträge im kumulativen Histogramm $\mathsf{H}(i)$ entweder auch Null sind oder identisch zum Nachbarwert $\mathsf{H}(i-1)$. Bereiche mit aufeinander folgenden Nullwerten im Histogramm $\mathsf{h}(i)$ entsprechen konstanten (d. h. flachen) Bereichen im kumulativen Histogramm $\mathsf{H}(i)$. Die Funktion $f_{\text{eq}}(a)$ bildet daher alle „inaktiven“ Pixelwerte innerhalb eines solchen Intervalls auf den nächsten niedrigeren „aktiven“ Wert ab. Da im Bild aber ohnehin keine solchen Pixel existieren, ist dieser Effekt nicht relevant. Wie in Abb. 5.10 deutlich sichtbar, kann ein Histogrammausgleich zum Verschmelzen von Histogrammlinien und damit zu einem Verlust an Bilddynamik führen (s. auch Abschn. 5.2).

Diese oder eine ähnliche Form des Histogrammausgleichs ist in praktisch jeder Bildverarbeitungssoftware implementiert, u. a. auch in ImageJ unter Process→Enhance Contrast (Equalize-Option).

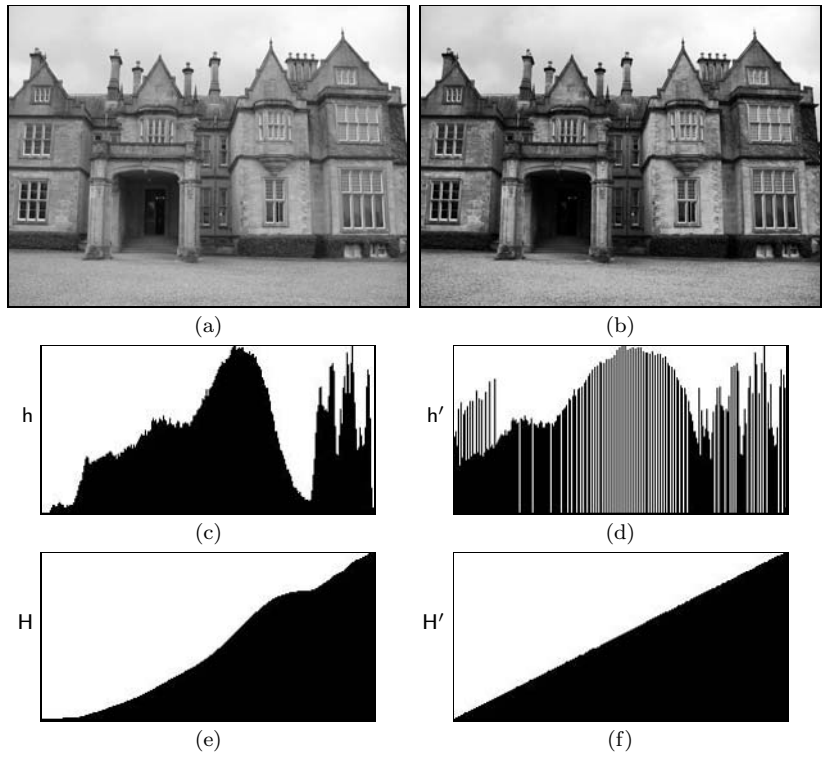


Abbildung 5.10
Linearer Histogrammausgleich (Beispiel). Originalbild I (a) und modifiziertes Bild I' (b), die zugehörigen Histogramme h bzw. h' (c–d) sowie die kumulativen Histogramme H und H' (e–f). Das kumulative Histogramm H' (f) entspricht nach der Operation dem eines gleichverteilten Bilds. Man beachte, dass im Histogramm h' (d) durch zusammenfallende Einträge neue Spitzen enstanden sind, vor allem im unteren und oberen Intensitätsbereich.

Programm 5.2
Histogrammausgleich (ImageJ-Plugin). Zunächst wird (Zeile 8) mit der in ImageJ verfügbaren Methode `ip.getHistogram()` das Histogramm des Bilds `ip` berechnet. Das kumulative Histogramm wird innerhalb desselben Arrays („in place“) berechnet, basierend auf der rekursiven Definition in Gl. 4.6 (Zeile 10). Die Abrundung erfolgt implizit durch die `int`-Division (Zeile 17).

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

5.5 Histogrammanpassung (*Histogram Specification*)

Obwohl weit verbreitet, erscheint das Ziel des im letzten Abschnitt beschriebenen Histogrammausgleichs – die *Gleichverteilung* der Intensitätswerte – recht willkürlich, da auch perfekt aufgenommene Bilder praktisch nie eine derartige Verteilung aufweisen. Meistens ist nämlich die Verteilung der Intensitätswerte nicht einmal annähernd gleichförmig, sondern ähnelt eher einer Gaußfunktion – sofern überhaupt eine allgemeine Verteilungsform relevant ist. Der lineare Histogrammausgleich liefert daher in der Regel unnatürlich wirkende Bilder und ist in der Praxis kaum sinnvoll einsetzbar.

Wertvoller ist hingegen die so genannte „Histogrammanpassung“ (*histogram specification*), die es ermöglicht, ein Bild an eine vorgegebene Verteilungsform oder ein bestehendes Histogramm anzugleichen. Hilfreich ist das beispielsweise bei der Vorbereitung einer Serie von Bildern, die etwa bei unterschiedlichen Aufnahmeverhältnissen oder mit verschiedenen Kameras entstanden sind, aber letztlich in der Reproduktion ähnlich aussehen sollen.

Der Vorgang der Histogrammanpassung basiert wie der lineare Histogrammausgleich auf der Abstimmung der kumulativen Histogramme durch eine homogene Punktoperation. Um aber von der Bildgröße (Anzahl der Pixel) unabhängig zu sein, definieren wir zunächst *normalisierte* Verteilungen, die wir nachfolgend anstelle der Histogramme verwenden.

5.5.1 Häufigkeiten und Wahrscheinlichkeiten

Jeder Eintrag in einem Histogramm beschreibt die beobachtete Häufigkeit des jeweiligen Intensitätswerts – das Histogramm ist daher eine diskrete *Häufigkeitsverteilung*. Für ein Bild I der Größe $M \times N$ ist die Summe aller Einträge in seinem Histogramm h gleich der Anzahl der Pixel, also

$$\text{Sum}(\mathsf{h}) = \sum_i \mathsf{h}(i) = M \cdot N \,. \tag{5.12}$$

Das zugehörige *normalisierte* Histogramm

$$\mathsf{p}(i) = \frac{\mathsf{h}(i)}{\text{Sum}(\mathsf{h})} \,, \quad \text{für } \; 0 \le i < K \,, \tag{5.13}$$

wird üblicherweise als *Wahrscheinlichkeitsverteilung*[3] interpretiert, wobei $\mathsf{p}(i)$ die Wahrscheinlichkeit für das Auftreten des Pixelwerts i darstellt. Die Gesamtwahrscheinlichkeit für das Auftreten eines beliebigen Pixelwerts ist 1 und es muss daher auch für die Verteilung p gelten

$$\sum_i \mathsf{p}(i) = 1 \,. \tag{5.14}$$

[3] Auch „Wahrscheinlichkeitsdichtefunktion“ oder *probability density function* (p.d.f.).

Das Pendant zum *kumulativen* Histogramm H (Gl. 4.5) ist die diskrete *Verteilungsfunktion*[4]

$$\mathsf{P}(i) = \frac{\mathsf{H}(i)}{\mathsf{H}(K-1)} = \frac{\mathsf{H}(i)}{\mathrm{Sum}(\mathsf{h})} = \sum_{j=0}^{i} \frac{\mathsf{h}(j)}{\mathrm{Sum}(\mathsf{h})}$$
$$= \sum_{j=0}^{i} \mathsf{p}(j)\,, \quad \text{für } 0 \le i < K\,. \tag{5.15}$$

Die Funktion $\mathsf{P}(i)$ ist (wie das kumulative Histogramm) monoton steigend und es gilt insbesondere

$$\mathsf{P}(0) = \mathsf{p}(0) \qquad \text{und} \qquad \mathsf{P}(K-1) = \sum_{i=0}^{K-1} \mathsf{p}(i) = 1\,. \tag{5.16}$$

Durch diese statistische Formulierung wird die Erzeugung des Bilds implizit als Zufallsprozess[5] modelliert, wobei die tatsächlichen Eigenschaften des zugrunde liegenden Zufallsprozesses meist nicht bekannt sind. Der Prozess wird jedoch in der Regel als *homogen* (unabängig von der Bildposition) angenommen, d. h. jeder Pixelwert $I(u, v)$ ist das Ergebnis eines Zufallsexperiments mit einer einzigen Zufallsvariablen i. Die Häufigkeitsverteilung im Histogramm $\mathsf{h}(i)$ genügt in diesem Fall als (grobe) Schätzung für die Wahrscheinlichkeitsverteilung $\mathsf{p}(i)$ dieser Zufallsvariablen.

5.5.2 Prinzip der Histogrammanpassung

Das Ziel ist, ein Ausgangsbild I_A durch eine homogene Punktoperation so zu modifizieren, dass seine Verteilungsfunktion P_A mit der Verteilungsfunktion P_R eines gegebenen *Referenzbilds* I_R möglichst gut übereinstimmt. Wir suchen also wiederum nach einer Funktion

$$a' = f_{\mathrm{hs}}(a) \tag{5.17}$$

für eine Punktoperation, die durch Anwendung auf die Pixelwerte a des Ausgangsbilds I_A ein neues Bild $I_{A'}$ mit den Pixelwerten a' erzeugt, so dass seine Verteilungsfunktion $\mathsf{P}_{A'}$ mit der des Referenzbilds übereinstimmt, d. h.

$$\mathsf{P}_{A'}(i) \approx \mathsf{P}_R(i)\,, \quad \text{für } \; 0 \le i < K\,. \tag{5.18}$$

Wie in Abb. 5.11 grafisch dargestellt, finden wir die gesuchte Abbildung f_{hs} durch Kombination der beiden Verteilungsfunktionen P_R und P_A (für Details s. [30, S. 180]). Zu einem gegebenen Pixelwert a im Ausgangsbild

[4] Auch „Kumulierte Wahrscheinlichkeitsdichte“ oder *cumulative distribution function* (c.d.f).

[5] Die statistische Modellierung der Bildgenerierung hat eine lange Tradition (siehe z. B. [49, Kap. 2]).

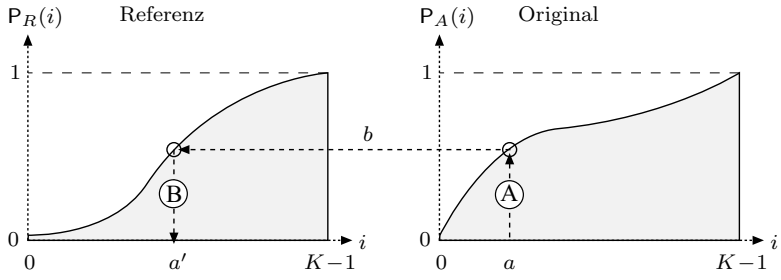


Abbildung 5.11
Prinzip der Histogrammanpassung. Gegeben ist eine Referenzverteilung P_R (links) und die Verteilungsfunktion P_A (rechts) für das Ausgangsbild I_A. Gesucht ist die Abbildung $f_{\text{hs}}\colon a \to a'$, die für jeden ursprünglichen Pixelwert a im Ausgangsbild I_A den modifizierten Pixelwert a' bestimmt. Der Vorgang verläuft in zwei Schritten: Ⓐ Für den Pixelwert a wird zunächst in der rechten Verteilungsfunktion $b = \mathsf{P}_A(a)$ bestimmt. Ⓑ a' ergibt sich dann durch die Inverse der linken Verteilungsfunktion als $a' = \mathsf{P}_R^{-1}(b)$. Insgesamt ist das Ergebnis daher $f_{\text{hs}}(a) = a' = \mathsf{P}_R^{-1}\big(\mathsf{P}_A(a)\big)$.

ermittelt sich der zugehörige neue Pixelwert a' durch

$$a' = \mathsf{P}_R^{-1}\big(\mathsf{P}_A(a)\big) \tag{5.19}$$

und die Abbildung f_{hs} (Gl. 5.17) ergibt sich daraus in der einfachen Form

$$f_{\text{hs}}(a) = \mathsf{P}_R^{-1}\big(\mathsf{P}_A(a)\big)\,, \quad \text{für } 0 \le a < K\,. \tag{5.20}$$

Dies setzt natürlich voraus, dass $\mathsf{P}_R(i)$ invertierbar ist, d. h., dass die Funktion $\mathsf{P}_R^{-1}(b)$ für $b \in [0,1]$ existiert.

5.5.3 Stückweise lineare Referenzverteilung

Liegt die Referenzverteilung P_R als kontinuierliche, invertierbare Funktion vor, dann ist die Abbildung f_{hs} ohne weiteres mit Gl. 5.20 zu berechnen. In der Praxis wird die Verteilung oft als stückweise lineare Funktion $\mathsf{P}_L(i)$ vorgegeben, die wir z. B. als Folge von $N+1$ Koordinatenpaaren

$$\big(\langle i_0, q_0\rangle, \langle i_1, q_1\rangle, \ldots \langle i_k, q_k\rangle, \ldots \langle i_N, q_N\rangle\big)\,,$$

bestehend aus den Intensitätswerten i_k und den zugehörigen Funktionswerten q_k, spezifizieren können. Dabei gilt $0 \le i_k < K$, $i_k < i_{k+1}$ sowie $0 \le q_k < 1$, und für die Invertierbarkeit muss die Funktion streng monoton steigend sein, d. h. $q_k < q_{k+1}$. Zusätzlich fixieren wir die beiden Endpunkte $\langle i_0, q_0\rangle$ bzw. $\langle i_N, q_N\rangle$ mit

$$i_0 = 0 \qquad \text{bzw.} \qquad i_N = K-1\,,\ q_N = 1\,.$$

Abb. 5.12 zeigt ein Beispiel für eine solche Funktion, die durch $N = 5$ variable Punkte $(q_0, \ldots q_4)$ und den fixen Endpunkt (q_5) spezifiziert ist und damit aus $N = 5$ linearen Abschnitten besteht. Durch Einfügen zusätzlicher Polygonpunkte kann die Verteilungsfunktionen natürlich beliebig genau spezifiziert werden.

Die kontinuierlichen Werte dieser Verteilungsfunktion $\mathsf{P}_L(i)$ ergeben sich durch lineare Interpolation in der Form

$$\mathsf{P}_L(i) = \begin{cases} q_m + (i - i_m)\cdot\frac{(q_{m+1}-q_m)}{(i_{m+1}-i_m)} & \text{für } 0 \le i < K-1 \\ 1 & \text{für } i = K-1 \end{cases} \tag{5.21}$$

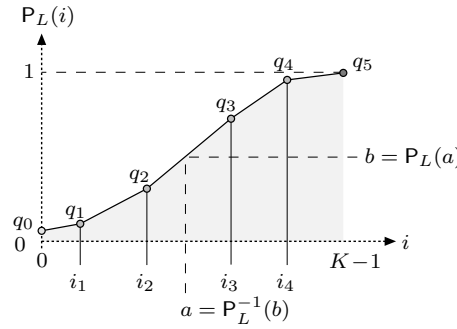


Abbildung 5.12
Stückweise lineare Verteilungsfunktion. $\mathsf{P}_L(i)$ ist spezifiziert durch die $N = 5$ einstellbaren Stützwerte $\langle 0, q_0\rangle$, $\langle i_1, q_1\rangle$, $\ldots \langle i_4, q_4\rangle$, mit $i_k < i_{k+1}$ und $q_k < q_{k+1}$. Zusätzlich ist der obere Endpunkt mit $\langle K-1, 1\rangle$ fixiert.

Dabei ist $m = \max\{j \in \{0,\ldots N-1\} \mid i_j \le i\}$ der Index jenes Polygonsegments $\langle i_m, q_m\rangle \to \langle i_{m+1}, q_{m+1}\rangle$, das die Position i überdeckt. In dem in Abb. 5.12 gezeigten Beispiel liegt etwa der Punkt a innerhalb des Segments mit dem Startpunkt $\langle i_2, q_2\rangle$, also ist $m = 2$.

Zur Histogrammanpassung benötigen wir nach Gl. 5.20 die inverse Verteilungsfunktion $\mathsf{P}_L^{-1}(b)$ für $b \in [0, 1]$. Wir sehen am Beispiel in Abb. 5.12, dass die Funktion $\mathsf{P}_L(i)$ für Werte $b < \mathsf{P}_L(0)$ im Allgemeinen nicht invertierbar ist. Wir behelfen uns damit, dass wir alle Werte $b < \mathsf{P}_L(0)$ auf $i = 0$ abbilden, und erhalten so die zu Gl. 5.21 (quasi-)inverse Verteilungsfunktion:

$$\mathsf{P}_L^{-1}(b) = \begin{cases} 0 & \text{für } 0 \le b < \mathsf{P}_L(0) \\ i_n + (b - q_n) \cdot \frac{(i_{n+1} - i_n)}{(q_{n+1} - q_n)} & \text{für } \mathsf{P}_L(0) \le b < 1 \\ K-1 & \text{für } b \ge 1 \end{cases} \tag{5.22}$$

Dabei ist $n = \max\{j \in \{0,\ldots N-1\} \mid q_j \le b\}$ der Index jenes Linearsegments $\langle i_n, q_n\rangle \to \langle i_{n+1}, q_{n+1}\rangle$, das den Argumentwert b überdeckt. Die Berechnung der für die Histogrammanpassung notwendigen Abbildung f_{hs} für ein gegebenes Bild mit der Verteilungsfunktion P_A erfolgt schließlich analog zu Gl. 5.20 durch

$$f_{\text{hs}}(a) = \mathsf{P}_L^{-1}\big(\mathsf{P}_A(a)\big), \quad \text{für } 0 \le a < K. \tag{5.23}$$

Ein konkretes Beispiel für die Histogrammanpassung mit einer stückweise linearen Verteilungsfunktion ist in Abb. 5.14 (Abschn. 5.5.5) gezeigt.

5.5.4 Anpassung an ein konkretes Histogamm

Bei der Anpassung an ein konkretes Histogramm ist die vorgegebene Verteilungsfunktion (Referenz) $\mathsf{P}_R(i)$ nicht stetig und kann daher im Allgemeinen nicht invertiert werden. Gibt es beispielsweise Lücken im Histogramm, also Pixelwerte k mit Wahrscheinlichkeit $\mathsf{p}(k) = 0$, dann weist die zugehörige Verteilungsfunktion P (wie auch das kumulative Histogramm) flache Stellen mit konstanten Funktionswerten auf, an denen die Funktion nicht invertierbar ist.

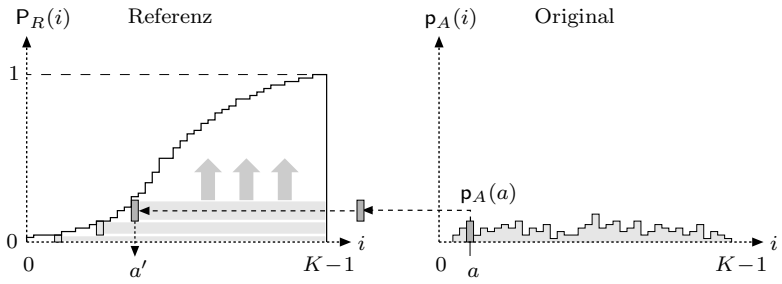


Abbildung 5.13
Diskrete Histogrammanpassung. Die Referenzverteilung P_R (links) wird schichtweise von unten nach oben und von rechts nach links „befüllt“. Dabei wird für jeden Pixelwert a des Ausgangsbilds I_A der zugehörige Wahrscheinlichkeitswert $\mathsf{p}_A(a)$ (rechts) als horizontaler Balken unterhalb der Verteilung P_R aufgetragen. Der Balken mit der Höhe $\mathsf{p}_A(a)$ wird von rechts nach links gezogen bis zur Stelle a', an der die Verteilungsfunktion P_R erreicht wird. a' ist das gesuchte Ergebnis der Abbildung $f_{\text{hs}}(a)$, die anschließend auf das Ausgangsbild I_A anzuwenden ist.

Im Folgenden beschreiben wir ein einfaches Verfahren zur Histogrammanpassung, das im Unterschied zu Gl. 5.20 mit diskreten Verteilungen arbeitet. Die grundsätzliche Idee ist zunächst in Abb. 5.13 veranschaulicht. Die Berechnung der Abbildung f_{hs} erfolgt hier nicht durch Invertieren, sondern durch schrittweises „Ausfüllen“ der Verteilungsfunktion $\mathsf{P}_R(i)$. Dabei wird, beginnend bei $i = 0$, für jeden Pixelwert i des Ausgangsbilds I_A der zugehörige Wahrscheinlichkeitswert $\mathsf{p}_A(i)$ von rechts nach links und Schicht über Schicht innerhalb der Referenzverteilung P_R aufgetragen. Die Höhe jedes horizontal aufgetragenen Balkens entspricht der aus dem Originalhistogramm berechneten Wahrscheinlichkeit $\mathsf{p}_A(i)$. Der Balken für einen bestimmten Grauwert a mit der Höhe $\mathsf{p}_A(a)$ läuft also nach links bis zu jener Stelle a', an der die Verteilungsfunktion P_R erreicht wird. Diese Position a' entspricht dem zu a gehörigen neuen Pixelwert.

Da die Summe aller Wahrscheinlichkeiten p_A und das Maximum der Verteilungsfunktion P_R jeweils 1 sind, d. h. $\sum_i \mathsf{p}_A(i) = \max_i \mathsf{P}_R(i) = 1$, können immer alle „Balken“ innerhalb von P_R untergebracht werden. Aus Abb. 5.13 ist auch zu erkennen, dass der an der Stelle a' resultierende Verteilungswert nichts anderes ist als der kumulierte Wahrscheinlichkeitswert $\mathsf{P}_A(a)$. Es genügt also, für einen gegebenen Pixelwert a den minimalen Wert a' zu finden, an dem die Referenzverteilung $\mathsf{P}_R(a')$ größer oder gleich der kumulierten Wahrscheinlichkeit $\mathsf{P}_A(a)$ ist, d. h.

$$f_{\text{hs}}(a) = \min\big\{\, j \mid (0 \le j < K) \wedge \big(\mathsf{P}_A(a) \le \mathsf{P}_R(j)\big)\big\}\,. \tag{5.24}$$

In Alg. 5.1 ist dieser Vorgang nochmals übersichtlich zusammengefasst und Prog. 5.3 zeigt dann die direkte Umsetzung des Algorithmus in Java. Sie besteht im Wesentlichen aus der Methode `matchHistograms()`, die unter Vorgabe des Ausgangshistogramms `Ha` und eines Referenzhistogramms `Hr` die Abbildung `map` für das zugehörige Ausgangsbild liefert.

Durch die Verwendung von normalisierten Verteilungsfunktionen ist die Größe der den Histogrammen `hA` und `hR` zugrunde liegenden Bilder natürlich nicht relevant. Nachfolgend ein kurzer Programmabschnitt, der die Verwendung der Methode `matchHistograms()` aus Prog. 5.3 in ImageJ demonstriert:

Algorithmus 5.1 Histogrammanpassung. Gegeben sind das Histogramm h_A des Originalbilds I_A und das Referenzhistogramm h_R, jeweils mit K Elementen. Das Ergebnis ist eine diskrete Abbildungsfunktion $F(a)$, die bei Anwendung auf das Originalbild I_A ein neues Bild $I_{A'}(u,v) \leftarrow f_{\text{hs}}\big(I_A(u,v)\big)$ erzeugt, das eine ähnliche Verteilungsfunktion wie das Referenzbild aufweist.

```
1:  MatchHistograms(h_A, h_R)
2:      Let K ← Size(h_A)        ▷ histograms h_A, h_R must be of same size K
3:      Let P_A ← Cdf(h_A)       ▷ cumulative distribution function for h_A
4:      Let P_R ← Cdf(h_R)       ▷ cumulative distribution function for h_R
5:      Create table F of size K                    ▷ pixel mapping function f_hs
6:      for a ← 0 ... (K−1) do
7:          j ← K−1
8:          repeat
9:              F(a) ← j
10:             j ← j − 1
11:         while (j ≥ 0) ∧ (P_A(a) ≤ P_R(j))
12:     return F.

13: Cdf(h)
        Returns the cumulative distribution function (c.d.f.) P(i) ∈ [0, 1] for
        a discrete histogram h of length K.
14:     Let K ← Size(h)
15:     Let n ← Σ_{i=0}^{K−1} h(i)                                  ▷ Sum(h)
16:     Create table P of size K
17:     Let c ← h(0)
18:     P(0) ← c/n
19:     for i ← 1 ... (K−1) do
20:         c ← c + h(i)                          ▷ cumulate histogram values
21:         P(i) ← c/n
22:     return P.
```

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

Die eigentliche Modifikation des Ausgangsbilds `ipA` durch die Abbildung f_{hs} (`F`) erfolgt in der letzten Zeile mit der ImageJ-Methode `applyTable()` (s. auch S. 82).

5.5.5 Beispiele

Stückweise lineare Verteilungsfunktion

Das erste Beispiel in Abb. 5.14 zeigt die Histogrammanpassung mit einer kontinuierlich definierten, stückweise linearen Verteilungsfunktion, wie in Abschn. 5.5.3 beschrieben. Die konkrete Verteilungsfunktion P_R (Abb. 5.14(f)) ist analog zu Abb. 5.12 durch einen Polygonzug definiert, bestehend aus 5 Kontrollpunkten $\langle i_k, q_k \rangle$ mit den Koordinaten

Programm 5.3
Histogrammanpassung nach Alg. 5.1 (Java-Implementierung). Die Methode `matchHistograms()` berechnet aus dem Histogramm `Ha` und dem Referenzhistogramm `Hr` die Abbildung `F` (Gl. 5.24). Die in Zeile 7 verwendete Methode `Cdf()` zur Berechnung der kumulierten Verteilungsfunktion (Gl. 5.15) ist im unteren Abschnitt ausgeführt.

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

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

$k =$	0	1	2	3	4
$i_k =$	0	28	75	150	210
$q_k =$	0.002	0.050	0.250	0.750	0.950

definiert (vgl. Abb. 5.12). Das zugehörige Referenzhistogramm (Abb. 5.14 (c)) ist stufenförmig, wobei die linearen Segmente in der Verteilungsfunktion den konstanten Abschnitten in der Wahrscheinlichkeitsdichtefunktion bzw. im Histogramm entsprechen. Die Verteilungsfunktion des angepassten Bilds (Abb. 5.14 (h)) stimmt weitgehend mit der

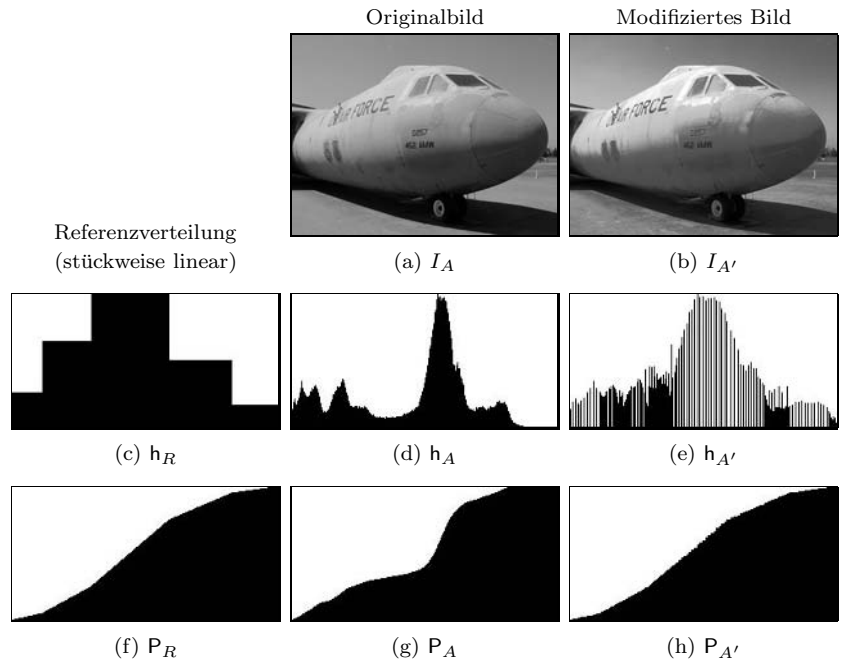


Abbildung 5.14
Histogrammanpassung mit stückweise linearer Referenzverteilung. Ausgangsbild I_A (a) und seine ursprüngliche Verteilung P_A (g), Referenzverteilung P_R (f), modifiziertes Bild $I_{A'}$ (b) und die resultierende Verteilung $\mathsf{P}_{A'}$ (h). In (c–e) sind die zugehörigen Histogramme h_R, h_A bzw. $\mathsf{h}_{A'}$ dargestellt.

Referenzfunktion überein. Das resultierende Histogramm (Abb. 5.14 (e)) weist naturgemäß hingegen wenig Ähnlichkeit mit der Vorgabe auf, doch mehr ist bei einer homogenen Punktoperation auch nicht zu erwarten.

Gaußförmiges Referenzhistogramm

Ein Beispiel für die Anpassung eines Bilds an ein konkretes Histogramm ist in Abb. 5.15 gezeigt. In diesem Fall ist die Verteilungsfunktion nicht kontinuierlich, sondern über ein diskretes Histogramm vorgegeben. Die Berechnung der Histogrammanpassung erfolgt daher nach der in Abschn. 5.5.4 beschriebenen Methode.

Das hier verwendete Ausgangsbild ist wegen seines extrem unausgeglichenen Histogramms bewusst gewählt und die resultierenden Histogramme des modifizierten Bilds zeigen naturgemäß wenig Ähnlichkeit mit einer Gauß'schen Kurvenform. Allerdings sind die zugehörigen kumulativen Histogramme durchaus ähnlich und entsprechen weitgehend dem Integral der jeweiligen Gaußfunktion, sieht man von den durch die einzelnen Histogrammspitzen hervorgerufenen Sprungstellen ab.

Histogrammanpassung an ein zweites Bild

Das dritte Beispiel in Abb. 5.16 demonstriert die Anpassung zweier Bilder in Bezug auf ihre Grauwerthistogramme. Eines der Bilder in Abb. 5.16(a) dient als Referenzbild I_R und liefert das Referenzhistogramm h_R (Abb. 5.16 (d)). Das zweite Bild I_A (Abb. 5.16 (b)) wird so modifiziert, dass

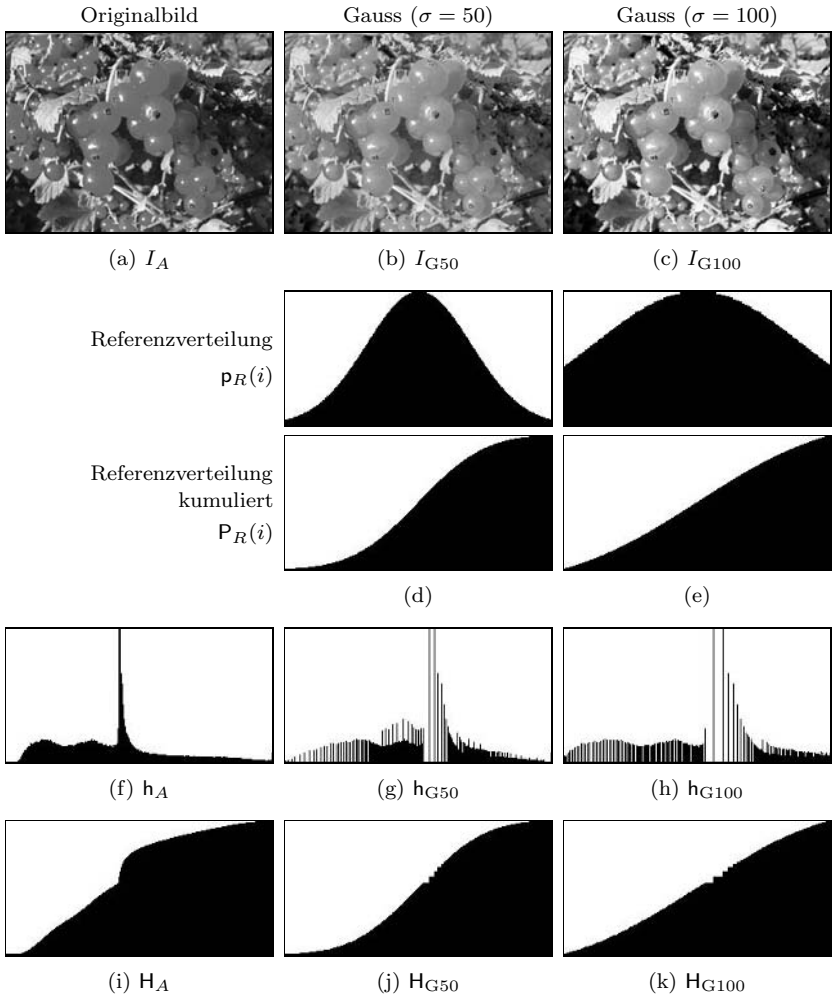


Abbildung 5.15
Anpassung an ein Gauß-förmiges Referenzhistogramm. Ausgangsbild I_A (a) und das zugehörige Histogramm (f) bzw. kumulative Histogramm (i). Die Gauß-förmigen Referenzhistogramme (d–e) haben $\mu = 128$ als Mittelwert und $\sigma = 50$ (d) bzw. $\sigma = 100$ (e). Die Ergebnisbilder I_{G50} (b) und I_{G100} (c) nach der Histogrammanpassung, die zugehörigen Histogramme (g–h) sowie die kumulativen Histogramme (j–k).

sein kumulatives Histogramm mit dem kumulativen Referenzhistogramm übereinstimmt. Das resultierende Bild $I_{A'}$ (Abb. 5.16 (c)) sollte bezüglich Tonumfang und Intensitätsverteilung dem Referenzbild sehr ähnlich sein.

Natürlich können mit dieser Methode auch mehrere Bilder auf das gleiche Referenzbild angepasst werden, etwa für den Druck einer Fotoserie, in der alle Bilder möglichst ähnlich aussehen sollen. Dabei kann entweder ein besonders typisches Exemplar als Referenzbild ausgewählt werden, oder man berechnet aus allen Bildern eine „durchschnittliche“ Referenzverteilung für die Anpassung (s. auch Aufg. 5.6).

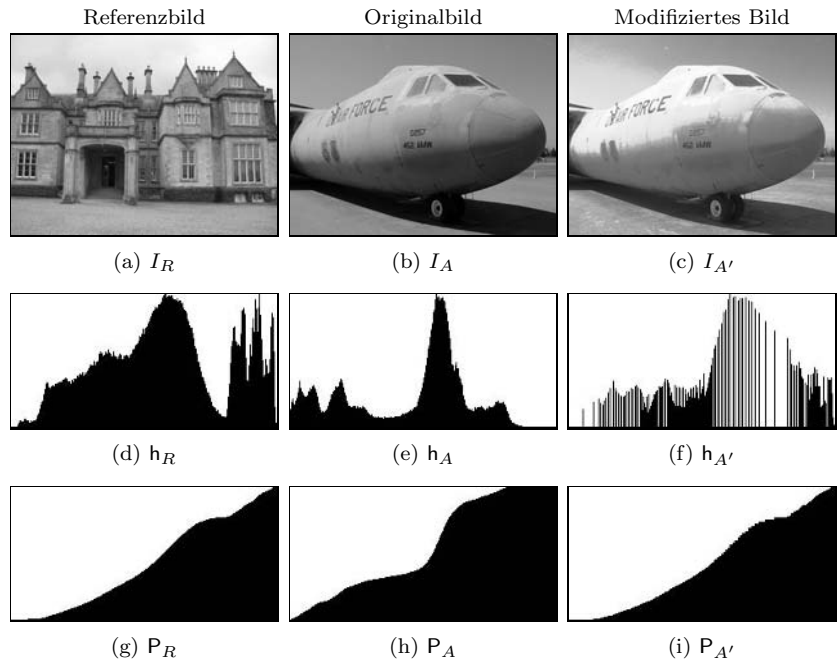


Abbildung 5.16
Histogrammanpassung an ein vorgegebenes Bild. Das Ausgangsbild I_A (b) wird durch eine Histogrammanpassung an das Referenzbild I_R (a) angeglichen, das Ergebnis ist das modifizierte Bild $I_{A'}$ (c). Darunter sind die zugehörigen Histogramme h_R, h_A, $\mathsf{h}_{A'}$ (d–f) sowie die kumulativen Histogramme (bzw. Verteilungsfunktionen) P_R, P_A, $\mathsf{P}_{A'}$ (g–i) gezeigt. Die Verteilungsfunktionen des Referenzbilds (g) und des modifizierten Bilds (i) stimmen offensichtlich weitgehend überein.

5.6 Gammakorrektur

Wir haben schon mehrfach die Ausdrücke „Intensität“ oder „Helligkeit“ verwendet, im stillen Verständnis, dass die numerischen Pixelwerte in unseren Bildern in irgendeiner Form mit diesen Begriffen zusammenhängen. In welchem Verhältnis stehen aber die Pixelwerte wirklich zu physischen Größen, wie z. B. zur Menge des einfallenden Lichts, der Schwärzung des Filmmaterials oder der Anzahl von Tonerpartikeln, die von einem Laserdrucker auf das Papier gebracht werden? Tatsächlich ist das Verhältnis zwischen Pixelwerten und den zugehörigen physischen Größen meist komplex und praktisch immer nichtlinear. Es ist jedoch wichtig, diesen Zusammenhang zumindest annähernd zu kennen, damit das Aussehen von Bildern vorhersehbar und reproduzierbar wird.

Ideal wäre dabei ein „kalibrierter Intensitätsraum“, der dem visuellen Intensitätsempfinden möglichst nahe kommt und einen möglichst großen Intensitätsbereich mit möglichst wenig Bits beschreibt. Die Gammakorrektur ist eine einfache Punktoperation, die dazu dient, die unterschiedlichen Charakteristiken von Aufnahme- und Ausgabegeräten zu kompensieren und Bilder auf einen gemeinsamen Intensitätsraum anzupassen.

5.6.1 Warum Gamma?

Der Ausdruck „Gamma“ stammt ursprünglich aus der „analogen“ Fototechnik, wo zwischen der Belichtungsstärke und der resultierenden Film-

dichte (Schwärzung) ein annähernd logarithmischer Zusammenhang besteht. Die so genannte „Belichtungsfunktion“ stellt den Zusammenhang zwischen der logarithmischen Belichtungsstärke und der resultierenden Filmdichte dar und verläuft über einen relativ großen Bereich als ansteigende Gerade (Abb. 5.17). Die Steilheit der Belichtungsfunktion innerhalb dieses geraden Bereichs wird traditionall als „Gamma“ des Filmmaterials bezeichnet. Später war man in der elektronischen Fernsehtechnik mit dem Problem konfrontiert, die Nichtlinearitäten der Bildröhren in Empfangsgeräten zu beschreiben und übernahm dafür ebenfalls den Begriff „Gamma“. Das TV-Signal wurde im Sender durch eine so genannte „Gammakorrektur“ vorkorrigiert, damit in den Empfängern selbst keine aufwendigen Maßnahmen mehr erforderlich waren.

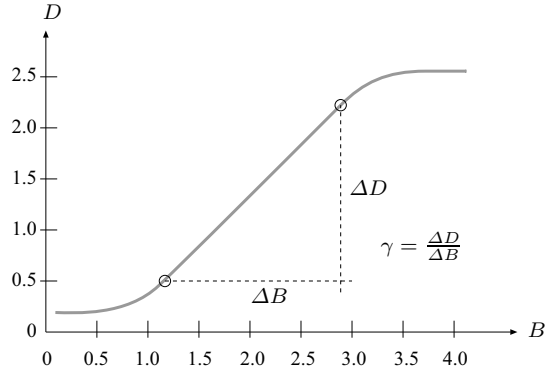


Abbildung 5.17
Belichtungskurve von fotografischem Film. Bezogen auf die logarithmische Beleuchtungsstärke B verläuft die resultierende Dichte D in einem weiten Bereich annähernd als Gerade. Die Steilheit dieses linearen Anstiegs bezeichnet man als „Gamma“ (γ) des Filmmaterials.

5.6.2 Die Gammafunktion

Grundlage der Gammakorrektur ist die Gamma*funktion*

$$b = f_\gamma(a) = a^\gamma \quad \text{für } a \in \mathbb{R}\,, \gamma > 0\,, \tag{5.25}$$

mit dem Parameter γ, dem so genannten *Gammawert*. Verwenden wir die Gammafunktion nur innerhalb des Intervalls $[0, 1]$, dann bleibt auch – unabhängig von γ – der Funktionswert a^γ im Intervall $[0, 1]$, und die Funktion verläuft immer durch die Punkte $(0, 0)$ und $(1, 1)$. Wie Abb. 5.18 zeigt, ergibt sich für $\gamma = 1$ die identische Funktion $f_\gamma(a) = a$, also eine Diagonale. Für Gammawerte $\gamma < 1$ verläuft die Funktion *oberhalb* dieser Geraden und für $\gamma > 1$ *unterhalb*, wobei die Krümmung mit der Abweichung vom Wert 1 nach beiden Seiten hin zunimmt. Die Gammafunktion kann also, gesteuert mit nur einem Parameter, einen kontinuierlichen Bereich von Funktionen mit sowohl logarithmischem wie auch exponentiellem Verhalten „imitieren“. Sie ist darüber hinaus im Definitionsbereich $[0, 1]$ stetig und streng monoton und zudem sehr einfach zu invertieren:

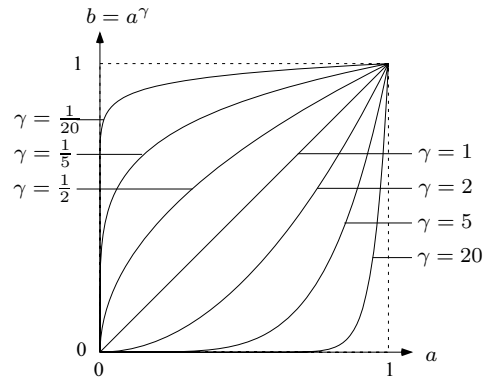


Abbildung 5.18
Gammafunktion $b = a^\gamma$ für $a \in [0, 1]$ und verschiedene Gammawerte.

$$a = f_\gamma^{-1}(b) = b^{1/\gamma} = f_{\bar{\gamma}}(b). \tag{5.26}$$

Die Umkehrung der Gammafunktion $f_\gamma(a)$ ist also wieder eine Gammafunktion $f_{\bar{\gamma}}(b)$ mit $\bar{\gamma} = \frac{1}{\gamma}$.

5.6.3 Reale Gammawerte

Die konkreten Gammawerte einzelner Geräte werden in der Regel von ihren Herstellern aufgrund von Messungen spezifiziert. Zum Beispiel liegen übliche Gammawerte für Röhrenmonitore im Bereich $1.8 \ldots 2.8$, ein typischer Wert ist 2.4. LCD-Monitore sind durch entsprechende Korrekturen auf ähnliche Werte voreingestellt. Video- und Digitalkameras emulieren ebenfalls durch interne Vorverarbeitung der Videosignale das Belichtungsverhalten von Film- bzw. Fotokameras, um den resultierenden Bildern ein ähnliches Aussehen zu geben.

In der Fernsehtechnik ist der theoretische Gammawert für Wiedergabegeräte mit 2.2 im NTSC- und 2.8 im PAL-System spezifiziert, wobei die tatsächlichen gemessenen Werte bei etwa 2.35 liegen. Für Aufnahmegeräte gilt sowohl im amerikanischen NTSC-System wie auch in der europäischen Norm[6] ein standardisierter Gammawert von $1/2.2 \approx 0.45$. Die aktuelle internationale Norm ITU-R BT.709[7] sieht einheitliche Gammawerte für Wiedergabegeräte von 2.5 bzw. $1/1.956 \approx 0.51$ für Kameras vor [24, 43]. Der ITU 709-Standard verwendet allerdings eine leicht modifizierte Form der Gammafunktion (s. Abschn. 5.6.6).

Bei Computern ist in der Regel der Gammawert für das Video-Ausgangssignal zum Monitor in einem ausreichenden Bereich einstellbar. Man muss dabei allerdings beachten, dass die Gammafunktion oft nur ein grobe Annäherung an das tatsächliche Transferverhalten eines Geräts darstellt, außerdem für die einzelnen Farbkanäle unterschiedlich sein kann und in der Realität daher beachtliche Abweichungen auftreten können. Kritische Anwendungen wie z. B. die digitale Druckvorstufe erfordern daher eine aufwendigere Kalibrierung mit exakt vermes-

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

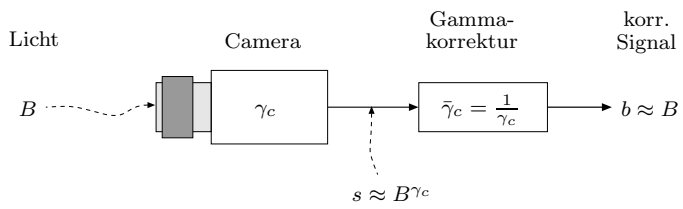


Abbildung 5.19
Prinzip der Gammakorrektur. Um das von einer Kamera mit dem Gammawert γ_c erzeugte Ausgangssignal s zu korrigieren, wird eine Gammakorrektur mit $\bar{\gamma}_c = 1/\gamma_c$ eingesetzt. Das korrigierte Signal b wird damit proportional zur einfallenden Lichtintensität B.

senen Geräteprofilen (siehe Abschn. 12.3.5), wofür eine einfache Gammakorrektur nicht ausreicht.

5.6.4 Anwendung der Gammakorrektur

Angenommen wir benutzen eine Kamera mit einem angegebenen Gammawert γ_c, d. h., ihr Ausgangssignal s steht mit der einfallenden Lichtintensität B im Zusammenhang

$$s = B^{\gamma_c} \,. \tag{5.27}$$

Um die Transfercharakteristik der Kamera zu kompensieren, also eine Messung b proportional zur Lichtintensität B zu erhalten ($b \approx B$), unterziehen wir das Kamerasignal s einer Gammakorrektur mit dem *inversen* Gammawert der Kamera $\bar{\gamma}_c = 1/\gamma_c$, also

$$b = f_{\bar{\gamma}_c}(s) = s^{1/\gamma_c}. \tag{5.28}$$

Für das Ergebnis gilt

$$b \;=\; s^{1/\gamma_c} \;=\; \left(B^{\gamma_c}\right)^{1/\gamma_c} \;=\; B^{(\gamma_c \frac{1}{\gamma_c})} \;=\; B^1 \,, \tag{5.29}$$

d. h., das korrigierte Signal b ist proportional (bzw. identisch) zur ursprünglichen Lichtintensität B (Abb. 5.19). Die allgemeine Regel, die genauso auch für Ausgabegeräte gilt, ist daher:

> Die Transfercharakteristik eines Geräts mit einem Gammawert γ wird kompensiert durch eine Gammakorrektur mit $\bar{\gamma} = 1/\gamma$.

Dabei wurde implizit angenommen, dass alle Werte im Intervall $[0, 1]$ liegen. Natürlich ist das in der Praxis meist nicht der Fall. Insbesondere liegen bei der Korrektur von digitalen Bildern üblicherweise diskrete Pixelwerte vor, beispielsweise im Bereich $[0, 255]$. Im Allgemeinen ist eine Gammakorrektur

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

für einen Pixelwert $a \in [0, a_{\max}]$ und einen Gammawert $\gamma > 0$, mit folgenden drei Schritten verbunden:

1. Skaliere a linear auf $\hat{a} \in [0, 1]$.
2. Wende auf $\hat{a}$ die Gammafunktion an: $\hat{b} \leftarrow f_\gamma(\hat{a}) = \hat{a}^\gamma$.

3. Skaliere $\hat{b} \in [0, 1]$ linear zurück auf $b \in [0, a_{\max}]$.

Oder, etwas kompakter formuliert:

$$b \leftarrow f_{\text{gc}}(a, \gamma) = \left(\frac{a}{a_{\max}}\right)^{\gamma} \cdot a_{\max} \tag{5.30}$$

Abb. 5.20 illustriert den Einsatz der Gammakorrektur anhand eines konkreten Szenarios mit je zwei Aufnahmegeräten (Kamera, Scanner) und Ausgabegeräten (Monitor, Drucker), die alle unterschiedliche Gammawerte aufweisen. Die Kernidee ist, dass alle Bilder geräteunabhängig in einem einheitlichen Intensitätsraum gespeichert und verarbeitet werden können.

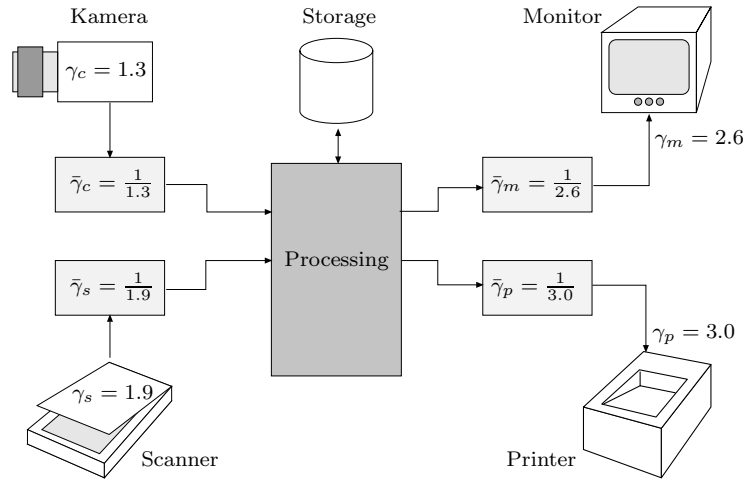


Abbildung 5.20
Einsatz der Gammakorrektur im digitalen Imaging-Workflow. Die eigentliche Verarbeitung wird in einem „linearen“ Intensitätsraum durchgeführt, wobei die spezifische Transfercharakteristik jedes Ein- und Ausgabegeräts durch eine entsprechende Gammakorrektur ausgeglichen wird. (Die angegebenen Gammawerte sind nur als Beispiele gedacht.)

5.6.5 Implementierung

Prog. 5.4 zeigt die Implementierung der Gammakorrektur als ImageJ-Plugin für 8-Bit-Grauwertbilder und einem fixen Gammawert. Die eigentliche Punktoperation ist als Anwendung einer Transformationstabelle (*lookup table*) und der ImageJ-Methode `applyTable()` realisiert (siehe auch Abschn. 5.7.1).

5.6.6 Modifizierte Gammafunktion

Ein Problem bei der Kompensation der Nichtlinearitäten mit der einfachen Gammafunktion $f_\gamma(a) = a^\gamma$ (Gl. 5.25) ist der Anstieg der Funktion in der Nähe des Nullpunkts, ausgedrückt durch ihre erste Ableitung $f'_\gamma(a) = \gamma \cdot a^{(\gamma-1)}$, wodurch

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

Programm 5.4
Gammakorrektur (ImageJ-Plugin). Der Gammawert `GAMMA` ist konstant. Die korrigierten Werte `b` werden nur einmal berechnet und in der Transformationstabelle (`lut`) eingefügt (Zeile 14). Die eigentliche Punktoperation erfolgt durch Anwendung der ImageJ-Methode `applyTable(lut)` auf das Bildobjekt `ip` (Zeile 17).

$$f'_\gamma(0) = \begin{cases} 0 & \text{für } \gamma > 1 \\ 1 & \text{für } \gamma = 1 \\ \infty & \text{für } \gamma < 1 \end{cases} \tag{5.31}$$

Dieser Umstand bewirkt zum einen eine extrem hohe Verstärkung und damit in der Praxis eine starke Rauschanfälligkeit im Bereich der niedrigen Intensitätswerte, zum anderen ist die Gammafunktion am Nullpunkt theoretisch nicht invertierbar.

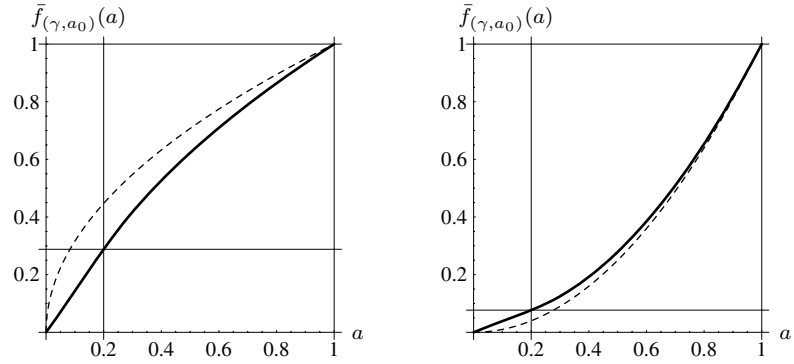
Modifizierte Gammakorrektur

Die gängige Lösung dieses Problems besteht darin, innerhalb eines begrenzten Bereichs $0 \le a \le a_0$ in der Nähe des Nullpunkts zunächst eine *lineare* Korrekturfunktion mit fixem Anstieg s zu verwenden und erst ab dem Punkt $a = a_0$ mit der Gammafunktion fortzusetzen. Die damit erzeugte Korrekturfunktion

$$\bar{f}_{(\gamma,a_0)}(a) = \begin{cases} s \cdot a & \text{für } 0 \le a \le a_0 \\ (1+d) \cdot a^\gamma - d & \text{für } a_0 < a \le 1 \end{cases} \tag{5.32}$$

$$\text{mit} \quad s = \frac{\gamma}{a_0(\gamma-1) + a_0^{(1-\gamma)}} \qquad \text{und} \qquad d = \frac{1}{a_0^\gamma(\gamma-1)+1} - 1 \tag{5.33}$$

teilt sich also in einen linearen Abschnitt ($0 \le a \le a_0$) und einen nichtlinearen Abschnitt ($a_0 < a \le 1$). Die Werte für die Steilheit des linearen Teils s und den Parameter d ergeben sich aus der Bedingung, dass an der Übergangsstelle $a = a_0$ für beide Funktionsteile sowohl $\bar{f}_{(\gamma,a_0)}(a)$ wie auch die erste Ableitung $\bar{f}'_{(\gamma,a_0)}(a)$ identisch sein müssen, um eine kontinuierliche Gesamtfunktion zu erzeugen. Abb. 5.21 zeigt zur Illustration



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

Abbildung 5.21
Gammakorrektur mit Offset. Die modifizierte Korrekturfunktion $\bar{f}_{(\gamma,a_0)}(a)$ verläuft innerhalb des Bereichs $a = 0 \ldots a_0$ linear mit fixem Anstieg s und geht an der Stelle $a = a_0$ in eine Gammafunktion mit dem Parameter γ über (Gl. 5.32).

zwei Beispiele für die Funktion $\bar{f}_{(\gamma,a_0)}(a)$ mit den Werten $\gamma = 0.5$ bzw. $\gamma = 2.0$. In beiden Fällen ist die Übergangsstelle $a_0 = 0.2$. Zum Vergleich ist jeweils auch die gewöhnliche Gammafunktion $f_\gamma(a)$ mit identischem γ gezeigt (unterbrochene Linie), die am Nullpunkt einen Anstieg von ∞ (Abb. 5.21 (a)) bzw. 0 (Abb. 5.21 (b)) aufweist.

Gammakorrektur in gängigen Standards

Im Unterschied zu den illustrativen Beispielen in Abb. 5.21 sind in der Praxis für a_0 wesentlich kleinere Werte üblich und γ muss so gewählt werden, dass die vorgesehene Korrekturfunktion insgesamt optimal angenähert wird. Beispielsweise gibt die in Abschn. 5.6.3 bereits erwähnte Spezifikation ITU-BT.709 [43] die Werte

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

vor, woraus sich gemäß Gl. 5.33 die Werte $s = 4.50681$ bzw. $d = 0.0991499$ ergeben. Diese Korrekturfunktion $\bar{f}_{\text{ITU}}(a)$ mit dem nominellen Gammawert 0.45 entspricht einem *effektiven* Gammawert $\gamma_{\text{eff}} = 1/1.956 \approx 0.511$. Auch im sRGB-Standard [81] (siehe auch Abschn. 12.3.3) ist die Intensitätskorrektur auf dieser Basis spezifiziert.

Abb. 5.22 zeigt die Korrekturfunktionen für den ITU- bzw. sRGB-Standard jeweils im Vergleich mit der entsprechenden gewöhnlichen Gammafunktion. Die ITU-Charakteristik (Abb. 5.22 (a)) mit $\gamma = 0.45$ und $a_0 = 0.018$ entspricht einer gewöhnlichen Gammafunktion mit effektivem Gammawert $\gamma_{\text{eff}} = 0.511$ (unterbrochene Linie). Die Kurven für sRGB (Abb. 5.22 (b)) unterscheiden sich nur durch die Parameter γ und a_0 (s. Tabelle 5.1).

Inverse Korrektur

Um eine modifizierte Gammakorrektur der Form $b = \bar{f}_{(\gamma,a_0)}(a)$ (Gl. 5.32) rückgängig zu machen, benötigen wir die zugehörige inverse Funktion, d. h. $a = \bar{f}^{-1}_{(\gamma,a_0)}(b)$, die wiederum stückweise definiert ist:

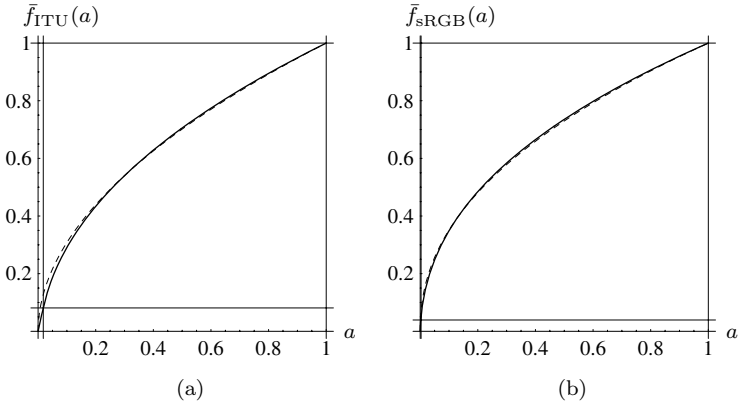


Abbildung 5.22
Korrekturfunktion gemäß ITU-R BT.709 (a) und sRGB (b). Die durchgehende Linie zeigt die modifizierte Gammafunktion (mit Offset) mit dem nominellen Gammawert γ und Übergangspunkt a_0.

Standard	nomineller Gammawert γ	a_0	s	d	effektiver Gammawert γ_{eff}
ITU-R BT.709	$1/2.222 \approx 0.450$	0.01800	4.5068	0.09915	$1/1.956 \approx 0.511$
sRGB	$1/2.400 \approx 0.417$	0.00304	12.9231	0.05500	$1/2.200 \approx 0.455$

Tabelle 5.1
Parameter für Standard-Korrekturfunktionen auf Basis der modifizierten Gammafunktion gemäß Gl. 5.32–5.33.

$$\bar{f}^{-1}_{(\gamma,a_0)}(b) = \begin{cases} \frac{b}{s} & \text{für } 0 \le b \le s \cdot a_0 \\ \left(\frac{b+d}{1+d}\right)^{\frac{1}{\gamma}} & \text{für } s \cdot a_0 < b \le 1 \end{cases} \tag{5.34}$$

Dabei sind s und d die Werte aus Gl. 5.33 und es gilt daher

$$a = \bar{f}^{-1}_{(\gamma,a_0)}\big(\bar{f}_{(\gamma,a_0)}(a)\big) \qquad \text{für} \quad a \in [0,1], \tag{5.35}$$

wobei zu beachten ist, dass in beiden Funktionen in Gl. 5.35 auch *derselbe* Wert für γ verwendet wird. Die Umkehrfunktion ist u. a. für die Umrechnung zwischen unterschiedlichen Farbräumen erforderlich, wenn nichtlineare Komponentenwerte dieser Form im Spiel sind (siehe auch Abschn. 12.3.2).

5.7 Punktoperationen in ImageJ

In ImageJ sind natürlich wichtige Punktoperationen bereits fertig implementiert, sodass man nicht jede Operation wie in Prog. 5.4 selbst programmieren muss. Insbesondere gibt es in ImageJ (a) die Möglichkeit zur Spezifikation von tabellierten Funktionen zur effizienten Ausführung beliebiger Punktoperationen, (b) arithmetisch-logische Standardoperationen für einzelne Bilder und (c) Standardoperationen zur punktweisen Verknüpfung von jeweils zwei Bildern.

5.7.1 Punktoperationen mit Lookup-Tabellen

Punktoperationen können zum Teil komplizierte Berechnungen für jedes einzelne Pixel erfordern, was in großen Bildern zu einem erheblichen

Zeitaufwand führt. Eine Lookup-Tabelle (LUT) realisiert eine diskrete Abbildung (Funktion) von den ursprünglichen K Pixelwerten zu den neuen Pixelwerten, d. h.

$$\mathbf{L} : [0, K-1] \longmapsto [0, K-1] \,. \tag{5.36}$$

Für eine Punktoperation, die durch die Funktion $a' = f(a)$ definiert ist, erhält die Tabelle $\mathbf{L}$ die Werte

$$\mathbf{L}[a] \leftarrow f(a) \quad \text{für } 0 \le a < K \tag{5.37}$$

Die Tabelleneinträge werden also nur einmal für die Werte $a = 0 \ldots K-1$ berechnet. Um die eigentliche Punktoperation im Bild durchzuführen, ist nur ein Nachschlagen in der Tabelle $\mathbf{L}$ erforderlich, also

$$I'(u, v) \leftarrow \mathbf{L}[I(u, v)] \,, \tag{5.38}$$

was wesentlich effizienter ist als jede Funktionsberechnung. ImageJ bietet die Methode

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

für Objekte vom Typ `ImageProcessor`, an die eine Lookup-Tabelle *`lut`* ($\mathbf{L}$) als eindimensionales `int`-Array der Größe K übergeben wird (s. Beispiel in Prog. 5.4). Der Vorteil ist eindeutig – für ein 8-Bit-Grauwertbild z. B. muss in diesem Fall die Abbildungsfunktion (unabhängig von der Bildgröße) nur 256-mal berechnet werden und nicht möglicherweise millionenfach. Die Benutzung von Tabellen für Punktoperationen ist also immer dann sinnvoll, wenn die Anzahl der Bildpixel ($M \times N$) die Anzahl der möglichen Pixelwerte K deutlich übersteigt (was fast immer zutrifft).

5.7.2 Arithmetische Standardoperationen

Die ImageJ-Klasse `ImageProcessor` stellt außerdem eine Reihe von häufig benötigten Operationen als entsprechende Methoden zur Verfügung, von denen die wichtigsten in Tabelle 5.2 zusammengefasst sind. Ein Beispiel für eine Kontrasterhöhung durch Multiplikation mit einem skalaren `double`-Wert zeigt folgendes Beispiel:

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

Das Bild in `ip` wird dabei destruktiv verändert, wobei die Ergebnisse durch „Clamping“ auf den minimalen bzw. maximalen Wert des Wertebereichs begrenzt werden.

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

Tabelle 5.2
ImageJ-Methoden der Klasse `ImageProcessor` für arithmetische Standardoperationen.

5.7.3 Punktoperationen mit mehreren Bildern

Punktoperationen können auch mehr als ein Bild gleichzeitig betreffen, insbesondere, wenn mehrere Bilder durch arithmetische Operationen punktweise verknüpft werden. Zum Beispiel können wir die punktweise *Addition* von zwei Bildern I_1 und I_2 (von gleicher Größe) in ein neues Bild I' ausdrücken als

$$I'(u,v) \;\leftarrow\; I_1(u,v) + I_2(u,v) \tag{5.39}$$

für alle (u,v). Im Allgemeinen kann natürlich jede Funktion $f(a_1, a_2, \ldots, a_n)$ über n Pixelwerte zur punktweisen Verknüpfung von n Bildern verwendet werden, d. h.

$$I'(u,v) \;\leftarrow\; f\big(I_1(u,v), I_2(u,v), \ldots I_n(u,v)\big). \tag{5.40}$$

ImageJ-Methoden für Punktoperationen mit zwei Bildern

ImageJ bietet fertige Methoden zur arithmetischen Verknüpfung von zwei Bildern über die `ImageProcessor`-Methode

`void copyBits(ImageProcessor ip2, int x, int y, int mode)`,

mit der alle Pixel aus dem Quellbild *ip2* an die Position (*x*,*y*) im Zielbild (`this`) kopiert und dabei entsprechend dem vorgegebenen Modus (*mode*) verknüpft werden. Hier ein kurzes Codesegment als Beispiel für die Addition von zwei Bildern:

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

Das Zielbild `ip1` wird durch diese Operation modifiziert, das zweite Bild `ip2` bleibt hingegen unverändert. Die Konstante `ADD` für den Modus ist – neben weiteren arithmetischen Operationen – durch das Interface `Blitter` definiert (Tabelle 5.3). Daneben sind auch (bitweise) logische

Tabelle 5.3
Modus-Konstanten für arithmetische Verknüpfungsoperationen mit der ImageJ-Methode `copyBits()` der Klasse `ImageProcessor` (definiert durch das `Blitter`-Interface).

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

Operationen wie `OR` und `AND` verfügbar (siehe Abschn. C.9.2 im Anhang). Bei arithmetischen Operationen führt die Methode `copyBits()` eine Begrenzung der Ergebnisse (clamping) auf den jeweils zulässigen Wertebereich durch. Bei allen Bildern – mit Ausnahme von Gleitkommabildern – werden die Ergebnisse jedoch *nicht* gerundet, sondern auf ganzzahlige Werte abgeschnitten.

5.7.4 ImageJ-Plugins für mehrere Bilder

Plugins in ImageJ sind primär für die Bearbeitung einzelner Bilder ausgelegt, wobei das aktuelle (vom Benutzer ausgewählte) Bild I als Objekt des Typs `ImageProcessor` (bzw. einer Subklasse davon) als Argument an die `run()`-Methode übergeben wird (s. Abschn. 3.2.3).

Sollen zwei (oder mehr) Bilder $I_1, I_2 \ldots I_k$ miteinander verknüpft werden, dann können die zusätzlichen Bilder $I_2 \ldots I_k$ nicht direkt an die `run()`-Methode des Plugin übergeben werden. Die übliche Vorgangsweise besteht darin, innerhalb des Plugins einen interaktiven Dialog vorzusehen, mit der Benutzer alle weiteren Bilder manuell auswählen kann. Wir zeigen dies nachfolgend anhand eines Beispiel-Plugins, das zwei Bilder transparent überblendet.

Alpha Blending

Alpha Blending ist eine einfache Methode, um zwei Bilder I_{BG} und I_{FG} transparent zu überblenden. Das Hintergrundbild I_{BG} wird von I_{FG} überdeckt, wobei dessen Durchsichtigkeit durch den Transparenzwert α gesteuert wird in der Form

$$I'(u,v) = \alpha \cdot I_{\text{BG}}(u,v) + (1-\alpha) \cdot I_{\text{FG}}(u,v)\,, \tag{5.41}$$

mit $0 \le \alpha \le 1$. Bei $\alpha = 0$ ist I_{FG} undurchsichtig (opak) und deckt dadurch I_{BG} völlig ab. Umgekehrt ist bei $\alpha = 1$ das Bild I_{FG} ganz transparent und nur I_{BG} ist sichtbar. Für dazwischenliegende α-Werte ergibt sich eine gewichtet Summe der entsprechenden Pixelwerte aus I_{BG} und I_{FG}.

Abb. 5.23 zeigt ein Beispiel für die Anwendung von Alpha-Blending mit verschiedenen α-Werten. Die zugehörige Implementierung als ImageJ-Plugin ist in Prog. 5.5–5.6 dargestellt. Die Auswahl des zweiten Bilds

```
1  import ij.IJ;
2  import ij.ImagePlus;
3  import ij.WindowManager;
4  import ij.gui.GenericDialog;
5  import ij.plugin.filter.PlugInFilter;
6  import ij.process.Blitter;
7  import ij.process.ByteBlitter;
8  import ij.process.ByteProcessor;
9  import ij.process.ImageProcessor;
10
11 public class AlphaBlend_ implements PlugInFilter {
12
13   static double alpha = 0.5; // transparency of foreground image
14   ImagePlus fgIm;            // foreground image
15
16   public int setup(String arg, ImagePlus imp) {
17     return DOES_8G;
18   }
19
20   public void run(ImageProcessor bgIp) { // background image
21     if(runDialog()) {
22       ImageProcessor fgIp
23          = fgIm.getProcessor().convertToByte(false);
24       fgIp = fgIp.duplicate();
25       fgIp.multiply(1-alpha);
26       bgIp.multiply(alpha);
27       ByteBlitter blitter
28          = new ByteBlitter((ByteProcessor)bgIp);
29       blitter.copyBits(fgIp, 0, 0, Blitter.ADD);
30     }
31   }
32
33   // continued ...
```

Programm 5.5
ImageJ-Plugin für Alpha Blending (Teil 1). Ein Hintergrundbild wird transparent mit einem auszuwählenden Vordergrundbild kombiniert. Das Plugin wird auf das Hintergrundbild angewandt, beim Start des Plugin muss auch das Vordergrundbild bereits geöffnet sein. Das Hintergrundbild `bgIp` wird der `run()`-Methode übergeben und mit α multipliziert (Zeile 26). Das in Teil 2 ausgewählte Vordergrundbild `fgIP` wird dupliziert (Zeile 24) und mit $(1-\alpha)$ multipliziert (Zeile 25), das Original des Vordergrundbilds bleibt dadurch unverändert. Anschließend werden die so gewichteten Bilder addiert (Zeile 29).

und des α-Werts erfolgt dabei durch eine Instanz der ImageJ-Klasse `GenericDialog`, durch die auf einfache Weise Dialogfenster mit unterschiedlichen Feldern realisiert werden können (s. auch Anhang C.17.3). Ein weiteres Beispiel, in dem aus zwei gegebenen Bildern eine schrittweise Überblendung als Bildfolge (Stack) erzeugt wird, findet sich in Anhang C.14.3.

5.8 Aufgaben

Aufg. 5.1. Erstellen Sie ein geändertes Autokontrast-Plugin, bei dem jeweils $s = 1\%$ aller Pixel an beiden Enden des Wertebereichs gesättigt werden, d. h. auf den Maximalwert 0 bzw. 255 gesetzt werden (Gl. 5.10).

Abbildung 5.23
Beispiel für Alpha Blending. Originalbilder I_{BG} (Hintergrund) und I_{FG} (Vordergrund). Ergebnisse für die Transparenzwerte $\alpha = 0.25, 0.50, 0.75$ und das zugehörige Dialogfenster (s. Implementierung in Prog. 5.5–5.6).



I_{BG}



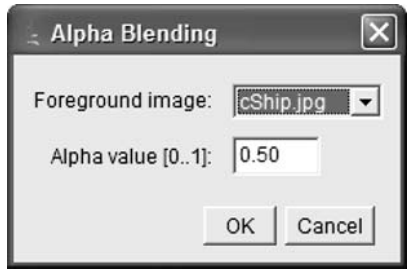
$\alpha = 0.25$



I_{FG}



$\alpha = 0.50$



`GenericDialog`



$\alpha = 0.75$

```
34   // class AlphaBlend_ (continued)
35
36   boolean runDialog() {
37     // get list of open images
38     int[] windowList = WindowManager.getIDList();
39     if(windowList==null){
40       IJ.noImage();
41       return false;
42     }
43     // get image titles
44     String[] windowTitles = new String[windowList.length];
45     for (int i = 0; i < windowList.length; i++) {
46       ImagePlus imp = WindowManager.getImage(windowList[i]);
47       if (imp != null)
48         windowTitles[i] = imp.getShortTitle();
49       else
50         windowTitles[i] = "untitled";
51     }
52     // create dialog and show
53     GenericDialog gd = new GenericDialog("Alpha Blending");
54     gd.addChoice("Foreground image:",
55            windowTitles, windowTitles[0]);
56     gd.addNumericField("Alpha blend [0..1]:", alpha, 2);
57     gd.showDialog();
58     if (gd.wasCanceled())
59       return false;
60     else {
61       int img2Index = gd.getNextChoiceIndex();
62       fgIm = WindowManager.getImage(windowList[img2Index]);
63       alpha = gd.getNextNumber();
64       return true;
65     }
66   }
67 }
```

Programm 5.6
ImageJ-Plugin für Alpha Blending (Teil 2, Dialog). Zur Auswahl des Vordergrundbilds werden zunächst die Liste der geöffneten Bilder (Zeile 38) und die zugehörigen Bildtitel (Zeile 45) ermittelt. Anschließend wird ein Dialog (`GenericDialog`) zusammengestellt und geöffnet, mit dem das zweite Bild (`fgIm`) und der α-Wert (`alpha`) ausgewählt werden (Zeile 53–63). `fgIm` und `alpha` sind Variablen der Klasse `AlphaBlend_` (definiert in Prog. 5.5).

Aufg. 5.2. Ändern Sie das Plugin für den Histogrammausgleich in Prog. 5.2 in der Form, dass es eine Lookup-Table (Abschn. 5.7.1) für die Berechnung verwendet.

Aufg. 5.3. Zeigen Sie formal, dass der Histogrammausgleich (Gl. 5.11) ein bereits gleichverteiltes Bild nicht verändert und dass eine mehrfache Anwendung auf dasselbe Bild nach dem ersten Durchlauf keine weiteren Veränderungen verursacht.

Aufg. 5.4. Zeigen Sie, dass der lineare Histogrammausgleich (Abschn. 5.4) nur ein Sonderfall der Histogrammanpassung (Abschn. 5.5) ist.

Aufg. 5.5. Implementieren Sie (z. B. in Java) die Histogrammanpassung mit einer stückweise linearen Verteilungsfunktion, wie in Abschn.

5.5.3 beschrieben. Modellieren Sie die Verteilungsfunktion selbst als Objektklasse mit den notwendigen Instanzvariablen und realisieren Sie die Funktionen $\mathsf{P}_L(i)$ (Gl. 5.21), $\mathsf{P}_L^{-1}(b)$ (Gl. 5.22) und die Abbildung $f_{\text{hs}}(a)$ (Gl. 5.23) als entsprechende Methoden.

Aufg. 5.6. Bei der gemeinsamen Histogrammanpassung (Abschn. 5.5) mehrerer Bilder kann man entweder ein typisches Exemplar als Referenzbild wählen oder eine „durchschnittliche“ Referenzverteilung verwenden, die aus allen Bildern berechnet wird. Implementieren Sie die zweite Variante und überlegen Sie, welche Vorteile damit verbunden sein könnten.

Aufg. 5.7. Implementieren Sie die modifizierte Gammakorrektur (Gl. 5.32) mit variablen Werten für γ und a_0 als ImageJ-Plugin unter Verwendung einer Lookup-Tabelle analog zu Prog. 5.4.

Aufg. 5.8. Zeigen Sie, dass die modifizierte Gammakorrektur $\bar{f}_{(\gamma,a_0)}(a)$ mit den in Gl. 5.32–5.33 dargestellten Werten für γ, a_0, s und d tatsächlich eine stetige Funktion ergibt.

6

Filter

Die wesentliche Eigenschaft der im vorigen Kapitel behandelten Punktoperationen war, dass der neue Wert eines Bildelements ausschließlich vom ursprünglichen Bildwert an derselben Position abhängig ist. Filter sind Punktoperationen dahingehend ähnlich, dass auch hier eine 1:1-Abbildung der Bildkoordinaten besteht, d. h., dass sich die Geometrie des Bilds nicht ändert. Viele Effekte sind allerdings mit Punktoperationen – egal in welcher Form – allein nicht durchführbar, wie z. B. ein Bild zu schärfen oder zu glätten (Abb. 6.1).



Abbildung 6.1
Mit einer Punktoperation allein ist z. B. die Glättung oder Verwaschung eines Bilds nicht zu erreichen. Wie eine Punktoperation lässt aber auch ein Filter die Bildgeometrie unverändert.

6.1 Was ist ein Filter?

Betrachten wir die Aufgabe des Glättens eines Bilds etwas näher. Bilder sehen vor allem an jenen Stellen scharf aus, wo die Intensität lokal stark ansteigt oder abfällt, also die Unterschiede zu benachbarten Bildelementen groß sind. Umgekehrt empfinden wir Bildstellen als unscharf

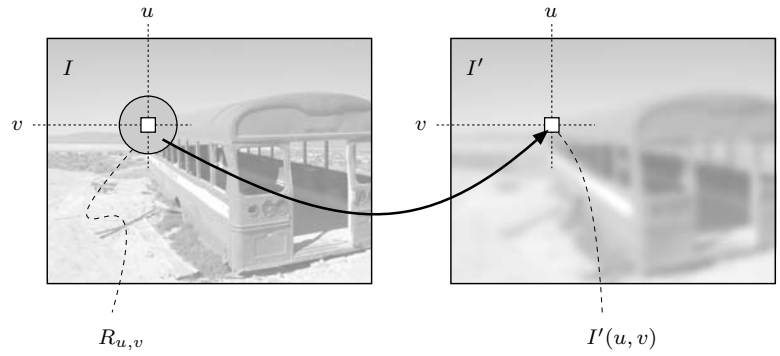


Abbildung 6.2
Prinzip des Filters. Jeder neue Pixelwert $I'(u,v)$ wird aus einer zugehörigen Region $R_{u,v}$ von Pixelwerten im ursprünglichen Bild I berechnet.

oder verschwommen, in denen die Helligkeitsfunktion glatt ist. Eine erste Idee zur Glättung eines Bilds ist daher, jedes Pixel einfach durch den *Durchschnitt* seiner benachbarten Pixel zu ersetzen.

Um also die Pixelwerte im neuen, geglätteten Bild $I'(u,v)$ zu berechnen, verwenden wir jeweils das entsprechende Pixel $I(u,v) = p_0$ plus seine acht Nachbarpixel $p_1, p_2, \ldots p_8$ aus dem ursprünglichen Bild I und berechnen den arithmetischen Durchschnitt dieser neun Werte:

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

In relativen Bildkoordinaten ausgedrückt heißt das

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

was sich kompakter beschreiben lässt in der Form

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

Diese lokale Durchschnittsbildung weist bereits alle Elemente eines typischen Filters auf. Tatsächlich ist es ein Beispiel für eine sehr häufige Art von Filter, ein so genanntes *lineares* Filter. Wie sind jedoch Filter im Allgemeinen definiert? Zunächst unterscheiden sich Filter von Punktoperationen vor allem dadurch, dass das Ergebnis nicht aus einem *einzigen* Ursprungspixel berechnet wird, sondern im Allgemeinen aus einer *Menge* von Pixeln des Originalbilds. Die Koordinaten der Quellpixel sind bezüglich der aktuellen Position (u,v) fix und sie bilden üblicherweise eine zusammenhängende *Region* (Abb. 6.2).

Die *Größe* der Filterregion ist ein wichtiger Parameter eines Filters, denn sie bestimmt, wie viele ursprüngliche Pixel zur Berechnung des neuen Pixelwerts beitragen und damit das räumliche Ausmaß des Filters. Im vorigen Beispiel benutzten wir zur Glättung z. B. eine 3×3-Filterregion, die über der aktuellen Koordinate (u,v) zentriert ist. Mit

größeren Filtern, etwa 5×5, 7×7 oder sogar 21×21 Pixel, würde man daher auch einen stärkeren Glättungseffekt erzielen.

Die *Form* der Filterregion muss dabei nicht quadratisch sein, tatsächlich wäre eine scheibenförmige Region für Glättungsfilter besser geeignet, um in alle Bildrichtungen gleichförmig zu wirken und eine Bevorzugung bestimmter Orientierungen zu vermeiden. Man könnte weiterhin die Quellpixel in der Filterregion mit *Gewichten* versehen, etwa um die näher liegenden Pixel stärker und weiter entfernten Pixel schwächer zu berücksichtigen. Die Filterregion muss auch nicht zusammenhängend sein und muss nicht einmal das ursprüngliche Pixel selbst beinhalten. Sie könnte theoretisch sogar unendlich groß sein.

So viele Optionen sind schön, aber auch verwirrend – wir brauchen eine systematische Methode, um Filter gezielt spezifizieren und einsetzen zu können. Bewährt hat sich die grobe Einteilung in *lineare* und *nichtlineare* Filter auf Basis ihrer mathematischen Eigenschaften. Der einzige Unterschied ist dabei die Form, in der die Pixelwerte innerhalb der Filterregion verknüpft werden: entweder durch einen *linearen* oder durch einen *nichtlinearen* Ausdruck. Im Folgenden betrachten wir beide Klassen von Filtern und zeigen dazu praktische Beispiele.

6.2 Lineare Filter

Lineare Filter werden deshalb so bezeichnet, weil sie die Pixelwerte innerhalb der Filterregion in linearer Form, d. h. durch eine gewichtete Summation verknüpfen. Ein spezielles Beispiel ist die lokale Durchschnittsbildung (Gl. 6.3), bei der alle neun Pixel in der 3×3-Filterregion mit gleichen Gewichten ($1/9$) summiert werden. Mit dem gleichen Mechanismus kann, nur durch Änderung der einzelnen Gewichte, eine Vielzahl verschiedener Filter mit unterschiedlichstem Verhalten definiert werden.

6.2.1 Die Filtermatrix

Bei linearen Filtern werden die Größe und Form der Filterregion, wie auch die zugehörigen Gewichte, allein durch eine Matrix von Filterkoeffizienten spezifiziert, der so genannten „Filtermatrix“ oder „Filtermaske“ $H(i, j)$. Die Größe der Matrix entspricht der Größe der Filterregion und jedes Element in der Matrix $H(i, j)$ definiert das *Gewicht*, mit dem das entsprechende Pixel zu berücksichtigen ist. Das 3×3-Glättungsfilter aus Gl. 6.3 hätte demnach die Filtermatrix

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

da jedes der neun Pixel ein Neuntel seines Werts zum Endergebnis beiträgt.

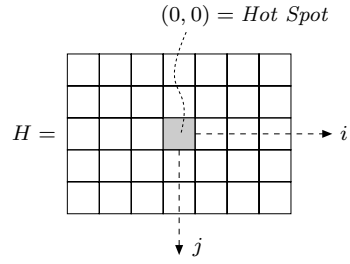


Abbildung 6.3
Filtermatrix und zugehöriges Koordinatensystem.

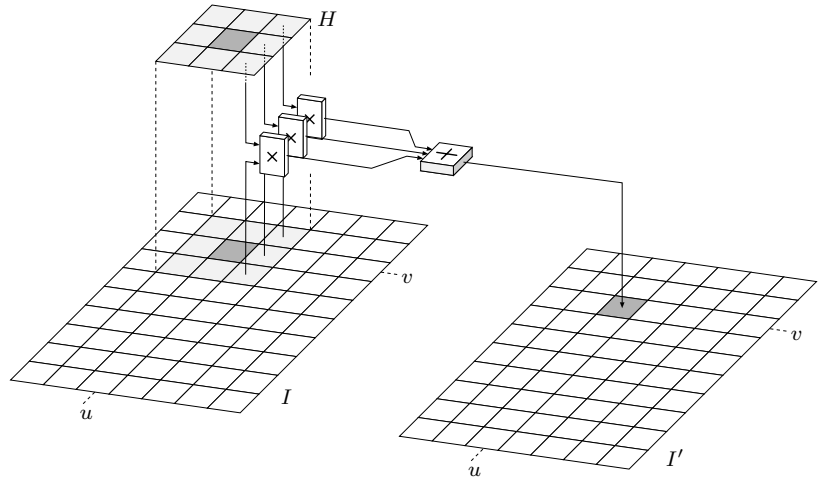


Abbildung 6.4
Lineares Filter. Die Filtermatrix wird mit ihrem Ursprung an der Stelle (u, v) im Bild I positioniert. Die Filterkoeffizienten $H(i, j)$ werden einzeln mit den „darunter“ liegenden Elementen des Bilds $I(u, v)$ multipliziert und die Resultate summiert. Das Ergebnis kommt im neuen Bild an die Stelle $I'(u, v)$.

Im Grunde ist die Filtermatrix $H(i, j)$ – genau wie das Bild selbst – eine diskrete, zweidimensionale, reellwertige Funktion, d. h. $H : \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{R}$. Die Filtermatrix besitzt ihr eigenes Koordinatensystem, wobei der Ursprung – häufig als „hot spot“ bezeichnet – üblicherweise im Zentrum liegt; die Filterkoordinaten sind daher in der Regel positiv *und* negativ (Abb. 6.3). Außerhalb des durch die Matrix definierten Bereichs ist der Wert der Filterfunktion $H(i, j)$ null.

6.2.2 Anwendung des Filters

Bei einem linearen Filter ist das Ergebnis eindeutig und vollständig bestimmt durch die Koeffizienten in der Filtermatrix. Die eigentliche Anwendung auf ein Bild ist – wie in Abb. 6.4 gezeigt – ein einfacher Vorgang: An jeder Bildposition (u, v) werden folgende Schritte ausgeführt:

1. Die Filterfunktion H wird über dem ursprünglichen Bild I positioniert, sodass ihr Koordinatenursprung $H(0, 0)$ auf das aktuelle Bildelement $I(u, v)$ fällt.
2. Als Nächstes werden alle Bildelemente mit dem jeweils darüber liegenden Filterkoeffizienten multipliziert und die Ergebnisse summiert.

3. Die resultierende Summe wird an der entsprechenden Position im Ergebnisbild $I'(u,v)$ gespeichert.

In anderen Worten, alle Pixel des neuen Bilds $I'(u,v)$ werden in folgender Form berechnet:

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

wobei R die Filterregion darstellt. Für ein typisches Filter mit einer Koeffizientenmatrix der Größe 3×3 und zentriertem Ursprung ist das konkret

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

für alle Bildkoordinaten (u,v). Nun, nicht ganz für alle, denn an den Bildrändern, wo die Filterregion über das Bild hinausragt und keine Bildwerte für die zugehörigen Koeffizienten findet, können wir vorerst kein Ergebnis berechnen. Auf das Problem der Randbehandlung kommen wir nachfolgend (in Abschn. 6.5.2) nochmals zurück.

6.2.3 Berechnung der Filteroperation

Nachdem wir seine prinzipielle Funktion (Abb. 6.4) kennen und wissen, dass wir an den Bildrändern etwas vorsichtig sein müssen, wollen wir sofort ein einfaches lineares Filter in ImageJ programmieren. Zuvor sollten wir uns aber noch einen zusätzlichen Aspekt überlegen. Bei einer Punktoperation (z. B. in Prog. 5.1 und Prog. 5.2) hängt das Ergebnis jeweils nur von einem einzigen Originalpixel ab, und es war kein Problem, dass wir das Ergebnis einfach wieder im ursprünglichen Bild gespeichert haben – die Verarbeitung erfolgte „in place“, d. h. ohne zusätzlichen Speicherplatz für die Ergebnisse. Bei Filtern ist das i. Allg. *nicht* möglich, da ein bestimmtes Originalpixel zu mehreren Ergebnissen beiträgt und daher nicht überschrieben werden darf, bevor alle Operationen abgeschlossen sind.

Wir benötigen daher zusätzlichen Speicherplatz für das Ergebnisbild, mit dem wir am Ende – falls erwünscht – das ursprüngliche Bild ersetzen. Die gesamte Filterberechnung kann auf zwei verschiedene Arten realisiert werden (Abb. 6.5):

A. Das Ergebnis der Filteroperation wird zunächst in einem neuen Bild gespeichert, das anschließend in das Originalbild zurückkopiert wird.
B. Das Originalbild wird zuerst in ein Zwischenbild kopiert, das dann als Quelle für die Filteroperation dient. Deren Ergebnis geht direkt in das Originalbild.

Beide Methoden haben denselben Speicherbedarf, daher können wir beliebig wählen. Wir verwenden in den nachfolgenden Beispielen Variante B.

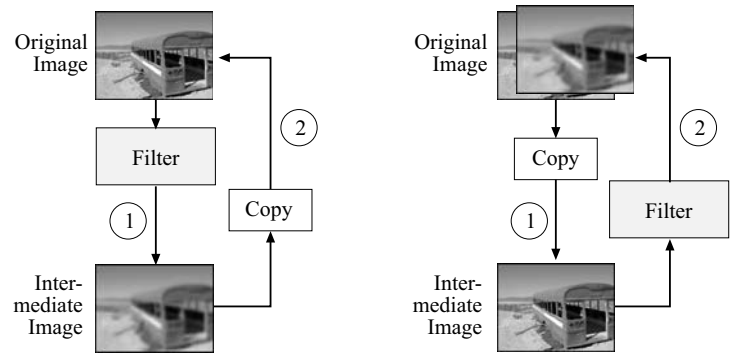


Abbildung 6.5
Praktische Implementierung von Filteroperationen. **Variante A**: Das Filterergebnis wird in einem Zwischenbild (*Intermediate Image*) gespeichert und dieses abschließend in das Originalbild kopiert. **Variante B**: Das Originalbild zuerst in ein Zwischenbild kopiert und dieses danach gefiltert, wobei die Ergebnisse im Originalbild abgelegt werden.

6.2.4 Beispiele für Filter-Plugins

Einfaches 3 × 3-Glättungsfilter („Box"-Filter)

Prog. 6.1 zeigt den Plugin-Code für ein einfaches 3×3-Durchschnittsfilter (Gl. 6.4), das häufig wegen seiner Form als „Box"-Filter bezeichnet wird. Da die Filterkoeffizienten alle gleich ($1/9$) sind, wird keine explizite Filtermatrix benötigt. Da außerdem durch diese Operation keine Ergebnisse außerhalb des Wertebereichs entstehen können, benötigen wir in diesem Fall auch kein *Clamping* (Abschn. 5.1.2).

Obwohl dieses Beispiel ein extrem einfaches Filter implementiert, zeigt es dennoch die allgemeine Struktur eines zweidimensionalen Filterprogramms. Wir benötigen i. Allg. *vier* geschachtelte Schleifen: *zwei*, um das Filter über die Bildkoordinaten (u, v) zu positionieren, und *zwei* weitere über die Koordinaten (i, j) innerhalb der Filterregion. Der erforderliche Rechenaufwand hängt also nicht nur von der Bildgröße, sondern gleichermaßen von der Größe des Filters ab.

Noch ein 3 × 3-Glättungsfilter

Anstelle der konstanten Gewichte wie im vorigen Beispiel verwenden wir nun eine echte Filtermatrix mit unterschiedlichen Koeffizienten. Dazu verwenden wir folgende glockenförmige 3×3-Filterfunktion $H(i, j)$, die das Zentralpixel deutlich stärker gewichtet als die umliegenden Pixel:

$$H(i,j) = \begin{bmatrix} 0.075 & 0.125 & 0.075 \\ 0.125 & \underline{0.200} & 0.125 \\ 0.075 & 0.125 & 0.075 \end{bmatrix} \tag{6.7}$$

Da alle Koeffizienten von $H(i, j)$ positiv sind und ihre Summe eins ergibt (die Matrix ist normalisiert), können auch in diesem Fall keine Ergebnisse außerhalb des ursprünglichen Wertebereichs entstehen. Auch in Prog. 6.2

Programm 6.1
3 × 3-Boxfilter (ImageJ-Plugin). Zunächst (Zeile 10) wird eine Kopie (`copy`) des Originalbilds angelegt, auf das anschließend die eigentliche Filteroperation angewandt wird (Zeile 18). Die Ergebnisse werden wiederum im Originalbild abgelegt (Zeile 23). Alle Randpixel bleiben unverändert.

```
1  import ij.*;
2  import ij.plugin.filter.PlugInFilter;
3  import ij.process.*;
4
5  public class Average3x3_ implements PlugInFilter {
6      ...
7      public void run(ImageProcessor orig) {
8          int w = orig.getWidth();
9          int h = orig.getHeight();
10         ImageProcessor copy = orig.duplicate();
11
12         for (int v=1; v<=h-2; v++) {
13             for (int u=1; u<=w-2; u++) {
14                 //compute filter result for position (u,v)
15                 int sum = 0;
16                 for (int j=-1; j<=1; j++) {
17                     for (int i=-1; i<=1; i++) {
18                         int p = copy.getPixel(u+i,v+j);
19                         sum = sum + p;
20                     }
21                 }
22                 int q = (int) (sum / 9.0);
23                 orig.putPixel(u,v,q);
24             }
25         }
26     }
27 }
```

ist daher kein *Clamping* notwendig und die Programmstruktur ist praktisch identisch zum vorherigen Beispiel. Die Filtermatrix (`filter`) ist ein zweidimensionales Array[1] vom Typ `double`. Jedes Pixel wird mit den entsprechenden Koeffizienten der Filtermatrix multipliziert, die entstehende Summe ist daher ebenfalls vom Typ `double`. Beim Zugriff auf die Koeffizienten ist zu bedenken, dass der Koordinatenursprung der Filtermatrix im Zentrum liegt, d. h. bei einer 3×3-Matrix auf Position $(1, 1)$. Man benötigt daher in diesem Fall einen Offset von 1 für die i- und j-Koordinate (Prog. 6.2, Zeile 20).

6.2.5 Ganzzahlige Koeffizienten

Anstatt mit Gleitkomma-Koeffizienten zu arbeiten, ist es oft einfacher (und meist auch effizienter), ganzzahlige Filterkoeffizienten in Verbindung mit einem gemeinsamen Skalierungsfaktor s zu verwenden, also

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

[1] Vgl. die Anmerkungen dazu in Anhang B.2.4.

Programm 6.2
3×3-Glättungsfilter (ImageJ-Plugin). Die Filtermatrix ist als zweidimensionales `double`-Array definiert (Zeile 5). Der Koordinatenursprung des Filters liegt im Zentrum der Matrix, also an der Array-Koordinate $[1, 1]$, daher der Offset von 1 für die Koordinaten i und j in Zeile 20. In Zeile 24 wird die für die aktuelle Bildposition berechnete Summe gerundet und anschließend (Zeile 25) im Originalbild eingesetzt.

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

mit $s \in \mathbb{R}$ und $H'(i,j) \in \mathbb{Z}$. Falls alle Koeffizienten positiv sind, wie bei Glättungsfiltern üblich, definiert man s reziprok zur Summe der Koeffizienten

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

um die Filtermatrix zu normalisieren. In diesem Fall liegt auch das Ergebnis in jedem Fall innerhalb des ursprünglichen Wertebereichs. Die Filtermatrix aus Gl. 6.7 könnte daher beispielsweise auch in der Form

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

definiert werden, mit dem gemeinsamen Skalierungsfaktor $s = \frac{1}{40} = 0.025$. Eine solche Skalierung wird etwa auch für die Filteroperation in Prog. 6.3 verwendet.

In *Adobe Photoshop* sind u. a. lineare Filter unter der Bezeichnung „Custom Filter“ in dieser Form realisiert. Auch hier werden Filter mit ganzzahligen Koeffizienten und einem gemeinsamen Skalierungsfaktor

Scale (der dem Kehrwert von s entspricht) spezifiziert. Zusätzlich kann ein konstanter Offset-Wert angegeben werden, etwa um negative Ergebnisse (aufgrund negativer Koeffizienten) in den sichtbaren Intensitätsbereich zu verschieben (Abb. 6.6). Das 5×5-Custom-Filter in Photoshop entspricht daher insgesamt folgender Operation (vgl. Gl. 6.6):

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

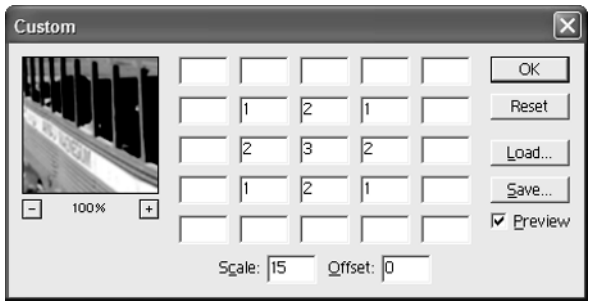


Abbildung 6.6
Das „Custom Filter“ in *Adobe Photoshop* realisiert lineare Filter bis zur Größe von 5×5. Der Koordinatenursprung des Filters („hot spot“) wird im Zentrum (Wert 3) angenommen, die leeren Felder entsprechen Koeffizienten mit Wert null. Neben den (ganzzahligen) Filterkoeffizienten und dem Skalierungsfaktor Scale kann ein Offset-Wert angegeben werden.

6.2.6 Filter beliebiger Größe

Während wir bisher nur mit 3×3-Filtermatrizen gearbeitet haben, wie sie in der Praxis auch häufig verwendet werden, können Filter grundsätzlich von beliebiger Größe sein. Nehmen wir dazu den (üblichen) Fall an, dass die Filtermatrix $H(i,j)$ zentriert ist und ungerade Seitenlängen mit $(2K+1)$ Spalten und $(2L+1)$ Zeilen aufweist, wobei $K, L \geq 0$. Wenn das Bild M Spalten und N Zeilen hat, also

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

dann kann das Filterergebnis im Bereich jener Bildkoordinaten (u', v') berechnet werden, für die gilt

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

(siehe Abb. 6.7). Die Implementierung beliebig großer Filter ist anhand eines 7×5 großen Glättungsfilters in Prog. 6.3 gezeigt, das aus Prog. 6.2 adaptiert ist. In diesem Beispiel sind ganzzahlige Filterkoeffizienten (Zeile 6) in Verbindung mit einem gemeinsamen Skalierungsfaktor `s` verwendet, wie bereits oben besprochen. Der „hot spot“ des Filters wird wie üblich im Zentrum angenommen und der Laufbereich aller Schleifenvariablen ist von den Dimensionen der Filtermatrix abhängig. Sicherheitshalber ist (in Zeile 32, 33) auch ein *Clamping* der Ergebniswerte vorgesehen.

Programm 6.3
ImageJ-Plugin für ein lineares Filter mit ganzzahliger Filteratrix der Größe $(2K+1) \times (2L+1)$, die als zentriert angenommen wird. Die Summenvariable `sum` ist in diesem Fall ganzzahlig (`int`), die Ergebnisse werden erst am Schluss mit dem konstanten Faktor `s` skaliert und gerundet (Zeile 31). Die Bildränder bleiben hier unbehandelt.

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

6.2.7 Arten von linearen Filtern

Die Funktion eines linearen Filters ist ausschließlich durch seine Filtermatrix spezifiziert, und da deren Koeffizienten beliebige Werte annehmen können, gibt es grundsätzlich unendlich viele verschiedene lineare Filter. Wofür kann man aber diese Filter einsetzen und welche Filter sind für eine bestimmte Aufgabe am besten geeignet? Zwei in der Praxis wichtige Klassen von Filtern sind *Glättungsfilter* und *Differenzfilter* (Abb. 6.8).

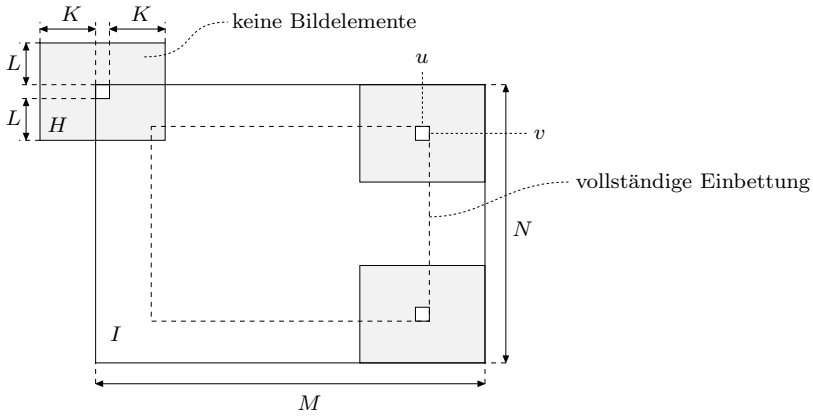
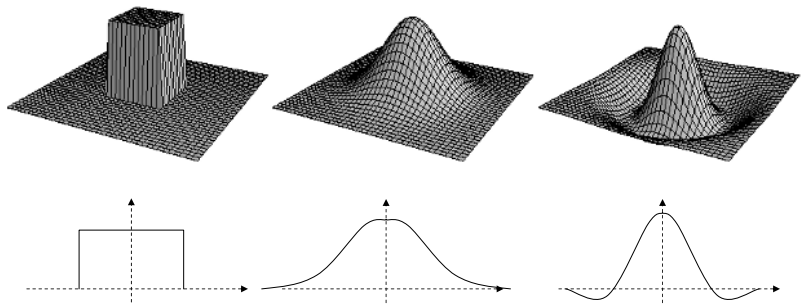


Abbildung 6.7
Randproblem bei Filtern. Die Filteroperation kann problemlos nur dort angewandt werden, wo die Filtermatrix H der Größe $(2K+1) \times (2L+1)$ vollständig in das Bild I eingebettet ist.



0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(a)

0	1	2	1	0
1	3	5	3	1
2	5	9	5	2
1	3	5	3	1
0	1	2	1	0

(b)

0	0	-1	0	0
0	-1	-2	-1	0
-1	-2	16	-2	-1
0	-1	-2	-1	0
0	0	-1	0	0

(c)

Abbildung 6.8
Typische Beispiele für lineare Filter. Darstellung als 3D-Plot (oben), Profil (Mitte) und Näherung als diskrete Filtermatrix (unten). Das „Box"-Filter (a) ist ebenso wie das Gauß-Filter (b) ein *Glättungsfilter* mit ausschließlich positiven Filterkoeffizienten. Das sog. „Laplace" (oder *Mexican Hat*) Filter (c) ist im Gegensatz dazu ein *Differenzfilter*. Es bildet eine gewichtete Differenz zwischen dem zentralen Pixel und den umliegenden Werten und reagiert daher besonders auf lokale Intensitätsspitzen.

Glättungsfilter

Alle bisher betrachteten Filter hatten irgendeine Art von Glättung zur Folge. Tatsächlich ist jedes lineare Filter, das nur positive Filterkoeffizienten aufweist, in gewissem Sinn ein Glättungsfilter, denn ein solches Filter berechnet nur einen gewichteten Durchschnitt über die Filterregion.

Box-Filter

Das einfachste aller Glättungsfilter, dessen 3D-Form an eine Schachtel erinnert (Abb. 6.8 (a)), ist ein alter Bekannter. Das Box-Filter ist aber aufgrund seiner scharf abfallenden Ränder und dem damit zusammenhängenden Frequenzverhalten (s. Kap. 14) ein eher schlechtes Glättungsfilter. Intuitiv erscheint es auch wenig plausibel, allen betroffenen Bildelementen gleiches Gewicht zuzuordnen und nicht das Zentrum

gegenüber den Rändern stärker zu gewichten. Glättungsfilter sollten auch weitgehend „isotrop“, d. h. nach allen Richtungen hin gleichförmig arbeiten, und auch das ist beim Box-Filter aufgrund seiner quadratischen Form nicht der Fall.

Gauß-Filter

Die Filtermatrix dieses Glättungsfilters (Abb. 6.8 (b)) entspricht einer diskreten, zweidimensionalen Gauß-Funktion

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

wobei die Standardabweichung σ den „Radius“ der glockenförmigen Funktion definiert (Abb. 6.8 (b)). Das mittlere Bildelement erhält das maximale Gewicht (1.0 skaliert auf den ganzzahligen Wert 9), die Werte der übrigen Koeffizienten nehmen mit steigender Entfernung zur Mitte kontinuierlich ab. Die Gauß-Funktion ist isotrop, vorausgesetzt die Filtermatrix ist groß genug für eine ausreichende Näherung (mindestens 5×5). Das Gauß-Filter weist ein „gutmütiges“ Frequenzverhalten auf und ist diesbezüglich dem Box-Filter eindeutig überlegen.

Differenzfilter

Wenn einzelne Filterkoeffizienten negativ sind, kann man die Filteroperation als Differenz von zwei Summen interpretieren: die gewichtete Summe aller Bildelemente mit zugehörigen *positiven* Koeffizienten abzüglich der gewichteten Summe von Bildelementen mit *negativen* Koeffizienten innerhalb der Filterregion R:

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

Dabei bezeichnet R^+ den Teil des Filters mit positiven Koeffizienten $H(i,j) > 0$ und R^- den Teil mit negativen Koeffizienten $H(i,j) < 0$. Das 5×5-Laplace-Filter (Abb. 6.8 (c)) bildet z. B. die Differenz zwischen einem zentralen Bildwert (mit Gewicht 16) und der Summe über 12 umliegende Bildwerte (mit Gewichten von -1 und -2). Die übrigen 12 Bildwerte haben zugehörige Koeffizienten mit dem Wert null und bleiben daher unberücksichtigt.

Während bei einer Durchschnittsbildung örtliche Intensitätsunterschiede geglättet werden, kann man bei einer Differenzbildung das genaue Gegenteil erwarten: Örtliche Unterschiede werden verstärkt. Wichtige Einsatzbereiche von Differenzfiltern sind daher vor allem das Verstärken von Kanten und Konturen (Abschn. 7.2) sowie das Schärfen von Bildern (Abschn. 7.6).

6.3 Formale Eigenschaften linearer Filter

Wir haben uns im vorigen Abschnitt dem Konzept des Filters in recht lockerer Weise angenähert, um rasch ein gutes Verständnis dafür zu bekommen, wie Filter aufgebaut und wofür sie nützlich sind. Obwohl das bisherige für den praktischen Einsatz oft völlig ausreichend ist, mag man über die scheinbar doch etwas eingeschränkten Möglichkeiten linearer Filter möglicherweise enttäuscht sein, obwohl sie doch so viele Gestaltungsmöglichkeiten bieten.

Die Bedeutung der linearen Filter – und vielleicht auch ihre formale Eleganz – werden oft erst deutlich, wenn man auch dem darunter liegenden theoretischen Fundament etwas mehr Augenmerk schenkt. Es mag daher vielleicht überraschen, dass wir den wichtigen Begriff der „Faltung“[2] bisher überhaupt nicht erwähnt haben. Das soll nun nachgeholt werden.

6.3.1 Lineare Faltung

Die Operation eines linearen Filters, wie wir sie im vorherigen Abschnitt definiert hatten, ist keine Erfindung der digitalen Bildverarbeitung, sondern ist in der Mathematik seit langem bekannt. Die Operation heißt „lineare Faltung“ (*linear convolution*) und verknüpft zwei Funktionen gleicher Dimensionalität, kontinuierlich oder diskret. Für diskrete, zweidimensionale Funktionen I und H ist die Faltungsoperation definiert als

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

oder abgekürzt

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

Das sieht fast genauso aus wie Gl. 6.6, mit Ausnahme der unterschiedlichen Wertebereiche für die Summenvariablen i, j und der umgekehrten Vorzeichen der Koordinaten in $I(u-i, v-j)$. Der erste Unterschied ist schnell erklärt: Da die Filterkoeffizienten außerhalb der Filtermatrix $H(i,j)$ – die auch als „Faltungskern“ bezeichnet wird – als null angenommen werden, sind die Positionen außerhalb der Matrix für die Summation nicht relevant. Bezüglich der Koordinaten sehen wir durch eine kleine Umformung von Gl. 6.14, dass gilt

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

[2] Ein von manchen Studierenden völlig zu Unrecht gefürchtetes Mysterium.

Das wiederum ist genau das lineare Filter aus Gl. 6.6, außer, dass die Filterfunktion $H(-i,-j)$ horizontal und vertikal gespiegelt (bzw. um 180° gedreht) ist. Um genau zu sein, beschreibt die Operation in Gl. 6.6 eigentlich eine lineare Korrelation (*correlation*), was aber identisch ist zur linearen Faltung (*convolution*), abgesehen von der gespiegelten Filtermatrix.[3]

Die mathematische Operation hinter *allen* linearen Filtern ist also die lineare Faltung ($*$) und das Ergebnis ist vollständig und ausschließlich durch den Faltungskern (die Filtermatrix) H definiert. Um das zu illustrieren, beschreibt man die Faltung häufig als „Black Box"-Operation (Abb. 6.9).

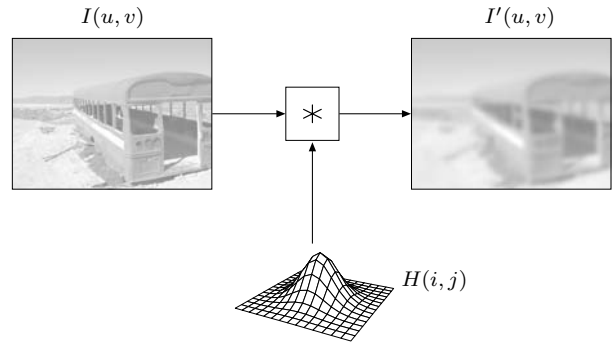


Abbildung 6.9
Faltung als „Black Box"-Operation. Das Originalbild I durchläuft eine lineare Faltungsoperation ($*$) mit dem Faltungskern H und erzeugt das Ergebnis I'.

6.3.2 Eigenschaften der linearen Faltung

Die Bedeutung der linearen Faltung basiert auf ihren einfachen mathematischen Eigenschaften und ihren vielfältigen Anwendungen und Erscheinungsformen. Wie wir in Kap. 13 zeigen, besteht sogar eine nahtlose Beziehung zur Fourieranalyse und den zugehörigen Methoden im Frequenzbereich. Zunächst aber einige Eigenschaften der linearen Faltung im „Ortsraum".

Kommutativität

Die lineare Faltungsoperation ist *kommutativ*, d. h.

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

man erhält also dasselbe Ergebnis, wenn man das Bild und die Filterfunktion vertauscht. Es macht daher keinen Unterschied, ob wir das

[3] Das gilt natürlich auch im eindimensionalen Fall. Die lineare Korrelation wird u. a. häufig zum Vergleichen von Bildmustern verwendet (s. Abschn. 17.1).

Bild I mit dem Filterkern H falten oder umgekehrt – beide Funktionen können gleichberechtigt und austauschbar dieselbe Rolle einnehmen.

Linearität

Lineare Filter tragen diese Bezeichnung aufgrund der Linearitätseigenschaften der Faltung. Wenn wir z. B. ein Bild mit einer skalaren Konstante a multiplizieren, dann multipliziert sich auch das Faltungsergebnis mit demselben Faktor, d. h.

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

Weiterhin, wenn wir zwei Bilder Pixel-weise addieren und nachfolgend die Summe einem Filter unterziehen, dann würde dasselbe Ergebnis erzielt, wenn wir beide Bilder vorher getrennt filtern und erst danach addieren:

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

Es mag in diesem Zusammenhang überraschen, dass die Addition einer Konstanten b zu einem Bild das Ergebnis eines linearen Filters *nicht* im gleichen Ausmaß erhöht, also

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

und ist also nicht Teil der Linearitätseigenschaft. Die Linearität von Filtern ist vor allem ein wichtiges theoretisches Konzept. In der Praxis sind jedoch viele Filteroperationen in ihrer Linearität eingeschränkt, z. B. durch Rundungsfehler oder durch die nichtlineare Begrenzung von Ergebniswerten (Clamping).

Assoziativität

Die lineare Faltung ist assoziativ, d. h., die Reihenfolge von nacheinander ausgeführten Filteroperationen ist ohne Belang:

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

Man kann daher bei mehreren aufeinander folgenden Filtern die Reihenfolge beliebig verändern und auch mehrere Filter beliebig zu neuen Filtern zusammenfassen.

6.3.3 Separierbarkeit von Filtern

Eine unmittelbare Konsequenz aus Gl. 6.21 ist die Möglichkeit, einen Filterkern H als Faltungsprodukt von zwei oder mehr – und möglicherweise kleineren – Faltungskernen $H_1, H_2, \ldots, H_n$ zu beschreiben, sodass $H = H_1 * H_2 * \ldots * H_n$. In diesem Fall kann die Filteroperation $I * H$, mit einem „großen“ Filter H, als Folge „kleinerer“ Filteroperationen

$$I * H = I * (H_1 * H_2 * \ldots * H_n) = (\ldots((I * H_1) * H_2) * \ldots * H_n) \tag{6.22}$$

durchgeführt werden, was i. Allg. erheblich weniger Rechenaufwand erfordert.

x/y-Separierbarkeit

Die Möglichkeit, ein *zwei*dimensionales Filter H in zwei *ein*dimensionale Filter H_x und H_y zu zerteilen, ist eine besonders häufige und wichtige Form der Separierbarkeit. Angenommen wir hätten zwei eindimensionale Filter H_x, H_y mit

$$H_x = \begin{bmatrix} 1 & 1 & \underline{1} & 1 & 1 \end{bmatrix} \qquad \text{bzw.} \qquad H_y = \begin{bmatrix} 1 \\ 1 \\ \underline{1} \\ 1 \end{bmatrix}. \tag{6.23}$$

Wenn wir diese beiden Filter nacheinander auf das Bild I anwenden,

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

dann ist das (nach Gl. 6.22) gleichbedeutend mit der Anwendung eines kombinierten Filters H_{xy}, wobei

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

Das zweidimensionale Box-Filter H_{xy} kann also in zwei eindimensionale Filter geteilt werden Die gesamte Faltungsoperation benötigt für das kombinierte Filter H_{xy} $3 \cdot 5 = 15$ Rechenoperationen pro Bildelement, im separierten Fall $5 + 3 = 8$ Operationen, also deutlich weniger. Im Allgemeinen wächst die Zahl der Operationen quadratisch mit der Seitenlänge des Filters, im Fall der x/y-Separierbarkeit jedoch nur linear. Beim Einsatz größerer Filter ist diese Möglichkeit daher von emminenter Bedeutung (s. auch Abschn. 6.5.1).

Separierbare Gauß-Filter

Im Allgemeinen ist ein zweidimensionales Filter x/y-separierbar, wenn die Filterfunktion $H(i, j)$, wie im obigen Beispiel, als (äußeres) Produkt zweier eindimensionaler Funktionen beschrieben werden kann, d. h.

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

denn in diesem Fall entspricht die Funktion auch dem Faltungsprodukt $H_{x,y} = H_x * H_y$. Ein prominentes Beispiel dafür ist die zweidimensionale Gauß-Funktion $G_\sigma(x, y)$ (Gl. 6.12), die als Produkt von eindimensionalen Funktionen

$$G_\sigma(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}} = e^{-\frac{x^2}{2\sigma^2}} \cdot e^{-\frac{y^2}{2\sigma^2}} = g_\sigma(x) \cdot g_\sigma(y) \tag{6.26}$$

darstellbar ist. Offensichtlich kann daher ein zweidimensionales Gauß-Filter $H^{G,\sigma}$ als ein Paar eindimensionaler Gauß-Filter $H_x^{G,\sigma}, H_y^{G,\sigma}$ in der Form

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

realisiert werden.

Die Gauß-Funktion fällt relativ langsam ab und ein diskretes Gauß-Filter sollte eine minimale Ausdehnung von ca. $\pm 2.5\,\sigma$ aufweisen, um Fehler durch abgeschnittene Koeffizienten zu vermeiden. Für $\sigma = 10$ benötigt man beispielsweise ein Filter mit der Mindestgröße 51×51, wobei das x/y-separierbare Filter in diesem Fall ca. 50-mal schneller läuft als ein entsprechendes 2D-Filter. Relativ große Gauß-Filter werden z. B. beim „Unsharp Mask"-Filter (Abschn. 7.6.2) benötigt.

6.3.4 Impulsantwort eines Filters

Es gibt für die lineare Faltungsoperation auch ein „neutrales Element" – das natürlich auch eine Funktion ist –, und zwar die *Impuls-* oder *Dirac*-Funktion $\delta()$, für die gilt

$$I * \delta = I. \tag{6.27}$$

Im zweidimensionalen, diskreten Fall ist die Impulsfunktion definiert als

$$\delta(i,j) = \begin{cases} 1 & \text{für } i = j = 0 \\ 0 & \text{sonst.} \end{cases} \tag{6.28}$$

Als Bild betrachtet ist die Dirac-Funktion ein einziger heller Punkt (mit dem Wert 1) am Koordinatenursprung umgeben von einer unendlichen, schwarzen Fläche (Abb. 6.10). Wenn wir die Dirac-Funktion als Filter-

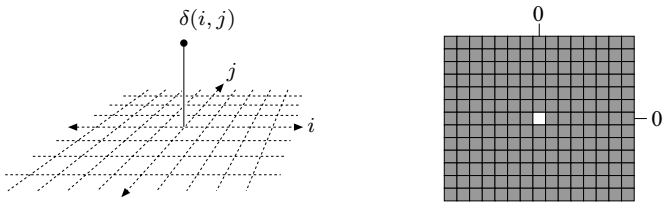


Abbildung 6.10
Zweidimensionale, diskrete Impuls- oder Dirac-Funktion.

funktion verwenden und damit eine lineare Faltung durchführen, dann ergibt sich (gemäß Gl. 6.27) wieder das ursprüngliche, unveränderte Bild (Abb. 6.11).

Der umgekehrte Fall ist allerdings interessanter: Wir verwenden die Dirac-Funktion als *Input* und wenden ein beliebiges lineares Filter H an. Was passiert? Aufgrund der Kommutativität der Faltung (Gl. 6.17) gilt

$$H * \delta = \delta * H = H \tag{6.29}$$

und damit erhalten wir als Ergebnis der Filteroperation mit δ wieder das Filter H (Abb. 6.12)! Man schickt also einen Impuls in ein Filter hinein

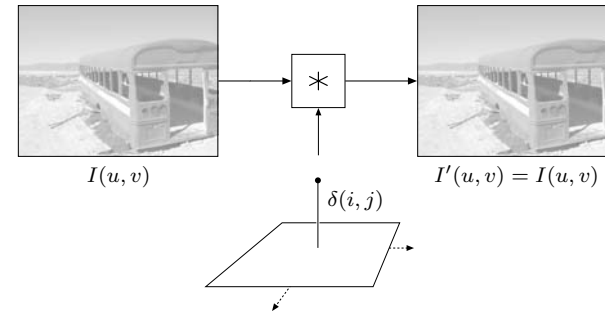


Abbildung 6.11
Das Ergebnis einer linearen Faltung mit der Impulsfunktion δ ist das ursprüngliche Bild I.

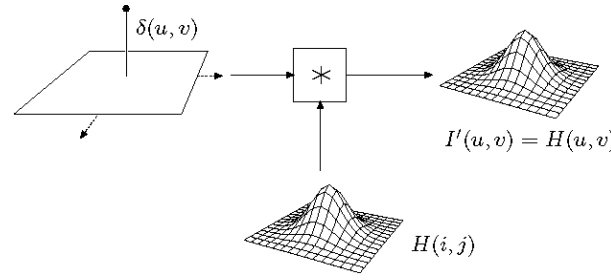


Abbildung 6.12
Die Impulsfunktion δ als Input eines linearen Filters liefert die Filterfunktion H selbst als Ergebnis

und erhält als Ergebnis die Filterfunktion selbst – wozu kann das gut sein? Das macht vor allem dann Sinn, wenn man die Eigenschaften des Filters zunächst nicht kennt oder seine Parameter messen möchte. Unter der Annahme, dass es sich um ein lineares Filter handelt, erhält man durch einen einzigen Impuls sofort sämtliche Informationen über das Filter. Man nennt das Ergebnis die „Impulsantwort“ des Filters H. Diese Methode wird u. a. auch für die Messung des Filterverhaltens von optischen Systemen verwendet und die dort als Impulsantwort enstehende Lichtverteilung wird traditionell als „point spread function“ (PSF) bezeichnet.

6.4 Nichtlineare Filter

Lineare Filter haben beim Einsatz zum Glätten und Entfernen von Störungen einen gravierenden Nachteil: Auch beabsichtigte Bildstrukturen, wie Punkte, Kanten und Linien, werden dabei ebenfalls verwischt und die gesamte Bildqualität damit reduziert (Abb. 6.13). Das ist mit linearen Filtern nicht zu vermeiden und ihre Einsatzmöglichkeiten sind für solche Zwecke daher beschränkt. Wir versuchen im Folgenden zu klären, ob sich dieses Problem mit anderen, also nichtlinearen Filtern besser lösen lässt.

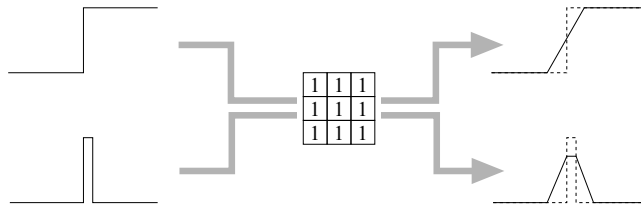


Abbildung 6.13
Lineare Glättungsfilter verwischen auch beabsichtigte Bildstrukturen. Sprungkanten (oben) oder dünne Linien (unten) werden verbreitert und gleichzeitig ihr Kontrast reduziert.

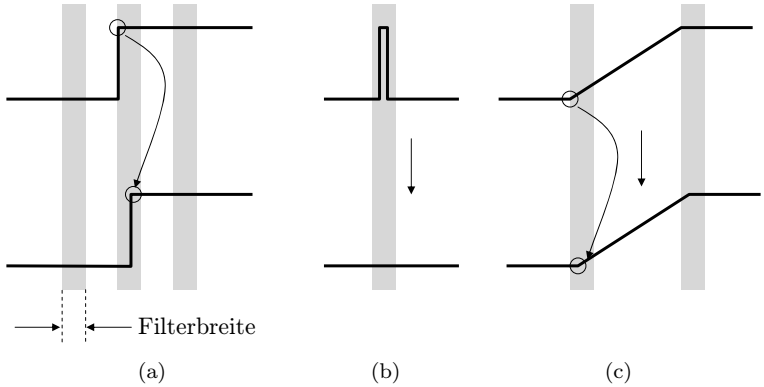


Abbildung 6.14
Auswirkungen eines Minimum-Filters auf verschiedene Formen lokaler Bildstrukturen. Die ursprüngliche Bildfunktion (Profil) ist oben, das Filterergebnis unten. Der vertikale Balken zeigt die Breite des Filters. Die Sprungkante (a) und die Rampe (c) werden um eine halbe Filterbreite nach rechts verschoben, der enge Puls (b) wird gänzlich eliminiert.

6.4.1 Minimum- und Maximum-Filter

Nichtlineare Filter berechnen, wie alle bisherigen Filter auch, das Ergebnis an einer bestimmten Bildposition (u, v) aus einer entsprechenden Region R im ursprünglichen Bild. Die einfachsten nichtlinearen Filter sind Minimum- und Maximum-Filter, die folgendermaßen definiert sind:

$$I'(u,v) \leftarrow \min\{I(u+i, v+j) \mid (i,j) \in R\} = \min\big(R_{u,v}\big) \qquad (6.30)$$

$$I'(u,v) \leftarrow \max\{I(u+i, v+j) \mid (i,j) \in R\} = \max\big(R_{u,v}\big) \qquad (6.31)$$

Dabei bezeichnet $R_{u,v}$ die Region der Bildwerte, die an der aktuellen Position (u, v) von der Filterregion (meist ein Quadrat der Größe 3×3) überdeckt werden. Abb. 6.14 illustriert die Auswirkungen eines Min-Filters auf verschiedene lokale Bildstrukturen.

Abb. 6.15 zeigt die Anwendung von 3×3-Min- und -Max-Filtern auf ein Grauwertbild, das künstlich mit „Salt-and-Pepper"-Störungen versehen wurde, das sind zufällig platzierte weiße und schwarze Punkte. Das *Min-Filter* entfernt die weißen (*Salt*) Punkte, denn ein einzelnes weißes Pixel wird innerhalb der 3×3-Filterregion $\mathbf{R}$ immer von kleineren Werten umgeben, von denen einer den Minimalwert liefert. Gleichzeitig werden durch das Min-Filter aber andere dunkle Strukturen räumlich erweitert.

Das *Max-Filter* hat natürlich genau den gegenteiligen Effekt. Ein einzelnes weißes Pixel ist immer der lokale Maximalwert, sobald es in-



(a) (b) (c)

Abbildung 6.15. Minimum- und Maximum-Filter. Das Originalbild (a) ist mit „Salt-and-Pepper"-Rauschen gestört. Das 3×3-Minimum-Filter (b) eliminiert die weißen Punkte und verbreitert dunkle Stellen. Das Maximum-Filter (c) hat den gegenteiligen Effekt.

nerhalb der Filterregion R auftaucht. Weiße Punkte breiten sich daher in einer entsprechenden Umgebung aus und helle Bildstrukturen werden erweitert, während nunmehr die schwarzen Punkte (*Pepper*) verschwinden.

6.4.2 Medianfilter

Es ist natürlich unmöglich, ein Filter zu bauen, das alle Störungen entfernt und gleichzeitig die wichtigen Strukturen intakt lässt, denn kein Filter kann unterscheiden, welche Strukturen für den Betrachter wichtig sind und welche nicht. Das Medianfilter ist aber ein guter Schritt in diese Richtung.

Das Medianfilter ersetzt jedes Bildelement durch den *Median* der Pixelwerte innerhalb der Filterregion R, also

$$I'(u,v) \leftarrow \text{median}\big(R_{u,v}\big), \tag{6.32}$$

wobei der Median von $2K+1$ Pixelwerten p_i definiert ist als

$$\text{median}\,(p_0, p_1, \ldots, p_K, \ldots, p_{2K}) = p_K, \tag{6.33}$$

also der mittlere Wert, wenn die Folge $(p_0, \ldots, p_{2K})$ nach der Größe ihrer Elemente sortiert ist ($p_i \le p_{i+1}$). Abb. 6.16 demonstriert die Berechnung des Medianfilters für eine Filterregion der Größe 3×3.

Da rechteckige Filter fast immer *ungerade* Seitenlängen aufweisen, ist auch die Zahl der Filterelemente in diesen Fällen ungerade. Falls die Anzahl der Elemente jedoch *gerade* sein sollte, dann ist der Median der

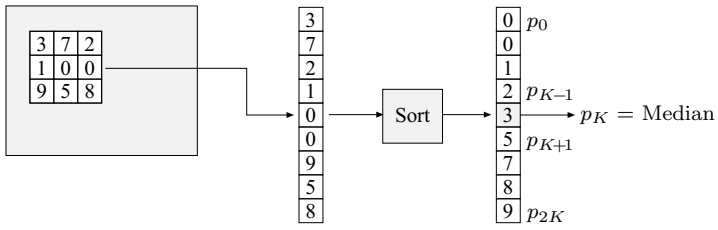


Abbildung 6.16
Berechnung des Median. Die 9 Bildwerte innerhalb der 3×3-Filterregion werden sortiert, der sich daraus ergebende mittlere Wert ist der Medianwert.

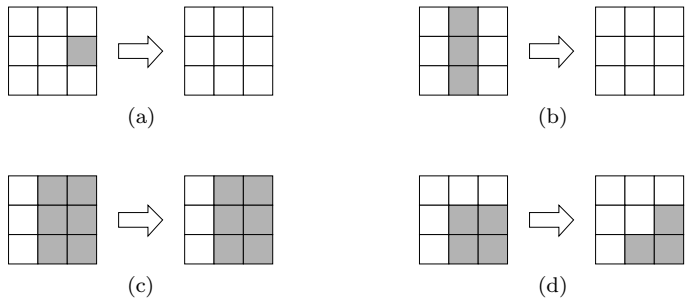


Abbildung 6.17
Auswirkung eines 3×3-Medianfilter auf Bildstrukturen in 2D. Ein einzelner Puls wird eliminiert (a), genauso wie eine 1 Pixel dünne horizontale oder vertikale Linie (b). Die Sprungkante (c) bleibt unverändert, eine Ecke (d) wird abgerundet.

entsprechenden, sortierten Folge $(p_0, \ldots, p_{2N-1})$ definiert als das arithmetische Mittel der beiden mittleren Werte, also $(p_{N-1} + p_N)/2$.

Das Medianfilter erzeugt also bei ungerader Größe der Filterregion keine neuen Intensitätswerte, sondern reproduziert nur bereits bestehende Pixelwerte.

Abbildung 6.17 illustriert die Auswirkungen eines 3×3-Medianfilters auf zweidimensionale Bildstrukturen. Sehr kleine Strukturen (kleiner als die Hälfte des Filters) werden eliminiert, alle anderen Strukturen bleiben weitgehend unverändert. Abb. 6.18 vergleicht schließlich anhand eines Grauwertbilds das Medianfilter mit einem linearen Glättungsfilter. Den Quellcode des ImageJ-Plugin für das Medianfilter, dessen grundsätzliche Struktur identisch zum linearen 3×3-Filter in Prog. 6.2 ist, zeigt Prog. 6.4.

6.4.3 Das gewichtete Medianfilter

Der Median ist ein Rangordnungsmaß und in gewissem Sinn bestimmt die „Mehrheit“ der beteiligten Werte das Ergebnis. Ein einzelner, außergewöhnlich hoher oder niedriger Wert, ein so genannter „Outlier“, kann das Ergebnis nicht dramatisch beeinflussen, sondern nur um maximal einen der anderen Werte nach oben oder unten verschieben (das Maß ist „robust“). Beim gewöhnlichen Medianfilter haben alle Pixel in der Filterregion dasselbe Gewicht und man könnte sich überlegen, ob man nicht etwa die Pixel im Zentrum stärker gewichten sollte als die am Rand.

Das gewichtete Medianfilter ist eine Variante des Medianfilters, das einzelnen Positionen innerhalb der Filterregion unterschiedliche Gewichte zuordnet. Ähnlich zur Koeffizientenmatrix H bei einem linearen



(a) (b) (c)

Abbildung 6.18. Vergleich zwischen linearem Glättungsfilter und Medianfilter. Das Originalbild (a) ist durch „Salt and Pepper"-Rauschen gestört. Das lineare 3×3-Box-Filter (b) reduziert zwar die Helligkeitsspitzen, führt aber auch zu allgemeiner Unschärfe im Bild. Das Medianfilter (c) eliminiert die Störspitzen sehr effektiv und lässt die übrigen Bildstrukturen dabei weitgehend intakt. Allerdings führt es auch zu örtlichen Flecken gleichmäßiger Intensität.

Filter werden die Gewichte in Form einer *Gewichtsmatrix* $W(i,j) \in \mathbb{N}$ spezifiziert. Zur Berechnung der Ergebnisse wird der Bildwert $I(u+i, v+j)$ entsprechend dem zugehörigen Gewicht $W(i,j)$-mal vervielfacht in die zu sortierenden Folge eingesetzt. Die so entstehende vergrößerte Folge

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

Aus dieser Folge Q wird anschließend durch Sortieren der Medianwert berechnet, genauso wie beim gewöhnlichen Medianfilter. Abb. 6.19 zeigt die Berechnung des gewichteten Medianfilters anhand eines Beispiels mit einer 3×3-Gewichtsmatrix

$$W(i,j) = \begin{bmatrix} 1 & 2 & 1 \\ 2 & \underline{3} & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{6.34}$$

und einer Wertefolge Q der Länge 15 entsprechend der Summe der Gewichte.

Diese Methode kann natürlich auch dazu verwendet werden, gewöhnliche Medianfilter mit nicht quadratischer oder rechteckiger Form zu spezifizieren, zum Beispiel ein kreuzförmiges Medianfilter durch die Gewichtsmatrix

$$W^+(i,j) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & \underline{1} & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{6.35}$$

Programm 6.4
3×3-Medianfilter (ImageJ-Plugin). Für die Sortierung der Pixelwerte in der Filterregion wird ein `int`-Array `P` definiert (Zeile 14), das für jede Filterposition (u, v) neuerlich befüllt wird. Die eigentliche Sortierung erfolgt durch die Java Utility-Methode `Arrays.sort` in Zeile 28. Der in der Mitte des Arrays (`P[4]`) liegende Medianwert wird als Ergebnis im Originalbild abgelegt (Zeile 29).

```
1  import ij.*;
2  import ij.plugin.filter.PlugInFilter;
3  import ij.process.*;
4  import java.util.Arrays;
5
6  public class Median3x3_ implements PlugInFilter {
7
8      public void run(ImageProcessor orig) {
9          int w = orig.getWidth();
10         int h = orig.getHeight();
11         ImageProcessor copy = orig.duplicate();
12
13         //vector to hold pixels from 3x3 neighborhood
14         int[] P = new int[9];
15
16         for (int v=1; v<=h-2; v++) {
17             for (int u=1; u<=w-2; u++) {
18
19                 //fill the pixel vector P for filter position (u,v)
20                 int k = 0;
21                 for (int j=-1; j<=1; j++) {
22                     for (int i=-1; i<=1; i++) {
23                         P[k] = copy.getPixel(u+i,v+j);
24                         k++;
25                     }
26                 }
27                 //sort the pixel vector and take center element
28                 Arrays.sort(P);
29                 orig.putPixel(u,v,P[4]);
30             }
31         }
32     }
33 }
```

Nicht jede Anordnung von Gewichten ist allerdings sinnvoll. Würde man etwa dem Zentralpixel die Hälfte der Gewichte („Stimmen“) oder mehr zuteilen, dann wäre natürlich das Ergebnis ausschließlich vom jeweiligen Wert dieses Pixels bestimmt.

6.4.4 Andere nichtlineare Filter

Medianfilter und gewichtetes Medianfilter sind nur zwei Beispiele für nichtlineare Filter, die häufig verwendet werden und einfach zu beschreiben sind. Da „nichtlinear“ auf alles zutrifft, was eben nicht linear ist, gibt es natürlich eine Vielzahl von Filter, die diese Eigenschaft erfüllen, wie z. B. die morphologischen Filter für Binär- und Grauwertbilder, die wir in Kap. 10 behandeln. Andere nichtlineare Filter (wie z. B. der Corner-Detektor in Kap. 8) sind oft in Form von Algorithmen definiert und entziehen sich einer kompakten Beschreibung.

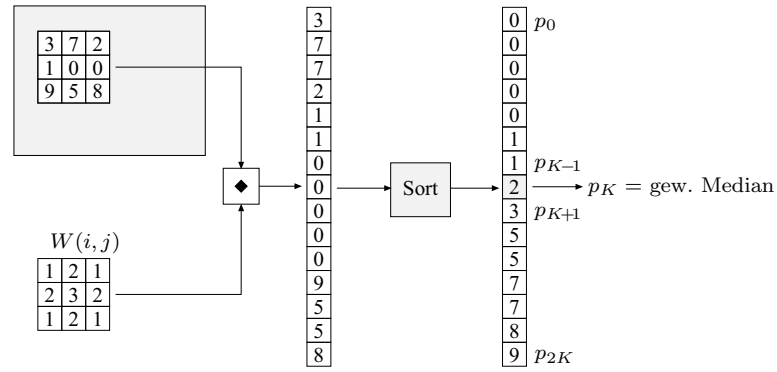


Abbildung 6.19
Berechnung des gewichteten Medianfilters. Jeder Bildwert wird, abhängig vom entsprechenden Gewicht in der Gewichtsmatrix $W(i, j)$, mehrfach in die zu sortierende Wertefolge eingesetzt. Zum Beispiel wird der Wert (0) des zentralen Pixels dreimal eingesetzt (weil $W(0, 0) = 3$), der Wert 7 zweimal. Anschließend wird innerhalb der erweiterten Folge durch Sortieren der Medianwert (2) ermittelt.

Im Unterschied zu linearen Filtern gibt es bei nichtlinearen Filtern generell keine „starke“ Theorie, die etwa den Zusammenhang zwischen der Addition von Bildern und dem Medianfilter in ähnlicher Form beschreiben würde wie bei einer linearen Faltung (Gl. 6.19). Meistens sind auch keine allgemeinen Aussagen über die Auswirkungen nichtlinearer Filter im Frequenzbereich möglich.

6.5 Implementierung von Filtern

6.5.1 Effizienz von Filterprogrammen

Die Berechnung von linearen Filtern ist in der Regel eine aufwendige Angelegenheit, speziell mit großen Bildern oder großen Filtern (im schlimmsten Fall beides). Für ein Bild der Größe $M \times N$ und einer Filtermatrix der Größe $(2K + 1) \times (2L + 1)$ benötigt eine direkte Implementierung

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

Operationen, d. h. Multiplikationen und Additionen. Wie wir in Abschn. 6.3.3 gesehen haben, sind substanzielle Einsparungen möglich, wenn Filter in kleinere, möglichst eindimensionale Filter separierbar sind.

Die Programmierbeispiele in diesem Kapitel wurden bewusst einfach und verständlich gehalten. Daher ist auch keine der bisher gezeigten Lösungen besonders effizient und es bleiben zahlreiche Möglichkeiten zur Verbesserung. Insbesondere ist es wichtig, alle dort nicht unbedingt benötigten Anweisungen aus den Schleifenkernen herauszunehmen und „möglichst weit nach außen“ zu bringen. Speziell trifft das für „teure“ Anweisungen wie Methodenaufrufe zu, die besonders in Java unverhältnismäßig viel Rechenzeit verbrauchen können.

Wir haben mit den ImageJ-Methoden `getPixel()` und `putPixel()` in den Beispielen auch bewusst die einfachste Art des Zugriffs auf Bildelemente benutzt, aber eben auch die langsamste. Wesentlich schneller ist der direkte Zugriff auf Pixel als Array-Elemente (s. Anhang 3.6).

6.5.2 Behandlung der Bildränder

Wie bereits in Abschn. 6.2.2 kurz angeführt wurde, gibt es bei der Berechnung von Filtern häufig Probleme an den Bildrändern. Wann immer die Filterregion gerade so positioniert ist, dass zumindest einer der Filterkoeffizienten außerhalb des Bildbereichs zu liegen kommt und damit kein zugehöriges Bildpixel hat, kann das entsprechende Filterergebnis eigentlich nicht berechnet werden. Es gibt zwar keine mathematisch korrekte Lösung dafür, aber doch einige praktische Methoden für den Umgang mit den verbleibenden Randbereichen:

1. Anstatt der Filterergebnisse im Randbereich einen konstanten Wert (z. B. „schwarz“) einsetzen.
2. Die ursprünglichen (ungefilterten) Bildwerte beibehalten.
3. Auch die Randbereiche berechnen unter der Annahme, dass ...
   a) die Pixel außerhalb des Bilds einen konstanten Wert (z. B. „schwarz“ oder „grau“) aufweisen.
   b) sich die Randpixel außerhalb des Bilds fortsetzen.
   c) sich das Bild in beiden Richtungen (horizontal und vertikal) zyklisch wiederholt.

Die letzten drei Methoden sind in Abb. 6.20 dargestellt. Keine der Methoden ist perfekt und die Wahl hängt wie so oft von der Art des Filters und der spezifischen Anwendung ab. Methode (1) ist zwar die einfachste, aber in vielen Fällen nicht akzeptabel, weil sich der sichtbare Bildbereich durch die Filteroperation verkleinert. Das gilt grundsätzlich auch für Methode (2). Methode (3a) kann bei größeren Filtern die Ergebnisse zu den Rändern hin stark verändern. Demgegenüber verursacht Methode (3b) relativ geringe Verfälschungen und wird daher meist bevorzugt. Methode (3c) erscheint zwar zunächst völlig unsinnig, allerdings sollte man bedenken, dass auch etwa in der Fourieranalyse (Kap. 13) die beteiligten Funktionen implizit als periodisch betrachtet werden.

Welche der Methoden man auch immer einsetzt, die Bildränder benötigen fast immer spezielle Vorkehrungen, die mitunter mehr Aufwand erfordern können als die eigentliche Verarbeitung im Inneren des Bilds.

6.6 Filteroperationen in ImageJ

ImageJ bietet eine Reihe fertig implementierter Filteroperationen, die allerdings von verschiedenen Autoren stammen und daher auch teilweise unterschiedlich umgesetzt sind. Die meisten dieser Operationen können auch manuell über das Process-Menü in ImageJ angewandt werden.

6.6.1 Lineare Filter

Filter auf Basis der linearen Faltung (*convolution*) sind in ImageJ bereits fertig in der Klasse ij.plugin.filter.Convolver implementiert. `Convolver`



(a) (b) (c)

Abbildung 6.20. Methoden zur Filterberechnung an den Bildrändern. Die Annahme ist, dass die (nicht vorhandenen) Pixel außerhalb des ursprünglichen Bilds einen konstanten Wert aufweisen (a), den Wert der nächstliegenden Randpixel aufweisen (b) oder sich zyklisch von der gegenüberliegenden Seite her wiederholen (c).

ist selbst eine Plugin-Klasse, die allerdings außer `run()` noch weitere Methoden zur Verfügung stellt. Am einfachsten zu zeigen ist das anhand eines Beispiels mit einem 8-Bit-Grauwertbild und der Filtermatrix (aus Gl. 6.7):

$$H(i,j) = \begin{bmatrix} 0.075 & 0.125 & 0.075 \\ 0.125 & \underline{0.200} & 0.125 \\ 0.075 & 0.125 & 0.075 \end{bmatrix}.$$

In der folgenden `run()`-Methode eines ImageJ-Plugins wird zunächst die Filtermatrix als eindimensionales `float`-Array definiert (man beachte die Form der `float`-Konstanten „`0.075f`“ usw.), dann wird in Zeile 8 ein neues `Convolver`-Objekt angelegt:

```
1  import ij.plugin.filter.Convolver;
2  ...
3  public void run(ImageProcessor I) {
4    float[] H = {
5        0.075f, 0.125f, 0.075f,
6        0.125f, 0.200f, 0.125f,
7        0.075f, 0.125f, 0.075f };
8    Convolver cv = new Convolver();
9    cv.setNormalize(false);   // turn off filter normalization
10   cv.convolve(I, H, 3, 3); // do the filter operation
11 }
```

Die Methode `convolve()` in Zeile 10 benötigt für die Filteroperation neben dem Bild `I` und der Filtermatrix `H` selbst auch deren Breite und Höhe (weil `H` ein eindimensionales Array ist). Das Bild `I` wird durch die Filteroperation modifiziert.

In diesem Fall hätte man auch die nicht normalisierte ganzzahlige Filtermatrix in Gl. 6.10 verwenden können, denn `convolve()` normalisiert das übergebene Filter automatisch (nach `cv.setNormalize(true)`).

6.6.2 Gauß-Filter

In der ImageJ-Klasse ij.plugin.filter.GaussianBlur ist ein einfaches Gauß-Filter implementiert, dessen Radius (σ) frei spezifiziert werden kann. Dieses Gauß-Filter ist natürlich mit separierten Filterkernen implementiert (s. Abschn. 6.3.3).[4] Hier ist ein Beispiel für dessen Anwendung:

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

6.6.3 Nichtlineare Filter

Ein kleiner Baukasten von nichtlinearen Filtern ist in der ImageJ-Klasse ij.plugin.filter.RankFilters indexMedianfilter implementiert, insbesondere Minimum-, Maximum- und gewöhnliches Medianfilter. Die Filterregion ist jeweils (annähernd) kreisförmig mit frei wählbarem Radius. Hier ein entsprechendes Anwendungsbeispiel:

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

6.7 Aufgaben

Aufg. 6.1. Erklären Sie, warum das „Custom Filter“ in *Adobe Photoshop* (Abb. 6.6) streng genommen kein lineares Filter ist.

Aufg. 6.2. Berechnen Sie den maximalen und minimalen Ergebniswert eines linearen Filters mit nachfolgender Filtermatrix $H(i, j)$ bei Anwendung auf ein 8-Bit-Grauwertbild (mit Pixelwerten im Bereich $[0, 255]$). Gehen Sie zunächst davon aus, dass dabei kein *Clamping* der Resultate erfolgt.

[4] Zur Implementierung in ImageJ ist anzumerken, dass die in der Methode `blur()` generierten Filterkerne relativ zum angegebenen Radius zu klein dimensioniert werden, und es dadurch zu erheblichen Fehlern kommt.

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

Aufg. 6.3. Erweitern Sie das Plugin in Prog. 6.3, sodass auch die Bildränder bearbeitet werden. Benutzen Sie dazu die Methode, bei der die ursprünglichen Randpixel außerhalb des Bilds fortgesetzt werden, wie in Abschn. 6.5.2 beschrieben.

Aufg. 6.4. Zeige, dass ein gewöhnliches Box-Filter nicht isotrop ist, d. h., nicht in alle Richtungen gleichmäßig glättet.

Aufg. 6.5. Implementieren Sie ein gewichtetes Medianfilter (Abschn. 6.4.3) als ImageJ-Plugin. Spezifizieren Sie die Gewichte als konstantes, zweidimensionales `int`-Array. Testen Sie das Filter und vergleichen Sie es mit einem gewöhnlichen Medianfilter. Erklären Sie, warum etwa die folgende Gewichtsmatrix keinen Sinn macht:

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

Aufg. 6.6. Überprüfen Sie die Eigenschaften des Dirac-Impulses in Bezug auf lineare Filter (Gl. 6.29). Erzeugen Sie dazu ein schwarzes Bild mit einem weißen Punkt im Zentrum und verwenden Sie dieses als Dirac-Signal. Stellen Sie fest, ob lineare Filter tatsächlich die Filtermatrix H als Impulsantwort liefern.

Aufg. 6.7. Überlegen Sie, welche Auswirkung ein lineares Filter mit folgender Filtermatrix hat:

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

Aufg. 6.8. Konstruieren Sie ein lineares Filter, das eine horizontale Verwischung über 7 Pixel während der Bildaufnahme modelliert.

Aufg. 6.9. Erstellen Sie ein eigenes ImageJ-Plugin für ein Gauß'sches Glättungsfilter. Die Größe des Filters (Radius σ) soll beliebig einstellbar sein. Erstellen Sie die zugehörige Filtermatrix dynamisch mit einer Größe von mindestens 5σ in beiden Richtungen. Nutzen Sie die x/y-Separierbarkeit der Gaußfunktion (Abschn. 6.3.3).

Aufg. 6.10. Das Laplace- oder eigentlich „Laplacian of Gaussian" (LoG) Filter (s. auch Abb. 6.8) basiert auf der Summe der zweiten Ableitungen (Laplace-Operator) der Gauß-Funktion. Es ist definiert als

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

Implementieren Sie analog zu Aufg. 6.9 ein LoG-Filter mit beliebiger Größe (σ). Überlegen Sie, ob dieses Filter separierbar ist (s. Abschn. 6.3.3).

7

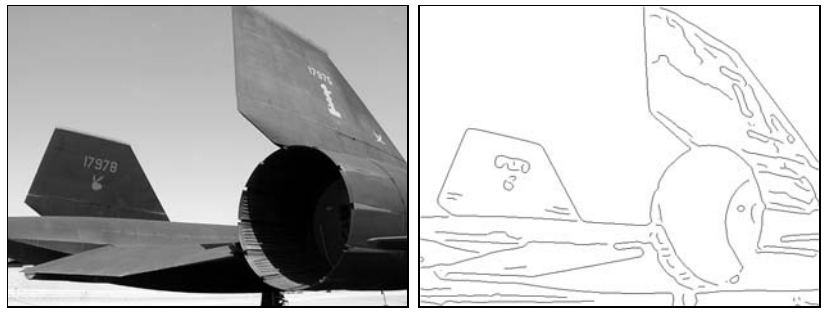
Kanten und Konturen

Markante „Ereignisse“ in einem Bild, wie Kanten und Konturen, die durch lokale Veränderungen der Intensität oder Farbe zustande kommen, sind für die visuelle Wahrnehmung und Interpretation von Bildern von höchster Bedeutung. Die subjektive „Schärfe“ eines Bilds steht in direktem Zusammenhang mit der Ausgeprägtheit der darin enthaltenen Diskontinuitäten und der Deutlichkeit seiner Strukturen. Unser menschliches Auge scheint derart viel Gewicht auf kantenförmige Strukturen und Konturen zu legen, dass oft nur einzelne Striche in Karikaturen oder Illustrationen für die eindeutige Beschreibung der Inhalte genügen. Aus diesem Grund sind Kanten und Konturen auch für die Bildverarbeitung ein traditionell sehr wichtige Thema. In diesem Kapitel betrachten wir zunächst einfache Methoden zur Lokalisierung von Kanten und anschließend das verwandte Problem des Schärfens von Bildern.

7.1 Wie entsteht eine Kante?

Kanten und Konturen spielen eine dominante Rolle im menschlichen Sehen und vermutlich auch in vielen anderen biologischen Sehsystemen. Kanten sind nicht nur auffällig, sondern oft ist es auch möglich, komplette Figuren aus wenigen dominanten Linien zu rekonstruieren (Abb. 7.1). Wodurch entstehen also Kanten und wie ist es technisch möglich, sie in Bildern zu lokalisieren?

Kanten könnte man grob als jene Orte im Bild beschreiben, an denen sich die Intensität auf kleinem Raum und entlang einer ausgeprägten Richtung stark ändert. Je stärker sich die Intensität ändert, umso stärker ist auch der Hinweis auf eine Kante an der entsprechenden Stelle. Die Stärke der Änderung bezogen auf die Distanz ist aber nichts anderes als die erste Ableitung, und diese ist daher auch ein wichtiger Ansatz zur Bestimmung der Kantenstärke.



(a) (b)

Abbildung 7.1
Kanten spielen eine dominante Rolle im menschlichen Sehen. Originalbild (a) und Kantenbild (b).

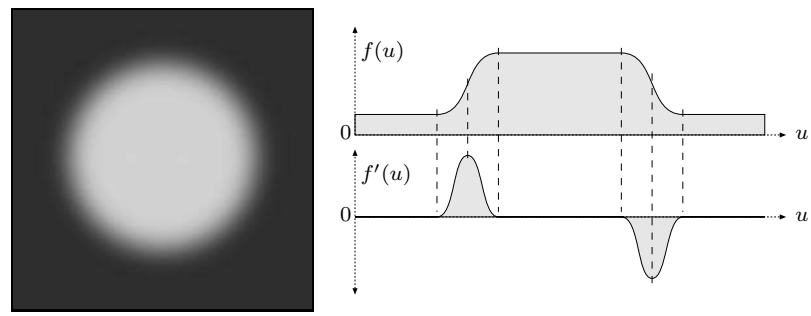
7.2 Gradienten-basierte Kantendetektion

Der Einfachheit halber betrachten wir die Situation zunächst in nur einer Dimension und nehmen als Beispiel an, dass ein Bild eine helle Region im Zentrum enthält, umgeben von einem dunklen Hintergrund (Abb. 7.2 (a)).

Das Intensitäts- oder Grauwertprofil entlang einer Bildzeile könnte etwa wie in Abb. 7.2 (b) aussehen. Bezeichnen wir diese eindimensionale Funktion mit $f(u)$ und berechnen wir ihre erste Ableitung von links nach rechts,

$$f'(u) = \frac{df}{du}(u), \tag{7.1}$$

so ergibt sich ein positiver Ausschlag überall dort, wo die Intensität ansteigt, und ein negativer Ausschlag, wo der Wert der Funktion abnimmt (Abb. 7.2 (c)). Allerdings ist für diskrete Funktionen wie $f(u)$ die Ableitung nicht definiert und wir benötigen daher eine Methode, um diese zu schätzen. Abbildung 7.3 gibt uns dafür eine Idee, zunächst weiterhin für den eindimensionalen Fall. Bekanntlich können wir die erste Ableitung einer kontinuierlichen Funktion an einer Stelle x als Anstieg der Tangente an dieser Stelle interpretieren. Bei einer diskreten Funktion können wir den Anstieg der Tangente an der Stelle u einfach dadurch schätzen, dass wir eine Gerade durch die Abtastwerte an den benachbarten Stellen $u-1$



(a) (b)

Abbildung 7.2
Erste Ableitung im eindimensionalen Fall. Originalbild (a), horizontales Intensitätsprofil $f(u)$ entlang der mittleren Bildzeile und Ergebnis der ersten Ableitung $f'(u)$ (b).

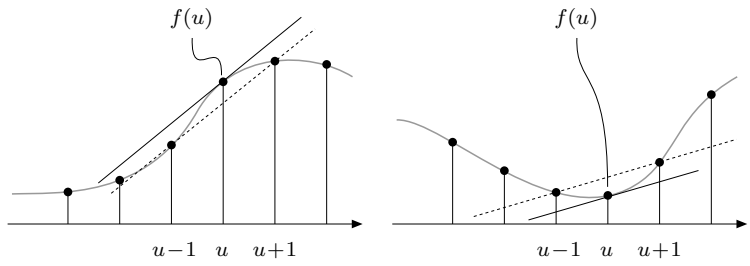


Abbildung 7.3
Schätzung der ersten Ableitung bei einer diskreten Funktion. Der Anstieg der Geraden durch die beiden Nachbarpunkte $f(u-1)$ und $f(u+1)$ dient als Schätzung für den Anstieg der Tangente in $f(u)$. Die Schätzung genügt in den meisten Fällen als grobe Näherung.

und $u+1$ legen und deren Anstieg berechnen:

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

Den gleichen Vorgang könnten wir natürlich auch in der vertikalen Richtung, also entlang der Bildspalten, durchführen.

7.2.1 Partielle Ableitung und Gradient

Bei der Ableitung einer *mehr*dimensionalen Funktion entlang einer der Koordinatenrichtungen spricht man von einer *partiellen* Ableitung, z. B.

$$\frac{\partial I}{\partial u}(u,v) \quad \text{und} \quad \frac{\partial I}{\partial v}(u,v) \tag{7.3}$$

für die partiellen Ableitungen der Bildfunktion $I(u,v)$ entlang der u- bzw. v-Koordinate.[1] Den Vektor

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

bezeichnet man als *Gradientenvektor* oder kurz *Gradient* der Funktion I an der Stelle (u,v). Der Betrag des Gradienten

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

ist invariant unter Bilddrehungen und damit auch unabhängig von der Orientierung der Bildstrukturen. Diese Eigenschaft ist für die richtungsunabhängige (isotrope) Lokalisierung von Kanten wichtig und daher ist $|\nabla I|$ auch die Grundlage vieler praktischer Kantendetektoren.

[1] ∂ ist der partielle Ableitungsoperator oder „del“-Operator.

7.2.2 Ableitungsfilter

Die Komponenten des Gradienten (Gl. 7.4) sind also nichts anderes als die ersten Ableitungen der Zeilen (Gl. 7.1) bzw. der Spalten (in vertikaler Richtung). Die in Gl. 7.2 skizzierte Schätzung der Ableitung in horizontaler Richtung können wir in einfacher Weise als lineares Filter (s. Abschn. 6.2) mit der Koeffizientenmatrix

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

realisieren, wobei der Koeffizient -0.5 das Bildelement $I(u-1,v)$ betrifft und $+0.5$ das Bildelement $I(u+1,v)$. Das mittlere Pixel $I(u,v)$ wird mit dem Wert null gewichtet und daher ignoriert. In gleicher Weise berechnet man die vertikale Richtungskomponente des Gradienten mit dem linearen Filter

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

Abb. 7.4 zeigt die Anwendung der Gradientenfilter aus Gl. 7.6 und Gl. 7.7 auf ein synthetisches Testbild.

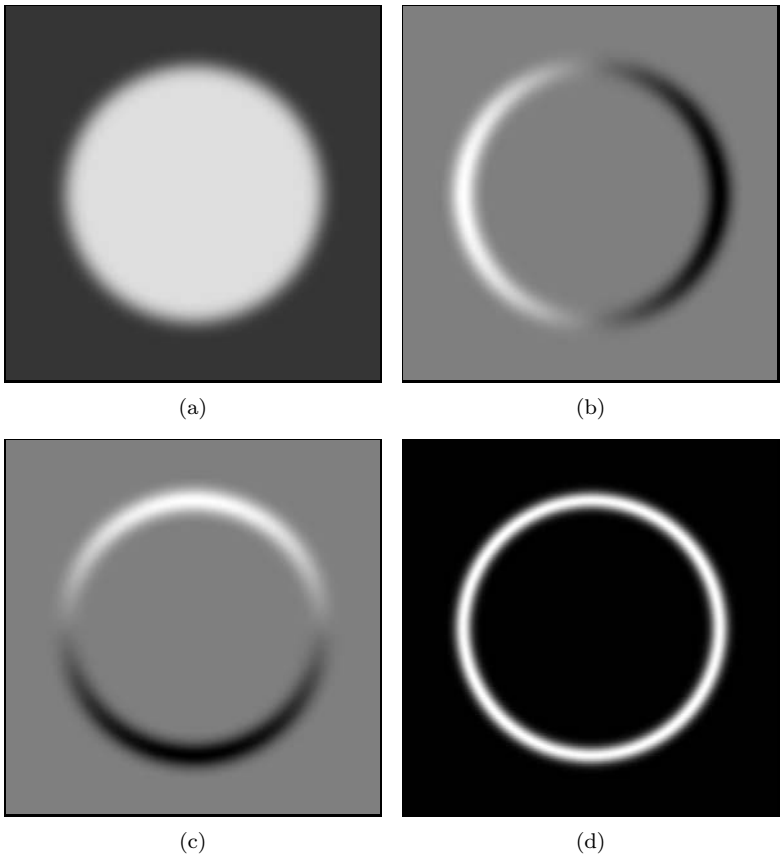
Die Richtungsabhängigkeit der Filterantwort ist klar zu erkennen. Das horizontale Gradientenfilter H_x^D reagiert am stärksten auf rapide Änderungen in der horizontalen Richtung, also auf *vertikale* Kanten; analog dazu reagiert das vertikale Gradientenfilter H_y^D besonders stark auf *horizontale* Kanten. In flachen Bildregionen ist die Filterantwort null (grau dargestellt), da sich dort die Ergebnisse des positiven und des negativen Filterkoeffizienten jeweils aufheben.

7.3 Filter zur Kantendetektion

Die Schätzung des lokalen Gradienten der Bildfunktion ist Grundlage der meisten Operatoren für die Kantendetektion. Sie unterscheiden sich praktisch nur durch die für die Schätzung der Richtungskomponenten eingesetzten Filter sowie die Art, in der diese Komponenten zum Endergebnis zusammengefügt werden. In vielen Fällen ist man aber nicht nur an der *Stärke* eines Kantenpunkts interessiert, sondern auch an der lokalen *Richtung* der zugehörigen Kante. Da beide Informationen im Gradienten enthalten sind, können sie auf relativ einfache Weise berechnet werden. Im Folgenden sind einige bekannte Kantenoperatoren zusammengestellt, die nicht nur in der Praxis häufig Verwendung finden, sondern teilweise auch historisch interessant sind.

7.3.1 Prewitt- und Sobel-Operator

Zwei klassische Kantenoperatoren sind die Verfahren von Prewitt und Sobel [20], die einander sehr ähnlich sind und sich nur durch geringfügig abweichende Filter unterscheiden.



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

Abbildung 7.4
Partielle erste Ableitungen. Synthetische Bildfunktion I (a), erste Ableitung in horizontaler Richtung $\partial I/\partial u$ (b) und in vertikaler Richtung $\partial I/\partial v$ (c). Betrag des Gradienten $|\nabla I|$ (d). In (b,c) sind maximal negative Werte schwarz, maximal positive Werte weiß und Nullwerte grau dargestellt.

Filter

Beide Operatoren verwenden Filter, die sich über jeweils drei Zeilen bzw. Spalten erstrecken, um der Rauschanfälligkeit des einfachen Gradientenoperators (Gl. 7.6 bzw. Gl. 7.7) entgegenzuwirken. Der **Prewitt**-Operator verwendet die Filter

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

die offensichtlich über jeweils drei benachbarte Zeilen bzw. Spalten mitteln. Zeigt man diese Filter in der separierten Form

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

dann wird klar, dass jeweils eine einfache (Box-)Glättung über drei Zeilen bzw. drei Spalten erfolgt, bevor der gewöhnliche Gradient (Gl. 7.6,

7.7) berechnet wird. Aufgrund der Kommutativität der Faltung (Abschn. 6.3.1) ist das genauso auch umgekehrt möglich, d. h., eine Glättung *nach* der Berechnung der Ableitung.

Die Filter des **Sobel**-Operators sind fast identisch, geben allerdings durch eine etwas andere Glättung mehr Gewicht auf die zentrale Zeile bzw. Spalte:

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

Die Ergebnisse der Filter ergeben daher nach einer entsprechenden Skalierung eine Schätzung des lokalen Bildgradienten:

$$\nabla I(u,v) \approx \frac{1}{6} \begin{bmatrix} H_x^P * I \\ H_y^P * I \end{bmatrix} \qquad \text{für den Prewitt-Operator,} \tag{7.11}$$

$$\nabla I(u,v) \approx \frac{1}{8} \begin{bmatrix} H_x^S * I \\ H_y^S * I \end{bmatrix} \qquad \text{für den Sobel-Operator.} \tag{7.12}$$

Kantenstärke und -richtung

Wir bezeichnen, unabhängig ob Prewitt- oder Sobel-Filter, die skalierten Filterergebnisse (Gradientenwerte) mit

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

Die Kanten*stärke* $E(u,v)$ wird in beiden Fällen als Betrag des Gradienten

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

definiert und die lokale Kanten*richtung* (d. h. ein Winkel) als[2]

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

Der Ablauf der gesamten Kantendetektion ist nochmals in Abb. 7.5 zusammengefasst. Zunächst wird das ursprüngliche Bild I mit den beiden Gradientenfiltern H_x und H_y gefiltert und nachfolgend aus den Filterergebnissen die Kantenstärke E und die Kantenrichtung Φ berechnet.

Die Schätzung der Kantenrichtung ist allerdings mit dem Prewitt-Operator und auch mit dem ursprünglichen Sobel-Operator relativ ungenau. In [48, S. 353] werden für den Sobel-Operator daher verbesserte Filter vorgeschlagen, die den Winkelfehler minimieren:

[2] Siehe Anhang B.1.6 zur Berechnung der inversen Tangensfunktion $\tan^{-1}(y/x)$ mit $\arctan_2(y,x)$.

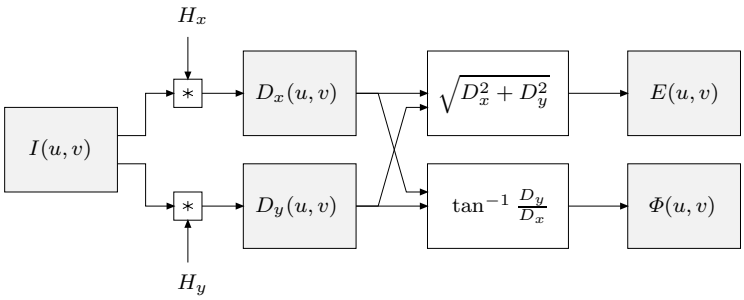


Abbildung 7.5
Typischer Einsatz von Gradientenfiltern. Mit den beiden Gradientenfiltern H_x und H_y werden zunächst zwei Gradientenbilder D_x und D_y erzeugt und daraus die Kantenstärke E und die Kantenrichtung Φ für jede Bildposition (u, v) berechnet.

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

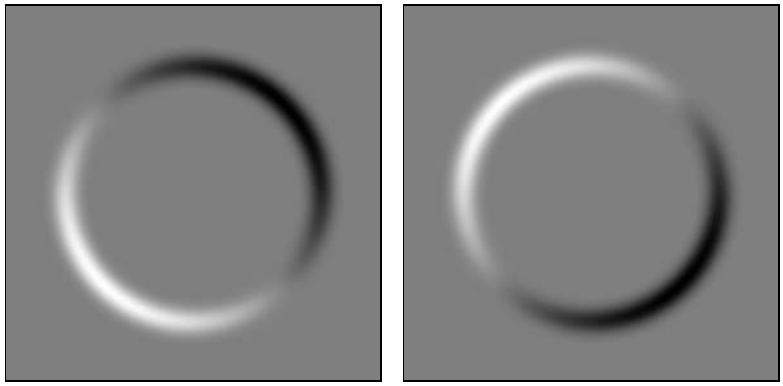
Vor allem der Sobel-Operator ist aufgrund seiner guten Ergebnisse (s. auch Abb. 7.9) und seiner Einfachheit weit verbreitet und in vielen Softwarepaketen für die Bildverarbeitung implementiert.

7.3.2 Roberts-Operator

Als einer der ältesten Kantenoperatoren ist der „Roberts-Operator“ [70] vor allem historisch interessant. Er benutzt zwei – mit 2×2 extrem kleine – Filter, um die Richtungsableitungen entlang der beiden Diagonalen zu schätzen:

$$H_1^R = \begin{bmatrix} 0 & \underline{1} \\ -1 & 0 \end{bmatrix} \quad \text{und} \quad H_2^R = \begin{bmatrix} -1 & 0 \\ 0 & \underline{1} \end{bmatrix}. \quad (7.16)$$

Diese Filter reagieren entsprechend stark auf diagonal verlaufende Kanten, wobei die Filter allerdings wenig richtungsselektiv sind, d. h., dass jedes der Filter über ein sehr breites Band an Orientierungen hinweg ähnlich stark reagiert (Abb. 7.6). Die Kantenstärke wird aus den beiden



D_1 D_2

Abbildung 7.6
Diagonale Richtungskomponenten beim Roberts-Operator.

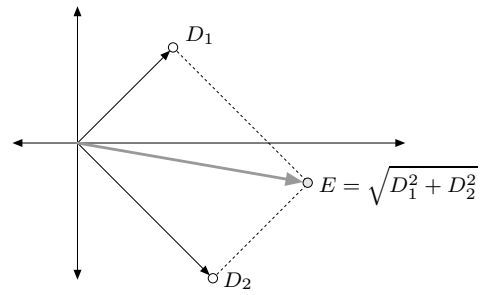


Abbildung 7.7
Kantenstärke beim Roberts-Operator. Die Kantenstärke $E(u, v)$ wird als Summe der beiden orthogonalen Richtungsvektoren $D_1(u, v)$ und $D_2(u, v)$ berechnet.

Filterantworten analog zum Betrag des Gradienten (Gl. 7.5) berechnet, allerdings mit (um 45°) gedrehten Richtungsvektoren (Abb. 7.7).

7.3.3 Kompass-Operatoren

Das Design eines guten Kantenfilters ist ein Kompromiss: Je besser ein Filter auf „kantenartige“ Bildstrukturen reagiert, desto stärker ist auch seine Richtungsabhängigkeit, d. h., umso enger ist der Winkelbereich, auf den das Filter anspricht.

Eine Lösung ist daher, nicht nur ein Paar von relativ „breiten“ Filtern für zwei (orthogonale) Richtungen einzusetzen, sondern einen Satz „engerer“ Filter für mehrere Richtungen. Ein klassisches Beispiel ist der Kantenoperator von Kirsch [52], der für acht verschiedene Richtungen im Abstand von 45° folgende Filter vorsieht:

$$H_0^K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad H_1^K = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} \tag{7.17}$$

$$H_2^K = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad H_3^K = \begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix}$$

$$H_4^K = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad H_5^K = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}$$

$$H_6^K = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad H_7^K = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$$

Von diesen acht Filtern $H_0, H_1, \ldots H_7$ müssen allerdings nur vier tatsächlich berechnet werden, denn die übrigen vier sind bis auf das Vorzeichen identisch zu den ersten. Zum Beispiel ist $H_4^K = -H_0^K$, sodass aufgrund der Linearitätseigenschaften der Faltung (Gl. 6.18) gilt

$$I * H_4^K = I * -H_0^K = -(I * H_0^K)\,. \tag{7.18}$$

Die acht Richtungsbilder für den Kirsch-Operator $D_0, D_1, \ldots D_7$ werden also folgendermaßen ermittelt:

$$\begin{array}{llll} D_0 = I * H_0^K & D_1 = I * H_1^K & D_2 = I * H_2^K & D_3 = I * H_3^K \\ D_4 = -D_0 & D_5 = -D_1 & D_6 = -D_2 & D_7 = -D_3 \end{array} \tag{7.19}$$

Die eigentliche Kantenstärke E^K an der Stelle (u, v) ist als Maximum der einzelnen Filterergebnisse definiert, d. h.

$$\begin{aligned} E^K(u,v) &= \max\big(D_0(u,v), D_1(u,v), \ldots D_7(u,v)\big) \\ &= \max\big(|D_0(u,v)|, |D_1(u,v)|, |D_2(u,v)|, |D_3(u,v)|\big)\ , \end{aligned} \tag{7.20}$$

und das am stärksten ansprechende Filter bestimmt auch die zugehörige Kantenrichtung

$$\Phi^K(u,v) = \frac{\pi}{4} j\ , \quad \text{wobei } \ j = \operatorname*{argmax}_{0 \le i \le 7} D_i(u,v)\ . \tag{7.21}$$

Im praktischen Ergebnis bieten derartige „Kompass"-Operatoren allerdings kaum Vorteile gegenüber einfacheren Operatoren, wie z. B. dem Sobel-Operator. Ein kleiner Vorteil des Kirsch-Operators ist allerdings, dass er keine Wurzelberechnung (die relativ aufwendig ist) benötigt.

7.3.4 Kantenoperatoren in ImageJ

In der aktuellen Version von ImageJ ist der Sobel-Operator (Abschn. 7.3.1) für praktisch alle Bildtypen implementiert und im Menü

Process→Find Edges

abrufbar. Der Operator ist auch als Methode `void findEdges()` für die Klasse `ImageProcessor` verfügbar.

7.4 Weitere Kantenoperatoren

Neben der in Abschn. 7.2 beschriebenen Gruppe von Kantenoperatoren, die auf der ersten Ableitung basieren, gibt es auch Operatoren auf Grundlage der *zweiten* Ableitung der Bildfunktion. Ein Problem der Kantendetektion mit der ersten Ableitung ist nämlich, dass Kanten genauso breit werden wie die Länge des zugehörigen Anstiegs in der Bildfunktion und ihre genaue Position dadurch schwierig zu lokalisieren ist.

7.4.1 Kantendetektion mit zweiten Ableitungen

Die zweite Ableitung einer Funktion misst deren lokale *Krümmung* und die Idee ist, die Nullstellen oder vielmehr die Positionen der Nulldurchgänge der zweiten Ableitung als Kantenpositionen zu verwenden. Die zweite Ableitung ist allerdings stark anfällig gegenüber Bildrauschen,

sodass zunächst eine Glättung mit einem geeigneten Glättungsfilter erforderlich ist. Der bekannteste Vertreter ist der so genannte „Laplacian-of-Gaussian"-Operator (LoG) [56], der die Glättung mit einem Gauß-Filter und die zweite Ableitung (Laplace-Filter, s. Abschn. 7.6.1) in ein gemeinsames, lineares Filter kombiniert.

Ein Beispiel für die Anwendung des LoG-Operators zeigt Abb. 7.9. Im direkten Vergleich mit dem Sobel- oder Prewitt-Operator fällt die klare Lokalisierbarkeit der Kanten und der relativ geringe Umfang an Streuergebnissen („clutter") auf. Weitere Details zum LoG-Operator sowie ein übersichtlicher Vergleich gängiger Kantenoperatoren finden sich in [72, Kap. 4] und [59].

7.4.2 Kanten auf verschiedenen Skalenebenen

Leider weichen die Ergebnisse einfacher Kantenoperatoren, wie die der bisher beschriebenen, oft stark von dem ab, was wir subjektiv als wichtige Kanten empfinden. Dafür gibt es vor allem zwei Gründe:

- Zum einen reagieren Kantenoperatoren nur auf lokale Intensitätsunterschiede, während unser visuelles System Konturen auch durch Bereiche minimaler oder verschwindender Unterschiede hindurch fortsetzen kann.
- Zweitens entstehen Kanten nicht nur in einer bestimmten Auflösung oder in einem bestimmten Maßstab, sondern auf vielen verschiedenen Skalenebenen.

Übliche, kleine Kantendetektoren, wie z. B. der Sobel-Operator, können ausschließlich auf Intensitätsunterschiede reagieren, die innerhalb ihrer 3×3-Filterregion stattfinden. Um Unterschiede über einen größeren Horizont wahrzunehmen, bräuchten wir entweder *größere* Kantenoperatoren (mit entsprechend großen Filtern) oder wir verwenden die ursprünglichen Operatoren auf verkleinerten (d. h. skalierten) Bildern. Das ist die Grundidee der so genannten „Multi-Resolution"-Techniken, die etwa auch als hierarchische Methoden oder Pyramidentechniken in vielen Bereichen der Bildverarbeitung eingesetzt werden [14]. In der Kantendetektion bedeutet dies, zunächst Kanten auf unterschiedlichen Auflösungsebenen zu finden und dann an jeder Bildposition zu entscheiden, welche Kante auf welcher der räumlichen Ebenen dominant ist.

7.4.3 Canny-Filter

Ein bekanntes Beispiel für ein derartiges Verfahren ist der Kantendetektor von Canny [15], der einen Satz von gerichteten (und relativ großen) Filtern auf mehreren Auflösungsebenen verwendet und deren Ergebnisse in ein gemeinsames Kantenbild („edge map") zusammenfügt. Die Methode versucht, drei Ziele gleichzeitig zu erreichen: (a) die Anzahl falscher Kantenmarkierungen zu minimieren, (b) Kanten möglichst

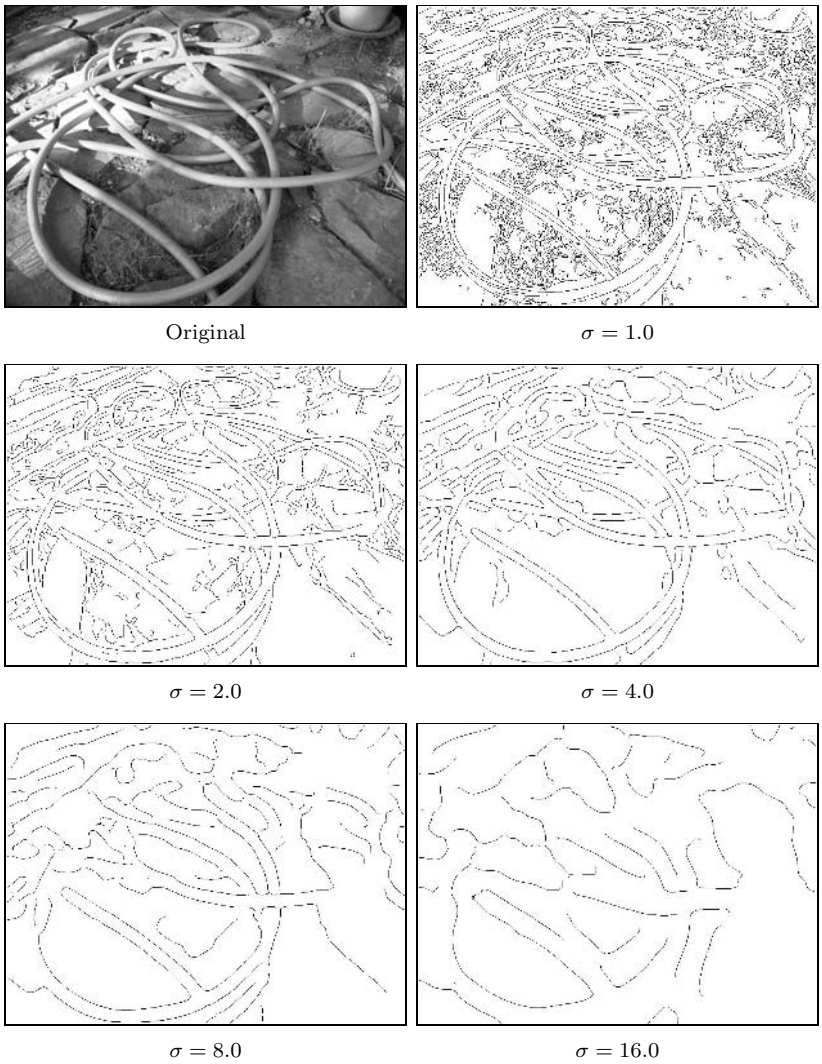


Abbildung 7.8
Canny-Kantendetektor mit unterschiedlichen Einstellungen des Glättungsparameters σ.

gut zu lokalisieren und (c) nur eine Markierung pro Kante zu liefern. Das „Canny-Filter“ ist im Kern ein Gradientenverfahren (Abschn. 7.2), benützt aber für die Kantenlokalisierung auch die Nulldurchgänge der zweiten Ableitungen. Meistens wird der Algorithmus nur in einer „single-scale“-Version verwendet, wobei aber durch Einstellung des Filterradius (Glättungsparameter σ) eine gegenüber einfachen Operatoren wesentlich verbesserte Kantendetektion möglich ist (Abb. 7.8 und Abb. 7.9). Der Canny-Detektor wird daher auch in seiner elementaren Form gegenüber einfacheren Kantendetektoren oft bevorzugt. Eine ausführliche Beschreibung des Algorithmus und eine Java-Implementierung findet man z. B. in [22, Kap. 7].

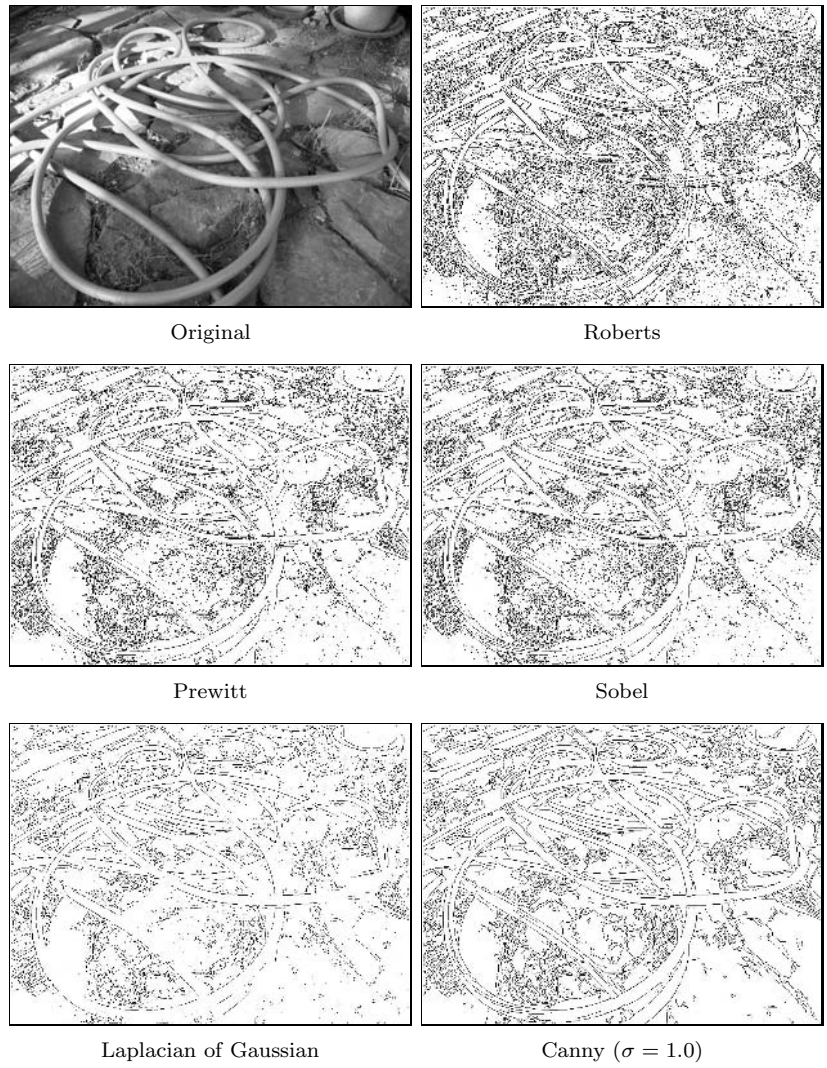


Abbildung 7.9
Vergleich unterschiedlicher Kantendetektoren. Wichtigstes Kriterium für die Beurteilung des Kantenbilds ist einerseits die Menge von irrelevanten Kantenelementen und andererseits der Zusammenhang der dominanten Kanten. Deutlich ist zu erkennen, dass der Canny-Detektor auch bei nur *einem* fixen und relativ kleinen σ ein wesentlich klareres Kantenbild liefert als die einfachen Operatoren, wie die von Roberts und Sobel.

7.5 Von Kanten zu Konturen

Welche Methode der Kantendetektion wir auch immer verwenden, am Ende erhalten wir einen kontinuierlichen Wert für die Kantenstärke an jeder Bildposition und eventuell auch den Winkel der lokalen Kantenrichtung. Wie können wir diese Information benutzen, um z. B. größere Bildstrukturen zu finden, insbesondere die Konturen von Objekten?

7.5.1 Konturen verfolgen

Eine verbreitete Idee ist, Konturen auf Basis der lokalen Kantenstärke und entlang der Kantenrichtung sequentiell zu verfolgen. Die Idee ist

grundsätzlich einfach: Beginnend bei einem Bildpunkt mit hoher Kantenstärke, verfolgt man die davon ausgehende Kontur schrittweise in beide Richtungen, bis sich die Spur zu einer durchgehenden Kontur schließt. Leider gibt es einige Hindernisse, die dieses Unterfangen schwieriger machen, als es zunächst erscheint:

- Kanten können einfach in Regionen enden, in denen der Helligkeitsgradient verschwindet,
- sich kreuzende Kanten machen die Verfolgung mehrdeutig, und
- Konturen können sich in mehrere Richtungen aufspalten.

Aufgrund dieser Schwierigkeiten wird Konturverfolgung kaum auf Originalbilder oder kontinuierlichen Kantenbilder angewandt, außer in einfachen Situationen, wenn z. B. eine klare Trennung zwischen Objekt und Hintergrund (*Figure-Background*) gegeben ist. Wesentlich einfacher ist die Verfolgung von Konturen natürlich anhand von Binärbildern (s. Kap. 11).

7.5.2 Kantenbilder

In vielen Anwendungen ist der nächste Schritt nach der Kantenverstärkung (durch entspechende Kantenfilter) die Auswahl der Kantenpunkte, d. h. eine binäre Entscheidung, welche Kantenpixel tatsächlich als Kantenpunkte betrachtet werden und welche nicht. Im einfachsten Fall kann dies durch Anwendung einer Schwellwertoperation auf die Ergebnisse der Kantenfilter erledigt werden. Das Ergebnis ist ein binäres Kantenbild – ein so genanntes „edge map“.

Kantenbilder zeigen selten perfekte Konturen, sondern vielmehr kleine, nicht zusammenhängende Konturfragmente, die an jenen Stellen unterbrochen sind, an denen die Kantenstärke zu gering ist. Nach der Schwellwertoperation ist natürlich an diesen Stellen überhaupt keine Kanteninformation mehr vorhanden, auf die man später noch zurückgreifen könnte. Trotz dieser Problematik werden Schwellwertoperationen aufgrund ihrer Einfachheit in der Praxis häufig eingesetzt und es gibt Methoden wie die Hough-Transformation (s. Kap. 9), die mit diesen oft sehr lückenhaften Kantenbildern dennoch gut zurechtkommen.

7.6 Kantenschärfung

Das nachträgliche Schärfen von Bildern ist eine häufige Aufgabenstellung, z. B. um Unschärfe zu kompensieren, die beim Scannen oder Skalieren von Bildern entstanden ist, oder um einen nachfolgenden Schärfeverlust (z. B. beim Druck oder bei der Bildschirmanzeige) zu kompensieren. Die übliche Methode des Schärfens ist das Anheben der hochfrequenten Bildanteile, die primär an den raschen Bildübergängen auftreten und für den Schärfeeindruck des Bilds verantwortlich sind.

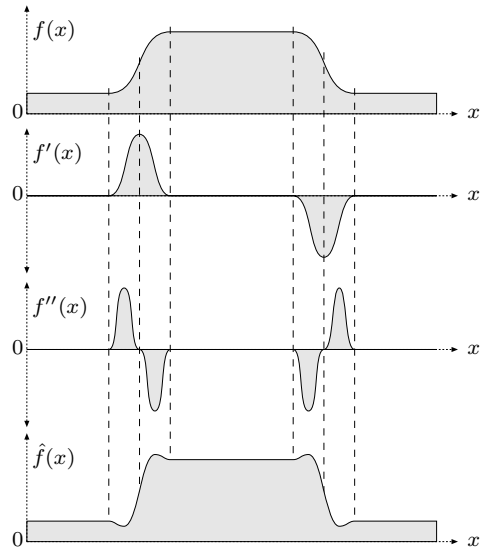


Abbildung 7.10
Kantenschärfung mit der zweiten Ableitung. Ursprüngliches Bildprofil $f(x)$, erste Ableitung $f'(x)$, zweite Ableitung $f''(x)$, geschärfte Funktion $\hat{f}(x) = f(x) - wf''(x)$ (w ist ein Gewichtungsfaktor).

Wir beschreiben im Folgenden zwei gängige Ansätze zur künstlichen Bildschärfung, die technisch auf ähnlichen Grundlagen basieren wie die Kantendetektion und daher gut in dieses Kapitel passen.

7.6.1 Kantenschärfung mit dem Laplace-Filter

Eine gängige Methode zur Lokalisierung von raschen Intensitätsänderungen sind Filter auf Basis der zweiten Ableitung der Bildfunktion. Abb. 7.10 illustriert die Idee anhand einer eindimensionalen, kontinuierlichen Funktion $f(x)$. Auf eine stufenförmige Kante reagiert die zweite Ableitung mit einem positiven Ausschlag am unteren Ende des Anstiegs und einem negativen Ausschlag am oberen Ende. Die Kante wird geschärft, indem das Ergebnis der zweiten Ableitung $f''(x)$ (bzw. ein gewisser Anteil davon) von der ursprünglichen Funktion $f(x)$ subtrahiert wird, d. h.

$$\hat{f}(x) = f(x) - w \cdot f''(x)\,. \tag{7.22}$$

Abhängig vom Gewichtungsfaktor w wird durch Gl. 7.22 ein Überschwingen der Bildfunktion zu beiden Seiten der Kante erzielt, wodurch eine Übersteigerung der Kante und damit ein subjektiver Schärfungseffekt entsteht.

Laplace-Operator

Für die Kantenschärfung im zweidimensionalen Fall verwendet man die zweiten Ableitungen in horizontaler und vertikaler Richtung kombiniert in Form des so genannten Laplace-Operators. Der Laplace-Operator ∇^2 einer zweidimensionalen Funktion $f(x, y)$ ist definiert als die Summe der zweiten partiellen Ableitungen in x- und y-Richtung, d. h.

$$(\nabla^2 f)(x,y) = \frac{\partial^2 f}{\partial^2 x}(x,y) + \frac{\partial^2 f}{\partial^2 y}(x,y). \tag{7.23}$$

Genauso wie die ersten Ableitungen (Abschn. 7.2.2) können auch die zweiten Ableitungen einer diskreten Bildfunktion mithilfe einfacher linearer Filter berechnet werden. Auch hier gibt es mehrere Varianten, z. B. die beiden Filter

$$\frac{\partial^2 f}{\partial^2 x} \approx H_x^L = \begin{bmatrix} 1 & -\underline{2} & 1 \end{bmatrix} \qquad \text{und} \qquad \frac{\partial^2 f}{\partial^2 y} \approx H_y^L = \begin{bmatrix} 1 \\ -\underline{2} \\ 1 \end{bmatrix}, \tag{7.24}$$

die zusammen ein zweidimensionales Laplace-Filter der Form

$$H^L = H_x^L + H_y^L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -\underline{4} & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{7.25}$$

bilden (für eine Herleitung siehe z. B. [48, S. 347]). Ein Beispiel für die Anwendung des Laplace-Filters H^L auf ein Grauwertbild zeigt Abb. 7.11. H^L ist übrigens kein separierbares Filter im herkömmlichen Sinn (Abschn. 6.3.3), kann aber wegen der Linearitätseigenschaften der Faltung (Gl. 6.17 und Gl. 6.19) mit eindimensionalen Filtern in der Form

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

dargestellt und berechnet werden. Wie bei den Gradientenfiltern ist auch bei allen Laplace-Filtern die Summe der Koeffizienten null, sodass sich in Bildbereichen mit konstanter Intensität die Filterantwort null ergibt (Abb. 7.11). Weitere gebräuchliche Varianten von 3×3-Laplace-Filtern sind

$$H_8^L = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -\underline{8} & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{oder} \quad H_{12}^L = \begin{bmatrix} 1 & 2 & 1 \\ 2 & -\underline{12} & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

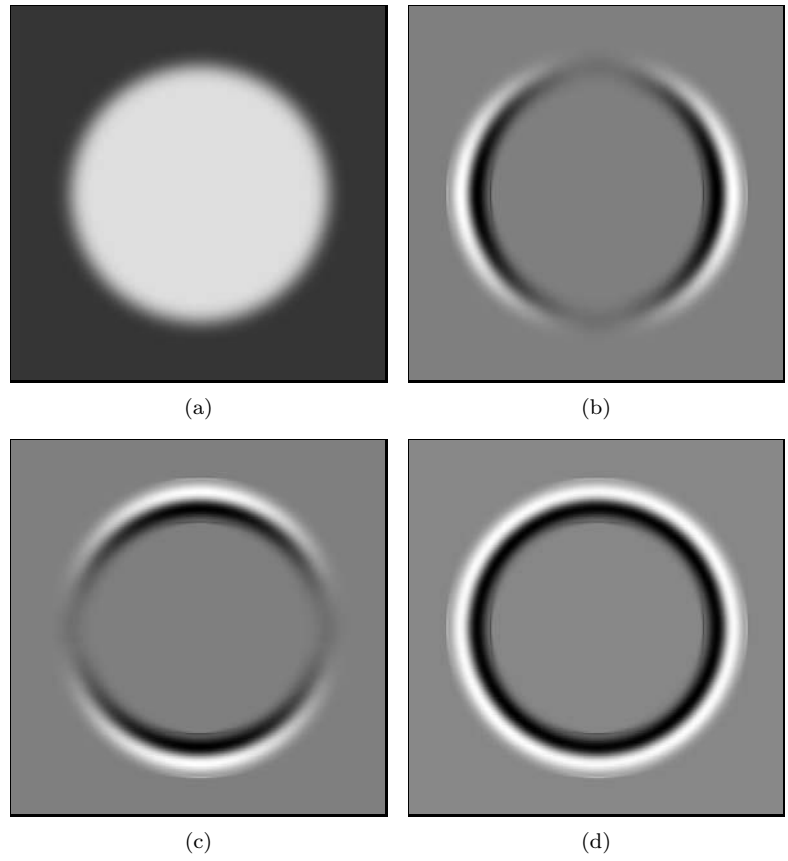
Schärfung

Für die eigentliche Schärfung filtern wir zunächst, wie in Gl. 7.22 für den eindimensionalen Fall gezeigt, das Bild I mit dem Laplace-Filter H^L und subtrahieren anschließend das Ergebnis vom ursprünglichen Bild, d. h.

$$I' \leftarrow I - w \cdot (H^L * I). \tag{7.26}$$

Der Faktor w bestimmt dabei den Anteil der Laplace-Komponente und damit die Stärke der Schärfung. Die richtige Wahl des Faktors ist u. a. vom verwendeten Laplace-Filter (Normalisierung) abhängig.

Abb. 7.11 zeigt die Anwendung eines Laplace-Filters mit der Filtermatrix aus Gl. 7.25 auf ein Testbild, wobei die paarweise auftretenden Pulse an beiden Seiten jeder Kante deutlich zu erkennen sind. Das Filter



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

Abbildung 7.11
Anwendung des Laplace-Filters H^L. Synthetisches Testbild (a), zweite Ableitung horizontal $\partial^2 I/\partial^2 u$ (b), zweite Ableitung vertikal $\partial^2 I/\partial^2 v$ (c), Laplace-Operator $\nabla^2 I(u, v)$ (d). Die Helligkeiten sind in (b–d) so skaliert, dass maximal negative Werte schwarz, maximal positive Werte weiß und Nullwerte grau dargestellt werden.

reagiert trotz der relativ kleinen Filtermatrix annähernd gleichförmig in alle Richtungen. Die Anwendung auf ein realistisches Grauwertbild unter Verwendung der Filtermatrix H^L (Gl. 7.25) und $w = 1.0$ ist in Abb. 7.12 gezeigt.

Wie bei Ableitungen zweiter Ordnung zu erwarten, ist auch das Laplace-Filter relativ anfällig gegenüber Bildrauschen, was allerdings wie in der Kantendetektion durch vorhergehende Glättung mit Gauß-Filtern gelöst werden kann (Abschn. 7.4.1).

7.6.2 Unscharfe Maskierung (*unsharp masking*)

Eine ähnliche, vor allem im Bereich der Astronomie, der digitalen Druckvorstufe und vielen anderen Bereichen der digitalen Bildverarbeitung sehr populäre Methode zur Kantenschärfung ist die so genannte „unscharfe Maskierung“ (*unsharp masking*, USM). Der Begriff stammt ursprünglich aus der analogen Filmtechnik, wo man durch optische Überlagerung mit unscharfen Duplikaten die Schärfe von Bildern erhöht hat. Das Verfahren ist in der digitalen Bildverarbeitung grundsätzlich das gleiche.

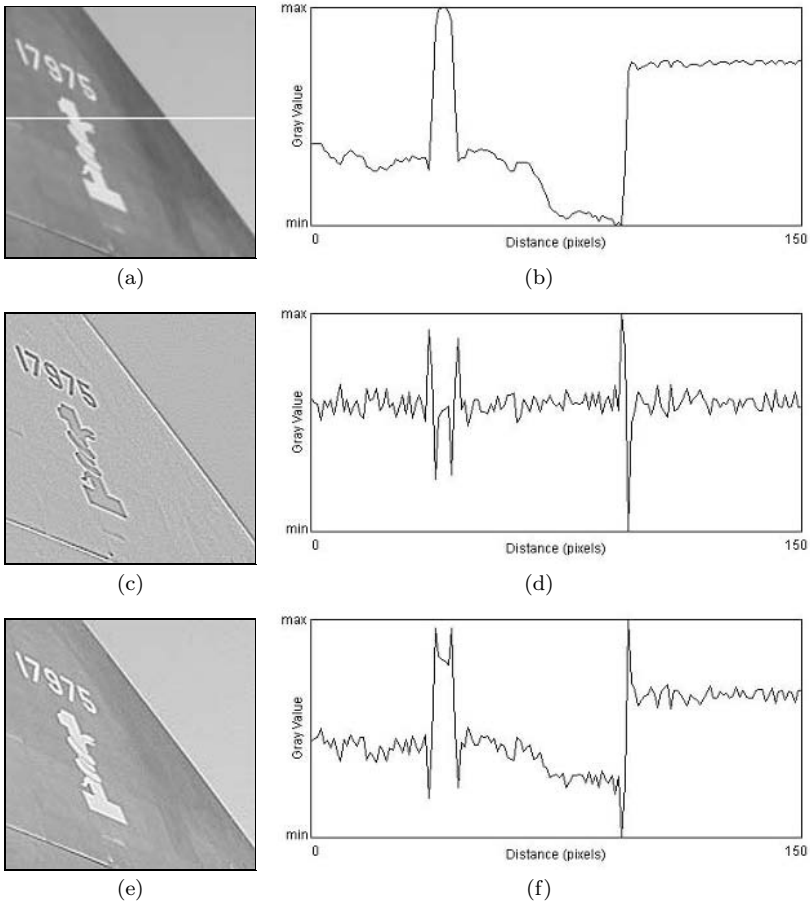


Abbildung 7.12
Kantenschärfung mit dem Laplace-Filter. Originalbild und Profil der markierten Bildzeile (a–b), Ergebnis des Laplace-Filters H^L (c–d), geschärftes Bild mit Schärfungsfaktor $w = 1.0$ (e–f).

Ablauf

Im USM-Filter wird zunächst eine geglättete Version des Bilds vom Originalbild subtrahiert. Das Ergebnis wird als „Maske“ bezeichnet und verstärkt die Kantenstrukturen. Nachfolgend wird die Maske wieder zum Original addiert, wodurch sich eine Schärfung der Kanten ergibt. In der analogen Filmtechnik wurde die dafür notwendige Unschärfe durch Defokussierung der Optik erreicht. Die einzelnen Schritte des USM-Filters zusammengefasst:

1. Zur Erzeugung der Maske M wird eine durch ein Filter $\tilde{H}$ geglättete Version des Originalbilds I vom Originalbild subtrahiert:

$$M \;=\; I - (I * \tilde{H}) \tag{7.27}$$

   Dabei wird angenommen, dass die Filtermatrix $\tilde{H}$ normalisiert ist (Abschn. 6.2.5).
2. Die Maske wird nun wieder zum Originalbild addiert, verbunden mit einem Gewichtungsfaktor a, der die Stärke der Schärfung steuert:

$$\begin{aligned} I' &= I + a \cdot M \\ &= I + a \cdot (I - (I * \tilde{H})) \\ &= (1 + a) \cdot I - a \cdot (I * \tilde{H}) \end{aligned} \tag{7.28}$$

Glättungsfilter

Prinzipiell könnte für $\tilde{H}$ in Gl. 7.27 jedes Glättungsfilter eingesetzt werden, üblicherweise werden aber Gauß-Filter $H^{G,\sigma}$ mit variablem Radius σ verwendet (s. auch Abschn. 6.2.7). Typische Werte für σ liegen im Bereich $1 \ldots 20$ und für den Gewichtungsfaktor a im Bereich $0.2 \ldots 4.0$ (entspr. $20\% \ldots 400\%$).

Abb. 7.13 zeigt einige Beispiele für die Auswirkungen des USM-Filters bei unterschiedlichem Glättungsradius σ. Der Vorteil des USM-Filters gegenüber der Schärfung mit dem Laplace-Filter ist zum einen die durch die Glättung geringere Rauschanfälligkeit und zum anderen die bessere Steuerungsmöglichkeit über die Parameter σ (räumliche Ausdehnung) und a (Stärke).

Erweiterungen

Das USM-Filter reagiert natürlich nicht nur auf echte Kanten, sondern in abgeschwächter Form auf jede lokale Intensitätsänderung im Bild, und verstärkt dadurch u. a. sichtbare Rauscheffekte in flachen Bildregionen. In einigen Implementierungen (z. B. in *Adobe Photoshop*) ist daher ein zusätzlicher Schwellwertparameter vorgesehen, der den minimalen lokalen Kontrast (zwischen dem aktuellen Pixel und seinen Nachbarn) definiert, ab dem eine Schärfung an dieser Stelle stattfindet. Wird dieser Kontrastwert nicht erreicht, dann bleibt das aktuelle Pixel unverändert. Im Unterschied zum ursprünglichen USM-Filter ist diese erweiterte Form natürlich kein lineares Filter mehr.

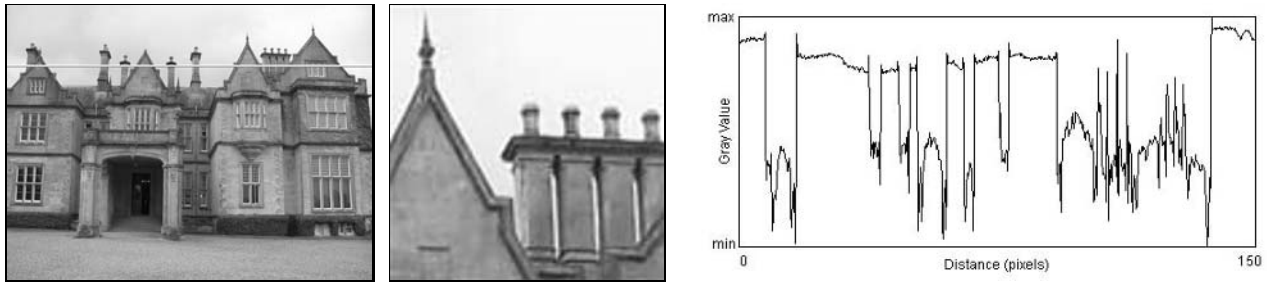
Das USM-Filter ist in praktisch jeder Bildbearbeitungssoftware verfügbar und stellt wegen seiner hohen Flexibilität für viele professionelle Benutzer ein unverzichtbares Werkzeug dar. Bei Farbbildern wird das USM-Filter normalerweise mit identischen Parametereinstellungen auf alle Farbkanäle einzeln angewandt.

In ImageJ ist das USM-Filter als Plugin-Klasse `ij.plugin.filter.UnsharpMask` implementiert und unter

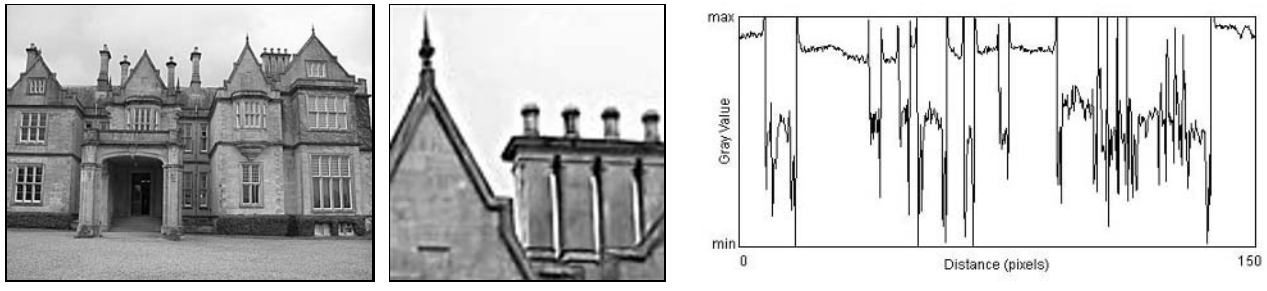
Process→Filter→Unsharp Mask...

verfügbar.[3] Innerhalb eigener Plugin-Programme kann man dieses Filter z. B. in folgender Form verwenden:

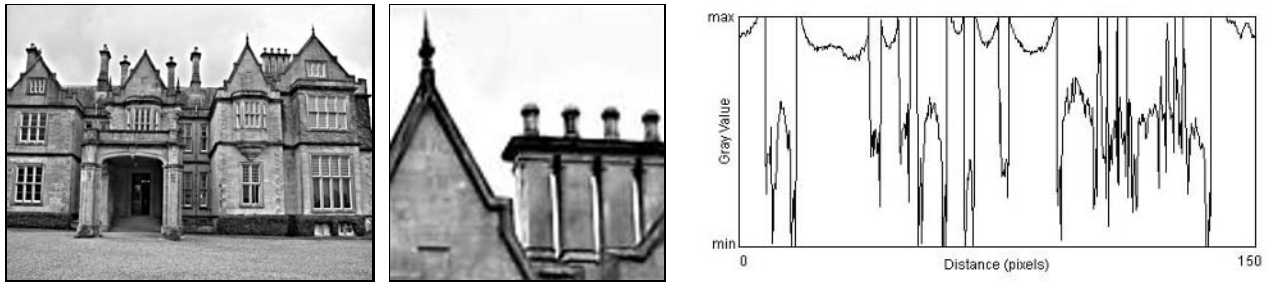
[3] Allerdings mit etwas zu klein dimensionierten Filterkernen für das Gauß-Filter (s. auch Abschn. 6.6.2).



Original



$\sigma = 2.5$



$\sigma = 10.0$

Abbildung 7.13. USM-Filter für unterschiedliche Radien $\sigma = 2.5$ und 10.0. Der Wert des Parameters a (Stärke der Schärfung) ist 100%. Die Profile rechts zeigen den Intensitätsverlauf der im Originalbild markierten Bildzeile.

```
1  import ij.plugin.filter.UnsharpMask;
2  ...
3  public void run(ImageProcessor imp) {
4    UnsharpMask usm = new UnsharpMask();
5    double r = 2.0; // standard settings for radius
6    double a = 0.6; // standard settings for weight
7    usm.sharpen(imp, r, a);
8    ...
9  }
```

Laplace- vs. USM-Filter

Bei einem näheren Vergleich der beiden Methoden sehen wir, dass die Schärfung mit dem Laplace-Filter (Abschn. 7.6.1) eigentlich ein spezieller Fall des USM-Filters ist. Wenn wir das Laplace-Filter aus Gl. 7.25 zerlegen in der Form

$$H^L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & \underline{-4} & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0\,1\,0 \\ 1\,1\,1 \\ 0\,1\,0 \end{bmatrix} - 5 \begin{bmatrix} 0\,0\,0 \\ 0\,1\,0 \\ 0\,0\,0 \end{bmatrix} = 5\big(\tilde{H}^L - \delta\big), \qquad (7.29)$$

wird deutlich, dass H^L aus einem einfachen 3×3-Glättungsfilter $\tilde{H}^L$ abzüglich der Impulsfunktion δ besteht. Die Laplace-Schärfung aus Gl. 7.26 können wir daher ausdrücken als

$$\begin{aligned} I'_L &\leftarrow I - w \cdot (H^L * I) \\ &= I - 5w \cdot (\tilde{H}^L * I - I) \\ &= I + 5w \cdot (I - \tilde{H}^L * I) \\ &= I + 5w \cdot M, \end{aligned} \qquad (7.30)$$

also in der Form des USM-Filters (Gl. 7.27) mit der Maske $M = (I - \tilde{H}^L * I)$. Die Schärfung mit dem Laplace-Filter bei einem Gewichtungsfaktor w ist damit ein Sonderfall des USM-Filters mit dem spezifischen Glättungsfilter

$$\tilde{H}^L = \frac{1}{5} \begin{bmatrix} 0\,1\,0 \\ 1\,1\,1 \\ 0\,1\,0 \end{bmatrix} \text{ und dem Schärfungsfaktor } a = 5w.$$

7.7 Aufgaben

Aufg. 7.1. Berechnen Sie manuell den Gradienten und den Laplace-Operator für folgende Bildmatrix unter Verwendung der Approximation in Gl. 7.2 bzw. Gl. 7.25:

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

Aufg. 7.2. Implementieren Sie den Sobel-Kantendetektor nach Gl. 7.10 und Abb. 7.5 als ImageJ-Plugin. Das Plugin soll zwei neue Bilder generieren, eines für die ermittelte Kantenstärke $E(u,v)$ und ein zweites für die Orientierung $\Phi(u,v)$. Überlegen Sie, wie man die Orientierung sinnvoll anzeigen könnte.

Aufg. 7.3. Stellen Sie den Sobel-Operator analog zu Gl. 7.9 in xy-separierbarer Form dar.

Aufg. 7.4. Implementieren Sie, wie in Aufg. 7.2, den Kirsch-Operator und vergleichen Sie vor allem die Schätzung der Kantenrichtung beider Verfahren.

Aufg. 7.5. Konzipieren Sie einen Kompass-Operator mit mehr als 8 (16?) unterschiedlich gerichteten Filtern.

Aufg. 7.6. Vergleichen Sie die Ergebnisse der *Unsharp Mask*-Filter in ImageJ und *Adobe Photoshop* anhand eines selbst gewählten, einfachen Testbilds. Wie sind die Parameterwerte für σ (*radius*) und a (*weight*) in beiden Implementierungen zu definieren?

8

Auffinden von Eckpunkten

Eckpunkte sind markante strukturelle Ereignisse in einem Bild und daher in einer Reihe von Anwendungen nützlich, wie z. B. beim Verfolgen von Elementen in aufeinander folgenden Bildern (*tracking*), bei der Zuordnung von Bildstrukturen in Stereoaufnahmen, als Referenzpunkte zur geometrischen Vermessung, Kalibrierung von Kamerasystemen usw. Eckpunkte sind nicht nur für uns Menschen auffällig, sondern sind auch aus technischer Sicht „robuste“ Merkmale, die in 3D-Szenen nicht zufällig entstehen und in einem breiten Bereich von Ansichtswinkeln sowie unter unterschiedlichen Beleuchtungsbedingungen relativ zuverlässig zu lokalisieren sind.

8.1 „Points of interest“

Trotz ihrer Auffälligkeit ist das automatische Bestimmen und Lokalisieren von Eckpunkten (corners) nicht ganz so einfach, wie es zunächst erscheint. Ein guter „corner detector“ muss mehrere Kriterien erfüllen: Er soll wichtige von unwichtigen Eckpunkten unterscheiden und Eckpunkte zuverlässig auch unter realistischem Bildrauschen finden, er soll die gefundenen Eckpunkte möglichst genau lokalisieren können und zudem effizient arbeiten, um eventuell auch in Echtzeitanwendungen (wie z. B. Video-Tracking) einsetzbar zu sein.

Wie immer gibt es nicht nur einen Ansatz für diese Aufgabe, aber im Prinzip basieren die meisten Verfahren zum Auffinden von Eckpunkten oder ähnlicher „interest points“ auf einer gemeinsamen Grundlage – während eine *Kante* in der Regel definiert wird als eine Stelle im Bild, an der der Gradient der Bildfunktion in *einer* bestimmten Richtung besonders hoch und normal dazu besonders niedrig ist, weist ein *Eckpunkt* einen starken Gradientenwert in *mehr als einer Richtung* gleichzeitig auf.

Die meisten Verfahren verwenden daher die ersten oder zweiten Ableitungen der Bildfunktion in x- und y-Richtung zur Bestimmung von Eckpunkten (z. B. [25, 34, 53, 54]). Ein für diese Methode repräsentatives Beispiel ist der so genannte Harris-Detektor (auch bekannt als „Plessey feature point detector“ [34]), den wir im Folgenden genauer beschreiben. Obwohl mittlerweile leistungsfähigere Verfahren bekannt sind (siehe z. B. [73, 79]), werden der Harris-Detektor und verwandte Ansätze in der Praxis häufig verwendet.

8.2 Harris-Detektor

Der Operator, der von Harris und Stephens [34] entwickelt wurde, ist einer von mehreren, ähnlichen Algorithmen basierend auf derselben Idee: ein Eckpunkt ist dort gegeben, wo der Gradient der Bildfunktion gleichzeitig in mehr als einer Richtung einen hohen Wert aufweist. Insbesondere sollen Stellen entlang von Kanten, wo der Gradient zwar hoch, aber nur in einer Richtung ausgeprägt ist, nicht als Eckpunkte gelten. Darüber hinaus sollen Eckpunkte natürlich unabhängig von ihrer Orientierung, d. h. in isotroper Weise, gefunden werden.

8.2.1 Lokale Strukturmatrix

Grundlage des Harris-Detektors sind die ersten partiellen Ableitungen der Bildfunktion $I(u, v)$ in horizontaler und vertikaler Richtung,

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

Für jede Bildposition (u, v) werden zunächst drei Werte $A(u, v)$, $B(u, v)$ und $C(u, v)$ berechnet,

$$A(u,v) = I_x^2(u,v) \qquad B(u,v) = I_y^2(u,v) \tag{8.2}$$

$$C(u,v) = I_x(u,v) \cdot I_y(u,v) \tag{8.3}$$

die als Elemente einer *lokalen Strukturmatrix* $M(u, v)$ interpretiert werden:[1]

$$\boldsymbol{M} = \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix} = \begin{pmatrix} A & C \\ C & B \end{pmatrix}. \tag{8.4}$$

Anschließend werden die drei Funktionen $A(u, v)$, $B(u, v)$, $C(u, v)$ individuell durch Faltung mit einem linearen Gauß-Filter $H^{G,\sigma}$ (siehe Abschn. 6.2.7) geglättet, also

$$\bar{M} = \begin{pmatrix} A{*}H^{G,\sigma} & C{*}H^{G,\sigma} \\ C{*}H^{G,\sigma} & B{*}H^{G,\sigma} \end{pmatrix} = \begin{pmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{pmatrix}. \tag{8.5}$$

[1] Wir notieren zur leichteren Lesbarkeit im Folgenden die Funktionen ohne Koordinaten (u, v), d. h. $I_x \equiv I_x(u, v)$ oder $A \equiv A(u, v)$ etc.

Die Matrix $\bar{M}$ lässt sich aufgrund ihrer Symmetrie diagonalisieren in

$$\bar{M}' = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}, \tag{8.6}$$

wobei λ_1 und λ_2 die beiden *Eigenwerte* der Matrix $\bar{M}$ sind, definiert als[2]

$$\begin{aligned} \lambda_{1,2} &= \frac{\mathrm{trace}(\bar{M})}{2} \pm \sqrt{\Big(\frac{\mathrm{trace}(\bar{M})}{2}\Big)^2 - \det(\bar{M})} \\ &= \frac{1}{2}\left(\bar{A} + \bar{B} \pm \sqrt{\bar{A}^2 - 2\bar{A}\bar{B} + \bar{B}^2 + 4\bar{C}^2}\right). \end{aligned} \tag{8.7}$$

Beide Eigenwerte λ_1 und λ_2 sind positiv und enthalten essentielle Informationen über die lokale Bildstruktur. Innerhalb einer uniformen (flachen) Bildregion ist $\bar{M} = 0$ und deshalb sind auch die Eigenwerte $\lambda_1 = \lambda_2 = 0$. Umgekehrt gilt auf einer perfekten Sprungkante $\lambda_1 > 0$ und $\lambda_2 = 0$, unabhängig von der Orientierung der Kante. Die Eigenwerte kodieren also die *Kantenstärke*, die zugehörigen *Eigenvektoren* die entsprechende *Kantenrichtung*.

An einem Eckpunkt sollte eine starke Kante sowohl in der Hauptrichtung (entsprechend dem größeren der beiden Eigenwerte) wie auch normal dazu (entsprechend dem kleineren Eigenwert) vorhanden sein, beide Eigenwerte müssen daher signifikante Werte haben. Da $\bar{A}, \bar{B} \geq 0$, können wir davon ausgehen, dass $\mathrm{trace}(\bar{M}) > 0$ und daher auch $|\lambda_1| \geq |\lambda_2|$. Also ist für die Bestimmung eines Eckpunkts nur der kleinere der beiden Eigenwerte, d. h. $\lambda_2 = \mathrm{trace}(\bar{M})/2 - \sqrt{\ \dots\ }$, relevant.

8.2.2 *Corner Response Function* (CRF)

Wie wir aus Gl. 8.7 sehen, ist die Differenz zwischen den beiden Eigenwerten

$$\lambda_1 - \lambda_2 \;=\; 2 \cdot \sqrt{\Big(\frac{\mathrm{trace}(\bar{M})}{2}\Big)^2 - \det(\bar{M})},$$

wobei in jedem Fall $\big(0.25 \cdot \mathrm{trace}(\bar{M})^2\big) > \det(\bar{M})$ gilt. An einem Eckpunkt soll dieser Ausdruck möglichst klein werden, daher definiert der Harris-Detektor als Maß für „corner strength“ die Funktion

$$\begin{aligned} Q(u,v) &= \det(\bar{M}) - \alpha \cdot \big(\mathrm{trace}(\bar{M})\big)^2 \\ &= (\bar{A}\bar{B} - \bar{C}^2) - \alpha \cdot (\bar{A} + \bar{B})^2, \end{aligned} \tag{8.8}$$

wobei der Parameter α die Empfindlichkeit der Detektors steuert. $Q(u,v)$ wird als „corner response function“ bezeichnet und liefert maximale Werte an ausgeprägten Eckpunkten. α wird üblicherweise auf einen fixen Wert im Bereich $0.04 \ldots 0.06$ (max. 0.25) gesetzt. Ist α groß, wird der Detektor weniger empfindlich, d. h., weniger Eckpunkte werden gefunden.

[2] $\det(\bar{M})$ bezeichnet die *Determinante* und $\mathrm{trace}(\bar{M})$ die *Spur* (*trace*) der Matrix $\bar{M}$ (siehe z. B. [12, 90]).

8.2.3 Bestimmung der Eckpunkte

Eine Bildposition (u, v) wird als Kandidat für einen Eckpunkt ausgewählt, wenn

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

wobei der Schwellwert t_H typischerweise im Bereich 10.000–1.000.000 angesetzt wird, abhängig vom Bildinhalt. Die so selektierten Eckpunkte $\boldsymbol{c}_i = (u_i, v_i, q_i)$ werden in einer Liste

$$\mathit{Corners} = (\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots \boldsymbol{c}_N)$$

gesammelt, die nach der in Gl. 8.8 definierten *corner strength* $q_i = Q(u_i, v_i)$ in absteigender Reihenfolge sortiert ist (d. h. $q_i \geq q_{i+1}$). Um zu dicht platzierte Eckpunkte zu vermeiden, werden anschließend innerhalb einer bestimmten räumlichen Umgebung alle bis auf den stärksten Eckpunkt eliminiert. Dazu wird die Liste *Corners* von vorne nach hinten durchlaufen und alle schwächeren Eckpunkte weiter hinten in der Liste, die innerhalb der Umgebung eines stärkeren Punkts liegen, werden gelöscht.

Der vollständige Algorithmus für den Harris-Detektor ist nochmals in Alg. 8.1 übersichtlich zusammengefasst mit einer Zusammenstellung der zugehörigen Parametereinstellungen in Abb. 8.1.

8.2.4 Beispiele

Abb. 8.2 illustriert anhand eines einfachen, synthetischen Beispiels die wichtigsten Schritte bei der Detektion von Eckpunkten mit dem Harris-Detektor. Die Abbildung zeigt die Ergebnisse der Gradientenberechnung und die daraus abgeleiteten drei Komponenten der Strukturmatrix $M = \left(\begin{smallmatrix} A & C \\ C & B \end{smallmatrix}\right)$ sowie die Werte der *corner response function* $Q(u, v)$ für jede Bildposition (u, v). Für dieses Beispiel wurden die Standardeinstellungen der Parameter (Abb. 8.1) verwendet.

Das zweite Beispiel (Abb. 8.3) zeigt die Detektion von Eckpunkten in einem natürlichen Grauwertbild und demonstriert u. a. die nachträgliche Auswahl der stärksten Eckpunkte innerhalb einer bestimmten Umgebung.

8.3 Implementierung

Der Harris-Detektor ist komplexer als alle Algorithmen, die wir bisher beschrieben haben, und gleichzeitig ein typischer Repräsentant für viele ähnliche Methoden in der Bildverarbeitung. Wir nehmen das zum Anlass, die einzelnen Schritte der Implementierung und gleichzeitig auch die dabei immer wieder notwendigen Detailentscheidungen etwas umfassender als sonst aufzuzeigen. Der vollständige Quellcode der Klasse `HarrisCornerDet` ist im Anhang zu finden (Abschn. 4.1, S. 480–486).

Algorithmus 8.1
Harris-Detektor. Aus einem Intensitätsbild $I(u,v)$ wird eine sortierte Liste von Eckpunkten berechnet. Details zu den Parametern H_p, H_{dx}, H_{dy}, H_b, α, t_H und $d_{\min}$ finden sich in Abb. 8.1.

```
1:  HarrisCorners(I(u,v))
2:      Prefilter (smooth) the original image: I' ← I * H_p
3:      Step 1 – compute the corner response function:
4:          Compute the horizontal and vertical derivatives:
                I_x ← I' * H_dx,   I_y ← I' * H_dy
5:          Compute the components of the local structure matrix
                M = (A C; C B): A ← I_x^2,   B ← I_y^2,   C ← I_x I_y
6:          Blur each components of the structure matrix: M̄ = (Ā C̄; C̄ B̄):
                Ā ← A * H_b,   B̄ ← B * H_b,   C̄ ← C * H_b
7:          Compute the corner response function:
                Q ← (Ā · B̄ − C̄^2) − α · (Ā + B̄)^2
8:      Step 2 – collect corner points:
9:          Create an empty list Corners ← {}
10:         for all image coordinates (u,v) do
11:             if Q(u,v) > t_H and IsLocalMax(Q,u,v) then
12:                 Create corner node c_i = (u_i, v_i, q_i) ← (u, v, Q(u,v))
13:                 Add c_i to Corners
14:         Sort Corners by q_i in descending order.
15:         GoodCorners ← CleanUpNeighbors(Corners)
16:     return GoodCorners.

17: IsLocalMax(Q,u,v)        ▷ determine if Q(u,v) is a local maximum
18:     Let q_c ← Q(u,v) (center pixel)
19:     Let 𝒩 ← Neighbors(Q,u,v)        ▷ values of all neighboring pixels
20:     if q_c > q_i for all q_i ∈ 𝒩 then
21:         return true
22:     else
23:         return false.

24: CleanUpNeighbors(Corners)    ▷ Corners is sorted by descending q
25:     Create an empty list GoodCorners ← {}
26:     while Corners is not empty do
27:         c_i = (u_i, v_i, q_i) ← RemoveFirst(Corners)
28:         Add c_i to GoodCorners
29:         Delete all nodes c_j from Corners if Dist(c_i, c_j) < d_min
30:     return GoodCorners.
```

8.3.1 Schritt 1 – Berechnung der *corner response function*

Um die positiven und negativen Werte der in diesem Schritt verwendeten Filter handhaben zu können, werden für die Zwischenergebnisse Gleitkommabilder verwendet, die auch die notwendige Dynamik und Präzision bei kleinen Werten sicherstellen. Die Kerne für die benötigten Filter, also das Filter für die Vorglättung H_p, die Gradientenfilter H_{dx}, H_{dy} und das Glättungsfilter für die Komponenten der Strukturmatrix H_b, sind als eindimensionale `float`-Arrays definiert:

Abbildung 8.1
Harris-Detektor – konkrete Parameterwerte. Die angegebenen Zeilennummern beziehen sich auf Alg. 8.1.

Pre-Filter (Zeile 2): Vorglättung mit einem kleinen, xy-separierbaren Filter $H_p = H_{px} * H_{py}$, wobei

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

Gradientenfilter (Zeile 4): Berechnung der partiellen ersten Ableitungen in x- und y-Richtung mit

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

Blur-Filter (Zeile 6): Glättung der einzelnen Komponenten der Strukturmatrix M mit separierbaren Gauß-Filtern $H_b = H_{bx} * H_{by}$, mit

$$H_{bx} = \frac{1}{64}\begin{bmatrix}1 & 6 & 15 & \mathbf{20} & 15 & 6 & 1\end{bmatrix} \quad \text{und} \quad H_{by} = H_{bx}^T = \frac{1}{64}\begin{bmatrix}1\\ 6\\ 15\\ \mathbf{20}\\ 15\\ 6\\ 1\end{bmatrix}$$

Steuerparameter (Zeile 7): $\alpha = 0.04 \ldots 0.06$ (default 0.05)
Response-Schwellwert (Zeile 13): $t_H = 10.000 \ldots 1.000.000$ (default 25.000)
Umgebungsradius (Zeile 29): $d_{\min} = 10$ pixels

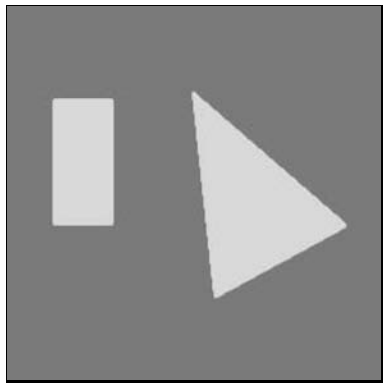
```
1 float[] pfilt = {0.223755f,0.552490f,0.223755f}; // Hp
2 float[] dfilt = {0.453014f,0.0f,-0.453014f}; // Hdx, Hdy
3 float[] bfilt = {0.01563f,0.09375f,0.234375f,0.3125f,
4                  0.234375f,0.09375f,0.01563f}; // Hb
```

Aus dem 8-Bit-Originalbild (vom Typ `ByteProcessor`) werden zunächst zwei Kopien `Ix` and `Iy` vom Typ `FloatProcessor` angelegt:

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

Als erster Verarbeitungsschritt wird eine Vorglättung mit dem Filter H_p durchgeführt (Alg. 8.1, Zeile 2), anschließend wird mit den Gradientenfiltern H_{dx} bzw. H_{dy} die horizontale und vertikale Ableitung berechnet (Alg. 8.1, Zeile 4). Da jeweils nur eindimensionale Filter derselben Richtung beteiligt sind, können die beiden Vorgänge in einem gemeinsamen Schritt durchgeführt werden:

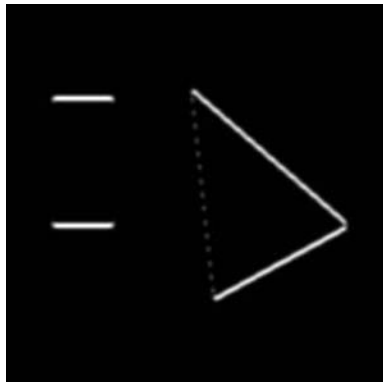
```
7 Ix = convolve1h(convolve1h(Ix,pfilt),dfilt);
8 Iy = convolve1v(convolve1v(Iy,pfilt),dfilt);
```



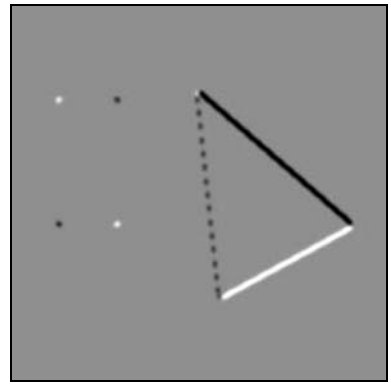
$I(u,v)$



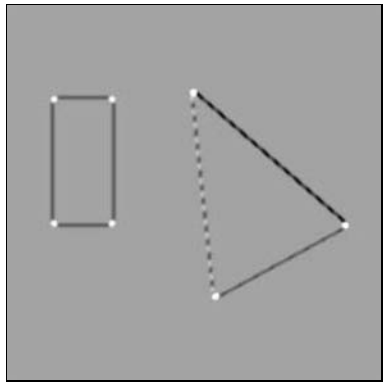
$A = I_x^2(u,v)$



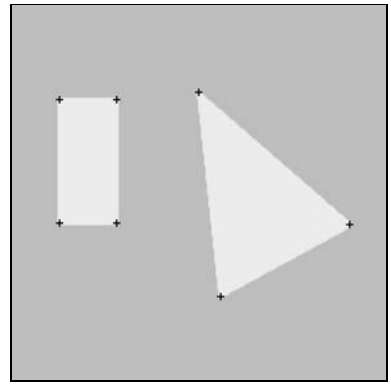
$B = I_y^2(u,v)$



$C = I_x I_y(u,v)$



$Q(u,v)$

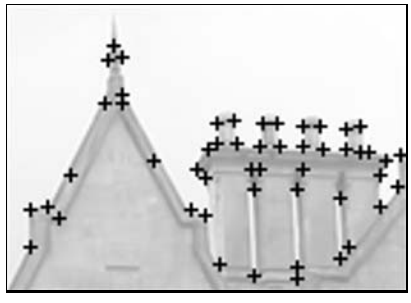


Eckpunkte

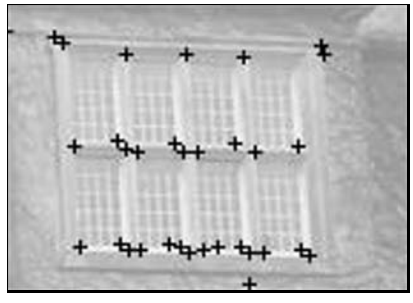
Abbildung 8.2
Harris-Detektor – Beispiel 1. Aus dem Originalbild $I(u,v)$ werden zunächst die ersten Ableitungen und daraus die Komponenten der Strukturmatrix $A = I_x^2$, $B = I_y^2$, $C = I_x I_y$ berechnet. Deutlich ist zu erkennen, dass A und B die horizontale bzw. vertikale Kantenstärke repräsentieren. In C werden die Werte nur dann stark positiv (weiß) oder stark negativ (schwarz), wenn beide Kantenrichtungen stark sind (Nullwerte sind grau dargestellt). Die *corner response function* Q zeigt markante, positive Spitzen an den Positionen der Eckpunkte. Die endgültigen Eckpunkte werden durch eine Schwellwertoperation und Auffinden der lokalen Maxima in Q bestimmt.



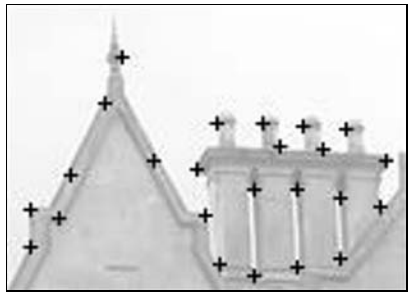
(a)



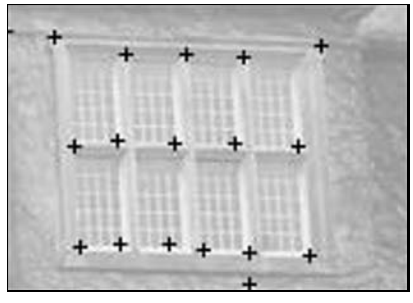
(b)



(c)



(d)



(e)

Abbildung 8.3
Harris-Detektor – Beispiel 2. Vollständiges Ergebnis mit markierten Eckpunkten (a). Nach Auswahl der stärksten Eckpunkte innerhalb eines Radius von 10 Pixel verbleiben von den ursprünglich gefundenen 615 Kandidaten noch 335 finale Eckpunkte. Details *vor* (b–c) und *nach* der Auswahl (d–e).

Dabei führen die Methoden `convolve1h(`p, h`)` und `convolve1v(`p, h`)` eindimensionale Filteroperationen in horizontaler bzw. vertikaler Richtung durch (s. unten „Filtermethoden“). Nun werden die Komponenten A, B, C der Strukturmatrix berechnet und anschließend jeweils mit dem separierbaren 2D-Filter H_b (`bfilt`) geglättet:

```
 9 A = sqr ((FloatProcessor) Ix.duplicate());
10 B = sqr ((FloatProcessor) Iy.duplicate());
11 C = mult((FloatProcessor) Ix.duplicate(),Iy);
12
13 A = convolve2(A,bfilt);    // convolve with Hb
14 B = convolve2(B,bfilt);
15 C = convolve2(C,bfilt);
```

Die Variablen `A`, `B`, `C` vom Typ `FloatProcessor` sind dabei in der Klasse `HarrisCornerDet` deklariert. Die Methode `convolve2(`I, h`)` führt eine separierbare 2D-Faltung mit dem 1D Filterkern h auf das Bild I aus (s. unten). `sqr()` und `mult()` sind Hilfsmethoden für das Quadrat bzw. die Quadratwurzel (Details in Anhang 4.1).

Die *corner response function* (Alg. 8.1, Zeile 7) wird schließlich durch die Methode `makeCrf()` als neues Bild vom Typ `FloatProcessor` berechnet:

```
16 void makeCrf() { // defined in class HarrisCornerDet
17   int w = ipOrig.getWidth();
18   int h = ipOrig.getHeight();
19   Q = new FloatProcessor(w,h);
20   float[] Apix = (float[]) A.getPixels();
21   float[] Bpix = (float[]) B.getPixels();
22   float[] Cpix = (float[]) C.getPixels();
23   float[] Qpix = (float[]) Q.getPixels();
24   for (int v=0; v<h; v++) {
25     for (int u=0; u<w; u++) {
26       int i = v*w+u;
27       float a = Apix[i], b = Bpix[i], c = Cpix[i];
28       float det = a*b-c*c;          // det(M̄)
29       float trace = a+b;            // trace(M̄)
30       Qpix[i] = det - alpha * (trace * trace);
31     }
32   }
33 }
```

Filtermethoden

Die oben verwendeten Filtermethoden benutzen die ImageJ-Klasse `Convolver` (definiert in ij.plugin.filter.*) für die eigentlichen Filteroperationen. Diese statischen Methoden sind in der Klasse `HarrisCornerDet` (Anhang 4.1) wie folgt definiert:

```
34 static FloatProcessor convolve1h (FloatProcessor p, float[] h)
       {
35   Convolver conv = new Convolver();
36   conv.setNormalize(false);
37   conv.convolve(p, h, 1, h.length);
38   return p; }
39
40 static FloatProcessor convolve1v (FloatProcessor p, float[] h)
       {
41   Convolver conv = new Convolver();
42   conv.setNormalize(false);
43   conv.convolve(p, h, h.length, 1);
44   return p; }
45
46 static FloatProcessor convolve2 (FloatProcessor p, float[] h)
      {
47   convolve1h(p,h);
48   convolve1v(p,h);
49   return p; }
```

8.3.2 Schritt 2 – Bestimmung der Eckpunkte

Das Ergebnis des ersten Schritts in Alg. 8.1 ist die *corner response function* $Q(u, v)$, die in unserer Implementierung als Gleitkommabild (`FloatProcessor`) definiert ist. Im zweiten Schritt werden aus Q die dominanten Eckpunkte ausgewählt. Dazu benötigen wir (a) einen Objekttyp zur Beschreibung der Eckpunkte und (b) einen flexiblen Container zur Aufbewahrung dieser Objekte, deren Zahl zunächst ja nicht bekannt ist.

Die `Corner`-Klasse

Zunächst definieren wir eine neue Klasse für die Repräsentation einzelner Eckpunkte $\boldsymbol{c} = (u, v, q)$ sowie eine zugehörige Konstruktor-Methode zum Erzeugen von Objekten der Klasse `Corner` aus den drei Argumenten u, v und q:

```
50 public class Corner implements Comparable {
51   int u;   //x-position
52   int v;   //y-postion
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 }
```

Die Klasse `Corner` implementiert bewusst das Java `Comparable`-Interface, damit `Corner`-Objekte miteinander vergleichbar und in der Folge sortierbar sind.

Auswahl eines Containers

In Alg. 8.1 haben wir die Notation von Listen (*lists*) und Mengen (*sets*) für Sammlungen von mehreren Eckpunkten verwendet. Würden wir diese als *Arrays* implementieren, müssten wir sie ziemlich groß anlegen, um alle gefundenen Eckpunkte in jedem Fall aufnehmen zu können. Stattdessen verwenden wir die Klasse `Vector`, eine der dynamischen Datenstrukturen, die Java in seinem *Collections Framework* (Package `java.util.*`) bereits fertig zur Verfügung stellt.

Ein `Vector` ist ähnlich einem Array, kann aber automatisch seine Kapazität erhöhen falls notwendig. Auf einzelne Elemente in einem `Vector` kann genauso wie in einem Array per Index zugegriffen werden und zudem implementiert die Klasse `Vector` das Java `List`-Interface, das eine Reihe weiterer Zugriffsmethoden bietet. Alternativ hätten wir auch die Klasse `ArrayList` als Container verwenden können, die sich von `Vector` nur geringfügig unterscheidet.

Die `collectCorners()`-Methode

Die nachfolgende Methode `collectCorners()` bestimmt aus der *corner response function* $Q(u, v)$ die dominanten Eckpunkte. Der Parameter *border* spezifiziert dabei die Breite des Bildrands, innerhalb dessen eventuelle Eckpunkte ignoriert werden sollen:

```
61 Vector<Corner> collectCorners(FloatProcessor Q, int border)
62   {
63   Vector<Corner> cornerList = new Vector<Corner>(1000);
64   int w = Q.getWidth(), h = Q.getHeight();
65   float[] Qpix = (float[]) Q.getPixels();
66   //traverse the Q-image and check for corners:
67   for (int v=border; v<h-border; v++){
68     for (int u=border; u<w-border; u++) {
69       float q = Qpix[v*w+u];
70       if (q>threshold && isLocalMax(crf,u,v)) {
71         Corner c = new Corner(u,v,q);
72         cornerList.add(c);
73       }
74     }
75   }
76   Collections.sort(cornerList);
77   return cornerList;
78 }
```

Zunächst wird (in Zeile 63) eine Variable `cornerList` vom Typ `Vector` mit einer Anfangskapazität von 1000 Objekten angelegt. Dann wird

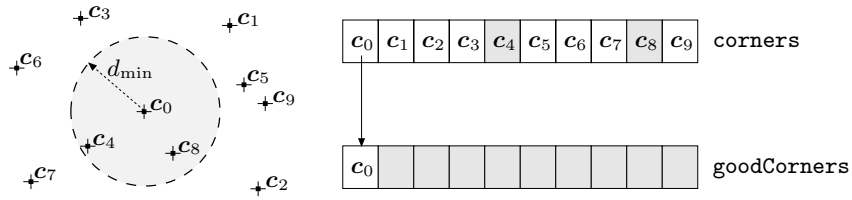


Abbildung 8.4
Auswahl der stärksten Eckpunkte innerhalb einer bestimmten Umgebung. Die ursprüngliche Liste von Eckpunkten (`corners`) ist nach „corner strength“ q in absteigender Reihenfolge sortiert, $\boldsymbol{c}_0$ ist also der stärkste Eckpunkt. Als Erstes wird $\boldsymbol{c}_0$ zur neuen Liste `goodCorners` angefügt und die Eckpunkte $\boldsymbol{c}_4$ und $\boldsymbol{c}_8$, die sich innerhalb der Distanz $d_{\min}$ von $\boldsymbol{c}_0$ befinden, werden aus `corners` gelöscht. In gleicher Weise wird als nächster Eckpunkt $\boldsymbol{c}_1$ behandelt usw., bis in `corners` keine Elemente mehr übrig sind. Keiner der verbleibenden Eckpunkte in `goodCorners` ist dadurch näher zu einem anderen Eckpunkt als $d_{\min}$.

das Bild `Q` durchlaufen und sobald eine Position als Eckpunkt in Frage kommt, wird ein neues `Corner`-Objekt erzeugt und zu `cornerList` angefügt (Zeile 72). Die Boole'sche Methode `isLocalMax(`Q, u, v`)`, die in der Klasse `HarrisCornerDet` definiert ist, stellt fest, ob $Q(u, v)$ an der Position u, v ein lokales Maximum aufweist (s. Definition in Anhang 4.1).

Abschließend werden (in Zeile 76) die Eckpunkte in `cornerList` durch Aufruf der Methode `sort()` (eine statische Methode der Klasse java.util.Collections) nach ihrer Stärke sortiert. Um das zu ermöglichen, muss die Klasse `Corner` wie erwähnt das Java `Comparable`-Interface implementieren, also auch eine `compareTo()`-Methode zur Verfügung stellen. Da wir die Eckpunkte in absteigender Reihenfolge nach ihrem jeweiligen q-Wert sortieren wollen, definieren wir diese Methode in der Klasse `Corner` folgendermaßen:

```
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 }
```

Aufräumen

Der abschließende Schritt ist das Beseitigen der schwächeren Eckpunkte in einem Umkreis bestimmter Größe, spezifiziert durch den Radius $d_{\min}$ (Alg. 8.1, Zeile 24–30). Dieser Vorgang ist in Abb. 8.4 skizziert und in der nachfolgenden Methode `cleanupCorners()` implementiert. Der bereits nach q sortierte `Vector corners` wird zunächst in ein gewöhnliches Array konvertiert (Zeile 90), das dann von Anfang bis Ende durchlaufen wird:

```
85 Vector<Corner> cleanupCorners(Vector<Corner> corners,
86           double dmin) {
87   //corners is sorted by q in descending order
88   double dmin2 = dmin*dmin;
89   Corner[] cornerArray = new Corner[corners.size()];
90   cornerArray = corners.toArray(cornerArray);
91   Vector<Corner> goodCorners
92        = new Vector<Corner>(corners.size());
```

```
93   for (int i=0; i<cornerArray.length; i++){
94     if (cornerArray[i] != null){
95       Corner c1 = cornerArray[i];
96       goodCorners.add(c1);
97       //remove all remaining corners close to c1
98       for (int j=i+1; j<cornerArray.length; j++){
99         if (cornerArray[j] != null){
100          Corner c2 = cornerArray[j];
101          if (c1.dist2(c2)<dmin2)    //compare squared distance
102            cornerArray[j] = null;   //remove corner c2
103        }
104      }
105    }
106  }
107  return goodCorners;
108 }
```

Jeweils nachfolgende Eckpunkte innerhalb der $d_{\min}$-Umgebung eines stärkeren Eckpunkts werden gelöscht (Zeile 102) und nur die verbleibenden Eckpunkte werden in die neue Liste `goodCorners` (die wieder als `Vector` realisiert ist) übernommen. Der Methodenaufruf `c1.dist2(c2)` in Zeile 101 liefert das Quadrat der Distanz $d^2 = (u_1 - u_2)^2 + (v_1 - v_2)^2$ zwischen den Eckpunkten `c1` und `c2`. Durch Verwendung des Quadrats der Distanzen wird die Wurzelfunktion vermieden.

8.3.3 Anzeigen der Eckpunkte

Zur Anzeige der gefundenen Eckpunkte werden an den entsprechenden Positionen im Originalbild Markierungen angebracht. Die nachfolgende Methode `showCornerPoints()` (definiert in der Klasse `HarrisCornerDet`) erzeugt zunächst eine Kopie des Originalbilds `ip` und erhöht dann mithilfe einer Lookup-Table die Gesamthelligkeit auf den Intensitätsbereich 128 . . . 255, bei gleichzeitiger Halbierung des Kontrasts (Zeilen 112–116). Dann wird die Liste `corners` durchlaufen und jedes `Corner`-Objekt „zeichnet sich selbst“ durch Aufruf der Methode `draw()` in das Ergebnisbild `ipResult` (Zeile 121):

```
109 ImageProcessor showCornerPoints(ImageProcessor ip){
110   ByteProcessor ipResult = (ByteProcessor)ip.duplicate();
111   //change background image contrast and brightness
112   int[] lookupTable = new int[256];
113   for (int i=0; i<256; i++){
114     lookupTable[i] = 128 + (i/2);
115   }
116   ipResult.applyTable(lookupTable);
117   //draw corners:
118   Iterator<Corner> it = corners.iterator();
119   for (int i=0; it.hasNext(); i++){
120     Corner c = it.next();
```

```
121     c.draw(ipResult);
122   }
123   return ipResult;
124 }
```

Die `draw()`-Methode selbst ist in der Klasse `Corner` definiert und zeichnet nur ein Kreuz fixer Größe an der Position des Eckpunkts (u, v):

```
125 void draw(ByteProcessor ip){ //defined in class Corner
126   //draw this corner as a black cross
127   int paintvalue = 0;       // set draw value to black
128   int size = 2;             // set size of cross marker
129   ip.setValue(paintvalue);
130   ip.drawLine(u-size,v,u+size,v);
131   ip.drawLine(u,v-size,u,v+size);
132 }
```

8.3.4 Zusammenfassung

Die meisten der oben beschriebenen Schritte dieser Implementierung sind in der Methode `findCorners()` zusammengefasst, die folgendermaßen aussieht:

```
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 }
```

Die eigentliche `run()`-Methode des zugehörigen Plugin `FindCorners_` reduziert sich damit auf nur wenige Zeilen. Sie erzeugt nur ein neues Objekt der Klasse `HarrisCornerDet`, wendet darauf die Methode `findCorners()` an und gibt die Ergebnisse in einem neuen Fenster aus:

```
139 public void run(ImageProcessor ip) {
140   HarrisCornerDet hcd = new HarrisCornerDet(ip);
141   hcd.findCorners();
142   ImageProcessor result = hcd.showCornerPoints(ip);
143   ImagePlus win = new ImagePlus("Corners",result);
144   win.show();
145 }
```

Der vollständige Quellcode zu diesem Abschnitt ist, wie bereits mehrfach erwähnt, in Anhang 4.1 verfügbar. Wie gewohnt sind die meisten dieser Codesegmente auf möglichst gute Verständlichkeit ausgelegt und nicht unbedingt auf Geschwindigkeit oder Speichereffizienz. Viele Details können daher (auch als Übung) mit relativ geringem Aufwand optimiert werden, sofern Effizienz ein wichtiges Thema ist.

8.4 Aufgaben

Aufg. 8.1. Adaptieren Sie die `draw()`-Methode in der Klasse `Corner` (S. 148), sodass auch die Stärke (q-Werte) der Eckpunkte grafisch dargestellt werden, z. B. durch die Größe der angezeigten Markierung.

Aufg. 8.2. Untersuchen Sie das Verhalten des Harris-Detektors bei Änderungen des Bildkontrasts und entwickeln Sie eine Idee, wie man den Parameter t_H automatisch an den Bildinhalt anpassen könnte.

Aufg. 8.3. Testen Sie die Zuverlässigkeit des Harris-Detektors bei Rotation und Verzerrung des Bilds. Stellen Sie fest, ob der Operator tatsächlich isotrop arbeitet.

Aufg. 8.4. Testen Sie das Verhalten des Harris-Detektors, vor allem in Bezug auf Positionierungsgenauigkeit und fehlende Eckpunkte, in Abhängigkeit vom Bildrauschen.

9

Detektion einfacher Kurven

In Kap. 7 haben wir gezeigt, wie man mithilfe von geeigneten Filtern Kanten finden kann, indem man an jeder Bildposition die Kantenstärke und möglicherweise auch die Orientierung der Kante bestimmt. Der darauf folgende Schritt bestand in der Entscheidung (z. B. durch Anwendung einer Schwellwertoperation auf die Kantenstärke), ob an einer Bildposition ein Kantenpunkt vorliegt oder nicht, mit einem binären Kantenbild (*edge map*) als Ergebnis. Das ist eine sehr frühe Festlegung, denn natürlich kann aus der beschränkten („myopischen“) Sicht eines Kantenfilters nicht zuverlässig ermittelt werden, ob sich ein Punkt tatsächlich auf einer Kante befindet oder nicht. Man muss daher davon ausgehen, dass in dem auf diese Weise produzierten Kantenbild viele vermeintliche Kantenpunkte markiert sind, die in Wirklichkeit zu keiner echten Kante gehören, und andererseits echte Kantenpunkte fehlen. Kantenbilder enthalten daher in der Regel zahlreiche irrelevante Strukturen und gleichzeitig sind wichtige Strukturen häufig unvollständig. Das Thema dieses Kapitels ist es, in einem vorläufigen, binären Kantenbild auffällige und möglicherweise bedeutsame Strukturen aufgrund ihrer Form zu finden.

9.1 Auffällige Strukturen

Ein intuitiver Ansatz zum Auffinden größerer Bildstrukturen könnte darin bestehen, beginnend bei einem beliebigen Kantenpunkt benachbarte Kantenpixel schrittweise aneinander zu fügen und damit die Konturen von Objekten zu bestimmen. Das könnte man sowohl im kontinuierlichen Kantenbild (mit Kantenstärke und Orientierung) als auch im binären *edge map* versuchen. In beiden Fällen ist aufgrund von Unterbrechungen, Verzweigungen und ähnlichen Mehrdeutigkeiten mit Problemen zu rechnen und ohne zusätzliche Kriterien und Informationen über

die Art der gesuchten Objekte bestehen nur geringe Aussichten auf Erfolg. Das lokale, sequentielle Verfolgen von Konturen (*contour tracing*) ist daher ein interessantes Optimierungsproblem [49] (s. auch Kap. 11.2).

Eine völlig andere Idee ist die Suche nach global auffälligen Strukturen, die von vornherein gewissen Formeigenschaften entsprechen. Wie das Beispiel in Abb. 9.1 zeigt, sind für das menschliche Auge derar-

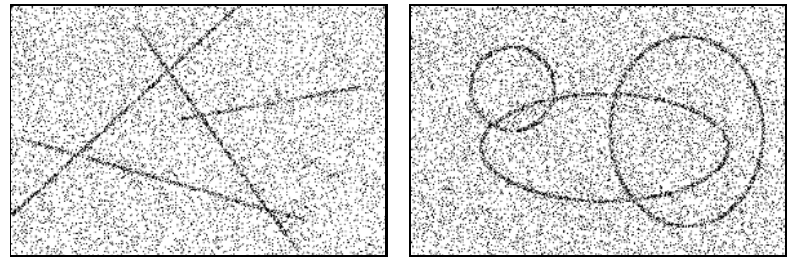


Abbildung 9.1
Das menschliche Sehsystem findet auffällige Bildstrukturen spontan auch unter schwierigen Bedingungen.

tige Strukturen auch dann auffällig, wenn keine zusammenhängenden Konturen gegeben sind, Überkreuzungen vorliegen und viele zusätzliche Elemente das Bild beeinträchtigen. Es ist auch heute weitgehend unbekannt, welche Mechanismen im biologischen Sehen für dieses spontane Zusammenfügen und Erkennen unter derartigen Bedingungen verantwortlich sind. Eine Technik, die zumindest eine vage Vorstellung davon gibt, wie derartige Aufgabenstellungen mit dem Computer möglicherweise zu lösen sind, ist die so genannte „Hough-Transformation“, die wir nachfolgend näher betrachten.

9.2 Hough-Transformation

Die Methode von Paul Hough – ursprünglich als US-Patent [38] publiziert und oft als „Hough-Transformation“ (HT) bezeichnet – ist ein allgemeiner Ansatz, um beliebige, parametrisierbare Formen in Punktverteilungen zu lokalisieren [21, 42]. Zum Beispiel können viele geometrische Formen wie Geraden, Kreise und Ellipsen mit einigen wenigen Parametern beschrieben werden. Da sich gerade diese Formen besonders häufig im Zusammenhang mit künstlichen, von Menschenhand geschaffenen Objekten finden, sind sie für die Analyse von Bildern besonders interessant (Abb. 9.2).

Betrachten wir zunächst den Einsatz der HT zur Detektion von Geraden in binären Kantenbildern, eine relativ häufige Anwendung. Eine Gerade in 2D kann bekanntlich mit zwei reellwertigen Parametern beschrieben werden, z. B. in der klassischen Form

$$y = kx + d, \tag{9.1}$$



Abbildung 9.2
Einfache geometrische Formen, wie gerade, kreisförmige oder elliptische Segmente, erscheinen häufig im Zusammenhang mit künstlichen bzw. technischen Objekten.

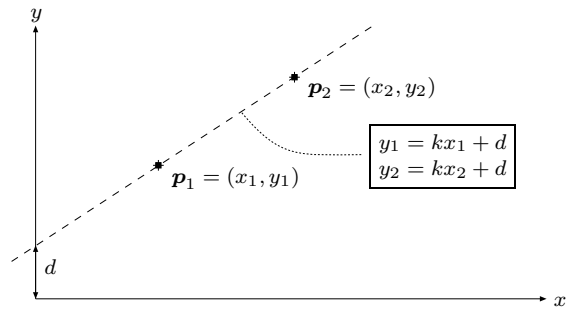


Abbildung 9.3
Zwei Bildpunkte $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ liegen auf derselben Geraden, wenn $y_1 = kx_1 + d$ und $y_2 = kx_2 + d$ für ein bestimmtes k und d.

wobei k die Steigung und d die Höhe des Schnittpunkts mit der y-Achse bezeichnet (Abb. 9.3). Eine Gerade, die durch zwei gegebene (Kanten-)Punkte $\boldsymbol{p}_1 = (x_1, y_1)$ und $\boldsymbol{p}_2 = (x_2, y_2)$ läuft, muss daher folgende beiden Gleichungen erfüllen:

$$y_1 = kx_1 + d \quad \text{und} \quad y_2 = kx_2 + d \tag{9.2}$$

für $k, d \in \mathbb{R}$. Das Ziel ist nun, jene Geradenparameter k und d zu finden, auf denen möglichst *viele* Kantenpunkte liegen, bzw. jene Geraden, die möglichst viele dieser Punkte „erklären“. Wie kann man aber feststellen, wie viele Punkte auf einer bestimmten Geraden liegen? Eine Möglichkeit wäre etwa, alle möglichen Geraden in das Bild zu „zeichnen“ und die Bildpunkte zu zählen, die jeweils exakt auf einer bestimmten Geraden liegen. Das ist zwar grundsätzlich möglich, wegen der großen Zahl an Geraden aber nicht besonders effizient.

9.2.1 Parameterraum

Die Hough-Transformation geht an dieses Problem auf dem umgekehrten Weg heran, indem sie alle möglichen Geraden ermittelt, die durch einen

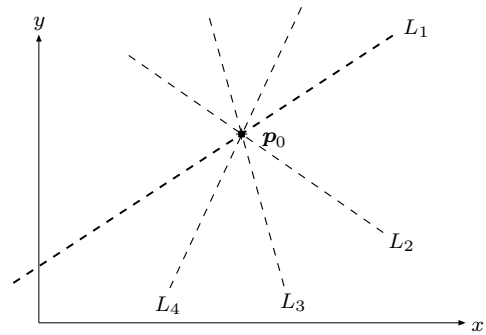


Abbildung 9.4
Geradenbündel durch einen Bildpunkt. Für alle möglichen Geraden L_j durch den Punkt $\boldsymbol{p}_0 = (x_0, y_0)$ gilt $y_0 = k_j x_0 + d_j$ für geeignete Parameter k_j, d_j.

einzelnen, gegebenen Bildpunkt laufen. Jede Gerade L_j, die durch einen Punkt $\boldsymbol{p}_0 = (x_0, y_0)$ läuft, muss die Gleichung

$$L_j : y_0 = k_j x_0 + d_j \tag{9.3}$$

für geeignete Werte von k_j, d_j erfüllen. Die Menge der Lösungen für k_j, d_j in Gl. 9.3 entspricht einem Bündel von unendlich vielen Geraden, die alle durch den gegebenen Punkt $\boldsymbol{p}_0$ laufen (Abb. 9.4). Für ein bestimmtes k_j ergibt sich die zugehörige Lösung für d_j aus Gl. 9.3 als

$$d_j = -x_0 k_j + y_0 \,, \tag{9.4}$$

also wiederum eine lineare Funktion (Gerade), wobei nun k_j, d_j die *Variablen* und x_0, y_0 die (konstanten) *Parameter* der Funktion sind. Die Lösungsmenge $\{(k_j, d_j)\}$ von Gl. 9.4 beschreibt die Parameter aller möglichen Geraden L_j, die durch einen Bildpunkt $\boldsymbol{p}_0 = (x_0, y_0)$ führen.

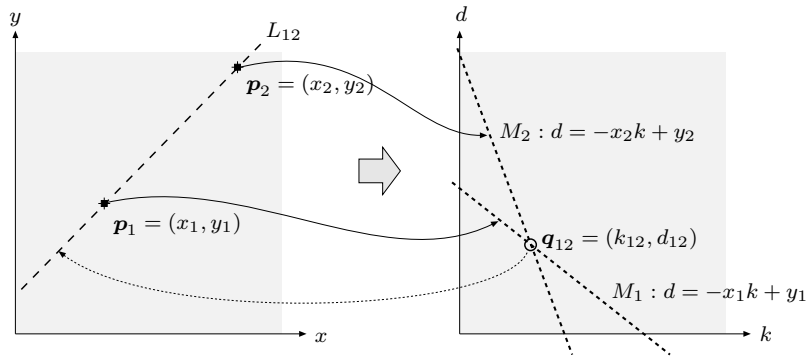
Für einen *beliebigen* Bildpunkt $\boldsymbol{p}_i = (x_i, y_i)$ entspricht Gl. 9.4 einer Geraden

$$M_i : d = -x_i k + y_i \tag{9.5}$$

mit den Parametern $-x_i, y_i$ im so genannten *Parameterraum* (auch „Hough-Raum“ genannt), der durch die Koordinaten k, d aufgespannt wird. Der Zusammenhang zwischen dem Bildraum und dem Parameterraum lässt sich folgendermaßen zusammenfassen:

Bildraum (x, y)		*Parameterraum* (k, d)	
Punkt	$\boldsymbol{p}_i = (x_i, y_i)$	$M_i : d = -x_i k + y_i$	Gerade
Gerade	$L_j : y = k_j x + d_j$	$\boldsymbol{q}_j = (k_j, d_j)$	Punkt

Jedem Punkt $\boldsymbol{p}_i$ und seinem zugehörigen Geradenbüschel im Bildraum entspricht also exakt eine Gerade M_i im Parameterraum. Am meisten sind wir jedoch an jenen Stellen interessiert, an denen sich Geraden im Parameterraum *schneiden*. Wie am Beispiel in Abb. 9.5 gezeigt, schneiden sich die Geraden M_1 und M_2 an der Position $\boldsymbol{q}_{12} = (k_{12}, d_{12})$ im Parameterraum, die den Parametern jener Geraden im Bildraum entspricht, die sowohl durch den Punkt $\boldsymbol{p}_1$ als auch durch den Punkt $\boldsymbol{p}_2$



(a) x/y-Bildraum (b) k/d-Parameterraum

Abbildung 9.5
Zusammenhang zwischen Bildraum und Parameterraum. Die Parameterwerte für alle möglichen Geraden durch den Bildpunkt $\boldsymbol{p}_i = (x_i, y_i)$ im Bildraum (a) liegen im Parameterraum (b) auf einer Geraden M_i. Umgekehrt entspricht jeder Punkt $\boldsymbol{q}_j = (k_j, d_j)$ im Parameterraum einer Geraden L_j im Bildraum. Der Schnittpunkt der zwei Geraden M_1, M_2 an der Stelle $\boldsymbol{q}_{12} = (k_{12}, d_{12})$ im Parameterraum zeigt an, dass im Bildraum eine Gerade L_{12} mit zwei Punkten und den Parametern k_{12} und d_{12} existiert.

verläuft. Je mehr Geraden M_i sich an einem Punkt im Parameterraum schneiden, umso mehr Bildpunkte liegen daher auf der entsprechenden Geraden im Bildraum! Allgemein ausgedrückt heißt das:

> Wenn sich N Geraden an einer Position (k', d') im *Parameterraum* schneiden, dann liegen auf der entsprechenden Geraden $y = k'x + d'$ im *Bildraum* insgesamt N Bildpunkte.

9.2.2 Akkumulator-Array

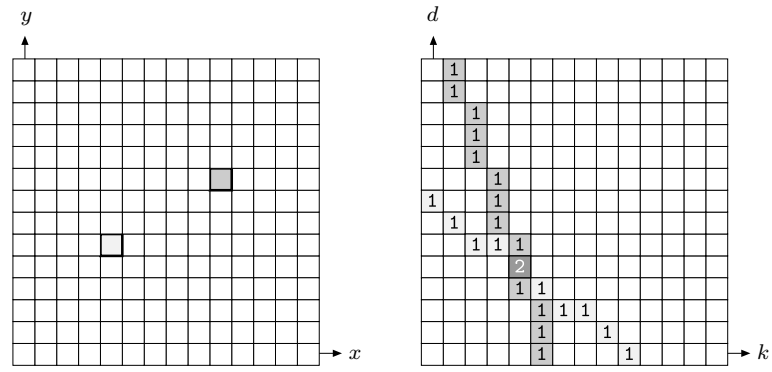
Das Ziel, die dominanten Bildgeraden zu finden, ist daher gleichbedeutend mit dem Auffinden jener Koordinaten im Parameterraum, an denen sich viele Parametergeraden schneiden. Genau das ist die Intention der HT. Um die HT zu berechnen, benötigen wir zunächst eine diskrete Darstellung des kontinuierlichen Parameterraums mit entsprechender Schrittweite für die Koordinaten k und d. Um die Anzahl der Überschneidungen im Parameterraum zu berechnen, wird jede Parametergerade M_i additiv in dieses „Akkumulator-Array“ gezeichnet, indem jede durchlaufene Zelle um den Wert 1 erhöht wird (Abb. 9.6).

9.2.3 Eine bessere Geradenparametrisierung

Leider ist die Geradenrepräsentation in Gl. 9.1 in der Praxis nicht wirklich brauchbar, denn es gilt $k = \infty$ für vertikale Geraden. Eine bessere Lösung ist die so genannte *Hesse'sche Normalform* (HNF) der Geradengleichung

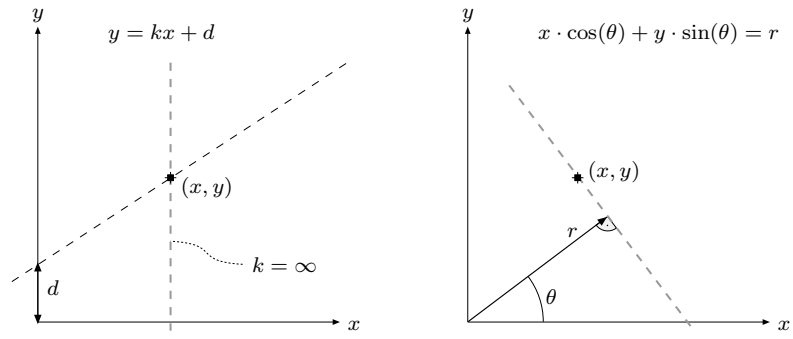
$$x \cdot \cos(\theta) + y \cdot \sin(\theta) = r\,, \tag{9.6}$$

die keine derartigen Singularitäten aufweist und außerdem eine lineare Quantisierung ihrer Parameter, des Winkels θ und des Radius r, ermöglicht (s. Abb. 9.7). Mit der HNF-Parametrisierung hat der Parameterraum die Koordinaten θ, r und jedem Bildpunkt $\boldsymbol{p}_i = (x_i, y_i)$ entspricht darin die Relation



(a) Bildraum (b) Akkumulator-Array

Abbildung 9.6
Kernidee der Hough-Transformation. Das Akkumulator-Array ist eine diskrete Repräsentation des Parameterraums (k, d). Für jeden gefundenen Bildpunkt (a) wird eine diskrete Gerade in den Parameterraum (b) gezeichnet. Diese Operation erfolgt *additiv*, d. h., jede durchlaufene Array-Zelle wird um den Wert 1 erhöht. Der Wert jeder Zelle des Akkumulator-Arrays entspricht der Anzahl von Parametergeraden, die sich dort schneiden (in diesem Fall 2).



(a) (b)

Abbildung 9.7
Parametrisierung von Geraden in 2D. In der üblichen k, d-Parametrisierung (a) ergibt sich bei vertikalen Geraden ein Problem, weil in diesem Fall $k = \infty$. Die Hesse'sche Normalform (b), bei der die Gerade durch den Winkel θ und den Abstand vom Ursprung r dargestellt wird, vermeidet dieses Problem.

$$r_{x_i,y_i}(\theta) = x_i \cdot \cos(\theta) + y_i \cdot \sin(\theta)\,, \tag{9.7}$$

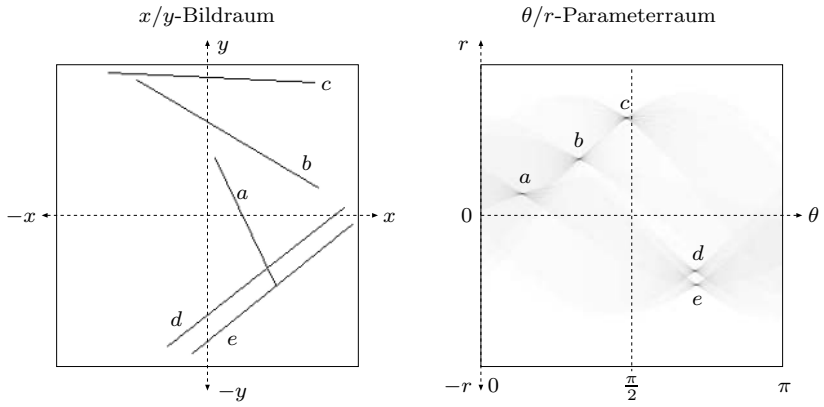
für den Winkelbereich $0 \le \theta < \pi$ (s. Abb. 9.8). Wenn wir das Zentrum des Bilds als Referenzpunkt für die x/y-Bildkoordinaten benutzen, dann ist der mögliche Bereich für den Radius auf die Hälfte der Bilddiagonale beschränkt, d. h.

$$-r_{\max} \le r_{x,y}(\theta) \le r_{\max}, \quad \text{wobei} \quad r_{\max} = \tfrac{1}{2}\sqrt{M^2 + N^2} \tag{9.8}$$

für ein Bild der Breite M und der Höhe N.

9.3 Implementierung der Hough-Transformation

Der grundlegende Hough-Algorithmus für die Geradenparametrisierung mit der HNF (Gl. 9.6) ist in Alg. 9.1 gezeigt. Ausgehend von einem binären Eingangsbild $I(u, v)$, das bereits markierte Kantenpixel (Wert 1) enthält, wird im ersten Schritt ein zweidimensionales Akkumulator-Array erzeugt und gefüllt. Im zweiten Schritt (FindMaxLines()) wird,



(a) (b)

Abbildung 9.8
Bildraum und Parameterraum für die HNF-Parametrisierung.

```
 1: HoughLines(I)
 2:     Set up a two-dimensional array Acc[θ, r] of counters, initialize to 0
 3:     Let (u_c, v_c) be the center coordinates of the image I
 4:     for all image coordinates (u, v) do
 5:         if I(u, v) is an edge point then
 6:             (x, y) ← (u−u_c, v−v_c)          ▷ relative coordinate to center
 7:             for θ_i = 0 ... π do
 8:                 r_i = x cos(θ_i) + y sin(θ_i)
 9:                 Increment Acc[θ_i, r_i]
10:     MaxLines ← FindMaxLines(Acc, K)
11:         ▷ return the list of parameter pairs (θ_j, r_j) for K strongest lines
12:     return MaxLines.
```

Algorithmus 9.1
Einfacher Hough-Algorithmus für Geraden. Dieser liefert eine Liste der Parameter (θ, r) für die K stärksten Geraden im binären Kantenbild $I(u, v)$.

wie nachfolgend beschrieben, das Akkumulator-Array nach maximalen Einträgen durchsucht und ein Vektor von Parameterwerten für die K stärksten Geraden

$$MaxLines = \big((\theta_1, r_1), (\theta_2, r_2), \ldots (\theta_K, r_K)\big)$$

wird zurückgegeben.

9.3.1 Füllen des Akkumulator-Arrays

Eine direkte Implementierung des ersten Teils von Alg. 9.1 zeigt Prog. 9.1, bestehend aus einer einzigen Java-Klasse `LinearHT`. Das Akkumulator-Array (`houghArray`) ist als zweidimensionales `int`-Array definiert. Aus dem ursprünglichen Bild `imp` wird die HT durch Erzeugen einer neuen Instanz der Klasse `LinearHT` berechnet, d. h.

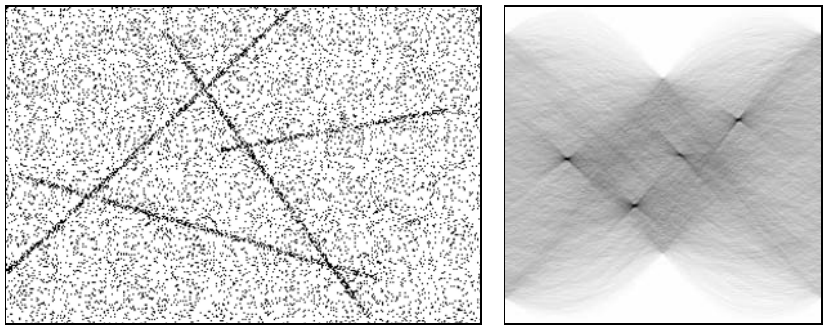
```
LinearHT HT = new LinearHT(imp, 256, 256);
```

Die letzten beiden Parameter (256, 256) spezifizieren die Anzahl der diskreten Schritte für den Winkel θ und den Radius r. `imp` wird als

Programm 9.1
Hough-Transformation für Geraden (Java-Implementierung).

```
1  class LinearHT {
2    ImageProcessor ip;  // reference to original image
3    int nAng, nRad;     // number of steps for angle and radius
4    double dAng, dRad;  // stepsize of angle and radius
5    int xCtr, yCtr;     // x/y-coordinate of image center
6    int[][] houghArray; // Hough accumulator
7
8    //constructor method:
9    LinearHT(ImageProcessor ip, int aSteps, int rSteps) {
10     this.ip = ip;
11     xCtr = ip.getWidth()/2; yCtr = ip.getHeight()/2;
12     nAng = aSteps; dAng = (Math.PI/nAng);
13     nRad = rSteps;
14     double rMax = Math.sqrt(xCtr*xCtr + yCtr*yCtr);
15     dRad = (2*rMax)/nRad;
16     houghArray = new int[nAng][nRad];
17     fillHoughAccumulator();
18   }
19
20   void fillHoughAccumulator() {
21     for (int v = 0; v < ip.getHeight(); v++) {
22       for (int u = 0; u < ip.getWidth(); u++) {
23         if (ip.getPixel(u, v) > 0) {
24           doPixel(u, v);
25         }
26       }
27     }
28   }
29
30   void doPixel(int u, int v) {
31     int x = u-xCtr, y = v-yCtr;
32     for (int a = 0; a < nAng; a++) {
33       double theta = dAng * a;
34       int r = (int) Math.round(
35         (x*Math.cos(theta) + y*Math.sin(theta)) / dRad) + nRad
               /2;
36       if (r >= 0 && r < nRad) {
37         houghArray[a][r]++;
38       }
39     }
40   }
41 }
```

`ByteProcessor` (d. h. als 8-Bit-Grauwertbild) angenommen und jeder Bildwert größer als null wird als Kantenpixel interpretiert. Die Anwendung dieses Programms auf ein stark verrauschtes Kantenbild ist in Abb. 9.9 gezeigt.



(a) (b)

Abbildung 9.9
Hough-Transformation für Geraden. Das Originalbild (a) hat eine Größe von 360 × 240 Pixel, womit sich ein maximaler Radius (Abstand vom Bildzentrum) $r_{\max} \approx 216$ ergibt. Im zugehörigen Parameterraum (b) wird eine Rasterung von jeweils 256 Schritten sowohl für den Winkel $\theta = 0 \ldots \pi$ (horizontale Achse) als auch für den Radius $r = -r_{\max} \ldots r_{\max}$ (vertikale Achse) verwendet. Die vier dunklen Maximalwerte im Akkumulator-Array entsprechen den Parametern der vier Geraden im Originalbild. Die Intensität wurde zur besseren Sichtbarkeit invertiert.

9.3.2 Auswertung des Akkumulator-Arrays

Der nächste Schritt ist die Lokalisierung der Maximalwerte im Akkumulator-Array $Acc[\theta, r]$. Wie in Abb. 9.9 (b) deutlich zu erkennen ist, schneiden sich auch beim Vorliegen von geometrisch exakten Bildgeraden die zugehörigen Akkumulator-Kurven nicht *genau* in einzelnen Zellen, sondern die Schnittpunkte sind über eine gewisse Umgebung verteilt. Die Ursache sind Rundungsfehler aufgrund der diskreten Koordinatengitter. Durch dieses Problem wird die Auswertung des Akkumulator-Arrays zum schwierigsten Teil der HT, und es gibt dafür auch keine Patentlösung. Zu den einfachen Möglichkeiten gehören folgende zwei Ansätze (s. Abb. 9.10):

Variante A: Schwelloperation

Zunächst unterziehen wir das Akkumulator-Array einer Schwellwertoperation mit dem Wert t_a und setzen dabei alle Akkumulator-Werte $Acc[\theta, r] < t_a$ auf null. Die verbleibenden Regionen (Abb. 9.10 (b)) könnte man z. B. mit einer morphologischen *Closing*-Operation (s. Abschn. 10.3.2) auf einfache Weise bereinigen. Als Nächstes lokalisieren wir die noch übrigen Regionen in $Acc[\theta, r]$ (z. B. mit einer der Techniken in Abschn. 11.1), berechnen die Schwerpunkte der Regionen (s. Abschn. 11.4.3) und verwenden deren (nicht ganzzahlige) Koordinaten als Parameter der gefundenen Geraden. Weiters ist die Summe der Akkumulator-Werte innerhalb einer Region ein guter Indikator für die Stärke (Anzahl der Bildpunkte) der Geraden.

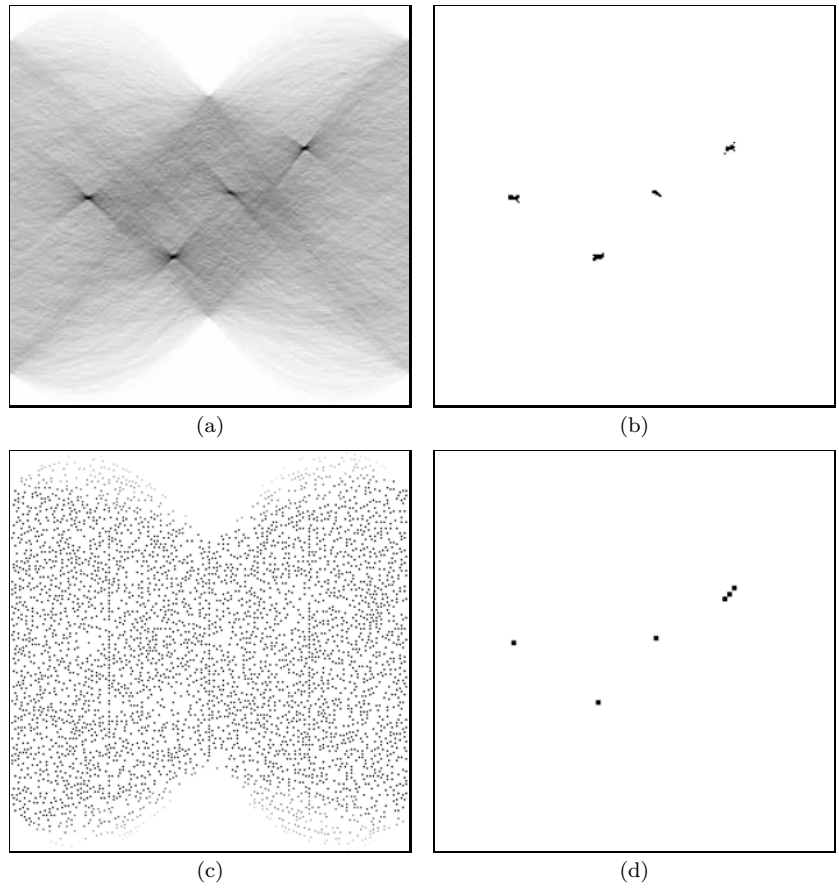


Abbildung 9.10
Bestimmung lokaler Maximalwerte im Akkumulator-Array. Ursprüngliche Verteilung der Werte im Hough-Akkumulator (a). **Variante A:** *Schwellwertoperation* mit 50% des Maximalwerts (b) – die verbleibenden Regionen entsprechen den vier dominanten Bildgeraden. Die Koordinaten der Schwerpunkte dieser Regionen ergeben eine gute Schätzung der echten Geradenparameter. **Variante B:** Durch *Non-Maximum Suppression* entsteht zunächst eine große Zahl lokaler Maxima (c), die durch eine anschließende Schwellwertoperation reduziert wird (d).

Variante B: Non-Maximum Suppression

Die Idee dieser Methode besteht im Auffinden lokaler Maxima im Akkumulator-Array durch Unterdrückung aller *nicht* maximalen Werte.[1] Dazu wird für jede Zelle in $Acc[\theta, r]$ festgestellt, ob ihr Wert höher ist als die Werte aller ihrer Nachbarzellen. Ist dies der Fall, dann wird der bestehende Wert beibehalten, ansonsten wird die Zelle auf null gesetzt (Abb. 9.10 (c)). Die (ganzzahligen) Koordinaten der verbleibenden Spitzen sind potentielle Geradenparameter und deren jeweilige Höhe entspricht der Stärke der Bildgeraden. Diese Methode kann natürlich mit einer Schwellwertoperation verbunden werden, um die Anzahl der Kandidatenpunkte einzuschränken. Das entsprechende Ergebnis zeigt Abb. 9.10 (d).

9.3.3 Erweiterungen der Hough-Transformation

Was wir bisher gesehen haben, ist nur die einfachste Form der Hough-Transformation. Für den praktischen Einsatz sind unzählige Verbesse-

[1] Non-Maximum Suppression wird auch in Abschn. 8.2.3 zur Isolierung von Eckpunkten verwendet.

rungen und Verfeinerungen möglich und oft auch unumgänglich. Hier eine kurze und keineswegs vollständige Liste von Möglichkeiten.

Bias

Da der Wert eine Zelle im Hough-Akkumulator der Anzahl der Bildpunkte auf der entsprechenden Geraden entspricht, können lange Geraden grundsätzlich höhere Werte als kurze Geraden erzielen. Zum Beispiel kann eine Gerade in der Nähe einer Bildecke nie dieselbe Anzahl von Treffern in ihrer Akkumulator-Zelle erreichen wie eine Gerade entlang der Bilddiagonalen (Abb. 9.11). Wenn wir daher im Ergebnis nur

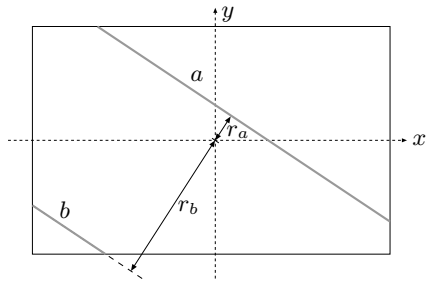


Abbildung 9.11
Bias-Problem. Innerhalb der endlichen Bildfläche sind Geraden mit einem kleinen Abstand r vom Zentrum i. Allg. länger als Geraden mit großem r. Zum Beispiel ist die maximal mögliche Zahl von Akkumulator-Treffern für Gerade a wesentlich höher als für Gerade b.

nach den maximalen Einträgen suchen, ist die Wahrscheinlichkeit hoch, dass kürzere Geraden überhaupt nicht gefunden werden. Eine Möglichkeit, diesen systematischen Fehler zu kompensieren, besteht darin, jeden Akkumulator-Eintrag $Acc[\theta, r]$ bezüglich der maximal möglichen Zahl von Bildpunkten $MaxHits[\theta, r]$ auf der Geraden mit den Parametern θ, r zu normalisieren:

$$Acc'[\theta, r] \leftarrow \frac{Acc[\theta, r]}{MaxHits[\theta, r]} \quad \text{für} \quad MaxHits[\theta, r] > 0. \tag{9.9}$$

$MaxHits[\theta, r]$ kann z. B. durch Berechnung der Hough-Transformation auf ein Bild mit den gleichen Dimensionen ermittelt werden, in dem alle Pixel aktiviert sind, oder durch ein zufälliges Bild, in dem die Pixel gleichförmig verteilt sind.

Endpunkte von Bildgeraden

Die einfache Version der Hough-Transformation liefert zwar die Parameter der Bildgeraden, nicht aber deren Endpunkte. Das nachträgliche Auffinden der Endpunkte bei gegebenen Geradenparametern ist nicht nur aufwendig, sondern reagiert auch empfindlich auf Diskretisierungs- bzw. Rundungsfehler. Eine Möglichkeit ist, die Koordinaten der Extrempunkte einer Geraden bereits innerhalb der Berechnung des Akkumulator-Arrays zu berücksichtigen. Dazu wird jede Akkumulator-Zelle durch zwei

zusätzliche Koordinatenpaare $(start_X, start_Y)$, (end_X, end_Y) ergänzt, d. h.

$$Acc[\theta, r] = (count, start_X, start_Y, end_X, end_Y).$$

Die Koordinaten für die beiden Endpunkte jeder Geraden werden beim Füllen des Akkumulator-Arrays mitgezogen, sodass sie am Ende des Vorgangs jeweils den am weitesten auseinander liegenden Endpunkten der Geraden entsprechen. Natürlich muss bei der Auswertung des Akkumulator-Arrays darauf geachtet werden, dass beim eventuellen Zusammenfügen von Akkumulator-Zellen in diesem Fall auch die Koordinaten der Endpunkte entsprechend berücksichtigt werden müssen.

Berücksichtigung von Kantenstärke und -orientierung

Die Ausgangsdaten für die Hough-Transformation ist üblicherweise ein Kantenbild, das wir bisher als binäres 0/1-Bild mit potentiellen Kantenpunkten betrachtet haben. Das ursprüngliche Ergebnis einer Kantendetektion enthält jedoch zusätzliche Informationen, die für die HT verwendet werden können, insbesondere die Kantenstärke $E(u, v)$ und die lokale Kantenrichtung $\Phi(u, v)$ (s. Abschn. 7.3).

Die *Kantenstärke* $E(u, v)$ ist besonders einfach zu berücksichtigen: Anstatt eine getroffene Akkumulator-Zelle nur um 1 zu erhöhen, wird der Wert der jeweiligen Kantenstärke addiert, d. h.

$$Acc[\theta, r] \leftarrow Acc[\theta, r] + E(u, v).$$

Mit anderen Worten, starke Kantenpunkte tragen auch mehr zum akkumulierten Zellwert bei als schwächere.

Die lokale *Kantenorientierung* $\Phi(u, v)$ ist ebenfalls hilfreich, denn sie schränkt den Bereich der möglichen Orientierungswinkel einer Geraden im Bildpunkt (u, v) ein. Die Anzahl der zu berechnenden Akkumulator-Zellen entlang der θ-Achse kann daher, abhängig von $\Phi(u, v)$, auf einen Teilbereich reduziert werden. Dadurch wird nicht nur die Effizienz des Verfahrens verbessert, sondern durch die Reduktion von irrelevanten „votes“ im Akkumulator auch die Trennschärfe der HT insgesamt erhöht (s. beispielsweise [47, S. 483]).

Hierarchische Hough-Transformation

Die Genauigkeit der Ergebnisse wächst mit der Größe des Parameterraums. Eine Größe von 256 entlang der θ-Achse bedeutet z. B. eine Schrittweite der Geradenrichtung von $\frac{\pi}{256} \approx 0.7°$. Eine Vergrößerung des Akkumulators führt zu feineren Ergebnissen, bedeutet aber auch zusätzliche Rechenzeit und insbesondere einen höheren Speicherbedarf. Die Idee der hierarchischen HT ist, schrittweise wie mit einem „Zoom“ den Parameterraum gezielt zu verfeinern. Zunächst werden mit einem relativ grob aufgelösten Parameterraum die wichtigsten Geraden gesucht,

dann wird der Parameterraum um die Ergebnisse herum mit höherer Auflösung „expandiert“ und die HT rekursiv wiederholt. Auf diese Weise kann trotz eines beschränkten Parameterraums eine relativ genaue Bestimmung der Parameter erreicht werden.

9.4 Hough-Transformation für Kreise und Ellipsen

9.4.1 Kreise und Kreisbögen

Geraden in 2D haben zwei Freiheitsgrade und sind daher mit zwei reellwertigen Parametern vollständig spezifiziert. Ein Kreis in 2D benötigt *drei* Parameter, z. B. in der Form

$$Circle = (\bar{x}, \bar{y}, \rho),$$

wobei $\bar{x}$, $\bar{y}$ die Koordinaten des Mittelpunkts und ρ den Kreisradius bezeichnen (Abb. 9.12). Ein Punkt $\boldsymbol{p} = (u, v)$ liegt auf einem Kreis, wenn

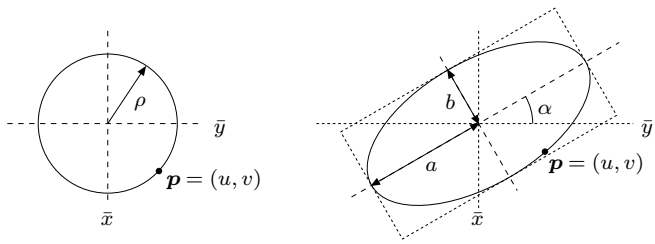


Abbildung 9.12
Parametrisierung von Kreisen und Ellipsen in 2D.

die Bedingung

$$(u - \bar{x})^2 + (v - \bar{y})^2 = \rho^2 \tag{9.10}$$

gilt. Wir benötigen daher für die Hough-Transformation einen dreidimensionalen Parameterraum $Acc[\bar{x}, \bar{y}, \rho]$, um Kreise (und Kreisbögen) mit beliebiger Position und Radius in einem Bild zu finden. Im Unterschied zur HT für Geraden besteht allerdings keine einfache, funktionale Abhängigkeit der Koordinaten im Parameterraum – wie kann man also jene Parameterkombinationen $(\bar{x}, \bar{y}, \rho)$ finden, die Gl. 9.10 für einen bestimmten Bildpunkt $\boldsymbol{p} = (u, v)$ erfüllen? Ein „brute force“-Ansatz wäre, schrittweise und exhaustiv alle Zellen des Parameterraums auf Gültigkeit der Relation in Gl. 9.10 zu testen, wie in Alg. 9.2 beschrieben.

Eine bessere Idee gibt uns Abb. 9.13, aus der wir sehen, dass die Koordinaten der passenden Mittelpunkte im Hough-Raum selbst wiederum Kreise bilden. Wir müssen daher für einen Bildpunkt $\boldsymbol{p} = (u, v)$ nicht den gesamten, dreidimensionalen Parameterraum durchsuchen, sondern brauchen nur in jeder ρ-Ebene des Akkumulator-Arrays die Zellen entlang eines entsprechenden Kreises zu erhöhen. Dazu lässt sich jeder Standardalgorithmus zum Generieren von Kreisen verwenden, z. B. eine Variante des bekannten *Bresenham*-Algorithmus [10].

Algorithmus 9.2
Exhaustiver Hough-Algorithmus zum Auffinden von Kreisen.

```
1:  HoughCircles(I)
2:      Set up a three-dimensional array Acc[x̄, ȳ, ρ] and initialize to 0
3:      for all image coordinates (u, v) do
4:          if I(u, v) is an edge point then
5:              for all (x̄_i, ȳ_i, ρ_i) in the accumulator space do
6:                  if (u − x̄_i)² + (v − ȳ_i)² = ρ_i² then
7:                      Increment Acc[x̄_i, ȳ_i, ρ_i]
8:      MaxCircles ← FindMaxCircles(Acc)   ▷ a list of tuples (x̄_j, ȳ_j, ρ_j)
9:      return MaxCircles.
```

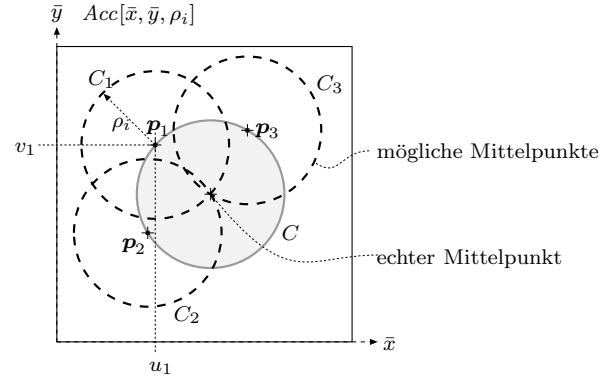


Abbildung 9.13
Hough-Transformation für Kreise. Die Abbildung zeigt eine Ebene des dreidimensionalen Akkumulator-Arrays $Acc[\bar{x}, \bar{y}, \rho]$ für einen bestimmten Kreisradius $\rho = \rho_i$. Die Mittelpunkte aller Kreise, die durch einen gegebenen Bildpunkt $\boldsymbol{p}_1 = (u_1, v_1)$ laufen, liegen selbst wieder auf einem Kreis C_1 (- - -) mit dem Radius ρ_i und dem Mittelpunkt $\boldsymbol{p}_1$. Analog dazu liegen die Mittelpunkte der Kreise, die durch $\boldsymbol{p}_2$ und $\boldsymbol{p}_3$ laufen, auf den Kreisen C_2 bzw. C_3. Die Zellen entlang der drei Kreise C_1, C_2, C_3 mit dem Radius ρ_i werden daher im Akkumulator-Array durchlaufen und erhöht. Die Kreise haben einen gemeinsamen Schnittpunkt im echten Mittelpunkt des Bildkreises C, wo drei „Treffer“ im Akkumulator-Array zu finden sind.

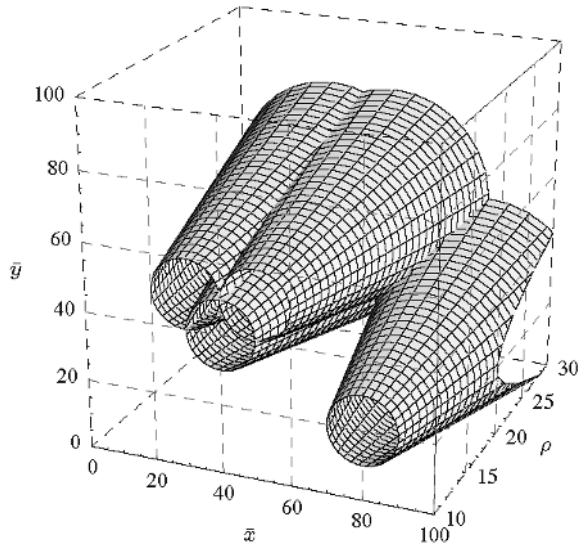
Abb. 9.14 zeigt die räumliche Struktur des dreidimensionalen Parameterraums für Kreise. Für einen Bildpunkt $\boldsymbol{p}_k = (u_k, v_k)$ wird in jeder Ebene entlang der ρ-Achse (für $\rho_i = \rho_{\min} \dots \rho_{\max}$) ein Kreis mit dem Mittelpunkt (u_k, v_k) und dem Radius ρ_i durchlaufen, d. h. entlang einer kegelförmigen Fläche. Die Parameter der dominanten Kreise findet man wiederum als Koordinaten der Akkumulator-Zellen mit den meisten „Treffern“, wo sich also die meisten Kegelflächen schneiden, wobei das *Bias*-Problem (s. Abschn. 9.3.3) natürlich auch hier zutrifft. Kreisbögen werden in gleicher Weise gefunden, allerdings ist die Anzahl der möglichen Treffer proportional zur Bogenlänge.

9.4.2 Ellipsen

In einer perspektivischen Abbildung erscheinen Kreise in der dreidimensionalen Realität in 2D-Abbildungen meist als Ellipsen, außer sie liegen auf der optischen Achse und werden frontal betrachtet. Exakt kreisförmige Strukturen sind daher in gewöhnlichen Fotografien relativ selten anzutreffen. Die Hough-Transformation funktioniert natürlich auch für Ellipsen, allerdings ist die Methode wegen des größeren Parameterraums wesentlich aufwendiger.

Eine allgemeine Ellipse in 2D hat 5 Freiheitsgrade und benötigt zu ihrer Beschreibung daher auch 5 Parameter, z. B.

$$Ellipse = (\bar{x}, \bar{y}, r_a, r_b, \alpha),$$



3D-Parameterraum:
$\bar{x}, \bar{y} = 0 \ldots 100$
$\rho = 10 \ldots 30$

Bildpunkte $\boldsymbol{p}_k$:
$\boldsymbol{p}_1 = (30, 50)$
$\boldsymbol{p}_2 = (50, 50)$
$\boldsymbol{p}_3 = (40, 40)$
$\boldsymbol{p}_4 = (80, 20)$

Abbildung 9.14
3D-Parameterraum für Kreise. Für jeden Bildpunkt $\boldsymbol{p}_k = (u_k, v_k)$ werden die Zellen entlang einer kegelförmigen Fläche im dreidimensionalen Akkumulator-Array $Acc[\bar{x}, \bar{y}, \rho]$ durchlaufen (inkrementiert).

wobei $(\bar{x}, \bar{y})$ die Koordinaten des Mittelpunkts, (r_a, r_b) die beiden Radien und α die Orientierung der Hauptachse bezeichnen (Abb. 9.12). Um Ellipsen in beliebiger Größe, Lage und Orientierung mit der Hough-Transformation zu finden, würde man daher einen 5-dimensionalen Parameterraum mit geeigneter Auflösung in jeder Dimension benötigen. Eine einfache Rechnung zeigt allerdings den enormen Aufwand: Bei einer Auflösung von nur $128 = 2^7$ Schritten in jeder Dimension ergeben sich bereits 2^{35} Akkumulator-Zellen, was bei einer Implementierung als 4-Byte-*Integers* einem Speicherbedarf von 2^{37} Bytes bzw. 128 Gigabytes entspricht. Auch der für das Füllen und für die Auswertung dieses riesigen Parameterraums notwendige Rechenaufwand lässt die Methode wenig praktikabel erscheinen.

Eine interessante Alternative ist in diesem Fall die *verallgemeinerte Hough-Transformation*, mit der grundsätzlich beliebige zweidimensionale Formen detektiert werden können [4,42]. Die Form der gesuchten Kontur wird dazu punktweise in einer Tabelle kodiert und der zugehörige Parameterraum bezieht sich auf die Position (x_c, y_c), den Maßstab S und die Orientierung θ der Form. Er ist damit höchstens vierdimensional, also kleiner als jener der Standardmethode für allgemeine Ellipsen.

9.5 Aufgaben

Aufg. 9.1. Implementieren Sie die Hough-Transformation zum Auffinden von Geraden unter Berücksichtigung der Endpunkte, wie in Abschn. 9.3.3 beschrieben.

Aufg. 9.2. Realisieren Sie eine *hierarchische* Form der Hough-Transformation (S. 166) für Geraden zur genauen Bestimmung der Parameter.

Aufg. 9.3. Implementieren Sie die Hough-Transformation zum Auffinden von Kreisen und Kreissegmenten mit variablem Radius. Verwenden Sie dazu einen schnellen Algorithmus zum Generieren von Kreisen im Akkumulator-Array, wie in Abschn. 9.4 beschrieben.

10

Morphologische Filter

Bei der Diskussion des Medianfilters in Kap. 6 konnten wir sehen, dass dieser Typ von Filter in der Lage ist, zweidimensionale Bildstrukturen zu verändern (Abschn. 6.4.2). Interessant war zum Beispiel, dass Ecken abgerundet werden und kleinere Strukturen, wie einzelne Punkte und dünne Linien, infolge der Filterung überhaupt verschwinden können (Abb. 10.1). Das Filter reagiert also selektiv auf die *Form* der lokalen Bildinformation. Diese Eigenschaft könnte auch für andere Zwecke nützlich sein, wenn es gelingt, sie nicht nur zufällig, sondern kontrolliert einzusetzen. In diesem Kapitel betrachten wir so genannten „morphologische“ Filter, die imstande sind, die *Struktur* von Bildern gezielt zu beeinflussen.



(a) (b)

Abbildung 10.1
3×3 Medianfilter angewandt auf ein Binärbild. Originalbild (a) und Ergebnis nach der Filterung (b).

Morphologische Filter sind – in ihrer ursprünglichen Form – primär für Binärbilder gedacht, d. h. für Bilder mit nur zwei verschiedenen Pixelwerten 0 und 1 bzw. schwarz und weiß. Binäre Bilder finden sich an vielen Stellen, speziell im Digitaldruck, in der Dokumentenübertragung und -archivierung, aber auch als Bildmasken bei der Bildbearbeitung

und im Video-Compositing. Binärbilder können z. B. aus Grauwertbildern durch eine einfache Schwellwertoperation (Abschn. 5.1.4) erzeugt werden, wobei wir Bildelemente mit dem Wert 1 als *Vordergrund*-Pixel (*foreground*) bzw. mit dem Wert 0 als *Hintergrund*-Pixel (*background*) definieren. In den meisten der folgenden Beispiele ist, wie auch im Druck üblich, der Vordergrund schwarz dargestellt und der Hintergrund weiß.

Am Ende dieses Kapitels werden wir sehen, dass morphologische Filter nicht nur für Binärbilder anwendbar sind, sondern auch für Grauwertbilder und sogar Farbbilder, allerdings unterscheiden sich diese Filter deutlich von den binären Operationen.

10.1 Schrumpfen und wachsen lassen

Ausgangspunkt ist also die Beobachtung, dass, wenn wir ein gewöhnliches 3×3-Medianfilter auf ein Binärbild anwenden, sich größere Bildstrukturen abrunden und kleinere Bildstrukturen, wie Punkte und dünne Linien, vollständig verschwinden. Das könnte nützlich sein, um etwa Strukturen unterhalb einer bestimmten Größe aus dem Bild zu eliminieren. Wie kann man aber die Größe und möglicherweise auch die Form der von einer solchen Operation betroffenen Strukturen kontrollieren?

Die strukturellen Auswirkungen eines Medianfilters sind zwar interessant, aber beginnen wir mit dieser Aufgabe nochmals von vorne. Nehmen wir also an, wir möchten kleine Strukturen in einem Binärbild entfernen, ohne die größeren Strukturen dabei wesentlich zu verändern. Dazu könnte die Kernidee folgende sein (Abb. 10.2):

1. Zunächst werden alle Strukturen im Bild schrittweise „geschrumpft“, wobei jeweils außen eine Schicht von bestimmter Stärke abgelöst wird.
2. Durch das Schrumpfen verschwinden kleinere Strukturen nach und nach, und nur die größeren Strukturen bleiben übrig.
3. Schließlich lassen wir die verbliebenen Strukturen wieder im selben Umfang wachsen.

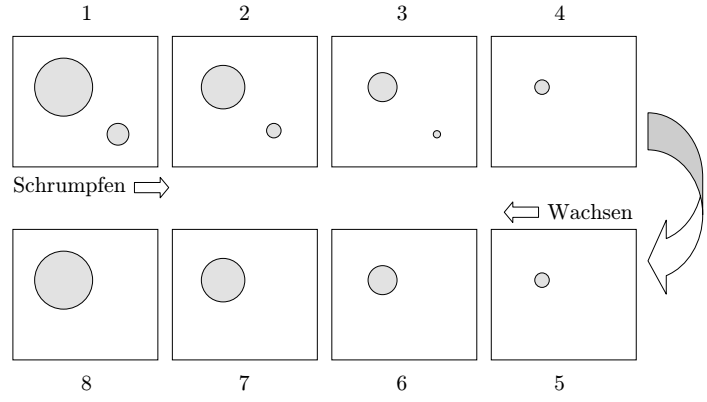


Abbildung 10.2
Durch schrittweises Schrumpfen und anschließendes Wachsen können kleinere Bildstrukturen entfernt werden.

4. Am Ende haben die größeren Regionen wieder annähernd ihre ursprüngliche Form, während die kleineren Regionen des Ausgangsbilds verschwunden sind.

Alles was wir dafür benötigen, sind also zwei Operationen: „Schrumpfen“ lässt sich eine Vordergrundstruktur, indem eine Schicht außen liegender Pixel, die direkt an den Hintergrund angrenzen, entfernt wird (Abb. 10.3). Umgekehrt bedeutet das „Wachsen“ einer Region, dass eine Schicht über die direkt angrenzenden Hintergrundpixel angefügt wird (Abb. 10.4).

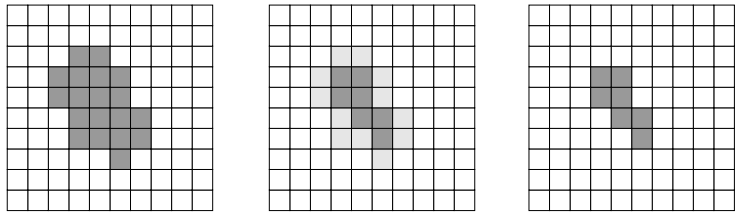


Abbildung 10.3
Schrumpfen einer Bildregion durch Entfernen der Randpixel. Originalbild (a), markierte Randpixel, die direkt an den Hintergrund angrenzen (b), geschrumpftes Ergebnis (c).

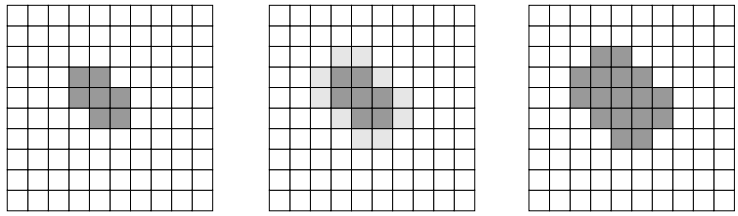


Abbildung 10.4
Wachsen einer Bildregion durch Anfügen zusätzlicher Bildelemente. Originalbild (a), markierte Hintergrundpixel, die direkt an die Region angrenzen (b), gewachsenes Ergebnis (c).

10.1.1 Nachbarschaft von Bildelementen

In beiden Operationen müssen wir festlegen, was es bedeutet, dass zwei Bildelemente aneinander angrenzen, d. h. „benachbart“ sind. Bei einem rechteckigen Bildraster werden traditionell zwei Definitionen von Nachbarschaft unterschieden (Abb. 10.5):

- **4er-Nachbarschaft:** die vier Pixel, die in horizontaler und vertikaler Richtung angrenzen;
- **8er-Nachbarschaft:** die Pixel der *4er-Nachbarschaft* plus die vier über die Diagonalen angrenzenden Pixel.



Abbildung 10.5
Definition der Nachbarschaft bei rechteckigem Bildraster. 4er-Nachbarschaft $N_1 \ldots N_4$ (links) und 8er-Nachbarschaft $N_1 \ldots N_8$ (rechts).

10.2 Morphologische Grundoperationen

Schrumpfen und Wachsen sind die beiden grundlegenden Operationen morphologischer Filter, die man in diesem Zusammenhang als „Erosion“ bzw. „Dilation“ bezeichnet. Diese Operationen sind allerdings allgemeiner als im obigen Beispiel illustriert. Insbesondere gehen sie über das Abschälen oder Anfügen einer einzelnen Schicht von Bildelementen hinaus und erlauben wesentlich komplexere Veränderungen.

10.2.1 Das Strukturelement

Ähnlich der Koeffizientenmatrix eines linearen Filters (Abschn. 6.2), wird auch das Verhalten von morphologischen Filtern durch eine Matrix spezifiziert, die man hier als „Strukturelement“ bezeichnet. Wie das Binärbild selbst enthält das Strukturelement nur die Werte 0 und 1, also

$$H(i,j) \in \{0,1\},$$

und besitzt ebenfalls ein eigenes Koordinatensystem mit dem *hot spot* als Ursprung (Abb. 10.6).

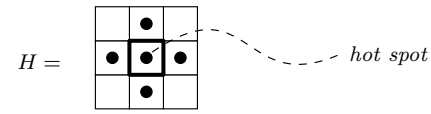


Abbildung 10.6
Beispiel eines Strukturelements für binäre morphologische Operationen. Elemente mit dem Wert 1 sind mit • markiert.

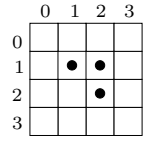
10.2.2 Punktmengen

Zur formalen Definition der Funktion morphologischer Filter ist es praktisch, Binärbilder als Mengen[1] zweidimensionaler Koordinaten-Tupel zu beschreiben. Für ein Binärbild $I(u,v)$ besteht die zugehörige Punktmenge Q_I aus allen Koordinatenpaaren (u,v) seiner Vordergrundpixel, d. h.

[1] *Morphologie* (Lehre der „Gestalt“) ist u. a. ein Teilgebiet der mathematischen Mengenlehre bzw. Algebra.

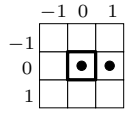
$$Q_I = \{(u,v) \mid I(u,v) = 1\}. \tag{10.1}$$

Wie das Beispiel in Abb. 10.7 zeigt, kann nicht nur ein Binärbild, sondern genauso auch ein binäres Strukturelement als Punktmenge beschrieben werden.



I

$Q_I = \{(1,1),(2,1),(2,2)\}$



H

$Q_H = \{(\mathbf{0},\mathbf{0}),(\mathbf{1},\mathbf{0})\}$

Abbildung 10.7
Beschreibung eines Binärbilds I und eines binären Strukturelements H als Mengen von Koordinatenpaaren Q_I bzw. Q_H.

Grundlegende Operationen auf Binärbilder können ebenfalls auf einfache Weise in dieser Mengennotation beschrieben werden. Zum Beispiel ist das Invertieren eines Binärbilds $I(u,v) \to \neg I(u,v)$, d. h. das Vertauschen von Vorder- und Hintergrund, äquivalent zur Bildung der Komplementärmenge

$$Q_{\neg I} = \overline{Q}_I. \tag{10.2}$$

Werden zwei Binärbilder I_1 und I_2 punktweise durch eine ODER-Operation verknüpft, dann ist die Punktmenge des Resultats die Vereinigung der zugehörigen Punktmengen Q_{I_1} und Q_{I_2}, also

$$Q_{I_1 \vee I_2} = Q_{I_1} \cup Q_{I_2}. \tag{10.3}$$

Da Punktmengen nur eine alternative Darstellung von binären Rasterbildern sind, werden wir beide Notationen im Folgenden je nach Bedarf synonym verwenden. Beispielsweise schreiben wir gelegentlich einfach $I_1 \cup I_2$ statt $Q_{I_1} \cup Q_{I_2}$, oder auch $\overline{I}$ statt $\overline{Q}_I$ für ein invertiertes (Hintergrund-)Bild.

10.2.3 Dilation

Eine *Dilation* ist jene morphologische Operation, die unserem intuitiven Konzept des Wachsens entspricht und in der Mengennotation definiert ist als

$$I \oplus H = \big\{(u',v') = (u+i, v+j) \mid (u',v') \in Q_I, (i,j) \in Q_H\big\}. \tag{10.4}$$

Die aus einer Dilation entstehende Punktmenge entspricht also der (Vektor-)Summe aller möglichen Kombinationen von Koordinatenpaaren aus den ursprünglichen Punktmengen Q_I und Q_H. Man könnte die Operation auch so interpretieren, dass das Strukturelement H an jedem Vordergrundpunkt des Bilds I *repliziert* wird. Oder, umgekehrt, das Bild I wird an jedem Punkt des Strukturelements H repliziert, wie das einfache Beispiel in Abb. 10.8 illustriert.

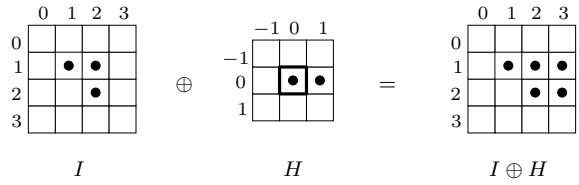


Abbildung 10.8
Beispiel für Dilation. Das Binärbild I wird einer Dilation mit dem Strukturelement H unterzogen. Das Strukturelement H wird im Ergebnis $I \oplus H$ an jedem Punkt des Originalbilds I repliziert.

$$I \oplus H = \{\ (1,1)+(\mathbf{0},\mathbf{0})\ ,\ (1,1)+(\mathbf{1},\mathbf{0})\ ,\ (2,1)+(\mathbf{0},\mathbf{0})\ ,\ (2,1)+(\mathbf{1},\mathbf{0})\ ,\ (2,2)+(\mathbf{0},\mathbf{0})\ ,\ (2,2)+(\mathbf{1},\mathbf{0})\ \}$$

10.2.4 Erosion

Die (quasi-)inverse Operation zur Dilation ist die *Erosion*, die wiederum in Mengennotation definiert ist als

$$I \ominus H = \big\{(u',v') \mid (u'+i, v'+j) \in Q_I, \forall (i,j) \in Q_H\big\}. \tag{10.5}$$

Dieser Vorgang lässt sich folgendermaßen interpretieren: Ein Pixel (u', v') wird im Ergebnis dort (und nur dort) auf 1 gesetzt, wo das Strukturelement H – mit seinem *hot spot* positioniert an der Position (u', v') – vollständig im ursprünglichen Bild eingebettet ist, d. h., wo sich für jeden 1-Punkt in H auch ein entsprechender 1-Punkt in I findet. Ein einfaches Beispiel für die Erosion zeigt Abb. 10.9.

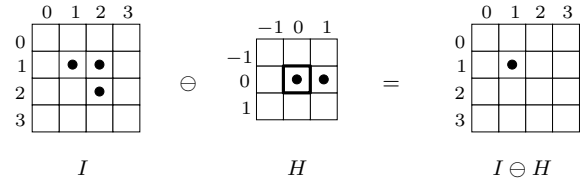


Abbildung 10.9
Beispiel für Erosion. Das Binärbild I wird einer Erosion mit dem Strukturelement H unterzogen. Das Strukturelement ist nur an der Position $(1, 1)$ vollständig in das Bild I eingebettet, sodass im Ergebnis nur das Pixel mit der Koordinate $(1, 1)$ den Wert 1 erhält.

$$I \ominus H = \{\,(1,1)\,\}, \text{weil}$$
$$(1,1)+(\mathbf{0},\mathbf{0}) = (1,1) \in Q_I \quad \text{und} \quad (1,1)+(\mathbf{1},\mathbf{0}) = (2,1) \in Q_I$$

10.2.5 Eigenschaften von Dilation und Erosion

Obwohl Dilation und Erosion nicht wirklich zueinander inverse Operationen sind (i. Allg. kann man die Auswirkungen einer Erosion nicht durch eine nachfolgende Dilation nicht vollständig rückgängig machen und umgekehrt), verbindet sie dennoch eine starke formale Beziehung. Zum einen sind Dilation und Erosion *dual* insofern, als eine Dilation des Vordergrunds durch eine Erosion des Hintergrunds durchgeführt werden kann. Das Gleiche gilt auch umgekehrt, also

$$\overline{I} \oplus H = \overline{(I \ominus H)} \quad \text{und} \quad \overline{I} \ominus H = \overline{(I \oplus H)}. \tag{10.6}$$

Die *Dilation* ist des Weiteren *kommutativ*, d. h.

$$I \oplus H = H \oplus I, \tag{10.7}$$

und daher können, genauso wie bei der linearen Faltung, Bild und Strukturelement (Filter) im Prinzip miteinander vertauscht werden. Analog existiert – ähnlich der Dirac-Funktion δ (s. Abschn. 6.3.4) – bezüglich der Dilation auch ein *neutrales Element*

$$I \oplus \delta = \delta \oplus I = I, \quad \text{wobei } Q_\delta = \big\{(0,0)\big\}. \tag{10.8}$$

Darüber hinaus ist die Dilation *assoziativ*, d. h.

$$(I_1 \oplus I_2) \oplus I_3 = I_1 \oplus (I_2 \oplus I_3), \tag{10.9}$$

also ist die Reihenfolge aufeinander folgender Dilationen nicht relevant. Das bedeutet – wie auch bei linearen Filtern (vgl. Gl. 6.21) –, dass eine Dilation mit einem großen Strukturelement der Form $H = H_1 \oplus H_2 \oplus \ldots \oplus H_K$ als Folge mehrerer Dilationen mit i. Allg. kleineren Strukturelementen realisiert werden kann:

$$I \oplus H = (\ldots((I \oplus H_1) \oplus H_2) \oplus \ldots \oplus H_K) \tag{10.10}$$

Die *Erosion* ist – wie die gewöhnliche arithmetische Subtraktion auch – *nicht* kommutativ, also

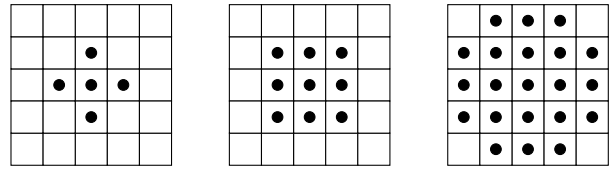
$$I \ominus H \neq H \ominus I. \tag{10.11}$$

Werden allerdings Erosion und Dilation miteinander kombiniert, dann gilt – wiederum analog zu Subtraktion und Addition –

$$(I_1 \ominus I_2) \ominus I_3 = I_1 \ominus (I_2 \oplus I_3). \tag{10.12}$$

10.2.6 Design morphologischer Filter

Morphologische Filter werden spezifiziert durch (a) den Typ der Filteroperation und (b) das entsprechende Strukturelement. Die Größe und Form des Strukturelements ist abhängig von der jeweiligen Anwendung, der Bildauflösung usw. In der Praxis werden häufig scheibenförmige Strukturelemente verschiedener Größe verwendet, wie in Abb. 10.10 gezeigt. Ein scheibenförmiges Strukturelement mit Radius r fügt bei einer Dilationsoperation eine Schicht mit der Dicke von r Pixel an jede Vordergrundstruktur an. Umgekehrt wird durch die Erosion mit diesem Strukturelement jeweils eine Schicht mit der gleichen Dicke abgeschält. Ausgehend von dem ursprünglichen Binärbild in Abb. 10.11, sind die Ergebnisse von Dilation und Erosion mit scheibenförmigen Strukturelementen verschiedener Radien in Abb. 10.12 gezeigt. Ergebnisse für verschiedene andere, frei gestaltete Strukturelemente sind in Abb. 10.13 dargestellt.



(a) (b) (c)

Abbildung 10.10
Typische binäre Strukturelemente verschiedener Größe. 4er-Nachbarschaft (a), 8er-Nachbarschaft (b), kleine Scheibe – *small disk* (c).

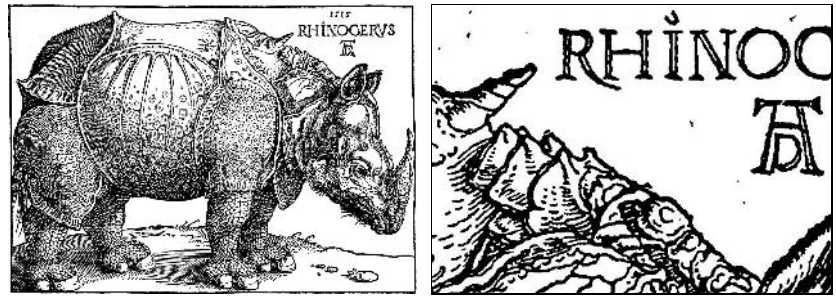


Abbildung 10.11
Binäres Originalbild und Ausschnitt für die nachfolgenden Beispiele (Illustration von Albrecht Dürer, 1515).

Im Unterschied zu linearen Filtern (Abschn. 6.3.3) ist es i. Allg. nicht möglich, ein *isotropes* zweidimensionales Strukturelement H° aus eindimensionalen Strukturelementen H_x und H_y zu bilden, denn die Verknüpfung $H_x \oplus H_y$ ergibt immer ein rechteckiges (also nicht isotropes) Strukturelement. Eine verbreitete Methode zur Implementierung großer scheibenförmiger Filter ist die wiederholte Anwendung kleiner scheibenförmiger Strukturelemente, wodurch sich allerdings in der Regel ebenfalls kein isotroper Gesamtoperator ergibt (Abb. 10.14).

10.2.7 Anwendungsbeispiel: *Outline*

Eine typische Anwendung morphologischer Operationen ist die Extraktion der Ränder von Vordergrundstrukturen. Der Vorgang ist sehr einfach: Zunächst wenden wir eine Erosion auf das Originalbild I an, um darin die Randpixel zu entfernen, d. h.

$$I' = I \ominus H_n,$$

wobei H_n ein Strukturelement z. B. für eine 4er- oder 8er-Nachbarschaft (Abb. 10.10) ist. Die eigentlichen Randpixel B sind jene, die zwar im Originalbild, aber *nicht* im erodierten Bild enthalten sind, also die Schnittmenge zwischen dem Originalbild I und dem invertierten Bild $\overline{I'}$,

$$B = I \cap \overline{I'} = I \cap \overline{(I \ominus H_n)}. \tag{10.13}$$

Ein Beispiel für die Extraktion von Rändern zeigt Abb. 10.15. Interessant ist dabei, dass die Verwendung der 4er-Nachbarschaft für das Strukturelement H_n zu einer „8-verbundenen“ Kontur führt und umgekehrt [47, S. 504].

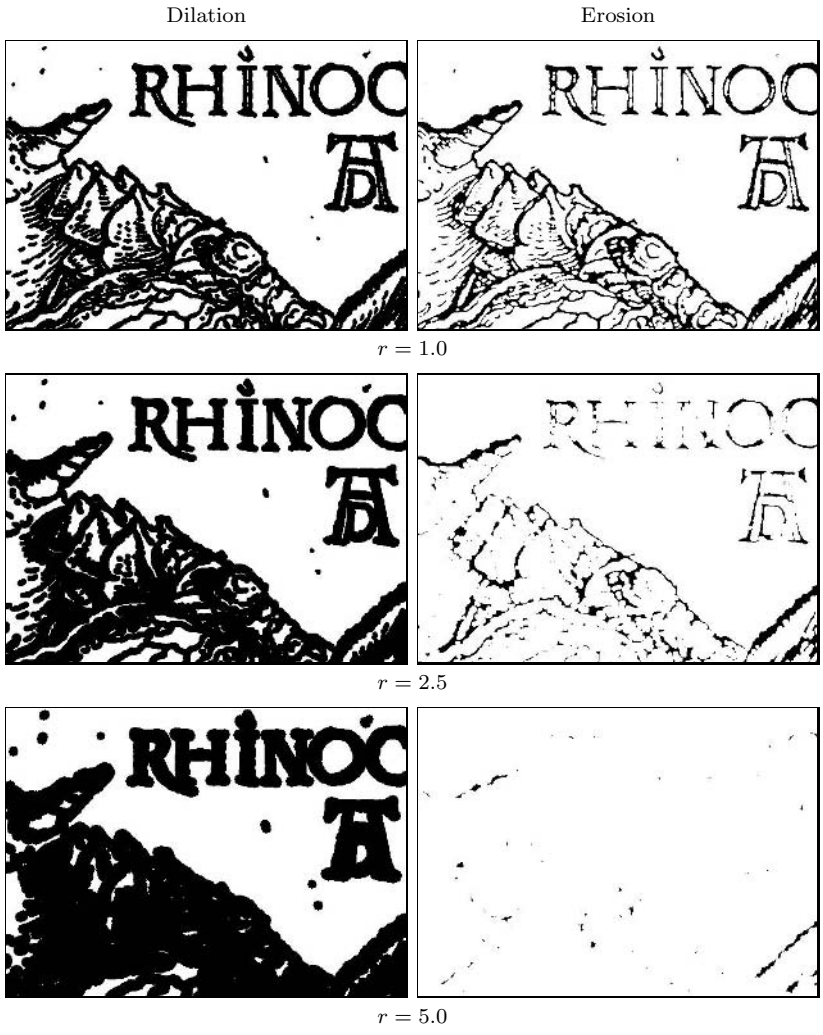


Abbildung 10.12
Ergebnisse der binären Dilation und Erosion mit scheibenförmigen Strukturelementen. Der Radius r des Strukturelements ist 1.0 (oben), 2.5 (Mitte) bzw. 5.0 (unten).

10.3 Zusammengesetzte Operationen

Aufgrund ihrer Semidualität werden Dilation und Erosion häufig in zusammengesetzten Operationen verwendet, von denen zwei so bedeutend sind, dass sie eigene Namen (und sogar Symbole) haben: „Opening“ und „Closing“.[2]

10.3.1 Opening

Ein Opening $(I \circ H)$ ist eine Erosion gefolgt von einer Dilation mit demselben Strukturelement H, d. h.

$$I \circ H = (I \ominus H) \oplus H. \tag{10.14}$$

[2] Die englischen Begriffe werden auch im Deutschen häufig verwendet.

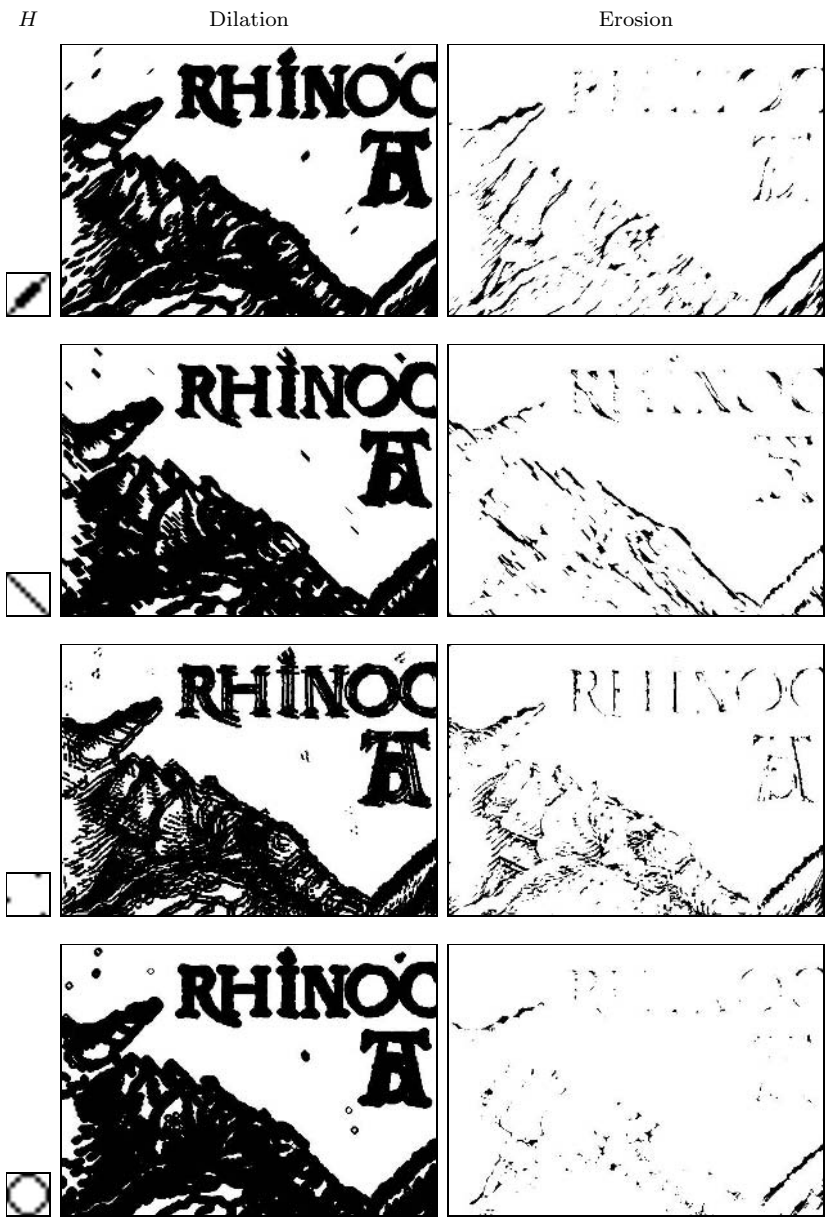


Abbildung 10.13
Binäre Dilation und Erosion mit verschiedenen, frei gestalteten Strukturelementen. Die Strukturelemente selbst sind links gezeigt. Man sieht deutlich, dass einzelne Bildpunkte bei der Dilation zur Form des Strukturelements expandieren, ähnlich einer „Impulsantwort“. Bei der Erosion bleiben nur jene Stellen übrig, an denen das Strukturelement im Bild vollständig abgedeckt ist.

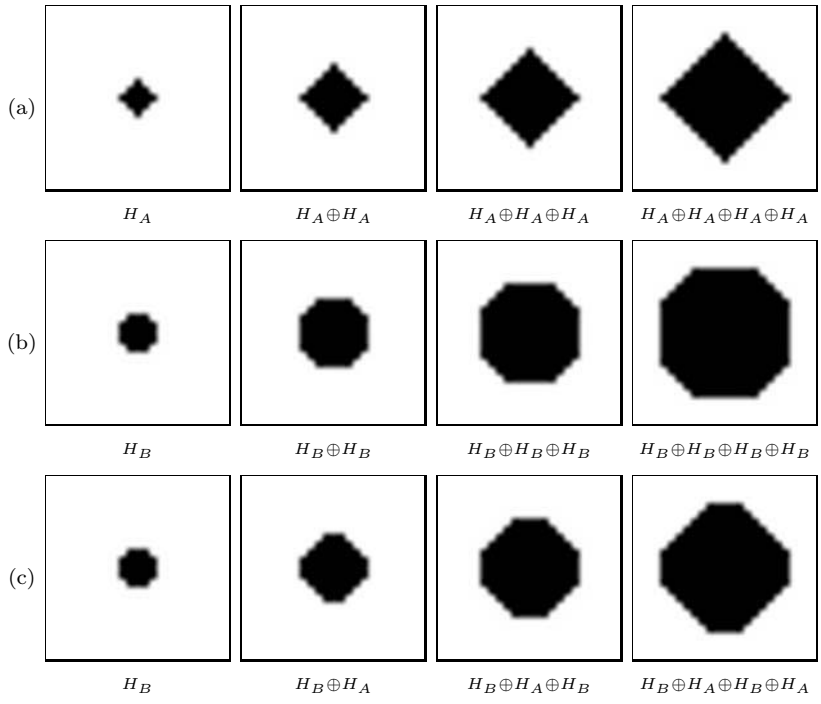


Abbildung 10.14
Bildung größerer Filter durch wiederholte Anwendung kleinerer Strukturelemente. Mehrfache Anwendung des Strukturelements H_A (a) und des Strukturelements H_B (b). Abwechselnde Anwendung von H_B und H_A (c).

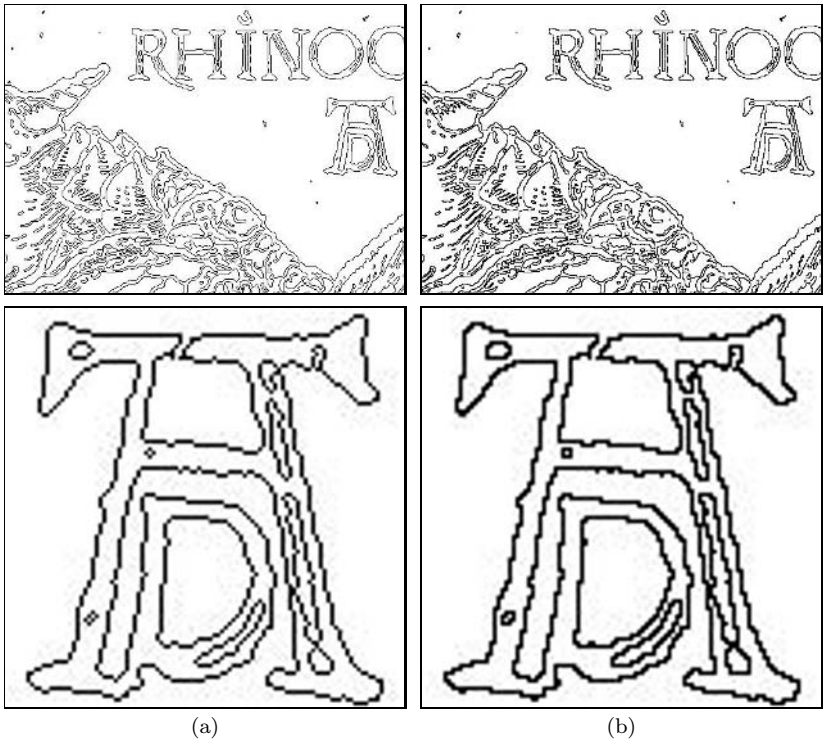


Abbildung 10.15
Extraktion von Rändern mit binären morphologischen Operationen. Das Strukturelement in (a) entspricht einer 4er-Nachbarschaft und führt zu einer „8-verbundenen“ Kontur. Umgekehrt führt ein Strukturelement mit einer 8er-Nachbarschaft (b) zu einer „4-verbundenen“ Kontur.

Ein Opening bewirkt, dass alle Vordergrundstrukturen, die kleiner sind als das Strukturelement, im ersten Teilschritt (Erosion) eliminiert werden. Die verbleibenden Strukturen werden durch die nachfolgende Dilation geglättet und wachsen ungefähr wieder auf ihre ursprüngliche Größe (Abb. 10.16). Diese Abfolge von „Schrumpfen“ und anschließendem „Wachsen“ entspricht der Idee, die wir in Abschn. 10.1 skizziert hatten.

10.3.2 Closing

Wird die Abfolge von Erosion und Dilation umgekehrt, bezeichnet man die resultierende Operation als Closing ($I \bullet H$), d. h.

$$I \bullet H = (I \oplus H) \ominus H. \tag{10.15}$$

Durch ein Closing werden Löcher in Vordergrundstrukturen und Zwischenräume, die kleiner als das Strukturelement H sind, gefüllt (Abb. 10.16).

10.3.3 Eigenschaften von Opening und Closing

Beide Operationen, Opening wie auch Closing, sind *idempotent*, d. h., ihre Ergebnisse sind insofern „final“, als jede weitere Anwendung derselben Operation das Bild nicht mehr verändert:

$$\begin{aligned} (I \circ H) \circ H &= I \circ H \\ (I \bullet H) \bullet H &= I \bullet H \end{aligned} \tag{10.16}$$

Die beiden Operationen sind darüber hinaus zueinander „dual“ in dem Sinn, dass ein Opening auf den Vordergrund äquivalent ist zu einem Closing des Hintergrunds und umgekehrt, d. h.

$$\begin{aligned} I \circ H &= \overline{\left(\overline{I} \bullet H\right)} \\ I \bullet H &= \overline{\left(\overline{I} \circ H\right)} \end{aligned} \tag{10.17}$$

10.4 Morphologische Filter für Grauwert- und Farbbilder

Morphologische Operationen sind nicht auf Binärbilder beschränkt, sondern in ähnlicher Form auch für Grauwertbilder definiert. Bei Farbbildern werden diese Methoden unabhängig auf die einzelnen Farbkanäle angewandt. Trotz der identischen Bezeichnung unterscheidet sich allerdings die Definition der morphologischen Operationen für Grauwertbilder stark von der für Binärbilder. Gleiches gilt auch für deren Anwendungen.

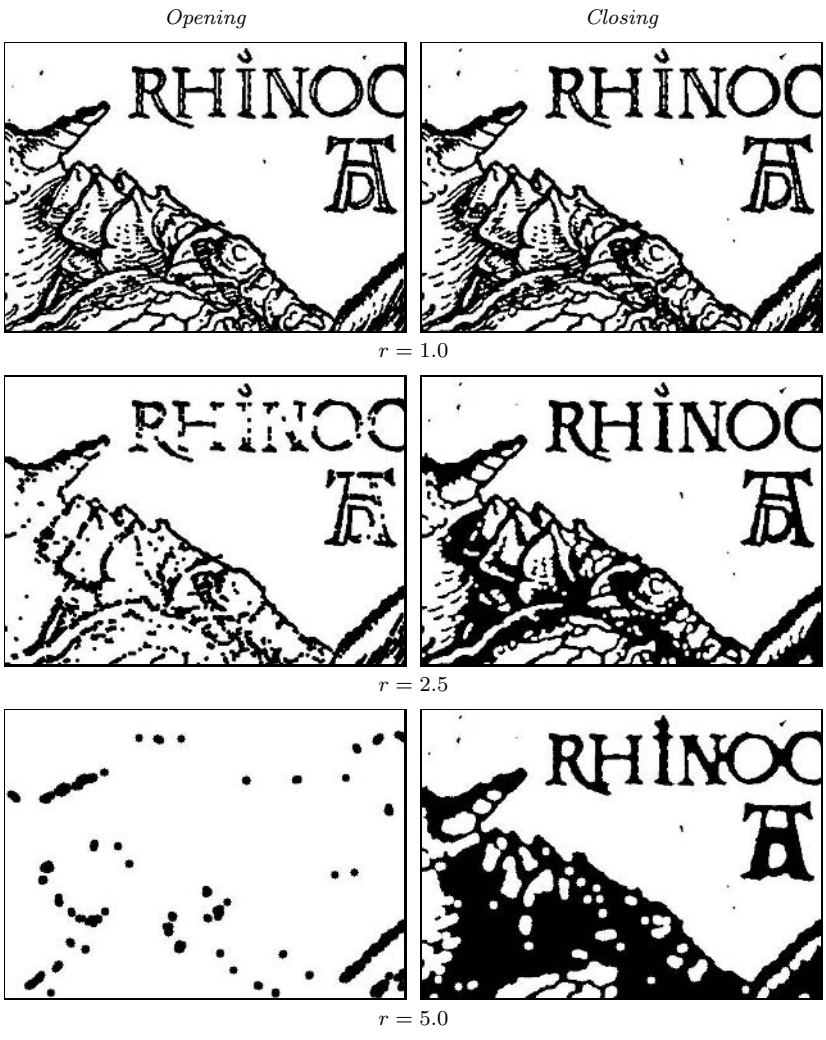


Abbildung 10.16
Binäres Opening und Closing mit scheibenförmigen Strukturelementen. Der Radius r des Strukturelements ist 1.0 (oben), 2.5 (Mitte) bzw. 5.0 (unten).

10.4.1 Strukturelemente

Zunächst werden die Strukturelemente bei morphologischen Filtern für Grauwertbilder nicht wie bei binären Filtern als Punktmengen, sondern als 2D-Funktionen mit beliebigen (reellen) Werten definiert, d. h.

$$H(i,j) \in \mathbb{R}.$$

Die Werte in $H(i,j)$ können auch negativ oder null sein, allerdings beeinflussen – im Unterschied zur linearen Faltung (Abschn. 6.3.1) – auch die Nullwerte das Ergebnis. Bei der Implementierung morphologischer Grauwert-Operationen muss daher bei den Zellen des Strukturelements H zwischen dem Wert 0 und *leeren* Zellen ($\times$ = „don't care“) explizit unterschieden werden, also z. B.

$$\begin{array}{|c|c|c|}\hline 0&1&0\\\hline 1&\mathbf{2}&1\\\hline 0&1&0\\\hline\end{array} \neq \begin{array}{|c|c|c|}\hline \times&1&\times\\\hline 1&\mathbf{2}&1\\\hline \times&1&\times\\\hline\end{array}. \tag{10.18}$$

10.4.2 Grauwert-Dilation und -Erosion

Die Grauwert-Dilation $\oplus$ wird definiert als *Maximum* der addierten Werte des Filters H und der entsprechenden Bildregion I, d. h.

$$(I \oplus H)(u,v) = \max_{(i,j)\in H} \big\{I(u+i, v+j) + H(i,j)\big\}. \tag{10.19}$$

Umgekehrt entspricht die Grauwert-Erosion dem *Minimum* der Differenzen, also

$$(I \ominus H)(u,v) = \min_{(i,j)\in H} \big\{I(u+i, v+j) - H(i,j)\big\}. \tag{10.20}$$

Abb. 10.17 zeigt anhand eines einfachen Beispiels die Wirkungsweise der Grauwert-Dilation, Abb. 10.18 demonstriert analog dazu die Erosion. In beiden Operationen können grundsätzlich negative Ergebniswerte entstehen, die bei einem eingeschränkten Wertebereich z. B. mittels *Clamping*

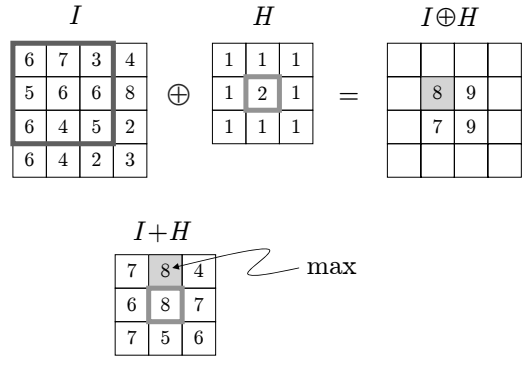


Abbildung 10.17
Grauwert-Dilation $I \oplus H$. Das 3×3-Strukturelement ist dem Bild I überlagert dargestellt. Die Werte in I werden elementweise zu den entsprechenden Werten in H addiert; das Zwischenergebnis $(I + H)$ für die gezeigte Filterposition ist darunter abgebildet. Dessen Maximalwert $8 = 7{+}1$ wird an der aktuellen Position des *hot spot* in das Ergebnisbild $(I \oplus H)$ eingesetzt. Zusätzlich sind die Ergebnisse für drei weitere Filterpositionen gezeigt.

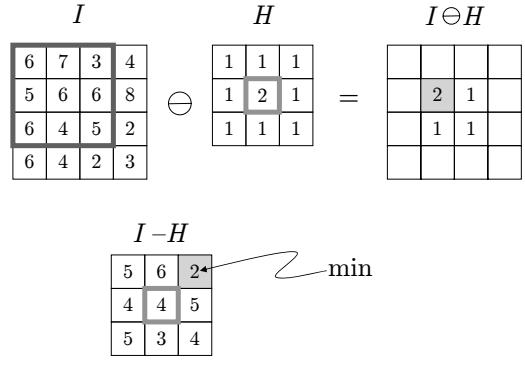


Abbildung 10.18
Grauwert-Erosion $I \ominus H$. Das 3×3-Strukturelement ist dem Bild I überlagert dargestellt. Die Werte von H werden elementweise von den entsprechenden Werten in I subtrahiert; das Zwischenergebnis $(I - H)$ für die gezeigte Filterposition ist darunter abgebildet. Dessen Minimalwert $3 - 1 = 2$ wird an der aktuellen Position des *hot spot* in das Ergebnisbild $(I \ominus H)$ eingesetzt.



Abbildung 10.19
Grauwert-Dilation und -Erosion mit scheibenförmigen Strukturelementen. Der Radius r des Strukturelements ist 2.5 (oben), 5.0 (Mitte) und 10.0 (unten).

(s. Abschn. 5.1.2) zu berücksichtigen sind. Ergebnisse von Dilation und Erosion mit konkreten Grauwertbildern und scheibenförmigen Strukturelementen verschiedener Größe zeigt Abb. 10.19.

10.4.3 Grauwert-Opening und -Closing

Opening und Closing für Grauwertbilder sind – genauso wie für Binärbilder (Gl. 10.14, 10.15) – als zusammengesetzte Dilation und Erosion mit jeweils demselben Strukturelement definiert. Abb. 10.20 zeigt Beispiele für diese Operationen, wiederum unter Verwendung scheibenförmiger Strukturelemente verschiedener Größe.

In Abb. 10.21 und 10.22 sind die Ergebnisse von Grauwert-Dilation und -Erosion bzw. von Grauwert-Opening und -Closing mit verschiedenen, frei gestalteten Strukturelementen dargestellt. Interessante Effekte können z. B. mit Strukturelementen erzielt werden, die der Form natürlicher Pinselstriche ähnlich sind.

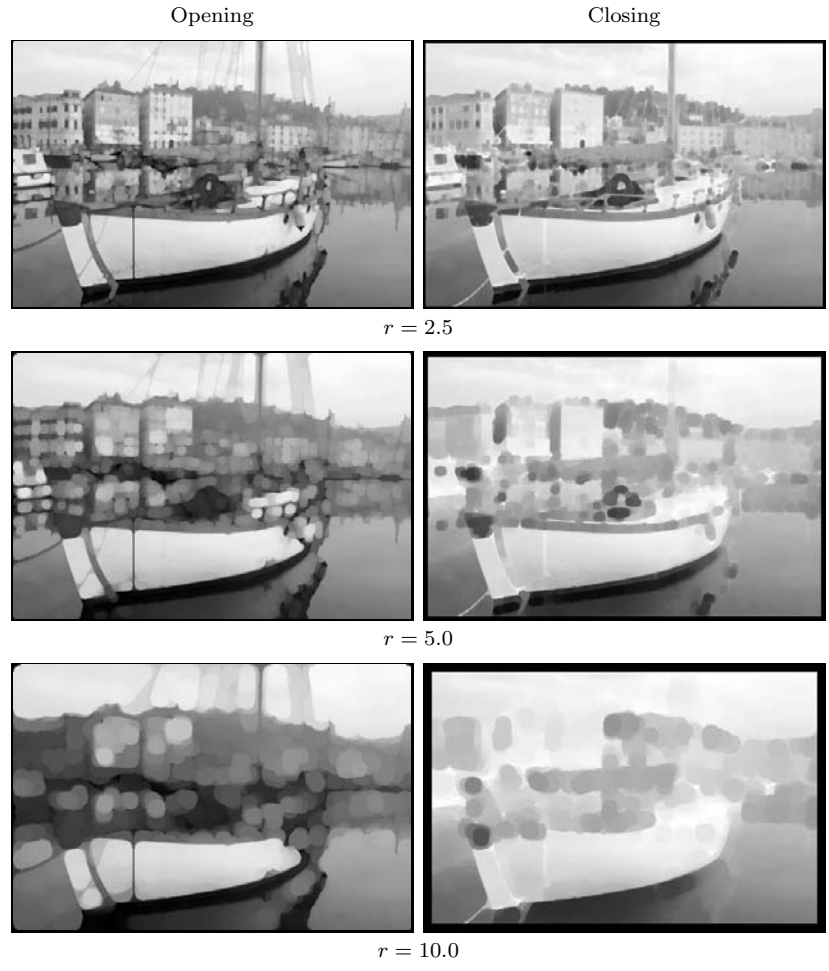


Abbildung 10.20
Grauwert-Opening und -Closing mit scheibenförmigen Strukturelementen. Der Radius r des Strukturelements ist 2.5 (oben), 5.0 (Mitte) und 10.0 (unten).

10.5 Implementierung morphologischer Filter

10.5.1 Binäre Bilder in ImageJ

Binäre Bilder werden in ImageJ – genauso wie Grauwertbilder – mit 8 Bits pro Pixel dargestellt.[3] Üblicherweise verwendet man die Intensitätswerte 0 und 255 für die binären Werte 0 bzw. 1. In diesem Fall werden Vordergrundpixel weiß und Hintergrundpixel schwarz dargestellt. Falls eine umgekehrte Darstellung (Vordergrund schwarz) gewünscht ist, kann dies am einfachsten durch Invertieren der Display-Funktion (Lookup-Table LUT) erreicht werden: entweder über das Menü

Image→Lookup Tables→Invert LUT

oder innerhalb des Programms durch die `ImageProcessor`-Methode

[3] In ImageJ gibt es kein spezielles Speicherformat für binäre Bilder, auch die Klasse `BinaryProcessor` verwendet 8-Bit Bilddaten.

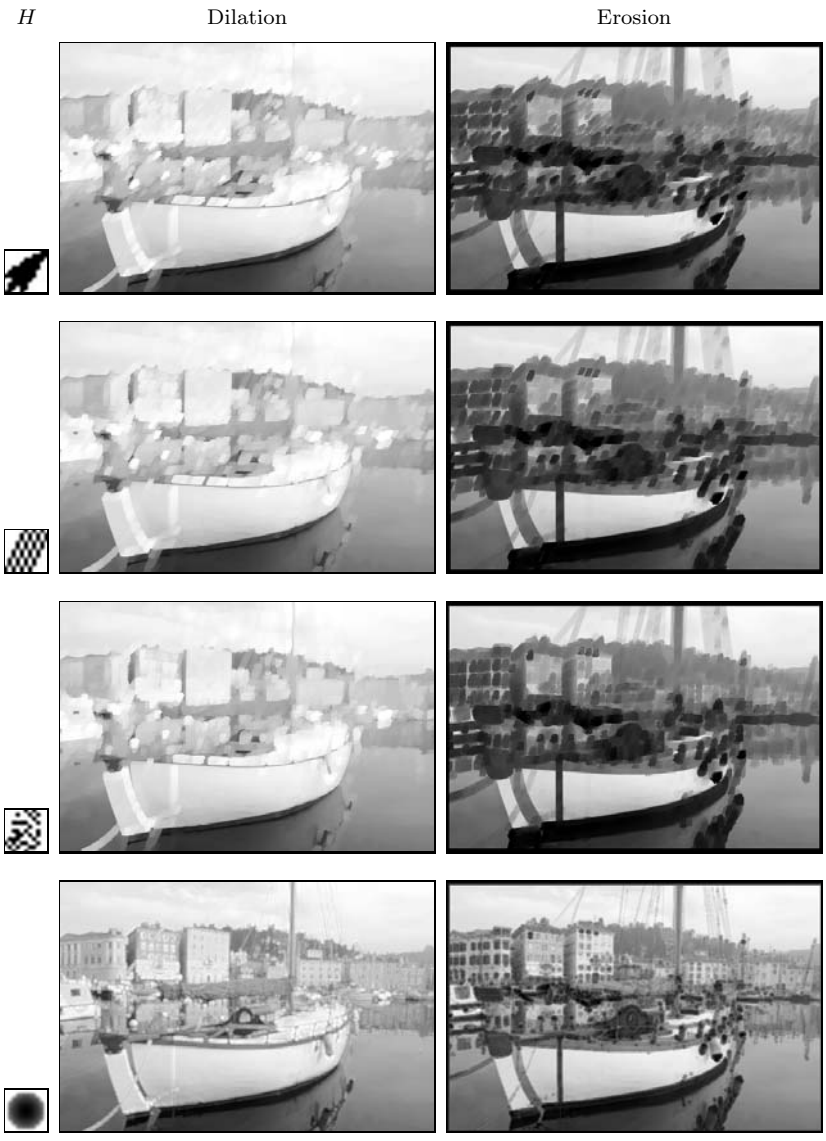


Abbildung 10.21
Grauwert-Dilation und -Erosion mit verschiedenen, frei gestalteten Strukturelementen.

`void invertLut()`.

Diese Anweisung verändert nur die Anzeige des jeweiligen Bilds und nicht die Bildinhalte (Pixelwerte) selbst.

10.5.2 Dilation und Erosion

Die wichtigsten morphologischen Operationen sind in ImageJ als Methoden der Klasse `ImageProcessor` (s. auch Abschn. 10.5.5) bereits fertig implementiert, allerdings beschränkt auf Strukturelemente der Größe 3×3.

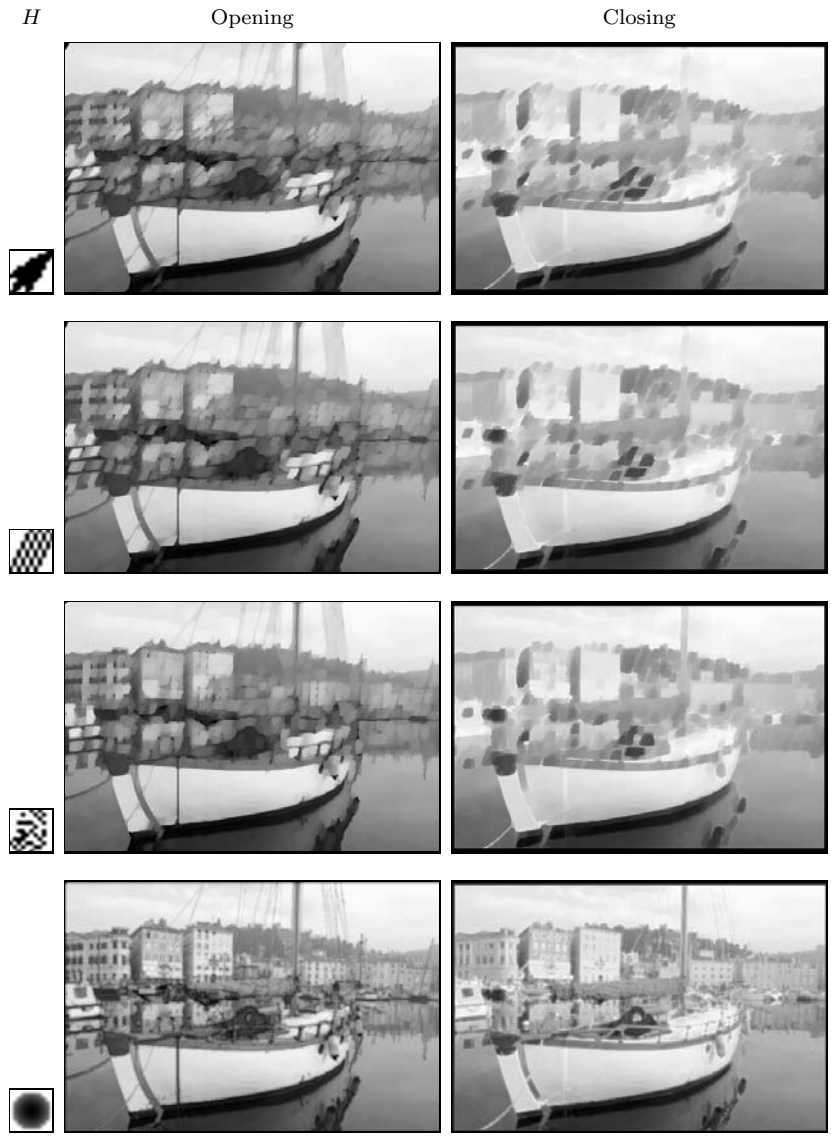


Abbildung 10.22
Grauwert-Opening und -Closing mit verschiedenen, frei gestalteten Strukturelementen.

Im Folgenden ist als Beispiel die Implementierung der binären Dilation gezeigt, über die – wegen der Dualität zur Erosion (Gl. 10.6) – auch die meisten anderen morphologischen Operationen realisiert werden können. Ausgangspunkt für die `dilate()`-Methode ist ein Binärbild `I` mit den Werten 0 (Hintergrund) und 255 (Vordergrund) sowie ein zweidimensionales Strukturelement `H` mit 0/1-Werten, dessen *hot spot* im Zentrum angenommen wird:

```
1  import ij.process.Blitter;
2  import ij.process.ImageProcessor;
3  ...
4  void dilate(ImageProcessor I, int[][] H){
5    //assume that the hot spot of H is at its center (ic,jc):
6    int ic = (H[0].length-1)/2;
7    int jc = (H.length-1)/2;
8
9    ImageProcessor np
10     = I.createProcessor(I.getWidth(),I.getHeight());
11
12   for (int j=0; j<H.length; j++){
13     for (int i=0; i<H[j].length; i++){
14       if (H[j][i] == 1) { // this pixel is set
15         //copy image into position (i-ic,j-jc):
16         np.copyBits(I,i-ic,j-jc,Blitter.MAX);
17       }
18     }
19   }
20   //copy result back to original image
21   I.copyBits(np,0,0,Blitter.COPY);
22 }
```

Für das Ergebnis wird zunächst (in Zeile 9) ein neues (leeres) Bild angelegt, das am Ende in das ursprüngliche Bild zurückkopiert wird (Zeile 21). Die eigentliche Dilation erfolgt iterativ, indem für jede Position (i, j) im Strukturelement mit dem Wert $H(i, j) = 1$ das ursprüngliche Bild, verschoben um (i, j), in das Ergebnis kopiert wird. Dazu wird (in Zeile 16) die `ImageProcessor`-Methode `copyBits()` mit dem Operationsschlüssel `Blitter.MAX` verwendet (s. auch Abschn. 5.7.3). Interpretiert man die Pixel als binäre Werte, entspricht dies einer ODER-Verknüpfung zwischen dem Ergebnis und dem verschobenen Ausgangsbild.

Die Dilation ist die einzige Operation, die tatsächlich im Detail implementiert werden muss, denn die Erosion kann als Dilation des Hintergrunds durchgeführt werden, also durch Invertieren des Bilds, Durchführung der Dilation und neuerlichem Invertieren:

```
23 void erode(ImageProcessor I, int[][] H){
24   I.invert();
25   dilate(I,H); //dilates the background
26   I.invert();
27 }
```

10.5.3 Opening und Closing

Opening- und Closing-Operation können nunmehr einfach als Abfolge von Dilation und Erosion mit jeweils demselben Strukturelement `H` (Abschn. 10.3) realisiert werden:

```
28  void open(ImageProcessor I, int[][] H){
29    erode(I,H);
30    dilate(I,H);
31  }
32
33  void close(ImageProcessor I, int[][] H){
34    dilate(I,H);
35    erode(I,H);
36  }
```

10.5.4 Outline

Für die in Abschn. 10.2.7 beschriebene *Outline*-Operation zur Extraktion von Rändern können wir z. B. ein 3×3-Strukturelement `H` für die 4er-Nachbarschaft verwenden. Zunächst wird eine Kopie des Bilds (`Ie`) erzeugt und anschließend erodiert (Zeile 43). Zwischen dem Ergebnis der Erosion und dem Originalbild wird in Zeile 44 die Differenz gebildet (durch Anwendung der Methode `copyBits()` mit dem Argument `Blitter.DIFFERENCE`). Am Ende enthält das ursprüngliche Bild `I` die Umrisse der Vordergrundstrukturen:

```
37  void outline(ImageProcessor I){
38    int[][] H = { //4-neighborhood structuring element
39      {0,1,0},
40      {1,1,1},
41      {0,1,0}};
42    ImageProcessor Ie = I.duplicate();
43    erode(Ie,H); // /* I' ← I ⊖ H */
44    I.copyBits(Ie,0,0,Blitter.DIFFERENCE); // I ← I ∩ I'
45  }
```

10.5.5 Morphologische Operationen in ImageJ

`ImageProcessor`

ImageJ stellt für die Klasse `ImageProcessor` einige fertige Methoden für einfache morphologische Filter zur Verfügung:

```
void dilate()
void erode()
void open()
void close()
```

Diese Methoden verwenden vollbesetzte 3×3-Strukturelemente (entsprechend Abb. 10.10 (b)), die – abhängig vom Bildinhalt – entweder binäre oder Grauwert-Operationen durchführen. Die Klasse `ColorProcessor` stellt diese Methoden auch für RGB-Farbbilder zur Verfügung, wobei die morphologischen Operationen individuell auf die einzelnen Farbkanäle wie für gewöhnliche Grauwertbilder angewandt werden.

BinaryProcessor

Daneben bietet die Klasse `BinaryProcessor` (eine Unterklasse von `ByteProcessor`) mit den Methoden

```
void outline()
void skeletonize()
```

spezielle morphologische Operationen, die ausschließlich für Binärbilder definiert sind. Die Methode `outline()` implementiert die Extraktion von Rändern mit dem Strukturelement einer 8er-Nachbarschaft, wie in Abschn. 10.2.7 beschrieben.

Die in der Methode `skeletonize()` implementierte Operation bezeichnet man als „Thinning“, das ist eine iterative Erosion, mit der Strukturen auf eine Dicke von 1 Pixel reduziert werden, ohne sie dabei in mehrere Teile zu zerlegen. Dabei muss, abhängig vom aktuellen Bildinhalt innerhalb der Filterregion (üblicherweise von der Größe 3×3), jeweils entschieden werden, ob tatsächlich eine Erosion durchgeführt werden soll oder nicht. Die Operation erfolgt in mehreren Durchläufen so lange, bis sich im Ergebnis keine Änderungen mehr ergeben (s. beispielsweise [30, S. 535], [48, S. 517]). Die konkrete Implementierung in ImageJ basiert auf einem effizienten Algorithmus von Zhang und Suen [91]. Abbildung 10.23 zeigt ein Beispiel für die Anwendung von `skeletonize()`.

Die Verwendung der Methoden `outline()` und `skeletonize()` setzt ein Objekt der Klasse `BinaryProcessor` voraus, das wiederum nur aus einem bestehenden `ByteProcessor` erzeugt werden kann. Dabei wird angenommen, dass das ursprüngliche Bild nur Werte mit 0 (Hintergrund) und 255 (Vordergrund) enthält. Das nachfolgende Beispiel zeigt den Einsatz von `outline()` innerhalb der `run()`-Methode eines ImageJ-Plugins:

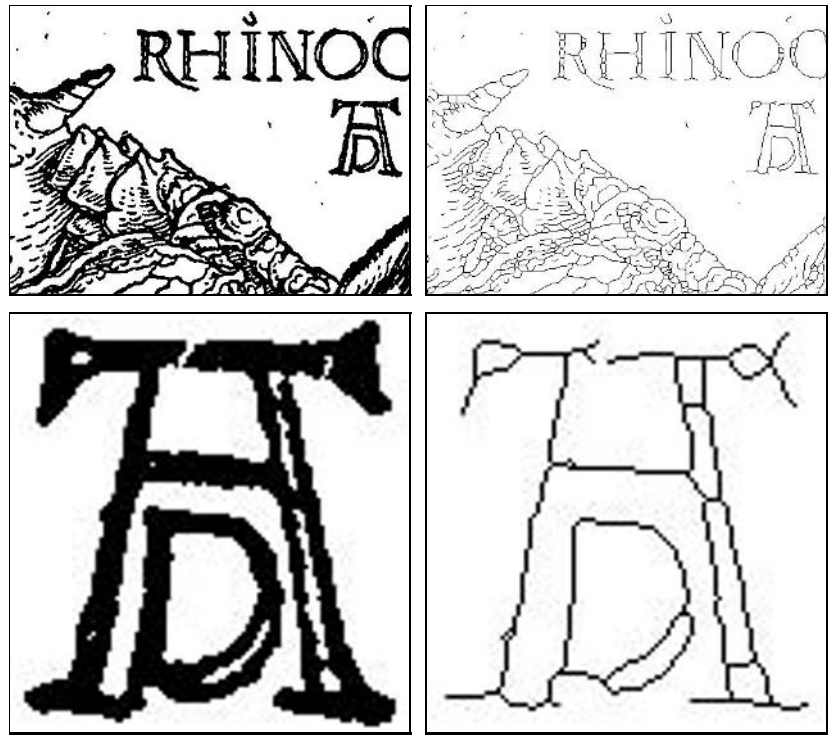
```
1 import ij.process.*;
2 ...
3
4    public void run(ImageProcessor ip) {
5       BinaryProcessor bp
6            = new BinaryProcessor((ByteProcessor)ip);
7       bp.outline();
8    }
```

Der neue `BinaryProcessor bp` legt übrigens keine eigenen Bilddaten an, sondern verweist lediglich auf die Bilddaten des ursprünglichen Bilds `ip`, sodass jede Veränderung von `bp` (z. B. durch den Aufruf von `outline()`) gleichzeitig auch `ip` betrifft.

Weitere morphologische Filter

Neben den in ImageJ direkt implementierten Methoden sind einzelne Plugins und ganze Packages für spezielle morphologische Filter online[4]

[4] http://rsb.info.nih.gov/ij/plugins



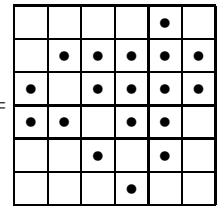
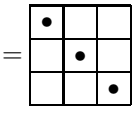
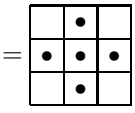
(a) (b)

Abbildung 10.23
Beispiel für die Anwendung der `skeletonize()`-Methode. Originalbild bzw. Detail (a) und zugehörige Ergebnisse (b).

verfügbar, wie z. B. das *Grayscale Morphology* Package von Dimiter Prodanov, bei dem die Strukturelemente weitgehend frei spezifiziert werden können (eine modifizierte Version wurde für einige der Beispiele in diesem Kapitel verwendet).

10.6 Aufgaben

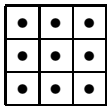
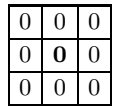
Aufg. 10.1. Berechnen Sie manuell die Ergebnisse für die Dilation und die Erosion zwischen dem folgenden Binärbild I und den Strukturelementen H_1 und H_2:

$I =$ 
$H_1 =$ 
$H_2 =$ 

Aufg. 10.2. Angenommen, in einem Binärbild I sind störende Vordergrundflecken mit einem Durchmesser von maximal 5 Pixel zu entfernen

und die restlichen Bildkomponenten sollen möglichst unverändert bleiben. Entwerfen Sie für diesen Zweck eine morphologische Operation und erproben Sie diese an geeigneten Testbildern.

Aufg. 10.3. Zeigen Sie, dass im Fall eines Strukturelements der Form

 für Binärbilder bzw.  für Grauwertbilder

eine Dilation äquivalent zu einem 3×3-Maximum-Filter und die entsprechende Erosion äquivalent zu einem 3×3-Minimum-Filter ist (Abschn. 6.4.1).

11

Regionen in Binärbildern

Binärbilder, mit denen wir uns bereits im vorhergehenden Kapitel ausführlich beschäftigt haben, sind Bilder, in denen ein Pixel einen von nur zwei Werten annehmen kann. Wir bezeichnen diese beiden Werte häufig als „Vordergrund“ bzw. „Hintergrund“, obwohl eine solche eindeutige Unterscheidung in natürlichen Bildern oft nicht möglich ist. In diesem Kapitel gilt unser Augenmerk zusammenhängenden Bildstrukturen und insbesondere der Frage, wie wir diese isolieren und beschreiben können.

Angenommen, wir müssten ein Programm erstellen, das die Anzahl und Art der in Abb. 11.1 abgebildeten Objekte interpretiert. Solange wir jedes einzelne Pixel isoliert betrachten, werden wir nicht herausfinden, wie viele Objekte überhaupt in diesem Bild sind, wo sie sich befinden und welche Pixel zu welchem der Objekte gehören. Unsere erste Aufgabe ist daher, zunächst einmal jedes einzelne Objekt zu finden, indem wir alle Pixel zusammenfügen, die Teil dieses Objekts sind. Im einfachsten

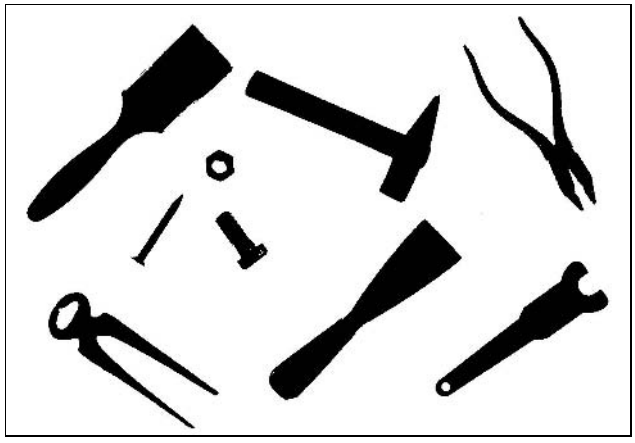


Abbildung 11.1
Binärbild mit 9 Objekten. Jedem Objekt entspricht jeweils eine zusammenhängende Region von Vordergrundpixel.

Fall ist ein Objekt eine Gruppe von aneinander angrenzenden Vordergrundpixeln bzw. eine *verbundene binäre Bildregion*.

11.1 Auffinden von Bildregionen

Bei der Suche nach binären Bildregionen sind die zunächst wichtigsten Aufgaben, herauszufinden, welche Pixel zu welcher Region gehören, wie viele Regionen im Bild existieren und wo sich diese Regionen befinden. Diese Schritte werden üblicherweise in *einem* Prozess durchgeführt, der als „Regionenmarkierung“ (*region labeling* oder auch *region coloring*) bezeichnet wird. Dabei werden zueinander benachbarte Pixel schrittweise zu Regionen zusammengefügt und allen Pixeln innerhalb einer Region eindeutige Identifikationsnummern („labels“) zugeordnet. Im Folgenden beschreiben wir zwei Varianten dieser Idee: Die erste Variante (Regionenmarkierung durch *flood filling*) füllt, ausgehend von einem gegebenen Startpunkt, jeweils eine einzige Region in alle Richtungen. Bei der zweiten Methode (*sequentielle Regionenmarkierung*) wird im Gegensatz dazu das Bild von oben nach unten durchlaufen und alle Regionen auf einmal markiert. In Abschn. 11.2.2 beschreiben wir noch ein drittes Verfahren, das die Regionenmarkierung mit dem Auffinden von Konturen kombiniert.

Unabhängig vom gewählten Ansatz müssen wir auch – durch Wahl der 4er- oder 8er-Nachbarschaft (Abb. 10.5) – fixieren, unter welchen Bedingungen zwei Pixel miteinander „verbunden“ sind, denn die beiden Arten der Nachbarschaft führen i. Allg. zu unterschiedlichen Ergebnissen. Für die Regionenmarkierung nehmen wir an, dass das zunächst binäre Ausgangsbild $I(u,v)$ die Werte 0 (Hintergrund) und 1 (Vordergrund) enthält und alle weiteren Werte für Markierungen, d. h. zur Nummerierung der Regionen, genutzt werden können:

$$I(u,v) = \begin{cases} 0 & \text{Hintergrundpixel (\textit{background})} \\ 1 & \text{Vordergrundpixel (\textit{foreground})} \\ 2, 3, \ldots & \text{Regionenmarkierung (\textit{label})} \end{cases}$$

11.1.1 Regionenmarkierung durch *Flood Filling*

Der grundlegende Algorithmus für die Regionenmarkierung durch *Flood Filling* ist einfach: Zuerst wird im Bild ein noch unmarkiertes Vordergrundpixel gesucht, von dem aus der Rest der zugehörigen Region „gefüllt“ wird. Dazu werden, ausgehend von diesem Startpixel, alle zusammenhängenden Pixel der Region besucht und markiert, ähnlich einer Flutwelle (*flood*), die sich über die Region ausbreitet. Für die Realisierung der Fülloperation gibt es verschiedene Methoden, die sich vor allem dadurch unterscheiden, wie die noch zu besuchenden Pixelkoordinaten verwaltet werden. Wir beschreiben nachfolgend drei Realisierungen der FloodFill()-Prozedur: eine rekursive Version, eine iterative *Depth-first*- und eine iterative *Breadth-first*- Version:

```
1:  RegionLabeling(I)
        I: binary image (0 = background, 1 = foreground)
2:      Initialize m ← 2 (the value of the next label to be assigned).
3:      Iterate over all image coordinates ⟨u, v⟩.
4:          if I(u, v) = 1 then
5:              FloodFill(I, u, v, m)        ▷ use any of the 3 versions below
6:              m ← m + 1.
7:      return.
```

```
8:  FloodFill(I, u, v, label)                          ▷ Recursive Version
9:      if coordinate ⟨u, v⟩ is within image boundaries and I(u, v) = 1 then
10:         Set I(u, v) ← label
11:         FloodFill(I, u+1, v, label)
12:         FloodFill(I, u, v+1, label)
13:         FloodFill(I, u, v−1, label)
14:         FloodFill(I, u−1, v, label)
15:     return.

16: FloodFill(I, u, v, label)                          ▷ Depth-First Version
17:     Create an empty stack S
18:     Put the seed coordinate ⟨u, v⟩ onto the stack: Push(S, ⟨u, v⟩)
19:     while S is not empty do
20:         Get the next coordinate from the top of the stack:
                ⟨x, y⟩ ← Pop(S)
21:         if coordinate (x, y) is within image boundaries and I(x, y) = 1
                then
22:             Set I(x, y) ← label
23:             Push(S, ⟨x+1, y⟩)
24:             Push(S, ⟨x, y+1⟩)
25:             Push(S, ⟨x, y−1⟩)
26:             Push(S, ⟨x−1, y⟩)
27:     return.

28: FloodFill(I, u, v, label)                          ▷ Breadth-First Version
29:     Create an empty queue Q
30:     Insert the seed coordinate ⟨u, v⟩ into the queue: Enqueue(Q, (u, v))
31:     while Q is not empty do
32:         Get the next coordinate from the front of the queue:
                ⟨x, y⟩ ← Dequeue(Q)
33:         if coordinate ⟨x, y⟩ is within image boundaries and I(x, y) = 1
                then
34:             Set I(x, y) ← label
35:             Enqueue(Q, ⟨x+1, y⟩)
36:             Enqueue(Q, ⟨x, y+1⟩)
37:             Enqueue(Q, ⟨x, y−1⟩)
38:             Enqueue(Q, ⟨x−1, y⟩)
39:     return.
```


Algorithmus 11.1
Regionenmarkierung durch *Flood Filling*. Das binäre Eingangsbild I enthält die Werte 0 für Hintergrundpixel und 1 für Vordergrundpixel. Es werden noch unmarkierte Vordergrundpixel gesucht, von denen aus die zugehörige Region gefüllt wird. Die FloodFill()-Prozedur ist in drei verschiedenen Varianten ausgeführt: *rekursiv*, *depth-first* und *breadth-first*.

A. **Rekursive Regionenmarkierung:** Die rekursive Version (Alg. 11.1, Zeile 8) benutzt zur Verwaltung der noch zu besuchenden Bildkoordinaten keine expliziten Datenstrukturen, sondern verwendet dazu die lokalen Variablen der rekursiven Prozedur.[1] Durch die Nachbarschaftsbeziehung zwischen den Bildelementen ergibt sich eine Baumstruktur, deren Wurzel der Startpunkt innerhalb der Region bildet. Die Rekursion entspricht einem Tiefendurchlauf (*depth-first traversal*) [19, 32] dieses Baums und führt zu einem sehr einfachen Programmcode, allerdings ist die Rekursionstiefe proportional zur Größe der Region und daher der Stack-Speicher rasch erschöpft. Die Methode ist deshalb nur für sehr kleine Bilder anwendbar.

B. **Iteratives *Flood Filling* (*depth-first*):** Jede rekursive Prozedur kann mithilfe eines eigenen *Stacks* auch iterativ implementiert werden (Alg. 11.1, Zeile 16). Der Stack dient dabei zur Verwaltung der noch „offenen“ (d. h. noch nicht bearbeiteten) Elemente. Wie in der rekursiven Version (A) wird der Baum von Bildelementen im *Depth-first*-Modus durchlaufen. Durch den eigenen, dedizierten Stack (der dynamisch im so genannten *Heap-Memory* angelegt wird) ist die Tiefe des Baums nicht mehr durch die Größe des Aufruf-Stacks beschränkt.

C. **Iteratives *Flood Filling* (*breadth-first*):** Ausgehend vom Startpunkt werden in dieser Version die jeweils angrenzenden Pixel schichtweise, ähnlich einer Wellenfront expandiert (Alg. 11.1, Zeile 28). Als Datenstruktur für die Verwaltung der noch unbearbeiteten Pixelkoordinaten wird (anstelle des Stacks) eine *Queue* (Warteschlange) verwendet. Ansonsten ist das Verfahren identisch zu Version B.

Java-Implementierung

Die rekursive Version (A) des Algorithmus ist in Java praktisch 1:1 umzusetzen. Allerdings erlaubt ein normales Java-Laufzeitsystem nicht mehr als ungefähr 10.000 rekursive Aufrufe der FloodFill()-Prozedur (Alg. 11.1, Zeile 8), bevor der Speicherplatz des Aufruf-Stacks erschöpft ist. Das reicht nur für relativ kleine Bilder mit weniger als ca. 200×200 Pixel.

Prog. 11.1 zeigt die vollständige Java-Implementierung beider Varianten der iterativen FloodFill()-Prozedur. Zunächst wird in Zeile 1 eine neue Java-Klasse `Node` zur Repräsentation einzelner Pixelkoordinaten definiert.

Zur Implementierung des Stacks in der iterativen *Depth-first*-Version (B) verwenden wir als Datenstruktur die Java-Klasse `Stack` (Prog. 11.1, Zeile 8), ein Container für beliebige Java-Objekte. Für die *Queue*-

[1] Für lokale Variablen wird in Java (oder auch in C und C++) bei jedem Prozeduraufruf der entsprechende Speicherplatz dynamisch im so genannten *Stack Memory* angelegt.

```
1 class Node {
2   int x, y;
3   Node(int x, int y) { //constructor method
4     this.x = x;  this.y = y;
5   }
6 }
```

Depth-first-Variante (mit *Stack*):

```
7 void floodFill(ImageProcessor ip, int x, int y, int label) {
8   Stack<Node> s = new Stack<Node>(); // Stack
9   s.push(new Node(x,y));
10  while (!s.isEmpty()){
11    Node n = s.pop();
12    if ((n.x>=0) && (n.x<width) && (n.y>=0) && (n.y<height)
13      && ip.getPixel(n.x,n.y)==1) {
14      ip.putPixel(n.x,n.y,label);
15      s.push(new Node(n.x+1,n.y));
16      s.push(new Node(n.x,n.y+1));
17      s.push(new Node(n.x,n.y-1));
18      s.push(new Node(n.x-1,n.y));
19    }
20  }
21 }
```

Breadth-first-Variante (mit *Queue*):

```
22 void floodFill(ImageProcessor ip, int x, int y, int label) {
23   LinkedList<Node> q = new LinkedList<Node>(); // Queue
24   q.addFirst(new Node(x,y));
25   while (!q.isEmpty()) {
26     Node n = q.removeLast();
27     if ((n.x>=0) && (n.x<width) && (n.y>=0) && (n.y<height)
28       && ip.getPixel(n.x,n.y)==1) {
29       ip.putPixel(n.x,n.y,label);
30       q.addFirst(new Node(n.x+1,n.y));
31       q.addFirst(new Node(n.x,n.y+1));
32       q.addFirst(new Node(n.x,n.y-1));
33       q.addFirst(new Node(n.x-1,n.y));
34     }
35   }
36 }
```

Programm 11.1
Flood Filling (Java-Implementierung). Die *Depth-first*-Variante verwendet als Datenstruktur die Java-Klasse `Stack` mit den Methoden `push()`, `pop()` und `isEmpty()`.

Datenstruktur in der *Breadth-first*-Variante (C) verwenden wir die Java-Klasse `LinkedList`[2] mit den Methoden `addFirst()`, `removeLast()` und `isEmpty()` (Prog. 11.1, Zeile 23). Beide Container-Klassen sind durch die Angabe `<Node>` auf den Objekttyp `Node` parametrisiert, d. h. sie können nur Objekte dieses Typs aufnehmen.[3]

Abb. 11.2 illustriert den Ablauf der Regionenmarkierung in beiden Varianten anhand eines konkreten Beispiels mit einer Region, wobei der Startpunkt („seed point“) – der normalerweise am Rand der Kontur liegt – willkürlich im Inneren der Region gewählt wurde. Deutlich ist zu sehen, dass die *Depth-first*-Methode zunächst entlang *einer* Richtung (in diesem Fall horizontal nach links) bis zum Ende der Region vorgeht und erst dann die übrigen Richtungen berücksichtigt. Im Gegensatz dazu breitet sich die Markierung bei der *Breadth-first*-Methode annähernd gleichförmig, d. h. Schicht um Schicht, in alle Richtungen aus.

Generell ist der Speicherbedarf bei der *Breadth-first*-Variante des *Flood-fill*-Verfahrens deutlich niedriger als bei der *Depth-first*-Variante. Für das Beispiel in Abb. 11.2 mit der in Prog. 11.1 gezeigten Implementierung erreicht die Größe des *Stacks* in der *Depth-first*-Variante 28.822 Elemente, während die *Queue* der *Breadth-first*-Variante nur maximal 438 Knoten aufnehmen muss.

11.1.2 Sequentielle Regionenmarkierung

Die sequentielle Regionenmarkierung ist eine klassische, nicht rekursive Technik, die in der Literatur auch als „region labeling“ bekannt ist. Der Algorithmus besteht im Wesentlichen aus zwei Schritten: (1) einer vorläufigen Markierung der Bildregionen und (2) der Auflösung von mehrfachen Markierungen innerhalb derselben Region. Das Verfahren ist (vor allem im 2. Schritt) relativ komplex, aber wegen seines moderaten Speicherbedarfs durchaus verbreitet, bietet in der Praxis allerdings gegenüber einfacheren Methoden kaum Vorteile. Der gesamte Ablauf ist in Alg. 11.2 dargestellt.

Schritt 1: Vorläufige Markierung

Im ersten Schritt des „region labeling“ wird das Bild sequentiell von links oben nach rechts unten durchlaufen und dabei jedes Vordergrundpixel mit einer vorläufigen Markierung versehen. Je nach Definition der Nachbarschaftsbeziehung werden dabei für jedes Pixel (u, v) die Umgebungen

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

[2] Die Klasse `LinkedList` ist Teil des *Java Collection Frameworks* (s. auch Anhang 2.2).

[3] Generische Typen und die damit verbundene Möglichkeit zur Parametrisierung von Klassen gibt es erst seit Java 5 (1.5).

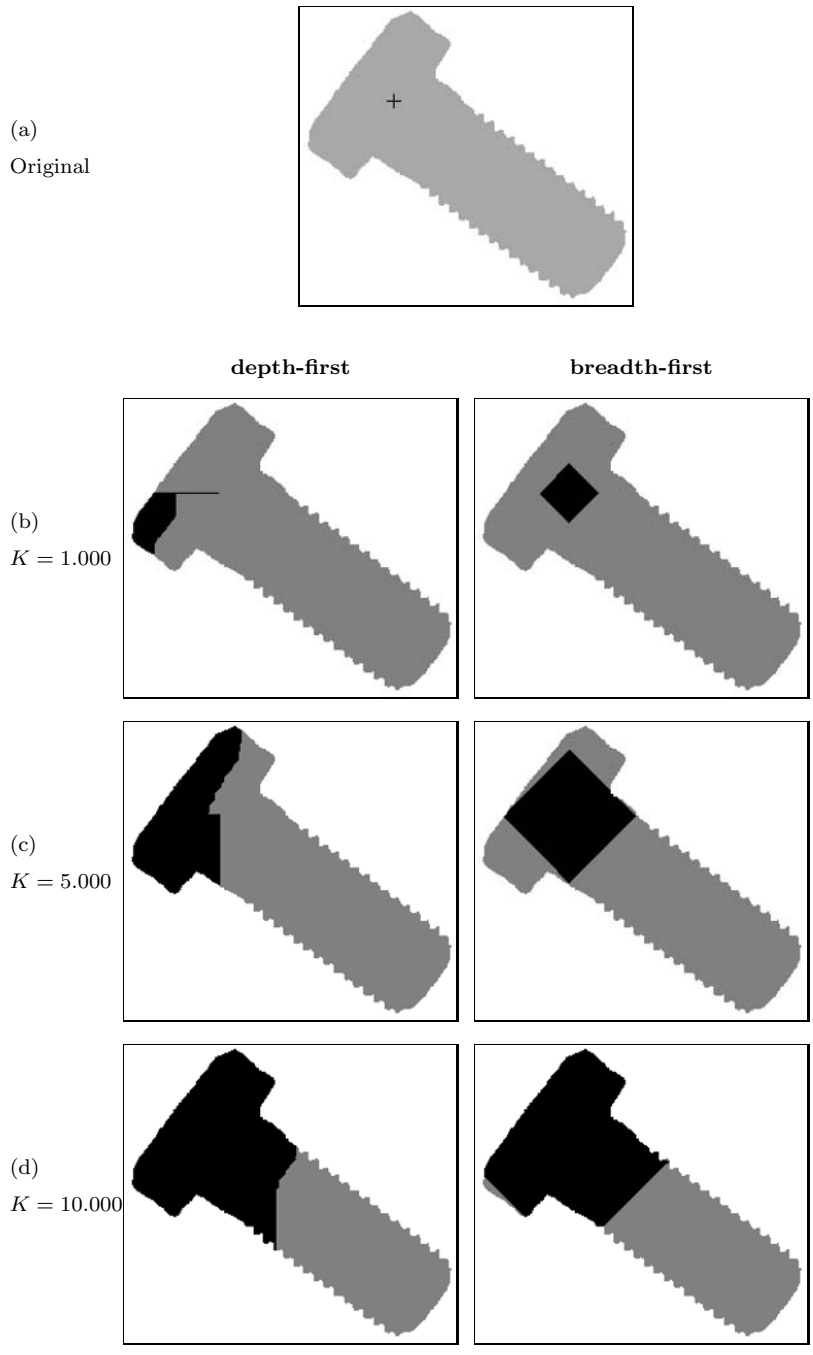


Abbildung 11.2
Iteratives *Flood Filling* – Vergleich zwischen *Depth-first*- und *Breadth-first*-Variante. Der mit + markierte Startpunkt im Originalbild (a) ist willkürlich gewählt. Zwischenergebnisse des *Flood-fill*-Algorithmus nach 1.000, 5.000 und 10.000 markierten Bildelementen (b–d). Das Bild hat eine Größe von 250×242 Pixel.

Algorithmus 11.2
Sequentielle Regionenmarkierung. Das binäre Eingangsbild I enthält die Werte $I(u,v) = 0$ für Hintergrundpixel und $I(u,v) = 1$ für Vordergrundpixel (Regionen). Die resultierenden Markierungen haben die Werte $2 \ldots m-1$.

1: SequentialLabeling(I)
I: binary image (0 = *background*, 1 = *foreground*)
2: Pass 1 – Assign Initial Labels:
3: Initialize $m \leftarrow 2$ (the value of the next label to be assigned).
4: Create an empty set $\mathcal{C}$ to hold the collisions: $\mathcal{C} \leftarrow \{\}$.
5: **for** $v \leftarrow 0 \ldots H\text{–}1$ **do** ▷ H = height of image I
6: **for** $u \leftarrow 0 \ldots W\text{–}1$ **do** ▷ W = width of image I
7: **if** $I(u,v) = 1$ **then** do one of:
8: **if** all neighbors are background pixels (all $n_i = 0$) **then**
9: $I(u,v) \leftarrow m$.
10: $m \leftarrow m + 1$.
11: **else if** exactly *one* of the neighbors has a label value $n_k > 1$ **then**
12: set $I(u,v) \leftarrow n_k$
13: **else if** *several* neighbors have label values $n_j > 1$ **then**
14: Select one of them as the new label:
$I(u,v) \leftarrow k \in \{n_j\}$.
15: **for** all other neighbors with label values $n_i > 1$ and $n_i \neq k$ **do**
16: register the pair $\langle n_i, k\rangle$ as a label collision:
$\mathcal{C} \leftarrow \mathcal{C} \cup \{\langle n_i, k\rangle\}$.

Remark: The image I now contains label values $0, 2, \ldots m\text{–}1$.

17: Pass 2 – Resolve Label Collisions:
18: Let $\mathcal{L} = \{2, 3, \ldots m\text{–}1\}$ be the set of preliminary region labels.
19: Create a partitioning of $\mathcal{L}$ as a *vector of sets*, one set for each label value: $\mathcal{R} \leftarrow [\mathcal{R}_2, \mathcal{R}_3, \ldots, \mathcal{R}_{m-1}] = [\{2\}, \{3\}, \{4\}, \ldots, \{m\text{–}1\}]$, so $\mathcal{R}_i = \{i\}$ for all $i \in \mathcal{L}$.
20: **for all** collisions $\langle a, b\rangle \in \mathcal{C}$ **do**
21: Find in $\mathcal{R}$ the sets $\mathcal{R}_a$, $\mathcal{R}_b$ containing the labels a, b, resp.:
$\mathcal{R}_a \leftarrow$ the set which currently contains label a
$\mathcal{R}_b \leftarrow$ the set which currently contains label b
22: **if** $\mathcal{R}_a \neq \mathcal{R}_b$ (a and b are contained in different sets) **then**
23: Merge sets $\mathcal{R}_a$ and $\mathcal{R}_b$ by moving all elements of $\mathcal{R}_b$ to $\mathcal{R}_a$:
$\mathcal{R}_a \leftarrow \mathcal{R}_a \cup \mathcal{R}_b$
$\mathcal{R}_b \leftarrow \{\}$

Remark: All equivalent label values (i.e., all labels of pixels in the same region) are now contained in the same sets within $\mathcal{R}$.

24: Pass 3: Relabel the Image:
25: Iterate through all image pixels (u,v):
26: **if** $I(u,v) > 1$ **then**
27: Find the set $\mathcal{R}_i$ in $\mathcal{R}$ which contains label $I(u,v)$.
28: Choose one unique, representative element k from the set $\mathcal{R}_i$ (e.g., the minimum value, $k \leftarrow \min(\mathcal{S})$).
29: Replace the image label: $I(u,v) \leftarrow k$.
30: **return** the labeled image I.

für eine 4er- bzw. 8er-Nachbarschaft berücksichtigt ($\times$ markiert das aktuelle Pixel an der Position (u, v)). Im Fall einer 4er-Nachbarschaft werden also nur die beiden Nachbarn $N_1 = I(u-1, v)$ und $N_2 = I(u, v-1)$ untersucht, bei einer 8er-Nachbarschaft die insgesamt vier Nachbarn $N_1 \ldots N_4$. Wir verwenden für das nachfolgende Beispiel eine 8er-Nachbarschaft und das Ausgangsbild in Abb. 11.3 (a).

Fortpflanzung von Markierungen

Wir nehmen an, dass die Bildwerte $I(u, v) = 0$ Hintergrundpixel und die Werte $I(u, v) = 1$ Vordergrundpixel darstellen. Nachbarpixel, die außerhalb der Bildmatrix liegen, werden als Hintergrund betrachtet. Die Nachbarschaftsregion $\mathcal{N}(u, v)$ wird nun in horizontaler und anschließend vertikaler Richtung über das Bild geschoben, ausgehend von der linken, oberen Bildecke. Sobald das aktuelle Bildelement $I(u, v)$ ein Vordergrundpixel ist, erhält es entweder eine neue Regionsnummer, oder es wird – falls bereits einer der zuvor besuchten Nachbarn in $\mathcal{N}(u, v)$ auch ein Vordergrundpixel ist – dessen Regionsnummer übernommen. Bestehende Regionsnummern (Markierungen) breiten sich dadurch von links nach rechts bzw. von oben nach unten im Bild aus (Abb. 11.3 (b–c)).

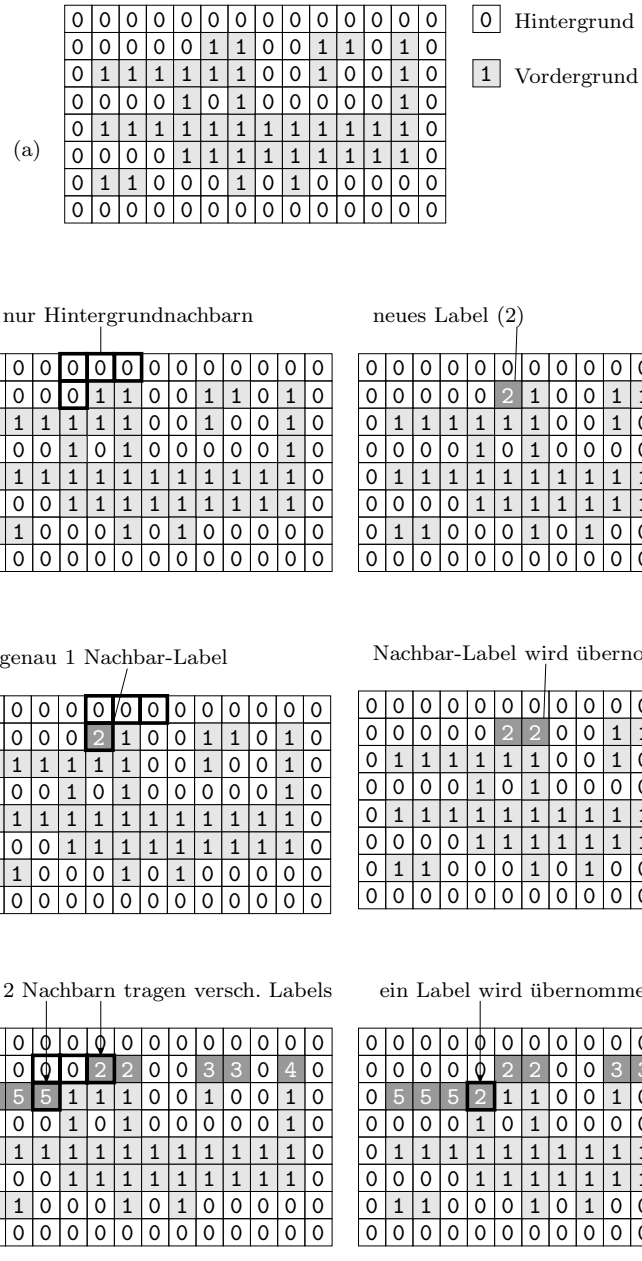
Kollidierende Markierungen

Falls zwei (oder mehr) Nachbarn bereits zu *verschiedenen* Regionen gehören, besteht eine Kollision von Markierungen, d. h., Pixel innerhalb einer zusammenhängenden Region tragen unterschiedliche Markierungen. Zum Beispiel erhalten bei einer U-förmigen Region die Pixel im linken und rechten Arm anfangs unterschiedliche Markierungen, da zunächst ja nicht sichtbar ist, dass sie tatsächlich zu einer gemeinsamen Region gehören. Die beiden Markierungen werden sich in der Folge unabhängig nach unten fortpflanzen und schließlich im unteren Teil des U kollidieren (Abb. 11.3 (d)).

Wenn zwei Markierungen a, b zusammenstoßen, wissen wir, dass sie „äquivalent“ sind und die beiden zugehörigen Bildregionen verbunden sind, also tatsächlich eine gemeinsame Region bilden. Diese Kollissionen werden im ersten Schritt nicht unmittelbar behoben, sondern nur in geeigneter Form bei ihrem Auftreten „registriert“ und erst in Schritt 2 des Algorithmus behandelt. Abhängig vom Bildinhalt können nur wenige oder auch sehr viele Kollisionen auftreten und ihre Gesamtanzahl steht erst am Ende des ersten Durchlaufs fest. Zur Verwaltung der Kollisionen benötigt man daher dynamischer Datenstrukturen, wie z. B. verkettete Listen oder *Hash*-Tabellen.

Als Ergebnis des ersten Schritts sind alle ursprünglichen Vordergrundpixel durch eine vorläufige Markierung ersetzt und die aufgetretenen Kollisionen zwischen zusammengehörigen Markierungen sind in einer geeigneten Form registriert. Das Beispiel in Abb. 11.4 zeigt das Ergebnis nach Schritt 1: alle Vordergrundpixel sind mit vorläufigen Markierungen versehen (Abb. 11.4 (a)), die aufgetretenen (als Kreise angezeigten)

Abbildung 11.3
Sequentielle Regionenmarkierung – Fortpflanzung der Markierungen. Ausgangsbild (a). Das erste Vordergrundpixel [**1**] wird in (b) gefunden: Alle Nachbarn sind Hintergrundpixel [**0**], das Pixel erhält die erste Markierung [**2**]. Im nächsten Schritt (c) ist genau *ein* Nachbarpixel mit dem Label **2** markiert, dieser Wert wird daher übernommen. In (d) sind *zwei* Nachbarpixel mit Label (**2** und **5**) versehen, einer dieser Werte wird übernommen und die Kollision $\langle \mathbf{2}, \mathbf{5} \rangle$ wird registriert.



Kollisionen zwischen Markierungen $\langle 2,4\rangle$, $\langle 2,5\rangle$ und $\langle 2,6\rangle$, wurden registriert. Die Markierungen (*labels*) $\mathcal{L} = \{2,3,4,5,6,7\}$ und Kollisionen $\mathcal{C} = \{\langle 2,4\rangle, \langle 2,5\rangle, \langle 2,6\rangle\}$ entsprechen den Knoten bzw. Kanten eines ungerichteten Graphen (Abb. 11.4 (b)).

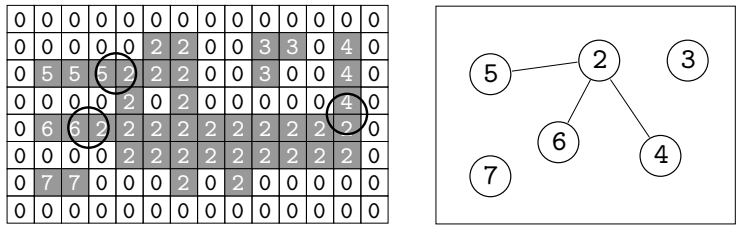


Abbildung 11.4
Sequentielle Regionenmarkierung – Ergebnis nach Schritt 1. Label-Kollisionen sind als Kreise angezeigt (a), Labels und Kollisionen entsprechen Knoten bzw. Kanten des Graphen in (b).

Schritt 2: Auflösung der Kollisionen

Aufgabe des zweiten Schritts ist die Auflösung der im ersten Schritt kollidierten Markierungen und die Verbindung der zugehörigen Teilregionen. Dieser Prozess ist nicht trivial, denn zwei unterschiedlich markierte Regionen können miteinander in transitiver Weise über eine dritte Region „verbunden“ sein bzw. im Allgemeinen über eine ganze Kette von Kollisionen. Tatsächlich ist diese Aufgabe identisch zum Problem des Auffindens zusammenhängender Teile von Graphen (*connected components problem*) [19], wobei die in Schritt 1 zugewiesenen Markierungen (*labels*) $\mathcal{L}$ den „Knoten“ des Graphen und die festgestellten Kollisionen $\mathcal{C}$ den „Kanten“ entsprechen (Abb. 11.4 (b)).

Nach dem Zusammenführen der unterschiedlichen Markierungen werden die Pixel jeder zusammenhängenden Region durch eine gemeinsame Markierung (z. B. die niedrigste der vorläufigen Markierungen innerhalb der Region) ersetzt (Abb. 11.5).

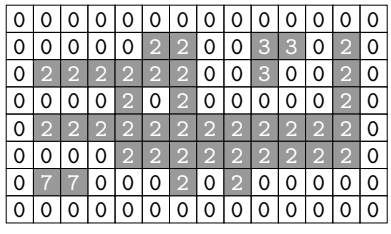


Abbildung 11.5
Sequentielle Regionenmarkierung – Endergebnis nach Schritt 2. Alle äquivalenten Markierungen wurden durch die jeweils niedrigste Markierung innerhalb einer Region ersetzt.

11.1.3 Regionenmarkierung – Zusammenfassung

Wir haben in diesem Abschnitt mehrere funktionsfähige Algorithmen zum Auffinden von zusammenhängenden Bildregionen beschrieben. Die zunächst attraktive (und elegante) Idee, die einzelnen Regionen von einem Startpunkt aus durch rekursives „flood filling“ (Abschn. 11.1.1) zu markieren, ist wegen der beschränkten Rekursionstiefe in der Praxis meist nicht anwendbar. Das klassische, sequentielle „region labeling“ (Abschn. 11.1.2) ist hingegen relativ komplex und bietet auch keinen echten Vorteil gegenüber der iterativen *Depth-first-* und *Breadth-first-*Methode, wobei letztere bei großen und komplexen Bildern generell am effektivsten ist. Abb. 11.6 zeigt ein Beispiel für eine fertige Regionenmarkierung anhand des Ausgangsbilds aus Abb. 11.1.

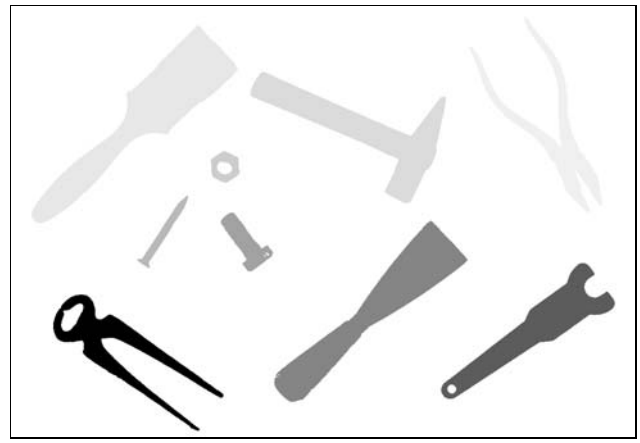


Abbildung 11.6
Beispiel für eine fertige Regionenmarkierung. Die Pixel innerhalb jeder der Region sind auf den zugehörigen, fortlaufend vergebenen Markierungswert $2, 3, \ldots 10$ gesetzt und als Grauwerte dargestellt.

11.2 Konturen von Regionen

Nachdem die Regionen eines Binärbilds gefunden sind, ist der nachfolgende Schritt häufig das Extrahieren der Umrisse oder Konturen dieser Regionen. Wie vieles andere in der Bildverarbeitung erscheint diese Aufgabe zunächst nicht schwierig – man folgt einfach den Rändern einer Region. Wie wir sehen werden, erfordert die algorithmische Umsetzung aber doch eine sorgfältige Überlegung, und tatsächlich ist das Finden von Konturen eine klassische Aufgabenstellung im Bereich der Bildanalyse.

11.2.1 Äußere und innere Konturen

Wie wir bereits in Abschn. 10.2.7 gezeigt haben, kann man die Pixel an den Rändern von binären Regionen durch morphologische Operationen und Differenzbildung auf einfache Weise identifizieren. Dieses Verfahren

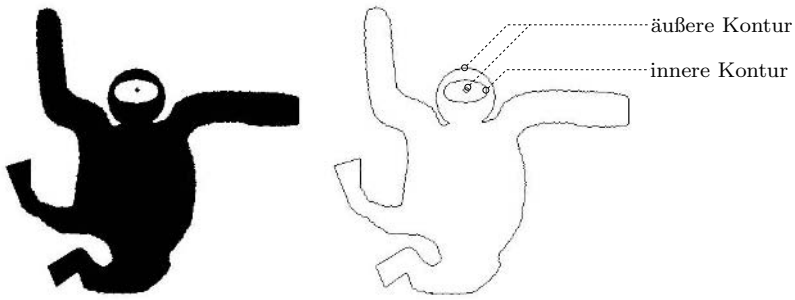


Abbildung 11.7
Binärbild mit äußeren und inneren Konturen. Äußere Konturen liegen an der Außenseite von Vordergrundregionen (dunkel). Innere Konturen umranden die Löcher von Regionen, die rekursiv weitere Regionen enthalten können.

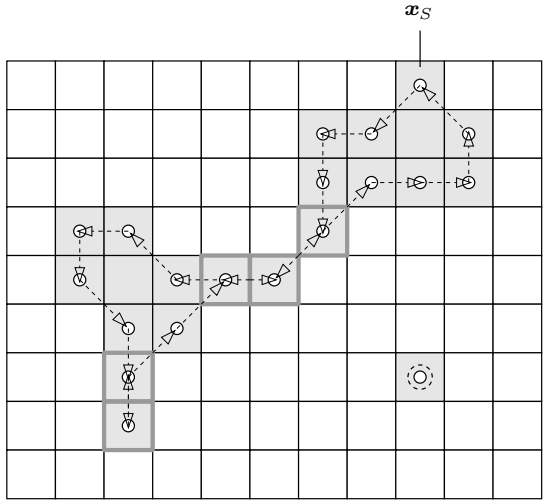


Abbildung 11.8
Pfad entlang einer Kontur als geordnete Folge von Pixelkoordinaten, ausgehend von einem beliebigen Startpunkt $\boldsymbol{x}_S$. Pixel können im Pfad mehrfach enthalten sein und auch Regionen, die nur aus einem isolierten Pixel bestehen (rechts unten), besitzen eine Kontur.

markiert die Pixel entlang der Konturen und ist z. B. für die Darstellung nützlich. Hier gehen wir jedoch einen Schritt weiter und bestimmen die Kontur jeder Region als *geordnete Folge* ihrer Randpixel. Zu beachten ist dabei, dass zusammengehörige Bildregionen zwar nur eine *äußere* Kontur aufweisen können, jedoch – innerhalb von Löchern – auch beliebig viele *innere* Konturen besitzen können. Innerhalb dieser Löcher können sich wiederum kleinere Regionen mit zugehörigen äußeren Konturen befinden, die selbst wieder Löcher aufweisen können, usw. (Abb. 11.7). Eine weitere Komplikation ergibt sich daraus, dass sich Regionen an manchen Stellen auf die Breite eines einzelnen Pixels verjüngen können, ohne ihren Zusammenhalt zu verlieren, sodass die zugehörige Kontur dieselben Pixel mehr als einmal in unterschiedlichen Richtungen durchläuft (Abb. 11.8). Wird daher eine Kontur von einem Startpunkt $\boldsymbol{x}_S$ beginnend durchlaufen, so reicht es i. Allg. nicht aus, nur wieder bis zu diesem Startpunkt zurückzukehren, sondern es muss auch die aktuelle Konturrichtung beachtet werden.

Eine Möglichkeit zur Bestimmung der Konturen besteht darin, zunächst – wie im vorherigen Abschnitt (11.1) beschrieben – die zusammengehörigen Vordergrundregionen zu identifizieren und anschließend die äußere Kontur jeder gefundenen Region zu umrunden, ausgehend von einem beliebigen Randpixel der Region. In ähnlicher Weise wären dann die inneren Konturen aus den Löchern der Regionen zu ermitteln. Für diese Aufgabe gibt es eine Reihe von Algorithmen, wie beispielsweise in [71], [64, S. 142–148] oder [76, S. 296] beschrieben.

Als Alternative zeigen wir nachfolgend ein *kombiniertes* Verfahren, das im Unterschied zum traditionellen Ansatz die Konturfindung und die Regionenmarkierung verbindet.

11.2.2 Kombinierte Regionenmarkierung und Konturfindung

Dieses Verfahren aus [17] verbindet Konzepte der sequentiellen Regionenmarkierung (Abschn. 11.1) und der traditionellen Konturverfolgung, um in einem Bilddurchlauf beide Aufgaben gleichzeitig zu erledigen. Es werden sowohl äußere wie innere Konturen sowie die zugehörigen Regionen identifiziert und markiert. Der Algorithmus benötigt keine komplizierten Datenstrukturen und ist im Vergleich zu ähnlichen Verfahren sehr effizient.

Die grundlegende Idee des Algorithmus ist einfach und nachfolgend skizziert. Demgegenüber ist die konkrete Realisierung im Detail relativ aufwendig und daher als vollständige Implementierung als Java-Programm bzw. als ImageJ-Plugin in Anhang 4.2 (S. 487–496) aufgelistet. Die wichtigsten Schritte des Verfahrens sind in Abb. 11.9 illustriert:

1. Das Binärbild I wird – ähnlich wie bei der sequentiellen Regionenmarkierung (Alg. 11.2) – von links oben nach rechts unten durchlaufen. Damit ist sichergestellt, dass alle Pixel im Bild berücksichtigt werden und am Ende eine entsprechende Markierung tragen.
2. An der aktuellen Bildposition können folgende Situationen auftreten:

   Fall A: Der Übergang von einem Hintergrundpixel auf ein bisher nicht markiertes Vordergrundpixel A bedeutet, dass A am Außenrand einer neuen Region liegt. Eine neue Marke (*label*) wird erzeugt und die zugehörige *äußere* Kontur wird (durch die Prozedur TraceContour in Alg. 11.3) umfahren und markiert (Abb. 11.9 (a)). Zudem werden auch alle unmittelbar angrenzenden Hintergrundpixel (mit dem Wert -1) markiert.

   Fall B: Der Übergang von einem Vordergrundpixel B auf ein nicht markiertes Hintergrundpixel bedeutet, dass B am Rand einer *inneren* Kontur liegt (Abb. 11.9 (b)). Ausgehend von B wird die innere Kontur umfahren und deren Pixel mit der Markierung der einschließenden Region versehen (Abb. 11.9 (c)). Auch alle angrenzenden Hintergrundpixel werden wiederum (mit dem Wert -1) markiert.

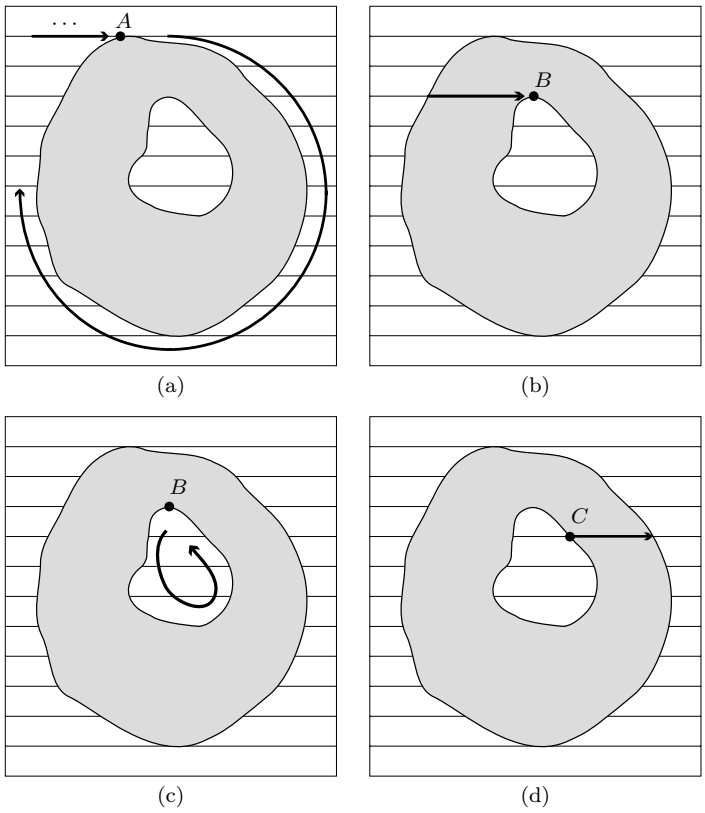


Abbildung 11.9
Kombinierte Regionenmarkierung und Konturverfolgung (nach [17]). Das Bild wird von links oben nach rechts unten zeilenweise durchlaufen. In (a) ist der erste Punkt A am äußeren Rand einer Region gefunden. Ausgehend von A werden die Randpixel entlang der äußeren Kontur besucht und markiert, bis A wieder erreicht ist. In (b) ist der erste Punkt B auf einer inneren Kontur gefunden. Die innere Kontur wird wiederum bis zum Punkt B zurück durchlaufen und markiert (c). In (d) wird ein bereits markierter Punkt C auf einer inneren Kontur gefunden. Diese Markierung wird entlang der Bildzeile innerhalb der Region fortgepflanzt.

Fall C: Bei einem Vordergrundpixel, das nicht an einer Kontur liegt, ist jedenfalls das linke Nachbarpixel bereits markiert (Abb. 11.9 (d)). Diese Markierung wird für das aktuelle Pixel übernommen.

In Alg. 11.3–11.4 ist der gesamte Vorgang nochmals exakt beschrieben. Die Prozedur CombinedContourLabeling durchläuft das Bild zeilenweise und ruft die Prozedur TraceContour auf, sobald eine neue innere oder äußere Kontur zu bestimmen ist. Die Markierungen der Bildelemente entlang der Konturen sowie der umliegenden Hintergrundpixel werden im „label map“ *LM* durch die Methode FindNextNode (Alg. 11.4) eingetragen.

11.2.3 Implementierung

Die vollständige Implementierung in Java bzw. ImageJ ist aus Platzgründen in Anhang 4.2 (S. 487–496) zusammengestellt. Sie folgt im Wesentlichen der Beschreibung in Alg. 11.3–11.4, weist jedoch einige zusätzliche Details auf:[4]

[4] Nachfolgend sind zu den im Algorithmus verwendeten Symbolen jeweils die Bezeichnungen im Java-Programm in Klammern angegeben.

Algorithmus 11.3
Kombinierte Konturfindung und Regionenmarkierung. Die Prozedur CombinedContourLabeling erzeugt aus dem Binärbild I eine Menge von Konturen sowie ein Array mit der Regionenmarkierung aller Bildpunkte. Wird ein neuer Konturpunkt (äußere oder innere Kontur) gefunden, dann wird die eigentliche Kontur als Folge von Konturpunkten durch den Aufruf von TraceContour (Zeile 20 bzw. Zeile 27) ermittelt. TraceContour selbst ist in Alg. 11.4 beschrieben.

```
1:  CombinedContourLabeling (I)
        I: binary image
2:      Create an empty set of contours: C ← {}
3:      Create a label map LM of the same size as I and initialize:
4:      for all (u, v) do
5:          LM(u, v) ← 0                                  ▷ label map LM
6:      R ← 0                                             ▷ region counter R
7:      Scan the image from left to right, top to bottom:
8:      for v ← 0 ... N−1 do
9:          L_c ← 0                                       ▷ current label L_c
10:         for u ← 0 ... M−1 do
11:             if I(u, v) is a foreground pixel then
12:                 if (L_c ≠ 0) then                     ▷ continue existing region
13:                     LM(u, v) ← L
14:                 else
15:                     L_c ← LM(u, v)
16:                     if (L_c = 0) then                 ▷ hit new outer contour
17:                         R ← R + 1
18:                         L_c ← R
19:                         x_S ← (u, v)
20:                         C_outer ← TraceContour(x_S, 0, L_c, I, LM)
21:                         C ← C ∪ {C_outer}             ▷ collect new contour
22:                         LM(u, v) ← L_c
23:             else                                      ▷ I(u, v) is a background pixel
24:                 if (L ≠ 0) then
25:                     if (LM(u, v) = 0) then            ▷ hit new inner contour
26:                         x_S ← (u−1, v)
27:                         C_inner ← TraceContour(x_S, 1, L_c, I, LM)
28:                         C ← C ∪ {C_inner}             ▷ collect new contour
29:                     L ← 0
30:     return (C, LM).                                   ▷ return the set of contours and the label map
```

Fortsetzung in Alg. 11.4 ▷▷

- Zunächst wird das Bild I (`pixelMap`) und das zugehörige *label map* LM (`labelMap`) an allen Rändern um ein zusätzliches Pixel vergrößert, wobei im Bild I diese Pixel als Hintergrund markiert werden. Dies vereinfacht die Verfolgung der Konturen, da bei der Behandlung der Ränder nun keine besonderen Vorkehrungen mehr notwendig sind.
- Die gefundenen Konturen werden in einem Objekt der Klasse `ContourSet` zusammengefasst, und zwar getrennt in äußere und innere Konturen. Diese sind wiederum Objekte der Klassen `OuterContour` bzw. `InnerContour` mit der gemeinsamen Überklasse `Contour`. Jede Kontur besteht aus einer geordneten Folge von Koordinatenpunkten der Klasse `Node` (Definition auf S. 199). Als dynamische Datenstruktur für die Koordinatenpunkte sowie für die Menge der äußeren und

Algorithmus 11.4
Kombinierte Konturfindung und Regionenmarkierung (*Fortsetzung*). Die Prozedur TraceContour durchläuft die zum Startpunkt $\boldsymbol{x}_S$ gehörigen Konturpunkte, beginnend mit der Suchrichtung $d_S = 0$ (äußere Kontur) oder $d_S = 1$ (innere Kontur). Dabei werden alle Konturpunkte sowie benachbarte Hintergrundpunkte im Label-Array LM markiert. TraceContour verwendet FindNextNode(), um zu einem gegebenen Punkt $\boldsymbol{x}_c$ den nachfolgenden Konturpunkt zu bestimmen (Zeile 10). Die Funktion Delta() dient lediglich zur Bestimmung der Folgekoordinaten in Abhängigkeit von der aktuellen Suchrichtung d.

```
 1: TraceContour(x_S, d_S, L_C, I, LM)
        x_S: start position
        d_S: initial search direction
        L_C: label for this contour
        I: image
        LM: label map
 2:     Create an empty contour C
 3:     (x_T, d) ← FindNextNode(x_S, d_s, I, LM)
 4:     Append(C, x_T)                          ▷ add x_T to contour C
 5:     x_p ← x_S                               ▷ previous position x_p = (u_p, v_p)
 6:     x_c ← x_T                               ▷ current position x_c = (u_c, v_c)
 7:     done ← (x_S = x_T)                      ▷ isolated pixel?
 8:     while (¬done) do
 9:         LM(u_c, v_c) ← L_C
10:         (x_n, d) ← FindNextNode(x_c, (d+6) mod 8, I, LM)
11:         x_p ← x_c
12:         x_c ← x_n
13:         done ← (x_p = x_S ∧ x_c = x_T)      ▷ back at starting position?
14:         if (¬done) then
15:             Append(C, x_n)                  ▷ add point x_n to contour C
16:     return C .                              ▷ return this contour

17: FindNextNode(x_c, d, I, LM)
        x_c: original position, d: search direction, I: image, LM: label map
18:     for i ← 0...6 do                        ▷ search in 7 directions
19:         x' ← x_c + Delta(d)                 ▷ x' = (u', v')
20:         if I(u', v') is a background pixel then
21:             LM(u', v') ← −1                 ▷ mark background as visited (−1)
22:             d ← (d + 1) mod 8
23:         else                                ▷ found a non-background pixel at x'
24:             return (x', d)
25:     return (x_c, d) .                       ▷ found no next node, return start position
```

26: Delta$(d) = (\Delta x, \Delta y)$, wobei

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

inneren Konturen wird die Java-Container-Klasse `ArrayList` (parametrisiert auf den Typ `Node`) verwendet.

- Die Methode `traceContour()` in Prog. 11.4 durchläuft eine äußere oder innere Kontur, beginnend bei einem Startpunkt $\boldsymbol{x}_S$ (`xS`, `yS`). Dafür wird zunächst die Methode `findNextNode()` aufgerufen, um den auf $\boldsymbol{x}_S$ folgenden Konturpunkt $\boldsymbol{x}_T$ (`xT`, `yT`) zu bestimmen:
  - Für den Fall, dass kein nachfolgender Punkt gefunden wird, gilt $\boldsymbol{x}_S = \boldsymbol{x}_T$ und es handelt sich um eine Region (Kontur), bestehend aus einem isolierten Pixel. In diesem Fall ist `traceContour()` fertig.
  - Andernfalls werden durch wiederholten Aufruf von `findNextNode()` die übrigen Konturpunkte schrittweise durchlaufen, wo-

Programm 11.2
Beispiel für die Verwendung der Klasse `ContourTracer`.

```
1 import java.util.ArrayList;
2 ...
3 public class ContourTracingPlugin_ implements PlugInFilter {
4   public void run(ImageProcessor ip) {
5     ContourTracer tracer = new ContourTracer(ip);
6     ContourSet cs = tracer.getContours();
7     // process outer and inner contours:
8     ArrayList<Contour> outer = cs.outerContours;
9     ArrayList<Contour> inner = cs.innerContours;
10    ...
11  }
12 }
```

bei jeweils ein aufeinander folgendes Paar von Punkten, der aktuelle (*current*) Punkt $\boldsymbol{x}_c$ (`xC`, `yC`) und der vorherige (*previous*) Punkt $\boldsymbol{x}_p$ (`xP`, `yP`), mitgeführt werden. Erst wenn *beide* Punkte mit den ursprünglichen Startpunkten der Kontur, $\boldsymbol{x}_S$ und $\boldsymbol{x}_T$, übereinstimmen, ist die Kontur vollständig durchlaufen.

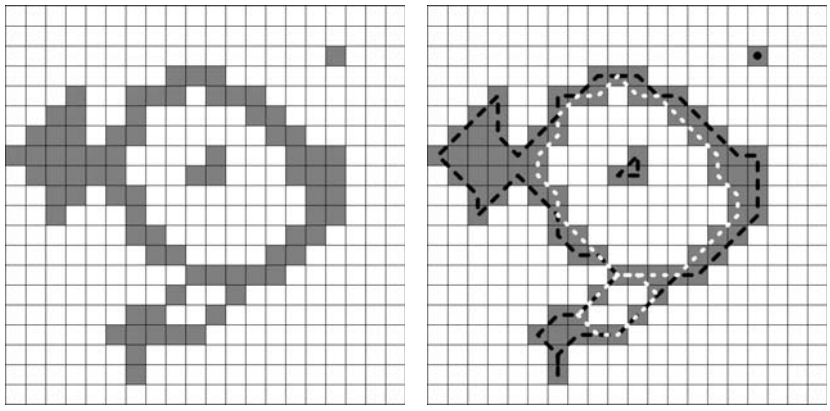
- Die Methode `findNextNode()` bestimmt den zum aktuellen Punkt $\boldsymbol{x}_c$ (`Xc`) nachfolgenden Konturpunkt, wobei die anfängliche Such*richtung* d (`dir`) von der Lage zum vorherigen Konturpunkt abhängt. Ausgehend von der ersten Suchrichtung werden maximal 7 Nachbarpunkte (alle außer dem vorherigen Konturpunkt) im Uhrzeigersinn untersucht, bis ein Vordergrundpixel (= Nachfolgepunkt) gefunden ist. Gleichzeitig werden alle gefundenen Hintergrundpixel mit dem Wert -1 im *label map LM* (`labelMap`) markiert, um wiederholte Besuche zu vermeiden. Wird unter den 7 möglichen Nachbarn kein gültiger Nachfolgepunkt gefunden, dann gibt `findNextNode()` den Ausgangspunkt $\boldsymbol{x}_c$ zurück.

Die Hauptfunktionalität ist bei dieser Implementierung in der Klasse `ContourTracer` verpackt, deren prinzipielle Verwendung innerhalb der `run()`-Methode eines ImageJ-Plugins in Prog. 11.2 gezeigt ist.

Die Implementierung in Anhang 4.2 zeigt außerdem die Darstellung der Konturen als Vektorgrafik mithilfe der Klasse `ContourOverlay`. Auf diese Weise können grafische Strukuren, die kleiner bzw. dünner sind als Bildpixel, über einem Rasterbild in ImageJ angezeigt werden.

11.2.4 Beispiele

Der kombinierte Algorithmus zur Regionenmarkierung und Konturverfolgung ist aufgrund seines bescheidenen Speichererbedarfs auch für große Binärbilder problemlos und effizient anwendbar. Abb. 11.10 zeigt ein Beispiel anhand eines vergrößerten Bildausschnitts, in dem mehrere spezielle Situationen in Erscheinung treten, wie einzelne, isolierte Pixel und Verdünnungen, die der Konturpfad in beiden Richtungen passieren



(a) (b)

Abbildung 11.10
Kombinierte Konturfindung und Regionenmarkierung. Originalbild, Vordergrundpixel sind grau markiert (a). Gefundene Konturen (b), mit schwarzen Linien für äußere und weißen Linien für innere Konturen. Die Konturen sind durch Polygone gekennzeichnet, deren Knoten jeweils im Zentrum eines Konturpixels liegen. Konturen für isolierte Pixel (z. B. rechts oben in (b)) sind durch kreisförmige Punkte dargestellt.

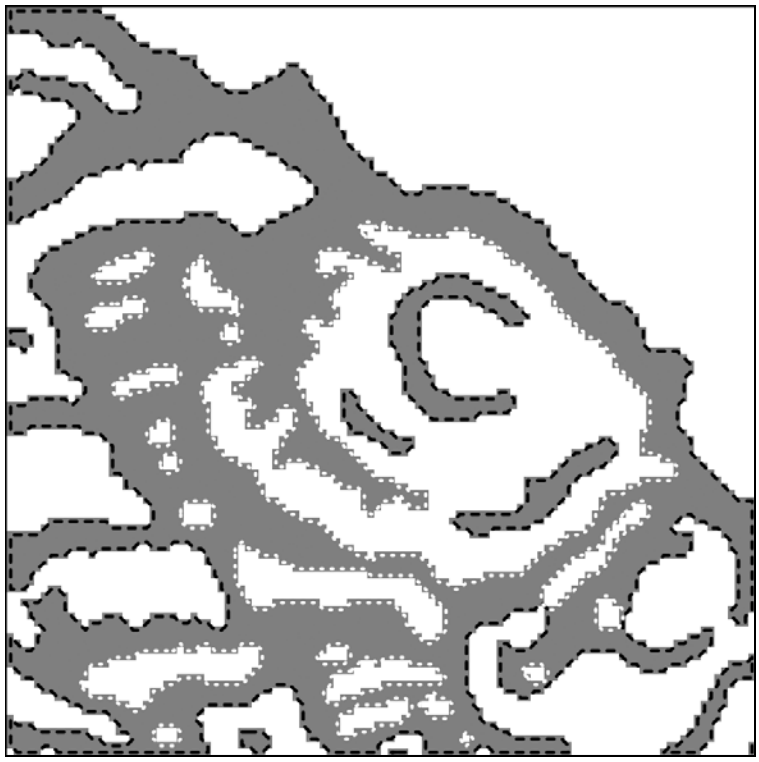


Abbildung 11.11
Bildausschnitt mit Beispielen von komplexen Konturen (Originalbild in Abb. 10.11). Äußere Konturen sind schwarz, innere Konturen weiß markiert.

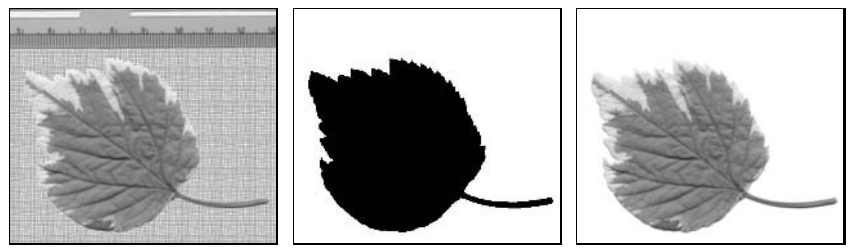
muss. Die Konturen selbst sind durch Polygonzüge zwischen den Mittelpunkten der enthaltenen Pixel dargestellt, äußere Konturen sind schwarz und innere Konturen sind weiß gezeichnet. Regionen bzw. Konturen, die nur aus einem Pixel bestehen, sind durch Kreise in der entsprechenden Farbe markiert. Abb. 11.11 zeigt das Ergebnis für einen größeren Ausschnitt aus einem konkreten Originalbild (Abb. 10.11) in der gleichen Darstellungsweise.

11.3 Repräsentation von Bildregionen

11.3.1 Matrix-Repräsentation

Eine natürliche Darstellungsform für Bilder ist eine Matrix bzw. ein zweidimensionales Array, in dem jedes Element die Intensität oder die Farbe der entsprechenden Bildposition enthält. Diese Repräsentation kann in den meisten Programmiersprachen einfach und elegant abgebildet werden und ermöglicht eine natürliche Form der Verarbeitung im Bildraster. Ein möglicher Nachteil ist, dass diese Darstellung die Struktur des Bilds nicht berücksichtigt. Es macht keinen Unterschied, ob das Bild nur ein paar Linien oder eine komplexe Szene darstellt – die erforderliche Speichermenge ist konstant und hängt nur von der Dimension des Bilds ab.

Binäre Bildregionen können mit einer logischen Maske dargestellt werden, die innerhalb der Region den Wert *true* und außerhalb den Wert *false* enthält (Abb. 11.12). Da ein logischer Wert mit nur einem Bit dargestellt werden kann, bezeichnet man eine solche Matrix häufig als „bitmap“.[5]



(a) (b) (c)

Abbildung 11.12
Verwendung einer logischen Bildmaske zu Spezifikation einer Bildregion. Originalbild (a), Bildmaske (b), maskiertes Bild (c).

11.3.2 Lauflängenkodierung

Bei der Lauflängenkodierung (*run length encoding*, RLE) werden aufeinander folgende Vordergrundpixel zu Blöcken zusammengefasst. Ein Block oder „run“ ist eine möglichst lange Folge von gleichartigen Pixeln innerhalb einer Bildzeile oder -spalte. Runs können in kompakter Form mit nur drei ganzzahligen Werten

$$Run_i = \langle row_i, column_i, length_i \rangle$$

dargestellt werden (Abb. 11.13). Werden die Runs innerhalb einer Zeile zusammengefasst, so ist natürlich die Zeilennummer redundant und kann

[5] In Java werden allerdings für Variablen vom Typ `boolean` immer 8 Bits verwendet und ein „kleinerer“ Datentyp ist nicht verfügbar. Echte „bitmaps“ können daher in Java nicht direkt realisiert werden.

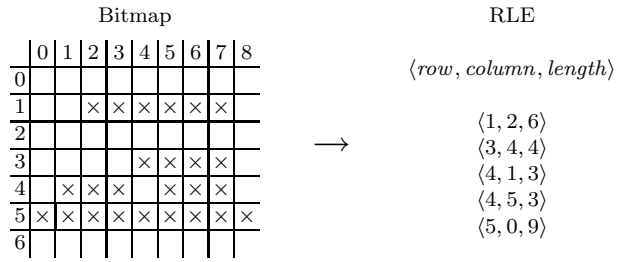


Abbildung 11.13
Lauflängenkodierung in Zeilenrichtung. Ein zusammengehöriger Pixelblock wird durch seinen Startpunkt (1, 2) und seine Länge (6) repräsentiert.

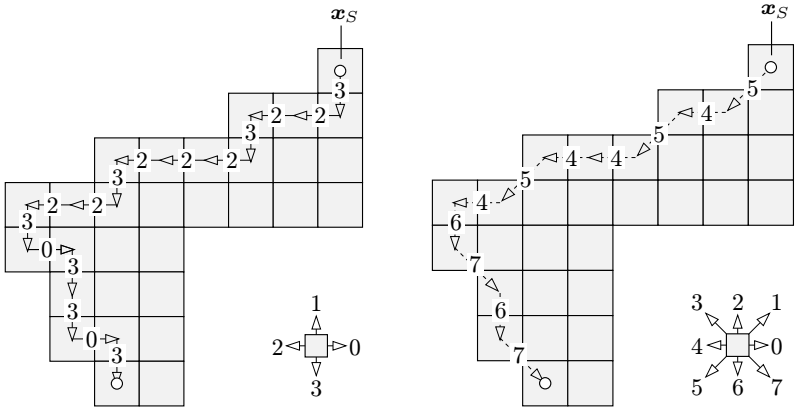
entfallen. In manchen Anwendungen kann es sinnvoll sein, die abschließende Spaltennummer anstatt der Länge des Blocks zu speichern.

Die RLE-Darstellung ist schnell und einfach zu berechnen. Sie ist auch als simple (verlustfreie) Kompressionsmethode seit langem in Gebrauch und wird auch heute noch verwendet, beispielsweise im TIFF-, GIF- und JPEG-Format sowie bei der Faxkodierung. Aus RLE-kodierten Bildern können auch statistische Eigenschaften, wie beispielsweise Momente (siehe Abschn. 11.4.3), auf direktem Weg berechnet werden.

11.3.3 *Chain Codes*

Regionen können nicht nur durch ihre innere Fläche, sondern auch durch ihre Konturen dargestellt werden. Eine klassische Form dieser Darstellung sind so genannte „Chain Codes“ oder „Freeman Codes“ [27]. Dabei wird die Kontur, ausgehend von einem Startpunkt $\boldsymbol{x}_S$, als Folge von Positionsänderungen im diskreten Bildraster repräsentiert (Abb. 11.14).

Für eine geschlossene Kontur, gegeben durch die Punktfolge $B_\mathcal{R} = (\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots \boldsymbol{x}_{M-1})$, mit $\boldsymbol{x}_i = (u_i, v_i)$, erzeugen wir die Elemente der zugehörigen Chain-Code-Folge $C_\mathcal{R} = (c_0, c_1, \ldots c_{M-1})$ mit



4-Chain Code
3223222322303303...111
length = 28

8-Chain Code
54544546767...222
length = $18 + 5\sqrt{2} \approx 25$

Abbildung 11.14
Chain Codes mit 4er- und 8er-Nachbarschaft. Zur Berechnung des Chain Codes wird die Kontur von einem Startpunkt $\boldsymbol{x}_S$ aus durchlaufen. Die relative Position zwischen benachbarten Konturpunkten bestimmt den Richtungscode innerhalb einer 4er-Nachbarschaft (links) oder einer 8er-Nachbarschaft (rechts). Die Länge des resultierenden Pfads, berechnet aus der Summe der Einzelsegmente, ergibt eine Schätzung für die tatsächliche Konturlänge.

$$c_i = \mathit{Code}(\Delta u_i, \Delta v_i)\,, \quad \text{wobei} \tag{11.1}$$
$$(\Delta u_i, \Delta v_i) = \begin{cases} (u_{i+1}-u_i, v_{i+1}-v_i) & \text{für } 0 \le i < M-1 \\ (u_0-u_i, v_0-v_i) & \text{für } i = M-1 \end{cases}$$

und $\mathit{Code}(\Delta u_i, \Delta v_i)$ aus folgender Tabelle[6] ermittelt wird:

Δu_i	1	1	0	−1	−1	−1	0	1
Δv_i	0	1	1	1	0	−1	−1	−1
$\mathit{Code}(\Delta u_i, \Delta v_i)$	0	1	2	3	4	5	6	7

Chain Codes sind kompakt, da nur die absoluten Koordinaten für den Startpunkt und nicht für jeden Konturpunkt gespeichert werden. Zudem können die 8 möglichen relativen Richtungen zwischen benachbarten Konturpunkten mit kleineren Zahlentypen (3 Bits) kodiert werden.

Differentieller Chain Code

Ein Vergleich von zwei mit Chain Codes dargestellten Regionen ist allerdings auf direktem Weg nicht möglich. Zum einen ist die Beschreibung vom gewählten Startpunkt $\boldsymbol{x}_S$ abhängig, zum anderen führt die Drehung der Region um 90° zu einem völlig anderen Chain Code. Eine geringfügige Verbesserung bringt der *differentielle* Chain Code, bei dem nicht die Differenzen der Positionen aufeinander folgender Konturpunkte, sondern die Änderungen der Richtung entlang der diskreten Kontur kodiert werden. Aus einem *absoluten* Chain-Code $C_\mathcal{R} = (c_0, c_1, \ldots c_{M-1})$ werden die Elemente des *differentiellen* Chain Codes $C'_\mathcal{R} = (c'_0, c'_1, \ldots c'_{M-1})$ in der Form

$$c'_i = \begin{cases} (c_{i+1} - c_i) \bmod 8 & \text{für } 0 \le i < M-1 \\ (c_0 - c_i) \bmod 8 & \text{für } i = M-1 \end{cases} \tag{11.2}$$

berechnet,[7] wiederum unter Annahme der 8er-Nachbarschaft. Das Element c'_i beschreibt also die Richtungsänderung (Krümmung) der Kontur zwischen den aufeinander folgenden Segmenten c_i und c_{i+1} des ursprünglichen Chain Codes $C_\mathcal{R}$. Für die Kontur in Abb. 11.14 (b) wäre das Ergebnis

$$C_\mathcal{R} = (5, 4, 5, 4, 4, 5, 4, 6, 7, 6, 7, \ldots 2, 2, 2)$$
$$C'_\mathcal{R} = (7, 1, 7, 0, 1, 7, 2, 1, 7, 1, 1, \ldots 0, 0, 3)$$

Die ursprüngliche Kontur kann natürlich bei Kenntnis des Startpunkts $\boldsymbol{x}_S$ und der Anfangsrichtung c'_0 auch aus einem differentiellen Chain Code wieder vollständig rekonstruiert werden.

[6] Unter Verwendung der 8er-Nachbarschaft.
[7] Zur Implementierung des mod-Operators in Java siehe Anhang B.1.2.

Shape Numbers

Der differentielle Chain Code bleibt zwar bei einer Drehung der Region um 90° unverändert, ist aber weiterhin vom gewählten Startpunkt abhängig. Möchte man zwei durch ihre differentiellen Chain Codes C'_1, C'_2 gegebenen Konturen von gleicher Länge M auf ihre Ähnlichkeit untersuchen, so muss zunächst ein gemeinsamer Startpunkt festgelegt werden. Eine häufig angeführte Methode [4,30] besteht darin, die Codefolge C' als Ziffern einer Zahl (zur Basis $B = 8$ bzw. $B = 4$ bei einer 4er-Nachbarschaft) zu interpretieren, d. h. mit dem Wert

$$\mathit{Value}(C') = c'_0 \cdot B^0 + c'_1 \cdot B^1 + \ldots + c'_{M-1} \cdot B^{M-1} = \sum_{i=0}^{M-1} c_i \cdot B^i \ . \tag{11.3}$$

Dann wird die Folge C' soweit zyklisch um k Positionen verschoben, bis sich ein maximaler Wert

$$\mathit{Value}(C' \rightarrow k) = \max! \qquad \text{für } \ 0 \le k < M$$

ergibt. $C' \rightarrow k$ bezeichnet dabei die zyklisch um k Positionen nach rechts verschobene Folge C', z. B.

$$C' = (0, 1, 3, 2, \ldots 0, 3, 7, 4) \ , \qquad C' \rightarrow 2 = (7, 4, 0, 1, \ldots 0, 3) \ .$$

Die resultierende Folge („Shape Number“) ist dann bezüglich des Startpunkts „normalisiert“ und kann ohne weitere Verschiebung elementweise mit anderen normalisierten Folgen verglichen werden. Allerdings würde die Funktion $\mathit{Value}(C')$ in Gl. 11.3 viel zu große Werte erzeugen, um sie tatsächlich berechnen zu können. Einfacher ist es, die Relation

$$\mathit{Value}(C'_1) > \mathit{Value}(C'_2)$$

auf Basis der *lexikographischen Ordnung* zwischen den (Zeichen-)Folgen C'_1 und C'_2 zu ermitteln, ohne deren arithmetische Werte wirklich zu berechnen.

Der Vergleich auf Basis der Chain Codes ist jedoch generell keine besonders zuverlässige Methode zur Messung der Ähnlichkeit von Regionen, allein deshalb, weil etwa Drehungen um beliebige Winkel ($\neq 90°$) zu großen Differenzen im Code führen können. Darüber hinaus sind natürlich Größenveränderungen (Skalierungen) oder andere Verzerrungen mit diesen Methoden überhaupt nicht handhabbar. Für diese Zwecke finden sich wesentlich bessere Werkzeuge im nachfolgenden Abschnitt 11.4.

Fourierdeskriptoren

Ein eleganter Ansatz zur Beschreibung von Konturen sind so genannte „Fourierdeskriptoren“, bei denen die zweidimensionale Kontur $B_{\mathcal{R}} =$

$(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots \boldsymbol{x}_{M-1})$ mit $\boldsymbol{x}_i = (u_i, v_i)$ als Folge von komplexen Werten $(z_0, z_1 \ldots z_{M-1})$ mit

$$z_i = (u_i + \mathrm{i} \cdot v_i) \in \mathbb{C} \tag{11.4}$$

interpretiert wird. Aus dieser Folge lässt sich (durch geeignete Interpolation) eine diskrete, eindimensionale, periodische Funktion $f(s) \in \mathbb{C}$ mit konstanten Abtastintervallen über die Konturlänge s ableiten. Die Koeffizienten des (eindimensionalen) *Fourierspektrums* (siehe Abschn. 13.3) der Funktion $f(s)$ bilden eine Formbeschreibung der Kontur im Frequenzraum, wobei die niedrigen Spektralkoeffizienten eine grobe Formbeschreibung liefern. Details zu diesem klassischen Verfahren finden sich beispielsweise in [30, 33, 48, 49, 80].

11.4 Eigenschaften binärer Bildregionen

Angenommen man müsste den Inhalt eines Digitalbilds einer anderen Person am Telefon beschreiben. Eine Möglichkeit bestünde darin, die einzelnen Pixelwerte in einer bestimmten Ordnung aufzulisten und durchzugeben. Ein weniger mühsamer Ansatz wäre, das Bild auf Basis von Eigenschaften auf einer höheren Ebene zu beschreiben, etwa als „ein rotes Rechteck auf einem blauen Hintergrund“ oder „ein Sonnenuntergang am Strand mit zwei im Sand spielenden Hunden“ usw. Während uns so ein Vorgehen durchaus natürlich und einfach erscheint, ist die Generierung derartiger Beschreibungen für Computer ohne menschliche Hilfe derzeit (noch) nicht realisierbar. Für den Computer einfacher ist die Berechnung mathematischer Eigenschaften von Bildern oder einzelner Bildteile, die zumindest eine eingeschränkte Form von Klassifikation ermöglichen. Dies wird als „Mustererkennung“ (*Pattern Recognition*) bezeichnet und bildet ein eigenes wissenschaftliches Fachgebiet, das weit über die Bildverarbeitung hinausgeht [21, 62, 83].

11.4.1 Formmerkmale (*Features*)

Der Vergleich und die Klassifikation von binären Regionen ist ein häufiger Anwendungsfall, beispielsweise bei der „optischen“ Zeichenerkennung (*optical character recognition*, OCR), beim automatischen Zählen von Zellen in Blutproben oder bei der Inspektion von Fertigungsteilen auf einem Fließband. Die Analyse von binären Regionen gehört zu den einfachsten Verfahren und erweist sich in vielen Anwendungen als effizient und zuverlässig.

Als „Feature“ einer Region bezeichnet man ein bestimmtes numerisches oder qualitatives Merkmal, das aus ihren Bildpunkten (Werten und Koordinaten) berechnet wird, im einfachsten Fall etwa die Größe (Anzahl der Pixel) einer Region. Um eine Region möglichst eindeutig zu beschreiben, werden üblicherweise verschiedene Features zu einem „Feature Vector“ kombiniert. Dieser stellt gewissermaßen die „Signatur“

einer Region dar, die zur Klassifikation bzw. zur Unterscheidung gegenüber anderen Regionen dient. Features sollen einfach zu berechnen und möglichst unbeeinflusst („robust“) von nicht relevanten Veränderungen sein, insbesondere gegenüber einer räumlichen Verschiebung, Rotation oder Skalierung.

11.4.2 Geometrische Eigenschaften

Die Region $\mathcal{R}$ eines Binärbilds kann als zweidimensionale Verteilung von Vordergrundpunkten $\boldsymbol{x}_i = (u_i, v_i)$ in der diskreten Ebene $\mathbb{Z}^2$ interpretiert werden, d. h.

$$\mathcal{R} = \{\boldsymbol{x}_1, \boldsymbol{x}_2 \dots \boldsymbol{x}_N\} = \{(u_1, v_1), (u_2, v_2) \dots (u_N, v_N)\}. \tag{11.5}$$

Für die Berechnung der meisten geometrischen Eigenschaften kann eine Region aus einer beliebigen Punktmenge bestehen und muss auch (im Unterschied zur Definition in Abschn. 11.1) nicht notwendigerweise zusammenhängend sein.

Umfang

Der Umfang (*perimeter*) einer Region $\mathcal{R}$ ist bestimmt durch die Länge ihrer äußeren Kontur, wobei $\mathcal{R}$ zusammenhängend sein muss. Wie Abb. 11.14 zeigt, ist bei der Berechnung die Art der Nachbarschaftsbeziehung zu beachten. Bei Verwendung der 4er-Nachbarschaft ist die Gesamtlänge der Konturesegmente (jeweils mit der Länge 1) i. Allg. größer als die tatsächliche Strecke.

Im Fall der 8er-Nachbarschaft wird durch Gewichtung der Horizontal- und Vertikalsegmente mit 1 und der Diagonalsegmente mit $\sqrt{2}$ eine gute Annäherung erreicht. Für eine Kontur mit dem 8-Chain Code $C_{\mathcal{R}} = (c_0, c_1, \dots c_{M-1})$ berechnet sich der Umfang daher in der Form

$$\mathit{Perimeter}(\mathcal{R}) = \sum_{i=0}^{M-1} \mathit{length}(c_i)\ , \tag{11.6}$$

wobei

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

Bei dieser gängigen Form der Berechnung[8] wird allerdings die Länge des Umfangs gegenüber dem tatsächlichen Wert $U(\mathcal{R})$ systematisch überschätzt. Ein einfacher Korrekturfaktor von 0.95 erweist sich bereits bei relativ kleinen Regionen als brauchbar, d. h.

$$U(\mathcal{R}) \approx \mathit{Perimeter}_{\text{corr}}(\mathcal{R}) = 0.95 \cdot \mathit{Perimeter}(\mathcal{R}). \tag{11.7}$$

[8] Auch die im Analyze-Menü von ImageJ verfügbaren Messfunktionen verwenden diese Form der Umfangsberechnung.

Fläche

Die Fläche einer Region $\mathcal{R}$ berechnet sich einfach durch die Anzahl der enthaltenen Bildpunkte, d. h.

$$\mathit{Area}(\mathcal{R}) = N = |\mathcal{R}|. \tag{11.8}$$

Die Fläche einer zusammenhängenden Region (ohne Löcher) kann auch näherungsweise über ihre geschlossene Kontur, definiert durch M Koordinatenpunkte $(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots \boldsymbol{x}_{M-1})$, wobei $\boldsymbol{x}_i = (u_i, v_i)$, mit der Gauß'schen Flächenformel für Polygone) berechnet werden:

$$\mathit{Area}(\mathcal{R}) = \frac{1}{2} \cdot \left| \sum_{i=0}^{M-1} \Big(u_i \cdot v_{[(i+1) \bmod M]} - u_{[(i+1) \bmod M]} \cdot v_i \Big) \right| \tag{11.9}$$

Liegt die Kontur in Form eines Chain Codes $C_\mathcal{R} = (c_0, c_1, \ldots c_{M-1})$ vor, so kann die Fläche durch Expandieren von $C_\mathcal{R}$ in eine Folge von Konturpunkten, ausgehend von einem beliebigen Startpunkt (z. B. $(0, 0)$), ebenso mit Gl. 11.9 berechnet werden.

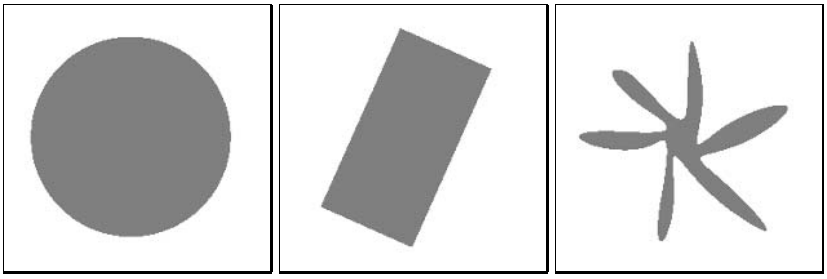
Einfache Eigenschaften wie Fläche und Umfang sind zwar (abgesehen von Quantisierungsfehlern) unbeeinflusst von Verschiebungen und Drehungen einer Region, sie verändern sich jedoch bei einer *Skalierung* der Region, wenn also beispielsweise ein Objekt aus verschiedenen Entfernungen aufgenommen wurde. Durch geschickte Kombination können jedoch neue Features konstruiert werden, die invariant gegenüber Translation, Rotation und Skalierung sind.

Kompaktheit und Rundheit

Unter „Kompaktheit" versteht man die Relation zwischen der Fläche einer Region und ihrem Umfang. Da der Umfang (*Perimeter*) U einer Region linear mit dem Vergrößerungsfaktor zunimmt, die Fläche (*Area*) A jedoch quadratisch, verwendet man das Quadrat des Umfangs in der Form A/U^2 zur Berechnung eines größenunabhängigen Merkmals. Dieses Maß ist invariant gegenüber Verschiebungen, Drehungen und Skalierungen und hat für eine kreisförmige Region mit beliebigem Durchmesser den Wert $\frac{1}{4\pi}$. Durch Normierung auf den Kreis ergibt sich daraus ein Maß für die „Rundheit" (*roundness*) oder „Kreisförmigkeit (*circularity*)

$$\mathit{Circularity}(\mathcal{R}) = 4\pi \cdot \frac{\mathit{Area}(\mathcal{R})}{\mathit{Perimeter}^2(\mathcal{R})}\,, \tag{11.10}$$

das für eine kreisförmige Region $\mathcal{R}$ den Maximalwert 1 ergibt und für alle übrigen Formen Werte im Bereich $[0, 1]$ (Abb. 11.15). Für eine absolute Schätzung der Kreisförmigkeit empfiehlt sich allerdings die Verwendung des korrigierten Umfangwerts aus Gl. 11.7, also



0.904 (1.001) | 0.607 (0.672) | 0.078 (0.086)

Abbildung 11.15
Circularity-Werte für verschiedene Regionsformen. Angegeben sind jeweils der Wert $Circularity(\mathcal{R})$ und in Klammern der korrigierte Wert $Circularity_{\text{corr}}(\mathcal{R})$ nach Gl. 11.10 bzw. 11.11.

$$Circularity_{\text{corr}}(\mathcal{R}) = 4\pi \cdot \frac{Area(\mathcal{R})}{Perimeter^2_{\text{corr}}(\mathcal{R})} \,. \tag{11.11}$$

In Abb. 11.15 sind die Werte für die Kreisförmigkeit nach Gl. 11.10 bzw. 11.11 für verschiedene Formen von Regionen dargestellt.

Bounding Box

Die Bounding Box einer Region $\mathcal{R}$ bezeichnet das minimale, achsenparallele Rechteck, das alle Punkte aus $\mathcal{R}$ einschließt:

$$BoundingBox(\mathcal{R}) = (u_{\min}, u_{\max}, v_{\min}, v_{\max}), \tag{11.12}$$

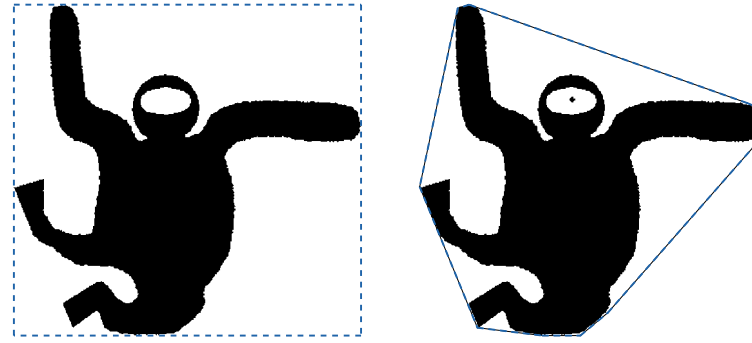
wobei $u_{\min}, u_{\max}$ und $v_{\min}, v_{\max}$ die minimalen und maximalen Koordinatenwerte aller Punkte $(u_i, v_i) \in \mathcal{R}$ in x- bzw. y-Richtung sind (Abb. 11.16 (a)).

Konvexe Hülle

Die konvexe Hülle (*convex hull*) ist das kleinste Polygon, das alle Punkte einer Region umfasst. Eine einfache Analogie ist die eines Nagelbretts, in dem für alle Punkte einer Region ein Nagel an der entsprechenden Position eingeschlagen ist. Spannt man nun ein elastisches Band rund um alle Nägel, dann bildet dieses die konvexe Hülle (Abb. 11.16 (b)). Sie kann z. B. mit dem *QuickHull*-Algorithmus [5] für N Konturpunkte mit einem Zeitaufwand von $\mathcal{O}(NH)$ berechnet werden, wobei H die Anzahl der resultierenden Polygonpunkte ist.[9]

Nützlich ist die konvexe Hülle beispielsweise zur Bestimmung der Konvexität oder der *Dichte* einer Region. Die *Konvexität* ist definiert als das Verhältnis zwischen der Länge der konvexen Hülle und dem Umfang der ursprünglichen Region. Unter *Dichte* versteht man hingegen das Verhältnis zwischen der Fläche der Region selbst und der Fläche der konvexen Hülle. Der *Durchmesser* wiederum ist die maximale Strecke zwischen zwei Knoten auf der konvexen Hülle.

[9] Zur Notation $\mathcal{O}()$ s. Anhang 1.3.



(a) (b)

Abbildung 11.16
Beispiel für *Bounding Box* (a) und konvexe Hülle (b) einer binären Bildregion.

11.4.3 Statistische Formeigenschaften

Bei der Berechnung von statistischen Formmerkmalen betrachten wir die Region $\mathcal{R}$ als Verteilung von Koordinatenpunkten im zweidimensionalen Raum. Statistische Merkmale können insbesondere auch für Punktverteilungen berechnet werden, die keine zusammengehörige Region bilden, und sind daher ohne vorherige Segmentierung einsetzbar. Ein wichtiges Konzept bilden in diesem Zusammenhang die so genannten *zentralen Momente* der Verteilung, die charakteristische Eigenschaften in Bezug auf deren Mittelpunkt bzw. *Schwerpunkt* ausdrücken.

Schwerpunkt

Den Schwerpunkt einer zusammenhängenden Region kann man sich auf einfache Weise so vorstellen, dass man die Region auf ein Stück Karton oder Blech zeichnet, ausschneidet und dann versucht, diese Form waagerecht auf einer Spitze zu balancieren. Der Punkt, an dem man die Region aufsetzen muss, damit dieser Balanceakt gelingt, ist der *Schwerpunkt* der Region.[10]

Der Schwerpunkt $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y})$ einer binären (nicht notwendigerweise zusammenhängenden) Region berechnet sich als arithmetischer Mittelwert der Koordinaten in x- und y-Richtung, d. h.

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} u \qquad \text{und} \qquad \bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} v \,. \tag{11.13}$$

Momente

Die Formulierung für den Schwerpunkt einer Region in Gl. 11.13 ist nur ein spezieller Fall eines allgemeineren Konzepts aus der Statistik, der so genannten „Momente“. Insbesondere beschreibt der Ausdruck

[10] Vorausgesetzt der Schwerpunkt liegt nicht innerhalb eines Lochs in der Region liegt, was durchaus möglich ist.

$$m_{pq} = \sum_{(u,v)\in\mathcal{R}} I(u,v) \cdot u^p v^q \tag{11.14}$$

das (gewöhnliche) Moment der Ordnung p, q für eine diskrete (Bild-)Funktion $I(u,v) \in \mathbb{R}$, also beispielsweise für ein Grauwertbild. Alle nachfolgenden Definitionen sind daher – unter entsprechender Einbeziehung der Bildfunktion $I(u,v)$ – grundsätzlich auch für Regionen in Grauwertbildern anwendbar. Für zusammenhängende, binäre Regionen können Momente auch direkt aus den Koordinaten der Konturpunkte berechnet werden [75, S. 148].

Für den speziellen Fall eines Binärbilds $I(u,v) \in \{0,1\}$ sind nur die Vordergrundpixel mit $I(u,v) = 1$ in der Region $\mathcal{R}$ enthalten, wodurch sich Gl. 11.14 reduziert auf

$$m_{pq} = \sum_{(u,v)\in\mathcal{R}} u^p v^q \,. \tag{11.15}$$

So kann etwa die **Fläche** einer binären Region als Moment nullter Ordnung in der Form

$$\mathit{Area}(\mathcal{R}) = |\mathcal{R}| = \sum_{(u,v)\in\mathcal{R}} 1 = \sum_{(u,v)\in\mathcal{R}} u^0 v^0 = m_{00}(\mathcal{R}) \tag{11.16}$$

ausgedrückt werden bzw. der **Schwerpunkt** $\bar{\boldsymbol{x}}$ (Gl. 11.13) als

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} u^1 v^0 = \frac{m_{10}(\mathcal{R})}{m_{00}(\mathcal{R})} \tag{11.17}$$

$$\bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} u^0 v^1 = \frac{m_{01}(\mathcal{R})}{m_{00}(\mathcal{R})} \tag{11.18}$$

Diese Momente repräsentieren also konkrete physische Eigenschaften einer Region. Insbesondere ist die Fläche m_{00} in der Praxis eine wichtige Basis zur Charakterisierung von Regionen und der Schwerpunkt $(\bar{x}, \bar{y})$ erlaubt die zuverlässige und (auf Bruchteile eines Pixelabstands) genaue Bestimmung der Position einer Region.

Zentrale Momente

Um weitere Merkmale von Regionen unabhängig von ihrer Lage, also invariant gegenüber Verschiebungen, zu berechnen, wird der in jeder Lage eindeutig zu bestimmende Schwerpunkt als Referenz verwendet. Anders ausgedrückt, man verschiebt den Ursprung des Koordinatensystems an den Schwerpunkt $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y})$ der Region und erhält dadurch die so genannten *zentralen* Momente der Ordnung p, q:

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} I(u,v) \cdot (u-\bar{x})^p \cdot (v-\bar{y})^q \tag{11.19}$$

Für Binärbilder (mit $I(u,v) = 1$ innerhalb der Region $\mathcal{R}$) reduziert sich Gl. 11.19 auf

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} (u - \bar{x})^p \cdot (v - \bar{y})^q \,. \tag{11.20}$$

Normalisierte zentrale Momente

Die Werte der zentralen Momente sind naturgemäß von der absoluten Größe der Region abhängig, da diese sich in den Distanzen aller Punkte vom Schwerpunkt direkt manifestiert. So multiplizieren sich bei einer gleichförmigen Vergrößerung einer Form um den Faktor $s \in \mathbb{R}$ die zentralen Momente mit dem Faktor

$$s^{(p+q+2)} \tag{11.21}$$

und man erhält größeninvariante (normalisierte) Momente durch Normierung mit dem Kehrwert der entsprechend potenzierten Fläche $\mu_{00} = m_{00}$ in der Form

$$\bar{\mu}_{pq}(\mathcal{R}) = \mu_{pq} \cdot \Big(\frac{1}{\mu_{00}(\mathcal{R})}\Big)^{(p+q+2)/2} \,, \tag{11.22}$$

für $(p+q) \geq 2$ [48, S. 529].

Prog. 11.3 zeigt eine direkte (*brute force*) Umsetzung der Berechnung von gewöhnlichen, zentralen und normalisierten Momenten in Java für binäre Bilder (`BACKGROUND = 0`). Dies ist nur zur Verdeutlichung gedacht und es sind natürlich weitaus effizientere Implementierungen möglich (z. B. in [50]).

11.4.4 Momentenbasierte geometrische Merkmale

Während die normalisierten Momente auch direkt zur Charakterisierung von Regionen verwendet werden können, sind einige interessante und geometrisch unmittelbar relevante Merkmale auf elegante Weise aus den Momenten ableitbar.

Orientierung

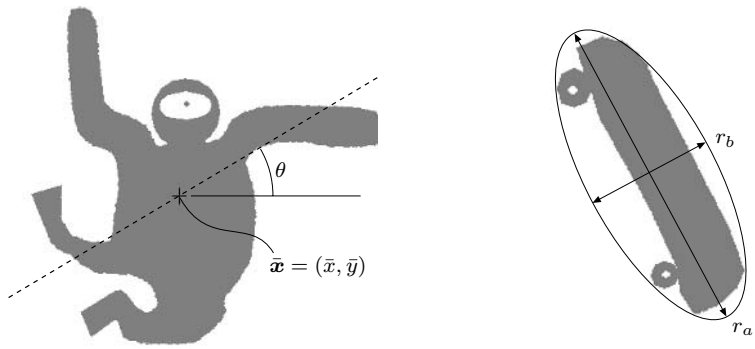
Die Orientierung bezeichnet die Richtung der Hauptachse, also der Achse, die durch den Schwerpunkt und entlang der größten Ausdehnung einer Region verläuft (Abb. 11.17 (a)). Dreht man die durch die Region beschriebene Fläche um ihre Hauptachse, so weist diese das geringste Trägheitsmoment aller möglichen Drehachsen auf. Wenn man etwa einen Bleistift zwischen beiden Händen hält und um seine Hauptachse (entlang der Bleistiftmine) dreht, dann treten wesentlich geringere Trägheitskräfte auf als beispielsweise bei einer propellerartigen Drehung quer zur Hauptachse (Abb. 11.18). Sofern die Region überhaupt eine

Programm 11.3
Beispiel für die direkte Berechnung von Momenten in Java. Die Methoden `moment()`, `centralMoment()` und `normalCentralMoment()` berechnen für ein Binärbild die Momente m_{pq}, μ_{pq} bzw. $\bar{\mu}_{pq}$ (Gl. 11.15, 11.20, 11.22).

```
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 }
```



(a) (b)

Abbildung 11.17
Orientierung und Exzentrizität. Die Hauptachse einer Region läuft durch ihren Schwerpunkt $\bar{\boldsymbol{x}}$ unter dem Winkel θ (a). Die Exzentrizität (b) ist ein Maß für das Seitenverhältnis (r_a/r_b) der kleinsten umhüllenden Ellipse, deren längere Achse parallel zur Hauptachse der Region liegt.

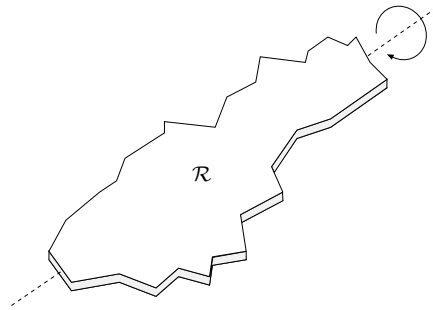


Abbildung 11.18
Hauptachse einer Region. Die Drehung einer länglichen Region $\mathcal{R}$ (interpretiert als physischer Körper) um ihre Hauptachse verursacht die geringsten Trägheitskräfte aller möglichen Achsen.

Orientierung aufweist ($\mu_{20}(\mathcal{R}) \neq \mu_{02}(\mathcal{R})$), ergibt sich die Richtung θ der Hauptachse aus den zentralen Momenten μ_{pq} als[11]

$$\theta(\mathcal{R}) = \frac{1}{2}\tan^{-1}\left(\frac{2\cdot\mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R})-\mu_{02}(\mathcal{R})}\right) . \qquad (11.23)$$

Exzentrizität

Ähnlich wie die *Richtung* der Hauptachse lässt sich auch die *Länglichkeit* der Region über die Momente bestimmen. Das Merkmal der Exzentrizität (*eccentricity* oder *elongation*) kann man sich am einfachsten so vorstellen, dass man eine Region so lange dreht, bis sich eine Bounding Box oder umhüllende Ellipse mit maximalem Seitenverhältnis ergibt (Abb. 11.17 (b)).[12] Aus dem Verhältnis zwischen der Breite und der Länge der resultierenden Bounding Box bzw. Ellipse können unterschiedliche Maße für die Exzentrizität der Region abgeleitet werden [4, 49] (s. auch Aufg.

[11] Siehe Anhang B.1.6 bezüglich der Winkelberechnung in Java mit `Math.atan2()`.

[12] Dieser Vorgang wäre in der Praxis natürlich allein wegen der vielfachen Bildrotation relativ rechenaufwendig.

11.11). In [48, S. 531] etwa wird auf Basis der zentralen Momente μ_{pq} ein Exzentrizitätsmaß in der Form

$$Eccentricity(\mathcal{R}) = \frac{\big[\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})\big]^2 + 4 \cdot \big[\mu_{11}(\mathcal{R})\big]^2}{\big[\mu_{20}(\mathcal{R}) + \mu_{02}(\mathcal{R})\big]^2} \tag{11.24}$$

definiert. Dieses Maß ergibt Werte im Bereich $[0, 1]$. Ein rundes Objekt weist demnach eine Exzentrizität von 0 auf, ein extrem langgezogenes Objekt den Wert 1, also entgegengesetzt zur „Rundheit“ in Gl. 11.10. Die Berechnung der Exzentrizität ist allerdings in diesem Fall ohne explizite Kenntnis des Umfangs und der Fläche möglich und somit auch für nicht zusammenhängende Regionen anwendbar.

Invariante Momente

Normalisierte zentrale Momente sind zwar unbeeinflusst durch eine Translation oder gleichförmige Skalierung einer Region, verändern sich jedoch im Allgemeinen bei einer *Rotation* des Bilds. Ein klassisches Beispiel zur Lösung dieses Problems durch geschickte Kombination einfacher Merkmale sind die nachfolgenden, als „Hu's Momente“ [39] bekannten Größen:[13]

$$\begin{aligned}
H_1 &= \bar{\mu}_{20} + \bar{\mu}_{02} \qquad (11.25)\\
H_2 &= (\bar{\mu}_{20} - \bar{\mu}_{02})^2 + 4\,\bar{\mu}_{11}^2\\
H_3 &= (\bar{\mu}_{30} - 3\,\bar{\mu}_{12})^2 + (3\,\bar{\mu}_{21} - \bar{\mu}_{03})^2\\
H_4 &= (\bar{\mu}_{30} + \bar{\mu}_{12})^2 + (\bar{\mu}_{21} + \bar{\mu}_{03})^2\\
H_5 &= (\bar{\mu}_{30} - 3\,\bar{\mu}_{12}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot \big[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3(\bar{\mu}_{21} + \bar{\mu}_{03})^2\big] +\\
&\quad (3\,\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot \big[3\,(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2\big]\\
H_6 &= (\bar{\mu}_{20} - \bar{\mu}_{02}) \cdot \big[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2\big] +\\
&\quad 4\,\bar{\mu}_{11} \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03})\\
H_7 &= (3\,\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot \big[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3\,(\bar{\mu}_{21} + \bar{\mu}_{03})^2\big] +\\
&\quad (3\,\bar{\mu}_{12} - \bar{\mu}_{30}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot \big[3\,(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2\big]
\end{aligned}$$

In der Praxis wird allerdings meist der Logarithmus der Ergebnisse (also $\log(H_k)$) verwendet, um den ansonsten sehr großen Wertebereich zu reduzieren. Diese Features werden auch als „*moment invariants*“ bezeichnet, denn sie sind weitgehend invariant unter Translation, Skalierung sowie Rotation. Sie sind auch auf Ausschnitte von Grauwertbildern anwendbar, Beispiele dafür finden sich etwa in [30, S. 517].

[13] Das Argument für die Region ($\mathcal{R}$) wird in Gl. 11.25 zur besseren Lesbarkeit weggelassen, die erste Zeile würde also vollständig lauten: $H_1(\mathcal{R}) = \bar{\mu}_{20}(\mathcal{R}) + \bar{\mu}_{02}(\mathcal{R})$, usw.

Programm 11.4
Berechnung von horizontaler und vertikaler Projektion. Die `run()`-Methode für ein ImageJ-Plugin (`ip` ist vom Typ `ByteProcessor` oder `ShortProcessor`) berechnet in einem Bilddurchlauf beide Projektionen als eindimensionale Arrays (`horProj`, `verProj`) mit Elementen vom Typ `long`.

```
1  public void run(ImageProcessor ip) {
2    int M = ip.getWidth();
3    int N = ip.getHeight();
4    long[] horProj = new long[N];
5    long[] verProj = new long[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 }
```

11.4.5 Projektionen

Projektionen von Bildern sind eindimensionale Abbildungen der Bilddaten, üblicherweise parallel zu den Koordinatenachsen. In diesem Fall ist die horizontale bzw. vertikale Projektion für ein Bild $I(u, v)$, mit $0 \le u < M$, $0 \le v < N$, definiert als

$$P_{\text{hor}}(v_0) = \sum_{u=0}^{M-1} I(u, v_0) \qquad \text{für } 0 < v_0 < N, \tag{11.26}$$

$$P_{\text{ver}}(u_0) = \sum_{v=0}^{N-1} I(u_0, v) \qquad \text{für } 0 < u_0 < M. \tag{11.27}$$

Die *horizontale* Projektion $P_{\text{hor}}(v_0)$ (Gl. 11.26) bildet also die Summe der Pixelwerte der Bild*zeile* v_0 und hat die Länge N, die der Höhe des Bilds entspricht. Umgekehrt ist die *vertikale* Projektion P_{ver} (der Länge M) die Summe aller Bildwerte in der *Spalte* u_0 (Gl. 11.27). Im Fall eines Binärbilds mit $I(u, v) \in 0, 1$ enthält die Projektion die Anzahl der Vordergrundpixel in der zugehörigen Bildzeile bzw. -spalte.

Prog. 11.4 zeigt eine einfache Implementierung der Projektionsberechnung als `run()`-Methode eines ImageJ-Plugin, wobei beide Projektionen in einem Bilddurchlauf berechnet werden. Für die Projektionen werden Arrays vom Typ `long` (64-Bit-Integer) verwendet, um auch bei großen Bildern einen arithmetischen Überlauf zu vermeiden.

Projektionen in Richtung der Koordinatenachsen sind beispielsweise zur schnellen Analyse von strukturierten Bildern nützlich, wie etwa zur Isolierung der einzelnen Zeilen in Textdokumenten oder auch zur Trennung der Zeichen innerhalb einer Textzeile (Abb. 11.19). Grundsätzlich sind aber Projektionen auf Geraden mit beliebiger Orientierung möglich, beispielsweise in Richtung der Hauptachse einer gerichteten Bildregion (Gl. 11.23). Nimmt man den Schwerpunkt der Region (Gl. 11.13) als Re-

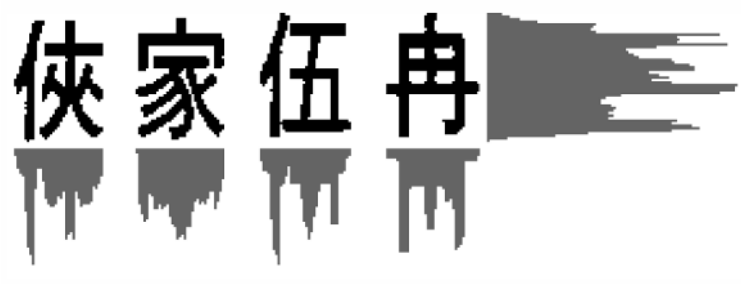


Abbildung 11.19
Beispiel für die horizontale und vertikale Projektion eines Binärbilds.

ferenz entlang der Hauptachse, so erhält man eine weitere, rotationsinvariante Beschreibung der Region in Form des Projektionsvektors.

11.4.6 Topologische Merkmale

Topologische Merkmale beschreiben nicht explizit die Form einer Region, sondern strukturelle Eigenschaften, die auch unter stärkeren Bildverformungen unverändert bleiben. Dazu gehört auch die Eigenschaft der Konvexität einer Region, die sich durch Berechnung ihrer konvexen Hülle (Abschn. 11.4.2) bestimmen lässt.

Ein einfaches und robustes topologisches Merkmal ist die *Anzahl der Löcher* $N_L(\mathcal{R})$, die sich aus der Berechnung der inneren Konturen einer Region ergibt, wie in Abschn. 11.2.2 beschrieben. Umgekehrt kann eine nicht zusammenhängende Region, wie beispielsweise der Buchstabe „**i**“, aus mehreren Komponenten bestehen, deren Anzahl ebenfalls als Merkmal verwendet werden kann.

Ein davon abgeleitetes Merkmal ist die so genannte *Euler-Zahl* N_E, das ist die Anzahl der zusammenhängenden Regionen N_R abzüglich der Anzahl ihrer Löcher N_L, d. h.

$$N_E(\mathcal{R}) = N_R(\mathcal{R}) - N_L(\mathcal{R}). \tag{11.28}$$

Bei nur *einer* zusammenhängenden Region ist dies einfach $1 - N_L$. So gilt für die Ziffer „**8**“ beispielsweise $N_L = 1 - 2 = -1$ oder $N_L = 1 - 1 = 0$ für den Buchstaben „**D**“.

Topologische Merkmale werden oft in Kombination mit numerischen Features zur Klassifikation verwendet, etwa für die Zeichenerkennung (*optical character recognition*, OCR) [13].

11.5 Aufgaben

Aufg. 11.1. Simulieren Sie manuell den Ablauf des *Flood-fill*-Verfahrens in Prog. 11.1 (*depth-first* und *breadth-first*) anhand einer Region des folgenden Bilds, beginnend bei der Startkoordinate $(5, 1)$:

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	1	1	0	1	0
0	1	1	1	1	1	1	0	0	1	0	0	1	0
0	0	0	0	1	0	1	0	0	0	0	0	1	0
0	1	1	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	1	1	1	1	1	1	1	1	1	0
0	1	1	0	0	0	1	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

0 Hintergrund

1 Vordergrund

Aufg. 11.2. Bei der Implementierung des *Flood-fill*-Verfahrens (Prog. 11.1) werden bei jedem bearbeiteten Pixel alle seine Nachbarn im *Stack* bzw. in der *Queue* vorgemerkt, unabhängig davon, ob sie noch innerhalb des Bilds liegen und Vordergrundpixel sind. Man kann die Anzahl der im *Stack* bzw. in der *Queue* zu speichernden Knoten reduzieren, indem man jene Nachbarpixel ignoriert, die diese Bedingungen nicht erfüllen. Modifizieren Sie die *Depth-first*- und *Breadth-first*-Variante in Prog. 11.1 entsprechend und vergleichen Sie die resultierenden Laufzeiten.

Aufg. 11.3. Implementieren Sie ein ImageJ-Plugin, das auf ein Grauwertbild eine Lauflängenkodierung (Abschn. 11.3.2) anwendet, das Ergebnis in einer Datei ablegt und ein zweites Plugin, das aus dieser Datei das Bild wieder rekonstruiert.

Aufg. 11.4. Berechnen Sie den erforderlichen Speicherbedarf zur Darstellung einer Kontur mit 1000 Punkten auf folgende Arten: (a) als Folge von Koordinatenpunkten, die als Paare von `int`-Werten dargestellt sind; (b) als 8-Chain Code mit Java-`byte`-Elementen; (c) als 8-Chain Code mit nur 3 Bits pro Element.

Aufg. 11.5. Implementieren Sie eine Java-Klasse zur Beschreibung von binären Bildregionen mit Chain Codes. Entscheiden Sie selbst, ob Sie einen absoluten oder differentiellen Chain Code verwenden. Die Implementierung soll in der Lage sein, geschlossene Konturen als Chain Codes zu kodieren und auch wieder zu rekonstruieren.

Aufg. 11.6. Durch Berechnung der konvexen Hülle kann auch der maximale Durchmesser (max. Abstand zwischen zwei beliebigen Punkten) einer Region auf einfache Weise berechnet werden. Überlegen Sie sich ein alternatives Verfahren, das diese Aufgabe ohne Verwendung der komplexen Hülle löst. Ermitteln Sie den Zeitaufwand Ihres Algorithmus in Abhängigkeit zur Anzahl der Punkte in der Region.

Aufg. 11.7. Implementieren Sie den Vergleich von Konturen auf Basis von „Shape Numbers“ (Gl. 11.3). Entwickeln Sie dafür eine Metrik, welche die Distanz zwischen zwei normalisierten Chain Codes misst. Stellen Sie fest, ob und unter welchen Bedingungen das Verfahren zuverlässig arbeitet.

Aufg. 11.8. Entwerfen Sie einen Algorithmus, der aus einer als 8-Chain Code gegebenen Kontur auf der Basis von Gl. 11.9 die Fläche der zugehörigen Region berechnet. Welche Abweichungen sind gegenüber der tatsächlichen Fläche der Region (Anzahl der Pixel) zu erwarten?

Aufg. 11.9. Skizzieren Sie Beispiele von binären Regionen, bei denen der Schwerpunkt selbst nicht in der Bildregion liegt.

Aufg. 11.10. Implementieren Sie die Momenten-Features von Hu (Gl. 11.25) und überprüfen Sie deren Eigenschaften unter Skalierung und Rotation anhand von Binär- und Grauwertbildern.

Aufg. 11.11. Für die Exzentrizität einer Region (Gl. 11.24) gibt es alternative Formulierungen, zum Beispiel [49, S. 394]

$$Eccentricity_2(\mathcal{R}) = \frac{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}}{m_{00}} \quad \text{oder} \tag{11.29}$$

$$Eccentricity_3(\mathcal{R}) = \frac{\mu_{20} + \mu_{02} + \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}}{\mu_{20} + \mu_{02} - \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}} \,. \tag{11.30}$$

Realisieren Sie alle drei Varianten (einschließlich Gl. 11.24) und vergleichen Sie die Ergebnisse anhand geeigneter Regionsformen.

Aufg. 11.12. Die Java-Methode in Prog. 11.4 berechnet die Projektionen eines Bilds in horizontaler und vertikaler Richtung. Bei der Verarbeitung von Dokumentenvorlagen werden u. a. auch Projektionen in Diagonalrichtung eingesetzt. Implementieren Sie diese Projektionen und überlegen Sie, welche Rolle diese in der Dokumentenverarbeitung spielen könnten.

12

Farbbilder

Farbbilder spielen in unserem Leben eine wichtige Rolle und sind auch in der digitalen Welt allgegenwärtig, ob im Fernsehen, in der Fotografie oder im digitalen Druck. Die Empfindung von Farbe ist ein faszinierendes und gleichzeitig kompliziertes Phänomen, das Naturwissenschaftler, Psychologen, Philosophen und Künstler seit Jahrhunderten beschäftigt [74, 77]. Wir beschränken uns in diesem Kapitel allerdings auf die wichtigsten technischen Zusammenhänge, die notwendig sind, um mit digitalen Farbbildern umzugehen. Die Schwerpunkte liegen dabei zum einen auf der programmtechnischen Behandlung von Farbbildern und zum anderen auf der Umwandlung zwischen unterschiedlichen Farbdarstellungen.

12.1 RGB-Farbbilder

Das RGB-Farbschema, basierend auf der Kombination der drei Primärfarben Rot (R), Grün (G) und Blau (B), ist aus dem Fernsehbereich vertraut und traditionell auch die Grundlage der Farbdarstellung auf dem Computer, bei Digitalkameras und Scannern sowie bei der Speicherung in Bilddateien. Die meisten Bildbearbeitungs- und Grafikprogramme verwenden RGB für die interne Darstellung von Farbbildern und auch in Java sind RGB-Bilder die Standardform.

RGB ist ein *additives* Farbsystem, d. h., die Farbmischung erfolgt ausgehend von Schwarz durch Addition der einzelnen Komponenten. Man kann sich diese Farbmischung als Überlagerung von drei Lichtstrahlen in den Farben Rot, Grün und Blau vorstellen, die in einem dunklen Raum auf ein weißes Blatt Papier gerichtet sind und deren Intensität individuell und kontinuierlich gesteuert werden kann. Die unterschiedliche Intensität der Farbkomponenten bestimmt dabei sowohl den Ton wie auch die Helligkeit der resultierenden Farbe. Auch Grau

und Weiß werden durch Mischung der drei Primärfarben in entsprechender Intensität erzeugt. Ähnliches passiert auch an der Bildfläche eines TV-Farbbildschirms oder CRT[1]-Computermonitors, wo kleine, eng aneinander liegende Leuchtpunkte in den drei Primärfarben durch einen Elektronenstrahl unterschiedlich stark angeregt werden und dadurch ein scheinbar kontinuierliches Farbbild erzeugen.

Der RGB-Farbraum bildet einen dreidimensionalen Würfel, dessen Koordinatenachsen den drei Primärfarben R, G und B entsprechen. Die RGB-Werte sind positiv und auf den Wertebereich $[0, C_{\max}]$ beschränkt, wobei für Digitalbilder meistens $C_{\max} = 255$ gilt. Jede mögliche Farbe $\mathbf{C}_i$ entspricht einem Punkt innerhalb des RGB-Farbwürfels mit den Komponenten

$$\mathbf{C}_i = (R_i, G_i, B_i),$$

wobei $0 \leq R_i, G_i, B_i \leq C_{\max}$. Häufig wird der Wertebereich der RGB-Komponenten auf das Intervall $[0, 1]$ normalisiert, sodass der Farbraum einen Einheitswürfel bildet (Abb. 12.1). Der Punkt $\mathbf{S} = (0, 0, 0)$ entspricht somit der Farbe Schwarz, $\mathbf{W} = (1, 1, 1)$ entspricht Weiß und alle Punkte auf der „Unbuntgeraden“ zwischen **S** und **W** sind Grautöne mit den Komponenten $R = G = B$.

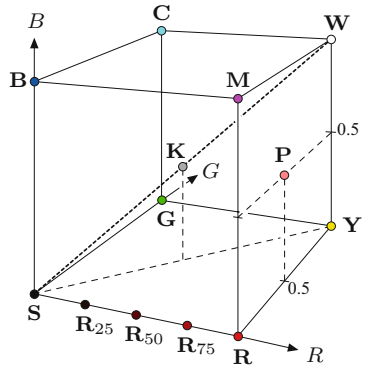


Abbildung 12.1
Darstellung des RGB-Farbraums als dreidimensionaler Einheitswürfel. Die Primärfarben Rot (R), Grün (G) und Blau (B) bilden die Koordinatenachsen. Die „reinen“ Farben Rot (**R**), Grün (**G**), Blau (**B**), Cyan (**C**), Magenta (**M**) und Gelb (**Y**) liegen an den Eckpunkten des Farbwürfels. Alle Grauwerte, wie der Farbpunkt **K**, liegen auf der Diagonalen („Unbuntgeraden“) zwischen dem Schwarzpunkt **S** und dem Weißpunkt **W**.

RGB-Werte

Pkt.	Farbe	R	G	B
S	Schwarz	0.00	0.00	0.00
R	Rot	1.00	0.00	0.00
Y	Gelb	1.00	1.00	0.00
G	Grün	0.00	1.00	0.00
C	Cyan	0.00	1.00	1.00
B	Blau	0.00	0.00	1.00
M	Magenta	1.00	0.00	1.00
W	Weiß	1.00	1.00	1.00
K	50% Grau	0.50	0.50	0.50
$\mathbf{R}_{75}$	75% Rot	0.75	0.00	0.00
$\mathbf{R}_{50}$	50% Rot	0.50	0.00	0.00
$\mathbf{R}_{25}$	25% Rot	0.25	0.00	0.00
P	Pink	1.00	0.50	0.50

Abb. 12.2 zeigt ein farbiges Testbild, das auch in den nachfolgenden Beispielen dieses Kapitels verwendet wird, sowie die zugehörigen RGB-Farbkomponenten als Intensitätsbilder.[2]

RGB ist also ein sehr einfaches Farbsystem und vielfach reicht bereits dieses elementare Wissen aus, um Farbbilder zu verarbeiten oder in andere Farbräume zu transformieren, wie nachfolgend in Abschn. 12.2 gezeigt. Vorerst nicht beantworten können wir die Frage, mit welchem

[1] *Cathode ray tube* (Kathodenstrahlröhre).

[2] Aus technischen Gründen sind in diesem Buch keine Farbabdrucke möglich, alle angeführten Farbbilder sind jedoch auf der zugehörigen Website zu finden.



RGB



R



G



B

Abbildung 12.2
Farbbild und zugehörige *RGB*-Komponenten. Die abgebildeten Früchte sind großteils gelb und rot und weisen daher einen hohen Anteil der *R*- und *G*-Komponenten auf. In diesen Bereichen ist der *B*-Anteil gering (dunkel dargestellt), außer an den hellen Glanzstellen der Äpfel, wo der Farbton in Weiß übergeht. Die Tischoberfläche im Vordergrund ist violett, weist also einen relativ hohen *B*-Anteil auf.

Farbwert ein bestimmtes RGB-Pixel in der Realität tatsächlich dargestellt wird oder was die Primärfarben *Rot*, *Grün* und *Blau* physisch wirklich bedeuten. Wir kümmern uns darum zwar zunächst nicht, widmen uns diesen wichtigen Details aber später wieder im Zusammenhang mit dem CIE-Farbraum (Abschn. 12.3.1).

12.1.1 Aufbau von Farbbildern

Farbbilder werden üblicherweise, genau wie Grauwertbilder, als Arrays von Pixeln dargestellt, wobei unterschiedliche Modelle für die Anordnung der einzelnen Farbkomponenten verwendet werden. Zunächst ist zu unterscheiden zwischen *Vollfarbenbildern*, die den gesamten Farbraum gleichförmig abdecken können, und so genannten *Paletten-* oder *Indexbildern*, die nur eine beschränkte Zahl unterschiedlicher Farben verwenden. Beide Bildtypen werden in der Praxis häufig eingesetzt.

Vollfarbenbilder

Ein Pixel in einem Vollfarbenbild kann jeden beliebigen Farbwert innerhalb des zugehörigen Farbraums annehmen, soweit es der (diskrete)

Wertebereich der einzelnen Farbkomponenten zulässt. Vollfarbenbilder werden immer dann eingesetzt, wenn Bilder viele unterschiedliche Farben enthalten können, wie etwa typische Fotografien oder gerenderte Szenen in der Computergrafik. Bei der Anordnung der Farbkomponenten unterscheidet man zwischen der so genannten *Komponentenanordnung* und der *gepackten Anordnung*.

Bei der **Komponentenanordnung** (auch als *planare* Anordung bezeichnet) sind die Farbkomponenten jeweils in getrennten Arrays von identischer Dimension angelegt. Ein Farbbild

$$I = (I_R, I_G, I_B)$$

wird daher gleichsam als Gruppe von zusammengehörigen Intensitätsbildern $I_R(u,v)$, $I_G(u,v)$ und $I_B(u,v)$ behandelt (Abb. 12.3) und der RGB-Farbwert des Komponentenbilds I an der Position (u,v) ergibt sich durch Zugriff auf die drei Teilbilder in der Form

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} \leftarrow \begin{pmatrix} I_R(u,v) \\ I_G(u,v) \\ I_B(u,v) \end{pmatrix}. \tag{12.1}$$

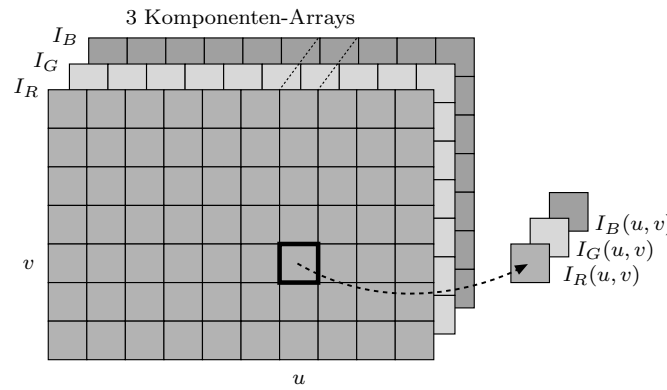


Abbildung 12.3
RGB-Farbbild in Komponentenanordnung. Die drei Farbkomponenten sind in getrennten Arrays I_R, I_G, I_B gleicher Größe angelegt.

Bei der **gepackten Anordnung** werden die einzelnen Farbkomponenten in ein gemeinsames Pixel zusammengefügt und in einem einzigen Bildarray gespeichert (Abb. 12.4), d. h.

$$I(u,v) = (R,G,B).$$

Den RGB-Farbwert eines gepackten Bilds I an der Stelle (u,v) erhalten wir durch Zugriff auf die einzelnen Komponenten des zugehörigen Farbpixels, also

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} \leftarrow \begin{pmatrix} \text{Red}(I(u,v)) \\ \text{Green}(I(u,v)) \\ \text{Blue}(I(u,v)) \end{pmatrix}. \tag{12.2}$$

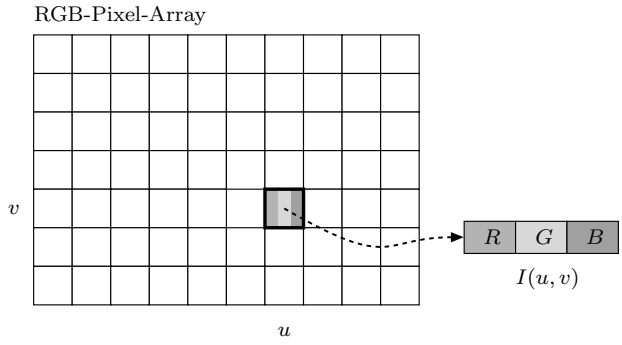


Abbildung 12.4
RGB-Farbbild in gepackter Anordnung. Die drei Farbkomponenten R, G, und B sind in ein gemeinsames Array-Element zusammengefügt.

Die Zugriffsfunktionen Red(), Green(), Blue() sind natürlich von der konkreten Realisierung der gepackten Farbpixel abhängig.

Indexbilder

Indexbilder erlauben nur eine beschränkte Zahl unterschiedlicher Farben und werden daher vor allem für Illustrationen, Grafiken und ähnlich „flache“ Bildinhalte verwendet, häufig etwa in der Form von GIF- oder PNG-Dateien für Web-Grafiken. Das eigentliche Pixel-Array selbst enthält dabei keine Farb- oder Helligkeitsdaten, sondern ganzzahlige Indizes k aus einer Farbtabelle oder „Palette“

$$P[k] = (P_R[k], P_G[k], P_B[k])$$

für $k = 0 \ldots N{-}1$ (Abb. 12.5). Dabei ist N die Größe der Farbtabelle und damit auch die maximale Anzahl unterschiedlicher Bildfarben (typischerweise $N = 2 \ldots 256$). Die Farbtabelle enthält beliebige RGB-Farbwerte (P_R, P_G, P_B) und muss daher als Teil des Bilds gespeichert werden. Der RGB-Farbwert eines Indexbilds I an der Stelle (u, v) ergibt sich als

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} \leftarrow \begin{pmatrix} P_R[k] \\ P_G[k] \\ P_B[k] \end{pmatrix}, \quad \text{wobei} \;\; k = I(u, v). \tag{12.3}$$

Bei der Umwandlung eines Vollfarbenbilds in ein Indexbild (z. B. von einem JPEG-Bild in ein GIF-Bild) besteht u. a. das Problem der optimalen Farbreduktion, also der Ermittlung der optimalen Farbtabelle und Zuordnung der ursprünglichen Farben. Darauf werden wir im Rahmen der Farbquantisierung (Abschn. 12.5) noch genauer eingehen.

12.1.2 Farbbilder in ImageJ

ImageJ stellt zwei einfache Formen von Farbbildern zur Verfügung:

- RGB-Vollfarbenbilder (RGB Color)
- Indexbilder (8-bit Color)

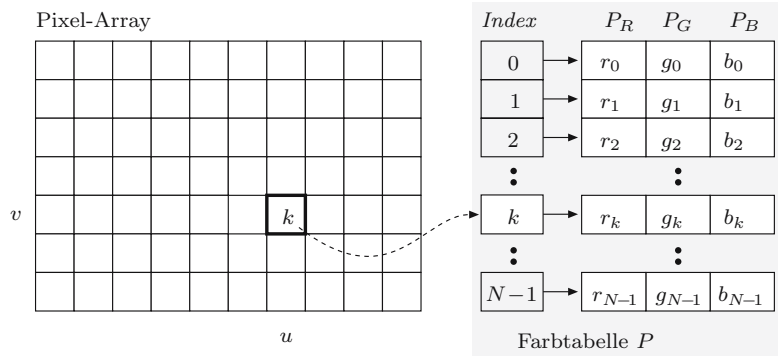


Abbildung 12.5
RGB-Indexbild. Das Bildarray selbst enthält keine Farbwerte, sondern für jedes Pixel einen Index k. Der eigentliche Farbwert wird durch den zugehörigen Eintrag in der Farbtabelle (Palette) $P[k]$ definiert.

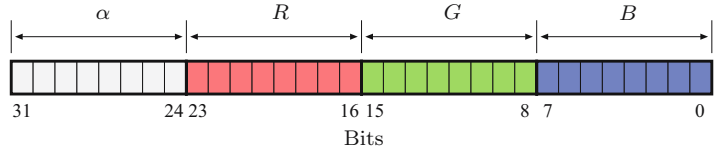


Abbildung 12.6
Aufbau eines RGB-Farbpixels in ImageJ. Innerhalb eines 32-Bit-`int`-Worts sind jeweils 8 Bits den Farbkomponenten R, G, B sowie dem (nicht benutzten) Transparenzwert α zugeordnet.

RGB-Vollfarbenbilder

RGB-Farbbilder in ImageJ haben eine gepackte Anordnung (siehe Abschn. 12.1.1), wobei jedes Farbpixel als 32-Bit-Wort vom Typ `int` dargestellt wird. Wie Abb. 12.6 zeigt, stehen für jede der *RGB*-Komponenten 8 Bit zur Verfügung, der Wertebereich der einzelnen Komponenten ist somit auf $0 \ldots 255$ beschränkt. Weitere 8 Bit sind für den Transparenzwert[3] α vorgesehen, und diese Anordnung entspricht auch dem in Java[4] allgemein üblichen Format für RGB-Farbbilder.

Zugriff auf RGB-Pixelwerte

Die Elemente des Pixel-Arrays eines RGB-Farbbilds sind vom Java-Standarddatentyp `int`. Die Zerlegung des gepackten `int`-Werts in die drei Farbkomponenten erfolgt durch entsprechende Bitoperationen, also Maskierung und Verschiebung von Bitmustern. Hier ein Beispiel, wobei wir annehmen, dass `ip` der Image-Prozessor eines RGB-Farbbilds ist:

```
1 int c = ip.getPixel(u,v);  // a color pixel
2 int r = (c & 0xff0000) >> 16; // red value
3 int g = (c & 0x00ff00) >> 8;  // green value
4 int b = (c & 0x0000ff);       // blue value
```

Dabei wird für jede der *RGB*-Komponenten der gepackte Pixelwert `c` zunächst durch eine bitweise UND-Operation (`&`) mit einer zugehörigen

[3] Der Transparenzwert α (Alphawert) bestimmt die „Durchsichtigkeit“ eines Farbpixels gegenüber dem Hintergrund oder bei Überlagerung mehrere Bilder. Der α-Komponente wird derzeit in ImageJ nicht verwendet.

[4] Java Advanced Window Toolkit – AWT (`java.awt`).

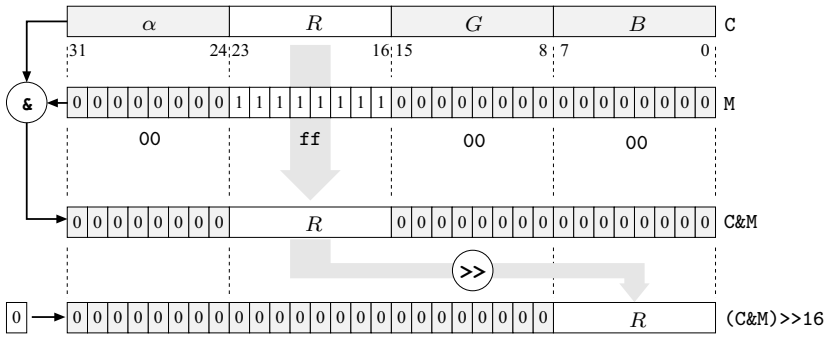


Abbildung 12.7
Zerlegung eines 32-Bit RGB-Farbpixels durch eine Folge von Bitoperationen. Die *R*-Komponente (Bits 16–23) des RGB-Pixels `C` (oben) wird zunächst durch eine bitweise UND-Operation (`&`) mit der Bitmaske `M = 0xff0000` isoliert. Alle Bits außerhalb der *R*-Komponente erhalten dadurch den Wert 0, das Bitmuster innerhalb der *R*-Komponente bleibt unverändert. Dieses Bitmuster wird anschließend um 16 Positionen nach rechts verschoben (`>>`), sodass die *R*-Komponente die untersten 8 Bits einnimmt und damit im Wertebereich 0 ... 255 liegt. Bei der Verschiebung werden von links Nullen eingefügt.

Bitmaske (angegeben in Hexadezimalnotation[5]) isoliert und anschließend mit dem Operator `>>` um 16 (für *R*) bzw. 8 (für *G*) Bitpositionen nach rechts verschoben (siehe Abb. 12.7).

Der „Zusammenbau“ eines RGB-Pixels aus einzelnen *R*-, *G*- und *B*-Werten erfolgt in umgekehrter Weise unter Verwendung der bitweisen ODER-Operation (`|`) und der Verschiebung nach links (`<<`):

```
1 int r = 169; // red value
2 int g = 212; // green value
3 int b = 17;  // blue value
4 int c = ((r & 0xff)<<16) | ((g & 0xff)<<8) | b & 0xff;
5 ip.putPixel(u,v,C);
```

Die Maskierung der Komponentenwerte (mit `0xff`) stellt in diesem Fall sicher, dass außerhalb der Bitpositionen 0 ... 7 (Wertebereich 0 ... 255) alle Bits auf 0 gesetzt werden. Ein vollständiges Beispiel für die Verarbeitung eines RGB-Farbbilds mithilfe dieser Bitoperationen ist in Prog. 12.1 dargestellt. Der Zugriff auf die Farbpixel erfolgt dabei ohne Zugriffsfunktionen (s. unten) direkt über das Pixel-Array, wodurch das Programm sehr effizient ist (siehe auch Abschn. B.1.3).

Als bequemere Alternative stellt die ImageJ-Klasse `ColorProcessor` erweiterte Zugriffsmethoden bereit, bei denen die *RGB*-Komponenten getrennt (als `int`-Array mit drei Elementen) übergeben werden. Hier ein Beispiel für deren Verwendung (`ip` ist vom Typ `ColorProcessor`):

```
1 int[] RGB = new int[3];
2 ...
3 RGB = ip.getPixel(u,v,RGB);
4 int r = RGB[0];
5 int g = RGB[1];
6 int b = RGB[2];
7 ...
8 ip.putPixel(u,v,RGB);
```

[5] Die Maske `0xff0000` ist von Typ `int` und entspricht dem 32-Bit-Binärmuster `00000000111111110000000000000000`.

Programm 12.1
Verarbeitung von RGB-Farbbildern mit Bitoperationen (ImageJ-Plugin, Variante 1). Das Plugin erhöht alle drei Farbkomponenten um 10 Einheiten. Es erfolgt ein direkter Zugriff auf das Pixel-Array (Zeile 12), die Farbkomponenten werden durch Bitoperationen getrennt (Zeile 14–16) und nach der Modifikation wieder zusammengefügt (Zeile 23). Der Rückgabewert `DOES_RGB` (definiert durch das Interface `PlugInFilter`) in der `setup()`-Methode zeigt an, dass dieses Plugin Vollfarbenbilder im RGB-Format bearbeiten kann (Zeile 28).

```
1  // File RGBbrighten1_.java
2  import ij.ImagePlus;
3  import ij.plugin.filter.PlugInFilter;
4  import ij.process.ImageProcessor;
5
6  public class RGBbrighten1_ implements PlugInFilter {
7
8    public void run(ImageProcessor ip) {
9      int[] pixels = (int[]) ip.getPixels();
10
11     for (int i = 0; i < pixels.length; i++) {
12       int c = pixels[i];
13       // split color pixel into rgb-components
14       int r = (c & 0xff0000) >> 16;
15       int g = (c & 0x00ff00) >> 8;
16       int b = (c & 0x0000ff);
17       // modify colors
18       r = r + 10; if (r > 255) r = 255;
19       g = g + 10; if (g > 255) g = 255;
20       b = b + 10; if (b > 255) b = 255;
21       // reassemble color pixel and insert into pixel array
22       pixels[i]
23         = ((r & 0xff)<<16) | ((g & 0xff)<<8) | b & 0xff;
24     }
25   }
26
27   public int setup(String arg, ImagePlus imp) {
28     return DOES_RGB; // this plugin works on RGB images
29   }
30 }
```

Ein ausführlicheres und vollständiges Beispiel zeigt Prog. 12.2 anhand eines einfachen Plugins, das alle drei Farbkomponenten eines RGB-Bilds um 10 Einheiten erhöht. Zu beachten ist dabei, dass die in das Bild eingesetzten Komponentenwerte den Bereich $0 \ldots 255$ nicht über- oder unterschreiten dürfen, da die `putPixel()`-Methode nur jeweils die untersten 8 Bits jeder Komponente verwendet und dabei selbst keine Wertebegrenzung durchführt. Fehler durch arithmetischen Überlauf sind andernfalls leicht möglich. Der Preis für die Verwendung dieser Zugriffsmethoden ist allerdings eine deutlich höhere Laufzeit (etwa Faktor 4 gegenüber Variante 1 in Prog. 12.1).

Öffnen und Speichern von RGB-Bildern

ImageJ unterstützt folgende Arten von Bilddateien für Vollfarbenbilder im RGB-Format:

- **TIFF** (nur unkomprimiert): 3×8-Bit-RGB. TIFF-Farbbilder mit 16 Bit Tiefe werden als Image-Stack mit drei 16-Bit Intensitätsbildern geöffnet.

Programm 12.2
Verarbeitung von RGB-Farbbildern ohne Bitoperationen (ImageJ-Plugin, Variante 2). Das Plugin erhöht alle drei Farbkomponenten um 10 Einheiten und verwendet dafür die erweiterten Zugriffsmethoden `getPixel(int, int, int[])` und `putPixel(int, int, int[])` der Klasse `ColorProcessor` (Zeile 17 bzw. 21). Die Laufzeit ist aufgrund der Methodenaufrufe ca. viermal höher als für Variante 1 (Prog. 12.1).

```
1  // File RGBbrighten2_.java
2  import ij.ImagePlus;
3  import ij.plugin.filter.PlugInFilter;
4  import ij.process.ColorProcessor;
5  import ij.process.ImageProcessor;
6
7  public class RGBbrighten2_ implements PlugInFilter {
8    static final int R = 0, G = 1, B = 2; // component indices
9
10   public void run(ImageProcessor ip) {
11     //make sure image is of type ColorProcessor
12     ColorProcessor cp = (ColorProcessor) ip;
13     int[] RGB = new int[3];
14
15     for (int v = 0; v < cp.getHeight(); v++) {
16       for (int u = 0; u < cp.getWidth(); u++) {
17         cp.getPixel(u, v, RGB);
18         RGB[R] = Math.min(RGB[R]+10, 255); // add 10 and
19         RGB[G] = Math.min(RGB[G]+10, 255); // limit to 255
20         RGB[B] = Math.min(RGB[B]+10, 255);
21         cp.putPixel(u, v, RGB);
22       }
23     }
24   }
25
26   public int setup(String arg, ImagePlus imp) {
27     return DOES_RGB; // this plugin works on RGB images
28   }
29 }
```

- **BMP**, **JPEG**: 3 × 8-Bit-RGB.
- **PNG** (nur lesen): 3 × 8-Bit-RGB.
- **RAW**: Über das ImageJ-Menü File→Import→Raw... können RGB-Bilddateien geöffnet werden, deren Format von ImageJ selbst nicht direkt unterstützt wird. Dabei ist die Auswahl unterschiedlicher Anordnungen der Farbkomponenten möglich.

Erzeugen von RGB-Bildern

Ein neues RGB-Farbbild erzeugt man in ImageJ am einfachsten durch Anlegen eines Objekts der Klasse `ColorProcessor`, wie folgendes Beispiel zeigt:

```
1 int w = 640, h = 480;
2 ColorProcessor cproc = new ColorProcessor(w,h);
3 ImagePlus cwin = new ImagePlus("My New Color Image", cproc);
4 cwin.show();
```

Wenn erforderlich, kann das Farbbild nachfolgend durch Erzeugen eines zugehörigen `ImagePlus`-Objekts (Zeile 3) und Anwendung der `show()`-

Methode angezeigt werden. Da `cproc` vom Typ `ColorProcessor` ist, wird natürlich auch das `ImagePlus`-Objekt `cwin` als Farbbild erzeugt. Dies könnte z. B. folgendermaßen überprüft werden:

```
4 if(cwin.getType()==ImagePlus.COLOR_RGB) {
5   int b = cwin.getBitDepth(); // b = 24
6   IJ.write("this is a RGB color image with " + b + " bits");
7 }
```

Indexbilder

Die Struktur von Indexbildern in ImageJ entspricht der in Abb. 12.5, wobei die Elemente des Index-Arrays 8 Bits groß sind, also maximal 256 unterschiedliche Farben dargestellt werden können. Programmtechnisch sind Indexbilder identisch zu Grauwertbildern, denn auch diese verfügen über eine „Farbtabelle“, die Pixelwerte auf entsprechende Grauwerte abbildet. Indexbilder unterscheiden sich nur dadurch von Grauwertbildern, dass die Einträge in der Farbtabelle echte RGB-Farbwerte sein können.

Öffnen und Speichern von Indexbildern

ImageJ unterstützt folgende Arten von Bilddateien für Indexbilder:

- **GIF**: Indexwerte mit $1 \ldots 8$ Bits ($2 \ldots 256$ Farben), 3×8-Bit-Farbwerte.
- **PNG** (nur lesen): Indexwerte mit $1 \ldots 8$ Bits ($2 \ldots 256$ Farben), 3×8-Bit-Farbwerte.
- **BMP**, **TIFF** (nur unkomprimiert): Indexwerte mit $1 \ldots 8$ Bits ($2 \ldots 256$ Farben), 3×8-Bit-Farbwerte.

Verarbeitung von Indexbildern

Das Indexformat dient vorrangig zur Speicherung von Bildern, denn auf Indexbilder selbst sind nur wenige Verarbeitungsschritte direkt anwendbar. Da die Indexwerte im Pixel-Array in keinem unmittelbaren Zusammenhang mit den zugehörigen Farbwerten (in der Farbtabelle) stehen, ist insbesondere die numerische Interpretation der Pixelwerte nicht zulässig. So etwa ist die Anwendung von Filteroperationen, die eigentlich für 8-Bit-Intensitätsbilder vorgesehen sind, i. Allg. wenig sinnvoll. Abbildung 12.8 zeigt als Beispiel die Anwendung eines Gauß-Filters und eines Medianfilters auf die Pixel eines Indexbilds, wobei durch den fehlenden quantitativen Zusammenhang mit den tatsächlichen Farbwerten natürlich völlig erratische Ergebnisse entstehen können. Auch die Anwendung des Medianfilters ist unzulässig, da zwischen den Indexwerten auch keine Ordnungsrelation existiert. Die bestehenden ImageJ-Funktionen lassen daher derartige Operationen in der Regel gar nicht zu. Im Allgemeinen erfolgt vor einer Verarbeitung eines Indexbilds eine Konvertierung in ein RGB-Vollfarbenbild und ggfs. eine anschließende Rückkonvertierung.



(a) (b) (c)

Abbildung 12.8
Anwendung eines Glättungsfilters auf ein Indexbild. Indexbild mit 16 Farben (a), Ergebnis nach Anwendung eines linearen Glättungsfilters (b) und eines 3×3-Medianfilters (c) auf das Pixel-Array. Die Anwendung des linearen Filters ist natürlich unsinnig, da zwischen den Indexwerten im Pixel-Array und der Bildintensität i. Allg. kein unmittelbarer Zusammenhang besteht. Das Medianfilter (c) liefert in diesem Fall zwar scheinbar plausible Ergebnisse, ist jedoch wegen der fehlenden Ordnungsrelation zwischen den Indexwerten ebenfalls unzulässig.

Soll ein Indexbild dennoch innerhalb eines ImageJ-Plugins verarbeitet werden, dann ist `DOES_8C` („8-bit color“) der zugehörige Rückgabewert für die `setup()`-Methode. Das Plugin in Prog. 12.3 zeigt beispielsweise, wie die Intensität der drei Farbkomponenten eines Indexbilds um jeweils 10 Einheiten erhöht wird (analog zu Prog. 12.1 und 12.2 für RGB-Bilder). Dabei wird ausschließlich die Farbtabelle modifiziert, während die eigentlichen Pixeldaten (Indexwerte) unverändert bleiben. Die Farbtabelle des `ImageProcessor` ist durch dessen `ColorModel`[6]-Objekt zugänglich, das über die Methoden `getColorModel()` und `setColorModel()` gelesen bzw. ersetzt werden kann.

Das `ColorModel`-Objekt ist bei Indexbildern (und auch bei 8-Bit-Grauwertbildern) vom Subtyp `IndexColorModel` und liefert die drei Farbtabellen (*maps*) für die Rot-, Grün- und Blaukomponenten als getrennte `byte`-Arrays. Die Größe dieser Tabellen ($2 \ldots 256$) wird über die Methode `getMapSize()` ermittelt. Man beachte, dass die `byte`-Elemente der Farbtabellen *ohne* Vorzeichen (*unsigned*) interpretiert werden, also im Wertebereich $0 \ldots 255$ liegen. Man muss daher – genau wie bei den Pixelwerten in Grauwertbildern – bei der Konvertierung auf `int`-Werte eine bitweise Maskierung mit `0xff` vornehmen (Prog. 12.3, Zeile 23–25).

Als weiteres Beispiel ist in Prog. 12.4 die Konvertierung eines Indexbilds in ein RGB-Vollfarbenbild vom Typ `ColorProcessor` gezeigt. Diese Form der Konvertierung ist problemlos möglich, denn es müssen lediglich für jedes Index-Pixel die zugehörigen *RGB*-Komponenten aus der Farbtabelle entnommen werden, wie in Gl. 12.3 beschrieben. Die Konvertierung in der Gegenrichtung erfordert hingegen die *Quantisierung*

[6] Definiert in der Klasse `java.awt.image.ColorModel`.

Programm 12.3
Beispiel für die Verarbeitung von Indexbildern (ImageJ-Plugin). Die Helligkeit des Bilds wird durch Veränderung der Farbtabelle um 10 Einheiten erhöht. Das eigentliche Pixel-Array (das die Indizes der Farbtabelle enthält) wird dabei nicht verändert.

```
1  // File IDXbrighten_.java
2
3  import ij.ImagePlus;
4  import ij.WindowManager;
5  import ij.plugin.filter.PlugInFilter;
6  import ij.process.ImageProcessor;
7  import java.awt.image.IndexColorModel;
8
9  public class IDXbrighten_ implements PlugInFilter {
10
11   public void run(ImageProcessor ip) {
12     IndexColorModel icm = (IndexColorModel) ip.getColorModel();
13     int pixBits = icm.getPixelSize();
14     int mapSize = icm.getMapSize();
15
16     //retrieve the current lookup tables (maps) for R,G,B
17     byte[] Rmap = new byte[mapSize]; icm.getReds(Rmap);
18     byte[] Gmap = new byte[mapSize]; icm.getGreens(Gmap);
19     byte[] Bmap = new byte[mapSize]; icm.getBlues(Bmap);
20
21     //modify the lookup tables
22     for (int idx = 0; idx < mapSize; idx++){
23       int r = 0xff & Rmap[idx]; //mask to treat as unsigned byte
24       int g = 0xff & Gmap[idx];
25       int b = 0xff & Bmap[idx];
26       Rmap[idx] = (byte) Math.min(r + 10, 255);
27       Gmap[idx] = (byte) Math.min(g + 10, 255);
28       Bmap[idx] = (byte) Math.min(b + 10, 255);
29     }
30     //create a new color model and apply to the image
31     IndexColorModel icm2 =
32       new IndexColorModel(pixBits, mapSize, Rmap, Gmap,Bmap);
33     ip.setColorModel(icm2);
34     //update the resulting image
35     WindowManager.getCurrentImage().updateAndDraw();
36   }
37
38   public int setup(String arg, ImagePlus imp) {
39     return DOES_8C; // this plugin works on indexed color images
40   }
41
42 }
```

des RGB-Farbraums (siehe Abschn. 12.5) und ist in der Regel aufwendiger. In der Praxis verwendet man dafür natürlich meistens die fertigen Konvertierungsmethoden in ImageJ (siehe S. 246).

Programm 12.4
Konvertierung eines Indexbilds in ein RGB-Vollfarbenbild (ImageJ-Plugin).

```
1  // File IDXtoRGB_.java
2
3  import ij.ImagePlus;
4  import ij.plugin.filter.PlugInFilter;
5  import ij.process.ColorProcessor;
6  import ij.process.ImageProcessor;
7  import java.awt.image.IndexColorModel;
8
9  public class IDXtoRGB_ implements PlugInFilter {
10   static final int R = 0, G = 1, B = 2;
11
12   public void run(ImageProcessor ip) {
13     int w = ip.getWidth();
14     int h = ip.getHeight();
15
16     //retrieve the lookup tables (maps) for R,G,B
17     IndexColorModel icm =
18         (IndexColorModel) ip.getColorModel();
19     int mapSize = icm.getMapSize();
20     byte[] Rmap = new byte[mapSize]; icm.getReds(Rmap);
21     byte[] Gmap = new byte[mapSize]; icm.getGreens(Gmap);
22     byte[] Bmap = new byte[mapSize]; icm.getBlues(Bmap);
23
24     //create new 24-bit RGB image
25     ColorProcessor cp = new ColorProcessor(w,h);
26     int[] RGB = new int[3];
27     for (int v = 0; v < h; v++) {
28       for (int u = 0; u < w; u++) {
29         int idx = ip.getPixel(u, v);
30         RGB[R] = Rmap[idx];
31         RGB[G] = Gmap[idx];
32         RGB[B] = Bmap[idx];
33         cp.putPixel(u, v, RGB);
34       }
35     }
36     ImagePlus cwin = new ImagePlus("RGB Image",cp);
37     cwin.show();
38   }
39
40   public int setup(String arg, ImagePlus imp) {
41     return DOES_8C + NO_CHANGES; //does not alter original image
42   }
43
44 }
```

Erzeugen von Indexbildern

Für die Erzeugung von Indexbildern ist in ImageJ keine spezielle Methode vorgesehen, da diese ohnehin fast immer durch Konvertierung bereits vorhandener Bilder generiert werden. Für den Fall, dass dies doch erforderlich ist, wäre z. B. folgende Methode geeignet:

```
1 ByteProcessor makeIndexColorImage(int w, int h, int nColors) {
2   byte[] Rmap = new byte[nColors]; // red, green, blue color map
3   byte[] Gmap = new byte[nColors];
4   byte[] Bmap = new byte[nColors];
5   // color maps need to be filled here
6   byte[] pixels = new byte[w * h];
7   IndexColorModel cm
8     = new IndexColorModel(8, nColors, Rmap, Gmap, Bmap);
9   return new ByteProcessor(w, h, pixels, cm);
10 }
```

Der Parameter `nColors` definiert die Anzahl der Farben – und damit die Größe der Farbtabellen – und muss einen Wert im Bereich 2 . . . 256 aufweisen. Natürlich müssten auch die drei Farbtabellen für die *RGB*-Komponenten (`Rmap`, `Gmap`, `Bmap`) und das Pixel-Array `pixels` noch mit geeigneten Werten befüllt werden.

Transparenz

Ein vor allem bei Web-Grafiken häufig verwendetes „Feature“ bei Indexbildern ist die Möglichkeit, einen der Indexwerte als vollständig transparent zu definieren. Dies ist in Java ebenfalls möglich und kann bei der Erzeugung des Farbmodells (`IndexColorModel`) eingestellt werden. Um beispielsweise in Prog. 12.3 den Farbindex 2 transparent zu machen, müsste man Zeile 32 etwa folgendermaßen ändern:

```
1   int tidx = 2; // index of transparent color
2   IndexColorModel icm2 =
3     new IndexColorModel(pixBits, mapSize, Rmap, Gmap, Bmap,
          tidx);
4   ip.setColorModel(icm2);
```

Allerdings wird die Transparenzeigenschaft derzeit in ImageJ sowohl bei der Darstellung wie auch beim Speichern von Bildern nicht berücksichtigt.

Konvertierung von Farbbildern in ImageJ

Für die Konvertierung zwischen verschiedenen Arten von Farb- und Grauwertbildern sind in ImageJ fertige Methoden für Bildobjekte vom Typ `ImagePlus` und Prozessor-Objekte vom Typ `ImageProcessor` verfügbar:

`ImageConverter(ImagePlus` *`ipl`*`)` Erzeugt ein `ImageConverter`-Objekt für das Bild *`ipl`*.
`void convertToGray8()` Konvertiert *`ipl`* in ein 8-Bit-Grauwertbild.
`void convertToGray16()` Konvertiert *`ipl`* in ein 16-Bit-Grauwertbild.
`void convertToGray32()` Konvertiert *`ipl`* in ein 32-Bit-Grauwertbild (`float`).
`void convertToRGB()` Konvertiert *`ipl`* in ein RGB-Farbbild.
`void convertRGBtoIndexedColor(int` *`nColors`*`)` Konvertiert das RGB-Vollfarbenbild *`ipl`* in ein Indexbild mit 8-Bit-Indexwerten und *`nColors`* Farben.
`void convertToHSB()` Konvertiert *`ipl`* in ein Farbbild im HSB-Farbraum (siehe Abschn. 12.2.3).
`void convertHSBToRGB()` Konvertiert das HSB-Farbbild *`ipl`* in ein RGB-Farbbild.

Tabelle 12.1
Methoden der ImageJ-Klasse `ImageConverter` zur Konvertierung von ImagePlus-Objekten.

`ImagePlus`-*Konvertierung mit* `ImageConverter`

ImageJ-Bildobjekte vom Typ `ImagePlus` können mithilfe der Klasse `ImageConverter` konvertiert werden, deren Methoden in Tabelle 12.1 zusammengefasst sind. Folgendes Beispiel erfordert `import ij.process.ImageConverter`:

```
1  ImagePlus ipl;
2  ...
3  ImageConverter ic = new ImageConverter(ipl);
4  ic.convertToRGB();
5  // ipl ist ab diesem Punkt ein RGB-Farbbild.
```

Zu beachten ist, dass dabei kein neues Bildobjekt angelegt, sondern das ursprüngliche Bild `ipl` selbst verändert wird.

`ImageProcessor`-*Konvertierung mit* `TypeConverter`

ImageJ-Objekte vom Typ `ImageProcessor` können mithilfe der Klasse `TypeConverter` konvertiert werden, deren Methoden in Tabelle 12.2 zusammengefasst sind. Folgendes Beispiel erfordert `import ij.process.TypeConverter`:

```
1  ImageProcessor ipr1;
2  ...
3  TypeConverter tc = new TypeConverter(ipr1, false);
4  ImageProcessor ipr2 = tc.convertToRGB();
5  // an dieser Stelle ist ipr2 vom Typ ColorProcessor,
6  // ipr1 ist unverändert.
```

Tabelle 12.2
Methoden der ImageJ-Klasse `TypeConverter` zur Konvertierung von `ImageProcessor`-Objekten.

`TypeConverter(ImageProcessor` *`ipr`*`, boolean` *`doScaling`*`)` Erzeugt ein `TypeConverter`-Objekt für den `ImageProcessor` *`ipr`*. *`doScaling`* gibt an, ob Werte bei der Konvertierung automatisch skaliert werden sollen.
`ImageProcessor convertToByte()` Erzeugt aus *`ipr`* einen neuen 8-Bit-Prozessor.
`ImageProcessor convertToShort()` Erzeugt aus *`ipr`* einen neuen 16-Bit-Grauwert-Prozessor.
`ImageProcessor convertToFloat(float[]` *`ctable`*`)` Erzeugt aus *`ipr`* einen neuen 32-Bit-Grauwert-Prozessor (`float`). *`ctable`* (*calibration table*) ist eine optionale Tabelle zur Umsetzung der Pixelwerte (kann `null` sein).
`ImageProcessor convertToRGB()` Erzeugt aus *`ipr`* einen neuen RGB-Color-Prozessor.

In diesem Fall wird ein neues Objekt (`ipr2`) vom Typ `ColorProcessor` erzeugt, das ursprüngliche Objekt (`ipr1`) bleibt unverändert.

12.2 Farbräume und Farbkonversion

Das RGB-Farbsystem ist aus Sicht der Programmierung eine besonders einfache Darstellungsform, die sich unmittelbar an den in der Computertechnik üblichen RGB-Anzeigegeräten orientiert. Dabei ist allerdings zu beachten, dass die Metrik des RGB-Farbraums mit der subjektiven Wahrnehmung nur wenig zu tun hat. So führt die Verschiebung von Farbpunkten im RGB-Raum um eine bestimmte Distanz, abhängig vom Farbbereich, zu sehr unterschiedlich wahrgenommenen Farbänderungen. Ebenso nichtlinear ist auch die Wahrnehmung von Helligkeitsänderungen im RGB-Raum.

Da sich Farbton, Farbsättigung und Helligkeit bei jeder Koordinatenbewegung gleichzeitig ändern, ist auch die manuelle Auswahl von Farben im RGB-Raum schwierig und wenig intuitiv. Alternative Farbräume, wie z. B. der HSV-Raum (s. Abschn. 12.2.3), erleichtern diese Aufgabe, indem subjektiv wichtige Farbeigenschaften explizit dargestellt werden. Ein ähnliches Problem stellt sich z. B. auch bei der automatischen Freistellung von Objekten vor einem farbigen Hintergrund, etwa in der *Blue-Box*-Technik beim Fernsehen oder in der Digitalfotografie. Auch in der TV-Übertragungstechnik oder im Druckbereich werden alternative Farbräume verwendet, die damit auch für die digitale Bildverarbeitung relevant sind.

Abb. 12.9 zeigt zur Illustration die Verteilung der Farben aus natürlichen Bildern in drei verschiedenen Farbräumen. Die Beschreibung dieser Farbräume und der zugehörigen Konvertierungen, einschließlich der Abbildung auf Grauwertbilder, ist Inhalt des ersten Teils dieses Abschnitts. Neben den klassischen und in der Programmierung häufig verwendeten

Definitionen wird jedoch der Einsatz von Referenzsystemen – insbesondere der am Ende dieses Abschnitts beschriebene CIEXYZ-Farbraum – beim Umgang mit digitalen Farbinformationen zunehmend wichtiger.

12.2.1 Umwandlung in Grauwertbilder

Die Umwandlung eines RGB-Farbbilds in ein Grauwertbild erfolgt über Berechnung des äquivalenten Grauwerts Y für jedes RGB-Pixel. In einfachster Form könnte Y als Durchschnittswert der drei Farbkomponenten in der Form

$$y = \text{Avg}(R, G, B) = \frac{R + G + B}{3} \tag{12.4}$$

ermittelt werden. Da die subjektive Helligkeit von Rot oder Grün aber wesentlich höher ist als die der Farbe Blau, ist das Ergebnis jedoch in Bildbereichen mit hohem Rot- oder Grünanteil zu dunkel und in blauen Bereichen zu hell. Üblicherweise verwendet man daher zur Berechnung des äquivalenten Intensitätswerts („Luminanz“) eine gewichtete Summe der Farbkomponenten in der Form

$$Y = \text{Lum}(R, G, B) = w_R \cdot R + w_G \cdot G + w_B \cdot B, \tag{12.5}$$

wobei meist die aus der Kodierung von analogen TV-Farbsignalen (s. auch Abschn. 12.2.4) bekannten Gewichte

$$w_R = 0.299 \qquad w_G = 0.587 \qquad w_B = 0.114 \tag{12.6}$$

bzw. die in ITU-BT.709 [43] für die digitale Farbkodierung empfohlenen Werte

$$w_R = 0.2125 \qquad w_G = 0.7154 \qquad w_B = 0.072 \tag{12.7}$$

verwendet werden. Die Gleichgewichtung der Farbkomponenten in Gl. 12.4 ist damit natürlich nur ein Sonderfall von Gl. 12.5.

Wegen der für TV-Signale geltenden Annahmen bzgl. der Gammakorrektur ist diese Gewichtung jedoch bei nichtlinearen RGB-Werten nicht korrekt. In [65] wurden für diesen Fall als Gewichte $w'_R = 0.309$, $w'_G = 0.609$ und $w'_B = 0.082$ vorgeschlagen. Korrekterweise müsste aber in lineare Komponentenwerte umgerechnet werden, wie beispielsweise in Abschn. 12.3.3 für sRGB gezeigt ist.

Neben der gewichteten Summe der RGB-Farbkomponenten werden mitunter auch (nichtlineare) Helligkeitsfunktionen anderer Farbsysteme, wie z. B. der *Value*-Wert V des HSV-Farbsystems (Gl. 12.11 in Abschn. 12.2.3) oder der *Luminance*-Wert L des HLS-Systems (Gl. 12.21) als Intensitätswert Y verwendet.

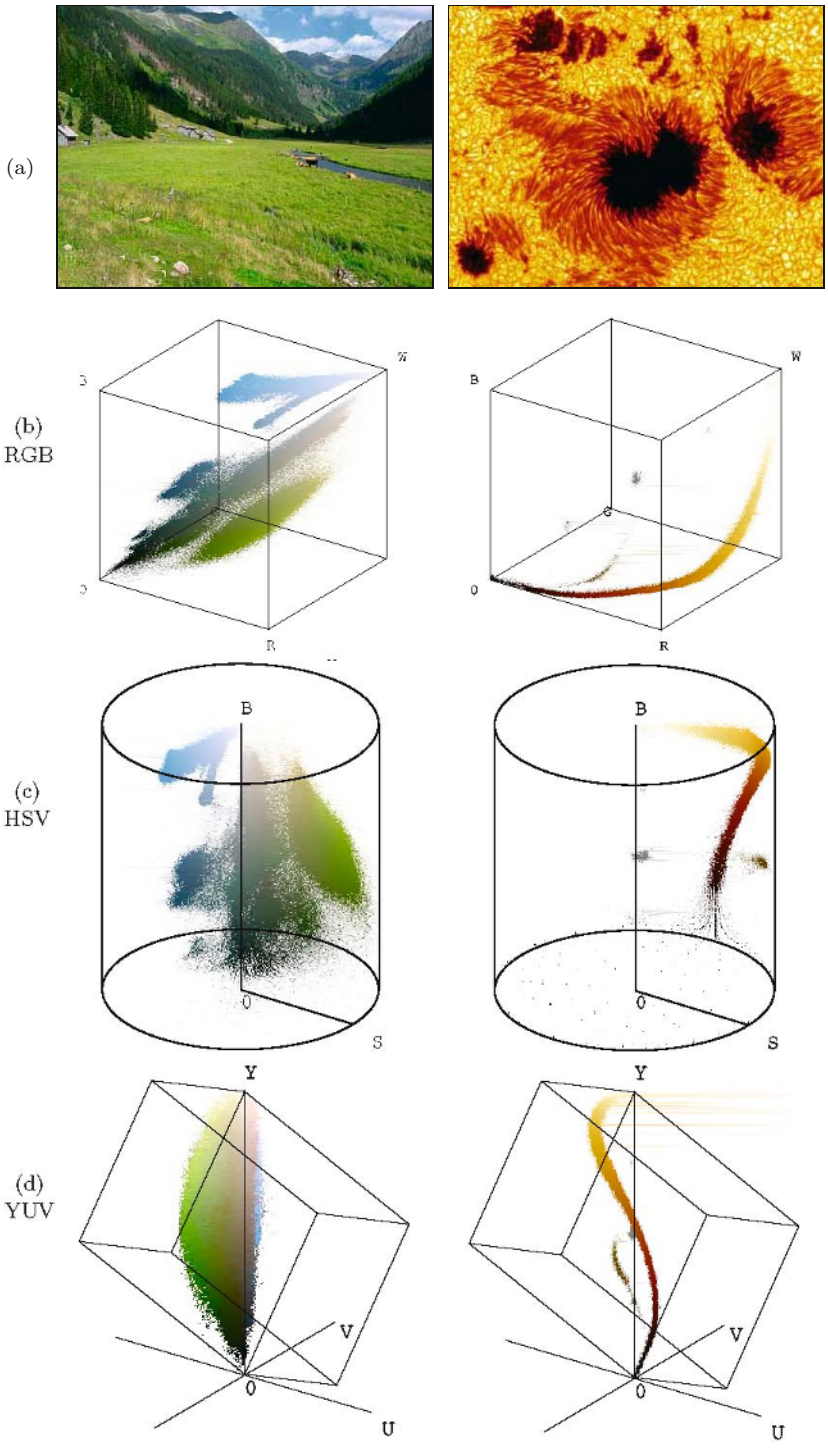


Abbildung 12.9
Beispiel für die Farbverteilung natürlicher Bilder in verschiedenen Farbräumen. Originalbilder: Landschaftsfoto mit dominanten Grün- und Blaukomponenten, Sonnenfleckenbild mit hohem Rot-/Gelb-Anteil (a), Verteilung im RGB-Raum (b), HSV-Raum (c) und YUV-Raum (d).

Unbunte Farbbilder

Ein RGB-Bild ist ein „unbuntes“ Grauwertbild, wenn für alle Bildelemente $I(u,v) = (R,G,B)$ gilt

$$R = G = B.$$

Um aus einem RGB-Bild die Farbigkeit vollständig zu entfernen, genügt es daher, die R,G,B-Komponenten durch den äquivalenten Grauwert Y (z. B. $Y = \mathrm{Lum}(R,G,B)$ aus Gl. 12.5–12.6) zu ersetzen, d. h.

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} \leftarrow \begin{pmatrix} Y \\ Y \\ Y \end{pmatrix}. \tag{12.8}$$

Das resultierende Grauwertbild sollte dabei den gleichen subjektiven Helligkeitseindruck wie das ursprüngliche Farbbild ergeben.

Grauwertkonvertierung in ImageJ

In ImageJ erfolgt die Umwandlung eines RGB-Farbbilds (vom Typ `ImageProcessor` bzw. `ColorProcessor`) in ein 8-Bit-Grauwertbild am einfachsten mithilfe der Klasse `TypeConverter` und der Methode

```
ImageProcessor convertToByte()
```

(siehe Tabelle 12.2 und Beispiel auf S. 247). Die in ImageJ verwendete Gewichtung der RGB-Komponenten entspricht mit $(0.299, 0.587, 0.114)$ der in Gl. 12.6.

12.2.2 Desaturierung von Farbbildern

Um die Farbanteile eines RGB-Bilds *kontinuierlich* zu reduzieren, wird für jedes Pixel zwischen dem ursprünglichen (R,G,B)-Farbwert und dem entsprechenden (Y,Y,Y)-Graupunkt im RGB-Raum linear interpoliert, d. h.

$$\mathbf{D} = \begin{pmatrix} R_D \\ G_D \\ B_D \end{pmatrix} = \begin{pmatrix} Y \\ Y \\ Y \end{pmatrix} + s_{\mathrm{col}} \cdot \begin{pmatrix} R-Y \\ G-Y \\ B-Y \end{pmatrix}, \tag{12.9}$$

wobei der Faktor $s_{\mathrm{col}} \in [0,1]$ die verbleibende Farbigkeit steuert (Abb. 12.10). Diesen graduellen Übergang bezeichnet man auch als „Desaturierung“ eines Farbbilds. Ein Wert $s_{\mathrm{col}} = 0$ eliminiert jede Farbigkeit und erzeugt ein reines Grauwertbild, bei $s_{\mathrm{col}} = 1$ bleiben die Farbwerte unverändert. Die kontinuierliche Desaturierung nach Gl. 12.9 ist als vollständiger ImageJ-Plugin in Prog. 12.5 realisiert.

Programm 12.5
Kontinuierliche Desaturierung von RGB-Farbbildern (ImageJ-Plugin). Die verbleibende Farbigkeit wird durch die Variable `s` in Zeile 8 gesteuert (entspr. s_{col} in Gl. 12.9).

```
1  // File DesaturateContRGB_.java
2
3  import ij.ImagePlus;
4  import ij.plugin.filter.PlugInFilter;
5  import ij.process.ImageProcessor;
6
7  public class DesaturateContRGB_ implements PlugInFilter {
8    double s = 0.3; // color saturation value
9
10   public void run(ImageProcessor ip) {
11     //iterate over all pixels
12     for (int v = 0; v < ip.getHeight(); v++) {
13       for (int u = 0; u < ip.getWidth(); u++) {
14
15         //get int-packed color pixel
16         int c = ip.getPixel(u, v);
17
18         //extract RGB components from color pixel
19         int r = (c & 0xff0000) >> 16;
20         int g = (c & 0x00ff00) >> 8;
21         int b = (c & 0x0000ff);
22
23         //compute equiv. gray value
24         double y = 0.299 * r + 0.587 * g + 0.114 * b;
25
26         // linear interpolate (yyy) ↔ (rgb)
27         r = (int) (y + s * (r - y));
28         g = (int) (y + s * (g - y));
29         b = (int) (y + s * (b - y));
30
31         // reassemble color pixel
32         c = ((r & 0xff)<<16) | ((g & 0xff)<<8) | b & 0xff;
33         ip.putPixel(u, v, c);
34       }
35     }
36   }
37
38   public int setup(String arg, ImagePlus imp) {
39     return DOES_RGB;
40   }
41
42 }
```

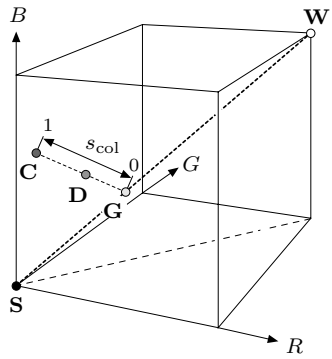
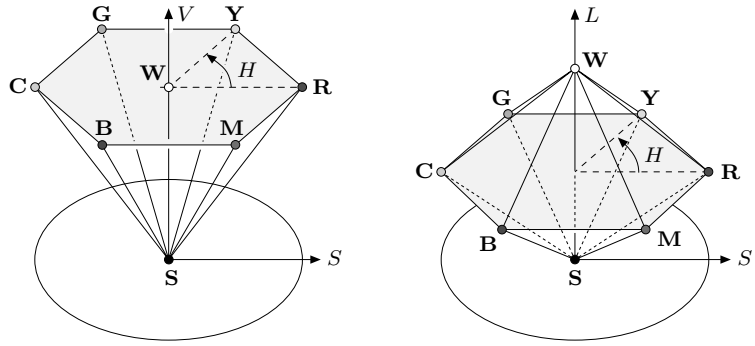


Abbildung 12.10
Desaturierung im RGB-Raum. Ein Farbpunkt $\mathbf{C} = (R, G, B)$ und sein zugehöriger Graupunkt $\mathbf{G} = (Y, Y, Y)$. Der mit dem Faktor s_{col} desaturierte Farbpunkt $\mathbf{D} = (R_D, G_D, B_D)$.



(a) HSV (b) HLS

Abbildung 12.11
HSV- und HLS-Farbraum – traditionelle Darstellung als hexagonale Pyramide bzw. Doppelpyramide. Der Helligkeitswert V bzw. L entspricht der vertikalen Richtung, die Farbsättigung S dem Radius von der Pyramidenachse und der Farbton H dem Drehwinkel. In beiden Fällen liegen die Grundfarben Rot (**R**), Grün (**G**), Blau (**B**) und die Mischfarben Gelb (**Y**), Cyan (**C**), Magenta (**M**) in einer gemeinsamen Ebene, Schwarz **S** liegt an der unteren Spitze. Der wesentliche Unterschied zwischen HSV- und HLS-Farbraum ist die Lage des Weißpunkts (**W**).

12.2.3 HSV/HSB- und HLS-Farbraum

Im **HSV**-Farbraum wird die Farbinformation durch die Komponenten *Hue*, *Saturation* und *Value* (Farbton, Farbsättigung, Helligkeit) dargestellt. Der Farbraum wird häufig – z. B. bei Adobe-Produkten oder im Java-API – auch mit „HSB“ (B = *Brightness*) bezeichnet.[7] Die Darstellung des HSV-Farbraums erfolgt traditionell in Form einer umgekehrten, sechseckigen Pyramide (Abb. 12.11 (a)), wobei die vertikale Achse dem V-Wert, der horizontale Abstand von der Achse dem S-Wert und der Drehwinkel dem H-Wert entspricht. Der Schwarzpunkt bildet die untere Spitze der Pyramide, der Weißpunkt liegt im Zentrum der Basisfläche. Die drei Grundfarben *Rot*, *Grün* und *Blau* und die paarweisen Mischfarben *Gelb*, *Cyan* und *Magenta* befinden sich an den sechs Eckpunkten der Basisfläche. Diese Darstellung als Pyramide ist zwar anschaulich, tatsächlich ergibt sich aus der mathematischen Definition aber eigentlich ein *zylindrischer* Raum, wie nachfolgend gezeigt (Abb. 12.12).

Der **HLS**-Farbraum[8] (*Hue*, *Luminance*, *Saturation*) ist dem HSV-Raum sehr ähnlich und sogar völlig identisch in Bezug auf die *Hue*-Komponente. Die Werte für *Luminance* und *Saturation* entsprechen

[7] Bisweilen wird der HSV-Raum auch als HSI (I = *Intensity*) bezeichnet.
[8] Die Bezeichnungen HLS und HSL werden synonym verwendet.

ebenfalls der vertikalen Koordinate bzw. dem Radius, werden aber anders als im HSV-Raum berechnet. Die übliche Darstellung des HLS-Raums ist die einer Doppelpyramide (Abb. 12.11 (b)), mit Schwarz und Weiß an der unteren bzw. oberen Spitze. Die Grundfarben liegen dabei an den Eckpunkten der Schnittebene zwischen den beiden Teilpyramiden. Mathematisch ist allerdings auch der HLS-Raum zylinderförmig (siehe Abb. 12.14).

RGB→HSV

Zur Konvertierung vom RGB- in den HSV-Farbraum berechnen wir aus den RGB-Farbkomponenten $R, G, B \in [0, C_{\max}]$ (typischerweise ist der maximale Komponentenwert $C_{\max} = 255$) zunächst die Sättigung (*saturation*)

$$S_{\text{HSV}} = \begin{cases} \frac{C_{\text{rng}}}{C_{\text{high}}} & \text{für } C_{\text{high}} > 0 \\ 0 & \text{sonst} \end{cases} \tag{12.10}$$

und die Helligkeit (*value*)

$$V_{\text{HSV}} = \frac{C_{\text{high}}}{C_{\max}}\,, \quad \text{wobei} \tag{12.11}$$

$$\begin{aligned} C_{\text{high}} = \max(R,G,B)\,, \quad C_{\text{low}} = \min(R,G,B)\,, \\ \text{und} \quad C_{\text{rng}} = C_{\text{high}} - C_{\text{low}}\,. \end{aligned} \tag{12.12}$$

Wenn alle drei RGB-Farbkomponenten denselben Wert aufweisen ($R = G = B$), dann handelt es sich um ein „unbuntes“ (graues) Pixel. In diesem Fall gilt $C_{\text{rng}} = 0$ und daher $S_{\text{HSV}} = 0$, der Farbton H_{HSV} ist unbestimmt. Für $C_{\text{rng}} > 0$ werden zur Berechnung von H_{HSV} zunächst die einzelnen Farbkomponenten in der Form

$$R' = \frac{C_{\text{high}} - R}{C_{\text{rng}}}, \qquad G' = \frac{C_{\text{high}} - G}{C_{\text{rng}}}, \qquad B' = \frac{C_{\text{high}} - B}{C_{\text{rng}}} \tag{12.13}$$

normalisiert. Abhängig davon, welche der drei ursprünglichen Farbkomponenten den Maximalwert darstellt, berechnet sich der Farbton als

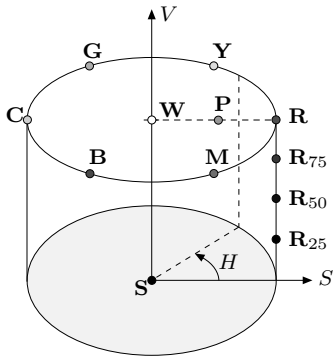
$$H' = \begin{cases} B' - G' & \text{wenn } R = C_{\text{high}} \\ R' - B' + 2 & \text{wenn } G = C_{\text{high}} \\ G' - R' + 4 & \text{wenn } B = C_{\text{high}} \end{cases} \tag{12.14}$$

Die resultierenden Werte für H' liegen im Intervall $[-1 \ldots 5]$. Wir normalisieren diesen Wert auf das Intervall $[0, 1]$ durch

$$H_{\text{HSV}} \leftarrow \frac{1}{6} \cdot \begin{cases} (H' + 6) & \text{für } H' < 0 \\ H' & \text{sonst.} \end{cases} \tag{12.15}$$

Alle drei Komponenten $H_{\text{HSV}}, S_{\text{HSV}}, V_{\text{HSV}}$ liegen damit im Intervall $[0, 1]$. Der Wert des Farbtons H_{HSV} ist bei Bedarf natürlich einfach in ein anderes Winkelintervall umzurechnen, z. B. in das $0 \ldots 360°$-Intervall durch $H^{\circ}_{\text{HSV}} \leftarrow H_{\text{HSV}} \cdot 360$.

Durch diese Definition wird der Einheitswürfel im RGB-Raum auf einen *Zylinder* mit Höhe und Radius der Länge 1 abgebildet (Abb. 12.12). Im Unterschied zur traditionellen Darstellung in Abb. 12.11 sind alle HSB-Punkte innerhalb des gesamten Zylinders auch zulässige Farbpunkte im RGB-Raum. Die Abbildung vom RGB- in den HSV-Raum ist nichtlinear, wobei sich interessanterweise der Schwarzpunkt auf die gesamte Grundfläche des Zylinders ausdehnt. Abbildung 12.12 beschreibt auch die Lage einiger markanter Farbpunkte im Vergleich zum RGB-Raum (siehe auch Abb. 12.1). In Abb. 12.13 sind für das Testbild aus Abb. 12.2 die einzelnen HSV-Komponenten als Grauwertbilder dargestellt.



RGB-/HSV-Werte

Pkt.	Farbe	R	G	B	H	S	V
S	Schwarz	0.00	0.00	0.00	—	0.00	0.00
R	Rot	1.00	0.00	0.00	0	1.00	1.00
Y	Gelb	1.00	1.00	0.00	1/6	1.00	1.00
G	Grün	0.00	1.00	0.00	2/6	1.00	1.00
C	Cyan	0.00	1.00	1.00	3/6	1.00	1.00
B	Blau	0.00	0.00	1.00	4/6	1.00	1.00
M	Magenta	1.00	0.00	1.00	5/6	1.00	1.00
W	Weiß	1.00	1.00	1.00	—	0.00	1.00
$\mathbf{R}_{75}$	75% Rot	0.75	0.00	0.00	0	1.00	0.75
$\mathbf{R}_{50}$	50% Rot	0.50	0.00	0.00	0	1.00	0.50
$\mathbf{R}_{25}$	25% Rot	0.25	0.00	0.00	0	1.00	0.25
P	Pink	1.00	0.50	0.50	0	0.5	1.00

Abbildung 12.12
HSV-Farbraum. Die Grafik zeigt den HSV-Farbraum als Zylinder mit den Koordinaten H (*hue*) als Winkel, S (*saturation*) als Radius und V (*brightness value*) als Distanz entlang der vertikalen Achse, die zwischen dem Schwarzpunkt **S** und dem Weißpunkt **W** verläuft. Die Tabelle listet die (R, G, B)- und (H, S, V)-Werte der in der Grafik markierten Farbpunkte auf. „Reine“ Farben (zusammengesetzt aus nur einer oder zwei Farbkomponenten) liegen an der Außenwand des Zylinders ($S = 1$), wie das Beispiel der graduell gesättigten Rotpunkte ($\mathbf{R}_{25}$, $\mathbf{R}_{50}$, $\mathbf{R}_{75}$, $\mathbf{R}$) zeigt.

Java-Implementierung

In Java ist die RGB-HSV-Konvertierung in der Klasse `java.awt.Color` durch die Klassenmethode

```
float[] RGBtoHSB (int r, int g, int b, float[] hsv)
```

implementiert (HSV und HSB bezeichnen denselben Farbraum). Die Methode erzeugt aus den `int`-Argumenten `r`, `g`, `b` (jeweils im Bereich $[0 \ldots 255]$) ein `float`-Array mit den Ergebnissen für H, S, V im Intervall $[0, 1]$. Falls das Argument `hsv` ein `float`-Array ist, werden die Ergebniswerte darin abgelegt, ansonsten (wenn `hsv = null`) wird ein neues Array erzeugt. Hier ein einfaches Anwendungsbeispiel:

```
1 import java.awt.Color;
2 ...
3 float[] hsv = new float[3];
4 int red = 128, green = 255, blue = 0;
5 hsv = Color.RGBtoHSB (red, green, blue, hsv);
6 float h = hsv[0];
7 float s = hsv[1];
8 float v = hsv[2];
9 ...
```

Eine mögliche Realisierung der Java-Methode `RGBtoHSB()` unter Verwendung der Definitionen in Gl. 12.11–12.15 ist in Prog. 12.6 gezeigt.

HSV→RGB

Zur Umrechnung eines HSV-Tupels $(H_{\text{HSV}}, S_{\text{HSV}}, V_{\text{HSV}})$, wobei H_{HSV}, S_{HSV} und $V_{\text{HSV}} \in [0,1]$, in entsprechende (R,G,B)-Farbwerte wird zunächst wiederum der zugehörige Farbsektor

$$H' = (6 \cdot H_{\text{HSV}}) \bmod 6 \tag{12.16}$$

ermittelt ($0 \le H' < 6$) und daraus die Zwischenwerte

$$\begin{array}{ll} c_1 = \lfloor H' \rfloor & x = (1 - S_{\text{HSV}}) \cdot v \\ c_2 = H' - c_1 & y = (1 - (S_{\text{HSV}} \cdot c_2)) \cdot V_{\text{HSV}} \\ & z = (1 - (S_{\text{HSV}} \cdot (1 - c_2))) \cdot V_{\text{HSV}} \end{array} \tag{12.17}$$

Die normalisierten RGB-Werte $R', G', B' \in [0,1]$ werden dann in Abhängigkeit von c_1 aus $v = V_{\text{HSV}}$, x, y und z wie folgt zugeordnet:[9]

$$(R', G', B') \leftarrow \begin{cases} (v, z, x) & \text{wenn } c_1 = 0 \\ (y, v, x) & \text{wenn } c_1 = 1 \\ (x, v, z) & \text{wenn } c_1 = 2 \\ (x, y, v) & \text{wenn } c_1 = 3 \\ (z, x, v) & \text{wenn } c_1 = 4 \\ (v, x, y) & \text{wenn } c_1 = 5 \end{cases} \tag{12.18}$$

Die Skalierung der RGB-Komponenten auf einen ganzzahligen Wertebereich $[0, N-1]$ (typischerweise $N = 256$) erfolgt abschließend durch

$$\begin{array}{l} R \leftarrow \min\big(\text{round}(N \cdot R'), N{-}1\big) \\ G \leftarrow \min\big(\text{round}(N \cdot G'), N{-}1\big) \\ B \leftarrow \min\big(\text{round}(N \cdot B'), N{-}1\big) \end{array} \tag{12.19}$$

[9] Die hier verwendeten Bezeichnungen x, y, z stehen in keinem Zusammenhang zum CIEXYZ-Farbraum (Abschn. 12.3.1).

```
1  static float[] RGBtoHSV (int R, int G, int B, float[] HSV) {
2    // R,G,B ∈ [0,255]
3    float H = 0, S = 0, V = 0;
4    float cMax = 255.0f;
5    int cHi = Math.max(R,Math.max(G,B)); // highest color value
6    int cLo = Math.min(R,Math.min(G,B)); // lowest color value
7    int cRng = cHi - cLo;         // color range
8
9    // compute value V
10   V = cHi / cMax;
11
12   // compute saturation S
13   if (cHi > 0)
14     S = (float) cRng / cHi;
15
16   // compute hue H
17   if (cRng > 0) { // hue is defined only for color pixels
18     float rr = (float)(cHi - R) / cRng;
19     float gg = (float)(cHi - G) / cRng;
20     float bb = (float)(cHi - B) / cRng;
21     float hh;
22     if (R == cHi)                  // r is highest color value
23       hh = bb - gg;
24     else if (G == cHi)             // g is highest color value
25       hh = rr - bb + 2.0f;
26     else                           // b is highest color value
27       hh = gg - rr + 4.0f;
28     if (hh < 0)
29       hh= hh + 6;
30     H = hh / 6;
31   }
32
33   if (HSV == null) // create a new HSV array if needed
34     HSV = new float[3];
35   HSV[0] = H; HSV[1] = S; HSV[2] = V;
36   return HSV;
37 }
```

Programm 12.6
RGB-HSV Konvertierung (Java-Methode) zur Umrechnung eines einzelnen Farbtupels. Die Methode entspricht bzgl. Parametern, Rückgabewert und Ergebnissen der Standard-Java-Methode `Color.RGBtoHSB()`.

Java-Implementierung

In Java ist die HSV→RGB-Konversion in der Klasse `java.awt.Color` durch die Klassenmethode

`int HSBtoRGB (float h, float s, float v)`

implementiert, die aus den drei `float`-Werten $h, s, v \in [0,1]$ einen `int`-Wert mit 3×8 Bit in dem in Java üblichen RGB-Format (siehe Abb. 12.6) erzeugt. Eine mögliche Implementierung dieser Methode ist in Prog. 12.7 gezeigt.

Programm 12.7
HSV-RGB Konvertierung zur Umrechnung eines einzelnen Farbtupels (Java-Methode). Die Methode entspricht bzgl. Parametern, Rückgabewert und Ergebnissen der Standard-Java-Methode `Color.HSBtoRGB()`.

```
1  static int HSVtoRGB (float h, float s, float v) {
2    // h, s, v ∈ [0, 1]
3    float rr = 0, gg = 0, bb = 0;
4    float hh = (6 * h) % 6;                    // h' ← (6 · h) mod 6
5    int  c1 = (int) hh;                        // c1 ← ⌊h'⌋
6    float c2 = hh - c1;
7    float x = (1 - s) * v;
8    float y = (1 - (s * c2)) * v;
9    float z = (1 - (s * (1 - c2))) * v;
10   switch (c1) {
11     case 0: rr=v; gg=z; bb=x; break;
12     case 1: rr=y; gg=v; bb=x; break;
13     case 2: rr=x; gg=v; bb=z; break;
14     case 3: rr=x; gg=y; bb=v; break;
15     case 4: rr=z; gg=x; bb=v; break;
16     case 5: rr=v; gg=x; bb=y; break;
17   }
18   int N = 256;
19   int r = Math.min(Math.round(rr*N),N-1);
20   int g = Math.min(Math.round(gg*N),N-1);
21   int b = Math.min(Math.round(bb*N),N-1);
22   // create int-packed RGB-color:
23   int rgb = ((r&0xff)<<16) | ((g&0xff)<<8) | b&0xff;
24   return rgb;
25 }
```

Abbildung 12.13
HSV-Komponenten für das Testbild aus Abb. 12.2. Die dunklen Bereiche in h_{HSV} entsprechen roten und gelben Farben mit *Hue*-Winkel nahe null.



h_{HSV} s_{HSV} v_{HSV}

RGB→HLS

Die Berechnung des *Hue*-Werts H_{HLS} für das HLS-Modell ist identisch zu HSV (Gl. 12.13–12.15), d. h.

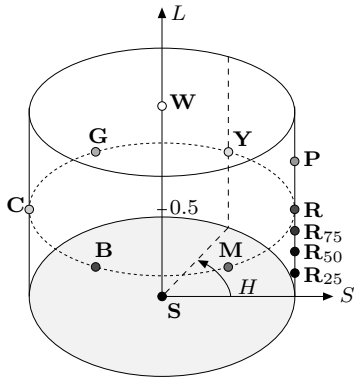
$$H_{\mathrm{HLS}} = H_{\mathrm{HSV}}. \tag{12.20}$$

Die übrigen Werte für L_{HLS} und S_{HLS} werden wie folgt berechnet (für C_{high}, C_{low}, C_{rng} siehe Gl. 12.12):

$$L_{\text{HLS}} \leftarrow \frac{C_{\text{high}} + C_{\text{low}}}{2} \tag{12.21}$$

$$S_{\text{HLS}} \leftarrow \begin{cases} 0 & \text{für } L_{\text{HLS}} = 0 \\ 0.5 \cdot \frac{C_{\text{rng}}}{L_{\text{HLS}}} & \text{für } 0 < L_{\text{HLS}} \leq 0.5 \\ 0.5 \cdot \frac{C_{\text{rng}}}{1 - L_{\text{HLS}}} & \text{für } 0.5 < L_{\text{HLS}} < 1 \\ 0 & \text{für } L_{\text{HLS}} = 1 \end{cases} \tag{12.22}$$

Durch diese Definition wird der Einheitswürfel im RGB-Raum wiederum auf einen Zylinder mit Höhe und Radius der Länge 1 abgebildet (Abb. 12.14). Im Unterschied zum HSV-Raum (Abb. 12.12) liegen die Grundfarben in einer gemeinsamen Ebene bei $L_{\text{HLS}} = 0.5$ und der Weißpunkt liegt außerhalb dieser Ebene bei $L_{\text{HLS}} = 1.0$. Der Schwarz- und der Weißpunkt werden durch diese nichtlineare Transformation auf die untere bzw. die obere Zylinderscheibe abgebildet. Alle HLS-Werte innerhalb des Zylinders haben zulässige Farbwerte im RGB-Raum. Abb. 12.15 zeigt die einzelnen HLS-Komponenten des Testbilds als Grauwertbilder.



RGB-/HLS-Werte

Pkt.	Farbe	R	G	B	H	S	L
S	Schwarz	0.00	0.00	0.00	—	0.00	0.00
R	Rot	1.00	0.00	0.00	0	1.00	0.50
Y	Gelb	1.00	1.00	0.00	1/6	1.00	0.50
G	Grün	0.00	1.00	0.00	2/6	1.00	0.50
C	Cyan	0.00	1.00	1.00	3/6	1.00	0.50
B	Blau	0.00	0.00	1.00	4/6	1.00	0.50
M	Magenta	1.00	0.00	1.00	5/6	1.00	0.50
W	Weiß	1.00	1.00	1.00	—	0.00	1.00
$\mathbf{R}_{75}$	75% Rot	0.75	0.00	0.00	0	1.00	0.375
$\mathbf{R}_{50}$	50% Rot	0.50	0.00	0.00	0	1.00	0.250
$\mathbf{R}_{25}$	25% Rot	0.25	0.00	0.00	0	1.00	0.125
P	Pink	1.00	0.50	0.50	0/6	1.00	0.75

Abbildung 12.14
HLS-Farbraum. Die Grafik zeigt den HLS-Farbraum als Zylinder mit den Koordinaten H (*hue*) als Winkel, S (*saturation*) als Radius und L (*lightness*) als Distanz entlang der vertikalen Achse, die zwischen dem Schwarzpunkt **S** und dem Weißpunkt **W** verläuft. Die Tabelle listet die (R, G, B)- und (H, S, L)-Werte der in der Grafik markierten Farbpunkte auf. „Reine" Farben (zusammengesetzt aus nur einer oder zwei Farbkomponenten) liegen an der unteren Hälfte der Außenwand des Zylinders ($S = 1$), wie das Beispiel der graduell gesättigten Rotpunkte ($\mathbf{R}_{25}$, $\mathbf{R}_{50}$, $\mathbf{R}_{75}$, **R**) zeigt. Mischungen aus drei Primärfarben, von denen mindesten eine Komponente voll gesättigt ist, liegen entlang der oberen Hälfte der Außenwand des Zylinders, wie z. B. der Punkt **P** (Pink).

HLS→RGB

Zur Rückkonvertierung von HLS in den RGB-Raum gehen wir davon aus, dass $H_{\text{HLS}}, S_{\text{HLS}}, L_{\text{HLS}} \in [0, 1]$. Falls $L_{\text{HLS}} = 0$ oder $L_{\text{HLS}} = 1$, so ist das Ergebnis

$$(R', G', B') \leftarrow \begin{cases} (0, 0, 0) & \text{für } L_{\text{HLS}} = 0 \\ (1, 1, 1) & \text{für } L_{\text{HLS}} = 1 \end{cases} \tag{12.23}$$

Andernfalls wird zunächst wiederum der zugehörige Farbsektor

$$H' = (6 \cdot H_{\text{HLS}}) \bmod 6 \tag{12.24}$$



H_{HLS} L_{HLS} S_{HLS}

Abbildung 12.15
HLS-Farbkomponenten H_{HLS} (*Hue*), L_{HLS} (*Luminance*) und S_{HLS} (*Saturation*).

ermittelt ($0 \le H' < 6$) und daraus die Werte

$$\begin{aligned} c_1 &= \lfloor H' \rfloor & d &= \begin{cases} S_{\text{HLS}} \cdot L_{\text{HLS}} & \text{für } L_{\text{HLS}} \le 0.5 \\ S_{\text{HLS}} \cdot (L_{\text{HLS}} - 1) & \text{für } L_{\text{HLS}} > 0.5 \end{cases} \\ c_2 &= H' - c_1 & & \\ w &= L_{\text{HLS}} + d & y &= w - (w - x) \cdot c_2 \\ x &= L_{\text{HLS}} - d & z &= x + (w - x) \cdot c_2 \end{aligned} \tag{12.25}$$

Die Zuordnung der *RGB*-Werte erfolgt dann ähnlich wie in Gl. 12.18 in der Form

$$(R', G', B') = \begin{cases} (w, z, x) & \text{wenn } c_1 = 0 \\ (y, w, x) & \text{wenn } c_1 = 1 \\ (x, w, z) & \text{wenn } c_1 = 2 \\ (x, y, w) & \text{wenn } c_1 = 3 \\ (z, x, w) & \text{wenn } c_1 = 4 \\ (w, x, y) & \text{wenn } c_1 = 5 \end{cases} \tag{12.26}$$

Die Rückskalierung der auf $[0, 1]$ normalisierten $R'G'B'$-Farbkomponenten in den Wertebereich $[0, 255]$ wird wie in Gl. 12.19 vorgenommen.

Java-Implementierung (RGB↔HLS)

Im Standard-Java-API oder in ImageJ ist derzeit keine Methode für die Konvertierung von Farbwerten von RGB nach HLS oder umgekehrt vorgesehen. Prog. 12.8 zeigt eine mögliche Implementierung der RGB-HLS-Konvertierung unter Verwendung der Definitionen in Gl. 12.20–12.22. Die Rückkonvertierung HLS→RGB ist in Prog. 12.9 gezeigt.

HSV und HLS im Vergleich

Trotz der großen Ähnlichkeit der beiden Farbräume sind die Unterschiede bei den V-/L- und S-Komponenten teilweise beträchtlich, wie Abb. 12.16 zeigt. Der wesentliche Unterschied zwischen dem HSV- und HLS-Raum ist die Anordnung jener Farben, die zwischen dem Weißpunkt **W** und den „reinen“ Farbwerten (wie **R**, **G**, **B**, **Y**, **C**, **M**) liegen, die aus maximal zwei Primärfarben bestehen, von denen mindestens eine vollständig gesättigt ist.

Programm 12.8
RGB-HLS Konvertierung (Java-Methode).

```
1  static float[] RGBtoHLS (float R, float G, float B) {
2    // R,G,B assumed to be in [0,1]
3    float cHi = Math.max(R,Math.max(G,B)); // highest color value
4    float cLo = Math.min(R,Math.min(G,B)); // lowest color value
5    float cRng = cHi - cLo;        // color range
6
7    // compute luminance L
8    float L = (cHi + cLo)/2;
9
10   // compute saturation S
11   float S = 0;
12   if (0 < L && L < 1) {
13     float d = (L <= 0.5f) ? L : (1 - L);
14     S = 0.5f * cRng / d;
15   }
16
17   // compute hue H
18   float H=0;
19   if (cHi > 0 && cRng > 0) {     // a color pixel
20     float rr = (float)(cHi - R) / cRng;
21     float gg = (float)(cHi - G) / cRng;
22     float bb = (float)(cHi - B) / cRng;
23     float hh;
24     if (R == cHi)                  // r is highest color value
25       hh = bb - gg;
26     else if (G == cHi)             // g is highest color value
27       hh = rr - bb + 2.0f;
28     else                           // b is highest color value
29       hh = gg - rr + 4.0f;
30
31     if (hh < 0)
32       hh= hh + 6;
33     H = hh / 6;
34   }
35
36   return new float[] {H,L,S};
37 }
```

Zur Illustration zeigt Abb. 12.17 die unterschiedlichen Verteilungen von Farbpunkten im RGB-, HSV- und HLS-Raum. Ausgangspunkt ist dabei eine gleichförmige Verteilung von 1331 ($11 \times 11 \times 11$) Farbtupel im RGB-Farbraum im Raster von 0.1 in jeder Dimension. Dabei ist deutlich zu sehen, dass im HSV-Raum die maximal gesättigten Farbwerte ($s = 1$) kreisförmige Bahnen bilden und die Dichte zur oberen Fläche des Zylinders hin zunimmt. Im HLS-Raum verteilen sich hingegen die Farbpunkte symmetrisch um die Mittelebene und die Dichte ist vor allem im Weißbereich wesentlich geringer. Eine bestimmte Bewegung in diesem Bereich führt daher zu geringeren Farbänderungen und ermöglicht so

Programm 12.9
HLS-RGB Konvertierung (Java-Methode).

```
1  static float[] HLStoRGB (float H, float L, float S) {
2    // H,L,S assumed to be in [0,1]
3    float R = 0, G = 0, B = 0;
4
5    if (L <= 0)      // black
6      R = G = B = 0;
7    else if (L >= 1)  // white
8      R = G = B = 1;
9    else {
10     float hh = (6 * H) % 6;
11     int   c1 = (int) hh;
12     float c2 = hh - c1;
13     float d = (L <= 0.5f) ? (S * L) : (S * (1 - L));
14     float w = L + d;
15     float x = L - d;
16     float y = w - (w - x) * c2;
17     float z = x + (w - x) * c2;
18     switch (c1) {
19       case 0: R=w; G=z; B=x; break;
20       case 1: R=y; G=w; B=x; break;
21       case 2: R=x; G=w; B=z; break;
22       case 3: R=x; G=y; B=w; break;
23       case 4: R=z; G=x; B=w; break;
24       case 5: R=w; G=x; B=y; break;
25     }
26   }
27   return new float[] {R,G,B};
28 }
```

feinere Abstufungen bei der Farbauswahl im HLS-Raum, insbesondere bei Farbwerten, die in der oberen Hälfte des HLS-Zylinders liegen.

Beide Farbräume – HSV und HLS – werden in der Praxis häufig verwendet, z. B. für die Farbauswahl bei Bildbearbeitungs- und Grafikprogrammen. In der digitalen Bildverarbeitung ist vor allem auch die Möglichkeit interessant, durch Isolierung der *Hue*-Komponente Objekte aus einem homogen gefärbten (aber nicht notwendigerweise gleichmäßig hellen) Hintergrund automatisch freizustellen (auch als *Color Keying* bezeichnet). Dabei ist natürlich zu beachten, dass mit abnehmendem Sättigungswert (S) auch der Farbwinkel (H) schlechter bestimmt bzw. bei $S = 0$ überhaupt undefiniert ist. In solchen Anwendungen sollte daher neben dem H-Wert auch der S-Wert in geeigneter Form berücksichtigt werden.

12.2.4 TV-Komponentenfarbräume – YUV, YIQ und YC_bC_r

Diese Farbräume dienen zur standardisierten Aufnahme, Speicherung, Übertragung und Wiedergabe im TV-Bereich und sind in entsprechenden Normen definiert. YUV und YIQ sind die Grundlage der Farbkodie-

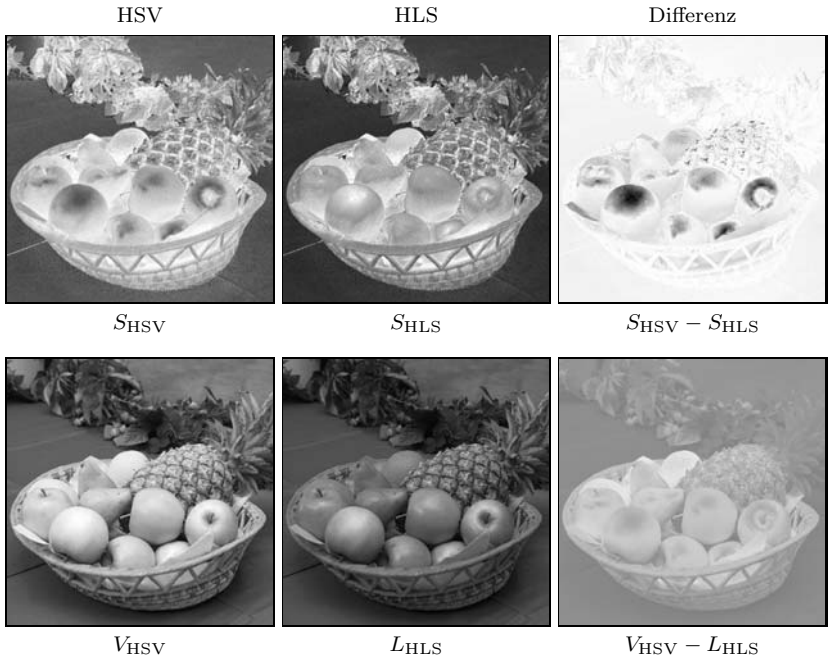


Abbildung 12.16
Vergleich zwischen HSV- und HLS-Komponenten. Im Differenzbild für die Farbsättigung $S_{\mathrm{HSV}} - S_{\mathrm{HLS}}$ (oben) sind positive Werte hell und negative Werte dunkel dargestellt. Der Sättigungswert ist in der HLS-Darstellung vor allem an den hellen Bildstellen deutlich höher, daher die entsprechenden negativen Werte im Differenzbild. Für die Intensität (*Value* bzw. *Luminance*) gilt allgemein, dass $V_{\mathrm{HSV}} \geq L_{\mathrm{HLS}}$, daher ist die Differenz $V_{\mathrm{HSV}} - L_{\mathrm{HLS}}$ (unten) immer positiv. Die H-Komponente (*Hue*) ist in beiden Darstellungen identisch.

rung beim analogen NTSC- und PAL-System, während $\mathrm{YC}_b\mathrm{C}_r$ Teil des internationalen Standards für digitales TV ist [40]. Allen Farbräumen gemeinsam ist die Trennung in eine Luminanz-Komponente Y und zwei gleichwertige Chroma-Komponenten, die unterschiedliche Farbdifferenzen kodieren. Dadurch konnte einerseits die Kompatibilität mit den ursprünglichen Schwarz/Weiß-Systemen erhalten werden, andererseits können bestehende Übertragungskanäle durch Zuweisung unterschiedliche Bandbreiten für Helligkeits- und Farbsignale optimal genutzt werden. Da das menschliche Auge gegenüber Unschärfe im Farbsignal wesentlich toleranter ist als gegenüber Unschärfe im Helligkeitssignal, kann die Übertragungsbandbreite für die Farbkomponenten deutlich (auf etwa 1/4 der Bandbreite des Helligkeitssignals) reduziert werden. Dieser Umstand wird auch bei der digitalen Farbbildkompression genutzt, u. a. beim JPEG-Verfahren, das z. B. eine $\mathrm{YC}_b\mathrm{C}_r$-Konvertierung von RGB-Bildern vorsieht. Aus diesem Grund sind diese Farbräume auch für die digitale Bildverarbeitung von Bedeutung, auch wenn man mit unkonvertierten YIQ- oder YUV-Bilddaten sonst selten in Berührung kommt.

YUV

YUV ist die Basis für die Farbkodierung im analogen Fernsehen, sowohl im nordamerikanischen NTSC- als auch im europäischen PAL-System. Die Luminanz-Komponente Y wird (wie bereits in Gl. 12.6 verwendet) aus den *RGB*-Komponenten in der Form

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \qquad (12.27)$$

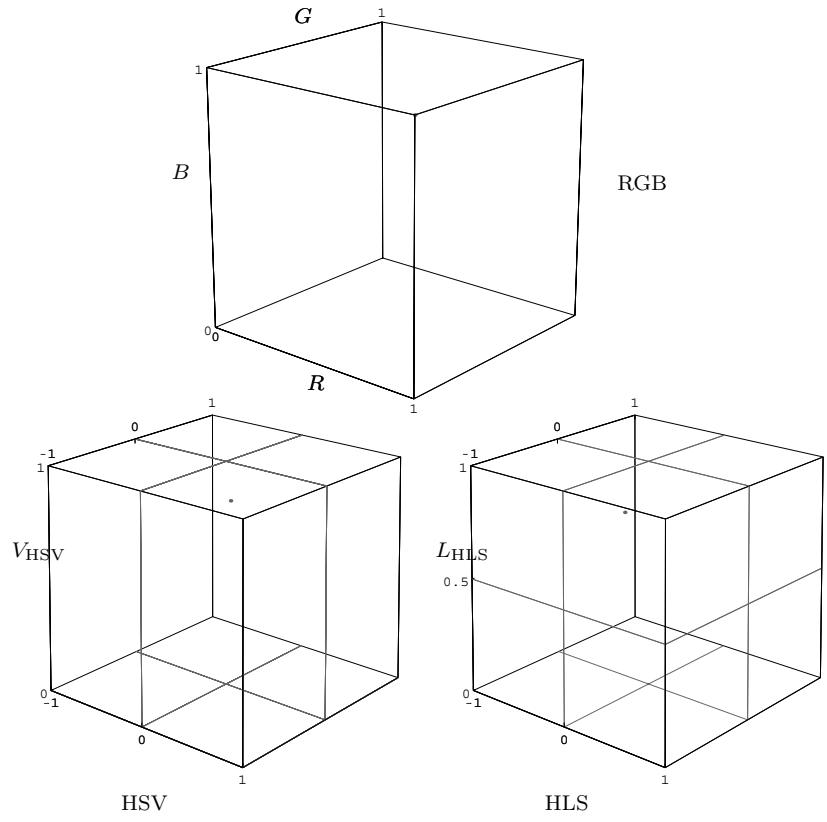


Abbildung 12.17
Verteilung von Farbwerten im RGB-, HSV- und HLS-Raum. Ausgangspunkt ist eine Gleichverteilung von Farbwerten im RGB-Raum (oben). Die zugehörigen Farbwerte im HSV- und HLS-Raum verteilen sich unsymmetrisch (HSV) bzw. symmetrisch (HLS) innerhalb eines zylindrischen Bereichs.

abgeleitet, wobei angenommen wird, dass die RGB-Werte bereits nach dem TV-Standard für die Wiedergabe gammakorrigiert sind ($\gamma_{\mathrm{NTSC}} = 2.2$ bzw. $\gamma_{\mathrm{PAL}} = 2.8$, siehe Abschn. 5.6). Die UV-Komponenten sind als gewichtete Differenz zwischen dem Luminanzwert und dem Blau- bzw. Rotwert definiert, konkret als

$$U = 0.492 \cdot (B - Y) \qquad \text{und} \qquad V = 0.877 \cdot (R - Y), \tag{12.28}$$

sodass sich insgesamt folgende Transformation von RGB nach YUV ergibt:

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{12.29}$$

Die umgekehrte Transformation von YUV nach RGB erhält man durch Inversion der Matrix in Gl. 12.29 als

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.000 & 1.140 \\ 1.000 & -0.395 & -0.581 \\ 1.000 & 2.032 & 0.000 \end{pmatrix} \cdot \begin{pmatrix} Y \\ U \\ V \end{pmatrix} . \tag{12.30}$$

YIQ

Eine im NTSC-System ursprünglich vorgesehene Variante des YUV-Schemas ist YIQ (I steht für „in-phase“, Q für „quadrature“), bei dem die durch U und V gebildeten Farbvektoren um 33° gedreht und gespiegelt sind, d. h.

$$\begin{pmatrix} I \\ Q \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} \cos\beta & \sin\beta \\ -\sin\beta & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} U \\ V \end{pmatrix}, \tag{12.31}$$

wobei $\beta = 0.576$ (33°). Die Y-Komponente ist gleich wie in YUV. Das YIQ-Schema hat bzgl. der erforderlichen Übertragungsbandbreiten gewisse Vorteile gegenüber dem YUV-Schema, wurde jedoch (auch in NTSC) praktisch vollständig von YUV abgelöst [46, S. 240].

YC_bC_r

Der YC_bC_r-Farbraum ist eine Variante von YUV, die international für Anwendungen im digitalen Fernsehen standardisiert ist und auch in der Bildkompression (z. B. bei JPEG) verwendet wird. Die Chroma-Komponenten C_b, C_r sind analog zu U, V Differenzwerte zwischen der Luminanz und der Blau- bzw. Rot-Komponente. Im Unterschied zu YUV steht allerdings die Gewichtung der *RGB*-Komponenten für die Luminanz Y in explizitem Zusammenhang zu den Koeffizienten für die Chroma-Werte C_b und C_r, und zwar in folgender Form [69, S. 16]:

$$\begin{aligned} Y &= w_R \cdot R + (1 - w_B - w_R) \cdot G + w_B \cdot B \\ C_b &= \frac{0.5}{1 - w_B} \cdot (B - Y) \\ C_r &= \frac{0.5}{1 - w_R} \cdot (R - Y) \end{aligned} \tag{12.32}$$

Analog dazu ist die Rücktransformation von YC_bC_r nach RGB definiert durch

$$\begin{aligned} R &= Y + \frac{1 - w_R}{0.5} \cdot C_r \\ G &= Y - \frac{w_B(1 - w_B)}{0.5 \cdot (1 - w_B - w_R)} \cdot C_b - \frac{w_k(1 - w_k)}{0.5 \cdot (1 - w_B - w_R)} \cdot C_r \\ B &= Y + \frac{1 - w_B}{0.5} \cdot C_b \end{aligned} \tag{12.33}$$

Die ITU[10]-Empfehlung BT.601 [44] spezifiziert die Werte $w_R = 0.299$ und $w_B = 0.114$ ($w_G = 0.587$)[11], und damit ergibt sich als zugehörige Transformation

[10] International Telecommunication Union (www.itu.int).

[11] Weil $w_R + w_G + w_B = 1$.

$$\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} , \tag{12.34}$$

bzw. als Rücktransformation

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.000 & 1.403 \\ 1.000 & -0.344 & -0.714 \\ 1.000 & 1.773 & 0.000 \end{pmatrix} \cdot \begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} . \tag{12.35}$$

In der für die digitale HDTV-Produktion bestimmten Empfehlung ITU-BT.709 [43] sind im Vergleich dazu die Werte $w_R = 0.2125$ und $w_B = 0.0721$ vorgesehen. Die UV-, IQ- und auch die C_bC_r-Werte können sowohl positiv als auch negativ sein. Bei der digitalen Kodierung der C_bC_r-Werte werden diese daher mit einem geeigneten Offset versehen, z. B. 128 bei 8-Bit-Komponenten, um ausschließlich positive Werte zu erhalten.

Abb. 12.18 zeigt die drei Farbräume YUV, YIQ und $\mathrm{YC}_b\mathrm{C}_r$ nochmals zusammen im Vergleich. Die UV-, IQ- und C_bC_r-Werte in den zwei rechten Spalten sind mit einem Offset von 128 versehen, um auch negative Werte darstellen zu können. Ein Wert von null entspricht daher im Bild einem mittleren Grau. Das $\mathrm{YC}_b\mathrm{C}_r$-Schema ist allerdings im Druckbild wegen der fast identischen Gewichtung der Farbkomponenten gegenüber YUV kaum unterscheidbar.

12.2.5 Farbräume für den Druck – CMY und CMYK

Im Unterschied zum *additiven* RGB-Farbmodell (und dazu verwandten Farbmodellen) verwendet man beim Druck auf Papier ein *subtraktives* Farbschema, bei dem jede Zugabe einer Druckfarbe die Intensität des reflektierten Lichts reduziert. Dazu sind wiederum zumindest drei Grundfarben erforderlich und diese sind im Druckprozess traditionell *Cyan* (C), *Magenta* (M) und *Gelb* (Y).[12]

Durch die subtraktive Farbmischung (auf weißem Grund) ergibt sich bei $C = M = Y = 0$ (keine Druckfarbe) die Farbe *Weiß* und bei $C = M = Y = 1$ (voller Sättigung aller drei Druckfarben) die Farbe *Schwarz*. Die Druckfarbe Cyan absorbiert *Rot* (R) am stärksten, Magenta absorbiert *Grün* (G), und Gelb absorbiert *Blau* (B). In der einfachsten Form ist das CMY-Modell daher definiert als

$$\begin{aligned} C &= 1 - R \\ M &= 1 - G \\ Y &= 1 - B \end{aligned} \tag{12.36}$$

Zur besseren Deckung und zur Vergrößerung des erzeugbaren Farbbereichs (Gamuts) werden die drei Grundfarben CMY in der Praxis durch die zusätzliche Druckfarbe Schwarz (K) ergänzt, wobei üblicherweise

[12] Y steht hier für *Yellow* und hat nichts mit der Luma-Komponente in YUV oder $\mathrm{YC}_b\mathrm{C}_r$ zu tun.

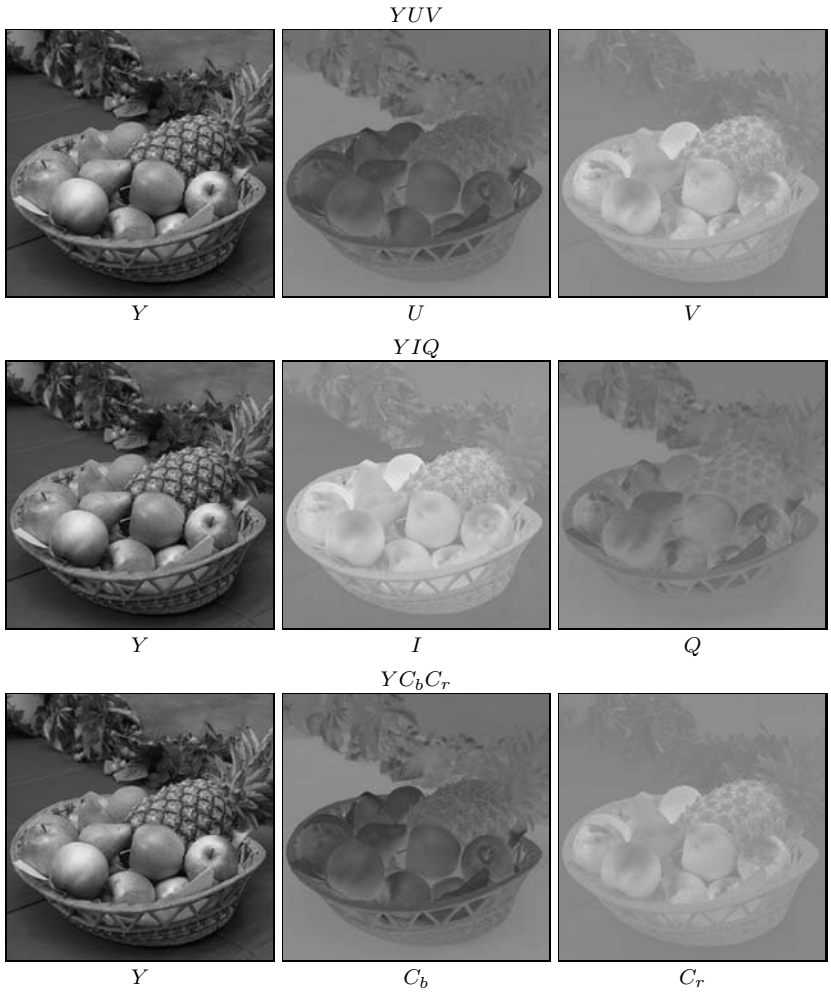


Abbildung 12.18
YUV-, YIQ- und YC_bC_r-Komponenten im Vergleich. Die Y-Werte sind in allen drei Farbräumen identisch.

$$K = \min(C, M, Y). \tag{12.37}$$

Gleichzeitig können die CMY-Werte bei steigendem Schwarzanteil reduziert werden, wobei man häufig folgende einfache Varianten für die Berechnung der modifizierten $C'M'Y'K'$-Komponenten findet:

CMY→CMYK (Version 1):

$$\begin{pmatrix} C' \\ M' \\ Y' \\ K' \end{pmatrix} \leftarrow \begin{pmatrix} C-K \\ M-K \\ Y-K \\ K \end{pmatrix} \tag{12.38}$$

CMY→CMYK (Version 2):

$$\begin{pmatrix} C' \\ M' \\ Y' \end{pmatrix} \leftarrow \begin{pmatrix} C-K \\ M-K \\ Y-K \end{pmatrix} \cdot \begin{cases} \frac{1}{1-K} & \text{für } K < 1 \\ 1 & \text{sonst} \end{cases} \quad K' \quad \leftarrow \quad K$$

In beiden Versionen wird als vierte Komponente der K-Wert unverändert (aus Gl. 12.37) übernommen und alle Grautöne (d. h., wenn $R = G = B$) werden ausschließlich mit der Druckfarbe K', also ohne Anteile von C', M', Y' dargestellt. Die zweite Variante (Gl. 12.39) ergibt durch die Korrektur der reduzierten CMY-Komponenten mit $\frac{1}{1-K}$ kräftigere Farbtöne in den dunklen Bildbereichen.

Beide dieser einfachen Definitionen führen jedoch in der Praxis kaum zu befriedigenden Ergebnissen und sind daher (trotz ihrer häufigen Erwähnung) nicht wirklich brauchbar. Abbildung 12.19 (a) zeigt das Ergebnis von Version 2 (Gl. 12.39) anhand eines Beispiels im Vergleich mit realistischen $CMYK$-Farbkomponenten, erzeugt mit Adobe Photoshop (Abb. 12.19 (c)). Besonders auffällig sind dabei die großen Unterschiede bei der Cyan-Komponente C. Außerdem wird deutlich, dass durch die Definition in Gl. 12.39 die Schwarz-Komponente K an den hellen Bildstellen generell zu hohe Werte aufweist.

In der Praxis sind der tatsächlich notwendige Schwarzanteil K und die Farbanteile CMY stark vom Druckprozess und vom verwendeten Papier abhängig und werden daher individuell kalibriert. In der Drucktechnik verwendet man spezielle Transferfunktionen für diese Aufgabe (z. B. im Adobe *PostScript*-Interpreter [51, S. 345]) die Funktionen $f_{\text{UCR}}(K)$ (*undercolor-removal function*) zur Korrektur der CMY-Komponenten und $f_{\text{BG}}(K)$ (*black-generation function*) zur Steuerung der Schwarz-Komponente, etwa in folgender Form:

CMY→CMYK (Version 3):

$$\begin{pmatrix} C' \\ M' \\ Y' \\ K' \end{pmatrix} \leftarrow \begin{pmatrix} C - f_{\text{UCR}}(K) \\ M - f_{\text{UCR}}(K) \\ Y - f_{\text{UCR}}(K) \\ f_{\text{BG}}(K) \end{pmatrix}, \tag{12.39}$$

wobei (wie in Gl. 12.37) $K = \min(C, M, Y)$. Die Funktionen f_{UCR} und f_{BG} sind in der Regel nichtlinear und die Ergebniswerte C', M', Y', K' werden (durch *Clamping*) auf das Intervall $[0, 1]$ beschränkt. Abb. 12.19 (b) zeigt ein Beispiel, wobei zur groben Annäherung an die Ergebnisse von Adobe Photoshop folgende Definitionen verwendet wurden:

$$f_{\text{UCR}}(K) = s_K \cdot K \tag{12.40}$$

$$f_{\text{BG}}(K) = \begin{cases} 0 & \text{für } K < K_0 \\ K_{\max} \cdot \frac{K-K_0}{1-K_0} & \text{für } K \geq K_0 \end{cases} \tag{12.41}$$

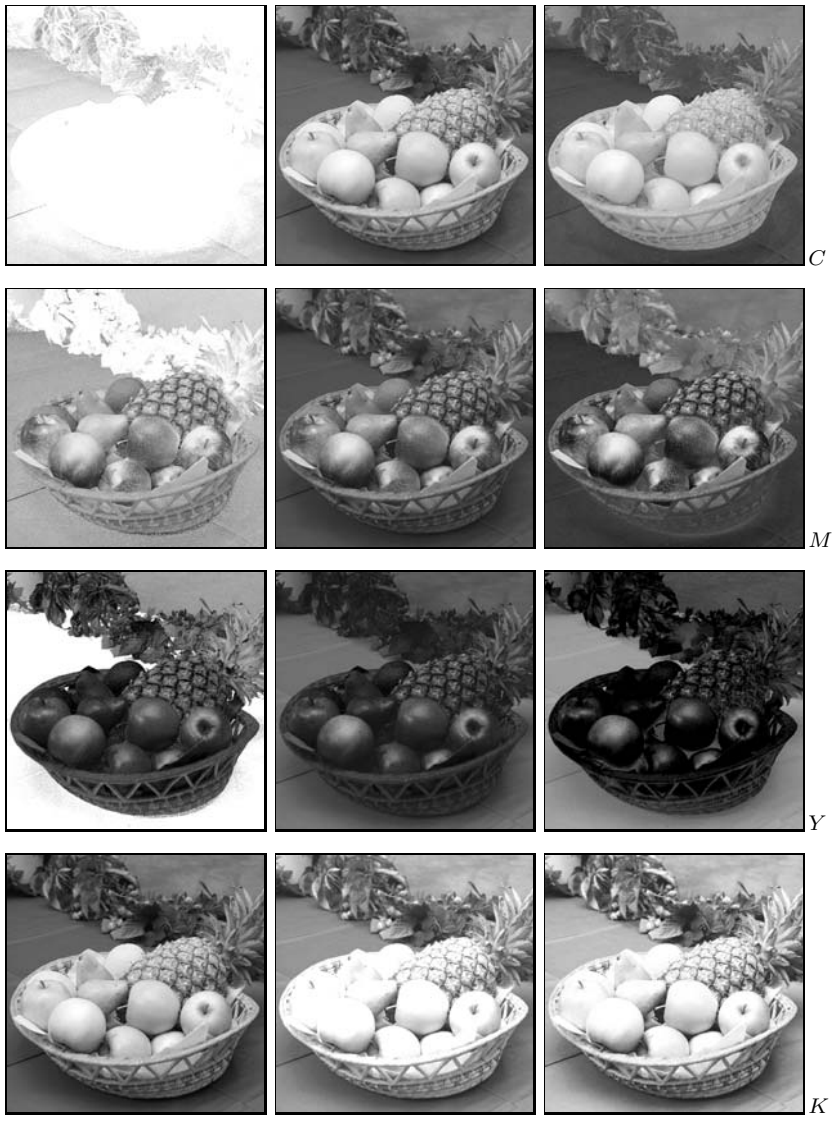


Abbildung 12.19
RGB-CMYK-Konvertierung im Vergleich. Einfache Konvertierung nach Gl. 12.39 (a), Verwendung von *undercolor-removal*- und *black-generation*-Funktionen nach Gl. 12.39 (b), Ergebnis aus Adobe Photoshop (c). Die Farbintensitäten sind invertiert dargestellt, dunkle Bildstellen entsprechen daher jeweils einem hohen CMYK-Farbanteil. Die einfache Konvertierung (a) liefert gegenüber dem Photoshop-Ergebnis (c) starke Abweichungen in den einzelnen Farbkomponenten, besonders beim C-Wert, und erzeugt einen zu hohen Schwarzanteil (K) an den hellen Bildstellen.

wobei $s_K = 0.1$, $K_0 = 0.3$ und $K_{\max} = 0.9$ (Abb. 12.20). f_{UCR} reduziert in diesem Fall (über Gl. 12.39) die CMY-Werte um 10% des K-Werts, was sich vorwiegend in den dunklen Bildbereichen mit hohem K-Wert auswirkt. Die Funktion f_{BG} (Gl. 12.41) bewirkt, dass für Werte $K < K_0$ – also in den hellen Bildbereichen – überhaupt kein Schwarzanteil beigefügt wird. Im Bereich $K = K_0 \ldots 1.0$ steigt der Schwarzanteil dann linear auf den Maximalwert $K_{\max}$. Das Ergebnis in Abb. 12.19 (b)

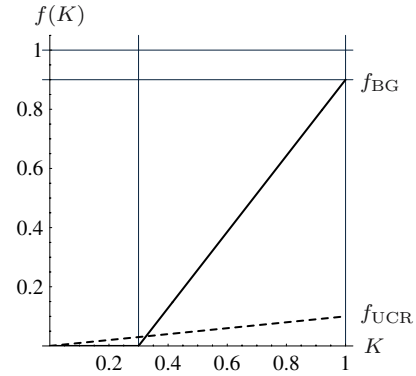


Abbildung 12.20
Beispielhafte *undercolor removal function* f_{UCR} (Gl. 12.40) zur Berechnung der $C'M'Y'$-Komponenten bzw. *black generation function* f_{BG} (Gl. 12.41) für die modifizierte K'-Komponente.

liegt vergleichsweise nahe an den als Referenz verwendeten CMYK-Komponenten aus Photoshop[13] (Abb. 12.19 (c)).

Trotz der verbesserten Ergebnisse ist auch diese letztgenannte Variante (3) zur Konvertierung von RGB nach CMYK nur eine grobe Annäherung, die allerdings bei unbekanntem oder wechselndem Wiedergabeverhalten durchaus brauchbar sein kann. Für professionelle Zwecke ist sie aber zu unpräzise und der technisch saubere Weg für die Konvertierung von CMYK-Komponenten führt über die Verwendung von CIE-basierten Referenzfarben, wie im nachfolgenden Abschnitt beschrieben.

12.3 Colorimetrische Farbräume

Für Anwendungen, die eine präzise, reproduzierbare und geräteunabhängige Darstellung von Farben erfordern, ist die Verwendung kalibrierter Farbsysteme unumgänglich. Diese Notwendigkeit ergibt sich z. B. in der gesamten Bearbeitungskette beim digitalen Farbdruck, aber auch bei der digitalen Filmproduktion oder bei Bilddatenbanken. Erfahrungsgemäß ist es keine einfache Angelegenheit, etwa einen Farbausdruck auf einem Laserdrucker zu erzeugen, der dem Erscheinungsbild auf dem Computermonitor einigermaßen nahekommt, und auch die Darstellung auf den Monitoren selbst sind in großem Ausmaß system- und herstellerabhängig.

Alle in Abschn. 12.2 betrachteten Farbräume beziehen sich, wenn überhaupt, auf die physischen Eigenschaften von Ausgabegeräten, also beispielsweise auf die Farben der Phosphorbeschichtungen in TV-Bildröhren oder der verwendeten Druckfarben. Um Farben in unterschiedlichen Ausgabemodalitäten ähnlich oder gar identisch erscheinen zu lassen, benötigt man eine Repräsentation, die unabhängig davon ist, in

[13] In Adobe Photoshop wird allerdings intern keine direkte Konvertierung von RGB nach CMYK durchgeführt, sondern als Zwischenstufe der CIE L*a*b*-Farbraum benutzt (siehe auch Abschn. 12.3.1).

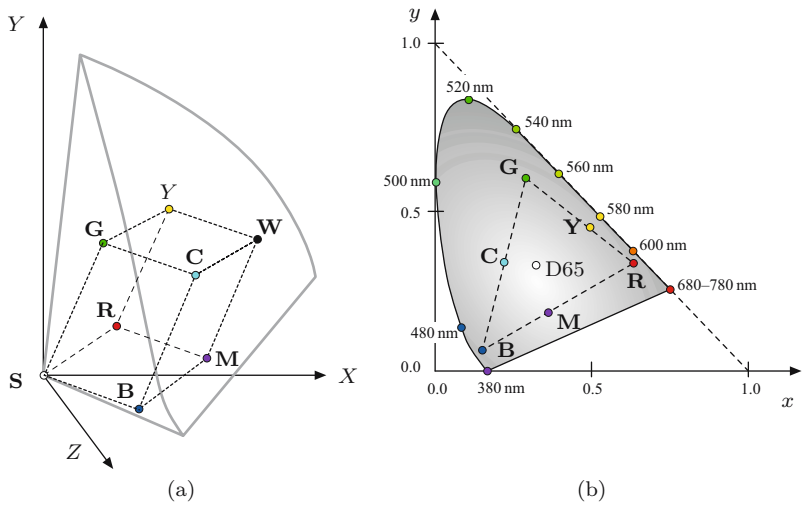


Abbildung 12.21
CIEXYZ-Farbraum und CIE-Farbdiagramm. Der CIEXYZ-Farbraum (a) wird durch die drei imaginären Primärfarben X, Y, Z aufgespannt. Die Y-Koordinate entspricht der Helligkeit, die X- und Z-Koordinaten bestimmen die Farbigkeit. Alle sichtbaren Farben liegen innerhalb des kegelförmigen Teilraums, in den der gewohnte RGB-Farbraum als verzerrter Würfel eingebettet ist. Das zweidimensionale CIE-Diagramm (b) entspricht einem horizontalen Schnitt durch den XYZ-Farbraum an der Höhe $Y = 1$, der nichtlinear abgebildet ist. Das CIE-Diagramm enthält daher alle sichtbaren Farbtöne (mit Wellenlängen von ca. 380–780 Nanometer), jedoch keine Helligkeitsinformation. Ein konkreter Farbraum mit beliebigen Primärfarben (Tristimuluswerten) **R**, **G**, **B** kann alle Farben innerhalb des dadurch definierten Dreiecks (lineare Hülle) darstellen. D65 markiert den xy-Farbwert der 6500° Normbeleuchtung.

welcher Weise ein bestimmtes Gerät diese Farben reproduziert. Farbsysteme, die Farben in einer geräteunabhängigen Form beschreiben können, bezeichnet man als *colorimetrisch* oder *kalibriert*.

12.3.1 CIE-Farbräume

Das bereits in den 1920er-Jahren entwickelte und von der CIE (*Commission Internationale d'Èclairage*)[14] 1931 standardisierte XYZ-Farbsystem ist Grundlage praktisch aller colorimetrischen Farbräume, die heute in Verwendung sind [68, S. 22].

CIEXYZ-Farbraum

Der CIEXYZ-Farbraum wurde durch umfangreiche Messungen unter streng definierten Bedingungen entwickelt und basiert auf drei imaginären Primärfarben X, Y, Z, die so gewählt sind, dass alle sichtbaren Farben mit ausschließlich positiven Komponenten beschrieben werden können. Die sichtbaren Farben liegen innerhalb einer dreidimensionalen Region, deren eigenartige Form einem Zuckerhut ähnlich ist, wobei die drei Primärfarben kurioserweise selbst nicht realisierbar sind (Abb. 12.21 (a)).

Die meisten gängigen Farbräume, wie z. B. der RGB-Farbraum, sind durch lineare Koordinatentransformationen (s. unten) in den XYZ-Farbraum überführbar und umgekehrt. Wie Abb. 12.21 (a) zeigt, ist daher der RGB-Farbraum als verzerrter Würfel im XYZ-Farbraum eingebettet, wobei durch die lineare Transformation die Geraden in RGB auch in XYZ wiederum Geraden sind. Das CIEXYZ-System ist (wie

[14] „Internat. Beleuchtungskommission“ (www.cie.co.at/cie/home.html).

auch der RGB-Farbraum) gegenüber dem menschlichen Sehsystem nichtlinear, d. h., Änderungen über Abstände fixer Größe werden nicht als gleichförmige Farbänderungen wahrgenommen.

CIE xy-Farbdiagramm

Im XYZ-Farbraum steigt, ausgehend vom Schwarzpunkt am Koordinatenursprung ($X = Y = Z = 0$), die Helligkeit der Farben entlang der Y-Koordinate an. Der Farbton selbst ist von der Helligkeit und damit von der Y-Koordinate unabhängig. Um die zugehörigen Farbtöne in einem zweidimensionalen Koordinatensystem übersichtlich darzustellen, definiert CIE als „Farbgewichte“ drei weitere Variable x, y, z als

$$x = \frac{X}{X+Y+Z}\,, \quad y = \frac{Y}{X+Y+Z}\,, \quad z = \frac{Z}{X+Y+Z}\,, \tag{12.42}$$

wodurch $x + y + z = 1$ und daher einer der Werte (z) redundant ist. Die Werte x und y bilden das Koordinatensystem für das bekannte, hufeisenförmige CIE-Diagramm (Abb. 12.21 (b)). Alle sichtbaren Farben des CIE-Systems können daher in der Form Yxy dargestellt werden, wobei Y die ursprüngliche Luminanzkomponente des XYZ-Systems ist.

Obwohl der mathematische Zusammenhang in Gl. 12.42 sehr einfach erscheint, ist diese Abbildung nicht leicht zu verstehen und keineswegs intuitiv. Das CIE-Diagramm bildet an einer konstanten Y-Position im XYZ-Farbraum einen horizontalen Schnitt, der nachfolgend nichtlinear auf das zweidimensionale xy-Koordinatensystem abgebildet wird. Für die Ebene $Y = 1$ gilt z. B.

$$x = \frac{X}{X+1+Z} \quad \text{und} \quad y = \frac{1}{X+1+Z}\,. \tag{12.43}$$

Die Rückrechnung auf Normfarben (mit der Helligkeit $Y = 1$) im dreidimensionalen XYZ-System erfolgt entsprechend durch

$$X = \frac{x}{y}\,, \qquad Y = 1\,, \qquad Z = \frac{z}{y} = \frac{1-x-y}{y}\,. \tag{12.44}$$

Das CIE-Diagramm bezieht sich zwar auf das menschliche Farbempfinden, ist jedoch gleichzeitig eine mathematische Konstruktion, die einige bemerkenswerte Eigenschaften aufweist. Entlang des hufeisenförmigen Rands liegen die xy-Werte aller *monochromatischen* („spektralreinen“) Farben mit dem höchsten Sättigungsgrad und unterschiedlichen Wellenlängen von unter 400 nm (violett) bis 780 nm (rot). Damit kann die Position jeder beliebigen Farbe in Bezug auf jede beliebige Primärfarbe berechnet werden. Eine Ausnahme ist die Verbindungsgerade („Purpurgerade“) zwischen 380 und 780 nm, auf der keine Spektralfarben liegen und deren zugehörige Purpurtöne nur durch das Komplement von gegenüberliegenden Farben erzeugt werden können.

Zur Mitte des CIE-Diagramms hin nimmt die Sättigung kontinuierlich ab bis zum Weißpunkt mit $x = y = \frac{1}{3}$ (bzw. $X = Y = Z = 1$) und Farbsättigung null. Auch alle farblosen Grauwerte werden auf diesen Weißpunkt abgebildet, genauso wie alle unterschiedlichen Helligkeitsausprägungen eines Farbtons jeweils nur einem einzigen xy-Punkt entsprechen. Alle möglichen Mischfarben liegen innerhalb jener konvexen Hülle, die im CIE-Diagramm durch die Koordinaten der verwendeten Primärfarben aufgespannt wird. Komplementärfarben liegen im CIE-Diagramm jeweils auf Geraden, die diagonal durch den (farblosen) Neutralpunkt verlaufen.

Normbeleuchtung

Ein zentrales Ziel der Colorimetrie ist die objektive Messung von Farben in der physischen Realität, wobei auch die Farbeigenschaften der *Beleuchtung* wesentlich sind. CIE definiert daher eine Reihe von Normbeleuchtungsarten (*illuminants*), von denen speziell zwei für digitale Farbräume wichtig sind:

D50 entspricht einer Farbtemperatur von ca. 5000° K und ähnelt damit einer typischen Glühlampenbeleuchtung. D50 wird als Referenzbeleuchtung für die Betrachtung von reflektierenden Bildern wie z. B. von Drucken empfohlen.

D65 entspricht einer Farbtemperatur von ca. 6500° K und simuliert eine typische Tageslichtbeleuchtung. D65 wird auch als Normweißlicht für emittierende Wiedergabegeräte (z. B. Bildschirme) verwendet.

Diese Normbeleuchtungsarten dienen zum einen zur Spezifikation des Umgebungslichts bei der Betrachtung, zum anderen aber auch zur Bestimmung von Referenzweißpunkten diverser Farbräume im CIE-Farbsystem (Tabelle 12.3). Darüber hinaus ist im CIE-System auch der zulässige Bereich des Betrachtungswinkels (mit ±2°) spezifiziert.

Dxx	Temp.	X	Y	Z	x	y
D50	5000° K	0.96429	1.00000	0.82510	0.3457	0.3585
D65	6500° K	0.95045	1.00000	1.08905	0.3127	0.3290
N	—	1.00000	1.00000	1.00000	1/3	1/3

Tabelle 12.3
CIE-Farbparameter für die Normbeleuchtungsarten D50 und D65. **N** ist der absolute Neutralpunkt im CIEXYZ-Raum.

Chromatische Adaptierung

Das menschliche Auge besitzt die Fähigkeit, Farben auch bei variierenden Betrachtungsverhältnissen und insbesondere bei Änderungen der Farbtemperatur der Beleuchtung als konstant zu empfinden. Ein weißes Blatt Papier erscheint uns sowohl im Tageslicht als auch unter einer Leuchtstoffröhre weiß, obwohl die spektrale Zusammensetzung des

Lichts, das das Auge erreicht, in beiden Fällen eine völlig andere ist. Im CIE-Farbsystem ist die Spezifikation der Farbtemperatur des Umgebungslichts berücksichtigt, denn die exakte Interpretation von XYZ-Farbwerten erfordert auch die Angabe des zugehörigen Referenzweißpunkts. So wird beispielsweise ein auf den Weißpunkt D50 bezogener Farbwert (X, Y, Z) bei der Darstellung auf einem Ausgabegerät mit Weißpunkt D65 im Allgemeinen anders wahrgenommen, auch wenn der absolute (gemessene) Farbwert derselbe ist.

Die Wahrnehmung eines Farbwerts erfolgt also relativ zum jeweiligen Weißpunkt. Beziehen sich zwei Farbsysteme auf unterschiedliche Weißpunkte $\mathbf{W}_1 = (X_{\mathrm{W1}}, Y_{\mathrm{W1}}, Z_{\mathrm{W1}})$ und $\mathbf{W}_2 = (X_{\mathrm{W2}}, Y_{\mathrm{W2}}, Z_{\mathrm{W2}})$, dann erfordert die korrekte Umrechnung im XYZ-Farbraum eine „chromatische Adaptierungstransformation“ (CAT) [40, Kap. 34]. Diese rechnet gegebene, auf den Weißpunkt $\mathbf{W}_1$ bezogene, XYZ-Werte (X_1, Y_1, Z_1) in die auf einen anderen Weißpunkt $\mathbf{W}_2$ bezogenen Werte (X_2, Y_2, Z_2) um. In der Praxis wird dazu meist eine lineare Transformation mit einer Matrix $\boldsymbol{M}_{\mathrm{CAT}}$ der Form

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \boldsymbol{M}_{\mathrm{CAT}}^{-1} \cdot \begin{pmatrix} \frac{r_2}{r_1} & 0 & 0 \\ 0 & \frac{g_2}{g_1} & 0 \\ 0 & 0 & \frac{b_2}{b_1} \end{pmatrix} \cdot \boldsymbol{M}_{\mathrm{CAT}} \cdot \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix} \tag{12.45}$$

verwendet, wobei (r_1, g_1, b_1) bzw. (r_2, g_2, b_2) die umgerechneten Tristimulus-Werte der beiden Weißpunkte $\mathbf{W}_1$ und $\mathbf{W}_2$ sind, d. h.

$$\begin{pmatrix} r_1 \\ g_1 \\ b_1 \end{pmatrix} = \boldsymbol{M}_{\mathrm{CAT}} \cdot \begin{pmatrix} X_{\mathrm{W1}} \\ Y_{\mathrm{W1}} \\ Z_{\mathrm{W1}} \end{pmatrix} \quad \text{und} \quad \begin{pmatrix} r_2 \\ g_2 \\ b_2 \end{pmatrix} = \boldsymbol{M}_{\mathrm{CAT}} \cdot \begin{pmatrix} X_{\mathrm{W2}} \\ Y_{\mathrm{W2}} \\ Z_{\mathrm{W2}} \end{pmatrix}. \tag{12.46}$$

Das häufig verwendete „Bradford“-Modell zur chromatische Adaptierung [40, S. 590] definiert

$$\boldsymbol{M}_{\mathrm{CAT}} = \begin{pmatrix} 0.8951 & 0.2664 & -0.1614 \\ -0.7502 & 1.7135 & 0.0367 \\ 0.0389 & -0.0685 & 1.0296 \end{pmatrix}. \tag{12.47}$$

In Verbindung mit Gl. 12.45 ergibt sich damit beispielsweise für die Umrechnung von D50-bezogenen XYZ-Koordinaten auf D65-bezogene Werte (Tabelle 12.3) die Transformation

$$\begin{aligned} \begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix} &= M_{65|50} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} \\ &= \begin{pmatrix} 0.955556 & -0.023049 & 0.063197 \\ -0.028302 & 1.009944 & 0.021018 \\ 0.012305 & -0.020494 & 1.330084 \end{pmatrix} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} \end{aligned} \tag{12.48}$$

bzw. in umgekehrter Richtung (von D65 nach D50)

$$\begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} = M_{50|65} \cdot \begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix} = M_{65|50}^{-1} \cdot \begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix}$$
$$= \begin{pmatrix} 1.047835 & 0.022897 & -0.050147 \\ 0.029556 & 0.990481 & -0.017056 \\ -0.009238 & 0.015050 & 0.752034 \end{pmatrix} \cdot \begin{pmatrix} X_{65} \\ Y_{65} \\ Z_{65} \end{pmatrix} . \quad (12.49)$$

Gamut

Die Gesamtmenge aller verschiedenen Farben, die durch ein Aufnahme- oder Ausgabegerät bzw. durch einen Farbraum dargestellt werden kann, bezeichnet man als „Gamut“. Dies ist normalerweise eine zusammenhängende Region im dreidimensionalen CIEXYZ-Raum bzw. – reduziert auf die möglichen Farbtöne ohne Berücksichtigung der Helligkeit – eine zweidimensionale, konvexe Region im CIE-Diagramm.

In Abb. 12.22 sind einige typische Beispiele für Gamut-Bereiche im CIE-Diagramm dargestellt. Das Gamut eines Ausgabegeräts ist primär von der verwendeten Technologie abhängig. So können typische Farbmonitore nicht sämtliche Farben innerhalb des zugehörigen Farbraum-Gamuts (z. B. sRGB) darstellen. Umgekehrt ist es möglich, dass technisch darstellbare Farben im verwendeten Farbraum nicht repräsentiert werden können. Besonders große Abweichungen sind beispielsweise zwischen dem RGB-Farbraum und dem Gamut von CMYK-Druckern möglich. Es existieren aber auch Ausgabegeräte mit sehr großem Gamut, wie das Beispiel des Laser-Displays in Abb. 12.22 demonstriert. Zur Repräsentation derart großer Farbbereiche und insbesondere zur Transformation zwischen unterschiedlichen Farbdarstellungen sind entsprechend dimensionierte Farbräume erforderlich, wie etwa der Adobe-RGB-Farbraum oder der L*a*b*-Farbraum (s. unten), der überhaupt den gesamten sichtbaren Teil des CIE-Diagramms umfasst.

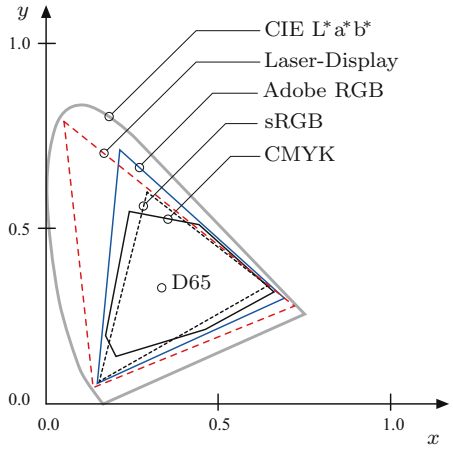


Abbildung 12.22
Gamut für verschiedene Farbräume bzw. Ausgabegeräte im CIE-Diagramm.

Varianten des CIE-Farbraums

Das ursprüngliche CIEXYZ- und das abgeleitete xy-Farbschema weisen vor allem den Nachteil auf, dass geometrische Abstände im Farbraum vom Betrachter visuell sehr unterschiedlich wahrgenommen werden. So erfolgen im Magenta-Bereich große Änderungen über relativ kurze Strecken, während im grünen Bereich die Farbtöne über weite Strecken vergleichsweise ähnlich sind. Es wurden daher Varianten des CIE-Systems für verschiedene Einsatzzwecke entwickelt mit dem Ziel, die Farbdarstellung besser an das menschliche Empfinden oder technische Gegebenheiten anzupassen, ohne dabei auf die formalen Qualitäten des CIE-Referenzsystems zu verzichten. Beispiele dafür sind die Farbräume CIE YUV, YU'V', L*u*v*, YC_bC_r und L*a*b* (s. unten).

Darüber hinaus stehen für die gängigsten Farbräume (siehe Abschn. 12.2) CIE-konforme Spezifikationen zur Verfügung, die eine verlässliche Umrechnung in jeden beliebigen anderen Farbraum ermöglichen.

12.3.2 CIE L*a*b*

Das L*a*b*-Modell (CIE 1976) wurde mit dem Ziel entwickelt, Farbdifferenzen gegenüber dem menschlichen Sehempfinden zu linearisieren und gleichzeitig ein intuitiv verständliches Farbsystem zu erhalten. L*a*b* wird beispielsweise in Adobe Photoshop[15] als Standardmodell für die Umrechnung zwischen Farbräumen verwendet. Die Koordinaten in diesem Farbraum sind die Helligkeit L^* und die beiden Farbkomponenten a^*, b^*, wobei a^* die Farbposition entlang der Grün-Rot-Achse und b^* entlang der Blau-Gelb-Achse im CIEXYZ-Farbraum spezifiziert. Alle drei Komponenten sind relativ und beziehen sich auf den neutralen Weißpunkt des Farbsystems $\mathbf{C}_{\mathrm{ref}} = (X_{\mathrm{ref}}, Y_{\mathrm{ref}}, Z_{\mathrm{ref}})$, wobei sie zusätzlich einer nichtlinearen Korrektur (ähnlich der modifizierten Gammafunktion in Abschn. 5.6.6) unterzogen werden.

Transformation CIEXYZ → L*a*b*

Die aktuelle Spezifikation[16] für die Umrechnung vom CIEXYZ-Farbraum in den L*a*b*-Farbraum ist nach ISO 13655 [45] folgende:

[15] Häufig wird L*a*b* einfach als „Lab"-Farbraum bezeichnet.

[16] Für die Umrechnung in den L*a*b*-Raum gibt es mehrere Definitionen, die sich allerdings nur geringfügig im Bereich sehr kleiner L-Werte unterscheiden.



L^* a^* b^*

Abbildung 12.23
L*a*b*-Komponenten. Zur besseren Darstellung wurde der Kontrast in den Bildern für a^* und b^* um 40% erhöht.

$$\begin{aligned} L^* &= 116 \cdot Y' - 16 \\ a^* &= 500 \cdot (X' - Y') \\ b^* &= 200 \cdot (Y' - Z') \end{aligned} \tag{12.50}$$

$$\text{wobei} \quad X' = f_1\big(\tfrac{X}{X_{\text{ref}}}\big), \quad Y' = f_1\big(\tfrac{Y}{Y_{\text{ref}}}\big), \quad Z' = f_1\big(\tfrac{Z}{Z_{\text{ref}}}\big)$$

$$\text{und} \quad f_1(c) = \begin{cases} c^{\frac{1}{3}} & \text{wenn } c > 0.008856 \\ 7.787 \cdot c + \frac{16}{116} & \text{wenn } c \le 0.008856 \end{cases}$$

Als Referenzweißpunkt $\mathbf{C}_{\text{ref}} = (X_{\text{ref}}, Y_{\text{ref}}, Z_{\text{ref}})$ in Gl. 12.50 wird üblicherweise D65 verwendet, d. h., $X_{\text{ref}} = 0.95047$, $Y_{\text{ref}} = 1.0$ und $Z_{\text{ref}} = 1.08883$ (Tabelle 12.3). Die Werte für L^* sind positiv und liegen normalerweise im Intervall $[0, 100]$ (häufig skaliert auf $[0, 255]$), können theoretisch aber auch darüber hinaus gehen. Die Werte für a^* und b^* liegen im Intervall $[-127, +127]$. Ein Beispiel für die Zerlegung eines Farbbilds in die zugehörigen L*a*b*-Komponenten zeigt Abb. 12.23. Tabelle 12.4 listet für einige ausgewählte RGB-Farbpunkte die zugehörigen CIE L*a*b*-Werte und als Referenz die CIEXYZ-Koordinaten. Die angegebenen $R'G'B'$-Werte sind (nichtlineare) sRGB-Koordinaten und beziehen sich auf den Referenzweißpunkt D65[17] (siehe Abschn. 12.3.5).

Transformation L*a*b* → CIEXYZ

Die Rücktransformation von L*a*b* den CIEXYZ-Raum ist folgendermaßen definiert:

$$\begin{aligned} X &= X_{\text{ref}} \cdot f_2\big(\tfrac{a^*}{500} + Y'\big) \\ Y &= Y_{\text{ref}} \cdot f_2\big(Y'\big) \\ Z &= Z_{\text{ref}} \cdot f_2\big(Y' - \tfrac{b^*}{200}\big) \end{aligned} \tag{12.51}$$

$$\text{wobei} \quad Y' = \tfrac{L^*+16}{116}$$

$$\text{und} \quad f_2(c) = \begin{cases} c^3 & \text{wenn } c^3 > 0.008856 \\ \frac{c - 16/116}{7.787} & \text{wenn } c^3 \le 0.008856 \end{cases}$$

[17] In Java sind die sRGB-Farbwerte allerdings nicht auf den Weißpunkt D65 sondern auf D50 bezogen, daher ergeben sich geringfügige Abweichungen.

Tabelle 12.4
CIE L*a*b*-Werte und zugehörige XYZ-Koordinaten für ausgewählte Farbpunkte in sRGB. Die sRGB-Komponenten R', G', B' sind nichtlinear (d. h. gammakorrigiert), Referenzweißpunkt ist D65 (s. auch Tabelle 12.3).

		sRGB			CIEXYZ			CIE L*a*b*		
Pkt.	Farbe	R'	G'	B'	X	Y	Z	L^*	a^*	b^*
S	Schwarz	0.00	0.00	0.00	0.0000	0.0000	0.0000	00.00	00.00	00.00
R	Rot	1.00	0.00	0.00	0.4124	0.2126	0.0193	53.23	80.11	67.22
Y	Gelb	1.00	1.00	0.00	0.7700	0.9278	0.1385	97.14	-21.55	94.48
G	Grün	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	0.5381	0.7874	1.0697	91.12	-48.08	-14.14
B	Blau	0.00	0.00	1.00	0.1805	0.0722	0.9505	32.30	79.20	-107.86
M	Magenta	0.00	1.00	1.00	0.5929	0.2848	0.9698	60.32	98.26	-60.83
W	Weiß	1.00	1.00	1.00	0.9505	1.0000	1.0890	100.00	0.00	0.00
K	Grau	0.50	0.50	0.50	0.2034	0.2140	0.2331	53.39	0.00	0.00
$\mathbf{R}_{75}$	75% Rot	0.75	0.00	0.00	0.2155	0.1111	0.0101	39.76	64.52	54.14
$\mathbf{R}_{50}$	50% Rot	0.50	0.00	0.00	0.0883	0.0455	0.0041	25.41	47.92	37.91
$\mathbf{R}_{25}$	25% Rot	0.25	0.00	0.00	0.0210	0.0108	0.0010	9.65	29.68	15.24
P	Pink	1.00	0.50	0.50	0.5276	0.3811	0.2483	68.10	48.40	22.82

Die vollständige Java-Implementierung dieser Konvertierung und einer entsprechenden Farbraum-Klasse (`ColorSpace`) sind in Prog. 12.10–12.11 (S. 286–287) dargestellt.

Bestimmung von Farbdifferenzen

Durch die relativ hohe Linearität in Bezug auf die menschliche Wahrnehmung von Farbabstufungen ist der L*a*b*-Farbraum zur Bestimmung von Farbdifferenzen gut geeignet [31, S. 57]. Konkret ist die Berechnung der Distanz zwischen zwei Farbpunkten $\mathbf{C}_1$ und $\mathbf{C}_2$ hier einfach über den euklidischen Abstand möglich, d. h.

$$\begin{aligned} \text{ColorDist}^*_{ab}(\mathbf{C}_1, \mathbf{C}_2) &= \|\mathbf{C}_1 - \mathbf{C}_2\| \\ &= \sqrt{(L^*_1 - L^*_2)^2 + (a^*_1 - a^*_2)^2 + (b^*_1 - b^*_2)^2}\,, \end{aligned} \tag{12.52}$$

wobei $\mathbf{C}_1 = (L^*_1, a^*_1, b^*_1)$ und $\mathbf{C}_2 = (L^*_2, a^*_2, b^*_2)$.

12.3.3 sRGB

CIE-basierte Farbräume wie L*a*b* (oder L*u*v*) sind geräteunabhängig und weisen ein ausreichend großes Gamut auf, um praktisch alle sichtbaren Farben des CIEXYZ-Farbraums darstellen zu können. Bei digitalen Anwendungen – wie etwa in der Computergrafik oder Multimedia –, die sich vor allem am Bildschirm als Ausgabemedium orientieren, ist die direkte Verwendung von CIE-basierten Farbräumen allerdings zu umständlich oder zu ineffizient.

sRGB („standard RGB“ [41]) wurde mit dem Ziel entwickelt, auch für diese Bereiche einen präzise definierten Farbraum zu schaffen, der durch entsprechende Abbildungsregeln im CIEXYZ-Farbraum verankert ist. Dies umfasst nicht nur die genaue Spezifikation der drei Primärfarben, sondern auch die des Weißpunkts, der Gammawerte und der Umgebungsbeleuchtung. sRGB besitzt im Unterschied zu L*a*b* ein relativ

kleines Gamut (Abb. 12.22) – das allerdings die meisten auf heutigen Monitoren darstellbaren Farben einschließt. sRGB ist auch nicht als universeller Farbraum konzipiert, erlaubt jedoch durch seine CIE-basierte Spezifikation eine exakte Umrechnung in andere Farbräume.

Standardisierte Speicherformate wie EXIF oder PNG basieren auf Ausgangsdaten in sRGB, das damit auch der De-facto-Standard für Digitalkameras und Farbdrucker im Consumer-Bereich ist [36]. sRGB eignet sich als vergleichsweise zuverlässiges Archivierungsformat für digitale Bilder vor allem in weniger kritischen Einsatzbereichen, die kein explizites Farbmanagement erfordern oder erlauben [82]. Nicht zuletzt ist sRGB auch das Standardfarbschema in Java und wird durch das Java-API umfassend unterstützt (siehe Abschn. 12.3.5).

Lineare vs. nichtlineare Farbwerte

Bei den Farbkomponenten in sRGB ist zu unterscheiden zwischen linearen und nichlinearen RGB-Werten. Die *nichtlinearen* Komponenten R', G', B' bilden die tatsächlichen sRGB-Farbtupel, die bereits mit einem fixen Gammawert (≈ 2.2) vorkorrigiert sind, so dass in den meisten Fällen eine ausreichend genaue Darstellung auf einem gängigen Farbmonitor ohne weitere Korrekturen möglich ist. Die zugehörigen *linearen RGB*-Komponenten beziehen sich durch lineare Abbildungen auf den CIEXYZ-Farbraum und können daher durch eine einfache Matrixmultiplikation aus XYZ-Koordinaten berechnet werden und umgekehrt, d. h.

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = M_{\mathrm{RGB}} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad \text{bzw.} \quad \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = M_{\mathrm{RGB}}^{-1} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} . \qquad (12.53)$$

Die wichtigsten Parameter des sRGB-Raums sind die xy-Koordinaten der Primärfarben (Tristimuluswerte) **R**, **G**, **B** (entsprechend der digitalen TV-Norm ITU-R 709-3 [43]) und des Weißpunkts **W** (D65), die eine eindeutige Zuordnung aller übrigen Farbwerte im CIE-Diagramm erlauben (Tabelle 12.5).

D65	R	G	B	x	y
R	1.00	0.00	0.00	0.6400	0.3300
G	0.00	1.00	0.00	0.3000	0.6000
B	0.00	0.00	1.00	0.1500	0.0600
W	1.00	1.00	1.00	0.3127	0.3290

Tabelle 12.5
Tristimuluswerte **R**, **G**, **B** und Weißpunkt **W** (D65) im sRGB-Farbraum.

Transformation CIEXYZ→sRGB

Zur Transformation XYZ→sRGB (Abb. 12.24) werden zunächst aus den CIE-Koordinaten X, Y, Z durch Multiplikation mit M_{RGB} entsprechend ITU-BT.709 [43] (Gl. 12.53) die *linearen* RGB-Werte R, G, B berechnet:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = M_{\text{RGB}} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 3.2406 & -1.5372 & -0.4986 \\ -0.9689 & 1.8758 & 0.0415 \\ 0.0557 & -0.2040 & 1.0570 \end{pmatrix} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad (12.54)$$

Anschließend erfolgt eine modifizierte Gammakorrektur (siehe Abschn. 5.6.6) mit $\gamma = 2.4$, entsprechend einem effektiven Gammawert von etwa 2.2, in der Form

$$\begin{aligned} R' = f_1(R), \quad G' = f_1(G), \quad B' = f_1(B), \\ \text{wobei} \quad f_1(c) = \begin{cases} 1.055 \cdot c^{\frac{1}{2.4}} - 0.055 & \text{wenn } c > 0.0031308 \\ 12.92 \cdot c & \text{wenn } c \le 0.0031308 \end{cases} \end{aligned} \quad (12.55)$$

Die resultierenden sRGB-Komponenten R', G', B' werden auf das Intervall $[0, 1]$ beschränkt (Tabelle 12.6 zeigt die entsprechenden Ergebnisse für ausgewählte Farbpunkte). Zur diskreten Darstellung werden die Werte anschließend linear auf den Bereich $[0, 255]$ skaliert und auf 8 Bit quantisiert.

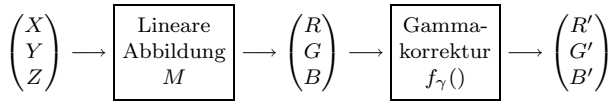


Abbildung 12.24
Transformation von Farbkoordinaten aus CIEXYZ nach sRGB.

Tabelle 12.6
CIEXYZ-Koordinaten und xy-Werte für ausgewählte Farbpunkte in sRGB. Die sRGB-Komponenten R', G', B' sind nichtlinear (d. h. gammakorrigiert), Referenzweißpunkt ist D65 (siehe Tabelle 12.3). Die xy-Werte in den beiden letzten Spalten (CIExy) beziehen sich jeweils auf den zugehörigen Y-Wert, d. h. jeder vollständige Farbwert ist durch Yxy spezifiziert. Dies wird etwa bei den unterschiedlichen Rottönen $\mathbf{R}$, $\mathbf{R}_{75}$, $\mathbf{R}_{50}$, $\mathbf{R}_{25}$ deutlich, die zwar identische xy-Werte aber unterschiedliche Y-Werte (Helligkeiten) aufweisen.

		sRGB			CIEXYZ			CIExy	
Pkt.	Farbe	R'	G'	B'	X	Y	Z	x	y
S	Schwarz	0.00	0.00	0.00	0.0000	0.0000	0.0000	0.3127	0.3290
R	Rot	1.00	0.00	0.00	0.4124	0.2126	0.0193	0.6400	0.3300
Y	Gelb	1.00	1.00	0.00	0.7700	0.9278	0.1385	0.4193	0.5053
G	Grün	0.00	1.00	0.00	0.3576	0.7152	0.1192	0.3000	0.6000
C	Cyan	0.00	1.00	1.00	0.5381	0.7874	1.0697	0.2247	0.3287
B	Blau	0.00	0.00	1.00	0.1805	0.0722	0.9505	0.1500	0.0600
M	Magenta	1.00	0.00	1.00	0.5929	0.2848	0.9698	0.3209	0.1542
W	Weiß	1.00	1.00	1.00	0.9505	1.0000	1.0890	0.3127	0.3290
K	50% Grau	0.50	0.50	0.50	0.2034	0.2140	0.2331	0.3127	0.3290
$\mathbf{R}_{75}$	75% Rot	0.75	0.00	0.00	0.2155	0.1111	0.0101	0.6401	0.3300
$\mathbf{R}_{50}$	50% Rot	0.50	0.00	0.00	0.0883	0.0455	0.0041	0.6401	0.3300
$\mathbf{R}_{25}$	25% Rot	0.25	0.00	0.00	0.0210	0.0108	0.0009	0.6401	0.3300
P	Pink	1.00	0.50	0.50	0.5276	0.3811	0.2483	0.4560	0.3295

Transformation sRGB→CIEXYZ

Zunächst werden die gegebenen (nichtlinearen) $R'G'B'$-Komponenten (im Intervall $[0,1]$) durch die Umkehrung der Gammakorrektur in Gl. 12.55 wieder linearisiert, d. h.

$$R = f_2(R'), \quad G = f_2(G'), \quad B = f_2(B'), \tag{12.56}$$

$$\text{wobei} \quad f_2(c) = \begin{cases} \left(\frac{c+0.055}{1.055}\right)^{2.4} & \text{wenn } c > 0.03928 \\ \frac{c}{12.92} & \text{wenn } c \le 0.03928 \end{cases} \tag{12.57}$$

Nachfolgend werden die linearen RGB-Koordinaten durch Multiplikation mit M_{RGB}^{-1} (Gl. 12.54) in den XYZ-Raum transformiert:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = M_{\text{RGB}}^{-1} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0193 & 0.1192 & 0.9505 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{12.58}$$

Rechnen mit sRGB-Werten

Durch den verbreiteten Einsatz von sRGB in der Digitalfotografie, im WWW, in Computerbetriebssystemen und in der Multimedia-Produktion kann man davon ausgehen, dass man es bei Vorliegen eines RGB-Farbbilds mit hoher Wahrscheinlichkeit mit einem sRGB-Bild zu tun hat. Öffnet man daher beispielsweise ein JPEG-Bild in ImageJ oder in Java, dann sind die im zugehörigen RGB-Array liegenden Pixelwerte darstellungsbezogene, also *nichtlineare* $R'G'B'$-Komponenten des sRGB-Farbraums. Dieser Umstand wird in der Programmierpraxis leider häufig vernachlässigt.

Bei arithmetischen Operationen mit den Farbkomponenten sollten grundsätzlich die *linearen* RGB-Werte verwendet werden, die man aus den $R'G'B'$-Werten über die Funktion f_2 (Gl. 12.57) erhält und über f_1 (Gl. 12.55) wieder zurückrechnen kann.

Beispiel: Grauwertkonvertierung

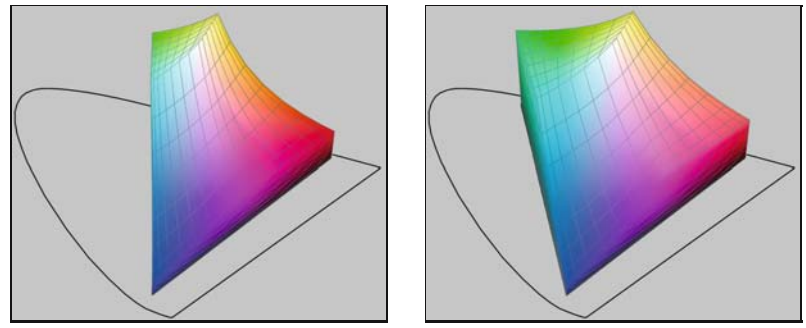
Bei der in Abschn. 12.2.1 beschriebenen Umrechnung von RGB- in Grauwertbilder (Gl. 12.7 auf S. 249) in der Form

$$Y = 0.2125 \cdot R + 0.7154 \cdot G + 0.072 \cdot B \tag{12.59}$$

sind mit R, G, B und Y explizit die *linearen* Werte gemeint. Die *exakte* Grauwertumrechnung mit sRGB-Farben wäre auf Basis von Gl. 12.59 demnach

$$Y' = f_1\big[0.2125 \cdot f_2(R') + 0.7154 \cdot f_2(G') + 0.0721 \cdot f_2(B')\big]. \tag{12.60}$$

In den meisten Fällen ist aber eine Annäherung *ohne* Umrechnung der sRGB-Komponenten (also direkt auf Basis der nichtlinearen $R'G'B'$-Werte) durch eine Linearkombination



(a) (b)

Abbildung 12.25
Gamut-Bereiche im CIEXYZ-Farbraum. Gamut für sRGB (a) und Adobe-RGB (b).

$$Y' \approx w'_R \cdot R' + w'_G \cdot G' + w'_B \cdot B' \tag{12.61}$$

mit leicht geänderten Koeffizienten w'_R, w'_G, w'_B ausreichend (z. B. mit $w'_R = 0.309$, $w'_G = 0.609$, $w'_B = 0.082$ [65]). Dass übrigens bei der Ersetzung eines sRGB-Farbpixels in der Form

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

überhaupt ein Grauwert (bzw. ein unbuntes Farbbild) entsteht, beruht auf dem Umstand, dass die Gammakorrektur (Gl. 12.55, 12.57) auf alle drei Farbkomponenten gleichermaßen angewandt wird und sich daher auch alle nichtlinearen sRGB-Farben mit drei identischen Komponentenwerten auf der Graugeraden im CIEXYZ-Farbraum bzw. am Weißpunkt **W** im xy-Diagramm befinden.

12.3.4 Adobe RGB

Ein Schwachpunkt von sRGB ist das relativ kleine Gamut, das sich praktisch auf die von einem üblichen Farbmonitor darstellbaren Farben beschränkt und besonders im Druckbereich häufig zu Problemen führt. Der von Adobe als eigener Standard „Adobe RGB (1998)“ [1] entwickelte Farbraum basiert auf dem gleichen Konzept wie sRGB, verfügt aber vor allem durch den gegenüber sRGB geänderten Primärfarbwert für Grün (mit $x = 0.21$, $y = 0.71$) über ein deutlich größeres Gamut (Abb. 12.22) und ist damit auch als RGB-Farbraum für den Druckbereich geeignet. Abb. 12.25 zeigt den deutlichen Unterschied der Gamut-Bereiche für sRGB und Adobe-RGB im dreidimensionalen CIEXYZ-Farbraum.

Der neutrale Farbwert von Adobe-RGB entspricht mit $x = 0.3127$, $y = 0.3290$ der Standardbeleuchtung D65, der Gammawert für die Abbildung von nichtlinearen $R'G'B'$-Werten zu linearen RGB-Werten ist 2.199 bzw. 1/2.199 für die umgekehrte Abbildung. Die zugehörige Dateispezifikation sieht eine Reihe verschiedener Kodierungen (8–16 Bit Integer sowie 32-Bit Float) für die Farbkomponenten vor. Adobe-RGB wird in Photoshop häufig als Alternative zum L*a*b*-Farbraum verwendet.

12.3.5 Farben und Farbräume in Java

sRGB-Werte in Java

sRGB ist der Standardfarbraum für Farbbilder in Java, d. h., die Komponenten von Farbobjekten sind bereits für die Ausgabe auf einem Monitor vorkorrigierte – also *nichtlineare* – $R'G'B'$-Werte (siehe Abb. 12.24). Der Zusammenhang zwischen den nichtlinearen Werten R', G', B' und den linearen Werten R, G, B (Gammakorrektur) entspricht dem sRGB-Standard, wie in Gl. 12.55 und 12.57 beschrieben.

Im Unterschied zu der in Abschn. 12.3.3 verwendeten Spezifikation beziehen sich in Java die XYZ-Koordinaten für den sRGB-Farbraum allerdings *nicht* auf den Weißpunkt D65, sondern auf den Weißpunkt D50 (mit $x = 0.3458$ und $y = 0.3585$), der üblicherweise für die Betrachtung von reflektierenden (gedruckten) Darstellungen vorgesehen ist. Die Primärfarben (Tristimuluswerte) **R**, **G**, **B** und der Weißpunkt **W** haben daher gegenüber dem sRGB-Standard (s. Tabelle 12.5) in Java-sRGB die in Tabelle 12.7 dargestellten RGB- bzw. xy-Koordinaten.

D50	R	G	B	x	y
R	1.00	0.00	0.00	0.6525	0.3252
G	0.00	1.00	0.00	0.3306	0.5944
B	0.00	0.00	1.00	0.1482	0.0774
W	1.00	1.00	1.00	0.3458	0.3585

Tabelle 12.7
Tristimuluswerte **R**, **G**, **B** und Weißpunkt **W** (D50) im Java-sRGB-Farbraum.

Die Umrechnung zwischen den auf den Weißpunkt D50 bezogenen XYZ-Koordinaten (X_{50}, Y_{50}, Z_{50}) und den D65-bezogenen, linearen RGB-Werten (R_{65}, G_{65}, B_{65}) bedingt gegenüber Gl. 12.54 bzw. Gl. 12.58 abweichende Abbildungsmatrizen, die sich aus der chromatischen Adaptierung $M_{65|50}$ (Gl. 12.48) und der XYZ→RGB-Transformation M_{RGB} (Gl. 12.54) zusammensetzen als

$$\begin{pmatrix} R_{65} \\ G_{65} \\ B_{65} \end{pmatrix} = M_{\text{RGB}} \cdot M_{65|50} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} \tag{12.62}$$

$$= \begin{pmatrix} 3.1339 & -1.6170 & -0.4906 \\ -0.9785 & 1.9160 & 0.0333 \\ 0.0720 & -0.2290 & 1.4057 \end{pmatrix} \cdot \begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} \tag{12.63}$$

beziehungsweise in der umgekehrten Richtung

$$\begin{pmatrix} X_{50} \\ Y_{50} \\ Z_{50} \end{pmatrix} = \left(M_{\mathrm{RGB}} \cdot M_{65|50} \right)^{-1} \cdot \begin{pmatrix} R_{65} \\ G_{65} \\ B_{65} \end{pmatrix} \tag{12.64}$$

$$= M_{50|65} \cdot M_{\mathrm{RGB}}^{-1} \cdot \begin{pmatrix} R_{65} \\ G_{65} \\ B_{65} \end{pmatrix} \tag{12.65}$$

$$= \begin{pmatrix} 0.4360 & 0.3851 & 0.1431 \\ 0.2224 & 0.7169 & 0.0606 \\ 0.0139 & 0.0971 & 0.7142 \end{pmatrix} \cdot \begin{pmatrix} R_{65} \\ G_{65} \\ B_{65} \end{pmatrix} \tag{12.66}$$

Java-Klassen

Für das Arbeiten mit Farbbildern und Farben bietet das Java-API bereits einiges an Unterstützung. Die wichtigsten Klassen sind

- `ColorModel`: zur Beschreibung der Struktur von Farbbildern, z. B. Vollfarbenbilder oder Indexbilder, wie in Abschn. 12.1.2 verwendet (Prog. 12.3).
- `Color`: zur Definition einzelner Farbobjekte.
- `ColorSpace`: zur Definition von Farbräumen.

`Color (java.awt.Color)`

Ein Objekt der Klasse `Color` dient zur Beschreibung einer bestimmten Farbe in einem zugehörigen Farbraum. Es enthält die durch den Farbraum definierten Farbkomponenten. Sofern der Farbraum nicht explizit vorgegeben ist, werden neue `Color`-Objekte als sRGB-Farben angelegt, wie folgendes Beispiel zeigt:

```
1 float r = 1.0f, g = 0.5f, b = 0.5f;
2 int R = 0, G = 0, B = 255;
3 Color pink = new Color(r, g, b); //float components [0..1]
4 Color blue = new Color(R, G, B); //int components [0..255]
```

`Color`-Objekte werden vor allem für grafische Operationen verwendet, wie etwa zur Spezifikation von Strich- oder Füllfarben. Daneben bietet `Color` zwei nützliche Klassenmethoden `RGBtoHSB()` und `HSBtoRGB()` zu Umwandlung von sRGB in den HSV-Farbraum[18] (Abschn. 12.2.3 ab S. 255).

`ColorSpace (java.awt.color.ColorSpace)`

Ein Objekt der Klasse `ColorSpace` repräsentiert einen Farbraum wie beispielsweise sRGB oder CMYK. Jeder Farbraum stellt Methoden zur Konvertierung von Farben in den sRGB- und CIEXYZ-Farbraum (und umgekehrt) zur Verfügung, sodass insbesondere über CIEXYZ Transformationen zwischen beliebigen Farbräumen möglich sind. In folgendem Beispiel wird eine Instanz des Standardfarbraums sRGB angelegt

[18] Im Java-API wird die Bezeichnung „HSB“ für den HSV-Farbraum verwendet.

und anschließend ein sRGB-Farbwert (R', B', G') in die entsprechenden Farbkoordinaten (X, Y, Z) im CIEXYZ-Farbraum konvertiert:[19]

```
ColorSpace sRGBsp
    = ColorSpace.getInstance(ColorSpace.CS_sRGB);
float[] pink_RGB = new float[] {1.0f, 0.5f, 0.5f};
float[] pink_XYZ = sRGBsp.toCIEXYZ(pink_RGB);
```

Neben dem Standardfarbraum sRGB stehen durch die im obigen Beispiel verwendete Methode `ColorSpace.getInstance()` folgende weitere Farbräume zur Verfügung: CIEXYZ (`CS_CIEXYZ`), lineare RGB-Grauwerte ohne Gammakorrektur (`CS_LINEAR_RGB`), 8-Bit-Grauwerte (`CS_GRAY`) und der YCC-Farbraum von Kodak (`CS_PYCC`).

Zusätzliche Farbräume können durch Erweiterung der Klasse `ColorSpace` definiert werden, wie anhand der Realisierung des L*a*b*-Farbraums durch die Klasse `Lab_ColorSpace` in Prog. 12.10–12.11 gezeigt ist. Die Konvertierungsmethoden entsprechen der Beschreibung in Abschn. 12.3.2. Die abstrakte Klasse `ColorSpace` erfordert neben den Methoden `fromCIEXYZ()` und `toCIEXYZ()` auch die Implementierung der Konvertierungen in den sRGB-Farbraum in Form der Methoden `fromRGB()` bzw. `toRGB()`. Diese Konvertierungen werden durch Umrechnung über CIEXYZ durchgeführt (Prog. 12.11).[20] Folgendes Beispiel zeigt die Verwendung der so konstruierten Klasse `Lab_ColorSpace`:

```
ColorSpace LABcs = new Lab_ColorSpace();
float[] pink_XYZ1 = {0.5276f, 0.3811f, 0.2483f};
// XYZ → LAB:
float[] pink_LAB1 = LABcs.fromCIEXYZ(pink_XYZ);
// LAB → XYZ:
float[] pink_XYZ2 = LABcs.toCIEXYZ(pink_XYZ);
```

ICC-Profile

Auch bei genauester Spezifikation reichen Farbräume zur präzisen Beschreibung des Abbildungsverhaltens konkreter Aufnahme- und Wiedergabegeräte nicht aus. ICC[21]-Profile sind standardisierte Beschreibungen dieses Abbildungsverhaltens und ermöglichen, dass ein zugehöriges Bild später von anderen Geräten exakt reproduziert werden kann. Profile sind damit ein wichtiges Instrument im Rahmen des digitalen Farbmanagements [86].

[19] Seltsamerweise sind die Ergebnisse der Java-Standardmethoden `toCIEXYZ()` und `fromCIEXYZ()` in den aktuellen API-Versionen zueinander nicht invers. Hier liegt ein (seit längerem bekannter) Java-Bug vor.

[20] Dabei muss der Bezug auf den richtigen Weißpunkt (D50 bzw. D65) beachtet werden.

[21] International Color Consortium ICC (www.color.org).

Programm 12.10
Implementierung der Klasse `Lab_ColorSpace` zur Repräsentation des L*a*b*-Farbraums. Die Konvertierung von CIEXYZ in den L*a*b*-Farbraum (Gl. 12.50) ist durch die Methode `fromCIEXYZ()` und die zugehörige Hilfsfunktion `f1()` realisiert.

```
1  import java.awt.color.ColorSpace;
2
3  public class Lab_ColorSpace extends ColorSpace {
4
5    // D65 reference illuminant coordinates:
6    private final double Xref = 0.95047;
7    private final double Yref = 1.00000;
8    private final double Zref = 1.08883;
9
10   protected Lab_ColorSpace(int type, int numcomponents) {
11     super(type, numcomponents);
12   }
13
14   public Lab_ColorSpace(){
15     super(TYPE_Lab,3);
16   }
17
18   // XYZ → CIELab
19   public float[] fromCIEXYZ (float[] XYZ) {
20     double xx = f1(XYZ[0] / Xref);
21     double yy = f1(XYZ[1] / Yref);
22     double zz = f1(XYZ[2] / Zref);
23
24     float L = (float)(116 * yy - 16);
25     float a = (float)(500 * (xx - yy));
26     float b = (float)(200 * (yy - zz));
27     return new float[] {L,a,b};
28   }
29
30   double f1 (double c) {
31     if (c > 0.008856)
32       return Math.pow(c, 1.0 / 3);
33     else
34       return (7.787 * c) + (16.0 / 116);
35   }
```

Das Java-2D-API unterstützt den Einsatz von ICC-Profilen durch die Klassen `ICC_ColorSpace` und `ICC_Profile`, die es erlauben, verschiedene Standardprofile zu generieren und ICC-Profildateien zu lesen.

Nehmen wir beispielsweise an, ein Bild, das mit einem kalibrierten Scanner aufgenommen wurde, soll möglichst originalgetreu auf einem Monitor dargestellt werden. In diesem Fall benötigen wir zunächst die ICC-Profile für den Scanner und den Monitor, die in der Regel als `.icc`-Dateien zur Verfügung stehen. Für Standardfarbräume sind die entsprechenden Profile häufig bereits im Betriebssystem des Computers vorhanden, wie z. B. `CIERGB.icc` oder `NTSC1953.icc`.

```
36 // class Lab_ColorSpace (continued)
37
38   // CIELab → XYZ
39   public float[] toCIEXYZ(float[] Lab) {
40     double yy = ( Lab[0] + 16 ) / 116;
41     float X = (float) (Xref * f2(Lab[1] / 500 + yy));
42     float Y = (float) (Yref * f2(yy));
43     float Z = (float) (Zref * f2(yy - Lab[2] / 200));
44     return new float[] {X,Y,Z};
45   }
46
47   double f2 (double c) {
48     double c3 = Math.pow(c, 3.0);
49     if (c3 > 0.008856)
50       return c3;
51     else
52       return (c - 16.0 / 116) / 7.787;
53   }
54
55   // sRGB → CIELab
56   public float[] fromRGB(float[] sRGB) {
57     ColorSpace sRGBcs = ColorSpace.getInstance(CS_sRGB);
58     float[] XYZ = sRGBcs.toCIEXYZ(sRGB);
59     return this.fromCIEXYZ(XYZ);
60   }
61
62   // CIELab → sRGB
63   public float[] toRGB(float[] Lab) {
64     float[] XYZ = this.toCIEXYZ(Lab);
65     ColorSpace sRGBcs = ColorSpace.getInstance(CS_CIEXYZ);
66     return sRGBcs.fromCIEXYZ(XYZ);
67   }
68
69 } // end of class Lab_ColorSpace
```

Programm 12.11
Implementierung der Klasse `Lab_ColorSpace` zur Repräsentation des L*a*b*-Farbraums (*Fortsetzung*). Die Konvertierung vom L*a*b*-Farbraum nach CIEXYZ (Gl. 12.51) ist durch die Methode `toCIEXYZ()` und die zugehörige Hilfsfunktion `f2()` realisiert. Die Methoden `fromRGB()` und `toRGB()` führen die Konvertierung von L*a*b* nach sRGB in 2 Schritten über den CIEXYZ-Farbraum durch.

Mit diesen Profildaten kann ein Farbraumobjekt erzeugt werden, mit dem aus den Bilddaten des Scanners entsprechende Farbwerte in CIEXYZ oder sRGB umgerechnet werden, wie folgendes Beispiel zeigt:

```
1 ICC_ColorSpace scannerCS =
2   new ICC_ColorSpace(ICC_ProfileRGB.getInstance("scanner.icc"
      ));
3 float[] RGBColor = scannerCS.toRGB(scannerColor);
4 float[] XYZColor = scannerCS.toCIEXYZ(scannerColor);
```

Genauso kann natürlich über den durch das ICC-Profil definierten Farbraum ein sRGB-Pixel in den Farbraum des Scanners oder des Monitors umgerechnet werden.

Programm 12.12
Zählen der Farben in einem RGB-Bild. Die Methode `countColors()` erzeugt zunächst eine Kopie des RGB-Pixel-Arrays (Zeile 3), sortiert dieses Array (Zeile 4) und zählt anschließend die Übergänge zwischen unterschiedlichen Farben.

```
1  static int countColors (ColorProcessor cp) {
2    // duplicate pixel array and sort
3    int[] pixels = ((int[]) cp.getPixels()).clone();
4    Arrays.sort(pixels); // requires java.util.Arrays
5
6    int k = 1; // image contains at least one color
7    for (int i = 0; i < pixels.length-1; i++) {
8      if (pixels[i] != pixels[i+1])
9        k = k + 1;
10   }
11   return k;
12 }
```

12.4 Statistiken von Farbbildern

12.4.1 Wie viele Farben enthält ein Bild?

Ein kleines aber häufiges Teilproblem im Zusammenhang mit Farbbildern besteht darin, zu ermitteln, wie viele unterschiedliche Farben in einem Bild überhaupt enthalten sind. Natürlich könnte man dafür ein Histogramm-Array mit einem Integer-Element für jede Farbe anlegen, dieses befüllen und anschließend abzählen, wie viele Histogrammzellen mindestens den Wert 1 enthalten. Da ein 24-Bit-RGB-Farbbild potenziell $2^{24} = 16.777.216$ Farbwerte enthalten kann, wäre ein solches Histogramm-Array (mit immerhin 64 MByte) in den meisten Fällen aber wesentlich größer als das ursprüngliche Bild selbst!

Eine einfachere Lösung besteht darin, die Farbwerte im Pixel-Array des Bilds zu *sortieren*, sodass alle gleichen Farbwerte beisammen liegen. Die Sortierreihenfolge ist dabei natürlich unwesentlich. Die Zahl der zusammenhängenden Farbblöcke entspricht der Anzahl der Farben im Bild. Diese kann, wie in Prog. 12.12 gezeigt, einfach durch Abzählen der Übergänge zwischen den Farbblöcken berechnet werden. Natürlich wird in diesem Fall nicht das ursprüngliche Pixel-Array sortiert (das würde das Bild verändern), sondern eine Kopie des Pixel-Arrays, die mit der Java-Standardmethode `clone()` erzeugt wird.[22] Das Sortieren erfolgt in Prog. 12.12 (Zeile 4) mithilfe der Java-Systemmethode `Arrays.sort()`, die sehr effizient implementiert ist.

12.4.2 Histogramme

Histogramme von Farbbildern waren bereits in Abschn. 4.5 ein Thema, wobei wir uns auf die eindimensionalen Verteilungen der einzelnen Farbkanäle bzw. der Intensitätswerte beschränkt haben. Auch die ImageJ-Methode `getHistogram()` berechnet bei Anwendung auf Objekte der Klasse `ColorProcessor` in der Form

[22] Die Java-Klasse `Array` implementiert das `Cloneable`-Interface.

```
ColorProcessor cp;
int[] H = cp.getHistogram();
```

lediglich das Histogramm der umgerechneten Grauwerte. Alternativ könnte man die Intensitätshistogramme der einzelnen Farbkomponenten berechnen, wobei allerdings (wie in Abschn. 4.5.2 beschrieben) keinerlei Information über die tatsächlichen Farbwerte zu gewinnen ist. In ähnlicher Weise könnte man natürlich auch die Verteilung der Komponenten für jeden anderen Farbraum (z. B. HSV oder $L^*a^*b^*$) darstellen.

Ein *volles* Histogramm des RGB-Farbraums wäre dreidimensional und enthielte, wie oben erwähnt, $256 \times 256 \times 256 = 2^{24}$ Zellen vom Typ `int`. Ein solches Histogramm wäre nicht nur groß, sondern auch schwierig zu visualisieren und brächte – im statistischen Sinn – auch keine zusammenfassende Information über das zugehörige Bild.[23]

2D-Farbhistogramme

Eine sinnvolle Darstellungsform sind hingegen zweidimensionale Projektionen des vollen RGB-Histogramms (Abb. 12.26). Je nach Projektionsrichtung ergibt sich dabei ein Histogramm mit den Koordinatenachsen Rot-Grün (H_{RG}), Rot-Blau (H_{RB}) oder Grün-Blau (H_{GB}) mit den Werten

$$\begin{aligned} H_{\text{RG}}(r,g) &\leftarrow \text{Anzahl der Pixel mit } I_{\text{RGB}}(u,v) = (r,g,*), \\ H_{\text{RB}}(r,b) &\leftarrow \text{Anzahl der Pixel mit } I_{\text{RGB}}(u,v) = (r,*,b), \\ H_{\text{GB}}(r,b) &\leftarrow \text{Anzahl der Pixel mit } I_{\text{RGB}}(u,v) = (*,g,b), \end{aligned} \tag{12.67}$$

wobei $*$ für einen beliebigen Komponentenwert steht. Das Ergebnis ist, unabhängig von der Größe des RGB-Farbbilds I_{RGB}, jeweils ein zweidimensionales Histogramm der Größe 256×256 (für 8-Bit RGB-Komponenten), das einfach als Bild dargestellt werden kann. Die Berechnung des vollen RGB-Histogramms ist natürlich zur Erstellung der kombinierten Farbhistogramme nicht erforderlich (siehe Prog. 12.13).

Wie die Beispiele in Abb. 12.27 zeigen, kommen in den kombinierten Farbhistogrammen charakteristische Farbeigenschaften eines Bilds zum Ausdruck, die zwar das Bild nicht eindeutig beschreiben, jedoch in vielen Fällen Rückschlüsse auf die Art der Szene oder die grobe Ähnlichkeit zu anderen Bildern ermöglichen (s. auch Aufg. 12.8).

12.5 Farbquantisierung

Das Problem der Farbquantisierung besteht in der Auswahl einer beschränkten Menge von Farben zur möglichst getreuen Darstellung eines

[23] Paradoxerweise ist trotz der wesentlich größeren Datenmenge des Histogramms aus diesem das ursprüngliche Bild dennoch nicht mehr rekonstruierbar.

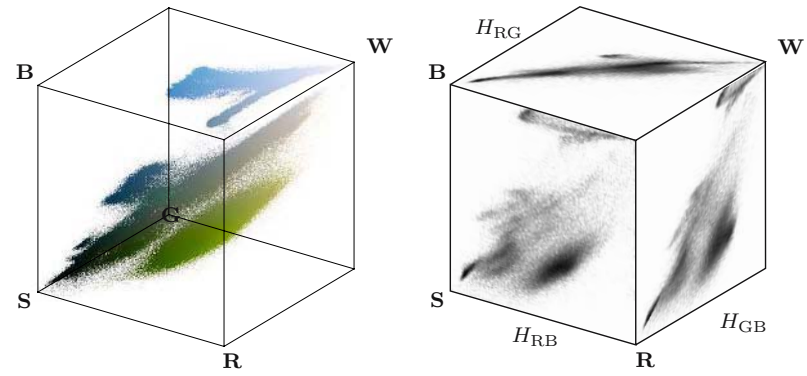


Abbildung 12.26
Projektionen des RGB-Histogramms. RGB-Farbwürfel mit Verteilung der Bildfarben (a). Die kombinierten Histogramme für *Rot-Grün* (H_{RG}), *Rot-Blau* (H_{RB}) und *Grün-Blau* (H_{GB}) sind zweidimensionale Projektionen des dreidimensionalen Histogramms (b). Originalbild siehe Abb. 12.9 (a).

Programm 12.13
Methode `get2dHistogram()` zur Berechnung eines kombinierten Farbhistogramms. Die gewünschten Farbkomponenten können über die Parameter `c1` und `c2` ausgewählt werden. Die Methode liefert die Histogrammwerte als zweidimensionales `int`-Array.

```
1  static int[][] get2dHistogram (ColorProcessor cp, int c1, int
       c2)
2  { // c1, c2: R = 0, G = 1, B = 2
3    int[] RGB = new int[3];
4    int[][] H = new int[256][256]; // histogram array H[c1][c2]
5
6    for (int v = 0; v < cp.getHeight(); v++) {
7      for (int u = 0; u < cp.getWidth(); u++) {
8        cp.getPixel(u, v, RGB);
9        int i = RGB[c1];
10       int j = RGB[c2];
11       // increment corresponding histogram cell
12       H[j][i]++; // i runs horizontal, j runs vertical
13     }
14   }
15   return H;
16 }
```

ursprünglichen Farbbilds. Stellen Sie sich vor, Sie wären ein Künstler und hätten gerade mit 150 unterschiedlichen Farbstiften eine Illustration mit den wunderbarsten Farbübergängen geschaffen. Einem Verleger gefällt Ihre Arbeit, er wünscht aber, dass Sie das Bild nochmals zeichnen, diesmal mit nur 10 verschiedenen Farben. Die (in diesem Fall vermutlich schwierige) Auswahl der 10 am besten geeigneten Farbstifte aus den ursprünglichen 150 ist ein Beispiel für Farbquantisierung.

Im allgemeinen Fall enthält das ursprüngliche Bild I eine Menge von m unterschiedlichen Farben $\mathcal{C} = \{\mathbf{C}_1, \mathbf{C}_2, \ldots \mathbf{C}_m\}$. Das können einige wenige sein oder viele Tausende, maximal aber 2^{24} bei einem 3×8-Bit-Farbbild. Die Aufgabe besteht darin, die ursprünglichen Farben durch eine (meist deutlich kleinere) Menge von Farben $\mathcal{C}' = \{\mathbf{C}'_1, \mathbf{C}'_2, \ldots \mathbf{C}'_n\}$ (mit $n < m$) zu ersetzen. Das Hauptproblem ist dabei die Auswahl einer reduzierten Farbpalette $\mathcal{C}'$, die das Bild möglichst wenig beeinträchtigt.

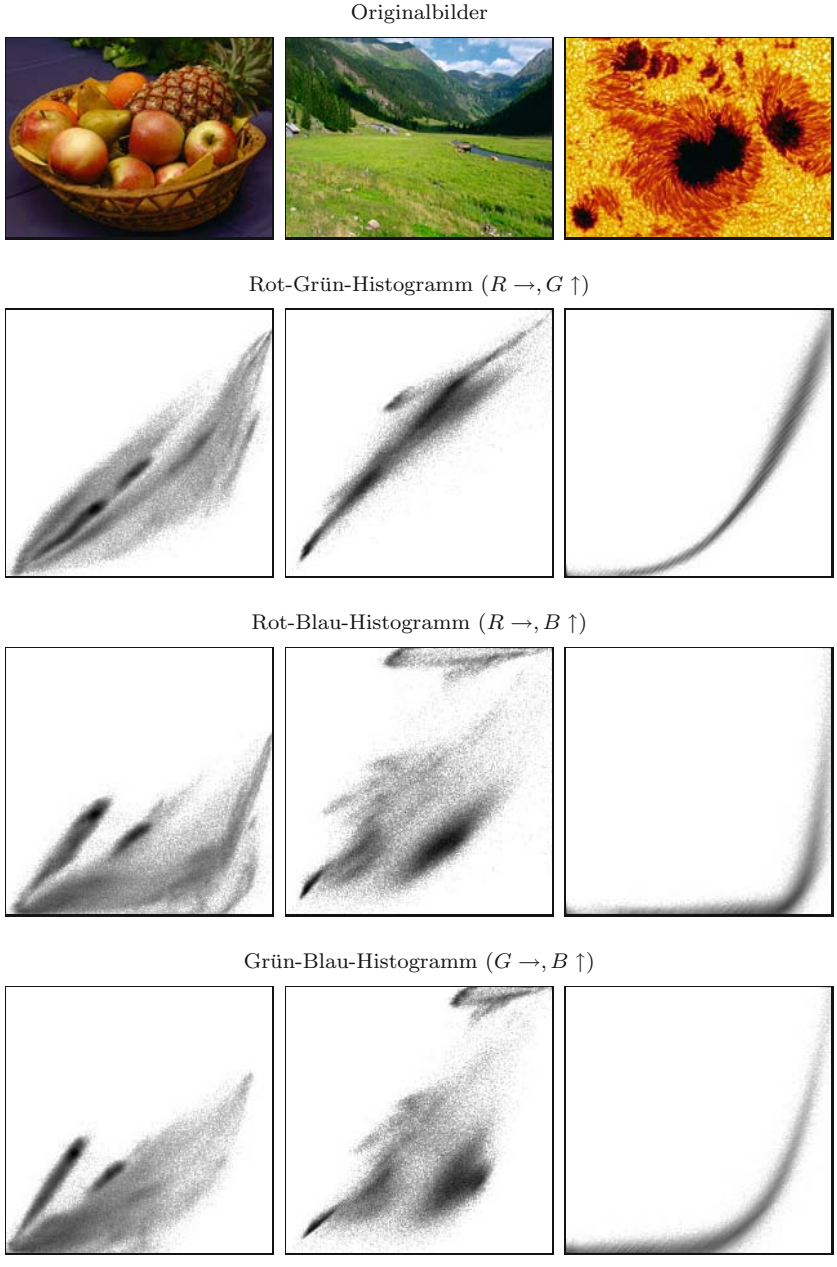


Abbildung 12.27
Beispiele für 2D-Farbhistogramme. Die Bilder sind zur besseren Darstellung invertiert (dunkle Bildstellen bedeuten hohe Häufigkeiten) und der Grauwert entspricht dem Logarithmus der Histogrammwerte, skaliert auf den jeweiligen Maximalwert.

In der Praxis tritt dieses Problem z. B. bei der Konvertierung von Vollfarbenbildern in Bilder mit kleinerer Pixeltiefe oder in Indexbilder auf, etwa beim Übergang von einem 24-Bit-Bild im TIFF-Format in ein 8-Bit-GIF-Bild mit nur 256 Farben. Ein ähnliches Problem gab es bis vor wenigen Jahren auch bei der Darstellung von Vollfarbenbildern auf Computerbildschirmen, da die verfügbare Grafik-Hardware aus Kostengründen oft auf nur 8 Bitebenen beschränkt war. Heute verfügen auch billige Grafikkarten über 24-Bit-Tiefe, das Problem der (schnellen) Farbquantisierung besteht hier also kaum mehr.

12.5.1 Skalare Farbquantisierung

Die *skalare* (oder *uniforme*) Quantisierung ist ein einfaches und schnelles Verfahren, das den Bildinhalt selbst nicht berücksichtigt. Jede der ursprünglichen Farbkomponenten c_i (z. B. R_i, G_i, B_i) im Wertebereich $[0 \ldots m{-}1]$ wird dabei unabhängig in den neuen Wertebereich $[0 \ldots n{-}1]$ überführt, im einfachsten Fall durch eine lineare Quantisierung in der Form

$$c_i' \leftarrow \left\lfloor c_i \cdot \frac{n}{m} \right\rfloor \tag{12.68}$$

für alle Farbkomponenten c_i. Ein typisches Beispiel ist die Konvertierung eines Farbbilds mit 3×12-Bit-Komponenten mit $m = 4096$ möglichen Werten (z. B. aus einem Scanner) in ein herkömmliches RGB-Farbbild mit 3×8-Bit-Komponenten, also jeweils $n = 256$ Werten. Jeder Komponentenwert wird daher durch $4096/256 = 16 = 2^4$ ganzzahlig dividiert oder, anders ausgedrückt, die untersten 4 Bits der zugehörigen Binärzahl werden einfach ignoriert (Abb. 12.28 (a)).

Ein (heute allerdings kaum mehr praktizierter) Extremfall ist die in Abb. 12.28 (b) gezeigte Quantisierung von 3×8-Bit-Farbwerten in nur *ein* Byte, wobei 3 Bits für Rot und Grün und 2 Bits für Blau verwendet werden. Die Umrechnung in derart gepackte 3:3:2-Pixel kann mit Bit-

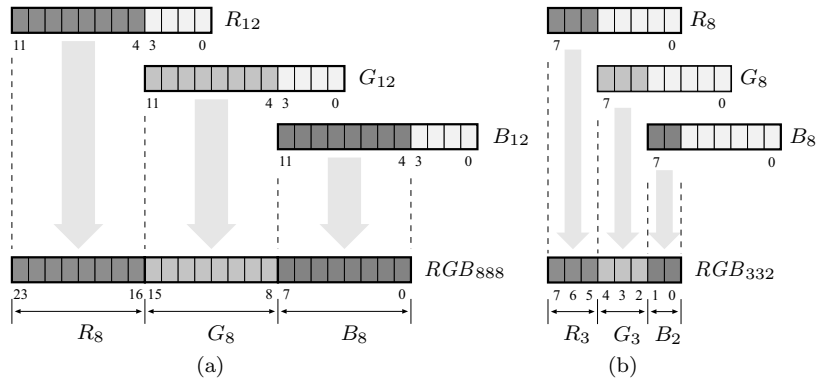
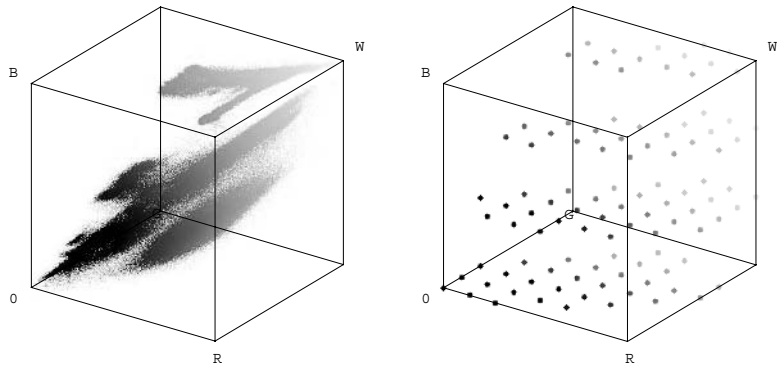


Abbildung 12.28
Skalare Quantisierung von Farbkomponenten durch Abtrennen niederwertiger Bits. Quantisierung von 3×12-Bit- auf 3×8-Bit-Farben (a). Quantisierung von 3×8-Bit auf 8-Bit-Farben (3:3:2) (b). Das Java-Codestück zeigt die notwendigen Bitoperationen für die Umrechnung von 8-Bit-RGB-Werten in 3:3:2-gepackte Farbpixel.



(a) (b)

Abbildung 12.29
Farbverteilung nach einer skalaren 3:3:2-Quantisierung. Ursprüngliche Verteilung der 226.321 Farben im RGB-Würfel (a). Verteilung der resultierenden $8 \times 8 \times 4 = 256$ Farben nach der 3:3:2-Quantisierung (b).

```
1 ColorProcessor cp = (ColorProcessor) ip;
2 int C = cp.getPixel(u, v);
3 int R = (C & 0x00ff0000) >> 16;
4 int G = (C & 0x0000ff00) >> 8;
5 int B = (C & 0x000000ff);
6 // 3:3:2 uniform color quantization
7 byte RGB =
8     (byte) ((R & 0xE0) | (G & 0xE0)>>3 | ((B & 0xC0)>>6));
```

Programm 12.14
Quantisierung eines 3×8-Bit RGB-Farbpixels auf 8 Bit in 3:3:2-Packung.

operationen in Java effizient durchgeführt werden, wie das Codesegment in Prog. 12.14 zeigt. Die resultierende Bildqualität ist wegen der kleinen Zahl von Farbabstufungen natürlich gering (Abb. 12.29).

Im Unterschied zu den nachfolgend gezeigten Verfahren nimmt die skalare Quantisierung keine Rücksicht auf die Verteilung der Farben im ursprünglichen Bild. Die skalare Quantisierung wäre ideal für den Fall, dass die Farben im RGB-Würfel gleichverteilt sind. Bei natürlichen Bildern ist jedoch die Farbverteilung in der Regel höchst ungleichförmig, sodass einzelne Regionen des Farbraums dicht besetzt sind, während andere Farben im Bild überhaupt nicht vorkommen. Der durch die skalare Quantisierung erzeugte Farbraum kann zwar auch die nicht vorhandenen Farben repräsentieren, dafür aber die Farben in dichteren Bereichen nicht fein genug abstufen.

12.5.2 Vektorquantisierung

Bei der Vektorquantisierung werden im Unterschied zur skalaren Quantisierung nicht die einzelnen Farbkomponenten getrennt betrachtet, sondern jeder im Bild enthaltene Farbvektor $\mathbf{C}_i = (r_i, g_i, b_i)$ als Ganzes. Das Problem der Vektorquantisierung ist, ausgehend von der Menge der ursprünglichen Farbwerte $\mathcal{C} = \{\mathbf{C}_1, \mathbf{C}_2, \ldots \mathbf{C}_m\}$,

a) eine Menge n repräsentativer Farbvektoren $\mathcal{C}' = \{\mathbf{C}'_1, \mathbf{C}'_2, \ldots \mathbf{C}'_n\}$ zu finden und

b) jeden der ursprünglichen Farbwerte $\mathbf{C}_i$ durch einen der neuen Farbvektoren $\mathbf{C}'_j$ zu ersetzen,

wobei n meist vorgegeben ist und die resultierende Abweichung gegenüber dem Originalbild möglichst gering sein soll. Dies ist allerdings ein kombinatorisches Optimierungsproblem mit einem ziemlich großen Suchraum, der durch die Zahl der möglichen Farbvektoren und Farbzuordnungen bestimmt ist. Im Allgemeinen kommt daher die Suche nach einem *globalen* Optimum aus Zeitgründen nicht in Frage und alle nachfolgend beschriebenen Verfahren berechnen lediglich ein „lokales" Optimum.

Populosity-Algorithmus

Der Populosity-Algorithmus[24] [35] verwendet die n häufigsten Farbwerte eines Bilds als repräsentative Farbvektoren $\mathcal{C}'$. Das Verfahren ist einfach zu implementieren und wird daher häufig verwendet. Die Ermittlung der n häufigsten Farbwerte ist über die in Abschn. 12.4.1 gezeigte Methode möglich. Die ursprünglichen Farbwerte $\mathbf{C}_i$ werden dem jeweils nächstliegenden Repräsentanten in $\mathcal{C}'$ zugeordnet, also jenem quantisierten Farbwert mit dem geringsten Abstand im 3D-Farbraum.

Das Verfahren arbeitet allerdings nur dann zufrieden stellend, solange die Farbwerte des Bilds nicht über einen großen Bereich verstreut sind. Durch vorherige Gruppierung ähnlicher Farben in größere Zellen (durch skalare Quantisierung) ist eine gewisse Verbesserung möglich. Allerdings gehen seltenere Farben – die für den Bildinhalt aber wichtig sein können – immer dann verloren, wenn sie nicht zu einer der n häufigsten Farben ähnlich sind.

Median-Cut-Algoritmus

Der Median-Cut-Algorithmus [35] gilt als klassisches Verfahren zur Farbquantisierung und ist in vielen Programmen (u. a. auch in ImageJ) implementiert. Wie im Populosity-Algorithmus wird zunächst ein Histogramm der ursprünglichen Farbverteilung berechnet, allerdings mit einer reduzierten Zahl von Histogrammzellen, z. B. $32 \times 32 \times 32$. Dieser Histogrammwürfel wird anschließend rekursiv in immer kleinere Quader zerteilt, bis die erforderliche Anzahl von Farben (n) erreicht ist. In jedem Schritt wird jener Quader ausgewählt, der zu diesem Zeitpunkt die meisten Bildpunkte enthält. Die Teilung des Quaders erfolgt quer zur längsten seiner drei Achsen, sodass in den restlichen Hälften gleich viele Bildpunkte verbleiben, also am Medianpunkt entlang dieser Achse (Abb. 12.30).

[24] Manchmal auch als „Popularity"-Algorithmus bezeichnet.

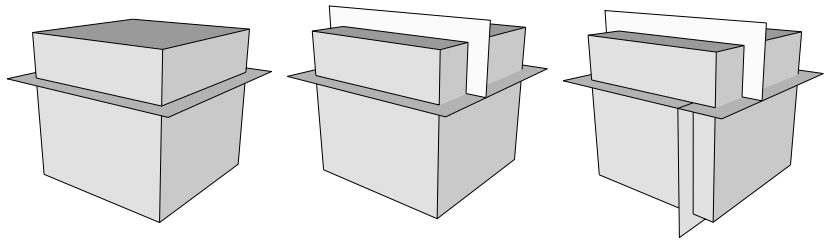


Abbildung 12.30
Median-Cut-Algorithmus. Der RGB-Farbraum wird schrittweise in immer kleinere Quader quer zu einer der Farbachsen geteilt.

Das Ergebnis am Ende dieses rekursiven Teilungsvorgangs sind n Quader im Farbraum, die idealerweise jeweils dieselbe Zahl von Bildpunkten enthalten. Als letzten Schritt wird für jeden Quader ein repräsentativer Farbvektor (z. B. der arithmetische Mittelwert der enthaltenen Farbpunkte) berechnet und alle zugehörigen Bildpunkte durch diesen Farbwert ersetzt.

Der Vorteil dieser Methode ist, dass Farbregionen mit hoher Dichte in viele kleinere Zellen zerlegt werden und dadurch die resultierenden Farbfehler gering sind. In Bereichen des Farbraums mit niedriger Dichte können jedoch relativ große Quader und somit auch große Farbabweichungen entstehen.

Octree-Algorithmus

Ähnlich wie der Median-Cut-Algorithmus basiert auch dieses Verfahren auf der Partitionierung des dreidimensionalen Farbraums in Zellen unterschiedlicher Größe. Der Octree-Algorithmus [28] verwendet allerdings eine hierarchische Struktur, in der jeder Quader im 3D-Raum wiederum aus 8 Teilquadern bestehen kann. Diese Partitionierung wird als Baumstruktur (Octree) repräsentiert, in der jeder Knoten einem Quader entspricht, der wieder Ausgangspunkt für bis zu 8 weitere Knoten sein kann. Jedem Knoten ist also ein Teil des Farbraums zugeordnet, der sich auf einer bestimmten Baumtiefe d (bei einem 3×8-Bit-RGB-Bild auf Tiefe $d = 8$) auf einen einzelnen Farbwert reduziert.

Zur Verarbeitung eines RGB-Vollfarbenbilds werden die Bildpunkte sequentiell durchlaufen und dabei der zugehörige Quantisierungsbaum dynamisch aufgebaut. Der Farbwert jedes Bildpixels wird in den Quantisierungsbaum eingefügt, wobei die Anzahl der Endknoten auf K (üblicherweise $K = 256$) beschränkt ist. Beim Einfügen eines neuen Farbwerts $\mathbf{C}_i$ kann einer von zwei Fällen auftreten:

1. Wenn die Anzahl der Knoten noch geringer ist als K und kein passender Knoten für den Farbwert $\mathbf{C}_i$ existiert, dann wird ein neuer Knoten für $\mathbf{C}_i$ angelegt.
2. Wenn die Anzahl der Knoten bereits K beträgt und die Farbe $\mathbf{C}_i$ noch nicht repräsentiert ist, dann werden bestehende Farbknoten auf der höchsten Baumtiefe (sie repräsentieren nahe aneinander liegende Farben) zu einem gemeinsamen Knoten reduziert.

Ein Vorteil des iterativen Octree-Verfahrens ist, dass die Anzahl der Farbknoten zu jedem Zeitpunkt auf K beschränkt und damit der Speicheraufwand gering ist. Auch die abschließende Zuordnung und Ersetzung der Bildfarben zu den repräsentativen Farbvektoren kann mit der Octree-Struktur besonders einfach und effizient durchgeführt werden, da für jeden Farbwert maximal 8 Suchschritte durch die Ebenen des Baums zur Bestimmung des zugehörigen Knotens notwendig sind.

Abb. 12.31 zeigt die unterschiedlichen Farbverteilungen im RGB-Farbraum nach Anwendung des Median-Cut- und des Octree-Algorithmus. In beiden Fällen wurde das Originalbild (Abb. 12.27 (b)) auf 256

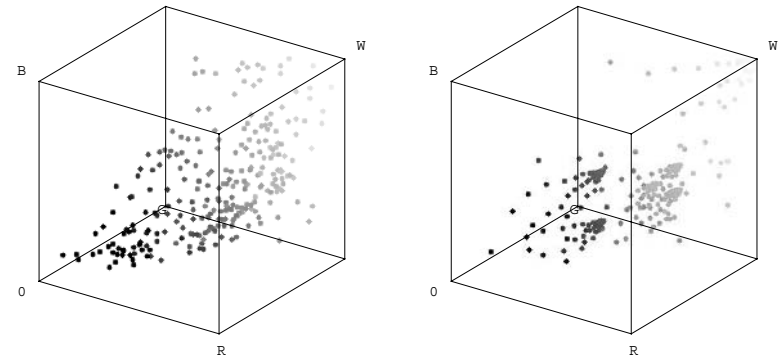


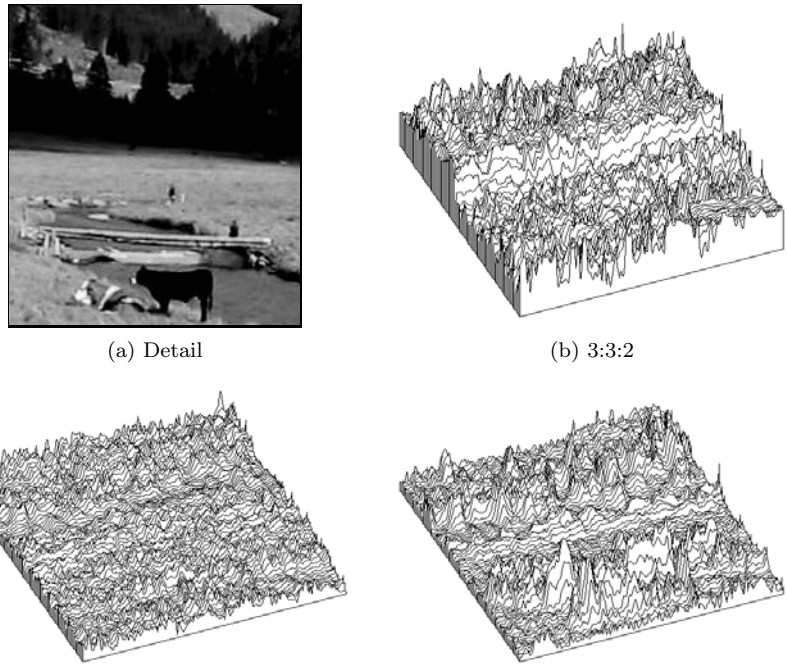
Abbildung 12.31
Farbverteilungen nach Anwendung des Median-Cut- (a) und Octree-Algorithmus (b). In beiden Fällen wurden die 226.321 Farben des Originalbilds (Abb. 12.27 (b)) auf 256 Farben reduziert.

Farben quantisiert. Auffällig beim Octree-Ergebnis ist vor allem die teilweise sehr dichte Platzierung im Bereich der Grünwerte. Die resultierenden Abweichungen gegenüber den Farben im Originalbild sind für diese beiden Verfahren und die skalare 3:3:2-Quantisierung in Abb. 12.32 dargestellt (als Distanzen im RGB-Farbraum). Der Gesamtfehler ist naturgemäß bei der 3:3:2-Quantisierung am höchsten, da hier die Bildinhalte selbst überhaupt nicht berücksichtigt werden. Die Abweichungen sind beim Octree-Algorithmus deutlich geringer als beim Median-Cut-Algorithmus, allerdings auf Kosten einzelner größerer Abweichungen, vor allem an den bunten Stellen im Bildvordergrund und im Bereich des Walds im Hintergrund.

Weitere Methoden zur Vektorquantisierung

Zur Bestimmung der repräsentativen Farbvektoren reicht es übrigens meist aus, nur einen Teil der ursprünglichen Bildpixel zu berücksichtigen. So genügt oft bereits eine zufällige Auswahl von nur 10 % aller Pixel, um mit hoher Wahrscheinlichkeit sicherzustellen, dass bei der Quantisierung keine wichtigen Farbwerte verloren gehen.

Neben den gezeigten Verfahren zur Farbquantisierung gibt es eine Reihe weiterer Methoden und verfeinerter Varianten. Dazu gehören u. a.



(a) Detail (b) 3:3:2

(c) Median-Cut (d) Octree

Abbildung 12.32
Quantisierungsfehler. Abweichung der quantisierten Farbwerte gegenüber dem Originalbild (a): skalare 3:3:2-Quantisierung (b), Median-Cut-Algorithmus (c) und Octree-Algorithmus (d).

statistische und Cluster-basierte Methoden, wie beispielsweise das klassische *k-means*-Verfahren, aber auch neuronale Netze und genetische Algorithmen (siehe [78] für eine aktuelle Übersicht).

12.6 Aufgaben

Aufg. 12.1. Programmieren Sie ein ImageJ-Plugin, das die einzelnen Farbkomponenten eines RGB-Farbbilds zyklisch vertauscht, also $R \rightarrow G \rightarrow B \rightarrow R$.

Aufg. 12.2. Programmieren Sie ein ImageJ-Plugin, das den Inhalt der Farbtabelle eines 8-Bit-Indexbilds als neues Bild mit 16×16 Farbfeldern anzeigt. Markieren Sie dabei die nicht verwendeten Tabelleneinträge in geeigneter Form. Als Ausgangspunkt dafür eignet sich beispielsweise Prog. 12.3.

Aufg. 12.3. Zeigen Sie, dass die in der Form $(r, g, b) \rightarrow (y, y, y)$ (Gl. 12.8) erzeugten „farblosen“ RGB-Pixel wiederum die subjektive Helligkeit y aufweisen.

Aufg. 12.4. Erweitern Sie das ImageJ-Plugin zur Desaturierung von Farbbildern in Prog. 12.5 so, dass es die vom Benutzer selektierte *Region of Interest* (ROI) berücksichtigt.

Aufg. 12.5. Berechnen Sie die Konvertierung von sRGB-Farbbildern in (unbunte) sRGB-Grauwertbilder nach den drei Varianten in Gl. 12.59 (unter fälschlicher Verwendung der nichtlinearen $R'G'B'$-Werte), 12.60 (exakte Berechnung) und 12.61 (Annäherung mit modifizierten Koeffizienten). Vergleichen Sie die Ergebnisse mithilfe von Differenzbildern und ermitteln Sie jeweils die Summe der Abweichungen.

Aufg. 12.6. Implementieren Sie auf Basis der Spezifikation in Abschn. 12.3.3 einen „echten" sRGB-Farbraum mit Referenzweißpunkt `D65` (und *nicht* `D50`) als neue Klasse `sRGB65_ColorSpace`, welche die Java-AWT-Klasse `ColorSpace` erweitert und alle erforderlichen Methoden realisiert. Überprüfen Sie, ob die Ergebnisse der Konvertierungsmethoden tatsächlich invers zueinander sind und vergleichen Sie diese mit denen des Java-sRGB-Raums.

Aufg. 12.7. Zur besseren Darstellung von Grauwertbildern werden bisweilen „Falschfarben" eingesetzt, z. B. bei medizinischen Bildern mit hoher Dynamik. Erstellen Sie ein ImageJ-Plugin für die Umwandlung eines 8-Bit-Grauwertbilds in ein Indexfarbbild mit 256 Farben, das die Glühfarben von Eisen (von Dunkelrot über Gelb bis Weiß) simuliert.

Aufg. 12.8. Die Bestimmung der visuellen Ähnlichkeit zwischen Bildern unabhängig von Größe und Detailstruktur ist ein häufiges Problem, z. B. im Zusammenhang mit der Suche in Bilddatenbanken. Farbstatistiken sind dabei ein wichtiges Element, denn sie ermöglichen auf relativ einfache und zuverlässige Weise eine grobe Klassifikation von Bildern, z. B. Landschaftsaufnahmen oder Portraits. Zweidimensionale Farbhistogramme (Abschn. 12.4.2) sind für diesen Zweck allerdings zu groß und umständlich. Eine einfache Idee könnte aber etwa darin bestehen, die 2D-Histogramme oder überhaupt das volle RGB-Histogramm in K (z. B. $3 \times 3 \times 3 = 27$) Würfel (*bins*) zu teilen und aus den zugehörigen Pixelhäufigkeiten einen K-dimensionalen Vektor zu bilden, der für jedes Bild berechnet wird und später zum ersten, groben Vergleich herangezogen wird. Überlegen Sie ein Konzept für ein solches Verfahren und auch die dabei möglichen Probleme.

Aufg. 12.9. In der `libjpeg` Open Source Software der *Independent JPEG Group* (www.ijg.org) ist der in Abschn. 12.5.2 beschriebene Median-Cut-Algorithmus zur Farbquantisierung mit folgender Modifikation implementiert: Die Auswahl des jeweils als Nächstes zu teilenden Quaders richtet sich abwechselnd (a) nach der Anzahl der enthaltenen Bildpixel und (b) nach dem geometrischen Volumen des Quaders. Überlegen Sie den Grund für dieses Vorgehen und argumentieren Sie anhand von Beispielen, ob und warum dies die Ergebnisse gegenüber dem herkömmlichen Verfahren verbessert.

13

Einführung in Spektraltechniken

In den folgenden drei Kapiteln geht es um die Darstellung und Analyse von Bildern im Frequenzbereich, basierend auf der Zerlegung von Bildsignalen in so genannte *harmonische* Funktionen, also Sinus- und Kosinusfunktionen, mithilfe der bekannten *Fouriertransformation*. Das Thema wird wegen seines etwas mathematischen Charakters oft als schwierig empfunden, weil auch die Anwendbarkeit in der Praxis anfangs nicht offensichtlich ist. Tatsächlich können die meisten gängigen Operationen und Methoden der digitalen Bildverarbeitung völlig ausreichend im gewohnten *Signal- oder Bildraum* dargestellt und verstanden werden, ohne Spektraltechniken überhaupt zu erwähnen bzw. zu kennen, weshalb das Thema hier (im Vergleich zu ähnlichen Texten) erst relativ spät aufgegriffen wird.

Wurden Spektraltechniken früher vorrangig aus Effizienzgründen für die Realisierung von Bildverarbeitungsoperationen eingesetzt, so spielt dieser Aspekt aufgrund der hohen Rechenleistung moderner Computer eine zunehmend untergeordnete Rolle. Dennoch gibt es einige wichtige Effekte und Verfahren in der digitalen Bildverarbeitung, die mithilfe spektraler Konzepte wesentlich einfacher oder ohne sie überhaupt nicht dargestellt werden können. Das Thema sollte daher nicht gänzlich umgangen werden. Die Fourieranalyse besitzt nicht nur eine elegante Theorie, sondern ergänzt auch in interessanter Weise einige bereits früher betrachtete Konzepte, insbesondere lineare Filter und die Faltungsoperation (Abschn. 6.2). Ebenso wichtig sind Spektraltechniken in vielen gängigen Verfahren für die Bild- und Videokompression, aber auch für das Verständnis der allgemeinen Zusammenhänge bei der Abtastung (Diskretisierung) von kontinuierlichen Signalen sowie bei der Rekonstruktion und Interpolation von diskreten Signalen.

Im Folgenden geben wir zunächst eine grundlegende Einführung in den Umgang mit Frequenzen und Spektralzerlegungen, die versucht, mit

einem Minimum an Formalismen auszukommen und daher auch für Leser ohne bisherigen Kontakt mit diesem Thema leicht zu „verdauen“ sein sollte. Wir beginnen mit der Darstellung eindimensionaler Signale und erweitern dies auf zweidimensionale Signale (Bilder) im nachfolgenden Kap. 14. Abschließend widmet sich Kap. 15 kurz der diskreten Kosinustransformation, einer Variante der Fouriertransformation, die vor allem bei der Bildkompression häufig Verwendung findet.

13.1 Die Fouriertransformation

Das allgemeine Konzept von „Frequenzen“ und der Zerlegung von Schwingungen in elementare, „harmonische“ Funktionen entstand ursprünglich im Zusammenhang von Schall, Tönen und Musik. Dabei erscheint die Idee, akustische Ereignisse auf der Basis „reiner“ Sinusfunktionen zu beschreiben, keineswegs unvernünftig, zumal Sinusschwingungen in natürlicher Weise bei jeder Form von Oszillation auftreten. Bevor wir aber fortfahren, zunächst (als Auffrischung) die wichtigsten Begriffe im Zusammenhang mit Sinus- und Kosinusfunktionen.

13.1.1 Sinus- und Kosinusfunktionen

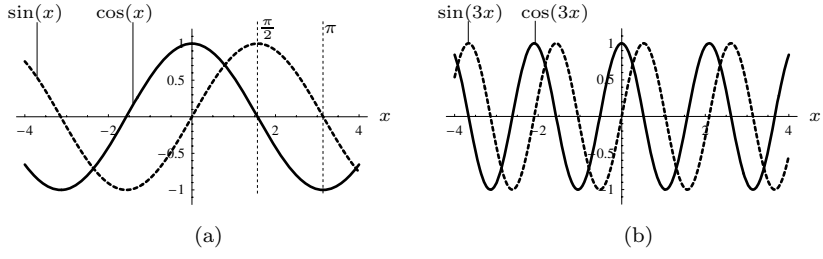
Die bekannte Kosinusfunktion

$$f(x) = \cos(x) \tag{13.1}$$

hat den Wert eins am Ursprung ($\cos(0) = 1$) und durchläuft bis zum Punkt $x = 2\pi$ eine volle *Periode* (Abb. 13.1 (a)). Die Funktion ist daher periodisch mit einer *Periodenlänge* $T = 2\pi$, d. h.

$$\cos(x) = \cos(x + 2\pi) = \cos(x + 4\pi) = \cdots = \cos(x + k2\pi) \tag{13.2}$$

für beliebige $k \in \mathbb{Z}$. Das Gleiche gilt für die entsprechende *Sinus*funktion $\sin(x)$ mit dem Unterschied, dass deren Wert am Ursprung null ist ($\sin(0) = 0$).



(a) (b)

Abbildung 13.1
Kosinus- und Sinusfunktion. Der Ausdruck $\cos(\omega x)$ beschreibt eine Kosinusfunktion mit der Kreisfrequenz ω an der Position x. Die periodische Funktion hat die Kreisfrequenz ω und damit die Periode $T = 2\pi/\omega$. Für $\omega = 1$ ist die Periode $T_1 = 2\pi$ (a), für $\omega = 3$ ist sie $T_3 = 2\pi/3 \approx 2.0944$ (b). Gleiches gilt für $\sin(\omega x)$.

Frequenz und Amplitude

Die Anzahl der Perioden von $\cos(x)$ innerhalb einer Strecke der Länge $T = 2\pi$ ist *eins* und damit ist auch die zugehörige *Kreisfrequenz*

$$\omega = \frac{2\pi}{T} = 1. \tag{13.3}$$

Wenn wir die Funktion modifizieren in der Form

$$f(x) = \cos(3x)\ , \tag{13.4}$$

dann erhalten wir eine gestauchte Kosinusschwingung, die dreimal schneller oszilliert als die ursprüngliche Funktion $\cos(x)$ (s. Abb. 13.1 (b)). Die Funktion $\cos(3x)$ durchläuft 3 volle Zyklen über eine Distanz von 2π und weist daher eine Kreisfrequenz $\omega = 3$ auf bzw. eine Periodenlänge $T = \frac{2\pi}{3}$. Im allgemeinen Fall gilt für die Periodenlänge

$$T = \tfrac{2\pi}{\omega}, \tag{13.5}$$

für $\omega > 0$. Die Sinus- und Kosinusfunktion oszilliert zwischen den Scheitelwerten $+1$ und -1. Eine Multiplikation mit einer Konstanten a ändert die *Amplitude* der Funktion und die Scheitelwerte auf $\pm a$. Im Allgemeinen ergibt

$$a \cdot \cos(\omega x) \qquad \text{und} \qquad a \cdot \sin(\omega x)$$

eine Kosinus- bzw. Sinusfunktion mit Amplitude a und Kreisfrequenz ω, ausgewertet an der Position (oder zum Zeitpunkt) x. Die Beziehung zwischen der Kreisfrequenz ω und der „gewöhnlichen“ Frequenz f ist

$$f = \frac{1}{T} = \frac{\omega}{2\pi} \quad \text{bzw.} \quad \omega = 2\pi f, \tag{13.6}$$

wobei f in Zyklen pro Raum- oder Zeiteinheit gemessen wird.[1] Wir verwenden je nach Bedarf ω oder f, und es sollte durch die unterschiedlichen Symbole jeweils klar sein, welche Art von Frequenz gemeint ist.

Phase

Wenn wir eine Kosinusfunktion entlang der x-Achse um eine Distanz φ verschieben, also

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

dann ändert sich die *Phase* der Kosinusschwingung und φ bezeichnet den *Phasenwinkel* der resultierenden Funktion. Damit ist auch die Sinusfunktion (vgl. Abb. 13.1) eigentlich nur eine Kosinusfunktion, die um eine Viertelperiode ($\varphi = \frac{2\pi}{4} = \frac{\pi}{2}$) nach rechts[2] verschoben ist, d. h.

[1] Beispielsweise entspricht die Frequenz $f = 1000\,\text{Zyklen/s}$ (Hertz) einer Periodenlänge von $T = 1/1000\,\text{s}$ und damit einer Kreisfrequenz von $\omega = 2000\pi$. Letztere ist eine einheitslose Größe.

[2] Die Funktion $f(x-d)$ ist allgemein die um die Distanz d nach rechts verschobene Funktion $f(x)$.

$$\sin(\omega x) = \cos\left(\omega x - \tfrac{\pi}{2}\right). \tag{13.7}$$

Nimmt man also die Kosinusfunktion als Referenz (mit Phase $\varphi_{\cos} = 0$), dann ist der Phasenwinkel der Sinusfunktion $\varphi_{\sin} = \frac{\pi}{2} = 90°$.

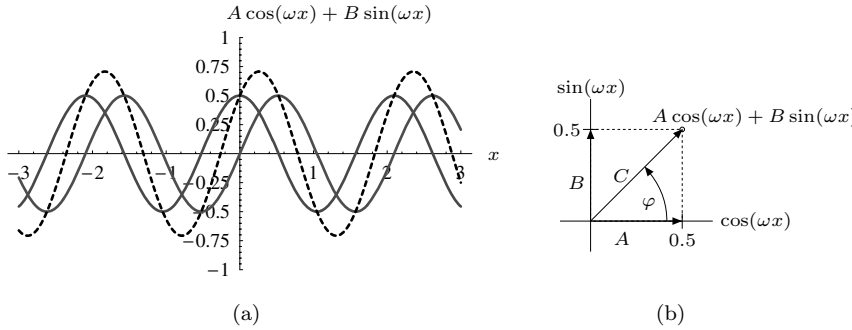
Kosinus- und Sinusfunktion sind also in gewissem Sinn „orthogonal“ und wir können diesen Umstand benutzen, um neue „sinusoidale“ Funktionen mit beliebiger Frequenz, Phase und Amplitude zu erzeugen. Insbesondere entsteht durch die Addition einer Kosinus- und Sinusfunktion mit identischer Frequenz ω und Amplituden A bzw. B eine weitere sinusoidale Funktion mit *derselben* Frequenz ω, d. h.

$$A \cdot \cos(\omega x) + B \cdot \sin(\omega x) = C \cdot \cos(\omega x - \varphi), \tag{13.8}$$

wobei die resultierende Amplitude C und der Phasenwinkel φ ausschließlich durch die beiden Amplituden A und B bestimmt sind als

$$C = \sqrt{A^2 + B^2} \quad \text{und} \quad \varphi = \tan^{-1}\left(\tfrac{B}{A}\right). \tag{13.9}$$

Abb. 13.2 zeigt ein Beispiel mit den Amplituden $A = B = 0.5$ und einem daraus resultierenden Phasenwinkel $\varphi = 45°$.



(a) (b)

Abbildung 13.2
Addition einer Kosinus- und einer Sinusfunktion mit identischer Frequenz: $A \cdot \cos(\omega x) + B \cdot \sin(\omega x)$, mit $\omega = 3$ und $A = B = 0.5$. Das Ergebnis ist eine phasenverschobene Kosinusfunktion (punktierte Kurve) mit Amplitude $C = \sqrt{0.5^2 + 0.5^2} \approx 0.707$ und Phasenwinkel $\varphi = 45°$.

Komplexwertige Sinusfunktionen – Euler’sche Notation

Das Diagramm in Abb. 13.2 (b) zeigt die Darstellung der Kosinus- und Sinuskomponenten als ein Paar orthogonaler, zweidimensionaler Vektoren, deren Länge den zugehörigen Amplituden A bzw. B entspricht. Dies erinnert uns an die Darstellung der reellen und imaginären Komponenten komplexer Zahlen in der zweidimensionalen Zahlenebene, also

$$z = a + \mathrm{i}\, b \in \mathbb{C},$$

wobei i die imaginäre Einheit bezeichnet ($\mathrm{i}^2 = -1$). Dieser Zusammenhang wird noch deutlicher, wenn wir die Euler’sche Notation einer beliebigen komplexen Zahlen z am Einheitskreis betrachten, nämlich

$$z = e^{\mathrm{i}\theta} = \cos(\theta) + \mathrm{i} \cdot \sin(\theta) \tag{13.10}$$

($e \approx 2.71828$ ist die Euler'sche Zahl). Betrachten wir den Ausdruck $e^{\mathrm{i}\theta}$ als Funktion über θ, dann ergibt sich ein „komplexwertiges Sinusoid", dessen reelle und imaginäre Komponente einer Kosinusfunktion bzw. einer Sinusfunktion entspricht, d. h.

$$\begin{aligned} \mathrm{Re}\{e^{\mathrm{i}\theta}\} &= \cos(\theta) \\ \mathrm{Im}\{e^{\mathrm{i}\theta}\} &= \sin(\theta) \end{aligned} \tag{13.11}$$

Da $z = e^{\mathrm{i}\theta}$ auf dem Einheitskreis liegt, ist die *Amplitude* des komplexwertigen Sinusoids $|z| = r = 1$. Wir können die Amplitude dieser Funktion durch Multiplikation mit einem reellen Wert $a \geq 0$ verändern, d. h.

$$|a \cdot e^{\mathrm{i}\theta}| = a \cdot |e^{\mathrm{i}\theta}| = a. \tag{13.12}$$

Die *Phase* eines komplexwertigen Sinusoids wird durch Addition eines Phasenwinkels bzw. durch Multiplikation mit einer komplexwertigen Konstante $e^{\mathrm{i}\varphi}$ am Einheitskreis verschoben,

$$e^{\mathrm{i}(\theta+\varphi)} = e^{\mathrm{i}\theta} \cdot e^{\mathrm{i}\varphi}. \tag{13.13}$$

Zusammenfassend verändert die Multiplikation mit einem reellen Wert nur die *Amplitude* der Sinusfunktion, eine Multiplikation mit einem komplexen Wert am Einheitskreis verschiebt nur die *Phase* (ohne Änderung der Amplitude) und die Multiplikation mit einem beliebigen komplexen Wert verändert sowohl *Amplitude* wie auch die *Phase* der Funktion (s. auch Anhang 1.2).

Die komplexe Notation ermöglicht es, Paare von Kosinus- und Sinusfunktionen $\cos(\omega x)$ bzw. $\sin(\omega x)$ mit identischer Frequenz ω in der Form

$$e^{\mathrm{i}\theta} = e^{\mathrm{i}\omega x} = \cos(\omega x) + \mathrm{i} \cdot \sin(\omega x) \tag{13.14}$$

in *einem* funktionalen Ausdruck zusammenzufassen. Wir kommen auf diese Notation bei der Behandlung der Fouriertransformation in Abschn. 13.1.4 nochmals zurück.

13.1.2 Fourierreihen als Darstellung periodischer Funktionen

Wie wir bereits in Gl. 13.8 gesehen haben, können sinusförmige Funktionen mit beliebiger Frequenz, Amplitude und Phasenlage als Summe entsprechend gewichteter Kosinus- und Sinusfunktionen dargestellt werden. Die Frage ist, ob auch andere, nicht sinusförmige Funktionen durch eine Summe von Kosinus- und Sinusfunktionenen zusammengesetzt werden können. Die Antwort ist natürlich *ja*. Es war Fourier[3], der diese Idee als Erster auf beliebige Funktionen erweiterte und zeigte, dass (beinahe) *jede* periodische Funktion $g(x)$ mit einer Grundfrequenz ω_0 als

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

(möglicherweise unendliche) Summe von „harmonischen“ Sinusfunktionen dargestellt werden kann in der Form

$$g(x) = \sum_{k=0}^{\infty} \left[A_k \cos(k\omega_0 x) + B_k \sin(k\omega_0 x) \right]. \tag{13.15}$$

Dies bezeichnet man als *Fourierreihe* und die konstanten Gewichte A_k, B_k als *Fourierkoeffizienten* der Funktion $g(x)$. Die Frequenzen der in der Fourierreihe beteiligten Funktionen sind ausschließlich ganzzahlige Vielfache („Harmonische“) der Grundfrequenz ω_0 (einschließlich der Frequenz 0 für $k = 0$). Die Koeffizienten A_k und B_k in Gl. 13.15, die zunächst unbekannt sind, können eindeutig aus der gegebenen Funktion $g(x)$ berechnet werden, ein Vorgang, der i. Allg. als *Fourieranalyse* bezeichnet wird.

13.1.3 Fourierintegral

Fourier wollte dieses Konzept nicht auf periodische Funktionen beschränken und postulierte, dass auch *nicht* periodische Funktionen in ähnlicher Weise als Summen von Sinus- und Kosinusfunktionen dargestellt werden können. Dies ist zwar grundsätzlich möglich, erfordert jedoch – über die Vielfachen der Grundfrequenz ($k\omega_0$) hinaus – i. Allg. unendlich viele, dicht aneinander liegende Frequenzen! Die resultierende Zerlegung

$$g(x) = \int_0^{\infty} A_\omega \cos(\omega x) + B_\omega \sin(\omega x)\, \mathrm{d}\omega \tag{13.16}$$

nennt man ein *Fourierintegral*, wobei die Koeffizienten A_ω und B_ω in Gl. 13.16 wiederum die Gewichte für die zugehörigen Kosinus- bzw. Sinusfunktionen mit der Frequenz ω sind. Das Fourierintegral ist die Grundlage für das *Fourierspektrum* und die *Fouriertransformation* [12, S. 745].

Jeder der Koeffizienten A_ω und B_ω spezifiziert, mit welcher Amplitude die zugehörige Kosinus- bzw. Sinusfunktion der Frequenz ω zur darzustellenden Signalfunktion $g(x)$ beiträgt. Was sind aber die richtigen Werte der Koeffizienten für eine gegebene Funktion $g(x)$ und können diese eindeutig bestimmt werden? Die Antwort ist *ja* und das „Rezept“ zur Bestimmung der Koeffizienten ist erstaunlich einfach:

$$\begin{aligned} A_\omega = A(\omega) &= \frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \cos(\omega x)\, \mathrm{d}x \\ B_\omega = B(\omega) &= \frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \sin(\omega x)\, \mathrm{d}x \end{aligned} \tag{13.17}$$

Da unendlich viele, kontinuierliche Frequenzwerte ω auftreten können, sind die Koeffizientenfunktionen $A(\omega)$ und $B(\omega)$ ebenfalls kontinuierlich. Sie enthalten eine Verteilung – also das „Spektrum“ – der im ursprünglichen Signal enthaltenen Frequenzkomponenten.

Das Fourierintegral beschreibt also die ursprüngliche Funktion $g(x)$ als Summe unendlich vieler Kosinus-/Sinusfunktionen mit kontinuierlichen (positiven) Frequenzwerten, wofür die Funktionen $A(\omega)$ bzw. $B(\omega)$ die zugehörigen Frequenzkoeffizienten liefern. Ein Signal $g(x)$ ist außerdem durch die zugehörigen Funktionen $A(\omega), B(\omega)$ eindeutig und vollständig repräsentiert. Dabei zeigt Gl. 13.17, wie wir zu einer Funktion $g(x)$ das zugehörige Spektrum berechnen können, und Gl. 13.16, wie man aus dem Spektrum die ursprüngliche Funktion bei Bedarf wieder rekonstruiert.

13.1.4 Fourierspektrum und -transformation

Von der in Gl. 13.17 gezeigten Zerlegung einer Funktion $g(x)$ bleibt nur mehr ein kleiner Schritt zur „richtigen“ Fouriertransformation. Diese betrachtet im Unterschied zum Fourierintegral sowohl die Ausgangsfunktion wie auch das zugehörige Spektrum als *komplexwertige* Funktionen, wodurch sich die Darstellung insgesamt wesentlich vereinfacht.

Ausgehend von den im Fourierintegral (Gl. 13.17) definierten Funktionen $A(\omega)$ und $B(\omega)$, ist das *Fourierspektrum* $G(\omega)$ einer Funktion $g(x)$ definiert als

$$\begin{aligned} G(\omega) &= \sqrt{\tfrac{\pi}{2}}\Big[A(\omega) - \mathrm{i}\cdot B(\omega)\Big] \qquad (13.18)\\ &= \sqrt{\tfrac{\pi}{2}}\left[\frac{1}{\pi}\int_{-\infty}^{\infty} g(x)\cdot\cos(\omega x)\,\mathrm{d}x \;-\; \mathrm{i}\cdot\frac{1}{\pi}\int_{-\infty}^{\infty} g(x)\cdot\sin(\omega x)\,\mathrm{d}x\right]\\ &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} g(x)\cdot\Big[\cos(\omega x) - \mathrm{i}\cdot\sin(\omega x)\Big]\,\mathrm{d}x\,, \end{aligned}$$

wobei $g(x), G(\omega) \in \mathbb{C}$. Unter Verwendung der Euler'schen Schreibweise für komplexe Zahlen (Gl. 13.14) ergibt sich aus Gl. 13.18 die übliche Formulierung für das kontinuierliche *Fourierspektrum*:

$$\boxed{G(\omega) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} g(x)\cdot e^{-\mathrm{i}\omega x}\,\mathrm{d}x} \qquad (13.19)$$

Der Übergang von der Funktion $g(x)$ zu ihrem Fourierspektrum $G(\omega)$ bezeichnet man als *Fouriertransformation*[4] ($\mathcal{F}$). Umgekehrt kann die ursprüngliche Funktion $g(x)$ aus dem Fourierspektrum $G(\omega)$ durch die *inverse* Fouriertransformation[5] ($\mathcal{F}^{-1}$)

$$\boxed{g(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} G(\omega)\cdot e^{\mathrm{i}\omega x}\,\mathrm{d}\omega} \qquad (13.20)$$

wiederum eindeutig rekonstruiert werden.

[4] Auch „direkte“ oder „Vorwärtstransformation“.
[5] Auch „Rückwärtstransformation“.

Auch für den Fall, dass eine der betroffenen Funktionen ($g(x)$ bzw. $G(\omega)$) reellwertig ist (was für konkrete Signale $g(x)$ üblicherweise zutrifft), ist die andere Funktion i. Allg. komplexwertig. Man beachte auch, dass die Vorwärtstransformation $\mathcal{F}$ (Gl. 13.19) und die inverse Transformation $\mathcal{F}^{-1}$ (Gl. 13.20) bis auf das Vorzeichen des Exponenten völlig symmetrisch sind.[6] *Ortsraum* und *Spektralraum* sind somit zueinander „duale“ Darstellungsformen, die sich grundsätzlich nicht unterscheiden.

13.1.5 Fourier-Transformationspaare

Zwischen einer Funktion $g(x)$ und dem zugehörigen Fourierspektrum $G(\omega)$ besteht ein eindeutiger Zusammenhang in beiden Richtungen: Das Fourierspektrum eines Signals ist eindeutig und zu einem bestimmten Spektrum gibt es nur ein zugehöriges Signal – die beiden Funktionen $g(x)$ und $G(\omega)$ bilden ein sog. „Transformationspaar“,

$$g(x) \circ\!\!-\!\!\bullet\, G(\omega).$$

Tabelle 13.1 zeigt einige ausgewählte Transformationspaare analytischer Funktionen, die in den Abbildungen 13.3 und 13.4 auch grafisch dargestellt sind.

So besteht etwa das Fourierspektrum einer **Kosinusfunktion** $\cos(\omega_0 x)$ aus zwei getrennten, dünnen Pulsen, die symmetrisch im Abstand von ω_0 vom Ursprung angeordnet sind (Abb. 13.3 (a, c)). Dies entspricht intuitiv auch unserer physischen Vorstellung eines Spektrums, etwa in Bezug auf einen völlig reinen, monophonen Ton in der Akustik oder der Haarlinie, die eine extrem reine Farbe in einem optischen Spektrum hinterlässt. Bei steigender Frequenz $\omega_0 x$ bewegen sich die resultierenden Pulse im Spektrum vom Ursprung weg. Man beachte, dass das Spektrum der Kosinusfunktion reellwertig ist, der Imaginärteil ist null. Gleiches gilt auch für die Sinusfunktion (Abb. 13.3 (b, d)), mit dem Unterschied, dass hier die Pulse nur im Imaginärteil des Spektrums und mit unterschiedlichen Vorzeichen auftreten. In diesem Fall ist also der Realteil des Spektrums null.

Interessant ist auch das Verhalten der **Gauß-Funktion** (Abb. 13.4 (a, b)), deren Fourierspektrum wiederum eine Gauß-Funktion ist. Die Gauß-Funktion ist damit eine von wenigen Funktionen, die im Ortsraum *und* im Spektralraum denselben Funktionstyp aufweisen. Im Fall der Gauß-Funktion ist auch deutlich zu erkennen, dass eine *Dehnung* des Signals im Ortsraum zu einer *Stauchung* der Funktion im Spektralraum führt und umgekehrt!

[6] Es gibt mehrere gängige Definitionen der Fouriertransformation, die sich u. a. durch den Faktor vor dem Integral und durch die Vorzeichen der Exponenten in der Vorwärts- und Rückwärtstransformation unterscheiden. Alle diese Versionen sind grundsätzlich äquivalent. Die hier gezeigte, symmetrische Version verwendet den gleichen Faktor $(1/\sqrt{2\pi})$ für beide Richtungen der Transformation.

Funktion	*Transformationspaar* $g(x) \circ\!\!-\!\!\bullet\, G(\omega)$	*Abb.*
Kosinusfunktion mit Frequenz ω_0	$g(x) = \cos(\omega_0 x)$ $G(\omega) = \sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega-\omega_0) + \delta(\omega+\omega_0)\big)$	13.3 (a,c)
Sinusfunktion mit Frequenz ω_0	$g(x) = \sin(\omega_0 x)$ $G(\omega) = \mathrm{i}\sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega-\omega_0) - \delta(\omega+\omega_0)\big)$	13.3 (b,d)
Gauß-Funktion der Breite σ	$g(x) = \frac{1}{\sigma} \cdot e^{-\frac{x^2}{2\sigma^2}}$ $G(\omega) = e^{-\frac{\sigma^2\omega^2}{2}}$	13.4 (a,b)
Rechteckpuls der Breite $2b$	$g(x) = \Pi_b(x) = \begin{cases} 1 & \lvert x\rvert \le b \\ 0 & \text{else} \end{cases}$ $G(\omega) = \frac{2b\sin(b\omega)}{\sqrt{2\pi}\omega}$	13.4 (c,d)

Tabelle 13.1
Fourier-Transformationspaare für ausgewählte Funktionen. $\delta()$ bezeichnet die Impuls- oder Dirac-Funktion (s. Abschn. 13.2.1).

Die Fouriertransformation eines **Rechteckpulses** (Abb. 13.4 (c,d)) ergibt die charakteristische „Sinc"-Funktion der Form $\sin(x)/x$, die mit zunehmenden Frequenzen nur langsam ausklingt und damit sichtbar macht, dass im ursprünglichen Rechtecksignal Komponenten enthalten sind, die über einen großen Bereich von Frequenzen verteilt sind. Rechteckpulse weisen also grundsätzlich ein sehr breites Frequenzspektrum auf.

13.1.6 Wichtige Eigenschaften der Fouriertransformation

Symmetrie

Das Fourierspektrum erstreckt sich über positive und negative Frequenzen und ist, obwohl im Prinzip beliebige komplexe Funktionen auftreten können, in vielen Fällen um den Ursprung symmetrisch (s. beispielsweise [16, S. 178]). Insbesondere ist die Fouriertransformierte eines reellwertigen Signals $g(x) \in \mathbb{R}$ eine so genannte *hermitesche* Funktion, d. h.

$$G(\omega) = G^*(-\omega), \tag{13.21}$$

wobei G^* den konjugiert komplexen Wert von G bezeichnet (s. auch Anhang 1.2).

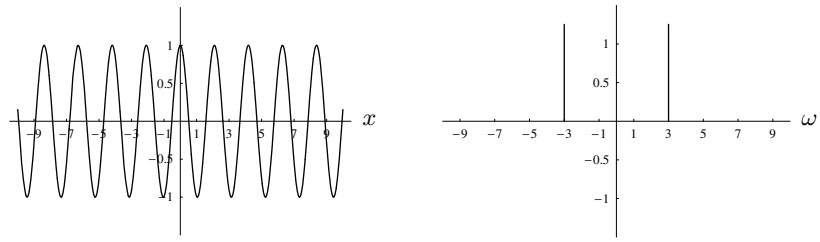
Linearität

Die Fouriertransformation ist eine *lineare* Operation, sodass etwa die Multiplikation des Signals mit einer beliebigen Konstanten $a \in \mathbb{C}$ in gleicher Weise auch das zugehörige Spektrum verändert, d. h.

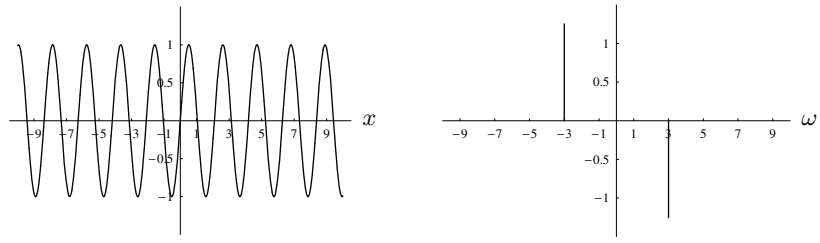
$$a \cdot g(x) \circ\!\!-\!\!\bullet\, a \cdot G(\omega). \tag{13.22}$$

Darüber hinaus bedingt die Linearität, dass die Transformation der Summe zweier Signale $g(x) = g_1(x) + g_2(x)$ identisch ist zur Summe der zugehörigen Fouriertransformierten $G_1(\omega)$ und $G_2(\omega)$:

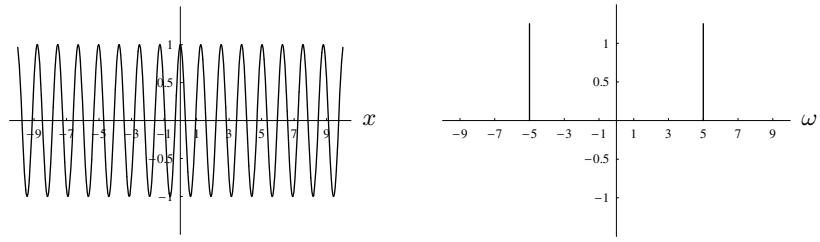
Abbildung 13.3
Fourier-Transformationspaare – Kosinus-/Sinusfunktionen.



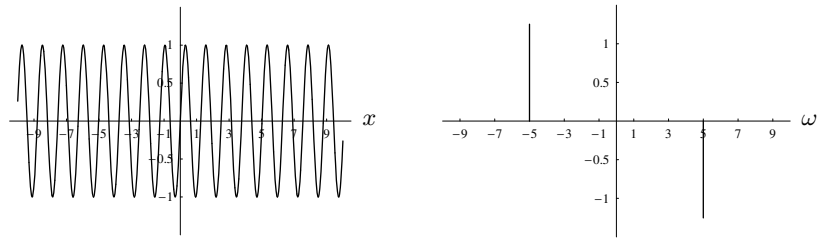
(a) Kosinus ($\omega_0 = 3$): $g(x) = \cos(3x)$ $\quad\circ\!\!-\!\!\bullet\quad$ $G(\omega) = \sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega-3) + \delta(\omega+3)\big)$



(b) Sinus ($\omega_0 = 3$): $g(x) = \sin(3x)$ $\quad\circ\!\!-\!\!\bullet\quad$ $G(\omega) = \mathrm{i}\sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega-3) - \delta(\omega+3)\big)$

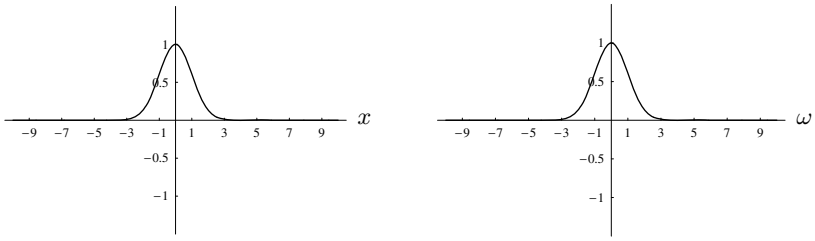


(c) Kosinus ($\omega_0 = 5$): $g(x) = \cos(5x)$ $\quad\circ\!\!-\!\!\bullet\quad$ $G(\omega) = \sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega-5) + \delta(\omega+5)\big)$

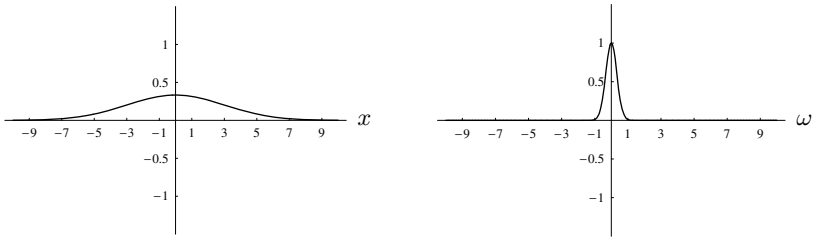


(d) Sinus ($\omega_0 = 5$): $g(x) = \sin(5x)$ $\quad\circ\!\!-\!\!\bullet\quad$ $G(\omega) = \mathrm{i}\sqrt{\frac{\pi}{2}} \cdot \big(\delta(\omega-5) - \delta(\omega+5)\big)$

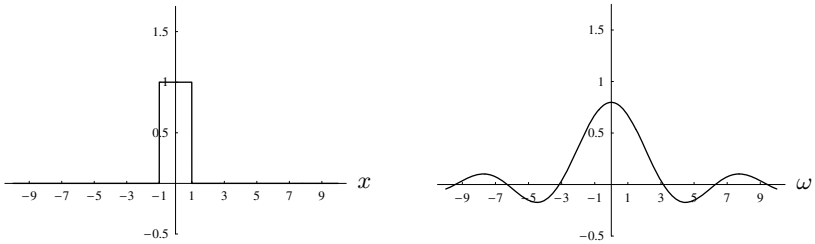
Abbildung 13.4
Fourier-Transformationspaare – Gauß-Funktion und Rechteckpuls.



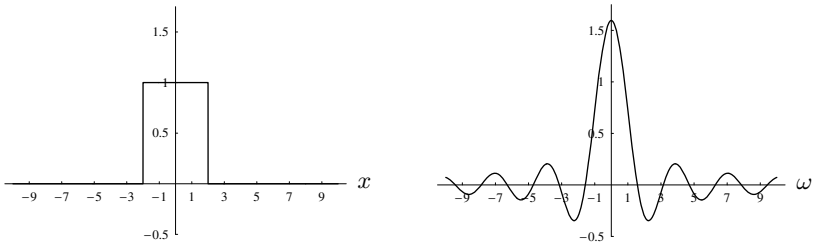
(a) Gauß ($\sigma=1$): $g(x) = e^{-\frac{x^2}{2}}$ $\circ\!\!-\!\!\bullet$ $G(\omega) = e^{-\frac{\omega^2}{2}}$



(b) Gauß ($\sigma=3$): $g(x) = \frac{1}{3} \cdot e^{-\frac{x^2}{2 \cdot 9}}$ $\circ\!\!-\!\!\bullet$ $G(\omega) = e^{-\frac{9\omega^2}{2}}$



(c) Rechteckpuls ($b=1$): $g(x) = \Pi_1(x)$ $\circ\!\!-\!\!\bullet$ $G(\omega) = \frac{2\sin(\omega)}{\sqrt{2\pi}\omega}$



(d) Rechteckpuls ($b=2$): $g(x) = \Pi_2(x)$ $\circ\!\!-\!\!\bullet$ $G(\omega) = \frac{4\sin(2\omega)}{\sqrt{2\pi}\omega}$

$$g_1(x) + g_2(x) \circ\!\!-\!\!\bullet\, G_1(\omega) + G_2(\omega). \tag{13.23}$$

Ähnlichkeit

Wird die ursprüngliche Funktion $g(x)$ in der Zeit oder im Raum skaliert, so tritt der jeweils umgekehrte Effekt im zugehörigen Fourierspektrum auf. Wie wir bereits in Abschn. 13.1.5 beobachten konnten, führt insbesondere eine Stauchung des Signals um einen Faktor s, d. h. $g(x) \to g(sx)$, zu einer entsprechenden Streckung der Fouriertransformierten, also

$$g(sx) \circ\!\!-\!\!\bullet\, \tfrac{1}{|s|} \cdot G\left(\tfrac{\omega}{s}\right). \tag{13.24}$$

Umgekehrt wird natürlich das Signal gestaucht, wenn das zugehörige Spektrum gestreckt wird.

Verschiebungseigenschaft

Wird die ursprüngliche Funktion $g(x)$ um eine Distanz d entlang der Koordinatenachse verschoben, also $g(x) \to g(x-d)$, so multipliziert sich dadurch das Fourierspektrum um einen von ω abhängigen komplexen Wert $e^{-\mathrm{i}\omega d}$:

$$g(x-d) \circ\!\!-\!\!\bullet\, e^{-\mathrm{i}\omega d} \cdot G(\omega). \tag{13.25}$$

Da der Faktor $e^{-\mathrm{i}\omega d}$ auf dem Einheitskreis liegt, führt die Multiplikation (vgl. Gl. 13.13) nur zu einer Phasenverschiebung der Spektralwerte, also einer Umverteilung zwischen Real- und Imaginärteil, ohne dabei den Betrag $|G(\omega)|$ zu verändern. Der Winkel dieser Phasenverschiebung (ωd) ändert sich offensichtlich linear mit der Kreisfrequenz ω.

Faltungseigenschaft

Der für uns vielleicht interessanteste Aspekt der Fouriertransformation ergibt sich aus ihrem Verhältnis zur linearen Faltung (Abschn. 6.3.1). Angenommen, wir hätten zwei Funktionen $g(x)$ und $h(x)$ sowie die zugehörigen Fouriertransformierten $G(\omega)$ bzw. $H(\omega)$. Unterziehen wir diese Funktionen einer linearen Faltung, also $g(x) * h(x)$, dann ist die Fouriertransformierte des Resultats gleich dem (punktweisen) *Produkt* der einzelnen Fouriertransformierten $G(\omega)$ und $H(\omega)$:

$$g(x) * h(x) \circ\!\!-\!\!\bullet\, G(\omega) \cdot H(\omega). \tag{13.26}$$

Aufgrund der Dualität von Orts- und Spektralraum gilt das Gleiche auch in umgekehrter Richtung, d. h., eine punktweise Multiplikation der Signale entspricht einer linearen Faltung der zugehörigen Fouriertransformierten:

$$g(x) \cdot h(x) \circ\!\!-\!\!\bullet\, G(\omega) * H(\omega). \tag{13.27}$$

Eine Multiplikation der Funktionen in *einem* Raum (Orts- oder Spektralraum) entspricht also einer linearen Faltung der zugehörigen Transformierten im jeweils *anderen* Raum.

13.2 Übergang zu diskreten Signalen

Die Definition der kontinuierlichen Fouriertransformation ist für die numerische Berechnung am Computer nicht unmittelbar geeignet. Weder können beliebige kontinuierliche (und möglicherweise unendliche) Funktionen dargestellt, noch können die dafür erforderlichen Integrale tatsächlich berechnet werden. In der Praxis liegen auch immer *diskrete* Daten vor und wir benötigen daher eine Version der Fouriertransformation, in der sowohl das Signal wie auch das zugehörige Spektrum als endliche Vektoren dargestellt werden – die „diskrete“ Fouriertransformation. Zuvor wollen wir jedoch unser bisheriges Wissen verwenden, um dem Vorgang der Diskretisierung von Signalen etwas genauer auf den Grund zu gehen.

13.2.1 Abtastung

Wir betrachten zunächst die Frage, wie eine kontinuierliche Funktion überhaupt in eine diskrete Funktion umgewandelt werden kann. Dieser Vorgang wird als *Abtastung* (Sampling) bezeichnet, also die Entnahme von Abtastwerten der zunächst kontinuierlichen Funktion an bestimmten Punkten in der Zeit oder im Raum, üblicherweise in regelmäßigen Abständen. Um diesen Vorgang in einfacher Weise auch formal beschreiben zu können, benötigen wir ein unscheinbares, aber wichtiges Stück aus der mathematischen Werkzeugkiste.

Die Impulsfunktion $\delta(x)$

Die Impulsfunktion (auch *Delta*- oder *Dirac*-Funktion) ist uns bereits im Zusammenhang mit der Impulsantwort von Filtern (Abschn. 6.3.4) sowie in den Fouriertransformierten der Kosinus- und Sinusfunktion (Abb. 13.3) begegnet. Diese Funktion, die einen kontinuierlichen, “idealen” Impuls modelliert, ist in mehrfacher Hinsicht ungewöhnlich: Ihr Wert ist überall null mit Ausnahme des Ursprungs, wo ihr Wert zwar ungleich null, aber undefiniert ist, und außerdem ist ihr Integral eins, also

$$\delta(x) = 0 \;\text{ für }\; x \neq 0 \quad \text{und} \quad \int_{-\infty}^{\infty} \delta(x)\,\mathrm{d}x = 1\,. \tag{13.28}$$

Man kann sich $\delta(x)$ als einzelnen Puls an der Position null vorstellen, der unendlich schmal ist, aber dennoch endliche Energie (1) aufweist. Bemerkenswert ist auch das Verhalten der Impulsfunktion bei einer Skalierung in der Zeit- oder Raumachse, also $\delta(x) \to \delta(sx)$, wofür gilt

$$\delta(sx) = \frac{1}{|s|} \cdot \delta(x) \quad \text{für } s \neq 0. \tag{13.29}$$

Obwohl $\delta(x)$ in der physischen Realität nicht existiert und eigentlich auch nicht gezeichnet werden kann (die entsprechenden Kurven in Abb. 13.3 dienen nur zur Illustration), ist diese Funktion – wie im Folgenden gezeigt – ein wichtiges Element zur formalen Beschreibung des Abtastvorgangs.

Abtastung mit der Impulsfunktion

Mit dem Konzept der idealen Impulsfunktion lässt sich der Abtastvorgang relativ einfach und anschaulich darstellen.[7] Wird eine kontinuierliche Funktion $g(x)$ mit der Impulsfunktion $\delta(x)$ punktweise multipliziert, so entsteht eine neue Funktion $\bar{g}(x)$ der Form

$$\bar{g}(x) = g(x) \cdot \delta(x) = \begin{cases} g(0) & \text{für } x = 0 \\ 0 & \text{sonst.} \end{cases} \tag{13.30}$$

$\bar{g}(x)$ besteht also aus einem einzigen Puls an der Position 0, dessen Höhe dem Wert der ursprünglichen Funktion $g(0)$ entspricht. Wir erhalten also durch die Multiplikation mit der Impulsfunktion einen einzelnen, diskreten Abtastwert der Funktion $g(x)$ an der Stelle $x = 0$. Durch Verschieben der Impulsfunktion um eine Distanz x_0 können wir $g(x)$ an jeder *beliebigen* Stelle $x = x_0$ abtasten, denn es gilt

$$\bar{g}(x) = g(x) \cdot \delta(x - x_0) = \begin{cases} g(x_0) & \text{für } x = x_0 \\ 0 & \text{sonst.} \end{cases} \tag{13.31}$$

Darin ist $\delta(x - x_0)$ die um x_0 verschobene Impulsfunktion und die resultierende Funktion $\bar{g}(x)$ ist null, außer an der Stelle x_0, wo sie den ursprünglichen Funktionswert $g(x_0)$ enthält. Dieser Zusammenhang ist in Abb. 13.5 für die Abtastposition $x_0 = 3$ dargestellt.

Um die Funktion $g(x)$ an mehr als einer Stelle gleichzeitig abzutasten, etwa an den Positionen x_1 und x_2, verwenden wir zwei individuell verschobene Exemplare der Impulsfunktion, multiplizieren $g(x)$ mit beiden und addieren anschließend die einzelnen Abtastergebnisse. In diesem speziellen Fall erhalten wir

$$\begin{aligned} \bar{g}(x) &= g(x) \cdot \delta(x - x_1) + g(x) \cdot \delta(x - x_2) && (13.32) \\ &= g(x) \cdot \big[\delta(x - x_1) + \delta(x - x_2)\big] && (13.33) \\ &= \begin{cases} g(x_1) & \text{für } x = x_1 \\ g(x_2) & \text{für } x = x_2 \\ 0 & \text{sonst.} \end{cases} \end{aligned}$$

Die Abtastung einer kontinuierlichen Funktion $g(x)$ an einer *Folge* von N Positionen $x_i = 1, 2, \ldots N$ kann daher (nach Gl. 13.33) als Summe der N Einzelabtastungen dargestellt werden, also durch

$$\begin{aligned} \bar{g}(x) &= g(x) \cdot \big[\delta(x-1) + \delta(x-2) + \ldots + \delta(x-N)\big] \\ &= g(x) \cdot \sum_{i=1}^{N} \delta(x - i)\,. \end{aligned} \tag{13.34}$$

[7] Der nachfolgende Abschnitt ist bewusst intuitiv und daher auch (im mathematischen Sinn) oberflächlich gehalten. Formal genauere Beschreibungen finden sich beispielsweise in [16, 49].

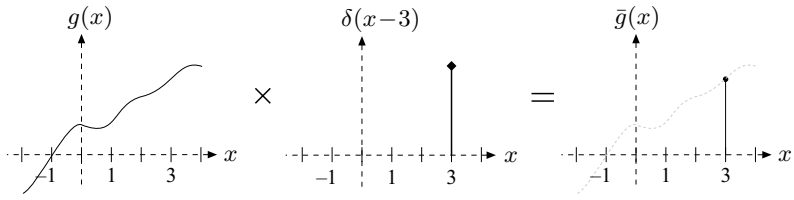


Abbildung 13.5
Abtastung mit der Impulsfunktion. Durch Multiplikation des kontinuierlichen Signals $g(x)$ mit der verschobenen Impulsfunktion $\delta(x-3)$ wird $g(x)$ an der Stelle $x_0 = 3$ abgetastet.

Die Kammfunktion

Die Summe von verschobenen Einzelpulsen $\sum_{i=1}^{N} \delta(x-i)$ in Gl. 13.34 wird auch als „Pulsfolge“ bezeichnet. Wenn wir die Pulsfolge in beiden Richtungen bis ins Unendliche erweitern, erhalten wir eine Funktion

$$\text{Ш}(x) = \sum_{i=-\infty}^{\infty} \delta(x-i)\,, \tag{13.35}$$

die als *Kammfunktion*[8] bezeichnet wird. Die Diskretisierung einer kontinuierlichen Funktion durch Abtastung in regelmäßigen, ganzzahligen Intervallen kann dann in der einfachen Form

$$\bar{g}(x) = g(x) \cdot \text{Ш}(x) \tag{13.36}$$

modelliert werden, d. h. als punktweise Multiplikation des ursprünglichen Signals $g(x)$ mit der Kammfunktion $\text{Ш}(x)$. Wie in Abb. 13.6 dargestellt, werden die Werte der Funktion $g(x)$ dabei nur an den ganzzahligen Positionen $x_i \in \mathbb{Z}$ in die diskrete Funktion $\bar{g}(x_i)$ übernommen und überall sonst ignoriert. Das Abtastintervall, also der Abstand zwischen benachbarten Abtastwerten, muss dabei keineswegs 1 sein. Um in beliebigen, regelmäßigen Abständen τ abzutasten, wird die Kammfunktion in Richtung der Zeit- bzw. Raumachse einfach entsprechend skaliert, d. h.

$$\bar{g}(x) = g(x) \cdot \text{Ш}\left(\tfrac{x}{\tau}\right)\,, \quad \text{für } \tau > 0\,. \tag{13.37}$$

Auswirkungen der Abtastung auf das Fourierspektrum

Trotz der eleganten Modellierung der Abtastung auf Basis der Kammfunktion könnte man sich zu Recht die Frage stellen, wozu bei einem derart simplen Vorgang überhaupt eine so komplizierte Formulierung notwendig ist. Eine Antwort darauf gibt uns das Fourierspektrum. Die Abtastung einer kontinuierlichen Funktion hat massive (wenn auch gut abschätzbare) Auswirkungen auf das Frequenzspektrum des resultierenden (diskreten) Signals, und der Einsatz der Kammfunktion als formales Modell des Abtastvorgangs macht es relativ einfach, diese spektralen

[8] Im Englischen wird $\text{Ш}(x)$ „comb function“ oder auch „Shah function“ genannt.

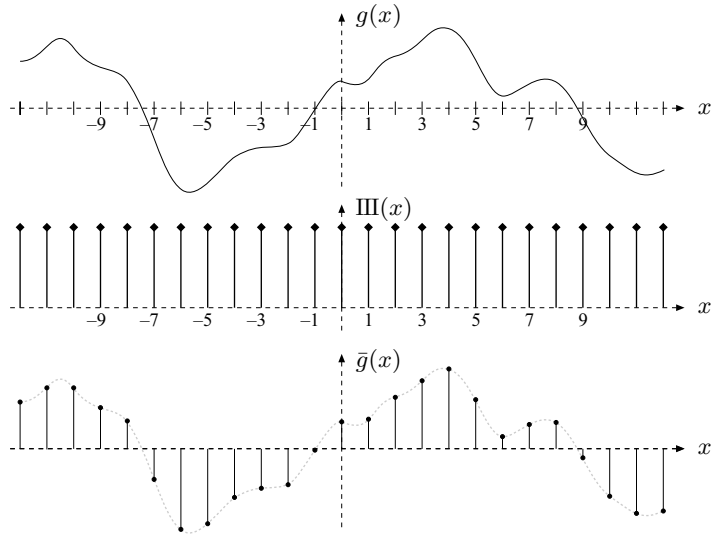


Abbildung 13.6
Abtastung mit der Kammfunktion. Das ursprüngliche, kontinuierliche Signal $g(x)$ wird mit der Kammfunktion $\text{Ш}(x)$ multipliziert. Nur an den ganzzahligen Positionen $x_i \in \mathbb{Z}$ wird der entsprechende Wert $g(x_i)$ in das Ergebnis $\bar{g}(x_i)$ übernommen, überall sonst ignoriert.

Auswirkungen vorherzusagen bzw. zu interpretieren. Die Kammfunktion besitzt, ähnlich der Gauß-Funktion, die seltene Eigenschaft, dass ihre Fouriertransformierte

$$\text{Ш}(x) \circ\!\!-\!\!\bullet\, \text{Ш}(\tfrac{1}{2\pi}\omega) \tag{13.38}$$

wiederum eine Kammfunktion ist, also den gleichen Funktionstyp hat. Skaliert auf ein beliebiges Abtastintervall τ ergibt sich aufgrund der Ähnlichkeitseigenschaft (Gl. 13.24) im allgemeinen Fall als Fouriertransformierte der Kammfunktion

$$\text{Ш}(\tfrac{x}{\tau}) \circ\!\!-\!\!\bullet\, \tau\text{Ш}\left(\tfrac{\tau}{2\pi}\omega\right). \tag{13.39}$$

Abb. 13.7 zeigt zwei Beispiele der Kammfunktionen $\text{Ш}_\tau(x)$ mit unterschiedlichen Abtastintervallen $\tau = 1$ bzw. $\tau = 3$ sowie die zugehörigen Fouriertransformierten.

Was passiert nun bei der Diskretisierung mit dem Fourierspektrum, wenn wir also im Ortsraum ein Signal $g(x)$ mit einer Kammfunktion $\text{Ш}(\frac{x}{\tau})$ multiplizieren? Die Antwort erhalten wir über die Faltungseigenschaft der Fouriertransformation (Gl. 13.26): Das Produkt zweier Funktionen in einem Raum (entweder im Orts- oder im Spektralraum) entspricht einer linearen Faltung im jeweils anderen Raum, d. h.

$$g(x) \cdot \text{Ш}(\tfrac{x}{\tau}) \circ\!\!-\!\!\bullet\, G(\omega) * \tau\text{Ш}\left(\tfrac{\tau}{2\pi}\omega\right). \tag{13.40}$$

Nun ist das Fourierspektrum der Abtastfunktion wiederum eine Kammfunktion und besteht daher aus einer regelmäßigen Folge von Impulsen (Abb. 13.7). Die Faltung einer beliebigen Funktion mit einem Impuls $\delta(x)$ ergibt aber wiederum die ursprüngliche Funktion, also $f(x) * \delta(x) = f(x)$. Die Faltung mit einem um d *verschobenen* Impuls $\delta(x-d)$ reproduziert

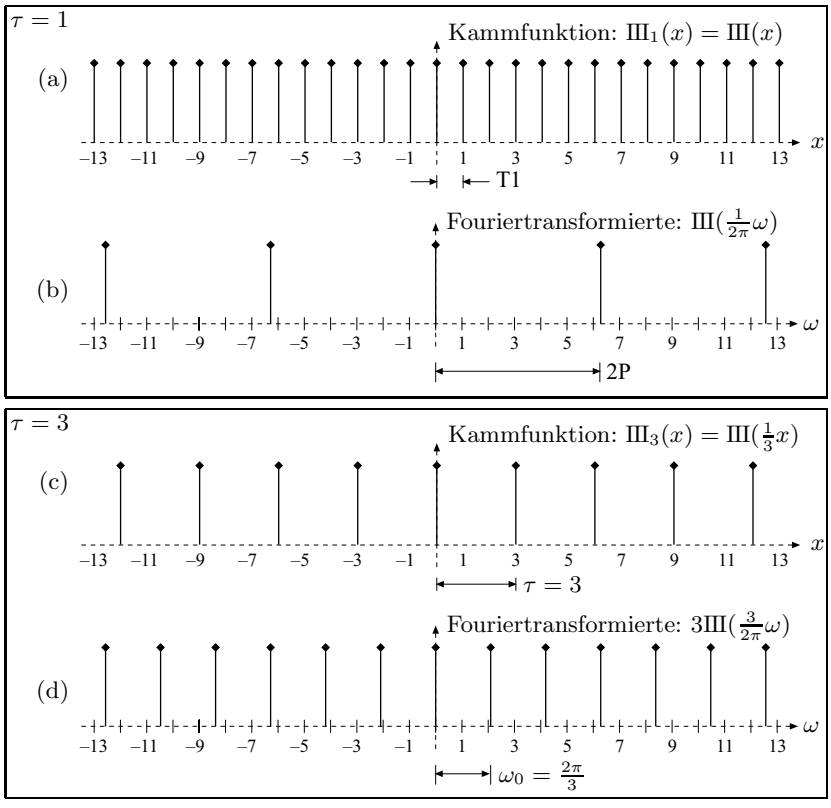


Abbildung 13.7
Kammfunktion und deren Fouriertransformierte. Kammfunktion $\text{Ш}_\tau(x)$ für das Abtastintervall $\tau = 1$ (a) und die zugehörige Fouriertransformierte (b). Kammfunktion für $\tau = 3$ (c) und Fouriertransformierte (d). Man beachte, dass die tatsächliche Höhe der einzelnen δ-Pulse nicht definiert ist und hier nur zur Illustration dargestellt ist.

ebenfalls die ursprüngliche Funktion $f(x)$, jedoch verschoben um die gleiche Distanz d:

$$f(x) * \delta(x-d) = f(x-d). \tag{13.41}$$

Das hat zur Folge, dass im Fourierspektrum des abgetasteten Signals $\bar{G}(\omega)$ das Spektrum $G(\omega)$ des ursprünglichen, kontinuierlichen Signals unendlich oft, nämlich an jedem Puls im Spektrum der Abtastfunktion, repliziert wird (Abb. 13.8 (a,b))!

Das daraus resultierende Fourierspektrum ist daher periodisch mit der Periodenlänge $\frac{2\pi}{\tau}$, also im Abstand der Abtastfrequenz ω_s.

Aliasing und das Abtasttheorem

Solange sich die durch die Abtastung replizierten Spektralkomponenten in $\bar{G}(\omega)$ nicht überlappen, kann das ursprüngliche Spektrum $G(\omega)$ – und damit auch das ursprüngliche, kontinuierliche Signal $g(x)$ – ohne Verluste aus einer beliebigen Replika von $G(\omega)$ aus dem periodischen Spektrum $\bar{G}(\omega)$ rekonstruiert werden. Dies erfordert jedoch offensichtlich (Abb. 13.8), dass die im ursprünglichen Signal $g(x)$ enthaltenen Frequenzen nach oben beschränkt sind, das Signal also keine Komponenten mit Frequenzen größer als $\omega_{\max}$ enthält. Die maximal zulässige Signalfrequenz

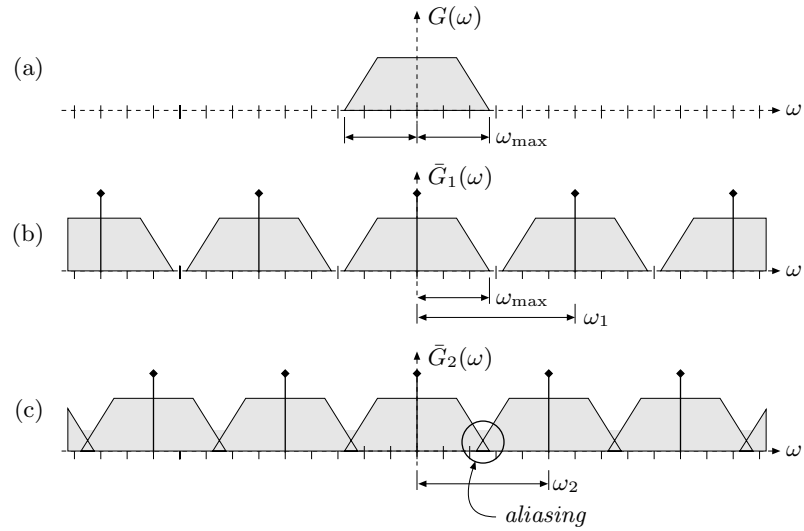


Abbildung 13.8 Auswirkungen der Abtastung im Fourierspektrum. Das Spektrum $G(\omega)$ des ursprünglichen, kontinuierlichen Signals ist angenommen bandbegrenzt im Bereich $\pm\omega_{\max}$ (a). Die Abtastung des Signals mit einer Abtastfrequenz $\omega_s = \omega_1$ bewirkt, dass das Signalspektrum $G(\omega)$ an jeweils Vielfachen von ω_1 entlang der Frequenzachse (ω) repliziert wird (b). Die replizierten Spektralteile überlappen sich nicht, solange $\omega_1 > 2\omega_{\max}$. In (c) ist die Abtastfrequenz $\omega_s = \omega_2$ kleiner als $2\omega_{\max}$, sodass sich die einzelnen Spektralteile überlappen, die Komponenten über $\omega_2/2$ gespiegelt werden und so das Originalspektrum überlagern. Dies wird als „aliasing" bezeichnet, da das Originalspektrum (und damit auch das ursprüngliche Signal) aus einem in dieser Form gestörten Spektrum nicht mehr korrekt rekonstruiert werden kann.

$\omega_{\max}$ ist daher abhängig von der zur Diskretisierung verwendeten Abtastfrequenz ω_s in der Form

$$\omega_{\max} \leq \tfrac{1}{2}\omega_s \quad \text{bzw.} \quad \omega_s \geq 2\omega_{\max}. \tag{13.42}$$

Zur Diskretisierung eines kontinuierlichen Signals $g(x)$ mit Frequenzanteilen im Bereich $0 \leq \omega \leq \omega_{\max}$ benötigen wir daher eine Abtastfrequenz ω_s, die mindestens *doppelt so hoch* wie die maximale Signalfrequenz $\omega_{\max}$ ist. Wird diese Bedingung nicht eingehalten, dann überlappen sich die replizierten Spektralteile im Spektrum des abgetasteten Signals (Abb. 13.8 (c)) und das Spektrum wird verfälscht mit der Folge, dass das ursprüngliche Signal nicht mehr fehlerfrei aus dem Spektrum rekonstruiert werden kann. Dieser Effekt wird häufig als „aliasing" bezeichnet.[9]

Was wir soeben festgestellt haben, ist nichts anderes als die Kernaussage des berühmten Abtasttheorems von Shannon bzw. Nyquist (s. beispielsweise [16, S. 256]). Dieses besagt eigentlich, dass die Abtastfrequenz mindestens doppelt so hoch wie die *Bandbreite* des kontinuierlichen Signals sein muss, um Aliasing-Effekte zu vermeiden.[10] Wenn

[9] Das Wort „aliasing" wird auch im deutschen Sprachraum häufig verwendet, allerdings oft unrichtig ausgesprochen – die Betonung liegt auf der ersten Silbe.

[10] Die Tatsache, dass die *Bandbreite* (und nicht die Maximalfrequenz) eines Signals ausschlaggebend ist, mag zunächst erstaunen, denn sie erlaubt grundsätzlich die Abtastung (und korrekte Rekonstruktion) eines hochfrequenten – aber schmalbandigen – Signals mit einer relativ niedrigen Abtastfrequenz, die eventuell weit unter der maximalen Signalfrequenz liegt! Das ist deshalb möglich, weil man ja auch bei der Rekonstruktion des kontinuierlichen Signals wieder ein entsprechend schmalbandiges Filter verwenden kann. So kann es beispielsweise genügen, eine Kirchenglocke (ein sehr

man allerdings annimmt, dass das Frequenzbereich eines Signals bei null beginnt, dann sind natürlich Bandbreite und Maximalfrequenz ohnehin identisch.

13.2.2 Diskrete und periodische Funktionen

Nehmen wir an, unser ursprüngliches, kontinuierliches Signal $g(x)$ ist *periodisch* mit einer Periodendauer T. In diesem Fall besteht das zugehörige Fouriersprektrum $G(\omega)$ aus einer Folge dünner Spektrallinien, die gleichmäßig im Abstand von $\omega_0 = 2\pi/T$ angeordnet sind. Das Fourierspektrum einer periodischen Funktion kann also (wie bereits in Abschn. 13.1.2 erwähnt) als Fourierreihe dargestellt werden und ist somit *diskret*. Wird, im umgekehrten Fall, ein kontinuierliches Signal $g(x)$ in regelmäßigen Intervallen τ *abgetastet* (also diskretisiert), dann wird das zugehörige Fourierspektrum *periodisch* mit der Periodenlänge $\omega_s = 2\pi/\tau$.

Diskretisierung im Ortsraum führt also zu Periodizität im Spektralraum und umgekehrt. Abb. 13.9 zeigt diesen Zusammenhang und illustriert damit den Übergang von einer kontinuierlichen, nicht periodischen Funktion zu einer diskreten, periodischen Funktion, die schließlich als endlicher Vektor von Werten dargestellt und digital verarbeitet werden kann.

Das Fourierspektrum eines *kontinuierlichen*, nicht periodischen Signals $g(x)$ ist i. Allg. wieder kontinuierlich und nicht periodisch (Abb. 13.9 (a,b)). Ist das Signal $g(x)$ *periodisch*, wird das zugehörige Spektrum *diskret* (Abb. 13.9 (c,d)). Umgekehrt führt ein diskretes – aber nicht notwendigerweise periodisches – Signal zu einem periodischen Spektrum (Abb. 13.9 (e,f)). Ist das Signal schließlich diskret *und* periodisch mit einer Periodenlänge von M Abtastwerten, dann ist auch das zugehörige Spektrum diskret und periodisch mit M Werten (Abb. 13.9 (g,h)). Die Signale und Spektra in Abb. 13.9 sind übrigens nur zur Veranschaulichung gedacht und korrespondieren nicht wirklich.

13.3 Die diskrete Fouriertransformation (DFT)

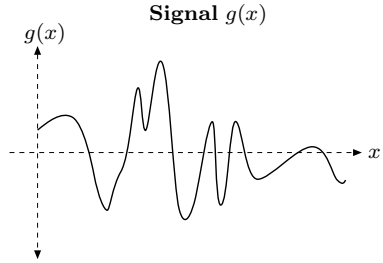
Im Fall eines diskreten, periodischen Signals benötigen wir also nur eine endliche Folge von M Abtastwerten, um sowohl das Signal $g(u)$ selbst als auch sein Fourierspektrum $G(m)$ vollständig abzubilden.[11] Durch die Darstellung als endliche Vektoren sind auch alle Voraussetzungen für die numerische Verarbeitung am Computer gegeben. Was uns jetzt noch fehlt ist eine Variante der Fouriertransformation für diskrete Signale.

schmalbandiges Schwingungssystem mit geringer Dämpfung) nur alle 5 Sekunden anzustoßen (bzw. „abzutasten“), um damit eine relativ hochfrequente Schallwelle eindeutig zu generieren.

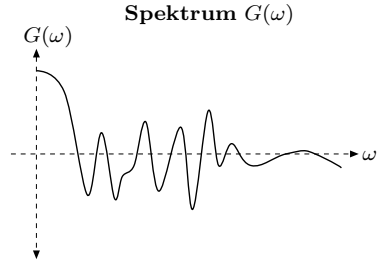
[11] Anm. zur Notation: Wir verwenden $g(x)$, $G(\omega)$ für ein *kontinuierliches* Signal oder Spektrum und $g(u)$, $G(m)$ für die *diskreten* Versionen.

Abbildung 13.9
Übergang von kontinuierlichen zu diskreten, periodischen Funktionen.

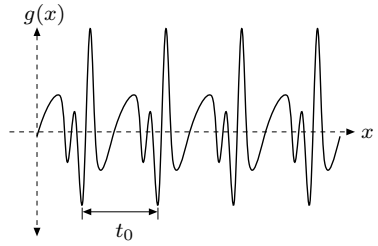
Signal $g(x)$ **Spektrum** $G(\omega)$



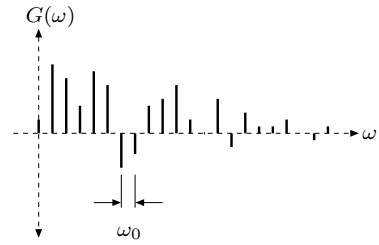
(a) Kontinuierliches, nicht periodisches Signal.



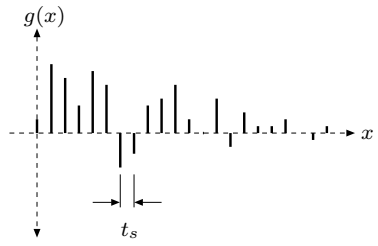
(b) Kontinuierliches, nicht periodisches Spektrum.



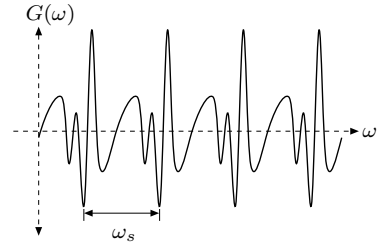
(c) Kontinuierliches, periodisches Signal mit Periodenlänge t_0.



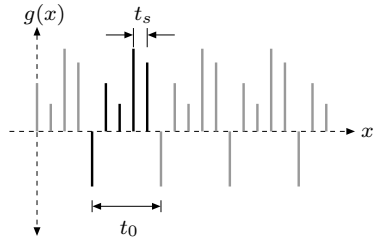
(d) Diskretes, nicht periodisches Spektrum mit Werten im Abstand $\omega_0 = 2\pi/t_0$.



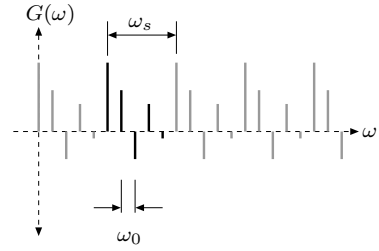
(e) Diskretes, nicht periodisches Signal mit Abtastwerten im Abstand t_s.



(f) Kontinuierliches, periodisches Spektrum mit der Periodenlänge $\omega_s = 2\pi/t_s$.



(g) Diskretes, periodisches Signal, abgetastet im Abstand t_s mit der Periodenlänge $t_0 = t_s M$.



(h) Diskretes, periodisches Spektrum mit Werten im Abstand $\omega_0 = 2\pi/t_0$ und Periodenlänge $\omega_s = 2\pi/t_s = \omega_0 M$.

13.3.1 Definition der DFT

Die diskrete Fouriertransformation ist, wie auch bereits die kontinuierliche FT, in beiden Richtungen identisch. Die Vorwärtstransformation (DFT) für ein diskretes Signal $g(u)$ der Länge M ($u = 0 \ldots M-1$) ist definiert als

$$\boxed{G(m) = \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g(u) \cdot e^{-\mathrm{i}2\pi \frac{mu}{M}} \quad \text{für } 0 \le m < M.} \tag{13.43}$$

Analog dazu ist die *inverse* Transformation (DFT^{-1})

$$\boxed{g(u) = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G(m) \cdot e^{\mathrm{i}2\pi \frac{mu}{M}} \quad \text{für } 0 \le u < M.} \tag{13.44}$$

Sowohl das Signal $g(u)$ wie auch das diskrete Spektrum $G(m)$ sind komplexwertige Vektoren der Länge M, d. h.

$$\begin{aligned} g(u) &= g_{\mathrm{Re}}(u) + \mathrm{i}\cdot g_{\mathrm{Im}}(u) \\ G(m) &= G_{\mathrm{Re}}(m) + \mathrm{i}\cdot G_{\mathrm{Im}}(m) \end{aligned} \tag{13.45}$$

für $u, m = 0 \ldots M-1$. Ein konkretes Beispiel der DFT mit $M = 10$ ist in Abb. 13.10 gezeigt.

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

Abbildung 13.10
Komplexwertige Vektoren. Bei der diskreten Fouriertransformation (DFT) sind das ursprüngliche Signal $g(u)$ und das zugehörige Spektrum $G(m)$ jeweils komplexwertige Vektoren der Länge M. Im konkreten Beispiel ist $M = 10$. Für die mit $*$ markierten Werte gilt $|G(m)| < 10^{-15}$.

Umgeformt aus der Euler'schen Schreibweise in Gl. 13.43 (s. auch Gl. 13.10) ergibt sich das diskrete Fourierspektrum in der Komponentennotation als

$$G(m) = \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} \Big[\underbrace{g_{\mathrm{Re}}(u) + \mathrm{i}\cdot g_{\mathrm{Im}}(u)}_{g(u)}\Big] \cdot \Big[\underbrace{\cos\big(2\pi \tfrac{mu}{M}\big)}_{\boldsymbol{C}_m^M(u)} - \mathrm{i}\cdot \underbrace{\sin\big(2\pi \tfrac{mu}{M}\big)}_{\boldsymbol{S}_m^M(u)}\Big], \tag{13.46}$$

wobei $\boldsymbol{C}_m^M$ und $\boldsymbol{S}_m^M$ diskrete Basisfunktionen (Kosinus- und Sinusfunktionen) bezeichnen, die im nachfolgenden Abschnitt näher beschrieben sind. Durch die gewöhnliche komplexe Multiplikation (s. Abschn. 1.2) erhalten wir aus Gl. 13.46 den Real- und Imaginärteil des diskreten Fourierspektrums in der Form

$$G_{\mathrm{Re}}(m) = \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g_{\mathrm{Re}}(u) \cdot \boldsymbol{C}_m^M(u) + g_{\mathrm{Im}}(u) \cdot \boldsymbol{S}_m^M(u) \tag{13.47}$$

$$G_{\mathrm{Im}}(m) = \frac{1}{\sqrt{M}} \sum_{u=0}^{M-1} g_{\mathrm{Im}}(u) \cdot \boldsymbol{C}_m^M(u) - g_{\mathrm{Re}}(u) \cdot \boldsymbol{S}_m^M(u) \tag{13.48}$$

für $m = 0 \ldots M{-}1$. Analog dazu ergibt sich der Real- bzw. Imaginärteil der *inversen* DFT aus Gl. 13.44 als

$$g_{\mathrm{Re}}(u) = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G_{\mathrm{Re}}(m) \cdot \boldsymbol{C}_u^M(m) - G_{\mathrm{Im}}(m) \cdot \boldsymbol{S}_u^M(m) \tag{13.49}$$

$$g_{\mathrm{Im}}(u) = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} G_{\mathrm{Im}}(m) \cdot \boldsymbol{C}_u^M(m) + G_{\mathrm{Re}}(m) \cdot \boldsymbol{S}_u^M(m) \tag{13.50}$$

für $u = 0 \ldots M{-}1$.

13.3.2 Diskrete Basisfunktionen

Die DFT (Gl. 13.44) beschreibt die Zerlegung einer diskreten Funktion $g(u)$ als endliche Summe diskreter Kosinus- und Sinusfunktionen ($\boldsymbol{C}^M$, $\boldsymbol{S}^M$) der Länge M, deren Gewichte oder „Amplituden“ durch die zugehörigen DFT-Koeffizienten $G(m)$ bestimmt werden. Jede dieser eindimensionalen *Basisfunktionen* (erstmals verwendet in Gl. 13.46)

$$\boldsymbol{C}_m^M(u) = \boldsymbol{C}_u^M(m) = \cos\bigl(2\pi \tfrac{mu}{M}\bigr), \tag{13.51}$$

$$\boldsymbol{S}_m^M(u) = \boldsymbol{S}_u^M(m) = \sin\bigl(2\pi \tfrac{mu}{M}\bigr) \tag{13.52}$$

ist eine Kosinus- bzw. Sinusfunktion mit einer diskreten Frequenz (Wellenzahl) m und einer Länge von M Abtastpunkten, ausgewertet an einer beliebigen Position u. Als Beispiel sind die Basisfunktionen für eine DFT der Länge $M = 8$ in Abb. 13.11–13.12 gezeigt, sowohl als diskrete Funktionen (mit ganzzahligen Ordinatenwerten $u \in \mathbb{Z}$) wie auch als kontinuierliche Funktionen (mit Ordinatenwerten $x \in \mathbb{R}$).

Für die Wellenzahl $m = 0$ hat die Kosinusfunktion $\boldsymbol{C}_0^M(u)$ (Gl. 13.51) den konstanten Wert 1. Daher spezifiziert der zugehörige DFT-Koeffizient $G_{\mathrm{Re}}(0)$ – also der Realteil von $G(0)$ – den konstanten Anteil des Signals oder, anders ausgedrückt, den durchschnittlichen Wert des Signals $g(u)$ in Gl. 13.49. Im Unterschied dazu ist der Wert von $\boldsymbol{S}_0^M(u)$ immer null und daher sind auch die zugehörigen Koeffizienten $G_{\mathrm{Im}}(0)$ in Gl. 13.49 bzw. $G_{\mathrm{Re}}(0)$ in Gl. 13.50 nicht relevant. Für ein reellwertiges

Signal (d. h. $g_{\mathrm{Im}}(u) = 0$ für alle u) muss also der Koeffizient $G_{\mathrm{Im}}(0)$ des zugehörigen Fourierspektrums ebenfalls null sein.

Wie wir aus Abb. 13.11 sehen, entspricht der Wellenzahl $m = 1$ eine Kosinus- bzw. Sinusfunktion, die über die Signallänge $M = 8$ exakt *einen* vollen Zyklus durchläuft. Eine Wellenzahl $m = 2 \ldots 7$ entspricht analog dazu $2 \ldots 7$ vollen Zyklen über die Signallänge hinweg (Abb. 13.11–13.12).

13.3.3 Schon wieder Aliasing!

Ein genauerer Blick auf Abb. 13.11 und 13.12 zeigt einen interessanten Sachverhalt: Die abgetasteten (diskreten) Kosinus- bzw. Sinusfunktionen für $m = 3$ und $m = 5$ sind *identisch*, obwohl die zugehörigen kontinuierlichen Funktionen unterschiedlich sind! Dasselbe gilt auch für die Frequenzpaare $m = 2, 6$ und $m = 1, 7$. Was wir hier sehen, ist die Manifestation des Abtasttheorems – das wir ursprünglich (Abschn. 13.2.1) im Frequenzraum beschrieben hatten – im *Ortsraum*. Offensichtlich ist also $m = 4$ die maximale Frequenzkomponente, die mittels eines diskreten Signals der Länge $M = 8$ beschrieben werden kann. Jede *höhere* Frequenzkomponente (in diesem Fall $m = 5 \ldots 7$) ist in der diskreten Version identisch zu einer anderen Komponente mit niedrigerer Wellenzahl und kann daher aus dem diskreten Signal nicht rekonstruiert werden!

Wenn ein kontinuierliches Signal im regelmäßigen Abstand τ abgetastet wird, wiederholt sich das zugehörige Spektrum an Vielfachen von $\omega_s = 2\pi/\tau$, wie bereits an früherer Stelle gezeigt (Abb. 13.8). Im diskreten Fall ist das Spektrum periodisch mit M. Weil das Fourierspektrum eines reellwertigen Signals um den Ursprung symmetrisch ist (Gl. 13.21), hat jede Spektralkomponente mit der Wellenzahl m ein gleich großes Duplikat mit der gegenüberliegenden Wellenzahl $-m$. Die Spektralkomponenten erscheinen also paarweise gespiegelt an Vielfachen von M, d. h.

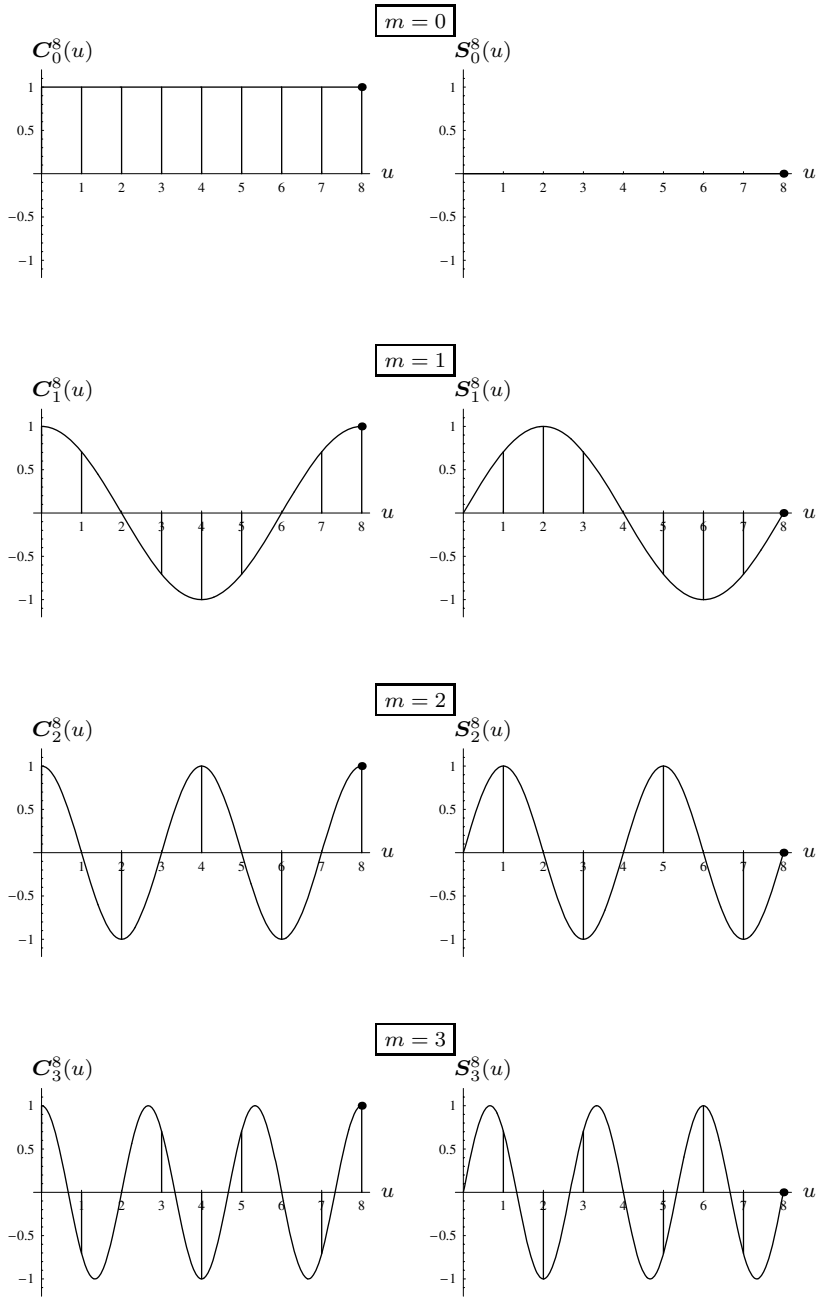
$$\begin{aligned} |G(m)| = |G(M-m)| &= |G(M+m)| \\ = |G(2M-m)| &= |G(2M+m)| \\ \ldots & \\ = |G(kM-m)| &= |G(kM+m)| \end{aligned} \tag{13.53}$$

für alle $k \in \mathbb{Z}$. Wenn also das ursprüngliche, kontinuierliche Signal Energie mit Frequenzen

$$\omega_m > \omega_{M/2}\,,$$

enthält, also Komponenten mit einer Wellenzahl $m > M/2$, dann überlagern (addieren) sich – entsprechend dem Abtasttheorem – die überlappenden Teile der replizierten Spektra im resultierenden, periodischen Spektrum des diskreten Signals.

Abbildung 13.11
Diskrete Basisfunktionen $\boldsymbol{C}_m^M(u)$ und $\boldsymbol{S}_m^M(u)$ für die Signallänge $M = 8$ und Wellenzahlen $m = 0 \ldots 3$. Jeder der Plots zeigt sowohl die diskreten Funktionswerte (durch runde Punkte markiert) wie auch die zugehörige kontinuierliche Funktion.



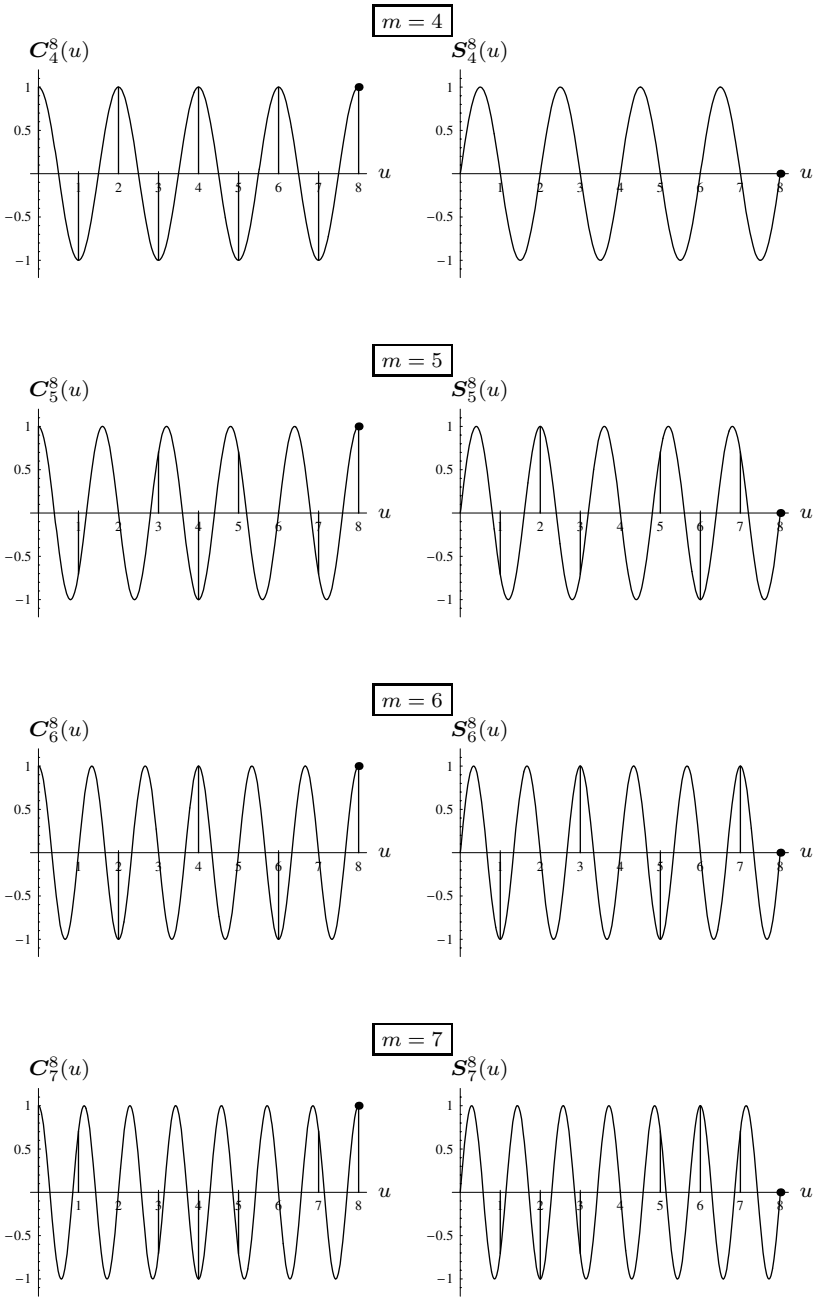


Abbildung 13.12
Diskrete Basisfunktionen (Fortsetzung). Signallänge $M = 8$ und Wellenzahlen $m = 4 \ldots 7$. Man beachte, dass z. B. die diskreten Funktionen für $m = 5$ und $m = 3$ (Abb. 13.11) identisch sind, weil $m = 4$ die maximale Wellenzahl ist, die in einem diskreten Spektrum der Länge $M = 8$ dargestellt werden kann.

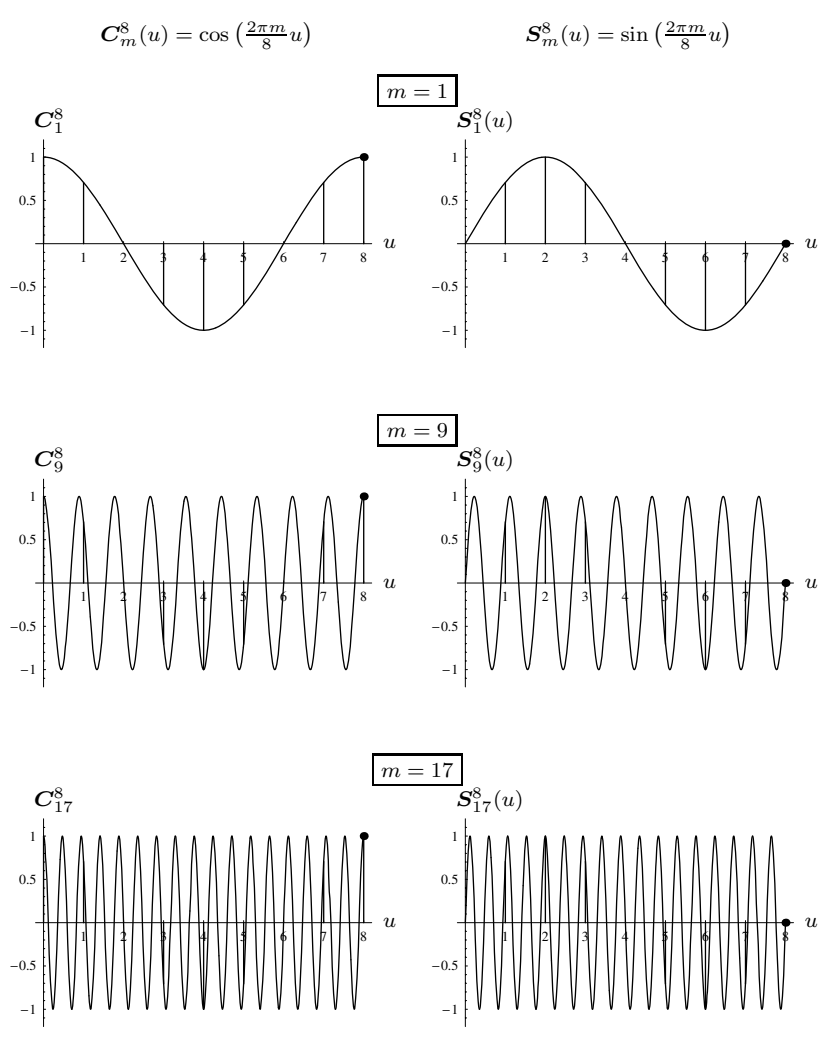


Abbildung 13.13
Aliasing im Ortsraum. Für die Signallänge $M = 8$ sind die diskreten Kosinus- und Sinusfunktionen für die Wellenzahlen $m = 1, 9, 17, \ldots$ (durch runde Punkte markiert) alle identisch. Die Abtastfrequenz selbst entspricht der Wellenzahl $m = 8$.

13.3.4 Einheiten im Orts- und Spektralraum

Das Verhältnis zwischen den Einheiten im Orts- und Spektralraum sowie die Interpretation der Wellenzahl m sind häufig Anlass zu Missverständnissen. Während sowohl das diskrete Signal wie auch das zugehörige Spektrum einfache Zahlenvektoren sind und zur Berechnung der DFT selbst keine Maßeinheiten benötigt werden, ist es dennoch wichtig, zu verstehen, in welchem Bezug die Koordinaten im Spektrum zu Größen in der realen Welt stehen.

Jeder komplexwertige Spektralkoeffizient $G(m)$ entspricht einem Paar von Kosinus- und Sinusfunktionen mit einer bestimmten Frequenz im Ortsraum. Angenommen ein kontinuierliches Signal wird an M aufeinander folgenden Positionen im Abstand τ (eine Zeitspanne oder eine Distanz im Raum) abgetastet. Die *Wellenzahl* $m = 1$ entspricht dann

der Grundperiode des diskreten Signals (das als periodisch angenommen wird) mit der Periodenlänge $M\tau$ und damit einer *Frequenz*

$$f_1 = \frac{1}{M\tau}\,. \tag{13.54}$$

Im Allgemeinen entspricht die Wellenzahl m eines diskreten Spektrums der realen Frequenz

$$f_m = m\frac{1}{M\tau} = m \cdot f_1 \tag{13.55}$$

für $0 \leq m < M$ oder – als Kreisfrequenz ausgedrückt –

$$\omega_m = 2\pi f_m = m\frac{2\pi}{M\tau} = m \cdot \omega_1. \tag{13.56}$$

Die Abtastfrequenz selbst, also $f_s = 1/\tau = M \cdot f_1$, entspricht offensichtlich der Wellenzahl $m_s = M$. Die maximale Wellenzahl, die im diskreten Spektrum ohne Aliasing dargestellt werden kann, ist

$$m_{\max} = \frac{M}{2} = \frac{m_s}{2}\,, \tag{13.57}$$

also wie erwartet die Hälfte der Wellenzahl der Abtastfrequenz m_s.

Beispiel 1: Zeitsignal

Nehmen wir beispielsweise an, $g(u)$ ist ein Zeitsignal (z. B. ein diskretes Tonsignal) bestehend aus $M = 500$ Abtastwerten im Intervall $\tau = 1\text{ms} = 10^{-3}\,\text{s}$. Die Abtastfrequenz ist daher $f_s = 1/\tau = 1000\,\text{Hertz}$ (Zyklen pro Sekunde) und die Gesamtdauer (Grundperiode) des Signals beträgt $M\tau = 0.5\,\text{s}$.

Aus Gl. 13.54 berechnen wir die Grundfrequenz des als periodisch angenommenen Signals als $f_1 = \frac{1}{500 \cdot 10^{-3}} = \frac{1}{0.5} = 2\,\text{Hertz}$. Die Wellenzahl $m = 2$ entspricht in diesem Fall einer realen Frequenz $f_2 = 2f_1 = 4\,\text{Hertz}$, $f_3 = 6\,\text{Hertz}$, usw. Die *maximale* Frequenz, die durch dieses diskrete Signal ohne Aliasing dargestellt werden kann, ist $f_{\max} = \frac{M}{2} f_1 = \frac{1}{2\tau} = 500\,\text{Hertz}$, also exakt die Hälfte der Abtastfrequenz f_s.

Beispiel 2: Signal im Ortsraum

Die gleichen Verhältnisse treffen auch für räumliche Signale zu, wenngleich mit anderen Maßeinheiten. Angenommen wir hätten ein eindimensionales Druckraster mit einer Auflösung (d. h. räumlichen Abtastfrequenz) von 120 Punkten pro cm, das entspricht etwa 300 *dots per inch* (dpi) und einer Signallänge von $M = 1800$ Abtastwerten. Dies entspricht einem räumlichen Abtastintervall von $\tau = 1/120\,\text{cm} \approx 83\,\mu\text{m}$ und einer Gesamtstrecke des Signals von $(1800/120)\,\text{cm} = 15\,\text{cm}$.

Die Grundfrequenz dieses (wiederum als periodisch angenommenen) Signals ist demnach $f_1 = \frac{1}{15}$, gemessen in Zyklen pro cm. Aus der Abtastfrequenz von $f_s = 120$ Zyklen pro cm ergibt sich eine maximale Signalfrequenz $f_{\max} = \frac{f_s}{2} = 60$ Zyklen pro cm und dies entspricht auch der feinsten Struktur, die mit diesem Druckraster aufgelöst werden kann.

13.3.5 Das Leistungsspektrum

Der Betrag des komplexwertigen Fourierspektrums

$$|G(m)| = \sqrt{G_{\mathrm{Re}}^2(m) + G_{\mathrm{Im}}^2(m)} \tag{13.58}$$

wird als *Leistungsspektrum* („power spectrum“) eines Signals bezeichnet. Es beschreibt die Energie (Leistung), die die einzelnen Frequenzkomponenten des Spektrums zum Signal beitragen. Das Leistungsspektrum ist reellwertig und positiv und wird daher häufig zur grafischen Darstellung der Fouriertransformierten verwendet (s. auch Abschn. 14.2).

Da die Phaseninformation im Leistungsspektrum verloren geht, kann das ursprüngliche Signal aus dem Leistungsspektrum allein nicht rekonstruiert werden. Das Leistungsspektrum ist jedoch – genau *wegen* der fehlenden Phaseninformation – unbeeinflusst von *Verschiebungen* des zugehörigen Signals und eignet sich daher zum Vergleich von Signalen. Genauer gesagt ist das Leistungsspektrum eines zyklisch verschobenen Signals identisch zum Leistungsspektrum des ursprünglichen Signals, d. h., für ein diskretes, periodisches Signal $g_1(u)$ der Länge M und das um den Abstand $d \in \mathbb{Z}$ zyklisch verschobene Signal

$$g_2(u) = g_1(u-d) \tag{13.59}$$

gilt für die zugehörigen Leistungsspektra

$$|G_2(m)| = |G_1(m)|, \tag{13.60}$$

obwohl die komplexwertigen Fourierspektra $G_1(m)$ und $G_2(m)$ selbst i. Allg. verschieden sind. Aufgrund der Symmetrieeigenschaft des Fourierspektrums (Gl. 13.53) gilt überdies

$$|G(m)| = |G(-m)| \tag{13.61}$$

für reellwertige Signale $g(u) \in \mathbb{R}$.

13.4 Implementierung der DFT

13.4.1 Direkte Implementierung

Auf Basis der Definitionen in Gl. 13.47 und Gl. 13.48 kann die DFT auf direktem Weg implementiert werden, wie in Prog. 13.1 gezeigt. Die dort angeführte Methode `DFT()` transformiert einen Signalvektor von beliebiger Länge M (nicht notwendigerweise eine Potenz von 2) und benötigt dafür etwa M^2 Operationen (Additionen und Multiplikationen), d. h., die Zeitkomplexität dieses DFT-Algorithmus beträgt $\mathcal{O}(M^2)$.

```
1 class Complex {
2     double re, im;
3
4     Complex(double re, double im) { //constructor method
5         this.re = re;
6         this.im = im;
7     }
8 }
```

```
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: [g_Re + i g_Im] · [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    }
```

Programm 13.1
Direkte Implementierung der DFT auf Basis der Definition in Gl. 13.47 und 13.48. Die Methode `DFT()` liefert einen komplexwertigen Ergebnisvektor der gleichen Länge wie der ebenfalls komplexwertige Input-Vektor `g`. Die Methode implementiert sowohl die Vorwärtstransformation wie auch die inverse Transformation, je nach Wert des Steuerparameters `forward`. Die Klasse `Complex` (oben) definiert die Struktur der komplexen Vektorelemente.

Eine Möglichkeit zur Verbesserung der Effizienz des DFT-Algorithmus ist die Verwendung von Lookup-Tabellen für die sin- und cos-Funktion (deren numerische Berechnung vergleichsweise aufwendig ist), da deren Ergebnisse ohnehin nur für M unterschiedliche Winkel φ_m benötigt werden. Für $m = 0 \ldots M-1$ sind die zugehörigen Winkel $\varphi_m = 2\pi\frac{m}{M}$ gleichförmig auf dem vollen 360°-Kreisbogen verteilt. Jedes ganzzahlige Vielfache $\varphi_m \cdot u$ (für $u \in \mathbb{Z}$) kann wiederum auf nur einen dieser Winkel fallen, denn es gilt

$$\varphi_m \cdot u = 2\pi\frac{mu}{M} \quad \equiv \quad \frac{2\pi}{M}(\underbrace{mu \bmod M}_{0 \le k < M}) = 2\pi\frac{k}{M} = \varphi_k \tag{13.62}$$

(mod ist der „Modulo“-Operator[12]). Wir können also zwei konstante Tabellen (Gleitkomma-Arrays) $\mathbf{W}_C[k]$ und $\mathbf{W}_S[k]$ der Größe M einrichten mit den Werten

[12] Siehe auch Anhang B.1.2.

$$\mathbf{W}_C[k] \leftarrow \cos(\omega_k) = \cos\left(2\pi\tfrac{k}{M}\right) \qquad (13.63)$$
$$\mathbf{W}_S[k] \leftarrow \sin(\omega_k) = \sin\left(2\pi\tfrac{k}{M}\right),$$

wobei $0 \le k < M$. Aus diesen Tabellen können die für die Berechnung der DFT notwendigen Kosinus- und Sinuswerte (Gl. 13.46) in der Form

$$\boldsymbol{C}_k^M(u) = \cos\left(2\pi\tfrac{mu}{M}\right) = \mathbf{W}_C[mu \bmod M] \qquad (13.64)$$
$$\boldsymbol{S}_k^M(u) = \sin\left(2\pi\tfrac{mu}{M}\right) = \mathbf{W}_S[mu \bmod M] \qquad (13.65)$$

ohne zusätzlichen Berechnungsvorgang für beliebige Werte von m und u ermittelt werden. Die entsprechende Modifikation der `DFT()`-Methode in Prog. 13.1 ist eine einfache Übung (Aufg. 13.5).

Trotz dieser deutlichen Verbesserung bleibt die direkte Implementierung der DFT rechenaufwendig. Tatsächlich war es lange Zeit unmöglich, die DFT in dieser Form auf gewöhnlichen Computern ausreichend schnell zu berechnen und dies gilt auch heute noch für viele konkrete Anwendungen.

13.4.2 Fast Fourier Transform (FFT)

Zur praktischen Berechnung der DFT existieren schnelle Algorithmen, in denen die Abfolge der Berechnungen so ausgelegt ist, dass gleichartige Zwischenergebnisse nur einmal berechnet und in optimaler Weise mehrfach wiederverwendet werden. Die sog. *Fast Fourier Transform*, von der es mehrere Varianten gibt, reduziert i. Allg. die Zeitkomplexität der Berechnung von $\mathcal{O}(M^2)$ auf $\mathcal{O}(M \log_2 M)$. Die Auswirkungen sind vor allem bei größeren Signallängen deutlich. Zum Beispiel bringt die FFT bei eine Signallänge $M = 10^3$ bereits eine Beschleunigung um den Faktor 100 und bei $M = 10^6$ um den Faktor 10.000, also ein eindrucksvoller Gewinn. Die FFT ist daher seit ihrer Erfindung ein unverzichtbares Werkzeug in praktisch jeder Anwendung der digitalen Spektralanalyse [11].

Die meisten FFT-Algorithmen, u. a. jener in der berühmten Publikation von Cooley und Tukey aus dem Jahr 1965 (ein historischer Überblick dazu findet sich in [30, S. 156]), sind auf Signallängen von $M = 2^k$, also Zweierpotenzen, optimiert. Spezielle FFT-Algorithmen wurden aber auch für andere Längen entwickelt, insbesondere für eine Reihe kleinerer Primzahlen [7], die wiederum zu FFTs unterschiedlichster Größe zusammengesetzt werden können.

Wichtig ist jedenfalls zu wissen, dass DFT und FFT *dasselbe* Ergebnis berechnen und die FFT nur eine spezielle – wenn auch äußerst geschickte – *Methode* zur Implementierung der diskreten Fouriertransformation (Gl. 13.43) ist.

13.5 Aufgaben

Aufg. 13.1. Berechnen Sie die Werte der Kosinusfunktion $f(x) = \cos(\omega x)$ mit der Kreisfrequenz $\omega = 5$ für die Positionen $x = -3, -2, \ldots, 2, 3$. Welche Periodenlänge hat diese Funktion?

Aufg. 13.2. Ermitteln Sie den Phasenwinkel φ der Funktion $f(x) = A\cos(\omega x) + B\sin(\omega x)$ für $A = -1$ und $B = 2$.

Aufg. 13.3. Berechnen Sie Real- und Imaginärteil sowie den Betrag der komplexen Größe $z = 1.5 \cdot e^{-\mathrm{i}\,2.5}$.

Aufg. 13.4. Ein eindimensionaler, optischer Scanner zur Abtastung von Filmen soll Bildstrukturen mit einer Genauigkeit von 4.000 dpi (*dots per inch*) auflösen. In welchem räumlichen Abstand (in mm) müssen die Abtastwerte angeordnet sein, sodass kein *Aliasing* auftritt?

Aufg. 13.5. Modifizieren Sie die Implementierung der eindimensionalen DFT in Prog. 13.1 durch Verwendung von Lookup-Tabellen für die cos- und sin-Funktion, wie in Gl. 13.64 und 13.65 beschrieben.

14

Diskrete Fouriertransformation in 2D

Die Fouriertransformation ist nicht nur für eindimensionale Signale definiert, sondern für Funktionen beliebiger Dimension, und daher sind auch zweidimensionale Bilder aus mathematischer Sicht nichts Besonderes.

14.1 Definition der 2D-DFT

Für eine zweidimensionale, periodische Funktion (also z. B. ein Intensitätsbild) $g(u, v)$ der Größe $M \times N$ ist die diskrete Fouriertransformation (2D-DFT) definiert als

$$\begin{aligned} G(m,n) &= \frac{1}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u,v) \cdot e^{-\mathrm{i}2\pi \frac{mu}{M}} \cdot e^{-\mathrm{i}2\pi \frac{nv}{N}} \\ &= \frac{1}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u,v) \cdot e^{-\mathrm{i}2\pi (\frac{mu}{M} + \frac{nv}{N})} \end{aligned} \tag{14.1}$$

für die Spektralkoordinaten $m = 0 \ldots M-1$ und $n = 0 \ldots N-1$. Die resultierende Fouriertransformierte ist also ebenfalls wieder eine zweidimensionale Funktion mit derselben Größe $(M \times N)$ wie das ursprüngliche Signal. Analog dazu ist die *inverse* 2D-DFT definiert als

$$\begin{aligned} g(u,v) &= \frac{1}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(m,n) \cdot e^{\mathrm{i}2\pi \frac{um}{M}} \cdot e^{\mathrm{i}2\pi \frac{vn}{N}} \\ &= \frac{1}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(m,n) \cdot e^{\mathrm{i}2\pi (\frac{um}{M} + \frac{vn}{N})} \end{aligned} \tag{14.2}$$

für die Bildkoordinaten $u = 0 \ldots M-1$ und $v = 0 \ldots N-1$.

14.1.1 2D-Basisfunktionen

Gl. 14.2 zeigt, dass eine zweidimensionale Funktion $g(u,v)$ als Linearkombination (d. h. als gewichtete Summe) zweidimensionaler, komplexwertiger Funktionen der Form

$$e^{\mathrm{i}2\pi(\frac{um}{M}+\frac{vn}{N})} = \underbrace{\cos\left[2\pi\left(\frac{um}{M}+\frac{vn}{N}\right)\right]}_{\boldsymbol{C}^{M,N}_{m,n}(u,v)} + \mathrm{i}\cdot\underbrace{\sin\left[2\pi\left(\frac{um}{M}+\frac{vn}{N}\right)\right]}_{\boldsymbol{S}^{M,N}_{m,n}(u,v)} \quad (14.3)$$

dargestellt werden kann. Dabei sind $\boldsymbol{C}^{M,N}_{m,n}(u,v)$ und $\boldsymbol{S}^{M,N}_{m,n}(u,v)$ zweidimensionale Kosinus- bzw. Sinusfunktionen mit horizontaler Wellenzahl m und vertikaler Wellenzahl n:

$$\boldsymbol{C}^{M,N}_{m,n}(u,v) = \cos\left[2\pi\left(\frac{um}{M}+\frac{vn}{N}\right)\right] \quad (14.4)$$

$$\boldsymbol{S}^{M,N}_{m,n}(u,v) = \sin\left[2\pi\left(\frac{um}{M}+\frac{vn}{N}\right)\right] \quad (14.5)$$

Beispiele

Die Abbildungen 14.1–14.2 zeigen einen Satz von 2D-Kosinusfunktionen $\boldsymbol{C}^{M,N}_{m,n}$ der Größe $M = N = 16$ für verschiedene Kombinationen von Wellenzahlen $m, n = 0 \ldots 3$. Wie klar zu erkennen ist, entsteht in jedem Fall eine gerichtete, kosinusförmige Wellenform, deren Richtung durch die Wellenzahlen m und n bestimmt ist. Beispielsweise entspricht den Wellenzahlen $m = n = 2$ eine Kosinusfunktion $\boldsymbol{C}^{M,N}_{2,2}(u,v)$, die jeweils zwei volle Perioden in horizontaler und in vertikaler Richtung durchläuft und dadurch eine zweidimensionale Welle in diagonaler Richtung erzeugt. Gleiches gilt natürlich auch für die entsprechenden Sinusfunktionen.

14.1.2 Implementierung der zweidimensionalen DFT

Wie im eindimensionalen Fall könnte man auch die 2D-DFT direkt auf Basis der Definition in Gl. 14.1 implementieren, aber dies ist nicht notwendig. Durch geringfügige Umformung von Gl. 14.1 in der Form

$$G(m,n) = \frac{1}{\sqrt{N}}\sum_{v=0}^{N-1}\Bigg[\underbrace{\frac{1}{\sqrt{M}}\sum_{u=0}^{M-1} g(u,v)\cdot e^{-\mathrm{i}2\pi\frac{um}{M}}}_{\text{1-dim. DFT der Zeile } g(\cdot,v)}\Bigg]\cdot e^{-\mathrm{i}2\pi\frac{vn}{N}} \quad (14.6)$$

wird deutlich, dass sich im Kern wiederum eine *eindimensionale* DFT (s. Gl. 13.43) des v-ten Zeilenvektors $g(\cdot,v)$ befindet, die unabhängig ist von den „vertikalen“ Größen v und N (die in Gl. 14.6 außerhalb der eckigen Klammern stehen). Wenn also im ersten Schritt jeder Zeilenvektor $g(\cdot,v)$ des ursprünglichen Bilds ersetzt wird durch seine (eindimensionale) Fouriertransformierte, d. h.

Algorithmus 14.1
Implementierung der zweidimensionalen DFT als Folge von eindimensionalen DFTs über Zeilen- bzw. Spaltenvektoren.

```
1: Separable 2D-DFT (g(u,v) ∈ ℂ)                    ▷ 0 ≤ u < M, 0 ≤ v < N
2:     for v ← 0 ... N–1 do
3:         Let g(·,v) be the v^th row vector of g:
               Replace g(·,v) by DFT(g(·,v)).
4:     for u ← 0 ... M–1 do
5:         Let g(u,·) be the u^th column vector of g:
               Replace g(u,·) by DFT(g(u,·)).
6:     Remark:
           g(u,v) = G(u,v) ∈ ℂ now contains the discrete 2D spectrum.
```

$$g'(\cdot, v) \leftarrow \mathsf{DFT}\big(g(\cdot, v)\big) \quad \text{für } 0 \le v < N,$$

dann muss nachfolgend nur mehr die eindimensionale DFT für jeden (vertikalen) Spaltenvektor berechnet werden, also

$$g''(u, \cdot) \leftarrow \mathsf{DFT}\big(g'(u, \cdot)\big) \quad \text{für } 0 \le u < M.$$

Das Resultat $g''(u, v)$ entspricht der zweidimensionalen Fouriertransformierten $G(m, n)$. Die *zwei*dimensionale DFT ist also, wie in Alg. 14.1 zusammengefasst, in zwei aufeinander folgende *ein*dimensionale DFTs über die Zeilen- bzw. Spaltenvektoren *separierbar*. Das bedeutet einerseits einen Effizienzvorteil und andererseits, dass wir auch zur Realisierung mehrdimensionaler DFTs ausschließlich eindimensionale DFT-Implementierungen (bzw. die eindimensionale FFT) verwenden können.

Wie aus Gl. 14.6 abzulesen ist, könnte diese Operation genauso gut in umgekehrter Reihenfolge durchgeführt werden, also beginnend mit einer DFT über alle Spalten und dann erst über die Zeilen. Bemerkenswert ist überdies, dass alle Operationen in Alg. 14.1 „in place“ ausgeführt werden können, d. h., das ursprüngliche Signal $g(u, v)$ wird destruktiv modifiziert und schrittweise durch seine Fouriertransformierte $G(m, n)$ derselben Größe ersetzt, ohne dass dabei zusätzlicher Speicherplatz angelegt werden müsste. Das ist durchaus erwünscht und üblich, zumal auch praktisch alle eindimensionalen FFT-Algorithmen – die man nach Möglichkeit zur Implementierung der DFT verwenden sollte – „in place“ arbeiten.

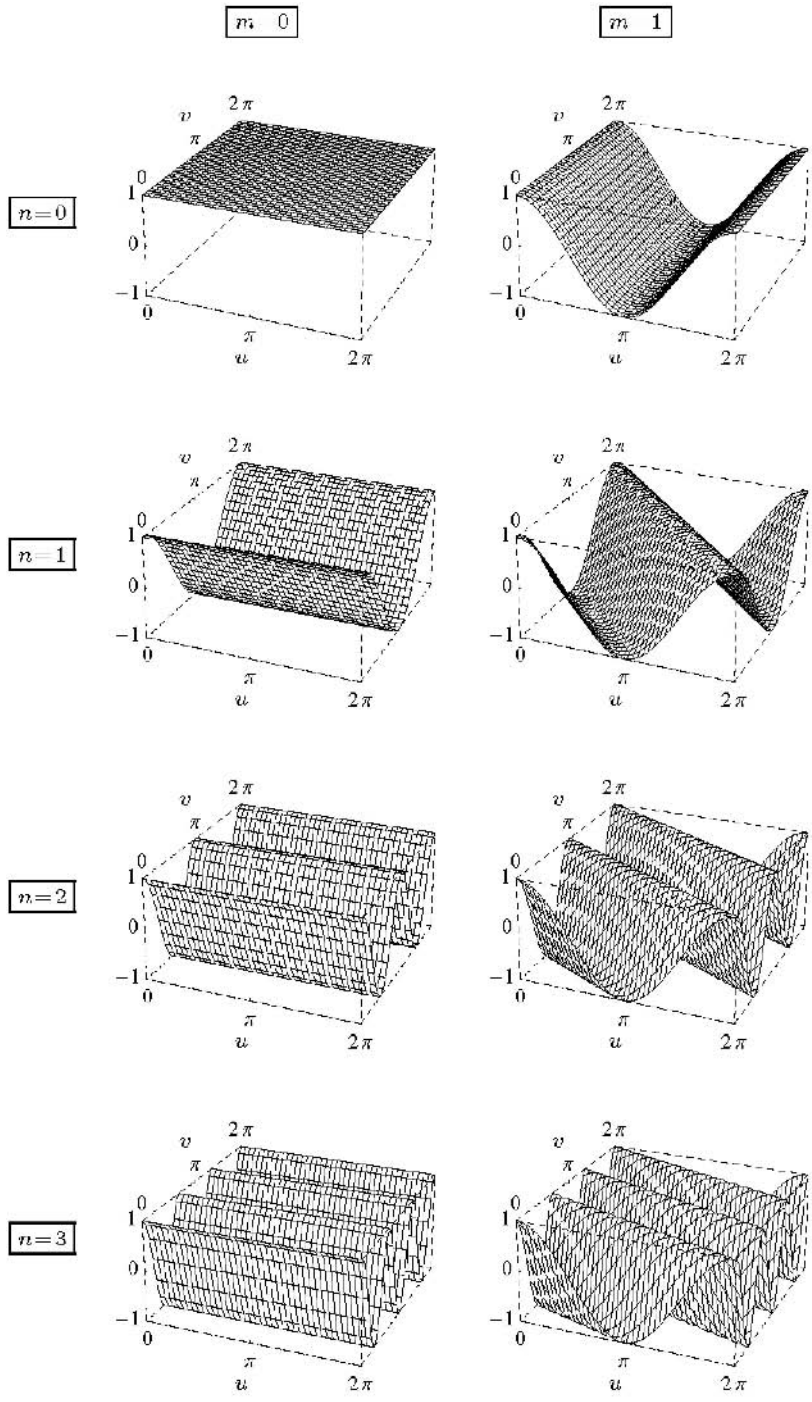
14.2 Darstellung der Fouriertransformierten in 2D

Zur Darstellung von zweidimensionalen, komplexwertigen Funktionen, wie die Ergebnisse der 2D-DFT, gibt es leider keine einfache Methode. Man könnte die Real- und Imaginärteile als Intensitätsbild oder als Oberflächengrafik darstellen, üblicherweise betrachtet man jedoch den Betrag der komplexen Funktion, im Fall der Fouriertransformierten also das Leistungsspektrum $|G(m, n)|$ (s. Abschn. 13.3.5).

14.2.1 Wertebereich

In den meisten natürlichen Bildern konzentriert sich die „spektrale Energie“ in den niedrigen Frequenzen mit einem deutlichen Maximum bei den

Abbildung 14.1
Zweidimensionale Kosinusfunktionen. $\boldsymbol{C}_{m,n}^{M,N}(u,v) = \cos\left[2\pi\left(\frac{um}{M} + \frac{vn}{N}\right)\right]$ für $M = N = 16$, $n = 0 \ldots 3$, $m = 0, 1$.



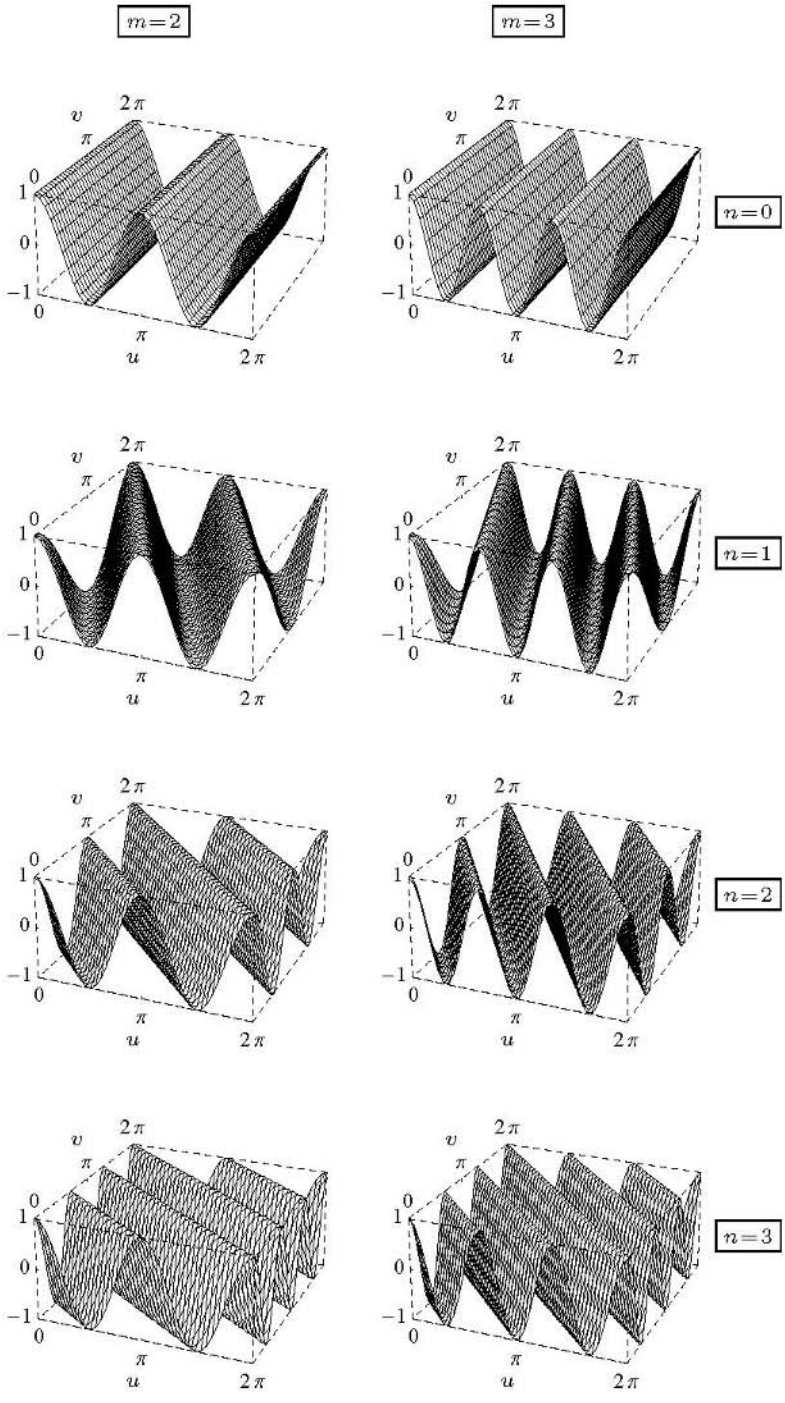


Abbildung 14.2
Zweidimensionale Kosinusfunktionen (*Fortsetzung*).
$\boldsymbol{C}_{m,n}^{M,N}(u,v) = \cos\left[2\pi\left(\frac{um}{M} + \frac{vn}{N}\right)\right]$ für $M = N = 16$, $n = 0 \ldots 3$, $m = 2, 3$.

Wellenzahlen $(0, 0)$, also am Koordinatenursprung (s. auch Abschn. 14.4). Um den hohen Wertebereich innerhalb des Spektrums und insbesondere die kleineren Werte an der Peripherie des Spektrums sichtbar zu machen, wird häufig die Quadratwurzel $\sqrt{|G(m,n)|}$ oder der Logarithmus $\log|G(m,n)|$ des Leistungsspektrums für die Darstellung verwendet.

14.2.2 Zentrierte Darstellung

Wie im eindimensionalen Fall ist das diskrete 2D-Spektrum eine periodische Funktion, d. h.

$$G(m,n) = G(m + pM, n + qN) \tag{14.7}$$

für beliebige $p, q \in \mathbb{Z}$, und bei reellwertigen 2D-Signalen ist das Leistungsspektrum (vgl. Gl. 13.53) überdies um den Ursprung symmetrisch, also

$$|G(m,n)| = |G(-m,-n)|. \tag{14.8}$$

Es ist daher üblich, den Koordinatenursprung $(0, 0)$ des Spektrums *zentriert* darzustellen, mit den Koordinaten m, n im Bereich

$$-\left\lfloor \tfrac{M}{2} \right\rfloor \le m \le \left\lfloor \tfrac{M-1}{2} \right\rfloor \quad \text{bzw.} \quad -\left\lfloor \tfrac{N}{2} \right\rfloor \le n \le \left\lfloor \tfrac{N-1}{2} \right\rfloor .$$

Wie in Abb. 14.3 gezeigt, kann dies durch einfaches Vertauschen der vier Quadranten der Fouriertransformierten durchgeführt werden. In der resultierenden Darstellung finden sich damit die Koeffizienten für die niedrigsten Wellenzahlen im Zentrum, und jene für die höchsten Wellenzahlen liegen an den Rändern. Abb. 14.4 zeigt die Darstellung des

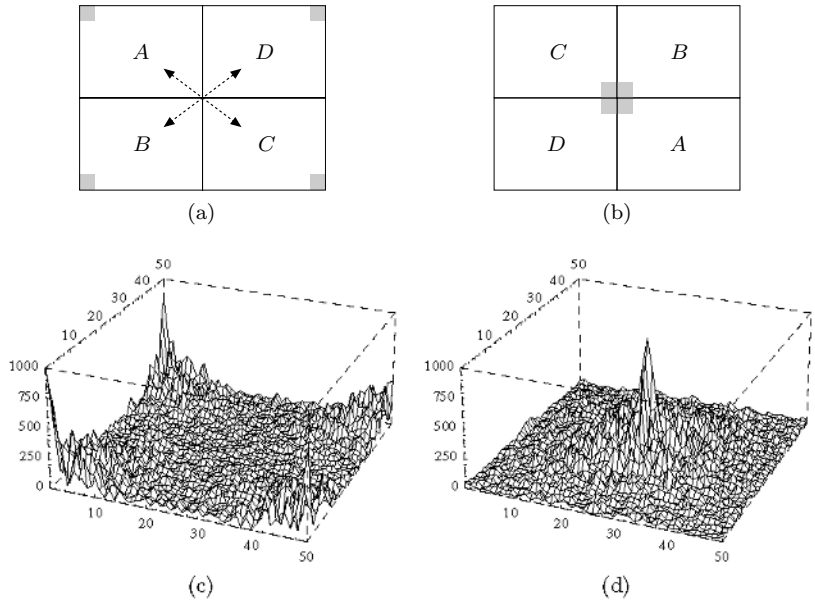
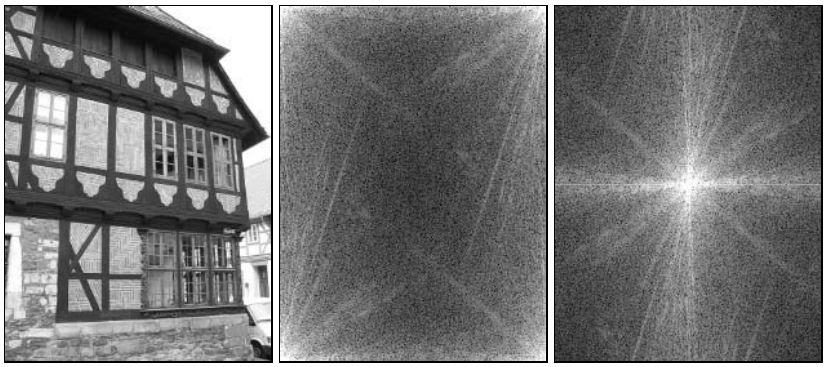


Abbildung 14.3
Zentrierung der 2D-Fourierspektrums. Im ursprünglichen Ergebnis der 2D-DFT liegt der Koordinatenursprung (und damit der Bereich niedriger Frequenzen) links oben und – aufgrund der Periodizität des Spektrums – gleichzeitig auch an den übrigen Eckpunkten (a). Die Koeffizienten der höchsten Wellenzahlen liegen hingegen im Zentrum. Durch paarweises Vertauschen der vier Quadranten werden der Koordinatenursprung und die niedrigen Wellenzahlen ins Zentrum verschoben, umgekehrt kommen die hohen Wellenzahlen an den Rand (b). Konkretes 2D-Fourierspektrum in ursprünglicher Darstellung (c) und zentrierter Darstellung (d).



(a) (b) (c)

Abbildung 14.4
Darstellung des 2D-Leistungsspektrums als Intensitätsbild. Originalbild (a), unzentriertes Spektrum (b) und zentrierte Darstellung (c).

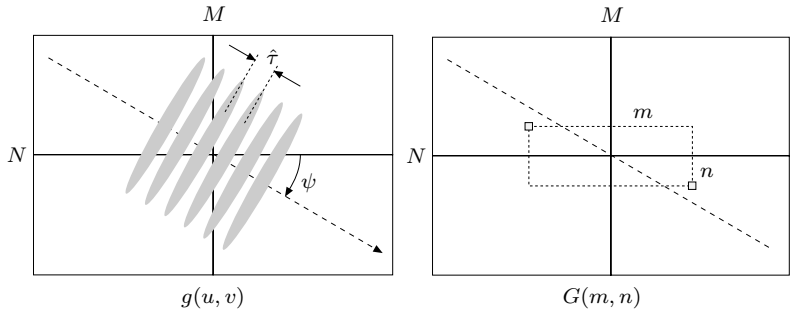


Abbildung 14.5
Frequenz und Orientierung im 2D-Spektrum. Das Bild (links) enthält ein periodisches Muster mit der effektiven Frequenz $\hat{f} = 1/\hat{\tau}$ mit der Richtung ψ. Der zu diesem Muster gehörende Koeffizient im Leistungsspektrum (rechts) befindet sich an der Position $(m, n) = \pm\hat{f} \cdot (M \cos\psi, N \sin\psi)$. Die Lage der Spektralkoordinaten (m, n) gegenüber dem Ursprung entspricht daher i. Allg. *nicht* der Richtung des Bildmusters.

2D-Leistungsspektrums als Intensitätsbild in der ursprünglichen und in der (üblichen) zentrierten Form, wobei die Intensität dem Logarithmus der Spektralwerte ($\log_{10}|G(m,n)|$) entspricht.

14.3 Frequenzen und Orientierung in 2D

Wie aus Abb. 14.1–14.2 hervorgeht, sind die Basisfunktionen gerichtete Kosinus- bzw. Sinusfunktionen, deren Orientierung und Frequenz durch die Wellenzahlen m und n (für die horizontale bzw. vertikale Richtung) bestimmt sind. Wenn wir uns entlang der Hauptrichtung einer solchen Basisfunktion bewegen (d. h. rechtwinklig zu den Wellenkämmen), erhalten wir eine eindimensionale Kosinus- bzw. Sinusfunktion mit einer bestimmten Frequenz $\hat{f}$, die wir als *gerichtete* oder *effektive* Frequenz der Wellenform bezeichnen (Abb. 14.5).

14.3.1 Effektive Frequenz

Wir erinnern uns, dass die Wellenzahlen m, n definieren, wie viele volle Perioden die zugehörige 2D-Basisfunktion innerhalb von M Einheiten in horizontaler Richtung bzw. innerhalb von N Einheiten in vertikaler

Richtung durchläuft. Die effektive Frequenz entlang der Wellenrichtung kann aus dem eindimensionalen Fall (Gl. 13.54) abgeleitet werden als

$$\hat{f}_{(m,n)} = \frac{1}{\tau}\sqrt{\left(\frac{m}{M}\right)^2 + \left(\frac{n}{N}\right)^2}, \tag{14.9}$$

wobei das gleiche räumliche Abtastintervall für die x- und y-Richtung angenommen wird, d. h. $\tau = \tau_x = \tau_y$. Die maximale Signalfrequenz entlang der x- und y-Achse beträgt daher

$$\hat{f}_{(\pm\frac{M}{2},0)} = \hat{f}_{(0,\pm\frac{N}{2})} = \tfrac{1}{\tau}\sqrt{\left(\tfrac{1}{2}\right)^2} = \tfrac{1}{2\tau} = \tfrac{1}{2}f_s, \tag{14.10}$$

wobei $f_s = \frac{1}{\tau}$ die Abtastfrequenz bezeichnet. Man beachte, dass die effektive Frequenz für die Eckpunkte des Spektrums, also

$$\hat{f}_{(\pm\frac{M}{2},\pm\frac{N}{2})} = \tfrac{1}{\tau}\sqrt{\left(\tfrac{1}{2}\right)^2 + \left(\tfrac{1}{2}\right)^2} = \tfrac{1}{\sqrt{2}\cdot\tau} = \tfrac{1}{\sqrt{2}}f_s, \tag{14.11}$$

um den Faktor $\sqrt{2}$ *höher* ist als entlang der beiden Koordinatenachsen (Gl. 14.10).

14.3.2 Frequenzlimits und Aliasing in 2D

Abb. 14.6 illustriert den in Gl. 14.10 und 14.11 beschriebenen Zusammenhang. Die maximal zulässigen Signalfrequenzen in jeder Richtung liegen am Rand des zentrierten, $M \times N$ großen 2D-Spektrums. Jedes Signal mit Komponenten ausschließlich innerhalb dieses Bereichs entspricht den Regeln des Abtasttheorems und kann ohne Aliasing rekonstruiert werden. Jede Spektralkomponente außerhalb dieser Grenze wird an dieser Grenze zum Ursprung hin in den inneren Bereich des Spektrums (also auf niedrigere Frequenzen) gespiegelt und verursacht daher sichtbares *Aliasing* im rekonstruierten Bild.

Offensichtlich ist die effektive Abtastfrequenz (Gl. 14.9) am niedrigsten in Richtung der beiden Koordinatenachsen des Abtastgitters. Um sicherzustellen, dass ein bestimmtes Bildmuster in jeder Lage (Rotation) ohne Aliasing abgebildet wird, muss die effektive Signalfrequenz $\hat{f}$ des Bildmusters in jeder Richtung auf $\frac{f_s}{2} = \frac{1}{2\tau}$ begrenzt sein, wiederum unter der Annahme, dass das Abtastintervall τ in beiden Achsenrichtungen identisch ist.

14.3.3 Orientierung

Die räumliche Richtung einer zweidimensionalen Kosinus- oder Sinuswelle mit den Spektralkoordinaten m, n ($0 \le m < M$, $0 \le n < N$) ist

$$\psi_{(m,n)} = \arctan_2\left(\tfrac{n}{N}, \tfrac{m}{M}\right) = \arctan_2\big(nM, mN\big), \tag{14.12}$$

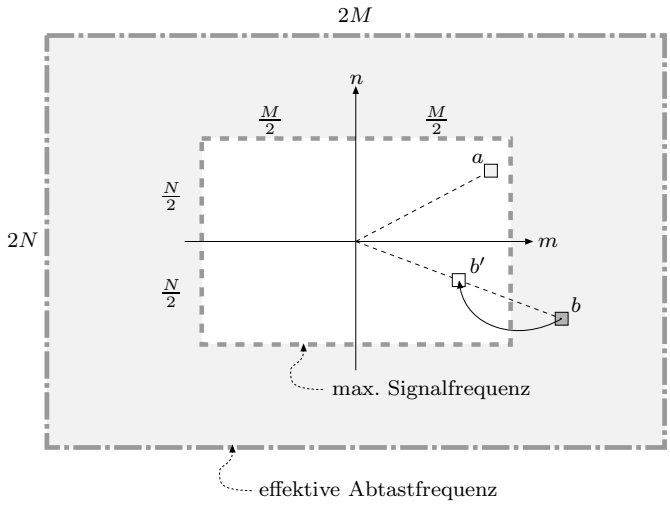


Abbildung 14.6
Maximale Signalfrequenzen und Aliasing in 2D. Der Rand des $M \times N$ großen 2D-Spektrums (inneres Rechteck) markiert die maximal zulässigen Signalfrequenzen für jede Richtung. Das äußere Rechteck bezeichnet die Lage der effektiven Abtastfrequenz, das ist jeweils das Doppelte der maximalen Signalfrequenz für dieselbe Richtung. Die Signalkomponente mit der Spektralposition a liegt innerhalb des maximal darstellbaren Frequenzbereichs und verursacht daher kein *Aliasing*. Im Gegensatz dazu ist die Komponente b außerhalb des zulässigen Bereichs und wird daher an der Grenzlinie auf eine Position b' („Alias“) mit niedrigeren Frequenzen gespiegelt.

wobei $\psi_{(m,n)}$ für $m = n = 0$ natürlich unbestimmt ist.[1] Umgekehrt wird ein zweidimensionales Sinusoid mit effektiver Frequenz $\hat{f}$ und Richtung ψ durch die Spektralkoordinaten

$$(m, n) = \pm \hat{f} \cdot (M \cos\psi, N \sin\psi) \tag{14.13}$$

repräsentiert, wie bereits in Abb. 14.5 dargestellt.

14.3.4 Geometrische Korrektur des 2D-Spektrums

Aus Gl. 14.13 ergibt sich, dass im speziellen Fall einer Sinus-/Kosinuswelle mit Orientierung $\psi = 45°$ die zugehörigen Spektralkoeffizienten an den Koordinaten

$$(m, n) = \pm(\lambda M, \lambda N) \quad \text{für } -\tfrac{1}{2} \le \lambda \le +\tfrac{1}{2} \tag{14.14}$$

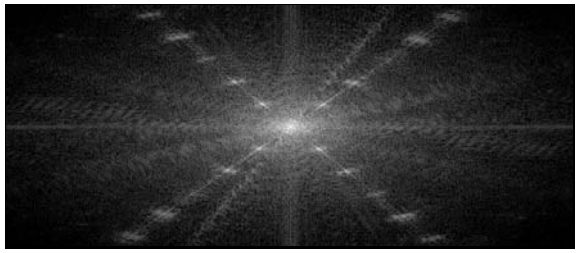
(s. Gl. 14.11) zu finden sind, d. h. auf der Diagonale des Spektrums. Sofern das Bild (und damit auch das Spektrum) nicht quadratisch ist (d. h. $M = N$), sind die Richtungswinkel im Bild und im Spektrum nicht identisch, fallen aber in Richtung der Koordinatenachsen jeweils zusammen. Dies bedeutet, dass bei der Rotation eines Bildmusters um einen Winkel α das Spektrum zwar in der gleichen Richtung gedreht wird, aber i. Allg. *nicht* um denselben Winkel α!

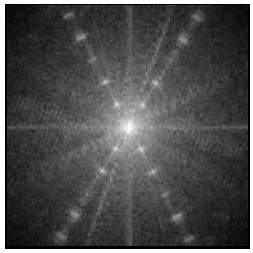
Um Orientierungen und Drehwinkel im Bild und im Spektrum identisch erscheinen zu lassen, genügt es, das Spektrum auf *quadratische* Form zu skalieren, sodass die spektrale Auflösung entlang beider Frequenzachsen die gleiche ist (wie in Abb. 14.7 gezeigt).

[1] $\arctan_2(y, x)$ in Gl. 14.12 steht für die inverse Tangensfunktion $\tan^{-1}(y/x)$ (s. auch Anhang B.1.6).



(a)





(b) (c)

Abbildung 14.7
Geometrische Korrektur des 2D-Spektrums. Ausgangsbild (a) mit dominanten, gerichteten Bildmustern, die im zugehörigen Spektrum (b) als deutliche Spitzen sichtbar werden. Weil Bild und Spektrum nicht quadratisch sind ($M \neq N$), stimmen die Orientierungen im ursprünglichen Spektrum (b) nicht mit denen im Bild überein. Erst wenn das Spektrum auf quadratische Form skaliert ist (c), wird deutlich, dass die Zylinder dieses Motors (*V-Rod Engine* von Harley-Davidson) tatsächlich im 60°-Abstand angeordnet sind.

14.3.5 Auswirkungen der Periodizität

Bei der Interpretation der 2D-DFT von Bildern muss man sich der Tatsache bewusst sein, dass die Signalfunktion bei der diskreten Fouriertransformation implizit und in jeder Koordinatenrichtung als periodisch angenommen wird. Die Übergänge an den Bildrändern, also von einer Periode zur nächsten, gehören daher genauso zum Signal wie jedes Ereignis innerhalb des eigentlichen Bilds. Ist der Intensitätsunterschied zwischen gegenüberliegenden Randpunkten groß (wie z. B. zwischen dem oberen und dem unteren Rand einer Landschaftsaufnahme), dann führt dies zu abrupten Übergängen in dem als periodisch angenommenen Signal. Steile Diskontinuitäten sind aber von hoher Bandbreite, d. h., die zugehörige Signalenergie ist im Fourierspektrum über viele Frequenzen entlang der Koordinatenachsen des Abtastgitters verteilt (siehe Abb. 14.8). Diese breitbandige Energieverteilung entlang der Hauptachsen, die bei realen Bildern häufig zu beobachten ist, kann dazu führen, dass andere, signalrelevante Komponenten völlig überdeckt werden.

14.3.6 *Windowing*

Eine Lösung dieses Problems besteht in der Multiplikation der Bildfunktion $g(u,v) = I(u,v)$ mit einer geeigneten Fensterfunktion (*windowing function*) $w(u,v)$ in der Form

$$\tilde{g}(u,v) = g(u,v) \cdot w(u,v),$$

für $0 \leq u < M$, $0 \leq v < N$, *vor* der Berechnung der DFT. Die Fensterfunktion $w(u,v)$ soll zu den Bildrändern hin möglichst kontinuierlich



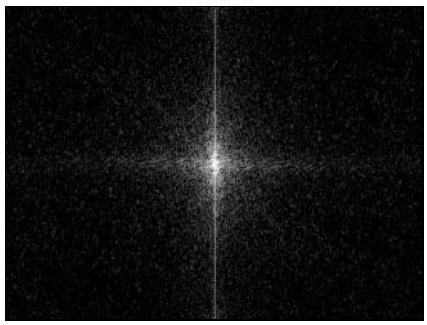


Abbildung 14.8
Auswirkungen der Periodizität im 2D-Spektrum. Die Berechnung der diskreten Fouriertransformation erfolgt unter der impliziten Annahme, dass das Bildsignal in beiden Dimensionen periodisch ist (oben). Größere Intensitätsunterschiede zwischen gegenüberliegenden Bildrändern – hier besonders deutlich in der vertikalen Richtung – führen zu breitbandigen Signalkomponenten, die hier im Spektrum (unten) als helle Linie entlang der vertikalen Achse sichtbar werden.

auf null abfallen und damit die Diskontinuitäten an den Übergängen zwischen einzelnen Perioden der Signalfunktion eliminieren. Die Multiplikation mit $w(u,v)$ hat jedoch weitere Auswirkungen auf das Fourierspektrum, denn entsprechend der Faltungseigenschaft entspricht – wie wir bereits (aus Gl. 13.26) wissen – die *Multiplikation* im Ortsraum einer *Faltung* der zugehörigen Spektra:

$$\tilde{G}(m,n) \leftarrow G(m,n) * W(m,n).$$

Um die Fouriertransformierte des Bilds möglichst wenig zu beeinträchtigen, wäre das Spektrum von $w(u,v)$ idealerweise die Impulsfunktion $\delta(m,n)$, die aber wiederum einer konstanten Funktion $w(u,v) = 1$ entspricht und damit keinen Fenstereffekt hätte. Grundsätzlich gilt, dass je *breiter* das Spektrum der Fensterfunktion $w(u,v)$ ist, desto stärker wird das Spektrum der damit gewichteten Bildfunktion „verwischt“ und umso schlechter können einzelne Spektralkomponenten identifiziert werden.

Die Aufnahme eines Bilds entspricht der Entnahme eines endlichen Abschnitts aus einem eigentlich unendlichen Bildsignal, wobei die Beschneidung an den Bildrändern implizit der Multiplikation mit einer *Rechteckfunktion* mit der Breite M und der Höhe N entspricht. In diesem Fall wird also das Spektrum der ursprünglichen Intensitätsfunktion mit dem Spektrum der Rechteckfunktion gefaltet. Das Problem dabei ist, dass das Spektrum der Rechteckfunktion (s. Abb. 14.9 (a)) extrem breitbandig ist, also von dem oben genannten Ideal einer möglichst schmalen Pulsfunktion weit entfernt ist.

Diese beiden Beispiele zeigen das Dilemma: Fensterfunktionen sollten einerseits möglichst breit sein, um einen möglichst großen Anteil des ursprünglichen Bilds zu berücksichtigen, andererseits zu den Bildrändern hin auf null abfallen und gleichzeitig nicht zu steil sein, um selbst kein breitbandiges Spektrum zu erzeugen.

14.3.7 Fensterfunktionen

Geeignete Fensterfunktionen müssen daher weiche Übergänge aufweisen und dafür gibt es viele Varianten, die in der digitalen Signalverarbeitung theoretisch und experimentell untersucht wurden (s. beispielsweise [11, Abschn. 9.3], [67, Kap. 10]). Tabelle 14.1 zeigt die Definitionen einiger gängiger Fensterfunktionen, die auch in Abb. 14.9–14.10 jeweils mit dem zugehörigen Spektrum dargestellt sind.

Das Spektrum der Rechteckfunktion (Abb. 14.9 (a)), die alle Bildelemente gleich gewichtet, weist zwar eine relativ dünne Spitze am Ursprung auf, die zunächst eine geringe Verwischung im resultierenden Gesamtspektrum verspricht. Allerdings fällt die spektrale Energie zu den höheren Frequenzen hin nur sehr langsam ab, sodass sich insgesamt ein ziemlich breitbandiges Spektrum ergibt. Wie zu erwarten zeigt die elliptische Fensterfunktion in Abb. 14.9 (b) ein sehr ähnliches Verhalten. Das Gauß-Fenster Abb. 14.9 (c) zeigt deutlich, dass durch eine schmälere Fensterfunktion $w(u, v)$ die Nebenkeulen effektiv eingedämmt werden können, allerdings auf Kosten deutlich verbreiterten Spitze im Zentrum. Tatsächlich stellt keine der Funktionen in Abb. 14.9 eine gute Fensterfunktion dar.

Die Auswahl einer geeigneten Fensterfunktion ist offensichtlich ein heikler Kompromiss, zumal trotz ähnlicher Form der Funktionen im Ortsraum große Unterschiede im Spektralverhalten möglich sind. Günstige Eigenschaften bieten z. B. das *Hanning*-Fenster (Abb. 14.10 (c)) und das *Parzen*-Fenster (Abb. 14.10 (d)), die einfach zu berechnen sind und daher in der Praxis auch häufig eingesetzt werden.

Abb. 14.11 zeigt die Auswirkungen einiger ausgewählter Fensterfunktionen auf das Spektrum eines Intensitätsbilds. Deutlich ist zu erkennen, dass mit zunehmender Verengung der Fensterfunktion zwar die durch die Periodizität des Signals verursachten Artefakte unterdrückt werden, jedoch auch die Auflösung im Spektrum abnimmt und dadurch einzelne

Tabelle 14.1
2D-Fensterfunktionen. Die Funktionen $w(u,v)$ sind jeweils in der Bildmitte zentriert, d. h. $w(M/2, N/2) = 1$, und beziehen sich auf die Radien r_u, r_v und $r_{u,v}$ (Definitionen am Tabellenkopf).

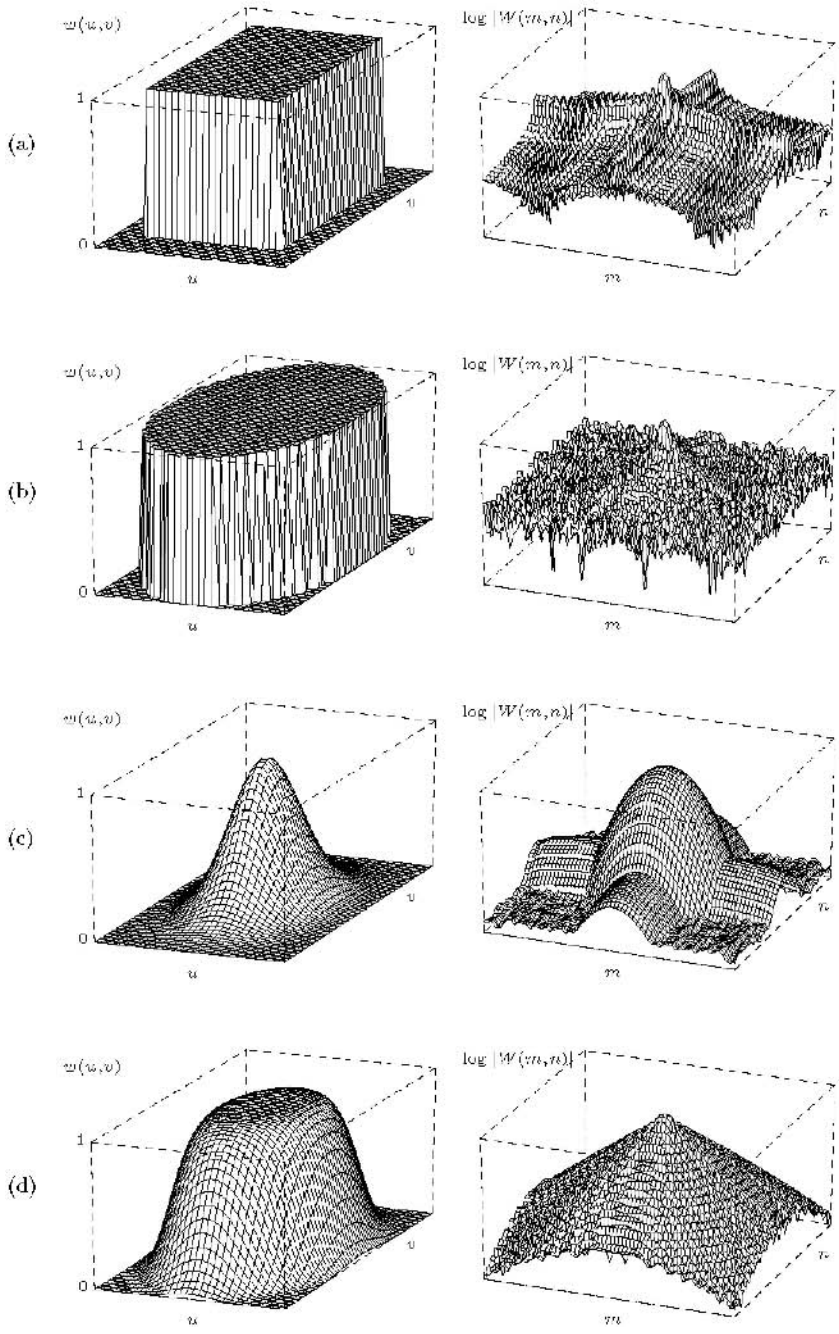
Definitionen:

$$r_u = \tfrac{u-M/2}{M/2} = \tfrac{2u}{M} - 1, \qquad r_v = \tfrac{v-N/2}{N/2} = \tfrac{2v}{N} - 1, \qquad r_{u,v} = \sqrt{r_u^2 + r_v^2}.$$

Elliptisches Fenster:	$w(u,v) = \begin{cases} 1 & \text{für } 0 \le r_{u,v} \le 1 \\ 0 & \text{sonst} \end{cases}$
Gauß-Fenster:	$w(u,v) = e^{\left(\frac{-r_{u,v}^2}{2\sigma^2}\right)}, \quad \sigma = 0.3 \ldots 0.4$
Supergauß-Fenster:	$w(u,v) = e^{\left(\frac{-r_{u,v}^n}{\kappa}\right)}, \quad n = 6,\ \kappa = 0.3 \ldots 0.4$
Kosinus^2-Fenster:	$w(u,v) = \begin{cases} \cos\left(\frac{\pi}{2} r_u\right) \cdot \cos\left(\frac{\pi}{2} r_v\right) & \text{für } 0 \le r_u, r_v \le 1 \\ 0 & \text{sonst} \end{cases}$
Bartlett-Fenster:	$w(u,v) = \begin{cases} 1 - r_{u,v} & \text{für } 0 \le r_{u,v} \le 1 \\ 0 & \text{sonst} \end{cases}$
Hanning-Fenster:	$w(u,v) = \begin{cases} 0.5 \cdot \cos(\pi r_{u,v} + 1) & \text{für } 0 \le r_{u,v} \le 1 \\ 0 & \text{sonst} \end{cases}$
Parzen-Fenster:	$w(u,v) = \begin{cases} 1 - 6r_{u,v}^2 + 6r_{u,v}^3 & \text{für } 0 \le r_{u,v} < 0.5 \\ 2 \cdot (1 - r_{u,v})^3 & \text{für } 0.5 \le r_{u,v} < 1 \\ 0 & \text{sonst} \end{cases}$

Spektralkomponenten zwar deutlicher hervortreten, aber auch in der Breite zunehmen und damit schlechter zu lokalisieren sind.

Abbildung 14.9
Beispiele für Fensterfunktionen und deren logarithmisches Leistungsspektrum. Rechteckfenster (a), elliptisches Fenster (b), Gauß-Fenster mit $\sigma = 0.3$ (c), Supergauß-Fenster der Ordnung $n = 6$ und $\kappa = 0.3$ (d). Die Größe der Fensterfunktion ist absichtlich *nicht* quadratisch gewählt ($M : N = 1 : 2$).



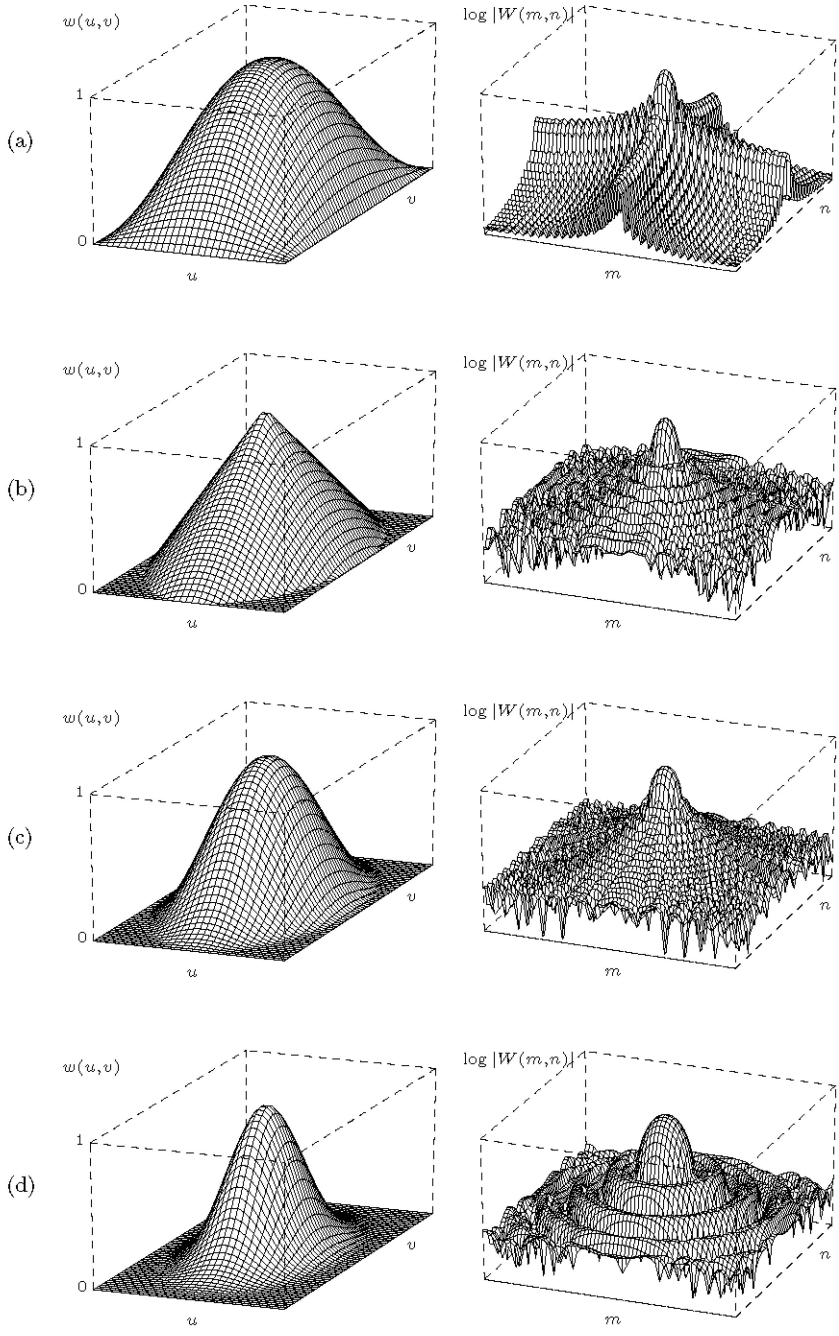


Abbildung 14.10
Beispiele für Fensterfunktionen (*Fortsetzung*). Kosinus^2-Fenster (a), Bartlett-Fenster (b) Hanning-Fenster (c), Parzen-Fenster (d).

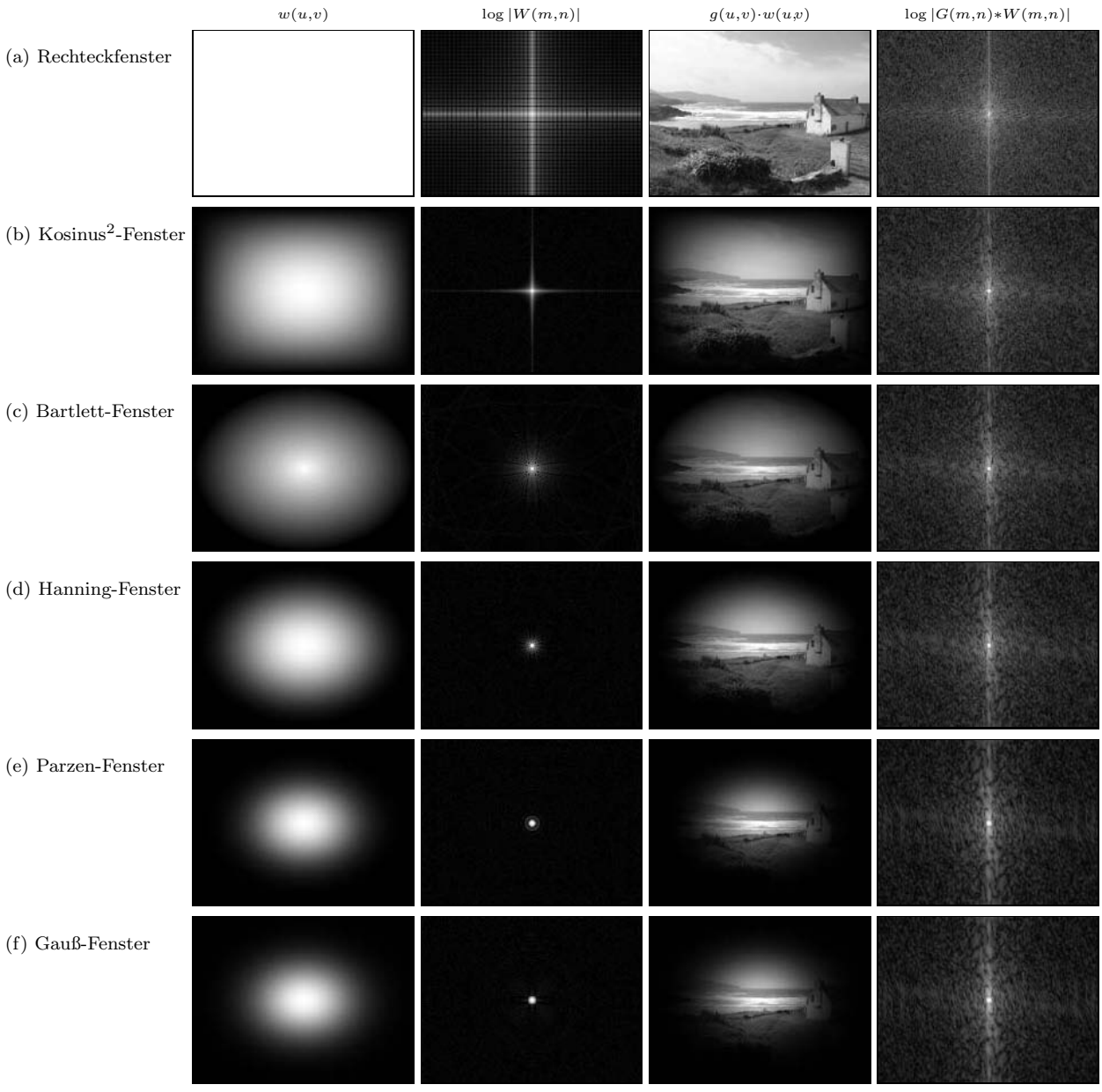


Abbildung 14.11. Anwendung von Fensterfunktionen auf Bilder. Gezeigt ist jeweils die Fensterfunktion $w(u,v)$, das Leistungsspektrum der Fensterfunktion $\log |W(m,n)|$, die gewichtete Bildfunktion $g(u,v) \cdot w(u,v)$ und das Leistungsspektrum des gewichteten Bilds $\log |G(m,n) * W(m,n)|$.

14.4 Beispiele für Fouriertransformierte in 2D

Die nachfolgenden Beispiele demonstrieren einige der grundlegenden Eigenschaften der zweidimensionalen DFT anhand konkreter Intensitätsbilder. Alle Beispiele in Abb. 14.12–14.18 zeigen ein zentriertes und auf quadratische Größe normalisiertes Spektrum, wobei eine logarithmische Skalierung der Intensitätswerte (s. Abschn. 14.2) verwendet wurde.

14.4.1 Skalierung

Abb. 14.12 zeigt, dass – genauso wie im eindimensionalen Fall (s. Abb. 13.4) – die Skalierung der Funktion im Bildraum den umgekehrten Effekt im Spektralraum hat.

14.4.2 Periodische Bildmuster

Die Bilder in Abb. 14.13 enthalten periodische, in unterschiedlichen Richtungen verlaufende Muster, die sich als isolierte Spitzen an den entsprechenden Positionen (s. Gl. 14.13) im zugehörigen Spektrum manifestieren.

14.4.3 Drehung

Abb. 14.14 zeigt, dass die Drehung des Bilds um einen Winkel α eine Drehung des (quadratischen) Spektrums in derselben Richtung und um denselben Winkel verursacht.

14.4.4 Gerichtete, längliche Strukturen

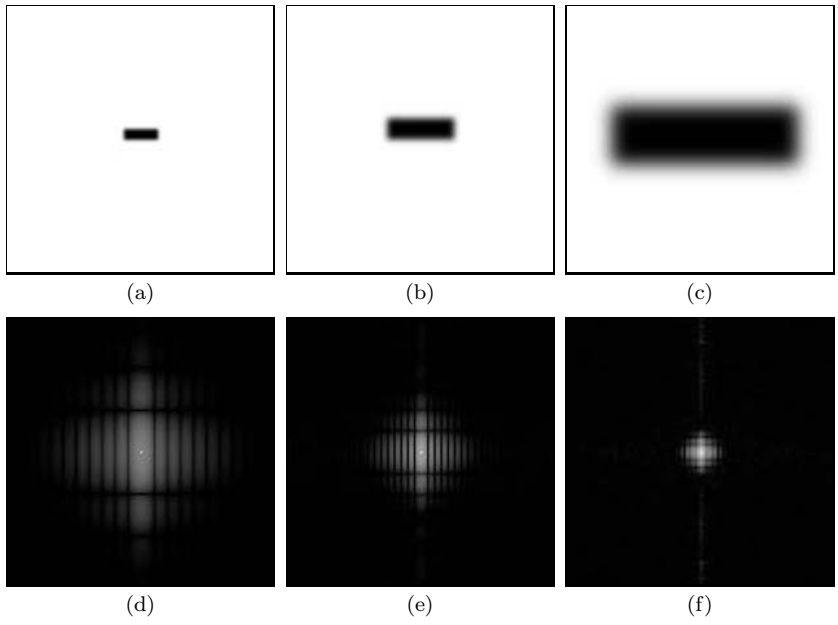
Bilder von künstlichen Objekten enthalten häufig regelmäßige Muster oder längliche Strukturen, die deutliche Spuren im zugehörigen Spektrum hinterlassen. Die Bilder in Abb. 14.15 enthalten mehrere längliche Strukturen, die im Spektrum als breite, rechtwinklig zur Orientierung im Bild ausgerichtete Streifen hervortreten.

14.4.5 Natürliche Bilder

In Abbildungen von natürlichen Objekten sind regelmäßige Anordnungen und gerade Strukturen weniger ausgeprägt als in künstlichen Szenen, daher sind auch die Auswirkungen im Spektrum weniger deutlich. Einige Beispiele dafür zeigen Abb. 14.16 und 14.17.

14.4.6 Druckraster

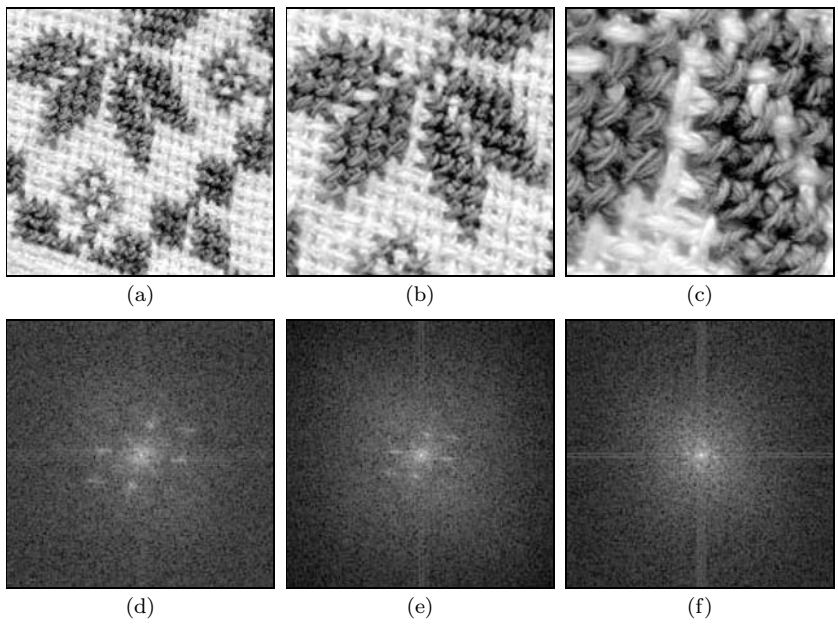
Das regelmäßige Muster, das beim üblichen Rasterdruckverfahren entsteht (Abb. 14.18), ist ein klassisches Beispiel für eine periodische, in mehreren Richtungen verlaufende Struktur, die in der Fouriertransformierten deutlich zu erkennen ist.



(a) (b) (c)

(d) (e) (f)

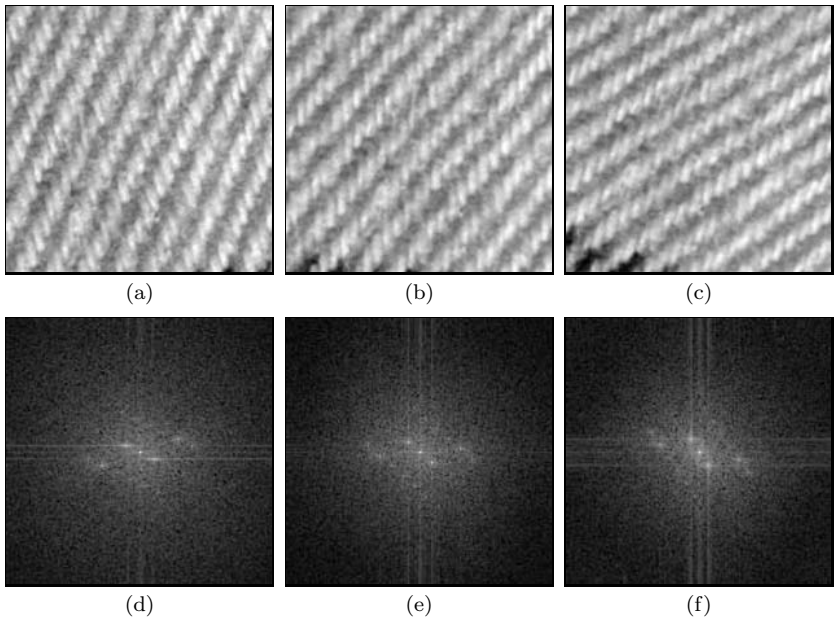
Abbildung 14.12
DFT – Skalierung. Der Rechteckpuls in der Bildfunktion (a–c) erzeugt, wie im eindimensionalen Fall, ein stark ausschwingendes Leistungsspektrum (d–f). Eine Streckung im Bildraum führt zu einer entsprechenden Stauchung im Spektralraum (und umgekehrt).



(a) (b) (c)

(d) (e) (f)

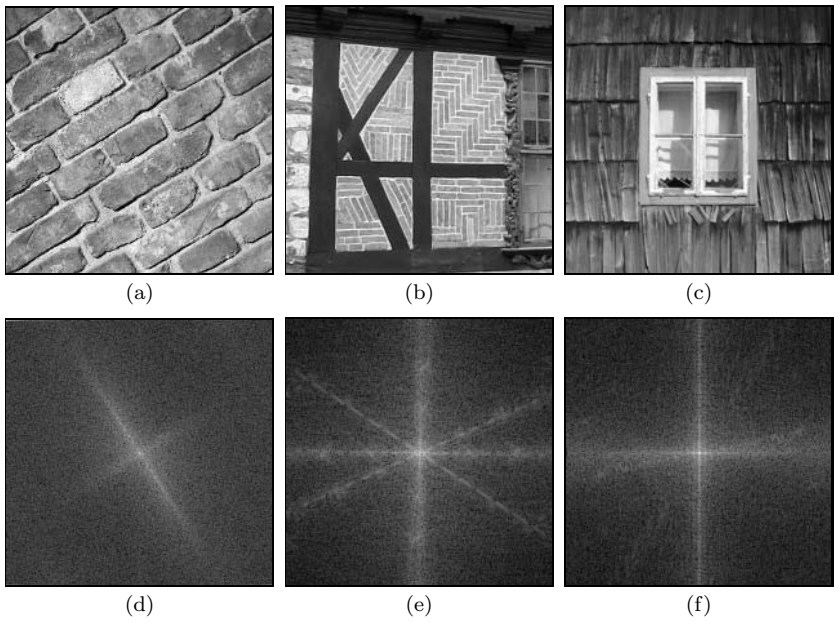
Abbildung 14.13
DFT – gerichtete, periodische Bildmuster. Die Bildfunktion (a–c) enthält Muster in drei dominanten Richtungen, die sich im zugehörigen Spektrum (d–f) als Paare von Spitzenwerten mit der entsprechenden Orientierung wiederfinden. Eine Vergrößerung des Bildmusters führt wie im vorigen Beispiel zur Kontraktion des Spektrums.



(a) (b) (c)

(d) (e) (f)

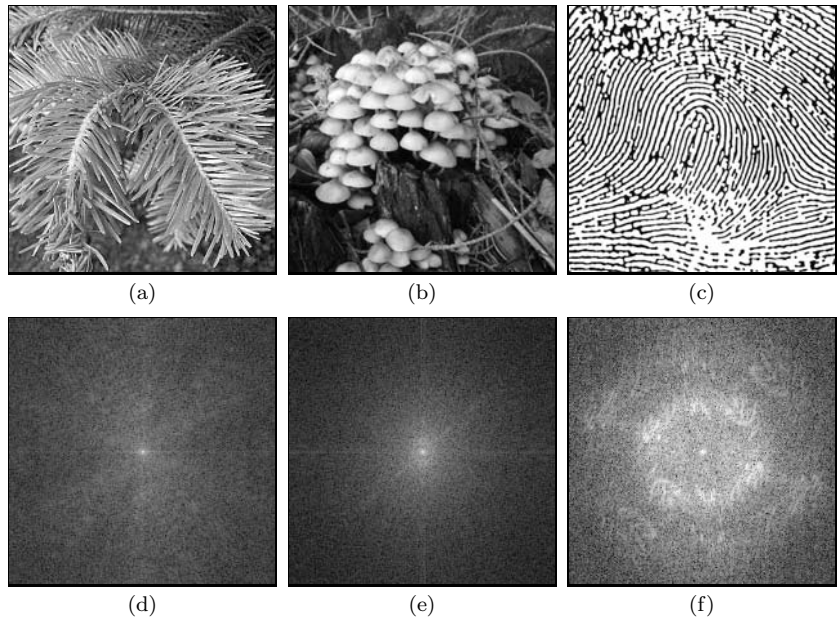
Abbildung 14.14
DFT – Rotation. Das Originalbild (a) wird im Uhrzeigersinn um 15° (b) und 30° (c) gedreht. Das zugehörige (quadratische) Spektrum dreht sich dabei in der gleichen Richtung und um exakt denselben Winkel (d–f).



(a) (b) (c)

(d) (e) (f)

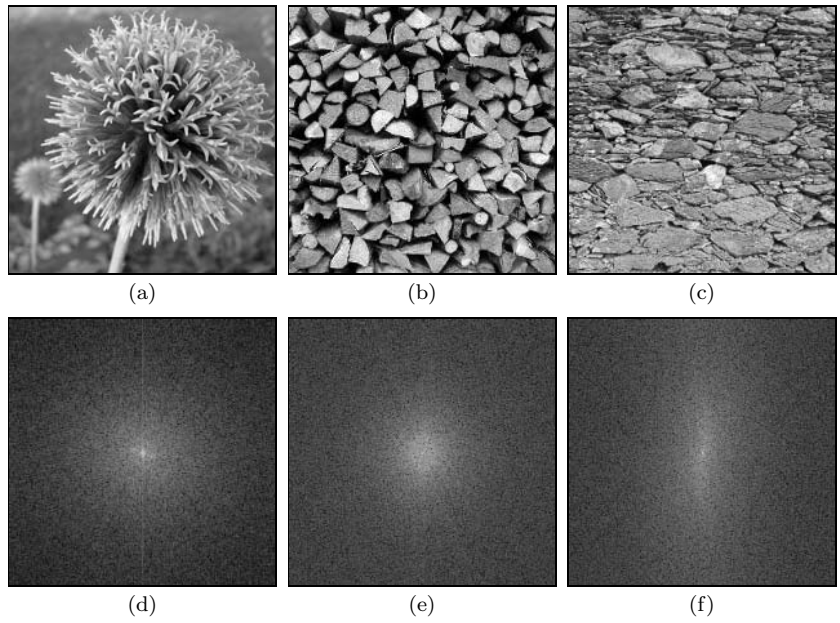
Abbildung 14.15
DFT – Überlagerung von Mustern. Dominante Orientierungen im Bild (a–c) erscheinen unabhängig im zugehörigen Spektrum (d–f). Charakteristisch sind die markanten, breitbandigen Auswirkungen der geraden Strukturen, wie z. B. die dunklen Balken im Mauerwerk (b, e).



(a) (b) (c)

(d) (e) (f)

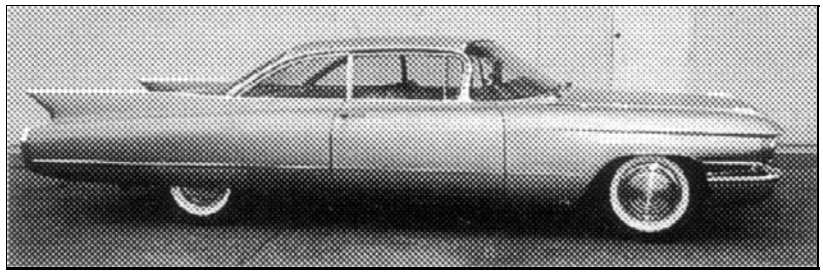
Abbildung 14.16
DFT – natürliche Bildmuster. Beispiele für natürliche Bilder mit repetitiven Mustern (a–c), die auch im zugehörigen Spektrum (d–f) deutlich sichtbar sind.



(a) (b) (c)

(d) (e) (f)

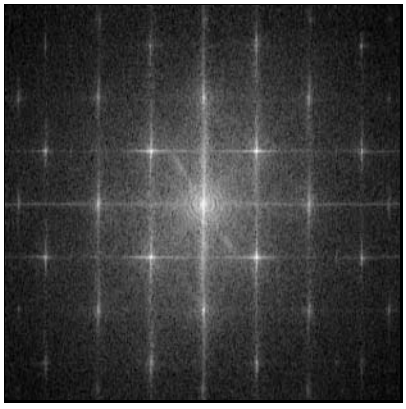
Abbildung 14.17
DFT – natürliche Bildmuster ohne ausgeprägte Orientierung. Obwohl natürliche Bilder (a–c) durchaus repetitive Strukturen enthalten können, sind sie oft nicht ausreichend regelmäßig oder einheitlich gerichtet, um im zugehörigen Fourierspektrum (d–f) deutlich zutage zu treten.



(a)



(b)



(c)

Abbildung 14.18
DFT eines Druckmusters. Das diagonal angeordnete, regelmäßige Druckraster im Originalbild (a, b) zeigt sich deutlich im zugehörigen Leistungsspektrum (c). Es ist möglich, derartige Muster zu entfernen, indem die entsprechenden Spitzen im Fourierspektrum gezielt gelöscht (geglättet) werden und das Bild nachfolgend aus dem geänderten Spektrum durch eine inverse Fouriertransformation wieder rekonstruiert wird.

14.5 Anwendungen der DFT

Die Fouriertransformation und speziell die DFT sind wichtige Werkzeuge in vielen Ingenieurstechniken. In der digitalen Signal- und Bildverarbeitung ist die DFT (und die FFT) ein unverzichtbares Arbeitspferd, mit Anwendungen u. a. im Bereich der Bildanalyse, Filterung und Bildrekonstruktion.

14.5.1 Lineare Filteroperationen im Spektralraum

Die Durchführung von Filteroperationen im Spektralraum ist besonders interessant, da sie die effiziente Anwendung von Filtern mit sehr großer räumlicher Ausdehnung ermöglicht. Grundlage dieser Idee ist die Faltungseigenschaft der Fouriertransformation, die besagt, dass einer linearen Faltung im Ortsraum eine punktweise Multiplikation im Spektralraum entspricht (s. Abschn. 13.1.6, Gl. 13.26). Die lineare Faltung $g * h \rightarrow g'$ zwischen einem Bild $g(u, v)$ und einer Filtermatrix $h(u, v)$ kann daher auf folgendem Weg durchgeführt werden:

$$\begin{array}{rccccc} \text{Ortsraum:} & g(u,v) & * & h(u,v) & = & g'(u,v) \\ & \downarrow & & \downarrow & & \uparrow \\ & \mathsf{DFT} & & \mathsf{DFT} & & \mathsf{DFT}^{-1} \\ & \downarrow & & \downarrow & & \uparrow \\ \text{Spektralraum:} & G(m,n) & \cdot & H(m,n) & \longrightarrow & G'(m,n) \end{array} \tag{14.15}$$

Zunächst werden das Bild g und die Filterfunktion h unabhängig mithilfe der DFT in den Spektralraum transformiert. Die resultierenden Spektra G und H werden punktweise multipliziert, das Ergebnis G' wird anschließend mit der inversen DFT in den Ortsraum zurücktransformiert und ergibt damit das gefilterte Bild g'.

Ein wesentlicher Vorteil dieses „Umwegs“ liegt in der möglichen Effizienz. Die direkte Faltung erfordert für ein Bild der Größe $M \times M$ und eine $N \times N$ große Filtermatrix $\mathcal{O}(M^2N^2)$ Operationen.[2] Die Zeitkomplexität wächst daher quadratisch mit der Filtergröße, was zwar für kleine Filter kein Problem darstellt, größere Filter aber schnell zu aufwendig werden lässt. So benötigt etwa ein Filter der Größe 50×50 bereits ca. 2.500 Multiplikationen und Additionen zur Berechnung jedes einzelnen Bildelements. Im Gegensatz dazu kann die Transformation in den Spektralraum und zurück mit der FFT in $\mathcal{O}(M \log_2 M)$ durchgeführt werden, unabhängig von der Größe des Filters (das Filter selbst braucht nur einmal in den Spektralraum transformiert zu werden), und die Multiplikation im Spektralraum erfordert nur M^2 Operationen, unabhängig von der Größe des Filters.

Darüber hinaus können bestimmte Filter im Spektralraum leichter charakterisiert werden als im Ortsraum, wie etwa ein ideales Tiefpassfilter, das im Spektralraum sehr kompakt dargestellt werden kann. Weitere Details zu Filteroperationen im Spektralraum finden sich z. B. in [30, Abschn. 4.4].

14.5.2 Lineare Faltung und Korrelation

Wie bereits in Abschn. 6.3 erwähnt, ist die lineare *Korrelation* identisch zu einer linearen Faltung mit einer gespiegelten Filterfunktion. Die Korrelation kann daher, genauso wie die Faltung, mit der in Gl. 14.15 beschriebenen Methode im Spektralraum berechnet werden. Das ist vor allem beim Vergleich von Bildern mithilfe von Korrelationsmethoden (s. auch Abschn. 17.1.1) vorteilhaft, da in diesem Fall Bildmatrix und Filtermatrix ähnliche Dimensionen aufweisen, also meist für eine Realisierung im Ortsraum zu groß sind.

Auch in ImageJ sind daher einige dieser Operationen, wie *correlate*, *convolve*, *deconvolve* (s. unten), über die zweidimensionale DFT in der „Fourier Domain“ (FD) implementiert (verfügbar über das Menü Process→FFT→FD Math...).

[2] Zur Notation $\mathcal{O}()$ s. Anhang 1.3.

14.5.3 Inverse Filter

Die Möglichkeit des Filterns im Spektralraum eröffnet eine weitere interessante Perspektive: die Auswirkungen eines Filters wieder rückgängig zu machen, zumindest unter eingeschränkten Bedingungen. Im Folgenden beschreiben wir nur die grundlegende Idee.

Nehmen wir an, wir hätten ein Bild g_{blur}, das aus einem ursprünglichen Bild g_{orig} durch einen Filterprozess entstanden ist, z. B. durch eine Verwischung aufgrund einer Kamerabewegung während der Aufnahme. Nehmen wir außerdem an, diese Veränderung kann als lineares Filter mit der Filterfunktion h_{blur} ausreichend genau modelliert werden, sodass gilt

$$g_{\text{blur}} = g_{\text{orig}} * h_{\text{blur}}.$$

Da dies im Spektralraum einer Multiplikation der zugehörigen Spektren

$$G_{\text{blur}} = G_{\text{orig}} \cdot H_{\text{blur}}$$

entspricht, sollte es möglich sein, das Originalbild einfach durch die inverse Fouriertransformation des Ausdrucks

$$G_{\text{orig}}(m,n) = \frac{G_{\text{blur}}(m,n)}{H_{\text{blur}}(m,n)}$$

zu rekonstruieren. Leider funktioniert dieses inverse Filter nur dann, wenn die Spektralwerte von H_{blur} ungleich null sind, denn andernfalls würden die resultierenden Koeffizienten unendlich. Aber auch kleine Werte von H_{blur}, wie sie typischerweise bei höheren Frequenzen fast immer auftreten, führen zu entsprechend großen Ausschlägen im Ergebnis und damit zu Rauschproblemen.

Es ist ferner wichtig, dass die tatsächliche Filterfunktion sehr genau approximiert werden kann, weil sonst die Ergebnisse vom ursprünglichen Bild erheblich abweichen. Abb. 14.19 zeigt ein Beispiel anhand eines Bilds, das durch eine gleichförmige horizontale Verschiebung verwischt wurde, deren Auswirkung sehr einfach durch eine lineare Faltung modelliert werden kann. Wenn die Filterfunktion, die die Unschärfe verursacht hat, exakt bekannt ist, dann ist die Rekonstruktion problemlos möglich



(a) (b) (c)

Abbildung 14.19
Entfernung von Unschärfe durch ein inverses Filter. Durch horizontale Bewegung erzeugte Unschärfe (a), Rekonstruktion mithilfe der exakten (in diesem Fall bekannten) Filterfunktion (b). Ergebnis des inversen Filters im Fall einer geringfügigen Abweichung von der tatsächlichen Filterfunktion (c).

(Abb. 14.19 (b)). Sobald das inverse Filter sich jedoch nur geringfügig vom tatsächlichen Filter unterscheidet, entstehen große Abweichungen (Abb. 14.19 (c)) und die Methode wird rasch nutzlos.

Über diese einfache Idee hinaus, die häufig als *deconvolution* („Entfaltung“) bezeichnet wird, gibt es allerdings verbesserte Methoden für inverse Filter, wie z. B. das Wiener-Filter und ähnliche Techniken (s. beispielsweise [30, Abschn. 5.4], [49, Abschn. 8.3], [48, Abschn. 17.8], [16, Kap. 16]).

14.6 Aufgaben

Aufg. 14.1. Verwenden Sie die eindimensionale DFT zur Implementierung der 2D-DFT, wie in Abschn. 14 beschrieben. Wenden Sie die 2D-DFT auf konkrete Intensitätsbilder beliebiger Größe an und stellen Sie das Ergebnis (durch Konvertierung in ein `float`-Bild) dar. Implementieren Sie auch die Rücktransformation und überzeugen Sie sich, dass dabei wiederum genau das Originalbild entsteht.

Aufg. 14.2. Das zweidimensionale DFT-Spektrum eines Bilds mit der Größe 640×480 und einer Auflösung von 72 dpi weist einen markanten Spitzenwert an der Stelle $\pm(100, 100)$ auf. Berechnen Sie Richtung und effektive Frequenz (in Perioden pro cm) der zugehörigen Bildstruktur.

Aufg. 14.3. Ein Bild mit der Größe 800×600 enthält ein wellenförmiges Helligkeitsmuster mit einer effektiven Periodenlänge von 12 Pixel und einer Wellenrichtung von 30°. An welcher Position im Spektrum wird sich diese Struktur im 2D-Spektrum widerspiegeln?

Aufg. 14.4. Verallgemeinern Sie Gl. 14.9 sowie Gl. 14.11–14.13 für den Fall, dass die Abtastintervalle in der x- und y-Richtung nicht identisch sind, also für $\tau_x \neq \tau_y$.

Aufg. 14.5. Implementieren Sie die elliptische Fensterfunktion und das Supergauß-Fenster (Tabelle 14.1) als ImageJ-Plugins und beurteilen Sie die Auswirkungen auf das resultierende 2D-Spektrum. Vergleichen Sie das Ergebnis mit dem ungewichteten Fall (ohne Fensterfunktion).

15

Die diskrete Kosinustransformation (DCT)

Die Fouriertransformation und die DFT sind für die Verarbeitung komplexwertiger Signale ausgelegt und erzeugen immer ein komplexwertiges Spektrum, auch wenn das ursprüngliche Signal ausschließlich reelle Werte aufweist. Der Grund dafür ist, dass weder der reelle noch der imaginäre Teil des Spektrums allein ausreicht, um das Signal vollständig darstellen (d. h. rekonstruieren) zu können, da die entsprechenden Kosinus- bzw. Sinusfunktionen jeweils für sich kein vollständiges System von Basisfunktionen bilden.

Andererseits wissen wir (Gl. 13.21), dass ein reellwertiges Signal zu einem symmetrischen Spektrum führt, sodass also in diesem Fall das komplexwertige Spektrum redundant ist und wir eigentlich nur die Hälfte aller Spektralwerte berechnen müssten, ohne dass dabei irgendwelche Informationen aus dem Signal verloren gingen.

Es gibt eine Reihe von Spektraltransformationen, die bezüglich ihrer Eigenschaften der DFT durchaus ähnlich sind, aber nicht mit komplexen Funktionswerten arbeiten. Ein bekanntes Beispiel ist die diskrete Kosinustransformation (DCT), die vor allem im Bereich der Bild- und Videokompression breiten Einsatz findet und daher auch für uns interessant ist. Die DCT verwendet ausschließlich Kosinusfunktionen unterschiedlicher Wellenzahl als Basisfunktionen und beschränkt sich auf reellwertige Signale und Spektralkoeffizienten. Analog dazu existiert auch eine diskrete Sinustransformation (DST) basierend auf einem System von Sinusfunktionen [49].

15.1 Eindimensionale DCT

Die diskrete Kosinustransformation ist allerdings nicht, wie man vielleicht annehmen könnte, nur eine „halbseitige“ Variante der diskreten

Fouriertransformation. Die eindimensionale Vorwärts- und Rückwärtstransformation ist für ein Signal $g(u)$ der Länge M definiert als

$$G(m) = \sqrt{\tfrac{2}{M}} \sum_{u=0}^{M-1} g(u) \cdot c_m \cos\Big(\pi \tfrac{m(2u+1)}{2M}\Big) \quad \text{für } 0 \le m < M \tag{15.1}$$

$$g(u) = \sqrt{\tfrac{2}{M}} \sum_{m=0}^{M-1} G(m) \cdot c_m \cos\Big(\pi \tfrac{m(2u+1)}{2M}\Big) \quad \text{für } 0 \le u < M \tag{15.2}$$

$$\text{wobei } c_m = \begin{cases} \frac{1}{\sqrt{2}} & \text{für } m = 0 \\ 1 & \text{sonst} \end{cases} \tag{15.3}$$

Man beachte, dass die Indexvariablen (u, m) in der Vorwärtstransformation (Gl. 15.1) bzw. der Rückwärtstransformation (Gl. 15.2) unterschiedlich verwendet werden, sodass die beiden Transformationen – im Unterschied zur DFT – *nicht* symmetrisch sind.

15.1.1 Basisfunktionen der DCT

Man könnte sich fragen, wie es möglich ist, dass die DCT ohne Sinusfunktionen auskommt, während diese für die DFT unentbehrlich sind. Der Trick der DCT besteht in der Halbierung aller Frequenzen, wodurch diese enger beisammen liegen und damit die Auflösung im Spektralraum verdoppelt wird. Im Vergleich zwischen den Kosinusanteilen der DFT-Basisfunktionen (Gl. 13.46) und denen der DCT (Gl. 15.1), also

$$\begin{aligned} &\text{DFT: } \boldsymbol{C}_m^M(u) = \cos\big(2\pi \tfrac{mu}{M}\big) \\ &\text{DCT: } \boldsymbol{D}_m^M(u) = \cos\big(\pi \tfrac{m(2u+1)}{2M}\big) = \cos\big(2\pi \tfrac{m(u+0.5)}{2M}\big), \end{aligned} \tag{15.4}$$

wird klar, dass in der DCT die Periodenlänge der Basisfunktionen mit $2M/m$ (gegenüber M/m bei der DFT) verdoppelt ist und zudem die Funktionen um 0.5 Einheiten phasenverschoben sind.

Abb. 15.1 zeigt die DCT-Basisfunktionen $\boldsymbol{D}_m^M(u)$ für eine Signallänge $M = 8$ und Wellenzahlen $m = 0 \ldots 7$. So durchläuft etwa bei der Wellenzahl $m = 7$ die zugehörige Basisfunktion $\boldsymbol{D}_7^8(u)$ 7 volle Perioden über eine Distanz von $2M = 16$ Einheiten und hat damit eine Kreisfrequenz von $\omega = m/2 = 3.5$.

15.1.2 Implementierung der eindimensionalen DCT

Da bei der DCT keine komplexen Werte entstehen und die Vorwärtstransformation (Gl. 15.1) und die inverse Transformation (Gl. 15.2) nahezu identisch sind, ist die direkte Implementierung in Java recht einfach, wie in Prog. 15.1 gezeigt. Zu beachten ist höchstens, dass der Faktor c_m in Gl. 15.1 unabhängig von der Laufvariablen u ist und daher außerhalb der inneren Summationsschleife (Prog. 15.1, Zeile 7) berechnet wird.

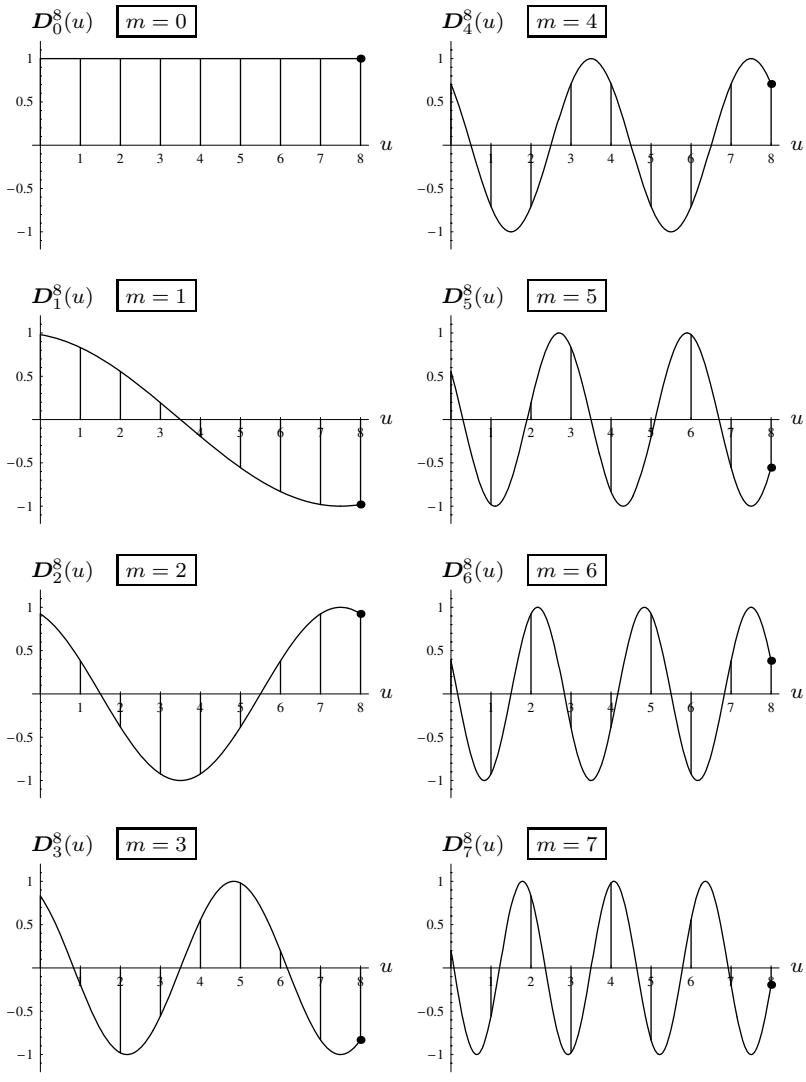


Abbildung 15.1
DCT-Basisfunktionen $\boldsymbol{D}_0^M(u) \ldots \boldsymbol{D}_7^M(u)$ für $M = 8$. Jeder der Plots zeigt sowohl die diskreten Funktionswerte (als dunkle Punkte) wie auch die zugehörige kontinuierliche Funktion. Im Vergleich mit den Basisfunktionen der DFT (Abb. 13.11–13.12) ist zu erkennen, dass alle Frequenzen der DCT-Basisfunktionen halbiert und um 0.5 Einheiten phasenverschoben sind. Alle DCT-Basisfunktionen sind also über die Distanz von $2M = 16$ (anstatt über M bei der DFT) Einheiten periodisch.

Natürlich existieren auch schnelle Algorithmen zur Berechnung der DCT und sie kann außerdem mithilfe der FFT mit einem Zeitaufwand von $\mathcal{O}(M \log_2 M)$ realisiert werden [49, p. 152].[1] Die DCT wird häufig in der Bildkompression eingesetzt, insbesondere im JPEG-Verfahren, wobei die Größe der transformierten Teilbilder auf 8×8 fixiert ist und die Berechnung daher weitgehend optimiert werden kann.

[1] Zur Notation $\mathcal{O}()$ s. Anhang 1.3.

Programm 15.1 Eindimensionale DCT (Java-Implementierung). Die Methode `DCT()` berechnet die Vorwärtstransformation für einen reellwertigen Signalvektor `g` beliebiger Länge entsprechend der Definition in Gl. 15.1. Die Methode liefert das DCT-Spektrum als reellwertigen Vektor derselben Länge wie der Inputvektor `g`. Die Rückwärtstransformation `iDCT()` berechnet die inverse DCT für das reellwertige Kosinusspektrum `G`.

```
1  double[] DCT (double[] g) { // forward DCT on signal g(u)
2    int M = g.length;
3    double s = Math.sqrt(2.0 / M); //common scale factor
4    double[] G = new double[M];
5    for (int m = 0; m < M; m++) {
6      double cm = 1.0;
7      if (m == 0) cm = 1.0 / Math.sqrt(2);
8      double sum = 0;
9      for (int u = 0; u < M; u++) {
10       double Phi = (Math.PI * m * (2 * u + 1)) / (2.0 * M);
11       sum += g[u] * cm * Math.cos(Phi);
12     }
13     G[m] = s * sum;
14   }
15   return G;
16 }
```

```
17 double[] iDCT (double[] G) { // inverse DCT on spectrum G(m)
18   int M = G.length;
19   double s = Math.sqrt(2.0 / M); //common scale factor
20   double[] g = new double[M];
21   for (int u = 0; u < M; u++) {
22     double sum = 0;
23     for (int m = 0; m < M; m++) {
24       double cm = 1.0;
25       if (m == 0) cm = 1.0 / Math.sqrt(2);
26       double Phi = (Math.PI * (2 * u + 1) * m) / (2.0 * M);
27       double cosPhi = Math.cos(Phi);
28       sum += cm * G[m] * cosPhi;
29     }
30     g[u] = s * sum;
31   }
32   return g;
33 }
```

15.2 Zweidimensionale DCT

Die zweidimensionale Form der DCT leitet sich direkt von der eindimensionalen Definition (Gl. 15.1, 15.2) ab, nämlich als Vorwärtstransformation

$$G(m,n) = \frac{2}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u,v) \cdot c_m \cos\Big(\tfrac{\pi(2u+1)m}{2M}\Big) \cdot c_n \cos\Big(\tfrac{\pi(2v+1)n}{2N}\Big)$$
$$= \frac{2c_m c_n}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} g(u,v) \cdot \boldsymbol{D}_m^M(u) \cdot \boldsymbol{D}_n^N(v) \qquad (15.5)$$

für $0 \le m < M$ und $0 \le n < N$, bzw. als inverse Transformation

$$g(u,v) = \frac{2}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(m,n) \cdot c_m \cos\Big(\tfrac{\pi(2u+1)m}{2M}\Big) \cdot c_n \cos\Big(\tfrac{\pi(2v+1)n}{2N}\Big)$$

$$= \frac{2}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(m,n) \cdot c_m \boldsymbol{D}_m^M(u) \cdot c_n \boldsymbol{D}_n^N(v) \qquad (15.6)$$

für $0 \le u < M$ und $0 \le v < N$. Die Faktoren c_m und c_n in Gl. 15.5 und 15.6 sind die gleichen wie im eindimensionalen Fall (Gl. 15.3). Man beachte, dass in der Vorwärtstransformation (und nur dort!) beide Faktoren c_m, c_n unabhängig von den Laufvariablen u, v sind und daher (wie in Gl. 15.5 gezeigt) außerhalb der Summation stehen können.

15.2.1 Separierbarkeit

Wie die DFT (s. Gl. 14.6) kann auch die zweidimensionale DCT in zwei aufeinander folgende, eindimensionale Transformationen getrennt werden. Um dies deutlich zu machen, lässt sich beispielsweise die Vorwärtstransformation in folgender Form ausdrücken:

$$G(m,n) = \sqrt{\tfrac{2}{N}} \sum_{v=0}^{N-1} \Bigg[\underbrace{\sqrt{\tfrac{2}{M}} \sum_{u=0}^{M-1} g(u,v) \cdot c_m \boldsymbol{D}_m^M(u)}_{\text{eindimensionale } \mathsf{DCT}[g(\cdot,v)]} \Bigg] \cdot c_n \boldsymbol{D}_n^N(v)\,. \qquad (15.7)$$

Der innere Ausdruck in Gl. 15.7 entspricht einer eindimensionalen DCT der v-ten Zeile $g(\cdot, v)$ der 2D Signalfunktion. Man kann daher, wie bei der 2D-DFT, zunächst eine eindimensionale DCT auf jede der Zeilen eines Bilds anwenden und anschließend eine DCT in jeder der Spalten. Natürlich könnte man genauso in umgekehrter Reihenfolge rechnen, also zuerst über die Spalten und dann über die Zeilen.

15.2.2 Beispiele

Die Ergebnisse der DFT und der DCT sind anhand eines Beispiels in Abb. 15.2 gegenübergestellt. Weil das DCT-Spektrum (im Unterschied zur DFT) nicht symmetrisch ist, verbleibt der Koordinatenursprung bei der Darstellung links oben und wird nicht ins Zentrum verschoben. Beim DCT-Spektrum ist der Absolutwert logarithmisch als Intensität dargestellt, bei der DFT wie üblich das zentrierte, logarithmische Leistungsspektrum. Man beachte, dass die DCT also nicht einfach ein Teilausschnitt der DFT ist, sondern die Strukturen aus zwei gegenüberliegenden Quadranten des Fourierspektrums kombiniert.

15.3 Andere Spektraltransformationen

Die diskrete Fouriertransformation ist also nicht die einzige Möglichkeit, um ein gegebenes Signal in einem Frequenzraum darzustellen.

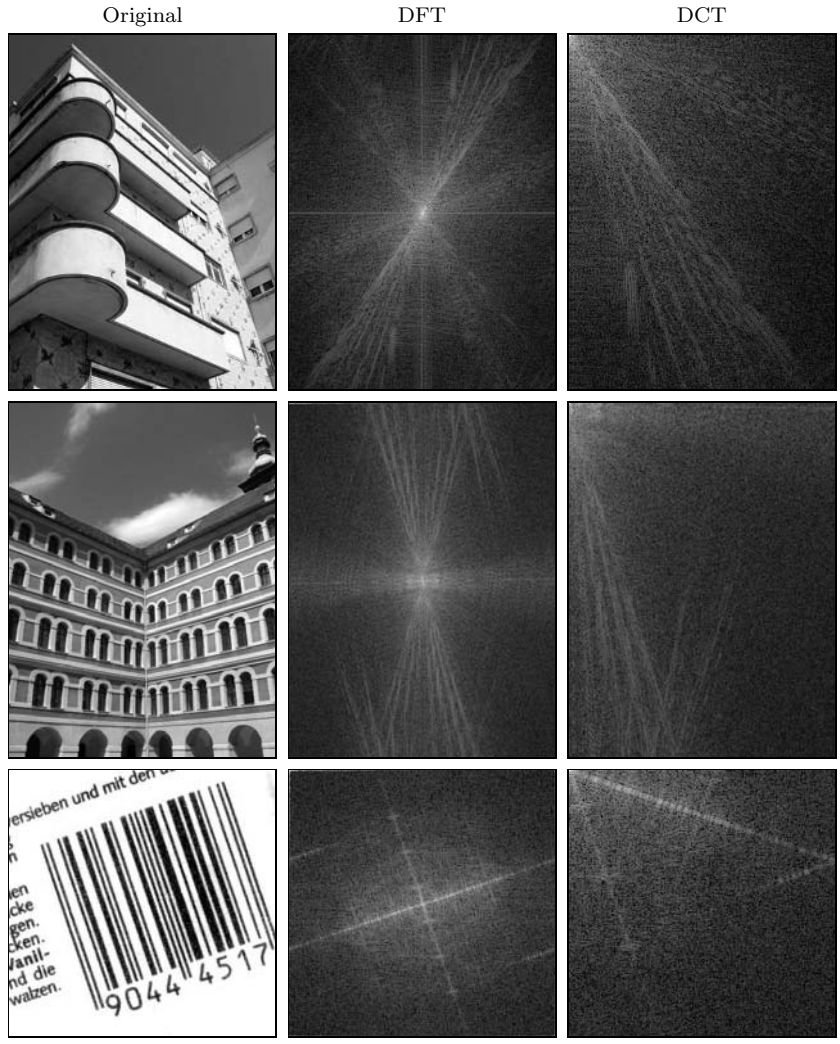


Abbildung 15.2
Vergleich zwischen zweidimensionaler DFT und DCT. Beide Transformationen machen offensichtlich Bildstrukturen in ähnlicher Weise sichtbar. Im reellwertigen DCT-Spektrum (rechts) liegen alle Koeffizienten in nur einem Quadranten beisammen und die spektrale Auflösung ist doppelt so hoch wie bei der DFT (Mitte). Das DFT-Leistungsspektrum ist wie üblich zentriert dargestellt, der Ursprung des DCT-Spektrums liegt hingegen links oben. In beiden Fällen sind die logarithmischen Werte des Spektrums dargestellt.

Tatsächlich existieren zahlreiche ähnliche Transformationen, von denen einige, wie etwa die diskrete Kosinustransformation, ebenfalls sinusoide Funktionen als Basis verwenden, während andere etwa – wie z. B. die Hadamard-Transformation (auch als Walsh-Transformation bekannt) – auf binären 0/1-Funktionen aufbauen [16, 48].

Alle diese Transformationen sind *globaler* Natur, d. h., die Größe jedes Spektralkoeffizienten wird in gleicher Weise von allen Signalwerten beeinflusst, unabhängig von ihrer räumlichen Position innerhalb des Signals. Eine Spitze im Spektrum kann daher aus einem lokal begrenzten Ereignis mit hoher Amplitude stammen, genauso gut aber auch aus einer breiten, gleichmäßigen Welle mit geringer Amplitude. Globale Transformationen sind daher für die Analyse von lokalen Erscheinungen von be-

grenztem Nutzen, denn sie sind nicht imstande, die räumliche Position und Ausdehnung von Signalereignissen darzustellen.

Eine Lösung dieses Problems besteht darin, anstelle einer fixen Gruppe von globalen, ortsunabhängigen Basisfunktionen *lokale*, in ihrer Ausdehnung beschränkte Funktionen zu verwenden, so genannte „Wavelets“. Die zugehörige *Wavelet*-Transformation, von der mehrere Versionen existieren, erlaubt die Lokalisierung von periodischen Signalstrukturen gleichzeitig im Ortsraum *und* im Frequenzraum [55].

15.4 Aufgaben

Aufg. 15.1. Implementieren Sie die zweidimensionale DCT (Abschn. 15.2) für Bilder beliebiger Größe als ImageJ-Plugin.

Aufg. 15.2. Implementieren Sie eine effiziente Java-Methode für die eindimensionale DCT der Länge $M = 8$, die ohne Iteration auskommt und in der alle notwendigen Koeffizienten als vorausberechnete Konstanten angelegt sind.

Aufg. 15.3. Verifizieren Sie durch numerische Berechnung, dass die Basisfunktionen der DCT, $\boldsymbol{D}_m^M(u)$ für $0 \leq m, u < M$ (Gl. 15.4), paarweise orthogonal sind, d. h., dass das innere Produkt der Vektoren $\boldsymbol{D}_m^M \cdot \boldsymbol{D}_n^M$ für $m \neq n$ jeweils null ist.

16

Geometrische Bildoperationen

Allen bisher besprochenen Bildoperationen, also Punkt- und Filteroperationen, war gemeinsam, dass sie zwar die Intensitätsfunktion verändern, die Geometrie des Bilds jedoch unverändert bleibt. Durch geometrische Operationen werden Bilder *verformt*, d. h., Pixelwerte können ihre Position verändern. Typische Beispiele sind etwa eine Verschiebung oder Drehung des Bilds, Skalierungen oder Verformungen, wie in Abb. 16.1 gezeigt. Geometrische Operationen sind in der Praxis sehr häufig, insbesondere in modernen, grafischen Benutzerschnittstellen. So wird heute als selbstverständlich angenommen, dass Bilder in jeder grafischen Anwendung kontinuierlich gezoomt werden können oder die Größe eines Video-Players auf dem Bildschirm beliebig einzustellen ist. In der Computergrafik sind geometrische Operationen etwa auch für die Anwendung von Texturen wichtig, die ebenfalls Rasterbilder sind und – abhängig von der zugehörigen 3D-Oberfläche – für die Darstellung am Bildschirm verformt werden müssen, nach Möglichkeit in Echtzeit. Während man sich leicht vorstellen kann, wie man etwa ein Bild durch einfaches Replizieren jedes Pixels auf ein Vielfaches vergrößern würde, sind allgemeine geometrische Transformationen nicht trivial und erfordern für qualitativ gute Ergebnisse auch auf modernen Computern einen respektablen Teil der verfügbaren Rechenleistung.

Grundsätzlich erzeugt eine geometrische Bildoperation aus dem Ausgangsbild I ein neues Bild I' in der Form

$$I(x, y) \to I'(x', y'), \tag{16.1}$$

wobei also nicht die *Werte* der Bildelemente, sondern deren *Koordinaten* geändert werden. Dafür benötigen wir als Erstes eine Koordinatentransformation in Form einer *geometrischen Abbildungsfunktion*

$$T : \mathbb{R}^2 \to \mathbb{R}^2,$$

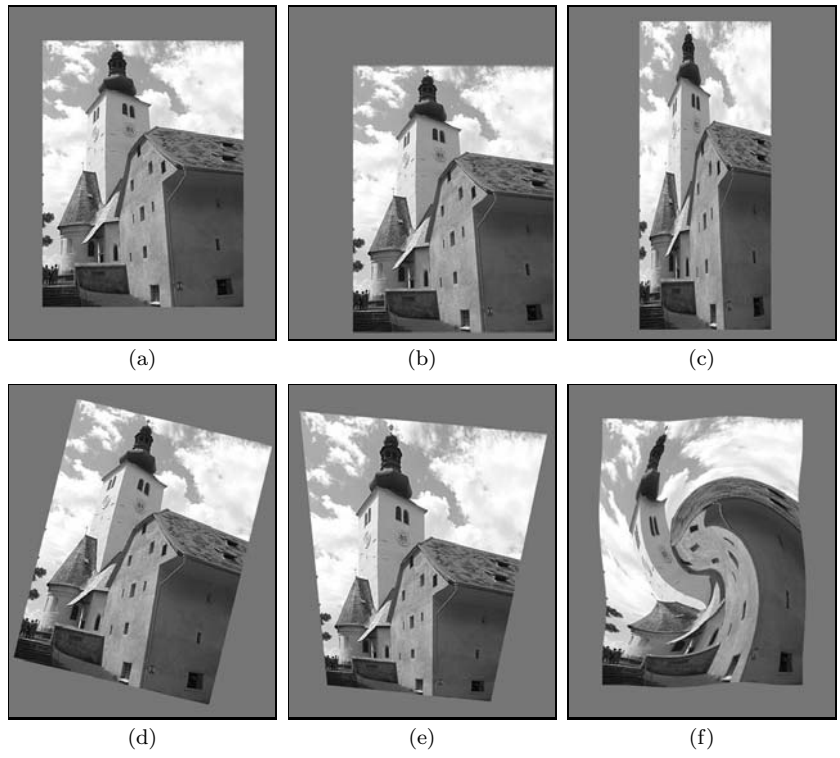


Abbildung 16.1
Typische Beispiele für geometrische Bildoperationen. Ausgangsbild (a), Translation (b), Skalierung (Stauchung bzw. Streckung) in x- und y-Richtung (c), Rotation um den Mittelpunkt (d), projektive Abbildung (e) und nichtlineare Verzerrung (f).

die für jede Ausgangskoordinate $\boldsymbol{x} = (x, y)$ des ursprünglichen Bilds I spezifiziert, an welcher Position $\boldsymbol{x}' = (x', y')$ diese im neuen Bild I' „landen“ soll, d. h.

$$\boldsymbol{x} \;\rightarrow\; \boldsymbol{x}' = T(\boldsymbol{x}). \tag{16.2}$$

Dabei behandeln wir die Bildkoordinaten (x, y) bzw. (x', y') zunächst bewusst als Punkte in der reellen Ebene $\mathbb{R} \times \mathbb{R}$, also als *kontinuierliche* Koordinaten. Das Hauptproblem bei der Transformation ist allerdings, dass die Werte von digitalen Bildern auf einem *diskreten* Raster $\mathbb{Z} \times \mathbb{Z}$ liegen, aber die zugehörige Abbildung $\boldsymbol{x}'$ auch bei ganzzahligen Ausgangskoordinaten $\boldsymbol{x}$ im Allgemeinen *nicht* auf einen Rasterpunkt trifft. Die Lösung dieses Problems besteht in der Berechnung von Zwischenwerten der transformierten Bildfunktion durch *Interpolation*, die damit ein wichtiger Bestandteil jeder geometrischen Operation ist.

16.1 2D-Koordinatentransformation

Die Abbildungsfunktion $T()$ in Gl. 16.2 ist grundsätzlich eine beliebige, stetige Funktion, die man zweckmäßigerweise in zwei voneinander unabhängige Teilfunktionen

$$x' = T_x(x, y) \qquad \text{und} \qquad y' = T_y(x, y) \tag{16.3}$$

trennen kann.

16.1.1 Einfache Abbildungen

Zu den einfachen Abbildungsfunktionen gehören Verschiebung, Skalierung, Scherung und Rotation:

Verschiebung (Translation) um den Vektor (d_x, d_y):

$$\begin{matrix} T_x : x' = x + d_x \\ T_y : y' = y + d_y \end{matrix} \quad \text{oder} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix} \tag{16.4}$$

Skalierung (Streckung oder Stauchung) in x- oder y-Richtung um den Faktor s_x bzw. s_y:

$$\begin{matrix} T_x : x' = s_x \cdot x \\ T_y : y' = s_y \cdot y \end{matrix} \quad \text{oder} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \tag{16.5}$$

Scherung in x- oder y-Richtung um den Faktor b_x bzw. b_y (bei einer Scherung in nur einer Richtung ist der jeweils andere Faktor null):

$$\begin{matrix} T_x : x' = x + b_x \cdot y \\ T_y : y' = y + b_y \cdot x \end{matrix} \quad \text{oder} \quad \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & b_x \\ b_y & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \tag{16.6}$$

Rotation (Drehung) um den Winkel α (mit dem Koordinatenursprung als Drehmittelpunkt):

$$\begin{matrix} T_x : x' = & x \cdot \cos\alpha + y \cdot \sin\alpha \\ T_y : y' = & -x \cdot \sin\alpha + y \cdot \cos\alpha \end{matrix} \quad \text{oder} \tag{16.7}$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \tag{16.8}$$

16.1.2 Homogene Koordinaten

Die Operationen in Gl. 16.4–16.8 bilden zusammen die wichtige Klasse der affinen Abbildungen (siehe Abschn. 16.1.3). Für die Verknüpfung durch Hintereinanderausführung ist es vorteilhaft, wenn alle Operationen jeweils als Matrixmultiplikation beschreibbar sind. Das ist bei der Translation (Gl. 16.4), die eine Vektoraddition ist, nicht der Fall. Eine mathematisch elegante Lösung dafür sind *homogene Koordinaten* [23, S. 204].

Bei homogenen Koordinaten wird jeder Vektor um eine zusätzliche Komponente (h) erweitert, d. h. für den zweidimensionalen Fall

$$\boldsymbol{x} = \begin{pmatrix} x \\ y \end{pmatrix} \quad \rightarrow \quad \hat{\boldsymbol{x}} = \begin{pmatrix} \hat{x} \\ \hat{y} \\ h \end{pmatrix} = \begin{pmatrix} h\,x \\ h\,y \\ h \end{pmatrix}. \tag{16.9}$$

Jedes gewöhnliche (kartesische) Koordinatenpaar $\boldsymbol{x} = (x, y)$ wird also durch den dreidimensionalen homogenen Koordinatenvektor $\hat{\boldsymbol{x}} = (\hat{x}, \hat{y}, h)$

dargestellt. Sofern die h-Komponente eines homogenen Vektors $\hat{\boldsymbol{x}}$ ungleich null ist, erhalten wir durch

$$x = \frac{\hat{x}}{h} \qquad \text{und} \qquad y = \frac{\hat{y}}{h} \tag{16.10}$$

wiederum die zugehörigen kartesischen Koordinaten (x, y). Es gibt also (durch unterschiedliche Werte für h) unendlich viele Möglichkeiten, einen bestimmten 2D-Punkt (x, y) in homogenen Koordinaten darzustellen. Insbesondere repräsentieren daher zwei homogene Koordinaten $\hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_2$ *denselben* Punkt in 2D, wenn sie Vielfache voneinander sind, d. h.

$$\boldsymbol{x}_1 = \boldsymbol{x}_2 \;\Leftrightarrow\; \hat{\boldsymbol{x}}_1 = s \cdot \hat{\boldsymbol{x}}_2 \qquad \text{für } s \neq 0. \tag{16.11}$$

Beispielsweise sind die homogenen Koordinaten $\hat{\boldsymbol{x}}_1 = (3, 2, 1)$, $\hat{\boldsymbol{x}}_2 = (6, 4, 2)$ und $\hat{\boldsymbol{x}}_3 = (30, 20, 10)$ alle äquivalent und entsprechen dem kartesischen Punkt $(3, 2)$.

16.1.3 Affine Abbildung (Dreipunkt-Abbildung)

Mithilfe der homogenen Koordinaten lässt sich nun jede Kombination aus Translation, Skalierung und Rotation in der Form

$$\begin{pmatrix} \hat{x}' \\ \hat{y}' \\ h' \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{16.12}$$

darstellen. Man bezeichnet diese Transformation als „affine Abbildung“ mit den 6 Freiheitsgraden, $a_{11} \ldots a_{23}$, wobei a_{13}, a_{23} (analog zu d_x, d_y in Gl. 16.4) die *Translation* und $a_{11}, a_{12}, a_{21}, a_{22}$ zusammen die *Skalierung*, *Scherung* und *Rotation* definieren. Durch die affine Abbildung werden Geraden in Geraden, Dreiecke in Dreiecke und Rechtecke in Parallelogramme überführt (Abb. 16.2). Charakteristisch für die affine Abbildung ist auch, dass das Abstandsverhältnis zwischen den auf einer Geraden liegenden Punkten durch die Abbildung unverändert bleibt.

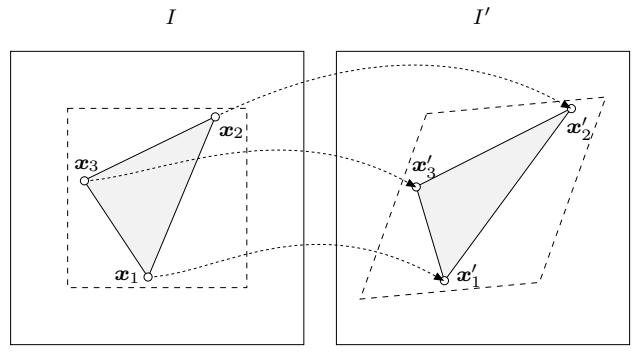


Abbildung 16.2
Affine Abbildung. Durch die Spezifikation von drei korrespondierenden Punktpaaren ist eine affine Abbildung eindeutig bestimmt. Sie kann beliebige Dreiecke ineinander überführen und bildet Geraden in Geraden ab, parallele Geraden bleiben parallel und die Abstandsverhältnisse zwischen Punkten auf einer Geraden verändern sich nicht.

Ermittlung der Abbildungsparameter

Die Parameter der Abbildung in Gl. 16.12 werden durch Vorgabe von 3 korrespondierenden Punktpaaren $(\boldsymbol{x}_1, \boldsymbol{x}'_1)$, $(\boldsymbol{x}_2, \boldsymbol{x}'_2)$, $(\boldsymbol{x}_3, \boldsymbol{x}'_3)$, mit jeweils einem Punkt $\boldsymbol{x}_i = (x_i, y_i)$ im Ausgangsbild und dem zugehörigen Punkt $\boldsymbol{x}'_i = (x'_i, y'_i)$ im Zielbild, eindeutig spezifiziert. Sie ergeben sich durch Lösung des linearen Gleichungssystems

$$\begin{aligned} x'_1 &= a_{11} \cdot x_1 + a_{12} \cdot y_1 + a_{13} \quad & y'_1 &= a_{21} \cdot x_1 + a_{22} \cdot y_1 + a_{23} \\ x'_2 &= a_{11} \cdot x_2 + a_{12} \cdot y_2 + a_{13} \quad & y'_2 &= a_{21} \cdot x_2 + a_{22} \cdot y_2 + a_{23} \\ x'_3 &= a_{11} \cdot x_3 + a_{12} \cdot y_3 + a_{13} \quad & y'_3 &= a_{21} \cdot x_3 + a_{22} \cdot y_3 + a_{23} \end{aligned} \tag{16.13}$$

unter der Voraussetzung, dass die Bildpunkte $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3$ linear unabhängig sind (d. h., nicht auf einer gemeinsamen Geraden liegen), als

$$\begin{aligned} a_{11} &= \tfrac{1}{S} \cdot \big[y_1(x'_2 - x'_3) &&+ y_2(x'_3 - x'_1) &&+ y_3(x'_1 - x'_2)\big] \\ a_{12} &= \tfrac{1}{S} \cdot \big[x_1(x'_3 - x'_2) &&+ x_2(x'_1 - x'_3) &&+ x_3(x'_2 - x'_1)\big] \\ a_{21} &= \tfrac{1}{S} \cdot \big[y_1(y'_2 - y'_3) &&+ y_2(y'_3 - y'_1) &&+ y_3(y'_1 - y'_2)\big] \\ a_{22} &= \tfrac{1}{S} \cdot \big[x_1(y'_3 - y'_2) &&+ x_2(y'_1 - y'_3) &&+ x_3(y'_2 - y'_1)\big] \\ a_{13} &= \tfrac{1}{S} \cdot \big[x_1(y_3 x'_2 - y_2 x'_3) &&+ x_2(y_1 x'_3 - y_3 x'_1) &&+ x_3(y_2 x'_1 - y_1 x'_2)\big] \\ a_{23} &= \tfrac{1}{S} \cdot \big[x_1(y_3 y'_2 - y_2 y'_3) &&+ x_2(y_1 y'_3 - y_3 y'_1) &&+ x_3(y_2 y'_1 - y_1 y'_2)\big] \\ \text{mit } \; S &= x_1(y_3 - y_2) + x_2(y_1 - y_3) + x_3(y_2 - y_1)\,. \end{aligned} \tag{16.14}$$

Inversion der affinen Abbildung

Die *Umkehrung* T^{-1} der affinen Abbildung, die in der Praxis häufig benötigt wird (siehe Abschn. 16.2.2), ergibt sich durch Inversion der Transformationsmatrix in Gl. 16.12 als

$$\begin{aligned} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \\ &= \frac{1}{a_{11}a_{22} - a_{12}a_{21}} \begin{pmatrix} a_{22} & -a_{12} & a_{12}a_{23} - a_{13}a_{22} \\ -a_{21} & a_{11} & a_{13}a_{21} - a_{11}a_{23} \\ 0 & 0 & a_{11}a_{22} - a_{12}a_{21} \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \end{aligned} \tag{16.15}$$

Natürlich lässt sich die inverse Abbildung auch aus drei korrespondierenden Punktpaaren nach Gl. 16.13 und 16.14 durch Vertauschung von Ausgangs- und Zielbild berechnen.

16.1.4 Projektive Abbildung (Vierpunkt-Abbildung)

Die affine Abbildung ist zwar geeignet, beliebige Dreiecke ineinander überzuführen, häufig benötigt man jedoch eine allgemeine Verformung

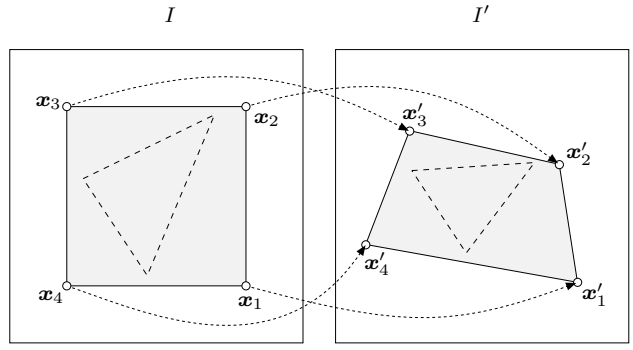


Abbildung 16.3
Projektive Abbildung. Durch vier korrespondierende Punktpaare ist eine projektive Abbildung eindeutig spezifiziert. Geraden werden wieder in Geraden, Rechtecke in beliebige Vierecke abgebildet. Parallelen bleiben nicht erhalten und auch die Abstandsverhältnisse zwischen Punkten auf einer Geraden werden i. Allg. verändert.

von Vierecken, etwa bei der Transformationen auf Basis einer Mesh-Partitionierung (Abschn. 16.1.7). Um vier beliebig angeordnete 2D-Punktpaare ineinander überzuführen, erfordert die zugehörige Abbildung insgesamt acht Freiheitsgrade. Die gegenüber der affinen Abbildung zusätzlichen zwei Freiheitsgrade ergeben sich in der so genannten *projektiven* Abbildung[1] durch die Koeffizienten a_{31}, a_{32}:

$$\begin{pmatrix} \hat{x}' \\ \hat{y}' \\ h' \end{pmatrix} = \begin{pmatrix} h'x' \\ h'y' \\ h' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{16.16}$$

Diese in homogenen Koordinaten lineare Abbildung entspricht in kartesischen Koordinaten der nichtlinearen Transformation

$$\begin{aligned} x' &= \frac{1}{h'} \cdot \big(a_{11}\, x + a_{12}\, y + a_{13}\big) = \frac{a_{11}\, x + a_{12}\, y + a_{13}}{a_{31}\, x + a_{32}\, y + 1} \\ y' &= \frac{1}{h'} \cdot \big(a_{21}\, x + a_{22}\, y + a_{23}\big) = \frac{a_{21}\, x + a_{22}\, y + a_{23}}{a_{31}\, x + a_{32}\, y + 1} \end{aligned} \tag{16.17}$$

Geraden bleiben aber trotz dieser Nichtlinearität auch unter einer projektiven Abbildung erhalten. Tatsächlich ist dies die allgemeinste Transformation, die Geraden auf Geraden abbildet und algebraische Kurven n-ter Ordnung wieder in algebraische Kurven n-ter Ordnung überführt. Insbesondere werden etwa Kreise oder Ellipsen wieder als Kurven zweiter Ordnung (Kegelschnitte) abgebildet. Im Unterschied zur affinen Abbildung müssen aber parallele Geraden nicht wieder auf parallele Geraden abgebildet werden und auch die Abstandsverhältnisse zwischen Punkten auf einer Geraden bleiben im Allgemeinen nicht erhalten (Abb. 16.3).

Ermittlung der Abbildungsparameter

Bei Vorgabe von vier korrespondierenden 2D-Punktpaaren $(\boldsymbol{x}_1, \boldsymbol{x}'_1) \ldots (\boldsymbol{x}_4, \boldsymbol{x}'_4)$, mit jeweils einem Punkt $\boldsymbol{x}_i = (x_i, y_i)$ im Ausgangsbild und

[1] Auch als *perspektivische* oder *pseudoperspektivische* Abbildung bezeichnet.

dem zugehörigen Punkt $\boldsymbol{x}'_i = (x'_i, y'_i)$ im Zielbild, können die acht unbekannten Parameter $a_{11} \ldots a_{32}$ der Abbildung durch Lösung des folgenden linearen Gleichungssystems berechnet werden, das sich durch Einsetzen der Punktkoordinaten in Gl. 16.17 ergibt:

$$\begin{aligned} x'_i &= a_{11}\,x_i + a_{12}\,y_i + a_{13} - a_{31}\,x_i\,x'_i - a_{32}\,y_i\,x'_i \\ y'_i &= a_{21}\,x_i + a_{22}\,y_i + a_{23} - a_{31}\,x_i\,y'_i - a_{32}\,y_i\,y'_i \end{aligned} \tag{16.18}$$

für $i = 1 \ldots 4$. Zusammengefasst ergeben diese acht Gleichungen in Matrixschreibweise

$$\begin{pmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4 \end{pmatrix} \cdot \begin{pmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \end{pmatrix} \tag{16.19}$$

beziehungsweise

$$\boldsymbol{x}' = \boldsymbol{M} \cdot \boldsymbol{a}. \tag{16.20}$$

Der unbekannte Parametervektor $\boldsymbol{a} = (a_{11}, a_{12}, \ldots a_{32})$ kann durch Lösung dieses Gleichungssystems mithilfe eines der numerischen Standardverfahren (z. B. mit dem Gauß-Algorithmus [90, S. 1099]) berechnet werden.[2]

Inversion der projektiven Abbildung

Eine lineare Abbildung der Form $\boldsymbol{x}' = \boldsymbol{A}\cdot\boldsymbol{x}$ kann allgemein durch Invertieren der Matrix $\boldsymbol{A}$ umgekehrt werden, d. h. $\boldsymbol{x} = \boldsymbol{A}^{-1}\cdot\boldsymbol{x}'$, vorausgesetzt $\boldsymbol{A}$ ist regulär ($\mathrm{Det}(\boldsymbol{A}) \neq 0$). Für eine 3×3-Matrix $\boldsymbol{A}$ lässt sich die Inverse auf relativ einfache Weise durch die Beziehung

$$\boldsymbol{A}^{-1} = \frac{1}{\mathrm{Det}(\boldsymbol{A})}\boldsymbol{A}_{\mathrm{adj}} \tag{16.21}$$

über die Determinante $\mathrm{Det}(\boldsymbol{A})$ und die zugehörige *adjungierte* Matrix $\boldsymbol{A}_{\mathrm{adj}}$ berechnen [12, S. 270], wobei im allgemeinen Fall

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \qquad \text{und}$$

[2] Dafür greift man am besten auf fertige Software zurück, wie z. B. *Jampack* (Java Matrix Package) von G. W. Stewart (ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html).

$$\begin{aligned}\mathrm{Det}(\boldsymbol{A}) = \quad & a_{11}\,a_{22}\,a_{33} + a_{12}\,a_{23}\,a_{31} + a_{13}\,a_{21}\,a_{32} \\ & - a_{11}\,a_{23}\,a_{32} - a_{12}\,a_{21}\,a_{33} - a_{13}\,a_{22}\,a_{31}\end{aligned} \tag{16.22}$$

$$\boldsymbol{A}_{\mathrm{adj}} = \begin{pmatrix} a_{22}\,a_{33} - a_{23}\,a_{32} & a_{13}\,a_{32} - a_{12}\,a_{33} & a_{12}\,a_{23} - a_{13}\,a_{22} \\ a_{23}\,a_{31} - a_{21}\,a_{33} & a_{11}\,a_{33} - a_{13}\,a_{31} & a_{13}\,a_{21} - a_{11}\,a_{23} \\ a_{21}\,a_{32} - a_{22}\,a_{31} & a_{12}\,a_{31} - a_{11}\,a_{32} & a_{11}\,a_{22} - a_{12}\,a_{21} \end{pmatrix} \tag{16.23}$$

Bei der projektiven Abbildung (Gl. 16.16) ist $a_{33} = 1$, was die Berechnung noch geringfügig vereinfacht. Da bei homogenen Koordinaten die Multiplikation eines Vektors mit einem Skalar wieder einen äquivalenten Vektor erzeugt (Gl. 16.11), ist die Einbeziehung der Determinante $\mathrm{Det}(\boldsymbol{A})$ eigentlich überflüssig. Zur Umkehrung der Transformation genügt es daher, den homogenen Koordinatenvektor mit der adjungierten Matrix zu multiplizieren und den resultierenden Vektor anschließend (bei Bedarf) zu „homogenisieren“, d. h.

$$\begin{pmatrix} x \\ y \\ h \end{pmatrix} \leftarrow \boldsymbol{A}_{\mathrm{adj}} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \qquad \text{und} \qquad \begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \frac{1}{h} \begin{pmatrix} x \\ y \end{pmatrix}. \tag{16.24}$$

Die in Gl. 16.15 ausgeführte Umkehrung der *affinen* Abbildung ist damit natürlich nur ein spezieller Fall dieser allgemeineren Methode für lineare Abbildungen, zumal auch die affine Abbildung selbst nur eine Unterklasse der projektiven Abbildungen ist.

Projektive Abbildung über das Einheitsquadrat

Eine Alternative zur iterativen Lösung des linearen Gleichungssystems mit 8 Unbekannten in Gl. 16.19 ist die zweistufige Abbildung über das Einheitsquadrat $\mathcal{S}_1$. Bei der in Abb. 16.4 dargestellten projektiven Transformation des Einheitsquadrats in ein beliebiges Viereck mit den Punkten $\boldsymbol{x}'_1, \ldots \boldsymbol{x}'_4$, also

$$\begin{aligned} (0,0) &\rightarrow \boldsymbol{x}'_1 & \qquad (1,1) &\rightarrow \boldsymbol{x}'_3 \\ (1,0) &\rightarrow \boldsymbol{x}'_2 & \qquad (0,1) &\rightarrow \boldsymbol{x}'_4 \end{aligned}$$

reduziert sich das ursprüngliche Gleichungssystem aus Gl. 16.19 auf

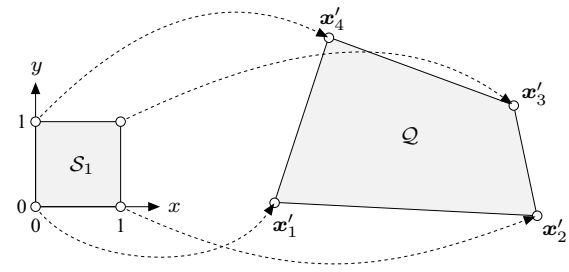


Abbildung 16.4
Projektive Abbildung des Einheitsquadrats $\mathcal{S}_1$ auf ein beliebiges Viereck $\mathcal{Q} = (\boldsymbol{x}'_1, \ldots \boldsymbol{x}'_4)$.

$$
\begin{aligned}
x'_1 &= a_{13} \\
y'_1 &= a_{23} \\
x'_2 &= a_{11} + a_{13} - a_{31} \cdot x'_2 \\
y'_2 &= a_{21} + a_{23} - a_{31} \cdot y'_2 \\
x'_3 &= a_{11} + a_{12} + a_{13} - a_{31} \cdot x'_3 - a_{32} \cdot x'_3 \\
y'_3 &= a_{21} + a_{22} + a_{23} - a_{31} \cdot y'_3 - a_{32} \cdot y'_3 \\
x'_4 &= a_{12} + a_{13} - a_{32} \cdot x'_4 \\
y'_4 &= a_{22} + a_{23} - a_{32} \cdot y'_4
\end{aligned}
\tag{16.25}
$$

und besitzt folgende Lösung für die Transformationsparameter $a_{11}, a_{12}, \ldots a_{32}$:

$$
\begin{aligned}
a_{31} &= \frac{(x'_1 - x'_2 + x'_3 - x'_4)\cdot(y'_4 - y'_3) \;-\; (y'_1 - y'_2 + y'_3 - y'_4)\cdot(x'_4 - x'_3)}{(x'_2 - x'_3)\cdot(y'_4 - y'_3) \;-\; (x'_4 - x'_3)\cdot(y'_2 - y'_3)} \\
a_{32} &= \frac{(y'_1 - y'_2 + y'_3 - y'_4)\cdot(x'_2 - x'_3) \;-\; (x'_1 - x'_2 + x'_3 - x'_4)\cdot(y'_2 - y'_3)}{(x'_2 - x'_3)\cdot(y'_4 - y'_3) \;-\; (x'_4 - x'_3)\cdot(y'_2 - y'_3)}
\end{aligned}
\tag{16.26}
$$

$$
\begin{aligned}
a_{11} &= x'_2 - x'_1 + a_{31}\, x'_2 \qquad & a_{12} &= x'_4 - x'_1 + a_{32}\, x'_4 \qquad & a_{13} &= x'_1 \\
a_{21} &= y'_2 - y'_1 + a_{31}\, y'_2 \qquad & a_{22} &= y'_4 - y'_1 + a_{32}\, y'_4 \qquad & a_{23} &= y'_1
\end{aligned}
$$

Durch Invertieren der zugehörigen Transformationsmatrix (Gl. 16.21) ist auf diese Weise natürlich auch die umgekehrte Abbildung, also von einem beliebigen Viereck in das Einheitsquadrat, möglich. Wie in Abb. 16.5 dargestellt, lässt sich so auch die Abbildung

$$\mathcal{Q} \xrightarrow{T} \mathcal{Q}'$$

eines beliebigen Vierecks $\mathcal{Q} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4)$ auf ein anderes, ebenfalls beliebiges Viereck $\mathcal{Q}' = (\boldsymbol{x}'_1, \boldsymbol{x}'_2, \boldsymbol{x}'_3, \boldsymbol{x}'_4)$ als zweistufige Transformation über das Einheitsquadrat $\mathcal{S}_1$ durchführen [89, S. 55], und zwar in der Form

$$\mathcal{Q} \xrightarrow{T_1^{-1}} \mathcal{S}_1 \xrightarrow{T_2} \mathcal{Q}'. \tag{16.27}$$

Die Transformationen T_1 und T_2 zur Abbildung des Einheitsquadrats auf die beiden Vierecke erhalten wir durch Einsetzen der entsprechenden Rechteckspunkte $\boldsymbol{x}_i$ bzw. $\boldsymbol{x}'_i$ in Gl. 16.26, die inverse Transformation T_1^{-1} durch Invertieren der zugehörigen Abbildungsmatrix $\boldsymbol{A}_1$ (Gl. 16.21–16.24). Die Gesamttransformation T ergibt sich schließlich durch Verkettung der Transformationen T_1^{-1} und T_2, d. h.

$$\boldsymbol{x}' \;=\; T\big(\boldsymbol{x}\big) \;=\; T_2\big(T_1^{-1}(\boldsymbol{x})\big) \tag{16.28}$$

beziehungsweise in Matrixschreibweise

$$\boldsymbol{x}' \;=\; \boldsymbol{A}\cdot\boldsymbol{x} \;=\; \boldsymbol{A}_2\cdot\boldsymbol{A}_1^{-1}\cdot\boldsymbol{x}\,. \tag{16.29}$$

Die Abbildungsmatrix $\boldsymbol{A} = \boldsymbol{A}_2\cdot\boldsymbol{A}_1^{-1}$ muss für eine bestimmte Abbildung natürlich nur einmal berechnet werden und kann dann auf beliebig viele Bildpunkte $\boldsymbol{x}_i$ angewandt werden.

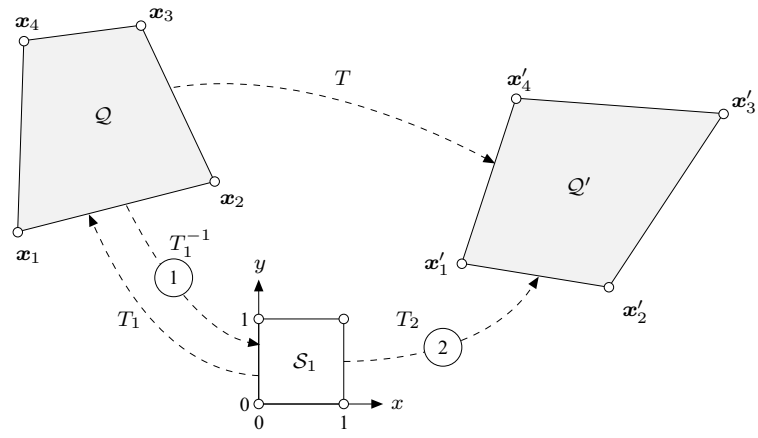


Abbildung 16.5
Projektive Abbildung zwischen den beliebigen Vierecken (*quadrilaterals*) $\mathcal{Q}$ und $\mathcal{Q}'$ durch zweistufige Transformation über das Einheitsquadrat $\mathcal{S}_1$. In Schritt 1 wird das ursprüngliche Viereck $\mathcal{Q}$ über T_1^{-1} auf das Einheitsquadrat $\mathcal{S}_1$ abgebildet. T_2 transformiert dann in Schritt 2 das Quadrat $\mathcal{S}_1$ auf das Zielviereck $\mathcal{Q}'$. Die Verkettung von T_1^{-1} und T_2 ergibt die Gesamttransformation T.

Beispiel

Das Ausgangsviereck $\mathcal{Q}$ und das Zielviereck $\mathcal{Q}'$ sind definiert durch folgende Koordinatenpunkte:

$$\begin{array}{lllll} \mathcal{Q}: & \boldsymbol{x}_1 = (2,5) & \boldsymbol{x}_2 = (4,6) & \boldsymbol{x}_3 = (7,9) & \boldsymbol{x}_4 = (5,9) \\ \mathcal{Q}': & \boldsymbol{x}'_1 = (4,3) & \boldsymbol{x}'_2 = (5,2) & \boldsymbol{x}'_3 = (9,3) & \boldsymbol{x}'_4 = (7,5) \end{array}$$

Daraus ergeben sich in Bezug auf das Einheitsquadrat $\mathcal{S}_1$ die projektiven Abbildungen $\boldsymbol{A}_1 : \mathcal{S}_1 \to \mathcal{Q}$ und $\boldsymbol{A}_2 : \mathcal{S}_1 \to \mathcal{Q}'$ mit

$$\boldsymbol{A}_1 = \begin{pmatrix} 3.3\dot{3} & 0.50 & 2.00 \\ 3.00 & -0.50 & 5.00 \\ 0.3\dot{3} & -0.50 & 1.00 \end{pmatrix} \quad \boldsymbol{A}_2 = \begin{pmatrix} 1.00 & -0.50 & 4.00 \\ -1.00 & -0.50 & 3.00 \\ 0.00 & -0.50 & 1.00 \end{pmatrix}$$

Durch Verkettung von $\boldsymbol{A}_2$ mit der inversen Abbildung $\boldsymbol{A}_1^{-1}$ erhalten wir schließlich die Gesamttransformation $\boldsymbol{A} = \boldsymbol{A}_2 \cdot \boldsymbol{A}_1^{-1}$, wobei

$$\boldsymbol{A}_1^{-1} = \begin{pmatrix} 0.60 & -0.45 & 1.05 \\ -0.40 & 0.80 & -3.20 \\ -0.40 & 0.55 & -0.95 \end{pmatrix} \quad \boldsymbol{A} = \begin{pmatrix} -0.80 & 1.35 & -1.15 \\ -1.60 & 1.70 & -2.30 \\ -0.20 & 0.15 & 0.65 \end{pmatrix}$$

Die Java-Methode `makeMapping()` der Klasse `ProjectiveMapping` (S. 402) ist eine Implementierung dieser Berechnung.

16.1.5 Bilineare Abbildung

Die *bilineare* Abbildung

$$\begin{array}{ll} T_x: & x' = a_1 x + a_2 y + a_3 xy + a_4 \\ T_y: & y' = b_1 x + b_2 y + b_3 xy + b_4 \end{array} \tag{16.30}$$

weist wie die projektive Abbildung (Gl. 16.16) acht Parameter ($a_1 \ldots a_4$, $b_1 \ldots b_4$) auf und kann durch vier Punktpaare spezifiziert werden. Durch

den gemischten Term xy ist die bilineare Transformation selbst im homogenen Koordinatensystem nicht als lineare Abbildung darzustellen. Im Unterschied zur projektiven Abbildung bleiben daher Geraden im Allgemeinen nicht erhalten, sondern gehen in quadratische Kurven über, auch Kreise werden nicht in Ellipsen abgebildet.

Eine bilineare Abbildung wird durch vier korrespondierende Punktpaare $(\boldsymbol{x}_1, \boldsymbol{x}'_1) \ldots (\boldsymbol{x}_4, \boldsymbol{x}'_4)$ eindeutig spezifiziert. Im allgemeinen Fall, also für die Abbildung zwischen beliebigen Vierecken, können die Koeffizienten $a_1 \ldots a_4, b_1 \ldots b_4$ als Lösung von zwei getrennten Gleichungssystemen mit jeweils vier Unbekannten bestimmt werden:

$$\begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & x_1y_1 & 1 \\ x_2 & y_2 & x_2y_2 & 1 \\ x_3 & y_3 & x_3y_3 & 1 \\ x_4 & y_4 & x_4y_4 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \qquad \text{bzw.} \quad X = \boldsymbol{M} \cdot \boldsymbol{a} \qquad (16.31)$$

$$\begin{pmatrix} y'_1 \\ y'_2 \\ y'_3 \\ y'_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & x_1y_1 & 1 \\ x_2 & y_2 & x_2y_2 & 1 \\ x_3 & y_3 & x_3y_3 & 1 \\ x_4 & y_4 & x_4y_4 & 1 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \qquad \text{bzw.} \quad Y = \boldsymbol{M} \cdot \boldsymbol{b} \qquad (16.32)$$

Eine konkrete Java-Implementierung dieser Berechnung dazu findet sich in der Methode `makeInverseMapping()` der Klasse `BilinearMapping` auf S. 404.

Für den speziellen Fall der Abbildung des *Einheitsquadrats* auf ein beliebiges Viereck $\mathcal{Q} = (\boldsymbol{x}'_1, \ldots \boldsymbol{x}'_4)$ durch die bilineare Transformation ist die Lösung für die Parameter $a_1 \ldots a_4$, $b_1 \ldots b_4$

$$\begin{aligned} a_1 &= x'_2 - x'_1 & b_1 &= y'_2 - y'_1 \\ a_2 &= x'_4 - x'_1 & b_2 &= y'_4 - y'_1 \\ a_3 &= x'_1 - x'_2 + x'_3 - x'_4 & b_3 &= y'_1 - y'_2 + y'_3 - y'_4 \\ a_4 &= x'_1 & b_4 &= y'_1 \end{aligned}$$

16.1.6 Weitere nichtlineare Bildverzerrungen

Die bilineare Transformation ist nur ein Beispiel für eine nichtlineare Abbildung im 2D-Raum, die nicht durch eine einfache Matrixmultiplikation in homogenen Koordinaten dargestellt werden kann. Darüber hinaus gibt es unzählige weitere nichtlineare Abbildungen, die in der Praxis etwa zur Realisierung diverser Verzerrungseffekte in der Bildgestaltung verwendet werden. Je nach Typ der Abbildung ist die Berechnung der inversen Abbildungsfunktion nicht immer einfach. In den folgenden drei Beispielen ist daher nur jeweils die Rückwärtstransformation

$$\boldsymbol{x} = T^{-1}(\boldsymbol{x}')$$

angegeben, sodass für die praktische Berechnung (durch *Target-to-Source Mapping*, siehe Abschn. 16.2.2) keine Inversion der Abbildungsfunktion erforderlich ist.

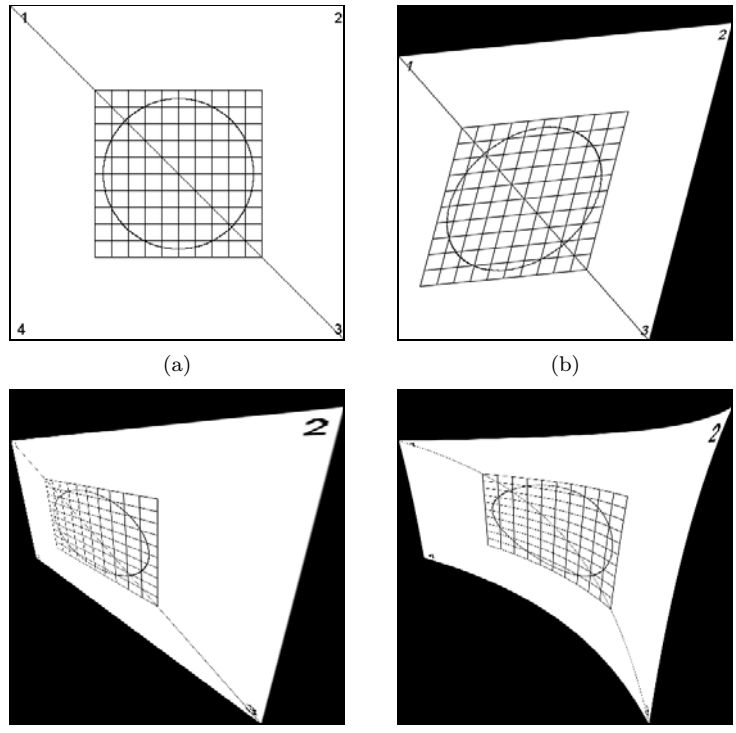


Abbildung 16.6
Geometrische Abbildungen im Vergleich. Originalbild (a), affine Abbildung in Bezug auf das Dreieck 1-2-3 (b), projektive Abbildung (c), bilineare Abbildung (d).

Twirl-Transformation

Die *Twirl*-Abbildung verursacht eine Drehung des Bilds um den vorgegebenen Mittelpunkt $\boldsymbol{x}_c = (x_c, y_c)$, wobei der Drehungswinkel im Zentrum einen vordefinierten Wert (α) aufweist und mit dem Abstand vom Zentrum proportional abnimmt. Außerhalb des Grenzradius $r_{\max}$ bleibt das Bild unverändert. Die zugehörige (inverse) Abbildungsfunktion ist folgendermaßen definiert:

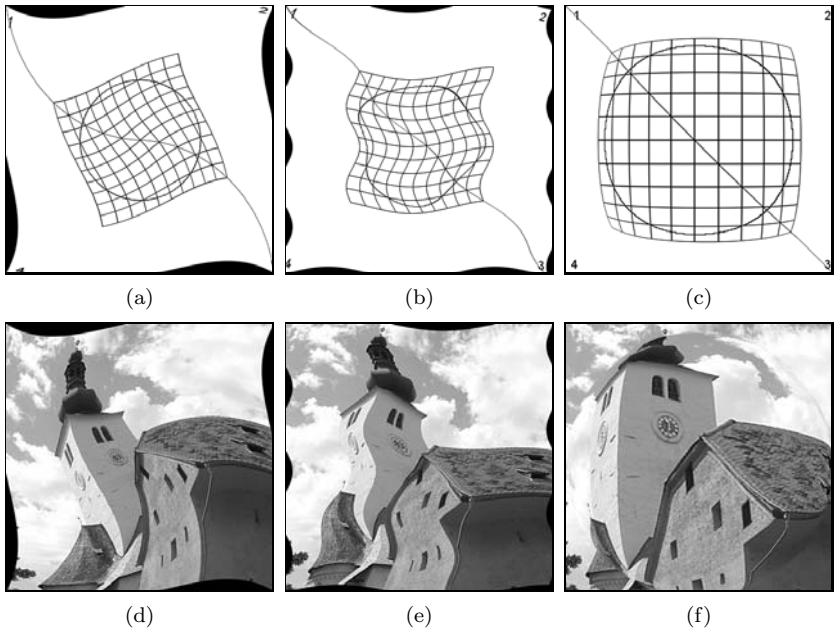
$$T_x^{-1}: \quad x = \begin{cases} x_c + r \cdot \cos(\beta) & \text{für } r \le r_{\max} \\ x' & \text{für } r > r_{\max} \end{cases} \tag{16.33}$$

$$T_y^{-1}: \quad y = \begin{cases} y_c + r \cdot \sin(\beta) & \text{für } r \le r_{\max} \\ y' & \text{für } r > r_{\max} \end{cases} \tag{16.34}$$

wobei

$$d_x = x' - x_c \qquad r = \sqrt{d_x^2 + d_y^2}$$
$$d_y = y' - y_c \qquad \beta = \arctan_2(d_y, d_x) + \alpha \cdot \left(\tfrac{r_{\max} - r}{r_{\max}}\right)$$

Abb. 16.7 (a,d) zeigt eine typische Twirl-Abbildung mit dem Drehpunkt $\boldsymbol{x}_c$ im Zentrum des Bilds, einem Grenzradius $r_{\max}$ mit der halben Länge der Bilddiagonale und einem Drehwinkel $\alpha = 43°$.



(a) (b) (c)

(d) (e) (f)

Abbildung 16.7
Diverse nichtlineare Bildverzerrungen. *Twirl* (a,d), *Ripple* (b,e), *Sphere* (c,f). Die Größe des Originalbilds ist 400×400 Pixel.

Ripple-Transformation

Die *Ripple*-Transformation bewirkt eine lokale, wellenförmige Verschiebung der Bildinhalte in x- und y-Richtung. Die Parameter dieser Abbildung sind die Periodenlängen $\tau_x, \tau_y \neq 0$ (in Pixel) für die Verschiebungen in beiden Richtungen sowie die zugehörigen Amplituden a_x, a_y:

$$\begin{aligned} T_x^{-1}: \quad & x = x' + a_x \cdot \sin\left(\tfrac{2\pi \cdot y'}{\tau_x}\right) \\ T_y^{-1}: \quad & y = y' + a_y \cdot \sin\left(\tfrac{2\pi \cdot x'}{\tau_y}\right) \end{aligned} \tag{16.35}$$

Abb. 16.7 (b,e) zeigt als Beispiel eine Ripple-Transformation mit $\tau_x = 120$, $\tau_y = 250$, $a_x = 10$ und $a_y = 15$.

Sphärische Verzerrung

Die sphärische Verzerrung bildet den Effekt einer auf dem Bild liegenden, halbkugelförmigen Glaslinse nach. Die Parameter dieser Abbildung sind das Zentrum der Linse $\boldsymbol{x}_c = (x_c, y_c)$, deren Radius $r_{\max}$ sowie der Brechungsindex der Linse ρ. Die Abbildung ist folgendermaßen definiert:

$$\begin{aligned} T_x^{-1}: \quad & x = x' - \begin{cases} z \cdot \tan(\beta_x) & \text{für } r \leq r_{\max} \\ 0 & \text{für } r > r_{\max} \end{cases} \\ T_y^{-1}: \quad & y = y' - \begin{cases} z \cdot \tan(\beta_y) & \text{für } r \leq r_{\max} \\ 0 & \text{für } r > r_{\max} \end{cases} \end{aligned} \tag{16.36}$$

wobei

$$d_x = x' - x_c, \quad r = \sqrt{d_x^2 + d_y^2}\,, \qquad \beta_x = \left(1 - \tfrac{1}{\rho}\right) \cdot \sin^{-1}\left(\tfrac{d_x}{\sqrt{(d_x^2 + z^2)}}\right),$$
$$d_y = y' - y_c, \quad z = \sqrt{r_{\max}^2 - r^2}\,, \quad \beta_y = \left(1 - \tfrac{1}{\rho}\right) \cdot \sin^{-1}\left(\tfrac{d_y}{\sqrt{(d_y^2 + z^2)}}\right).$$

Abb. 16.7 (c, f) zeigt eine sphärische Abbildung, bei der die Linse einen Radius $r_{\max}$ mit der Hälfte der Bildbreite und einen Brechungsindex $\rho = 1.8$ aufweist und das Zentrum $\boldsymbol{x}_c$ im Abstand von 10 Pixel rechts der Bildmitte liegt.

16.1.7 Lokale Transformationen

Die bisher beschriebenen geometrischen Transformationen sind *globaler* Natur, d. h., auf alle Bildkoordinaten wird dieselbe Abbildungsfunktion angewandt. Häufig ist es notwendig, ein Bild so zu verzerren, dass eine größere Zahl von Bildpunkten $\boldsymbol{x}_1 \ldots \boldsymbol{x}_n$ exakt in vorgegebene neue Koordinatenpunkte $\boldsymbol{x}'_1 \ldots \boldsymbol{x}'_n$ abgebildet wird. Für $n = 3$ ist dieses Problem mit einer affinen Abbildung (Abschn. 16.1.3) zu lösen bzw. mit einer projektiven oder bilinearen Abbildung für $n = 4$ abzubildende Punkte (Abschn. 16.1.4, 16.1.5). Für $n > 4$ ist auf Basis einer globalen Koordinatentransformation eine entsprechend komplizierte Funktion $T(\boldsymbol{x})$, z. B. ein Polynom höherer Ordnung, erforderlich.

Eine Alternative dazu sind *lokale* oder stückweise Abbildungen, bei denen die einzelnen Teile des Bilds mit unterschiedlichen, aber aufeinander abgestimmten Abbildungsfunktionen transformiert werden. In der Praxis sind vor allem netzförmige Partitionierungen des Bilds in der Form von Drei- oder Viereckflächen üblich, wie in Abb. 16.8 dargestellt.

Bei der Partitionierung des Bilds in ein *Mesh* von Dreiecken $\mathcal{D}_i$ (Abb. 16.8 (a)) kann für die Transformation zwischen zugehörigen Paaren von Dreiecken $\mathcal{D}_i \rightarrow \mathcal{D}'_i$ eine affine Abbildung verwendet werden, die natürlich für jedes Paar von Dreiecken getrennt berechnet werden muss. Für den Fall einer Mesh-Partitionierung in Vierecke $\mathcal{Q}_i$ (Abb. 16.8 (b)) eignet sich hingegen die projektive Abbildung. In beiden Fällen ist durch die Erhaltung der Geradeneigenschaft bei der Transformation sichergestellt, dass zwischen aneinander liegenden Drei- bzw. Vierecken kontinuierliche Übergänge und keine Lücken entstehen.

Lokale Transformationen dieser Art werden beispielsweise zur Entzerrung und Registrierung von Luft- und Satellitenaufnahmen verwendet. Auch beim so genannten „Morphing“ [89], das ist die schrittweise geometrische Überführung eines Bilds in ein anderes Bild bei gleichzeitiger Überblendung, kommt dieses Verfahren häufig zum Einsatz.[3]

[3] Image Morphing ist z. B. als ImageJ-Plugin `iMorph` von Hajime Hirase implementiert (http://rsb.info.nih.gov/ij/plugins/morph.html).

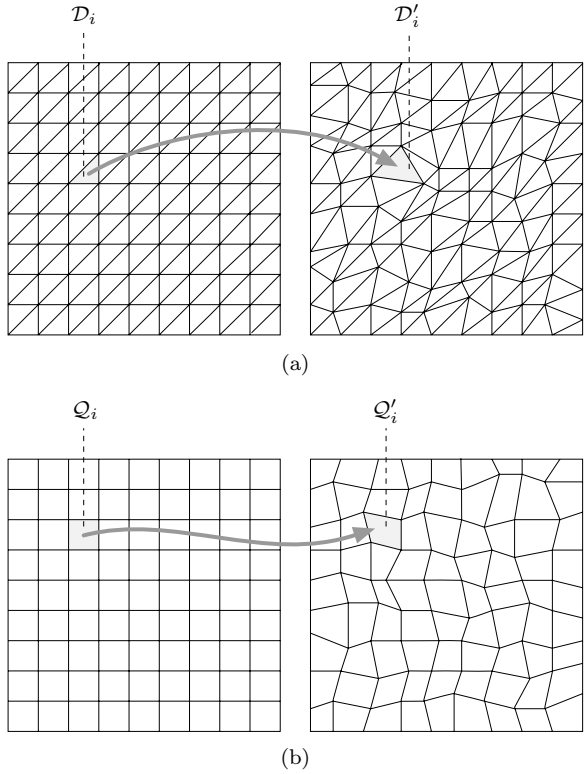


Abbildung 16.8
Beispiele für *Mesh*-Partitionierungen. Durch Zerlegung der Bildfläche in nicht überlappende Dreiecke $\mathcal{D}_i, \mathcal{D}'_i$ (a) oder Vierecke $\mathcal{Q}_i, \mathcal{Q}'_i$ (b) können praktisch beliebige Verzerrungen durch einfache, lokale Transformationen realisiert werden. Jedes Mesh-Element wird separat transformiert, die entsprechenden Abbildungsparameter werden jeweils aus den korrespondierenden 3 bzw. 4 Punktpaaren berechnet.

16.2 Resampling

Bei der Betrachtung der geometrischen Transformationen sind wir bisher davon ausgegangen, dass die Bildkoordinaten *kontinuierlich* (reellwertig) sind. Im Unterschied dazu liegen aber die Elemente digitaler Bilder auf *diskreten*, also ganzzahligen Koordinaten und ein nicht triviales Detailproblem bei geometrischen Transformationen ist die möglichst verlustfreie Überführung des diskret gerasterten Ausgangsbilds in ein neues, ebenfalls diskret gerastertes Zielbild.

Es ist also erforderlich, basierend auf einer geometrischen Abbildungsfunktion $T(x, y)$, aus einem bestehenden Bild $I(u, v)$ ein transformiertes Bild $I'(u', v')$ zu erzeugen, wobei alle Koordinaten diskret sind, d. h. $u, v \in \mathbb{Z}$ und $u', v' \in \mathbb{Z}$.[4] Dazu sind grundsätzlich folgende zwei Vorgangsweisen denkbar, die sich durch die Richtung der Abbildung unterscheiden: *Source-to-Target* bzw. *Target-to-Source Mapping*.

[4] Anm. zur Notation: Ganzzahlige Koordinaten werden mit (u, v) bzw. (u', v') bezeichnet, reellwertige Koordinaten mit (x, y) bzw. (x', y').

16.2.1 *Source-to-Target Mapping*

In diesem auf den ersten Blick plausiblen Ansatz wird für jedes Pixel (u, v) im Ausgangsbild I (*source*) die zugehörige transformierte Position

$$(x', y') = T(u, v)$$

im Zielbild I' (*target*) berechnet, die natürlich im Allgemeinen *nicht* auf einem Rasterpunkt liegt (Abb. 16.9). Anschließend ist zu entscheiden, in welches Bildelement in I' der zugehörige Intensitäts- oder Farbwert aus $I(u, v)$ gespeichert wird, oder ob der Wert eventuell sogar auf mehrere Pixel in I' verteilt werden soll.

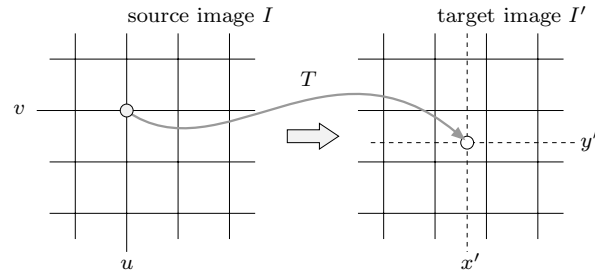


Abbildung 16.9
Source-to-Target Mapping. Für jede diskrete Pixelposition (u, v) im Ausgangsbild (*source*) I wird die zugehörige transformierte Position $(x', y') = T(u, v)$ im Zielbild (*target*) I' berechnet, die i. Allg. nicht auf einem Rasterpunkt liegt. Der Pixelwert $I(u, v)$ wird in eines der Bildelemente (oder in mehrere Bildelemente) in I' übertragen.

Das eigentliche Problem dieses Verfahrens ist, dass – abhängig von der geometrischen Transformation T – einzelne Elemente im Zielbild I' möglicherweise überhaupt nicht getroffen werden, z. B. wenn das Bild vergrößert wird. In diesem Fall entstehen Lücken in der Intensitätsfunktion, die nachträglich nur mühsam zu schließen wären. Umgekehrt müsste man auch berücksichtigen, dass (z. B. bei einer Verkleinerung des Bilds) ein Bildelement im Target I' durch *mehrere* Quellpixel hintereinander „getroffen“ werden kann, und dabei eventuell Bildinformation verloren geht.

16.2.2 *Target-to-Source Mapping*

Dieses Verfahren geht genau umgekehrt vor, indem für jeden Rasterpunkt (u', v') im Zielbild zunächst die zugehörige Position

$$(x, y) = T^{-1}(u', v')$$

im Ausgangsbild berechnet wird. Natürlich liegt auch diese Position i. Allg. wiederum nicht auf einem Rasterpunkt (Abb. 16.10) und es ist zu entscheiden, aus welchem (oder aus welchen) der Pixel in I die entsprechenden Bildwerte entnommen werden sollen. Dieses Problem der *Interpolation* der Intensitätswerte betrachten wir anschließend (in Abschn. 16.3) noch ausführlicher.

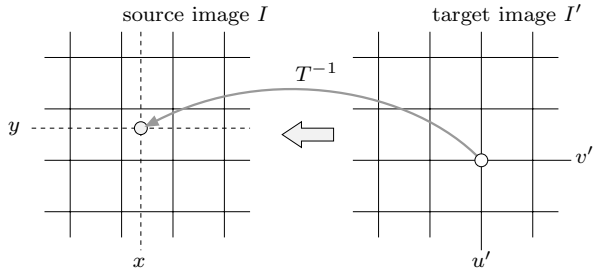


Abbildung 16.10
Target-to-Source Mapping. Für jede diskrete Pixelposition (u', v') im Zielbild (*target*) I' wird über die inverse Abbildungsfunktion T^{-1} die zugehörige Position $(x, y) = T^{-1}(u', v')$ im Ausgangsbild (*source*) berechnet. Der neue Pixelwert für $I'(u', v')$ wird durch Interpolation der Werte des Ausgangsbilds I in der Umgebung von (x, y) berechnet.

Der Vorteil des *Target-to-Source*-Verfahrens ist jedenfalls, dass garantiert alle Pixel des neuen Bilds I' (und nur diese) berechnet werden und damit keine Lücken oder Mehrfachtreffer entstehen können. Es erfordert die Verfügbarkeit der *inversen* geometrischen Abbildung T^{-1}, was allerdings in den meisten Fällen kein Nachteil ist, da die Vorwärtstransformation T selbst dabei gar nicht benötigt wird. Durch die Einfachheit des Verfahrens, die auch in Alg. 16.1 deutlich wird, ist *Target-to-Source Mapping* die gängige Vorgangsweise bei der geometrischen Transformation von Bildern.

```
1: TransformImage (I, T)
       I: source image
       T: coordinate transform function
2:     Create target image I'.
3:     for all target image coordinates (u', v') do
4:         Let (x, y) ← T^{-1}(u', v')
5:         I'(u', v') ← InterpolateValue(I, x, y)
6:     return target image I'.
```

Algorithmus 16.1
Geometrische Bildtransformation mit *Target-to-Source Mapping*. Gegeben sind das Ausgangsbild I, das Zielbild I' und die Koordinatentransformation T. Die Funktion InterpolateValue(I, x, y) berechnet den interpolierten Pixelwert an der kontinuierlichen Position (x, y) im Originalbild I.

16.3 Interpolation

Als *Interpolation* bezeichnet man den Vorgang, die Werte einer diskreten Funktion für Positionen abseits ihrer Stützstellen zu schätzen. Bei geometrischen Bildoperationen ergibt sich diese Aufgabenstellung aus dem Umstand, dass durch die geometrische Abbildung T (bzw. T^{-1}) diskrete Rasterpunkte im Allgemeinen *nicht* auf diskrete Bildpositionen im jeweils anderen Bild transformiert werden (wie im vorherigen Abschnitt beschrieben). Konkretes Ziel ist daher eine möglichst gute Schätzung für den Wert der zweidimensionalen Bildfunktion $I()$ für beliebige Positionen (x, y), insbesondere zwischen den bekannten, diskreten Bildpunkten $I(u, v)$.

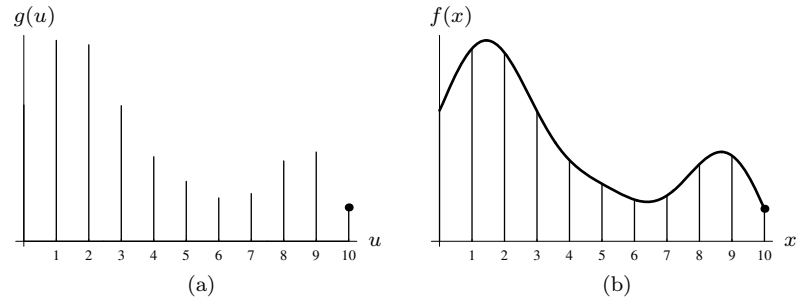


Abbildung 16.11
Interpolation einer diskreten Funktion. Die Aufgabe besteht darin, aus den diskreten Werten der Funktion $g(u)$ (a) die Werte der ursprünglichen Funktion $f(x)$ an beliebigen Positionen $x \in \mathbb{R}$ zu schätzen (b).

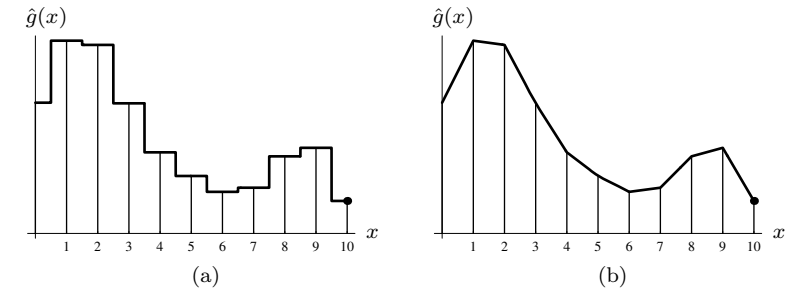


Abbildung 16.12
Einfache Interpolationsverfahren. Bei der *Nearest-Neighbor-Interpolation* (a) wird für jede kontinuierliche Position x der jeweils nächstliegende, diskrete Funktionswert $g(u)$ übernommen. Bei der *linearen Interpolation* (b) liegen die geschätzten Zwischenwerte auf Geraden, die benachbarte Funktionswerte $g(u)$ und $g(u+1)$ verbinden.

16.3.1 Einfache Interpolationsverfahren

Zur Illustration betrachten wir das Problem zunächst im eindimensionalen Fall (Abb. 16.11). Um die Werte einer diskreten Funktion $g(u)$, $u \in \mathbb{Z}$, an beliebigen Positionen $x \in \mathbb{R}$ zu interpolieren, gibt es verschiedene Ad-hoc-Ansätze. Am einfachsten ist es, die kontinuierliche Koordinate x auf den nächstliegenden ganzzahligen Wert u_0 zu runden und den zugehörigen Funktionswert $g(u_0)$ zu übernehmen, d. h.

$$\hat{g}(x) = g(u_0), \tag{16.37}$$

$$\text{wobei} \quad u_0 = \text{Round}(x) = \lfloor x + 0.5 \rfloor. \tag{16.38}$$

Das Ergebnis dieser so genannten *Nearest-Neighbor*-Interpolation ist anhand eines Beispiels in Abb. 16.12 (a) gezeigt.

Ein ähnlich einfaches Verfahren ist die *lineare Interpolation*, bei der die zu x links und rechts benachbarten Funktionsswerte $g(u_0)$ und $g(u_0+1)$, mit $u_0 = \lfloor x \rfloor$, proportional zum jeweiligen Abstand gewichtet werden:

$$\begin{aligned} \hat{g}(x) &= g(u_0) + (x - u_0) \cdot \big(g(u_0+1) - g(u_0)\big) \\ &= g(u_0) \cdot \big(1 - (x - u_0)\big) + g(u_0+1) \cdot (x - u_0) \end{aligned} \tag{16.39}$$

Wie in Abb. 16.12 (b) gezeigt, entspricht dies der stückweisen Verbindung der diskreten Funktionswerte durch Geradensegmente.

16.3.2 Ideale Interpolation

Offensichtlich sind aber die Ergebnisse dieser einfachen Interpolationsverfahren keine gute Annäherung an die ursprüngliche, kontinuierliche

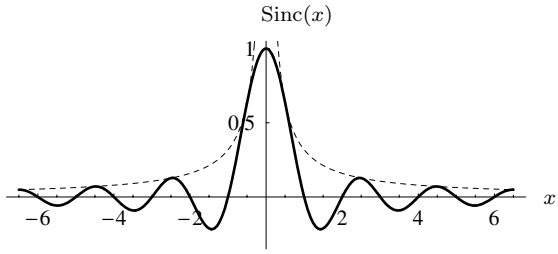


Abbildung 16.13
Sinc-Funktion in 1D. Die Funktion Sinc(x) weist an allen ganzzahligen Positionen Nullstellen auf und hat am Ursprung den Wert 1. Die unterbrochene Linie markiert die mit $|\frac{1}{x}|$ abfallende Amplitude der Funktion.

Funktion (Abb. 16.11). Man könnte sich fragen, wie es möglich wäre, die unbekannten Funktionswerte zwischen den diskreten Stützstellen noch besser anzunähern. Dies mag zunächst hoffnungslos erscheinen, denn schließlich könnte die diskrete Funktion $g(u)$ von unendlich vielen kontinuierlichen Funktionen stammen, deren Werte zwar an den diskreten Abtaststellen übereinstimmen, dazwischen jedoch beliebig sein können.

Die Antwort auf diese Frage ergibt sich (einmal mehr) aus der Betrachtung der Funktionen im Spektralbereich. Wenn bei der Diskretisierung des kontinuierlichen Signals $f(x)$ das *Abtasttheorem* (s. Abschn. 13.2.1) beachtet wurde, so bedeutet dies, dass $f(x)$ *bandbegrenzt* ist, also keine Frequenzkomponenten enthält, die über die Hälfte der Abtastfrequenz ω_s hinausgehen. Wenn aber im rekonstruierten Signal nur endlich viele Frequenzen auftreten können, dann ist damit auch dessen Form zwischen den diskreten Stützstellen entsprechend eingeschränkt.

Bei diesen Überlegungen sind absolute Größen nicht von Belang, da sich bei diskreten Signalen alle Frequenzwerte auf die Abtastfrequenz beziehen. Wenn wir also ein (dimensionsloses) Abtastintervall $\tau_s = 1$ annehmen, so ergibt sich daraus die Abtastfrequenz

$$\omega_s = 2\pi$$

und damit eine maximale Signalfrequenz $\omega_{\max} = \frac{\omega_s}{2} = \pi$. Um im zugehörigen (periodischen) Spektrum den Signalbereich $-\omega_{\max} \ldots \omega_{\max}$ zu isolieren, multiplizieren wir dieses Fourierspektrum (im Spektralraum) mit einer Rechteckfunktion der Breite $\pm\omega_{\max} = \pm\pi$. Im Ortsraum entspricht diese Operation einer linearen *Faltung* (Gl. 13.27, Tabelle 13.1) mit der zugehörigen Fouriertransformierten, das ist in diesem Fall die *Sinc*-Funktion

$$\mathrm{Sinc}(x) = \frac{\sin(\pi x)}{\pi x} \tag{16.40}$$

(Abb. 16.13). Dieser in Abschn. 13.1.6 beschriebene Zusammenhang zwischen dem Signalraum und dem Fourierspektrum ist in Abb. 16.14 nochmals übersichtlich dargestellt.

Theoretisch ist also Sinc(x) die ideale Interpolationsfunktion zur Rekonstruktion eines kontinuierlichen Signals. Um den interpolierten Wert der Funktion $g(u)$ für eine beliebige Position x_0 zu bestimmen, wird die Sinc-Funktion mit dem Ursprung an die Stelle x_0 verschoben und punktweise mit allen Werten von $g(u)$ – mit $u \in \mathbb{Z}$ – multipliziert und

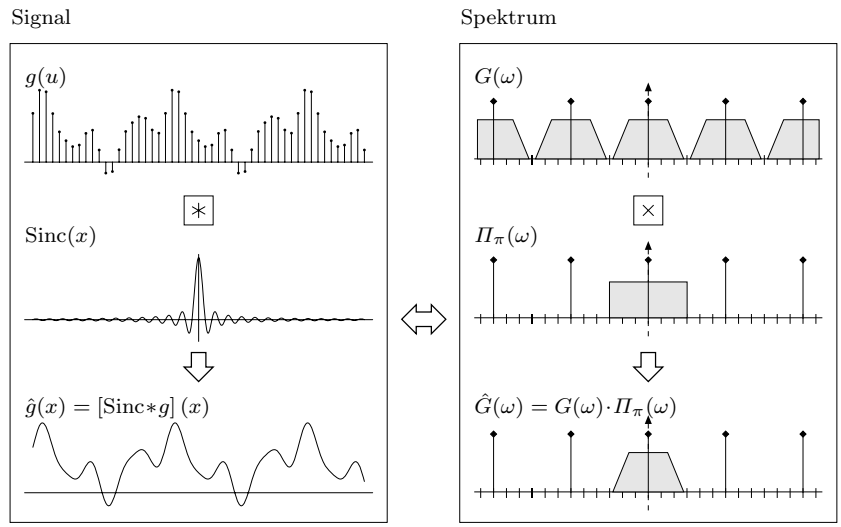


Abbildung 16.14
Interpolation eines diskreten Signals – Zusammenhang zwischen Signalraum und Fourierspektrum. Dem diskreten Signal $g(u)$ im Ortsraum (links) entspricht das periodische Fourierspektrum $G(\omega)$ im Spektralraum (rechts). Das Spektrum $\hat{G}(\omega)$ des kontinuierlichen Signals wird aus $G(\omega)$ durch Multiplikation ($\times$) mit der Rechteckfunktion $\Pi_\pi(\omega)$ isoliert. Im Ortsraum entspricht diese Operation einer linearen Faltung ($*$) mit der Funktion $\mathrm{Sinc}(x)$.

die Ergebnisse addiert, also „gefaltet“. Der rekonstruierte Wert der kontinuierlichen Funktion an der Stelle x_0 ist daher

$$\hat{g}(x_0) = [\mathrm{Sinc} * g]\,(x_0) = \sum_{u=-\infty}^{\infty} \mathrm{Sinc}(x_0 - u) \cdot g(u) \tag{16.41}$$

($*$ ist der Faltungsoperator, s. Abschn. 6.3.1). Ist das diskrete Signal $g(u)$, wie in der Praxis meist der Fall, *endlich* mit der Länge N, so wird es als periodisch angenommen, d. h., $g(u + kN) = g(u)$ für $k \in \mathbb{Z}$.[5] In diesem Fall ändert sich Gl. 16.41 zu

$$\hat{g}(x_0) = \sum_{u=-\infty}^{\infty} \mathrm{Sinc}(x_0 - u) \cdot g(u \bmod N) \tag{16.42}$$

Dabei mag die Tatsache überraschen, dass zur idealen Interpolation einer diskreten Funktion $g(u)$ an einer Stelle x_0 offensichtlich nicht nur einige wenige benachbarte Stützstellen zu berücksichtigen sind, sondern im Allgemeinen *unendlich viele Werte* von $g(u)$, deren Gewichtung mit der Entfernung von x_0 stetig (mit $|\frac{1}{x_0-u}|$) abnimmt. Die Sinc-Funktion nimmt allerdings nur langsam ab und benötigt daher für eine ausreichend genaue Rekonstruktion eine unpraktikabel große Zahl von Abtastwerten. Abb. 16.15 zeigt als Beispiel die Interpolation der Funktion $g(u)$ für die Positionen $x_0 = 4.4$ und $x_0 = 5$. Wird an einer ganzzahlige Position wie beispielsweise $x_0 = 5$ interpoliert, dann wird der Funktionswert $g(x_0)$ mit 1 gewichtet, während alle anderen Funktionswerte für $u \neq u_0$ mit

[5] Diese Annahme ist u. a. dadurch begründet, dass einem diskreten Fourierspektrum implizit ein periodisches Signal entspricht (s. auch Abschn. 13.2.2).

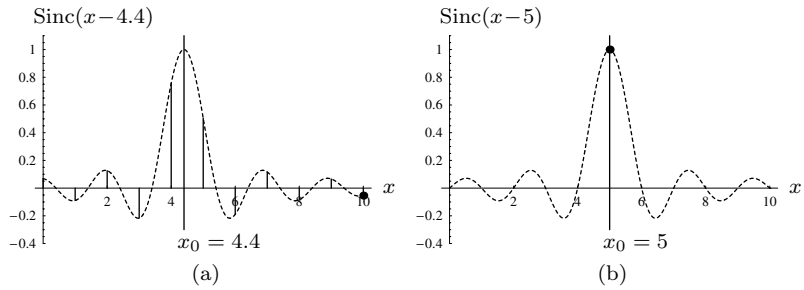


Abbildung 16.15
Interpolation durch Faltung mit der Sinc-Funktion. Die Sinc-Funktion wird mit dem Ursprung an die Interpolationsstelle $x_0 = 4.4$ (a) bzw. $x_0 = 5$ (b) verschoben. Die Werte der Sinc-Funktion an den ganzzahligen Positionen bilden die Koeffizienten für die zugehörigen Werte der diskreten Funktion $g(u)$. Bei der Interpolation für $x_0 = 4.4$ (a) wird das Ergebnis aus (unendlich) vielen Koeffizienten berechnet. Bei der Interpolation an der ganzzahligen Position $x_0 = 5$ (b), wird nur der Funktionswert $g(5)$ – gewichtet mit dem Koeffizienten 1 – berücksichtigt, alle anderen Signalwerte fallen mit den Nullstellen der Sinc-Funktion zusammen und tragen daher nicht zum Ergebnis bei.

den Nullstellen der Sinc-Funktion zusammenfallen und damit unberücksichtigt bleiben. Dadurch stimmen an den ganzzahligen Positionen die interpolierten Werte mit den entsprechenden Werten der diskreten Funktion exakt überein.

16.3.3 Interpolation durch Faltung

Für die Interpolation mithilfe der linearen Faltung können neben der Sinc-Funktion auch andere Funktionen als „Interpolationskern“ $w(x)$ verwendet werden. In allgemeinen Fall wird dann (analog zu Gl. 16.41) die Interpolation in der Form

$$\hat{g}(x_0) = [w * g](x_0) = \sum_{u=-\infty}^{\infty} w(x_0 - u) \cdot g(u) \tag{16.43}$$

berechnet. Beispielsweise kann die eindimensionale *Nearest-Neighbor*-Interpolation (Gl. 16.38, Abb. 16.12 (a)) durch eine Faltung mit dem Interpolationskern

$$w_{\text{nn}}(x) = \begin{cases} 1 & \text{für } -0.5 \le x < 0.5 \\ 0 & \text{sonst} \end{cases} \tag{16.44}$$

dargestellt werden, bzw. die *lineare* Interpolation (Gl. 16.39, Abb. 16.12 (b)) mit dem Kern

$$w_{\text{lin}}(x) = \begin{cases} 1 - x & \text{für } |x| < 1 \\ 0 & \text{für } |x| \ge 1 \end{cases} \tag{16.45}$$

Beide Interpolationskerne sind in Abb. 16.16 dargestellt.

16.3.4 Kubische Interpolation

Aufgrund des unendlich großen Interpolationskerns ist die Interpolation durch Faltung mit der Sinc-Funktion in der Praxis nicht realisierbar. Man versucht daher, auch aus Effizienzgründen, die ideale Interpolation durch kompaktere Interpolationskerne anzunähern. Eine häufig verwendete Annäherung ist die so genannte „kubische“ Interpolation, deren Interpolationskern durch stückweise, kubische Polynome folgendermaßen definiert ist:

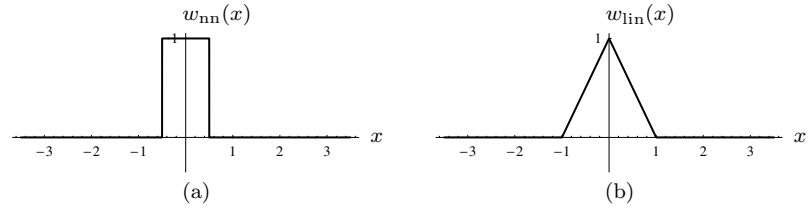


Abbildung 16.16
Interpolationskerne für Nearest-Neighbor-Interpolation $w_{\mathrm{nn}}(x)$ und lineare Interpolation $w_{\mathrm{lin}}(x)$.

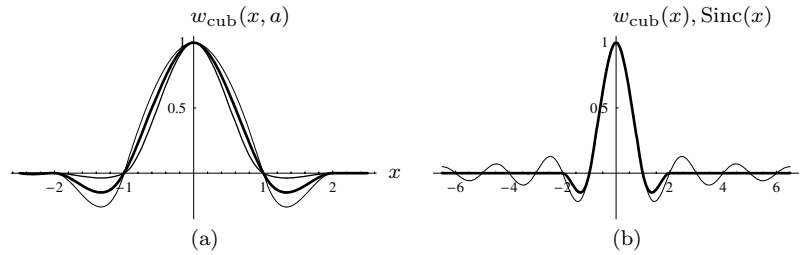


Abbildung 16.17
Kubischer Interpolationskern. Funktion $w_{\mathrm{cub}}(x,a)$ für die Werte $a = -0.25$ (mittelstarke Kurve), $a = -1$ (dicke Kurve) und $a = -1.75$ (dünne Kurve) (a). Kubische Funktion $w_{\mathrm{cub}}(x)$ und Sinc-Funktion im Vergleich (b).

$$w_{\mathrm{cub}}(x,a) = \begin{cases} (a+2)\cdot|x|^3 - (a+3)\cdot|x|^2 + 1 & \text{für } 0 \le |x| < 1 \\ a\cdot|x|^3 - 5a\cdot|x|^2 + 8a\cdot|x| - 4a & \text{für } 1 \le |x| < 2 \\ 0 & \text{für } |x| \ge 2 \end{cases} \tag{16.46}$$

Dabei ist a ein Steuerparameter, mit dem die Steilheit der Funktion bestimmt werden kann (Abb. 16.17 (a)). Für den Standardwert $a = -1$ ergibt sich folgende, vereinfachte Definition:

$$w_{\mathrm{cub}}(x) = \begin{cases} |x|^3 - 2\cdot|x|^2 + 1 & \text{für } 0 \le |x| < 1 \\ -|x|^3 + 5\cdot|x|^2 - 8\cdot|x| + 4 & \text{für } 1 \le |x| < 2 \\ 0 & \text{für } |x| \ge 2 \end{cases} \tag{16.47}$$

Der Vergleich zwischen der Sinc-Funktion und der kubischen Funktion $w_{\mathrm{cub}}(x)$ in Abb. 16.17 (b) zeigt, dass außerhalb von $x = \pm 2$ relativ große Koeffizienten unberücksichtigt bleiben, wodurch entsprechende Fehler zu erwarten sind. Allerdings ist die Interpolation wegen der Kompaktheit der kubischen Funktion sehr effizient zu berechnen. Da $w_{\mathrm{cub}}(x) = 0$ für $|x| \ge 2$, sind bei der Berechnung der Faltungsoperation (Gl. 16.43) an jeder beliebigen Position $x_0 \in \mathbb{R}$ jeweils nur *vier* Werte der diskreten Funktion $g(u)$ zu berücksichtigen, nämlich

$$g(u_0-1),\ g(u_0),\ g(u_0+1),\ g(u_0+2), \quad \text{wobei } u_0 = \lfloor x_0 \rfloor.$$

Dadurch reduziert sich die eindimensionale kubische Interpolation auf die Berechnung des Ausdrucks

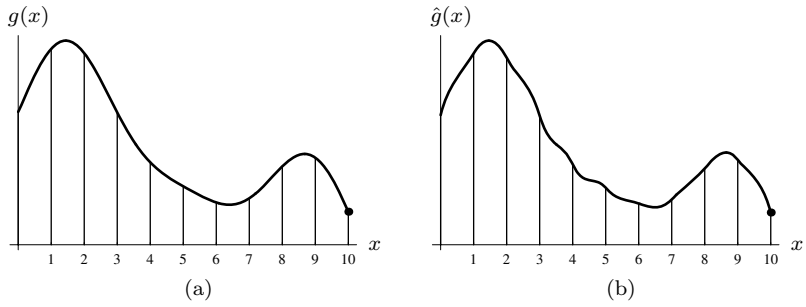


Abbildung 16.18
Kubische Interpolation. Ursprüngliches Signal bzw. ideale Rekonstruktion mit der Sinc-Interpolation (a), Interpolation mit kubischem Faltungskern nach Gl. 16.47–16.48 (b).

$$\hat{g}(x_0) = \sum_{u=\lfloor x_0 \rfloor - 1}^{\lfloor x_0 \rfloor + 2} w_{\text{cub}}(x_0 - u) \cdot g(u) \; . \tag{16.48}$$

16.3.5 Lanczos-Interpolation

Die Sinc-Funktion ist trotz ihrer Eigenschaft als ideale Interpolationsfunktion u. a. wegen ihrer unendlichen Ausdehnung nicht realisierbar. Während etwa bei der kubischen Interpolation eine polynomiale Approximation der Sinc-Funktion innerhalb eines kleinen Bereichs erfolgt, wird bei den so genannten „windowed sinc"-Verfahren die Sinc-Funktion selbst durch Gewichtung mit einer geeigneten Fensterfunktion $r(x)$ als Interpolationskern verwendet, d. h.

$$w(x) = r(x) \cdot \text{Sinc}(x) \; . \tag{16.49}$$

Als bekanntes Beispiel dafür verwendet die *Lanczos*[6]-Interpolation eine Fensterfunktion der Form

$$\text{L}n(x) = \begin{cases} \frac{\sin(\pi \frac{x}{n})}{\pi \frac{x}{n}} & \text{für } \; 0 \le |x| < n \\ 0 & \text{für } \; |x| \ge n \end{cases} \tag{16.50}$$

($n \in \mathbb{N}$ bezeichnet die Ordnung des Filters) [63, 85]. Interessanterweise ist also die Fensterfunktion selbst wiederum eine örtlich begrenzte Sinc-Funktion. Für die in der Bildverarbeitung am häufigsten verwendeten Lanczos-Filter der Ordnung $n = 2, 3$ sind die Fensterfunktionen daher

$$\text{L2}(x) = \begin{cases} \frac{\sin(\pi \frac{x}{2})}{\pi \frac{x}{2}} & \text{für } \; 0 \le |x| < 2 \\ 0 & \text{für } \; |x| \ge 2 \end{cases} \tag{16.51}$$

$$\text{L3}(x) = \begin{cases} \frac{\sin(\pi \frac{x}{3})}{\pi \frac{x}{3}} & \text{für } \; 0 \le |x| < 3 \\ 0 & \text{für } \; |x| \ge 3 \end{cases} \tag{16.52}$$

Beide Funktionen sind in Abb. 16.19(a,b) dargestellt. Die zugehörigen, eindimensionalen Interpolationskerne w_{L2} und w_{L3} ergeben sich durch

[6] Cornelius Lanczos (1893–1974).

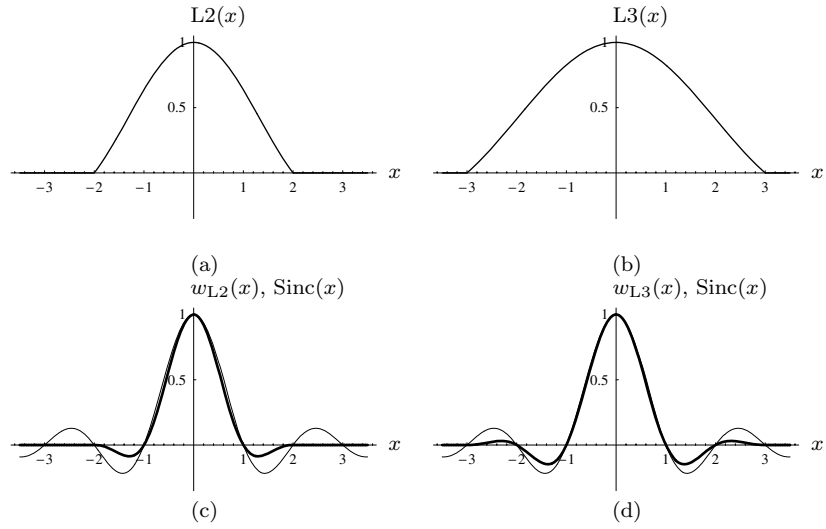


Abbildung 16.19
Eindimensionale Lanczos-Interpolationskerne. Lanczos-Fensterfunktionen L2 (a) und L3 (b). Die zugehörigen Interpolationskerne w_{L2} (c) und w_{L3} (d), zum Vergleich jeweils überlagert mit der Sinc-Funktion (dünne Linien).

Multiplikation der jeweiligen Fensterfunktion mit der Sinc-Funktion (Gl. 16.40, 16.50) als

$$w_{L2}(x) = L2(x) \cdot \frac{\sin(\pi x)}{\pi x} = \begin{cases} 2 \cdot \frac{\sin(\pi \frac{x}{2}) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{für } 0 \le |x| < 2 \\ 0 & \text{für } |x| \ge 2 \end{cases} \tag{16.53}$$

beziehungsweise

$$w_{L3}(x) = L3(x) \cdot \frac{\sin(\pi x)}{\pi x} = \begin{cases} 3 \cdot \frac{\sin(\pi \frac{x}{3}) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{für } 0 \le |x| < 3 \\ 0 & \text{für } |x| \ge 3 \end{cases} \tag{16.54}$$

Abb. 16.19(c,d) zeigt die resultierenden Interpolationskerne zusammen mit der ursprünglichen Sinc-Funktion. Die Funktion w_{L2} ist dem kubischen Interpolationskern w_{cub} (Gl. 16.46 mit dem Standardwert $a = -1$, s. Abb. 16.17) sehr ähnlich, daher sind auch bei der Interpolation ähnliche Ergebnisse zu erwarten. Das „3-tap“ Filter w_{L3} hingegen berücksichtigt deutlich mehr Abtastwerte und liefert daher potentiell bessere Interpolationsergebnisse.

16.3.6 Interpolation in 2D

In dem für die Interpolation von Bildfunktionen interessanten zweidimensionalen Fall sind die Verhältnisse naturgemäß ähnlich. Genau wie bei eindimensionalen Signalen besteht die ideale Interpolation aus einer linearen Faltung mit der zweidimensionalen Sinc-Funktion

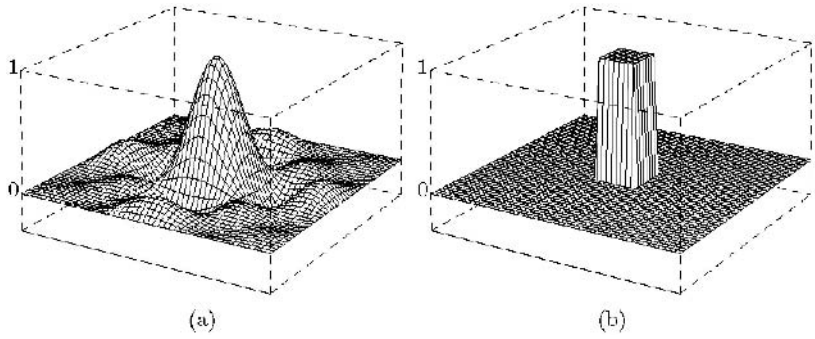


Abbildung 16.20
Interpolationskerne in 2D. Idealer Interpolationskern Sinc(x, y) (a), Nearest-Neighbor-Interpolationskern (b) für $-3 \le x, y \le 3$.

$$\text{Sinc}(x, y) = \text{Sinc}(x) \cdot \text{Sinc}(y) = \frac{\sin(\pi x)}{\pi x} \cdot \frac{\sin(\pi y)}{\pi y} \tag{16.55}$$

(Abb. 16.20 (a)), die natürlich in der Praxis nicht realisierbar ist. Gängige Verfahren sind hingegen, neben der im Anschluss beschriebenen (jedoch selten verwendeten) *Nearest-Neighbor*-Interpolation, die *bilineare*, *bikubische* und die *Lanczos*-Interpolation, die sich direkt von den bereits beschriebenen, eindimensionalen Varianten ableiten.

Nearest-Neighbor-Interpolation in 2D

Zur Bestimmung der zu einem beliebigen Punkt (x_0, y_0) nächstliegenden Pixelkoordinate (u_0, v_0) genügt es, die x- und y-Komponenten unabhängig auf ganzzahlige Werte zu runden, d. h.

$$\hat{I}(x_0, y_0) = I(u_0, v_0), \quad \text{wobei} \begin{cases} u_0 = \text{Round}(x_0) = \lfloor x_0 + 0.5 \rfloor \\ v_0 = \text{Round}(y_0) = \lfloor y_0 + 0.5 \rfloor \end{cases} \tag{16.56}$$

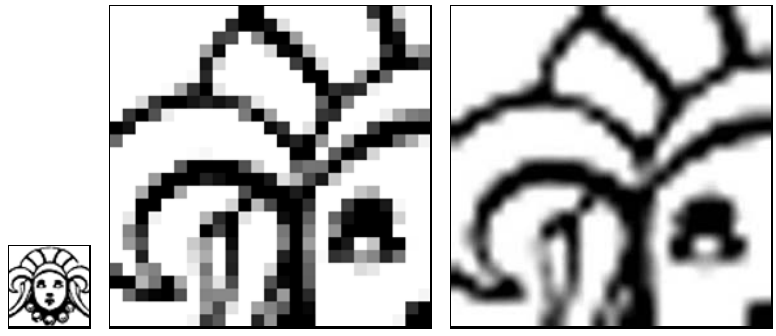
Der zugehörige 2D-Interpolationskern ist in Abb. 16.20 (b) dargestellt. In der Praxis wird diese Form der Interpolation heute nur mehr in Ausnahmefällen verwendet, etwa wenn bei der Vergrößerung eines Bilds die Pixel absichtlich als Blöcke mit einheitlicher Intensität ohne weiche Übergänge erscheinen sollen (Abb. 16.21 (b)).

Bilineare Interpolation

Das Gegenstück zur linearen Interpolation im eindimensionalen Fall ist die so genannte *bilineare* Interpolation[7], deren Arbeitsweise in Abb. 16.22 dargestellt ist. Dabei werden zunächst die zur Koordinate (x_0, y_0) nächstliegenden vier Bildwerte A, B, C, D mit

$$\begin{aligned} A &= I(u_0, v_0) & B &= I(u_0+1, v_0) \\ C &= I(u_0, v_0+1) & D &= I(u_0+1, v_0+1) \end{aligned} \tag{16.57}$$

[7] Nicht zu verwechseln mit der bilinearen *Abbildung* (Transformation) in Abschn. 16.1.5.



(a) (b) (c)

Abbildung 16.21
Bildvergrößerung mit Nearest-Neighbor-Interpolation. Original (a), 8-fach vergrößerter Ausschnitt mit Nearest-Neighbor-Interpolation (b) und bikubischer Interpolation (c).

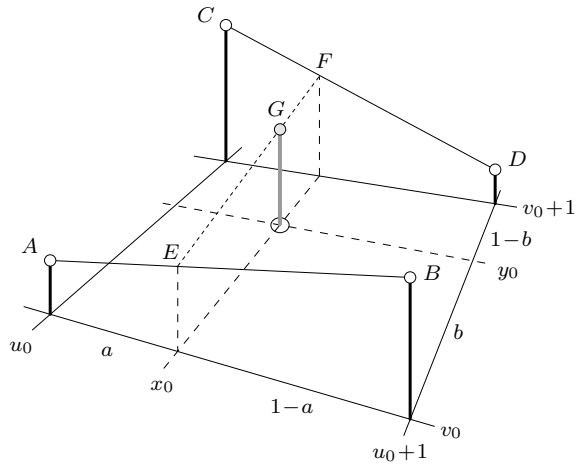


Abbildung 16.22
Bilineare Interpolation. Der Interpolationswert G für die Position (x_0, y_0) wird in zwei Schritten aus den zu (x_0, y_0) nächstliegenden Bildwerten A, B, C, D ermittelt. Zunächst werden durch lineare Interpolation über den Abstand zum Bildraster $a = (x_0 - u_0)$ die Zwischenwerte E und F bestimmt. Anschließend erfolgt ein weiterer Interpolationsschritt in vertikaler Richtung zwischen den Werten E und F, abhängig von der Distanz $b = (y_0 - v_0)$.

ermittelt, wobei $u_0 = \lfloor x_0 \rfloor$ und $v_0 = \lfloor y_0 \rfloor$, und anschließend jeweils in horizontaler und vertikaler Richtung linear interpoliert. Die Zwischenwerte E, F ergeben sich aus dem Abstand $a = (x_0 - u_0)$ der gegebenen Interpolationsstelle (x_0, y_0) von der Rasterkoordinate u_0 als

$$\begin{aligned} E &= A + (x_0 - u_0) \cdot (B - A) = A + a \cdot (B - A) \\ F &= C + (x_0 - u_0) \cdot (D - C) = C + a \cdot (D - C) \end{aligned}$$

und der finale Interpolationswert G aus dem Abstand $b = (y_0 - v_0)$ als

$$\begin{aligned} \hat{I}(x_0, y_0) &= G = E + (y_0 - v_0) \cdot (F - E) = E + b \cdot (F - E) \\ &= (a-1)(b-1)\, A + a(1-b)\, B + (1-a)\, b\, C + a\, b\, D\,. \end{aligned} \quad (16.58)$$

Als lineare Faltung formuliert ist der zugehörige zweidimensionale Interpolationskern $W_{\text{bilin}}(x, y)$ das Produkt der eindimensionalen Kerne $w_{\text{lin}}(x)$ und $w_{\text{lin}}(y)$ (Gl. 16.45), d. h.

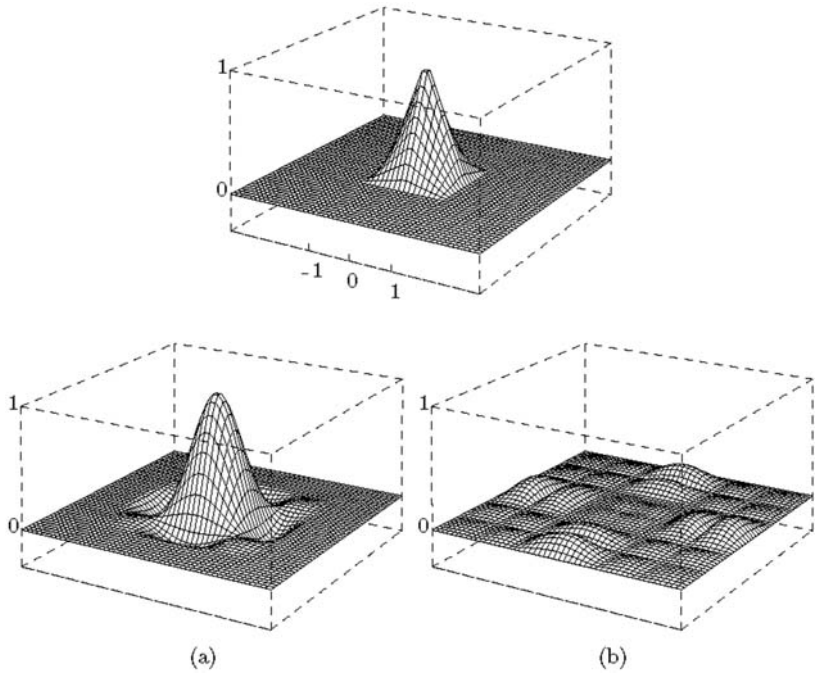


Abbildung 16.23
Faltungskern der bilinearen Interpolation in 2D. $W_{\text{bilin}}(x, y)$ für $-3 \leq x, y \leq 3$.

Abbildung 16.24
2D-Faltungskern für die bikubische Interpolation. Interpolationskern $W_{\text{bic}}(x, y)$ (a) und Differenz zur Sinc-Funktion, d. h. $|\text{Sinc}(x, y) - W_{\text{bic}}(x, y)|$ (b), jeweils für $-3 \leq x, y \leq 3$.

$$W_{\text{bilin}}(x, y) = w_{\text{lin}}(x) \cdot w_{\text{lin}}(y)$$
$$= \begin{cases} 1-x-y+xy & \text{für } 0 \leq |x|, |y| < 1 \\ 0 & \text{sonst.} \end{cases} \tag{16.59}$$

In dieser Funktion, die in Abb. 16.23 dargestellt ist, wird auch die „Bilinearform“ deutlich, die der Interpolationsmethode den Namen gibt.

Bikubische Interpolation

Auch der Faltungskern für die zweidimensionale kubische Interpolation besteht aus dem Produkt der zugehörigen eindimensionalen Kerne (Gl. 16.47),

$$W_{\text{bic}}(x, y) = w_{\text{cub}}(x) \cdot w_{\text{cub}}(y). \tag{16.60}$$

Diese Funktion ist in Abb. 16.24 dargestellt, zusammen mit der Differenz gegenüber dem idealen Interpolationskern $\text{Sinc}(x, y)$.

Die Berechnung der zweidimensionalen Interpolation ist daher (wie auch die vorher gezeigten Verfahren) in x- und y-Richtung *separierbar* und lässt sich gemäß Gl. 16.48 in folgender Form darstellen:

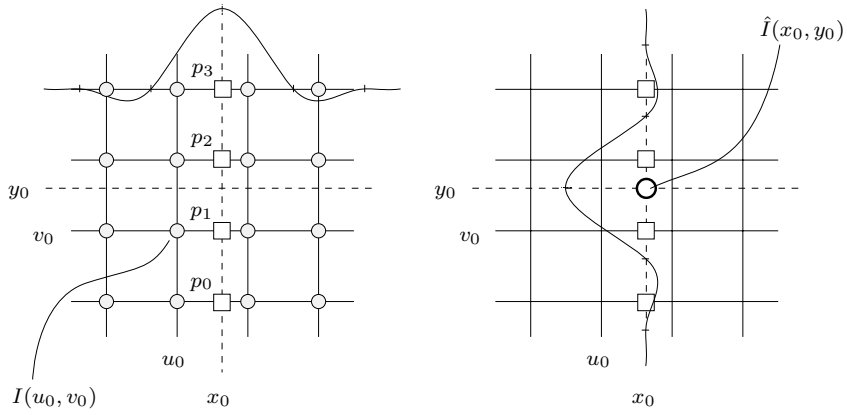


Abbildung 16.25
Bikubische Interpolation in 2 Schritten. Das diskrete Bild I (Pixel sind mit ○ markiert) soll an der Stelle (x_0, y_0) interpoliert werden. In **Schritt 1** (links) wird mit $w_{\text{cub}}(\cdot)$ horizontal über jeweils 4 Pixel $I(u_i, v_j)$ interpoliert und für jede betroffene Zeile ein Zwischenergebnis p_j (mit □ markiert) berechnet (Gl. 16.61). In **Schritt 2** (rechts) wird nur *einmal* vertikal über die Zwischenergebnisse $p_0 \ldots p_3$ interpoliert und damit das Ergebnis $\hat{I}(x_0, y_0)$ berechnet. Insgesamt sind somit $16 + 4 = 20$ Interpolationsschritte notwendig.

$$\begin{aligned}\tilde{I}(x_0, y_0) &= \sum_{\substack{v=\\ \lfloor y_0 \rfloor - 1}}^{\lfloor y_0 \rfloor + 2} \left[\sum_{\substack{u=\\ \lfloor x_0 \rfloor - 1}}^{\lfloor x_0 \rfloor + 2} I(u,v) \cdot W_{\text{bic}}(x_0 - u, y_0 - v) \right] \\ &= \sum_{j=0}^{3} \left[w_{\text{cub}}(y_0 - v_j) \cdot \underbrace{\sum_{i=0}^{3} I(u_i, v_j) \cdot w_{\text{cub}}(x_0 - u_i)}_{p_j} \right], \end{aligned} \tag{16.61}$$

wobei $u_i = \lfloor x_0 \rfloor - 1 + i$ und $v_j = \lfloor y_0 \rfloor - 1 + j$. Die nach Gl. 16.61 sehr einfache Berechnung der bikubischen Interpolation unter Verwendung des eindimensionalen Interpolationskerns $w_{\text{cub}}(x)$ ist in Abb. 16.25 schematisch dargestellt und in Alg. 16.2 auch nochmals zusammengefasst.

Algorithmus 16.2
Bikubische Interpolation des Bilds I an der Position (x_0, y_0). Die eindimensionale, kubische Interpolationsfunktion $w_{\text{cub}}(\cdot)$ wird für die separate Interpolation in der x- und y-Richtung verwendet (Gl. 16.46, 16.61), wobei eine Umgebung von 4×4 Bildpunkten berücksichtigt wird.

```
1:  BicubicInterpolation (I, x0, y0)                    ▷ x0, y0 ∈ ℝ
2:      q ← 0
3:      for j ← 0 ... 3 do
4:          v ← ⌊y0⌋ − 1 + j
5:          p ← 0
6:          for i ← 0 ... 3 do
7:              u ← ⌊x0⌋ − 1 + i
8:              p ← p + I(u, v) · w_cub(x0 − u)
9:          q ← q + p · w_cub(y0 − v)
10:     return q.
```

Lanczos-Interpolation in 2D

Die zweidimensionalen Interpolationskerne ergeben sich aus den eindimensionalen Lanczos-Kernen (Gl. 16.53, 16.54) analog zur zweidimensio-

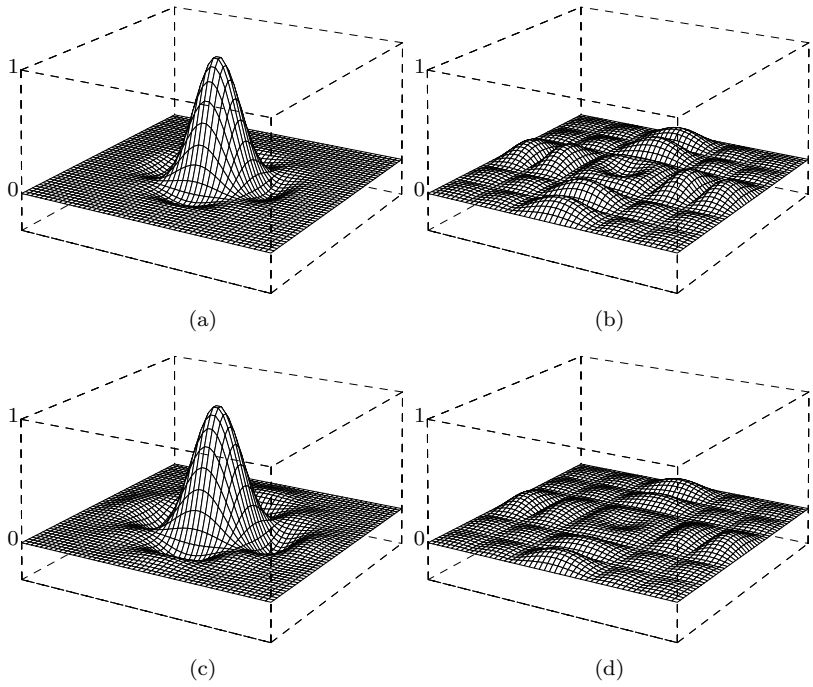


Abbildung 16.26
Zweidimensionale Lanczos-Interpolationskerne. Interpolationskern $W_{\mathrm{L}2}$ (a) und Differenz zur Sinc-Funktion $|\textsc{Sinc}(x,y) - W_{\mathrm{L}2}(x,y)|$ (b), jeweils für $-3 \le x, y \le 3$. Interpolationskern $W_{\mathrm{L}3}$ (c) und Differenz zur Sinc-Funktion (d).

nalen Sinc-Funktion (Gl. 16.55) in der Form

$$W_{\mathrm{L}n}(x,y) = w_{\mathrm{L}n}(x) \cdot w_{\mathrm{L}n}(y)\,. \tag{16.62}$$

Die Interpolationskerne $W_{\mathrm{L}2}(x,y)$ und $W_{\mathrm{L}3}(x,y)$ sind in Abb. 16.26 dargestellt, ebenso deren Abweichungen gegenüber der zweidimensionalen Sinc-Funktion.

Die Berechnung der Lanczos-Interpolation in 2D kann genauso wie bei der bikubischen Interpolation in x- und y-Richtung getrennt und hintereinander durchgeführt werden. Der Kern $W_{\mathrm{L}2}$ ist, genau wie der Kern der bikubischen Interpolation, außerhalb des Bereichs $-2 \le x, y \le 2$ null; das in Gl. 16.61 (bzw. Abb. 16.25 und Alg. 16.2) beschriebene Verfahren kann daher direkt übernommen werden. Für den größeren Kern $W_{\mathrm{L}3}$ erweitert sich der Interpolationsbereich gegenüber Gl. 16.61 um zwei zusätzliche Zeilen und Spalten. Die Berechnung des interpolierten Pixelwerts an der Stelle (x_0, y_0) erfolgt daher in der Form

$$\begin{aligned}\tilde{I}(x_0,y_0) &= \sum_{\substack{v=\\ \lfloor y_0\rfloor-2}}^{\lfloor y_0\rfloor+3} \left[\sum_{\substack{u=\\ \lfloor x_0\rfloor-2}}^{\lfloor x_0\rfloor+3} I(u,v)\cdot W_{\mathrm{L}3}(x_0-u, y_0-v)\right] \\ &= \sum_{j=0}^{5}\left[w_{\mathrm{L}3}(y_0-v_j)\cdot \sum_{i=0}^{5} I(u_i,v_j)\cdot w_{\mathrm{L}3}(x_0-u_i)\right], \end{aligned} \tag{16.63}$$

mit $u_i = \lfloor x_0 \rfloor - 2 + i$ und $v_j = \lfloor y_0 \rfloor - 2 + j$. Damit wird bei der zweidimensionalen Lanczos-Interpolation mit W_{L3}-Kern für einen Interpolationspunkt jeweils eine Umgebung von $6 \times 6 = 36$ Pixel des Originalbilds berücksichtigt, also um 20 Pixel mehr als bei der bikubischen Interpolation.

Beispiele

Abb. 16.27 zeigt die drei gängigsten Interpolationsverfahren im Vergleich. Das Originalbild, bestehend aus dunklen Linien auf grauem Hintergrund, wurde einer Drehung um 15° unterzogen.

Das Ergebnis der *Nearest-Neighbor*-Interpolation (Abb. 16.27 (b)) zeigt die erwarteten blockförmigen Strukturen und enthält keine Pixelwerte, die nicht bereits im Originalbild enthalten sind. Die *bilineare* Interpolation (Abb. 16.27 (c)) bewirkt im Prinzip eine lokale Glättung über vier benachbarte, positiv gewichtete Bildwerte und daher kann kein Ergebniswert kleiner als die Pixelwerte in seiner Umgebung sein. Dies ist bei der *bikubischen* Interpolation (Abb. 16.27 (d)) nicht der Fall: Durch die teilweise negativen Gewichte des kubischen Interpolationskerns entstehen zu beiden Seiten von Übergängen hellere bzw. dunklere Bildwerte, die sich auf dem grauen Hintergrund deutlich abheben und einen subjektiven Schärfungseffekt bewirken. Die bikubische Interpolation liefert bei ähnlichem Rechenaufwand deutlich bessere Ergebnisse als die bilineare Interpolation und gilt daher als Standardverfahren in praktisch allen gängigen Bildbearbeitungsprogrammen.

Die *Lanczos*-Interpolation (hier nicht gezeigt) mit W_{L2} unterscheidet sich weder im Rechenaufwand noch in der Qualität der Ergebnisse wesentlich von der bikubischen Interpolation. Hingegen bringt der Einsatz von Lanczos-Filtern der Form W_{L3} durch die größere Zahl der berücksichtigten Originalpixel eine geringfügige Verbesserung gegenüber der bikubischen Interpolation. Diese Methode wird daher trotz des erhöhten Rechenaufwands häufig für hochwertige Graphik-Anwendungen, Computerspiele und in der Videoverarbeitung eingesetzt.

16.3.7 Aliasing

Wie im Hauptteil dieses Kapitels dargestellt, besteht die übliche Vorgangsweise bei der Realisierung von geometrischen Abbildungen im Wesentlichen aus drei Schritten (Abb. 16.28):

1. Alle diskreten Bildpunkte (u'_0, v'_0) des Zielbilds (*target*) werden durch die inverse geometrische Transformation T^{-1} auf die Koordinaten (x_0, y_0) im Ausgangsbild projiziert.
2. Aus der diskreten Bildfunktion $I(u, v)$ des Ausgangsbilds wird durch Interpolation eine kontinuierliche Funktion $\hat{I}(x, y)$ rekonstruiert.
3. Die rekonstruierte Bildfunktion $\hat{I}$ wird an der Position (u'_0, v'_0) abgetastet und der zugehörige Abtastwert $\hat{I}(x_0, y_0)$ wird für das Targetpixel $I'(u'_0, v'_0)$ im Ergebnisbild übernommen.

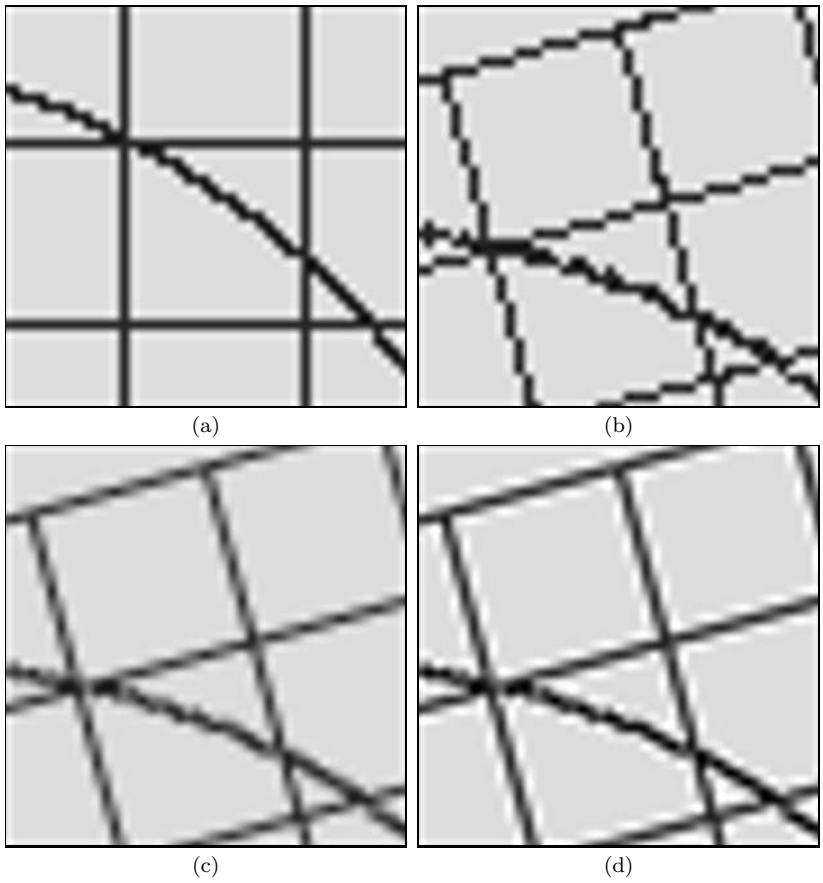


Abbildung 16.27
Interpolationsverfahren im Vergleich. Ausschnitt aus dem Originalbild (a), das einer Drehung um 15° unterzogen wird. Nearest-Neighbor-Interpolation (b), bilineare Interpolation (c), bikubische Interpolation (d).

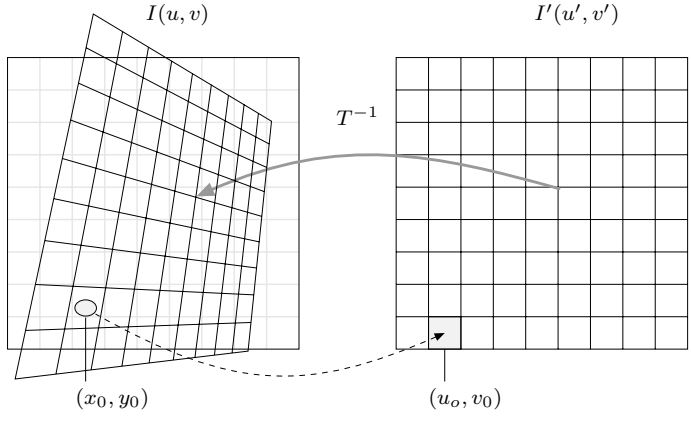


Abbildung 16.28
Abtastfehler durch geometrische Operationen. Bewirkt die geometrische Transformation T eine lokale Verkleinerung des Bilds (ensprechend einer Vergrößerung durch T^{-1}, wie im linken Teil des Rasters), so vergrößern sich die Abstände zwischen den Abtastpunkten in I. Dadurch reduziert sich die Abtastfrequenz und damit auch die zulässige Grenzfrequenz der Bildfunktion, was schließlich zu Abtastfehlern (Aliasing) führt.

Abtastung der rekonstruierten Bildfunktion

Ein Problem, das wir bisher nicht beachtet hatten, bezieht sich auf die Abtastung der rekonstruierten Bildfunktion im obigen Schritt 3. Für den Fall nämlich, dass durch die geometrische Transformation T in einem Teil des Bilds eine räumliche *Verkleinerung* erfolgt, vergrößern sich durch die inverse Transformation T^{-1} die Abstände zwischen den Abtastpunkten im Originalbild. Eine Vergrößerung dieser Abstände bedeutet jedoch eine Reduktion der Abtastrate und damit eine Reduktion der zulässigen Frequenzen in der kontinuierlichen (rekonstruierten) Bildfunktion $\hat{I}(x, y)$. Dies führt zu einer Verletzung des Abtastkriteriums und wird als „Aliasing“ im generierten Bild sichtbar.

Das Beispiel in Abb. 16.29 demonstriert, dass dieser Effekt von der verwendeten Interpolationsmethode weitgehend unabhängig ist. Besonders deutlich ausgeprägt ist er natürlich bei der Nearest-Neighbor-Interpolation, bei der die dünnen Linien an manchen Stellen einfach nicht mehr „getroffen“ werden und somit verschwinden. Dadurch geht wichtige Bildinformation verloren. Die bikubische Interpolation besitzt

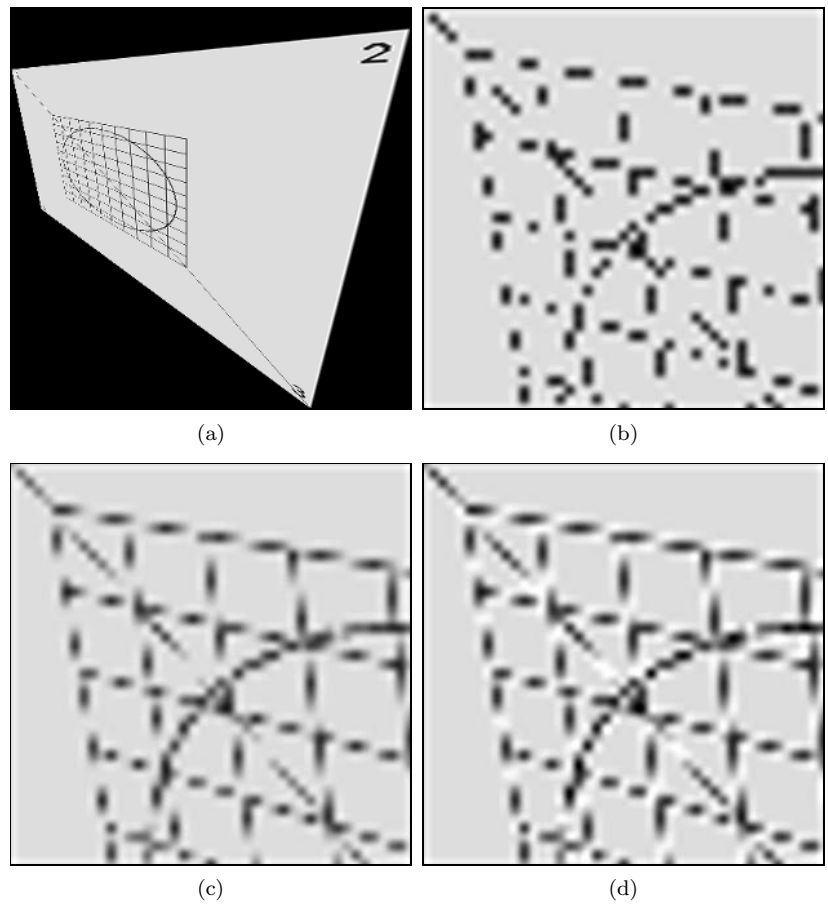


Abbildung 16.29
Aliasing-Effekt durch lokale Bildverkleinerung. Der Effekt ist weitgehend unabhängig vom verwendeten Interpolationsverfahren: transformiertes Gesamtbild (a), Nearest-Neighbor-Interpolation (b), bilineare Interpolation (c), bikubische Interpolation (d).

zwar den breitesten Interpolationskern, kann aber diesen Effekt ebenfalls nicht verhindern. Noch größere Maßstabsänderungen (z. B. bei eine Verkleinerung um den Faktor 8) wären ohne zusätzliche Maßnahmen überhaupt nicht zufriedenstellend durchführbar.

Tiefpassfilter

Eine Lösung dieses Problems besteht darin, die für die Abtastung erforderliche Bandbegrenzung der rekonstruierten Bildfunktion sicherzustellen. Dazu wird auf die rekonstruierte Bildfunktion vor der Abtastung ein entsprechendes Tiefpassfilter angewandt (Abb. 16.30).

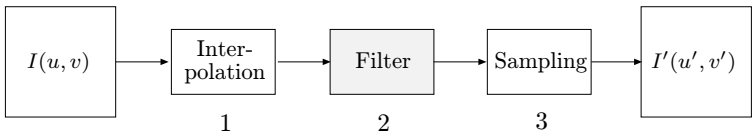


Abbildung 16.30
Tiefpassfilter zur Vermeidung von Aliasing. Die interpolierte Bildfunktion (nach Schritt 1) wird vor der Abtastung (Schritt 3) einem ortsabhängigen Tiefpassfilter unterzogen, das die Bandbreite des Bildsignals auf die lokale Abtastrate anpasst.

Am einfachsten ist dies bei einer Abbildung, bei der die Maßstabsänderung über das gesamte Bild gleichmäßig ist, wie z. B. bei einer globalen Skalierung oder einer affinen Transformation. Ist die Maßstabsänderung über das Bild jedoch ungleichmäßig, so ist ein Filter erforderlich, dessen Parameter von der geometrischen Abbildungsfunktion T und der aktuellen Bildposition abhängig ist. Wenn sowohl für die Interpolation und das Tiefpassfilter ein Faltungsfilter verwendet werden, so können diese in ein gemeinsames, ortsabhängiges Rekonstruktionsfilter zusammengefügt werden.

Die Anwendung derartiger ortsabhängiger (*space-variant*) Filter ist allerdings aufwendig und wird daher teilweise auch in professionellen Applikationen (wie z. B. in Adobe Photoshop) vermieden. Solche Verfahren sind jedoch u. a. bei der Projektion von Texturen in der Computergrafik von praktischer Bedeutung [23, 89].

16.4 Java-Implementierung

In ImageJ sind nur wenige, einfache geometrischen Operationen, wie horizontale Spiegelung und Rotation, in der Klasse `ImageProcessor` implementiert. Einige weitere Operationen, wie z. B. die affine Transformation, sind als Plugin-Klassen im `TransformJ`-Package verfügbar [57].

Im Folgenden zeigen wir eine rudimentäre Implementierung für geometrische Bildtransformationen in Java bzw. ImageJ, bestehend aus mehreren Klassen, deren Hierarchie in Abb. 16.31 zusammengefasst ist.

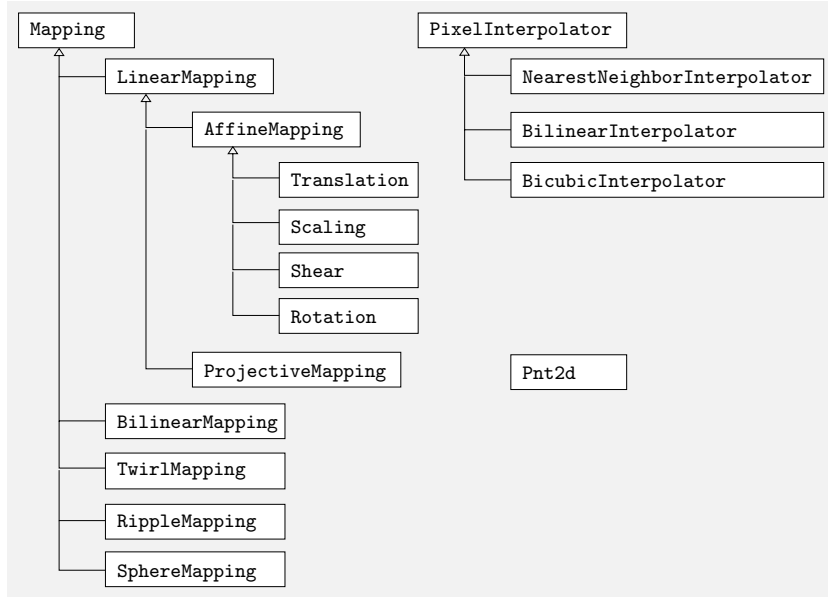


Abbildung 16.31
Klassenstruktur für die Java-Implementierung geometrischer Abbildungen und der Pixel-Interpolation.

Die Java-Klassen sind in zwei Gruppen geteilt: Die erste Gruppe betrifft die in Abschn. 16.1 dargestellten geometrischen Abbildungen,[8] die zweite Gruppe implementiert die wichtigsten der in Abschn. 16.3 beschriebenen Verfahren zur Pixel-Interpolation. Den Abschluss bildet ein Anwendungsbeispiel in Form eines einfachen ImageJ-Plugins.

16.4.1 Geometrische Abbildungen

Die folgenden Java-Klassen repräsentieren geometrische Abbildungen für zweidimensionale Koordinaten und stellen Methoden zur Berechnung der Abbildung aus vorgegebenen Punktpaaren zur Verfügung.

`Pnt2d` (Klasse)

Zweidimensionale Koordinatenpunkte $\boldsymbol{x} = (x, y) \in \mathbb{R}^2$ werden durch Objekte der Klasse `Pnt2d` repräsentiert:

```
1 public class Pnt2d {
2   double x, y;
3
4   Pnt2d (double x, double y){
5     this.x = x; this.y = y;
6   }
7 }
```

[8] In der Standardversion von Java ist derzeit nur die *affine Abbildung* als Klasse (java.awt.geom.AffineTransform) implementiert.

`Mapping` (Klasse)

Die abstrakte Klasse `Mapping` ist die Überklasse für alle nachfolgenden Abbildungsklassen. Sie schreibt die Methode `applyTo(Pnt2d pnt)` vor, womit die geometrische Abbildung auf den Koordinatenpunkt `pnt` angewandt wird und deren konkrete Implementierung durch jede der Subklassen (Abbildungen) erfolgt.

Demgegenüber ist die Methode `applyTo (ImageProcessor ip, PixelInterpolator intPol)` in der Klasse `Mapping` selbst implementiert (Zeile 21) und für alle Abbildungen gleich. Durch sie wird diese Koordinatentransformation auf ein ganzes Bild angewandt, wobei das Objekt `intPol` für die Interpolation der Pixelwerte sorgt (Zeile 38).

Die eigentliche Bildtransformation arbeitet nach dem *Target-to-Source*-Verfahren und benötigt dazu die inverse Koordinatentransformation T^{-1}, die durch die Methode `getInverse()` erzeugt wird (Zeile 14, 25). Falls die Abbildung keine Rückwärtsabbildung ist (`isInverse == false`), wird die Abbildung invertiert. Diese Inversion ist nur für lineare Abbildungen (Klasse `LinearMapping` und deren Subklassen) implementiert, bei den anderen Abbildungsklassen wird bereits zu Beginn eine Rückwärtstransformation erzeugt.

```
1  import ij.process.ImageProcessor;
2
3  public abstract class Mapping implements Cloneable {
4    boolean isInverse = false;
5
6    // subclasses must implement this method:
7    abstract Pnt2d applyTo(Pnt2d pnt);
8
9    Mapping invert() {
10     throw new
11       IllegalArgumentException("cannot invert mapping");
12   }
13
14   Mapping getInverse() {
15     if (isInverse)
16       return this;
17     else
18       return this.invert(); // only linear mappings invert
19   }
20
21   void applyTo(ImageProcessor ip, PixelInterpolator intPol){
22     ImageProcessor targetIp = ip;
23     ImageProcessor sourceIp = ip.duplicate();
24
25     Mapping invMap = this.getInverse(); // get inverse mapping
26     intPol.setImageProcessor(sourceIp);
27
28     int w = sourceIp.getWidth();
29     int h = sourceIp.getHeight();
```

```
30
31    Pnt2d pt = new Pnt2d(0,0);
32    for (int v=0; v<h; v++){
33      for (int u=0; u<w; u++){
34        pt.x = u;
35        pt.y = v;
36        invMap.applyTo(pt);
37        int p =
38          (int) Math.rint(intPol.getInterpolatedPixel(pt));
39        targetIp.putPixel(u,v,p);
40      }
41    }
42  }
43
44  Mapping duplicate() { //clones any mapping object
45    Mapping newMap = null;
46    try {
47      newMap = (Mapping) this.clone();
48    }
49    catch (CloneNotSupportedException e){};
50    return newMap;
51  }
52
53 }
```

`LinearMapping` (Klasse)

`LinearMapping` ist eine konkrete Subklasse von `Mapping` und realisiert eine beliebige lineare Abbildung mit homogenen Koordinaten in 2D. Die Abbildungsparameter sind durch die 3×3-Matrix mit den Elementen $a_{11}, a_{12}, \ldots a_{33}$ repräsentiert. Diese Klasse wird gewöhnlich selbst nicht instantiiert, sondern liefert die allgemeine Funktionalität der linearen Abbildung, insbesondere die Anwendung auf Punktkoordinaten (`applyTo(Pnt2d pnt)`), Inversion (`invert()`) und Verküpfung mit einer zweiten linearen Abbildung (`concat(LinearMapping B)`):

```
1 public class LinearMapping extends Mapping {
2   double
3     a11 = 1, a12 = 0, a13 = 0, // transformation matrix
4     a21 = 0, a22 = 1, a23 = 0,
5     a31 = 0, a32 = 0, a33 = 1;
6
7   LinearMapping() {}
8
9   LinearMapping (   // constructor method
10      double a11, double a12, double a13,
11      double a21, double a22, double a23,
12      double a31, double a32, double a33,
13      boolean inv) {
14    this.a11 = a11; this.a12 = a12; this.a13 = a13;
```

```
15    this.a21 = a21; this.a22 = a22; this.a23 = a23;
16    this.a31 = a31; this.a32 = a32; this.a33 = a33;
17    isInverse = inv;
18  }
19
20  Pnt2d applyTo (Pnt2d pnt) {      // s. Gl. 16.16
21    double h = (a31*pnt.x + a32*pnt.y + a33);
22    double x = (a11*pnt.x + a12*pnt.y + a13) / h;
23    double y = (a21*pnt.x + a22*pnt.y + a23) / h;
24    pnt.x = x;
25    pnt.y = y;
26    return pnt;
27  }
28
29  Mapping invert() {               // s. Gl. 16.21
30    LinearMapping lm = (LinearMapping) duplicate();
31    double det =
32      a11*a22*a33 + a12*a23*a31 + a13*a21*a32 -
33      a11*a23*a32 - a12*a21*a33 - a13*a22*a31;
34    lm.a11 = (a22*a33 - a23*a32) / det;
35    lm.a12 = (a13*a32 - a12*a33) / det;
36    lm.a13 = (a12*a23 - a13*a22) / det;
37    lm.a21 = (a23*a31 - a21*a33) / det;
38    lm.a22 = (a11*a33 - a13*a31) / det;
39    lm.a23 = (a13*a21 - a11*a23) / det;
40    lm.a31 = (a21*a32 - a22*a31) / det;
41    lm.a32 = (a12*a31 - a11*a32) / det;
42    lm.a33 = (a11*a22 - a12*a21) / det;
43    lm.isInverse = !isInverse;
44    return lm;
45  }
46
47  // concatenates this transform matrix A with B: C ← B · A
48  LinearMapping concat(LinearMapping B) {
49    LinearMapping lm = (LinearMapping) duplicate();
50    lm.a11 = B.a11*a11 + B.a12*a21 + B.a13*a31;
51    lm.a12 = B.a11*a12 + B.a12*a22 + B.a13*a32;
52    lm.a13 = B.a11*a13 + B.a12*a23 + B.a13*a33;
53
54    lm.a21 = B.a21*a11 + B.a22*a21 + B.a23*a31;
55    lm.a22 = B.a21*a12 + B.a22*a22 + B.a23*a32;
56    lm.a23 = B.a21*a13 + B.a22*a23 + B.a23*a33;
57
58    lm.a31 = B.a31*a11 + B.a32*a21 + B.a33*a31;
59    lm.a32 = B.a31*a12 + B.a32*a22 + B.a33*a32;
60    lm.a33 = B.a31*a13 + B.a32*a23 + B.a33*a33;
61    return lm;}
62 }
```

`AffineMapping` (Klasse)

`AffineMapping` erweitert die Klasse `LinearMapping` durch zwei zusätzliche Funktionen: erstens durch eine spezielle Konstruktor-Methode, in der die Elemente a_{31}, a_{32}, a_{33} der Abbildungsmatrix – wie für affine Abbildungen erforderlich (s. Gl. 16.12) – auf die Werte $0, 0, 1$ initialisiert werden; zweitens die Methode `makeMapping()`, mit der die affine Abbildung (Vorwärtstransformation T) für beliebige Paare von Dreiecken berechnet wird (Gl. 16.14):

```
1  public class AffineMapping extends LinearMapping {
2
3    AffineMapping (
4        double a11, double a12, double a13,
5        double a21, double a22, double a23,
6        boolean inv) {
7      super(a11,a12,a13,a21,a22,a23,0,0,1,inv);
8    }
9
10   // create the affine transform between
11   // arbitrary triangles (A1..A3) and (B1..B3)
12   static AffineMapping makeMapping (
13       Pnt2d A1, Pnt2d A2, Pnt2d A3,
14       Pnt2d B1, Pnt2d B2, Pnt2d B3) {
15
16     double ax1 = A1.x, ax2 = A2.x,  ax3 = A3.x;
17     double ay1 = A1.y, ay2 = A2.y,  ay3 = A3.y;
18     double bx1 = B1.x, bx2 = B2.x,  bx3 = B3.x;
19     double by1 = B1.y, by2 = B2.y,  by3 = B3.y;
20
21     double S = ax1*(ay3-ay2) + ax2*(ay1-ay3) + ax3*(ay2-ay1);
22     double a11 =
23         (ay1*(bx2-bx3)+ay2*(bx3-bx1)+ay3*(bx1-bx2)) / S;
24     double a12 =
25         (ax1*(bx3-bx2)+ax2*(bx1-bx3)+ax3*(bx2-bx1)) / S;
26     double a21 =
27         (ay1*(by2-by3)+ay2*(by3-by1)+ay3*(by1-by2)) / S;
28     double a22 =
29         (ax1*(by3-by2)+ax2*(by1-by3)+ax3*(by2-by1)) / S;
30     double a13 =
31         (ax1*(ay3*bx2-ay2*bx3) + ax2*(ay1*bx3-ay3*bx1)
32         + ax3*(ay2*bx1-ay1*bx2)) / S;
33     double a23 =
34         (ax1*(ay3*by2-ay2*by3) + ax2*(ay1*by3-ay3*by1)
35         + ax3*(ay2*by1-ay1*by2)) / S;
36
37     return new AffineMapping(a11,a12,a13,a21,a22,a23,false);
38   }
```

`Translation`, `Scaling`, `Shear`, `Rotation` (Klassen)

Die Abbildungen `Translation`, `Scaling`, `Shear` und `Rotation` sind Subklassen von `AffineMapping`. Die Definition dieser Klassen enthält jeweils nur die zugehörige Konstruktor-Methode, die übrige Funktionalität wird aus den Superklassen `AffineTransform` bzw. `LinearTransform` abgeleitet. Der Aufruf `super()` bezieht sich auf die Konstruktor-Methode der direkten Superklasse `AffineMapping`:

```
1 class Translation extends AffineMapping { // s. Gl. 16.4
2   Translation (double dx, double dy) {
3     super(
4       1, 0, dx,
5       0, 1, dy,
6       false );
7   }
8 }
```

```
1 class Scaling extends AffineMapping { // s. Gl. 16.5
2   Scaling(double sx, double sy) {
3     super(
4       sx, 0, 0,
5       0, sy, 0,
6       false );
7   }
8 }
```

```
1 class Shear extends AffineMapping { // s. Gl. 16.6
2   Shear(double bx, double by) {
3     super(
4       1, bx, 0,
5       by, 1, 0,
6       false );
7 }
```

```
1 class Rotation extends AffineMapping { // s. Gl. 16.8
2   Rotation(double alpha) {
3     super(
4        Math.cos(alpha), Math.sin(alpha), 0,
5       -Math.sin(alpha), Math.cos(alpha), 0,
6        false);
7   }
8 }
```

`ProjectiveMapping` (Klasse)

Die Klasse `ProjectiveMapping` implementiert die projektive Abbildung (Gl. 16.16). Sie stellt neben einer Konstruktor-Methode für die Initilisierung der 8 Abbildungsparameter $a_{11}, a_{12}, \ldots a_{32}$ ($a_{33} = 1$) zwei Methoden zur Berechnung der Abbildung zwischen Vierecken zur Verfügung.

Die erste `makeMapping()`-Methode (Zeile 12) erzeugt die projektive Abbildung zwischen dem Einheitsquadrat ($\mathcal{S}_1$) und einem beliebigen Viereck, definiert durch die Koordinatenpunkte `P1..P4`. Die zweite `makeMapping()`-Methode (Zeile 37) erzeugt die projektive Abbildung zwischen zwei beliebigen Vierecken `A1..A4` und `B1..B4` in zwei Schritten über das Einheitsquadrat (s. Gl. 16.28). Diese Methode verwendet dafür u. a. die Methoden `invert()` und `concat()` aus der Klasse `LinearMapping`:

```
1  class ProjectiveMapping extends LinearMapping {
2
3    ProjectiveMapping(
4        double a11, double a12, double a13,
5        double a21, double a22, double a23,
6        double a31, double a32, boolean inv) {
7      super(a11,a12,a13,a21,a22,a23,a31,a32,1,inv);
8    }
9
10   // creates the projective mapping from the unit square S to
11   // the arbitrary quadrilateral Q given by points P1..P4:
12   static ProjectiveMapping makeMapping(
13       Pnt2d P1, Pnt2d P2, Pnt2d P3, Pnt2d P4) {
14     double x1 = P1.x, x2 = P2.x, x3 = P3.x, x4 = P4.x;
15     double y1 = P1.y, y2 = P2.y, y3 = P3.y, y4 = P4.y;
16     double S = (x2-x3)*(y4-y3) - (x4-x3)*(y2-y3);
17
18     double a31 =
19         ((x1-x2+x3-x4)*(y4-y3)-(y1-y2+y3-y4)*(x4-x3))/S;
20     double a32 =
21         ((y1-y2+y3-y4)*(x2-x3)-(x1-x2+x3-x4)*(y2-y3))/S;
22
23     double a11 = x2 - x1 + a31*x2;
24     double a12 = x4 - x1 + a32*x4;
25     double a13 = x1;
26
27     double a21 = y2 - y1 + a31*y2;
28     double a22 = y4 - y1 + a32*y4;
29     double a23 = y1;
30
31     return new
32       ProjectiveMapping(a11,a12,a13,a21,a22,a23,a31,a32,false);
33   }
34
35   // creates the projective mapping between arbitrary
36   // quadrilaterals Q_a, Q_b via the unit square S_1: Q_a → S_1 → Q_b
37   static ProjectiveMapping makeMapping (
38       Pnt2d A1, Pnt2d A2, Pnt2d A3, Pnt2d A4,
39       Pnt2d B1, Pnt2d B2, Pnt2d B3, Pnt2d B4) {
40     ProjectiveMapping T1 = makeMapping(A1, A2, A3, A4);
41     ProjectiveMapping T2 = makeMapping(B1, B2, B3, B4);
42     LinearMapping T1i = (LinearMapping) T1.invert();
```

```
43     LinearMapping T = T1i.concat(T2);
44     T.isInverse = false;
45     return (ProjectiveMapping) T;
46   }
47 }
```

`BilinearMapping` (Klasse)

Diese Klasse implementiert die in Abschn. 16.1.5 beschriebene bilineare Abbildung. Diese nichtlineare Abbildung ist eine direkte Subklasse von `Mapping`, besitzt 8 Parameter (`a1..b4`) und – da sie keine lineare Abbildung ist – eine eigene `applyTo(Pnt2d pnt)`-Methode zur Koordinatentransformation (nach Gl. 16.30):

```
1  import Jampack.*; // use Jampack linear algebra package
2
3  class BilinearMapping extends Mapping {
4    double a1, a2, a3, a4;
5    double b1, b2, b3, b4;
6
7    BilinearMapping( // constructor method
8        double a1, double a2, double a3, double a4,
9        double b1, double b2, double b3, double b4,
10       boolean inv) {
11     this.a1 = a1; this.a2 = a2; this.a3 = a3; this.a4 = a4;
12     this.b1 = b1; this.b2 = b2; this.b3 = b3; this.b4 = b4;
13     isInverse = inv;
14   }
15
16   Pnt2d applyTo (Pnt2d pnt){
17     double x = pnt.x;
18     double y = pnt.y;
19     pnt.x = a1 * x + a2 * y + a3 * x * y + a4;
20     pnt.y = b1 * x + b2 * y + b3 * x * y + b4;
21     return pnt;
22   }
```

Um die Inversion der Abbildung zu umgehen, wird in diesem Fall direkt die Rücktransformation mit der Methode `makeInverseMapping()` erzeugt. Die folgende Methode berechnet die bilineare Abbildung zwischen dem Viereck $\mathcal{P}$, definiert durch die Koordinatenpunkte `P1..P4`, und einem zweiten Viereck $\mathcal{Q}$ (`Q1..Q4`):

```
23   // map between arbitrary quadrilaterals P → Q
24   static BilinearMapping makeInverseMapping (
25       Pnt2d P1, Pnt2d P2, Pnt2d P3, Pnt2d P4, // source quad P
26       Pnt2d Q1, Pnt2d Q2, Pnt2d Q3, Pnt2d Q4 // target quad Q
27       ) {
```

```
28    double[] X = new double[] {Q1.x, Q2.x, Q3.x, Q4.x};
29    double[] Y = new double[] {Q1.y, Q2.y, Q3.y, Q4.y};
30    Zmat zX = makeColumnVector(X);
31    Zmat zY = makeColumnVector(Y);
32
33    double[][] M = new double[][]
34      {{P1.x, P1.y, P1.x * P1.y, 1},
35       {P2.x, P2.y, P2.x * P2.y, 1},
36       {P3.x, P3.y, P3.x * P3.y, 1},
37       {P4.x, P4.y, P4.x * P4.y, 1}};
38    Zmat zM = new Zmat(M);
39
40    double a1, a2, a3, a4; a1 = a2 = a3 = a4 = 0;
41    double b1, b2, b3, b4; b1 = b2 = b3 = b4 = 0;
42
43    try {
44      Zmat Ax = Solve.aib(zM,zX); // solve X = M · a (Gl. 16.31)
45      Zmat By = Solve.aib(zM,zY); // solve Y = M · b (Gl. 16.32)
46      a1 = zA.get(1,1).re; a2 = zA.get(2,1).re;
47      a3 = zA.get(3,1).re; a4 = zA.get(4,1).re;
48      b1 = zB.get(1,1).re; b2 = zB.get(2,1).re;
49      b3 = zB.get(3,1).re; b4 = zB.get(4,1).re;
50    } catch(JampackException e) {}
51
52    return new BilinearMapping(a1,a2,a3,a4,b1,b2,b3,b4,true);
53  }
```

In Zeile 44 und 45 der obigen Methode wird jeweils ein lineares 4×4-Gleichungssystem (s. Gl. 16.31, 16.32) mithilfe der statischen Methode `Solve.aib()` aus der `Jampack`-Bibliothek[9] gelöst, die grundsätzlich mit komplexwertigen Matrizen bzw. Vektoren arbeitet. Zur einfacheren Generierung der notwendigen Spaltenvektoren dient folgende Hilfsmethode:

```
54  // utility method for Jampack linear algebra package
55  static Zmat makeColumnVector(double[] x){
56    int n = x.length;
57    Znum z = new Znum(0,0);
58    Zmat y = new Zmat(n,1);
59    for (int i=0; i<n; i++){
60      z.re = x[i];
61      y.put(i+1,1,z);
62    }
63    return y; } }
```

`TwirlMapping` (Klasse)

Diese Klasse implementiert als Beispiel für eine typische „Warp“-Abbildung die *Twirl*-Transformation (Gl. 16.33, 16.34). Auch für diese

[9] ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html

Abbildung wird mit `makeInverseMapping()` direkt die Rücktransformation erzeugt:

```
1  public class TwirlMapping extends Mapping {
2    double xc;
3    double yc;
4    double angle;
5    double rad;
6
7    TwirlMapping (
8        double xc, double yc, double angle, double rad, boolean
             inv)
9    {
10     this.xc = xc;
11     this.yc = yc;
12     this.angle = angle;
13     this.rad = rad;
14     this.isInverse = inv;
15   }
16
17   static TwirlMapping makeInverseMapping (
18       double xc, double yc, double angle, double rad)
19   {
20     return new TwirlMapping(xc, yc, angle, rad, true);
21   }
22
23   Pnt2d applyTo (Pnt2d pnt) {
24     double x = pnt.x;
25     double y = pnt.y;
26     double dx = x - xc;
27     double dy = y - yc;
28     double d = Math.sqrt(dx*dx + dy*dy);
29     if (d < rad) {
30       double a = Math.atan2(dy,dx) + angle * (rad-d) / rad;
31       pnt.x = xc + d*Math.cos(a);
32       pnt.y = yc + d*Math.sin(a);
33     }
34     return pnt;
35   }
36 }
```

16.4.2 Pixel-Interpolation

Die nachfolgenden Klassen implementieren die drei in Abschn. 16.3 beschriebenen Verfahren zur Interpolation der diskreten Bildfunktion an einer beliebigen Position (x_0, y_0). Die zugehörige Klassenhierarchie ist in Abb. 16.31 dargestellt.

PixelInterpolator (Klasse)

PixelInterpolator ist die (abstrakte) Überklasse für alle übrigen Interpolator-Klassen. Sie spezifiziert insbesondere die Methode getInterpolatedPixel (Pnt2d pnt), die in allen Subklassen implementiert werden muss und die von der Methode applyTo() in der Klasse Mapping (S. 398) aufgerufen wird:

```
1 import ij.process.ImageProcessor;
2
3 public abstract class PixelInterpolator {
4   ImageProcessor ip;
5
6   PixelInterpolator() {}
7
8   void setImageProcessor(ImageProcessor ip) {
9     this.ip = ip;
10  }
11
12  abstract double getInterpolatedPixel(Pnt2d pnt);
13 }
```

NearestNeighborInterpolator (Klasse)

Diese Klasse implementiert die Nearest-Neighbor-Interpolation (Gl. 16.56):

```
1 public class NearestNeighborInterpolator extends
      PixelInterpolator {
2
3   double getInterpolatedPixel(Pnt2d pnt) {
4     int u = (int) Math.rint(pnt.x);
5     int v = (int) Math.rint(pnt.y);
6     return ip.getPixel(u,v);
7   }
8 }
```

BilinearInterpolator (Klasse)

Diese Klasse implementiert die bilineare Interpolation (Gl. 16.58):

```
1 public class BilinearInterpolator extends PixelInterpolator
2 {
3   double getInterpolatedPixel(Pnt2d pnt) {
4     int u = (int) Math.floor(pnt.x);
5     int v = (int) Math.floor(pnt.y);
6     double a = pnt.x - u;
7     double b = pnt.y - v;
8     int A = ip.getPixel(u,v);
9     int B = ip.getPixel(u+1,v);
```

```
10     int C = ip.getPixel(u,v+1);
11     int D = ip.getPixel(u+1,v+1);
12     double E = A + a*(B-A);
13     double F = C + a*(D-C);
14     double G = E + b*(F-E);
15     return G;
16   }
17 }
```

Die bilineare Interpolation ist übrigens auch in der ImageJ-Klasse `ImageProcessor` in Form der Methode

```
double getInterpolatedPixel (double x, double y)
```

bereits verfügbar.

`BicubicInterpolator` (Klasse)

Diese Klasse implementiert die bikubische Interpolation nach Gl. 16.61 bzw. Alg. 16.2. Der Steuerfaktor a hat normalerweise den Wert -1, kann aber mithilfe der zweiten Konstruktor-Methode (Zeile 6) initialisiert werden. Die eindimensionale kubische Interpolation (Gl. 16.48) ist durch die Methode `double cubic (double r)` realisiert:

```
1  public class BicubicInterpolator extends PixelInterpolator {
2    double a = -1;
3
4    BicubicInterpolator() {}
5
6    BicubicInterpolator(double a) {
7      this.a = a;
8    }
9
10   double getInterpolatedPixel(Pnt2d pnt) {
11     double x = pnt.x;
12     double y = pnt.y;
13     // use floor to handle negative coordinates:
14     int x0 = (int) Math.floor(x);
15     int y0 = (int) Math.floor(y);
16
17     double q = 0;
18     for (int j = 0; j < 4; j++) {
19       int v = y0 - 1 + j;
20       double p = 0;
21       for (int i = 0; i < 4; i++) {
22         int u = x0 - 1 + i;
23         p = p + ip.getPixel(u,v) * cubic(x - u);
24       }
25       q = q + p * cubic(y - v);
26     }
27     return q;
```

```
28   }
29
30   double cubic(double r) {
31     if (r < 0) r = -r;
32     double w = 0;
33     if (r < 1)
34       w = (a+2)*r*r*r - (a+3)*r*r + 1;
35     else if (r < 2)
36       w = a*r*r*r - 5*a*r*r + 8*a*r - 4*a;
37     return w;
38   }
39 }
```

16.4.3 Anwendungsbeispiele

Die folgenden zwei ImageJ-Plugins zeigen einfache Beispiele für die Anwendung der oben beschriebenen Klassen für geometrische Abbildungen und Interpolation.

Rotation

Das erste Beispiel zeigt ein Plugin (`PluginRotation_`) für die Rotation des Bilds um 15°. Zunächst wird (in Zeile 14) das Transformationsobjekt (`map`) der Klasse `Rotation` erzeugt, die angegebenen Winkelgrade werden dafür in Radianten umgerechnet. Anschließend wird (in Zeile 15) das Interpolator-Objekt `ipol` angelegt. Die eigentliche Transformation des Bilds erfolgt in Zeile 16 durch Aufruf der Methode `applyTo()`:

```
1  import ij.ImagePlus;
2  import ij.plugin.filter.PlugInFilter;
3  import ij.process.*;
4
5  public class PluginRotation_ implements PlugInFilter {
6
7    double angle = 15; // rotation angle (in degrees)
8
9      public int setup(String arg, ImagePlus imp) {
10         return DOES_8G;
11     }
12
13     public void run(ImageProcessor ip) {
14     Rotation map = new Rotation((2 * Math.PI * angle) / 360);
15     PixelInterpolator ipol = new BicubicInterpolator();
16     map.applyTo(ip, ipol);
17     }
18 }
```

Projektive Transformation

Im zweiten Beispiel ist die Anwendung der projektiven Abbildung gezeigt. Die Abbildung T ist durch zwei Vierecke in Form der Punkte `p1..p4` bzw. `q1..q4` definiert. Diese Punkte würde man in einer konkreten Anwendung interaktiv bestimmen oder wären durch eine Mesh-Partitionierung vorgegeben.

Die Vorwärtstransformation T und das zugehörige Objekt `map` wird durch die Methode `ProjectiveMapping.makeMapping()` erzeugt (Zeile 22). In diesem Fall wird ein bilinearer Interpolator verwendet (Zeile 24), die Anwendung der Transformation erfolgt wie im vorherigen Beispiel:

```
1  import ij.ImagePlus;
2  import ij.plugin.filter.PlugInFilter;
3  import ij.process.*;
4
5  public class PluginProjectiveMapping_ implements PlugInFilter
       {
6
7      public int setup(String arg, ImagePlus imp) {
8          return DOES_8G;
9      }
10
11     public void run(ImageProcessor ip) {
12     Pnt2d p1 = new Pnt2d(0,0);
13     Pnt2d p2 = new Pnt2d(400,0);
14     Pnt2d p3 = new Pnt2d(400,400);
15     Pnt2d p4 = new Pnt2d(0,400);
16
17     Pnt2d q1 = new Pnt2d(0,60);
18     Pnt2d q2 = new Pnt2d(400,20);
19     Pnt2d q3 = new Pnt2d(300,400);
20     Pnt2d q4 = new Pnt2d(30,200);
21
22     ProjectiveMapping map =
23       ProjectiveMapping.makeMapping(p1,p2,p3,p4,q1,q2,q3,q4);
24     PixelInterpolator ipol = new BilinearInterpolator();
25     map.applyTo(ip, ipol);
26     }
27 }
```

Die korrespondierenden Punkte der Vierecke `P1..P4` und `Q1..Q4` würde man im praktischen Einsatz natürlich interaktiv spezifizieren.

16.5 Aufgaben

Aufg. 16.1. Zeigen Sie, dass eine Gerade $y = kx + d$ in 2D durch eine projektive Abbildung (Gl. 16.16) wiederum in eine Gerade abgebildet wird.

Aufg. 16.2. Zeigen Sie, dass die Parallelität von Geraden durch eine affine Abbildung (Gl. 16.12) erhalten bleibt.

Aufg. 16.3. Implementieren Sie die geometrischen Abbildungen `RippleMapping` (Gl. 16.35) und `SphereMapping` (sphärische Verzerrung, Gl. 16.36) als Transformationsklassen. Verwenden Sie als Muster die Java-Klasse `TwirlMapping` (S. 404). Erzeugen Sie auch entsprechende ImageJ-Plugins, um diese Klassen zu verwenden.

Aufg. 16.4. Konzipieren Sie eine geometrische Abbildung ähnlich der Ripple-Transformation (Gl. 16.35), die anstatt der Sinusfunktion eine sägezahnförmige Funktion für die Verzerrung in horizontaler und vertikaler Richtung benutzt. Verwenden Sie zur Implementierung die Java-Klasse `TwirlMapping` (S. 404) als Muster.

Aufg. 16.5. Implementieren Sie die zweidimensionale Lanczos-Interpolation mit W_{L3}-Kern nach Gl. 16.63 als Java-Klasse analog zur Klasse `BicubicInterpolator` (S. 407). Vergleichen Sie die Ergebnisse mit denen der bikubischen Interpolation.

Aufg. 16.6. Der eindimensionale Lanczos-Interpolationskern mit 4 Stützstellen ist analog zu Gl. 16.54 definiert als

$$w_{\mathrm{L4}} = \begin{cases} 4 \cdot \frac{\sin(\pi \frac{x}{4}) \cdot \sin(\pi x)}{\pi^2 x^2} & \text{für } 0 \le |x| < 4 \\ 0 & \text{für } |x| \ge 4 \end{cases} \tag{16.64}$$

Erweitern Sie den zweidimensionalen Interpolationskern aus Gl. 16.63 für w_{L4} und implementieren Sie das Verfahren (analog zur Klasse `BicubicInterpolator` auf S. 407). Wie viele Originalpixel müssen jeweils bei der Berechnung berücksichtigt werden? Testen Sie, ob sich gegenüber der bikubischen bzw. der Lanczos3-Interpolation (Aufg. 16.5) erkennbare Verbesserungen ergeben.

17

Bildvergleich

Wenn wir Bilder miteinander vergleichen, stellt sich die grundlegende Frage: Wann sind zwei Bilder gleich oder wie kann man deren Ähnlichkeit messen? Natürlich könnte man einfach definieren, dass zwei Bilder I_1 und I_2 genau dann gleich sind, wenn alle ihre Bildwerte identisch sind bzw. wenn – zumindest für Intensitätsbilder – die Differenz $I_1 - I_2$ null ist. Die direkte Subtraktion von Bildern kann tatsächlich nützlich sein, z. B. zur Detektion von Veränderungen in aufeinander folgenden Bildern unter konstanten Beleuchtungs- und Aufnahmebedingungen. Darüber hinaus ist aber die numerische Differenz allein kein sehr zuverlässiges Mittel zur Bestimmung der Ähnlichkeit von Bildern. Eine leichte Erhöhung der Gesamthelligkeit, die Quantisierung der Intensitätswerte, eine Verschiebung des Bilds um nur ein Pixel oder eine geringfügige Rotation – all das würde zwar die Erscheinung des Bilds kaum verändern und möglicherweise für den menschlichen Betrachter überhaupt nicht feststellbar sein, aber dennoch große numerische Unterschiede gegenüber dem Ausgangsbild verursachen! Als Menschen empfinden wir oft Bilder aufgrund ihrer Struktur oder ihres Inhalts als ähnlich, obwohl im direkten Vergleich zwischen einzelnen Pixeln kaum eine Übereinstimmung besteht. Das Vergleichen von Bildern ist daher i. Allg. kein einfaches Problem und ist auch, etwa im Zusammenhang mit der ähnlichkeitsbasierten Suche in Bilddatenbanken oder im Internet, ein interessantes Forschungsthema.

Dieses Kapitel widmet sich einem Teilproblem des Bildvergleichs, nämlich der Lokalisierung eines bekannten Teilbilds – das oft als „template“ bezeichnet wird – innerhalb eines größeren Bilds. Dieses Problem stellt sich häufig, z. B. beim Auffinden zusammengehöriger Bildpunkte in Stereobildern, bei der Lokalisierung eines bestimmtes Objekts in einer Szene oder bei der Verfolgung eines Objekts in einer Bildsequenz. Die grundlegende Idee des *Template Matching* ist einfach: Wir bewegen das gesuchte Bildmuster (Template) über das Bild und messen die Differenz

gegenüber dem darunterliegenden Teilbild und markieren jene Stellen, an denen das Template mit dem Teilbild übereinstimmt oder ihm zumindest ausreichend ähnlich ist. Natürlich ist auch das nicht so einfach, wie es zunächst klingt, denn was ist ein brauchbares Abstandsmaß, welche Distanz ist zulässig und was passiert, wenn Bilder zueinander gedreht, skaliert oder verformt werden?

Wir hatten mit dem Thema Ähnlichkeit bereits im Zusammenhang mit den Eigenschaften von Regionen in segmentierten Binärbildern (Abschn. 11.4.2) zu tun. Im Folgenden beschreiben wir unterschiedliche Ansätze für das *Template Matching* in Intensitätsbildern und unsegmentierten Binärbildern.

17.1 Template Matching in Intensitätsbildern

Zunächst betrachten wir das Problem, ein gegebenes Referenzbild $R(i,j)$ in einem Intensitäts- oder Grauwertbild $I(u,v)$ zu lokalisieren. Die Aufgabe ist, jene Stelle(n) in $I(u,v)$ zu finden, an denen eine optimale Übereinstimmung der entsprechenden Bildinhalte besteht. Wenn wir $R_{r,s}(u,v) = R(u-r, v-s)$ als das um (r,s) in horizontaler bzw. vertikaler Richtung verschobene Referenzbild bezeichnen, dann können wir das Template-Matching-Problem in folgender Weise zusammenfassen (s. Abb. 17.1):

> Gegeben sind ein Zielbild I und ein Referenzbild R. Finde den Offset (r,s), bei dem die Ähnlichkeit zwischen dem um (r,s) verschobenen Referenzbild $R_{r,s}$ und dem davon überdeckten Ausschnitt des Zielbilds maximal ist.

Zu klären sind dabei drei Punkte: *Erstens* benötigen wir ein geeignetes Maß für die „Ähnlichkeit“ zwischen zwei Teilbildern, *zweitens* eine

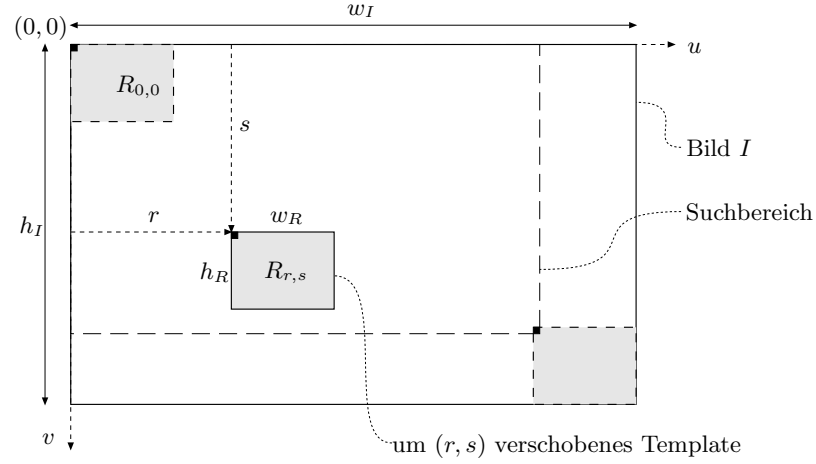


Abbildung 17.1
Geometrie des Template Matching. Das Referenzbild R wird mit dem Offset (r,s) über dem Zielbild I verschoben, wobei der Koordinatenursprung als Referenzpunkt dient. Die Größe des Zielbilds $(w_I \times h_I)$ und des Referenzbilds $(w_R \times h_R)$ bestimmen den maximalen Suchbereich für den Vergleich.

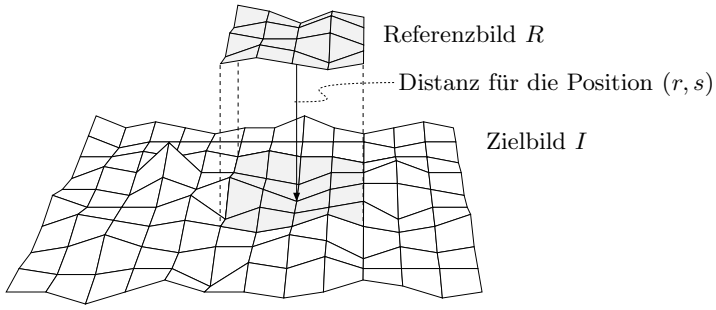


Abbildung 17.2
Messung des Abstands zwischen zweidimensionalen Bildfunktionen. Das Referenzbild R ist an der Position (r, s) über dem Zielbild I positioniert.

Suchstrategie, um die optimale Verschiebung möglichst rasch zu finden, und *drittens* müssen wir entscheiden, welche minimale Ähnlichkeit für eine Übereinstimmung zulässig ist. Zunächst interessiert uns aber nur der erste Punkt.

17.1.1 Abstand zwischen Bildmustern

Um herauszufinden, wo eine hohe Übereinstimmung besteht, berechnen wir den *Abstand* zwischen dem verschobenen Referenzbild R und dem entsprechenden Ausschnitt des Zielbilds I für jede Position r, s (Abb. 17.2). Als Maß für den Abstand zwischen zweidimensionalen Bildfunktionen gibt es verschiedene gebräuchliche Definitionen, wobei folgende am gängigsten sind:

Summe der Differenzbeträge:

$$\mathrm{d}_A(r,s) = \sum_{(i,j)\in R} |I(r+i, s+j) - R(i,j)| \tag{17.1}$$

Maximaler Differenzbetrag:

$$\mathrm{d}_M(r,s) = \max_{(i,j)\in R} |I(r+i, s+j) - R(i,j)| \tag{17.2}$$

Summe der quadratischen Abstände (N-dimensionaler euklidischer Abstand):

$$\mathrm{d}_E(r,s) = \Big[\sum_{(i,j)\in R} \big(I(r+i, s+j) - R(i,j)\big)^2 \Big]^{1/2} \tag{17.3}$$

Abstand und Korrelation

Der N-dimensionale euklidische Abstand (Gl. 17.3) ist von besonderer Bedeutung und wird wegen seiner formalen Qualitäten auch in der Statistik häufig verwendet. Um die beste Übereinstimmung zwischen dem Referenzbild $R(u,v)$ und dem Zielbild $I(u,v)$ zu finden, genügt es, das Quadrat von d_E (das in jedem Fall positiv ist) zu minimieren, welches in der Form

$$\begin{aligned} \mathrm{d}_E^2(r,s) &= \sum_{(i,j)\in R} \big(I(r+i,s+j) - R(i,j)\big)^2 \qquad (17.4)\\ &= \underbrace{\sum_{(i,j)\in R} \big(I(r+i,s+j)\big)^2}_{A(r,s)} + \underbrace{\sum_{(i,j)\in R} \big(R(i,j)\big)^2}_{B} - \underbrace{2\sum_{(i,j)\in R} I(r+i,s+j)\cdot R(i,j)}_{C(r,s)} \end{aligned}$$

expandiert werden kann. Der Ausdruck B in Gl. 17.4 ist dabei die quadratische Summe aller Werte im Referenzbild R, also eine von r,s unabhängige Konstante, die ignoriert werden kann. Der Ausdruck $A(r,s)$ ist die quadratische Summe der Werte des entsprechenden Bildausschnitts in I beim aktuellen Offset (r,s), und $C(r,s)$ entspricht der so genannten *linearen Kreuzkorrelation* zwischen I und R. Diese ist für den allgemeinen Fall definiert als

$$(I \circledast R)(r,s) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(r+i,s+j)\cdot R(i,j), \qquad (17.5)$$

was – da R und I außerhalb ihrer Grenzen als null angenommen werden – wiederum äquivalent ist zu

$$\sum_{i=0}^{w_R-1} \sum_{j=0}^{h_R-1} I(r+i,s+j)\cdot R(i,j) = \sum_{(i,j)\in R} I(r+i,s+j)\cdot R(i,j) \;=\; C(r,s).$$

Die Korrelation ist damit im Grunde dieselbe Operation wie die lineare *Faltung* (Abschn. 6.3.1, Gl. 6.14), außer dass bei der Korrelation der Faltungskern (in diesem Fall $R(i,j)$) implizit gespiegelt ist.

Wenn nun $A(r,s)$ in Gl. 17.4 innerhalb des Bilds I weitgehend konstant ist – also die „Signalenergie“ annähernd gleichförmig im Bild verteilt ist –, dann befindet sich an der Position des Maximalwerts der Korrelation $C(r,s)$ gleichzeitig auch die Stelle der höchsten Übereinstimmung zwischen R und I. In diesem Fall kann also der Minimalwert von $d_E^2(r,s)$ (Gl. 17.4) allein durch Berechnung des Maximalwerts der Korrelation $I \circledast R$ ermittelt werden. Das ist u. a. deshalb interessant, weil die Korrelation über die *Fouriertransformation* im Spektralraum sehr effizient berechnet werden kann (s. Abschn. 14.5).

Normalisierte Kreuzkorrelation

In der Praxis trifft leider die Annahme, dass $A(r,s)$ über das Bild hinweg konstant ist, meist nicht zu und das Ergebnis der Korrelation ist in der Folge stark von Intensitätsänderungen im Bild I abhängig. Die normalisierte Kreuzkorrelation $C_N(r,s)$ kompensiert diese Abhängigkeit, indem sie die Gesamtenergie im aktuellen Bildausschnitt berücksichtigt:

$$C_N(r,s) = \frac{C(r,s)}{\sqrt{A(r,s)\cdot B}} = \frac{C(r,s)}{\sqrt{A(r,s)}\cdot\sqrt{B}} \tag{17.6}$$

$$= \frac{\sum_{(i,j)\in R} I(r+i,s+j)\cdot R(i,j)}{\Big[\sum_{(i,j)\in R}\big(I(r+i,s+j)\big)^2\Big]^{1/2}\cdot\Big[\sum_{(i,j)\in R}\big(R(i,j)\big)^2\Big]^{1/2}}$$

Weisen Bild und Template ausschließlich positive Werte auf, dann ist das Ergebnis $C_N(r,s)$ immer im Bereich $0\ldots1$, unabhängig von den übrigen Werten in I und R. Ein Wert $C_N(r,s)=1$ zeigt dabei eine maximale Übereinstimmung zwischen R und dem aktuellen Bildausschnitt I bei einem Offset (r,s) an. Die normalisierte Korrelation hat daher den zusätzlichen Vorteil, dass sie auch ein standardisiertes Maß für den Grad der Übereinstimmung liefert, der direkt für die Entscheidung über die Akzeptanz der entsprechenden Position verwendet werden kann.

Die Formulierung in Gl. 17.6 gibt – im Unterschied zu Gl. 17.5 – zwar ein *lokales* Abstandsmaß an, ist aber immer noch mit dem Problem behaftet, dass die *absolute* Distanz zwischen dem Template und der Bildfunktion gemessen wird. Wird also beispielsweise die Gesamthelligkeit des Bilds I erhöht, so wird sich in der Regel auch das Ergebnis der normalisierten Korrelation $C_N(r,s)$ dramatisch verändern.

Korrelationskoeffizient

Eine Möglichkeit zur Vermeidung dieses Problems besteht darin, nicht die ursprünglichen Funktionswerte zu vergleichen, sondern die Differenz in Bezug auf die lokalen Durchschnittswerte in R einerseits und des zugehörigen Bildausschnitts von I andererseits. Gl. 17.6 ändert sich damit zu

$$C_L(r,s) = \tag{17.7}$$
$$\frac{\sum_{(i,j)\in R}\big(I(r+i,s+j)-\bar{I}(r,s)\big)\cdot\big(R(i,j)-\bar{R}\big)}{\Big[\sum_{(i,j)\in R}\big(I(r+i,s+j)-\bar{I}(r,s)\big)^2\Big]^{1/2}\cdot\underbrace{\Big[\sum_{(i,j)\in R}\big(R(i,j)-\bar{R}\big)^2\Big]^{1/2}}_{\sigma_R^2}},$$

wobei die Durchschnittswerte $\bar{I}(r,s)$ und $\bar{R}$ definiert sind als

$$\bar{I}(r,s) = \frac{1}{K} \sum_{(i,j)\in R} I(r+i, s+j)\,, \quad \bar{R} = \frac{1}{K} \sum_{(i,j)\in R} R(i,j) \tag{17.8}$$

und $K = |R|$, also die Anzahl der Elemente im Template R. Der Ausdruck in Gl. 17.7 wird in der Statistik als *Korrelationskoeffizient* bezeichnet. Im Unterschied zur üblichen Verwendung in der Statistik ist $C_L(r,s)$ aber keine *globale* Korrelation über sämtliche Daten, sondern eine *lokale*, stückweise Korrelation zwischen dem Template R und dem von diesem aktuell (d. h. bei einem Offset (r,s)) überdeckten Teilbild von I! Die Ergebniswerte von $C_L(r,s)$ liegen im Intervall $-1 \ldots 1$, wobei der Wert 1 wiederum der höchsten Übereinstimmung der verglichenen Bildmuster und -1 der maximalen Abweichung entspricht.

Der im Nenner von Gl. 17.7 enthaltene Ausdruck

$$\sigma_R^2 = \sum_{(i,j)\in R} \Big(R(i,j) - \bar{R}\Big)^2 = \sum_{(i,j)\in R} \Big(R(i,j)\Big)^2 \;-\; K \cdot \bar{R}^2\,, \tag{17.9}$$

bezeichnet die *Varianz* der Werte im Template R, die konstant ist und daher nur einmal ermittelt werden muss. Durch entsprechende Ersetzung in Gl. 17.7 ergibt sich in der Form

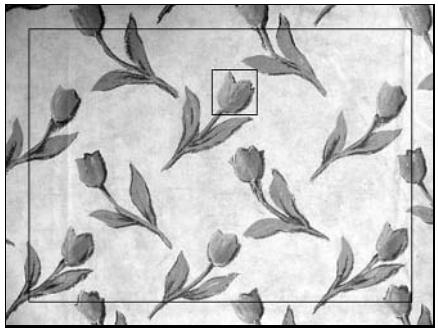
$$C_L(r,s) = \frac{\displaystyle\sum_{(i,j)\in R} \Big(I(r+i, s+j) \cdot R(i,j)\Big) \;-\; K \cdot \bar{I}(r,s) \cdot \bar{R}}{\displaystyle\left[\sum_{(i,j)\in R} \Big(I(r+i, s+j)\Big)^2 \;-\; K \cdot \big(\bar{I}(r,s)\big)^2\right]^{1/2} \cdot \sigma_R} \tag{17.10}$$

eine effiziente Möglichkeit zur Berechnung des lokalen Korrelationskoeffizienten. Da $\bar{R}$ und $\sigma_R = \sqrt{\sigma_R^2}$ nur einmal berechnet werden müssen und der lokale Durchschnittswert der Bildfunktion $\bar{I}(r,s)$ bei der Berechnung der Differenzen zunächst nicht benötigt wird, kann der gesamte Ausdruck in Gl. 17.10 in einer einzigen, gemeinsamen Iteration berechnet werden (s. Methode `getMatchValue()` in Prog. 17.2).

Der Korrelationskoeffizient in Gl. 17.10 besitzt als lokales Maß – im Gegensatz zur globalen linearen Korrelation (Gl. 17.5) – keine ähnlich einfache und effiziente Möglichkeit zur Realisierung im Spektralraum. Die Berechnung erfolgt daher im Ortsraum, wie die Java-Implementierung in Abschn. 17.1.3 zeigt.

Beispiele

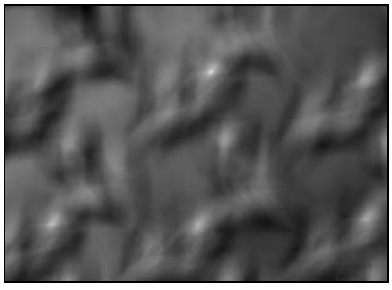
Abb. 17.3 zeigt einen Vergleich zwischen den oben angeführten Distanzfunktionen anhand eines typischen Beispiels. Das Originalbild (Abb. 17.3 (a)) weist ein sich wiederholendes Muster auf, gleichzeitig aber auch eine ungleichmäßige Beleuchtung und dadurch deutliche Unterschiede in der Helligkeit. Ein charakteristisches Detail wurde als Template dem Bild entnommen (Abb. 17.3 (b)).



(a) Originalbild I



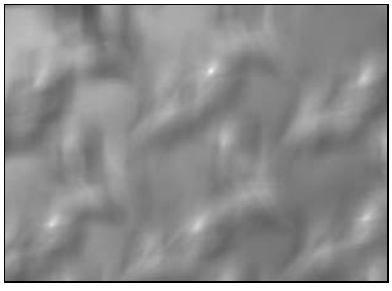
(b) Referenzbild R



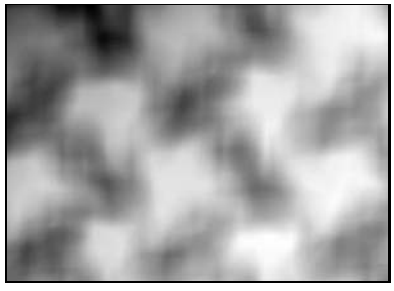
(c) Summe der Differenzbeträge



(d) Maximaler Differenzbetrag



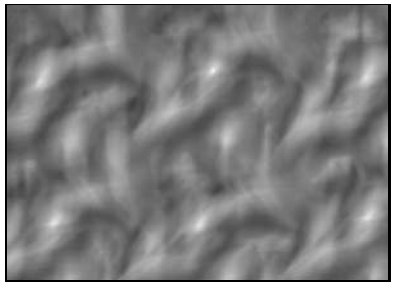
(e) Summe der quadr. Abstände



(f) Globale Kreuzkorrelation



(g) Normalisierte Kreuzkorrelation



(h) Korrelationskoeffizient

Abbildung 17.3
Vergleich unterschiedlicher Abstandsfunktionen. Aus dem Originalbild I (a) wurde an der markierten Stelle das Referenzbild R (b) entnommen. Die Helligkeit der Ergebnisbilder (c–h) entspricht dem berechneten Maß der Übereinstimmung.

- Die *Summe der Differenzbeträge* (Gl. 17.1) in Abb. 17.3 (c) liefert einen deutlichen Spitzenwert an der Originalposition, ähnlich wie auch die *Summe der quadratischen Abstände* (Gl. 17.3) in Abb. 17.3 (e). Beide Maße funktionieren zwar zufriedenstellend, werden aber von globalen Intensitätsänderungen stark beeinträchtigt, wie Abb. 17.4 und 17.5 deutlich zeigen.
- Der *maximale Differenzbetrag* (Gl. 17.2) in Abb. 17.3 (d) erweist sich als Distanzmaß als völlig nutzlos, zumal er stärker auf Beleuchtungsunterschiede als auf die Ähnlichkeit zwischen Bildmustern reagiert. Wie erwartet ist auch das Verhalten der *globalen Kreuzkorrelation* in Abb. 17.3 (f) nicht zufriedenstellend. Obwohl das Ergebnis an der ursprünglichen Template-Position ein (im Druck kaum sichtbares) lokales Maximum aufweist, wird dieses durch die großflächigen hohen Werte in den hellen Bildteilen völlig überdeckt.
- Das Ergebnis der *normalisierten Kreuzkorrelation* (Gl. 17.6) in Abb. 17.3 (g) ist naturgemäß sehr ähnlich zur Summe der quadratischen Abstände, denn es ist im Grunde dasselbe Maß. Der *Korrelationskoeffizient* (Gl. 17.7) in Abb. 17.3 (h) liefert wie erwartet das beste Ergebnis. In diesem Fall liegen die Werte im Bereich -1.0 (schwarz) und $+1.0$ (weiß), Nullwerte sind grau dargestellt.

Abb. 17.4 vergleicht das Verhalten der *Summe der quadratischen Abstände* einerseits und des *Korrelationskoeffizienten* andererseits bei Änderung der globalen Intensität. Dazu wurde die Intensität des Templates nachträglich um 50 erhöht, sodass im Bild selbst keine Stelle mehr mit einem zum Template identischen Bildmuster existiert. Wie deutlich zu erkennen ist, verschwinden bei der *Summe der quadratischen Abstände* die ursprünglich ausgeprägten Spitzenwerte (Abb. 17.4 (c)), während der *Korrelationskoeffizient* davon naturgemäß nicht beeinflusst wird (Abb. 17.4 (d)).

Zusammengefasst ist unter realistischen Abbildungsverhältnissen der lokale Korrelationskoeffizient als zuverlässiges Maß für den intensitätsbasierten Bildvergleich zu empfehlen. Diese Methode ist vergleichsweise robust gegenüber globalen Intensitäts- und Kontraständerungen sowie gegenüber geringfügigen Veränderungen der zu vergleichenden Bildmuster. Wie in Abb. 17.6 gezeigt, ist dabei die Lokalisierung der optimalen Match-Punkte oft durch eine einfache Schwellwertoperation möglich.

Geometrische Form der Template-Region

Die Template-Region R muss nicht, wie in den bisherigen Beispielen, von rechteckiger Form sein. Konkret werden häufig kreisförmige, elliptische oder auch nicht konvexe Templates verwendet, wie z. B. kreuz- oder $\times$-förmige Regionen.

Eine weitere Möglichkeit ist die individuelle Gewichtung der Elemente innerhalb des Templates, etwa um die Differenzen im Zentrum des Templates gegenüber den Rändern stärker zu betonen. Die Realisierung

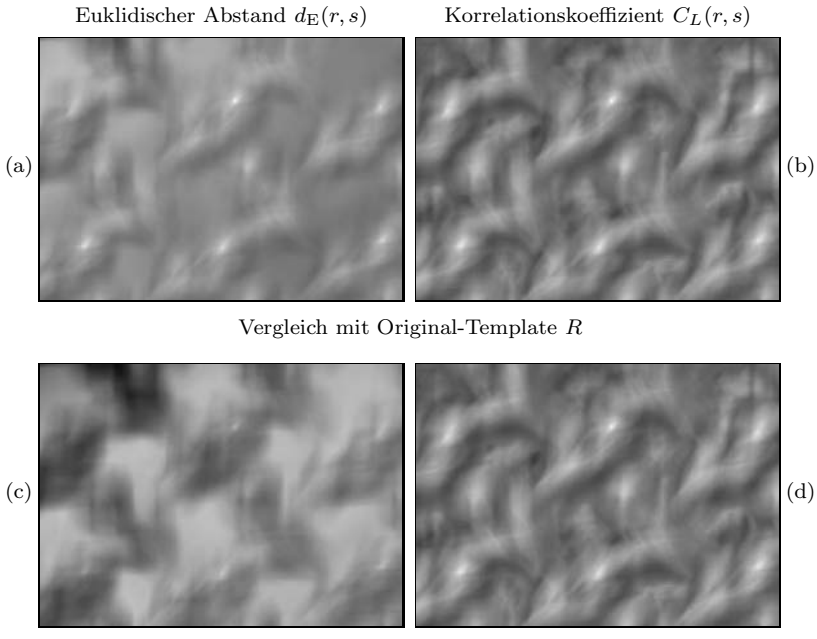


Abbildung 17.4
Auswirkung einer globalen Intensitätsänderung. Beim Vergleich mit dem Original-Template R zeigen sich sowohl beim *Euklidischen Abstand* (a) wie auch beim *Korrelationskoeffizienten* (b) deutliche Spitzenwerte an den Stellen höchster Übereinstimmung. Beim modifizierten Template R' verschwinden die Spitzenwerte beim *Euklidischen Abstand* (c), während der *Korrelationskoeffizient* (d) davon unbeeinflusst bleibt.

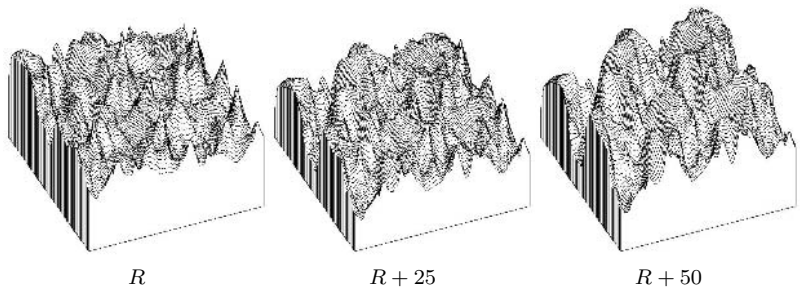


Abbildung 17.5
Euklidischer Abstand – Auswirkung globaler Intensitätsänderung. Matching mit dem Original-Template R (links) und Templates mit einer um 25 Einheiten (Mitte) bzw. 50 Einheiten (rechts) erhöhten Intensität. Man beachte, wie bei steigendem Gesamtabstand zwischen Template und Bildfunktion die lokalen Spitzenwerte verschwinden.

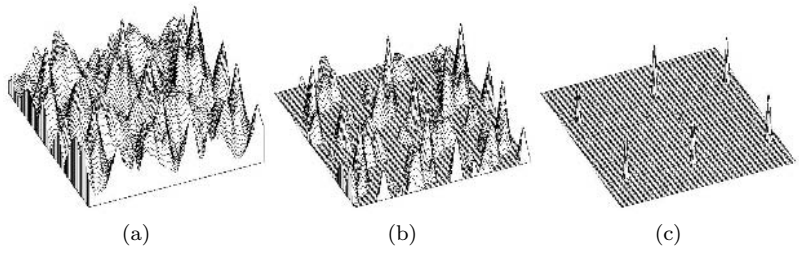


Abbildung 17.6
Detektion der Match-Punkte durch einfache Schwellwertoperation. Lokaler Korrelationskoeffizient (a), nur positive Ergebniswerte (b), Werte größer als 0.5 (c). Die resultierenden Spitzen markieren die Positionen der 6 ähnlichen (aber nicht identischen) Tulpenmuster im Originalbild (Abb. 17.3 (a)).

dieses „windowed matching“ ist einfach und erfordert nur geringfügige Modifikationen.

17.1.2 Umgang mit Drehungen und Größenänderungen

Korrelationsbasierte Matching-Methoden sind im Allgemeinen nicht imstande, substantielle Verdrehungen oder Größenänderungen zwischen Bild und Template zu bewältigen. Eine Möglichkeit zur Berücksichtigung der Rotation ist, das Bild mit mehreren, unterschiedlich gedrehten Versionen des Templates zu vergleichen und dabei die optimale Übereinstimmung zu suchen. In ähnlicher Weise könnte man auch unterschiedlich skalierte Versionen eines Templates zum Vergleich heranziehen, zumindest innerhalb eines bestimmten Größenbereichs. Die Suche nach gedrehten *und* skalierten Bildmustern würde allerdings eine kombinatorische Vielfalt unterschiedlicher Templates und zugehöriger Matching-Durchläufe erfordern, was meistens nicht praktikabel ist.

17.1.3 Implementierung

Prog. 17.1–17.2 zeigt eine Java-Implementierung des Template Matching auf Basis des lokalen Korrelationskoeffizienten (Gl. 17.7). Für die Anwendung wird vorausgesetzt, dass Bild (`imageFp`) und Template (`templateFp`) bereits als Objekte vom Typ `FloatProcessor` vorliegen. Damit wird ein neues Objekt der Klasse `CorrCoeffMatcher` angelegt, wie in folgendem Beispiel:

```
1 FloatProcessor imageFp = ...     // image
2 FloatProcessor templateFp = ...  // template
3 CorrCoeffMatcher matcher =
4                 new CorrCoeffMatcher(imageFp, templateFp);
5 FloatProcessor matchFp = matcher.computeMatch();
```

Das Match-Ergebnis wird durch die anschließende Anwendung der Methode `computeMatch()` in Form eines neuen Bilds (`matchFp`) vom Typ `FloatProcessor` ermittelt. Durch direkten Zugriff auf die Pixel-Arrays (anstelle der hier verwendeten Zugriffsmethoden `getPixelValue()` und `putPixelValue()`, s. auch Abschn. 3.6) ist eine deutliche Steigerung der Effizienz möglich.

17.2 Vergleich von Binärbildern

Wie im vorigen Abschnitt deutlich wurde, ist das Vergleichen von Intensitätsbildern auf Basis der Korrelation zwar keine optimale Lösung, aber unter eingeschränkten Bedingungen ausreichend zuverlässig und effizient. Im Prinzip könnte man diese Technik auch für Binärbilder anwenden. Wenn wir jedoch zwei übereinander liegende Binärbilder direkt

```
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
7    float meanR;      // mean value of template (R̄)
8    float sigmaR;     // square root of variance of template (σ_R)
9
10   CorrCoeffMatcher (FloatProcessor img,FloatProcessor tmpl) {
11     I = img;
12     R = tmpl;
13     wI = I.getWidth();
14     hI = I.getHeight();
15     wR = R.getWidth();
16     hR = R.getHeight();
17     kR = wR * hR;
18
19     // compute mean and variance of template
20     float sumR = 0;                 // ∑ R(i,j)
21     float sumR2 = 0;                // ∑(R(i,j))^2
22     for (int j = 0; j < hR; j++) {
23       for (int i = 0; i < wR; i++) {
24         float valR = R.getPixelValue(i,j);
25         sumR += valR;
26         sumR2 += valR * valR;
27       }
28     }
29     meanR = sumR / kR;            // R̄ = (∑ R(i,j))/K
30     sigmaR =                      // σ_R = (∑(R(i,j))^2 − K · R̄^2))^1/2
31         (float) Math.sqrt(sumR2 - kR * meanR * meanR);
32   }
```

Programm 17.1
Klasse `CorrCoeffMatcher`. Klassendefinition und zugehörige Konstruktor-Methode. Bei der Initialisierung durch die Konstruktor-Methode (Zeilen 10–32) werden vorab der Durchschnittswert `meanR` ($\bar{R}$ in Gl. 17.8) und die Wurzel der Varianz `sigmaR` ($\sigma_T = \sqrt{\sigma_T^2}$ in Gl. 17.9) des Templates berechnet.

vergleichen, dann wird die Gesamtdifferenz zwischen beiden nur dann gering, wenn Pixel für Pixel weitgehend eine exakte Übereinstimmung besteht. Da es keine kontinuierlichen Übergänge zwischen den Intensitätswerten gibt, zeigt die Abstandsfunktion – abhängig von der relativen Verschiebung der Muster – im Allgemeinen ein unangenehmes Verhalten und weist insbesondere viele lokale Spitzenwerte auf (Abb. 17.7).

17.2.1 Direkter Vergleich von Binärbildern

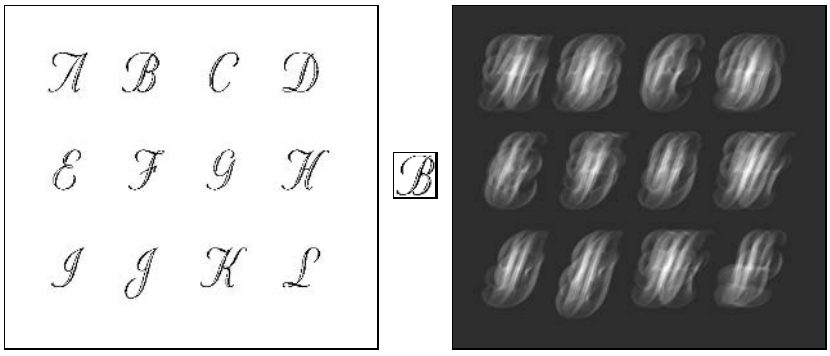
Das Problem beim direkten Vergleich zwischen Binärbildern ist, dass selbst kleinste Abweichungen zwischen den Bildmustern – etwa aufgrund einer geringfügigen Verschiebung, Drehung oder Verzerrung – zu starken Änderungen des Abstands führen können. So kann etwa im Fall einer aus dünnen Linien bestehenden Strichgrafik bereits eine Verschiebung um *ein* Pixel genügen, um von maximaler Übereinstimmung zu völliger

Programm 17.2
Klasse `CorrCoeffMatcher` (Fortsetzung). Die Methode `computeMatch()` (Zeilen 33–47) berechnet den Match-Wert zwischen dem Bild und dem Template für alle Positionen und erzeugt daraus ein neues Gleitkommabild `matchFp`. Der lokale Match-Wert ($C_L(r,s)$, s. Gl. 17.10) an der Position (r,s) wird durch die Methode `getMatchValue(r,s)` (Zeilen 50–67) berechnet.

```
33  FloatProcessor computeMatch () {
34    FloatProcessor matchFp = new FloatProcessor(wI,hI);
35    int umax = wI-wR;
36    int vmax = hI-hR;
37    int rc = wR/2;          // center coordinates of template
38    int sc = hR/2;
39
40    for (int r=0; r<=umax; r++){
41      for (int s=0; s<=vmax; s++){
42        float q = getMatchValue(r,s);
43        matchFp.putPixelValue(r+rc,s+sc,q);
44      }
45    }
46    return matchFp;
47  }
48
49
50  float getMatchValue (int r, int s) {
51    float sumI = 0;                  // ∑ I(r+i,s+j)
52    float sumI2 = 0;                 // ∑ (I(r+i,s+j))^2
53    float covIR = 0;                 // ∑ I(r+i,s+j) · R(i,j)
54    for (int j=0; j<hR; j++){
55      for (int i=0; i<wR; i++){
56        float valI = I.getPixelValue(r+i,s+j);
57        float valR = R.getPixelValue(i,j);
58        sumI += valI;
59        sumI2 += valI * valI;
60        covIR += valI * valR;
61      }
62    }
63    float meanI = sumI / kR; // Ī(r,s) = (∑ I(r+i,s+j))/K
64    return (covIR - kR * meanI * meanR) /
65          ((float)Math.sqrt(sumI2 - kR*meanI*meanI) * sigmaR);
66  }
67 } // end of class CorrCoeffMatcher
```

fehlender Überdeckung zu wechseln. Die Distanzfunktion weist daher sprunghafte Übergänge auf und liefert damit keinen Anhaltspunkt über die Entfernung zu einer eventuell vorhandenen Übereinstimmung.

Die Frage ist, wie man den Vergleich von Binärbildern toleranter gegenüber kleineren Abweichungen der Bildmuster machen kann. Das Ziel besteht also darin, nicht nur jene Bildposition zu finden, an der die größte Zahl von Vordergrundpixel im Template und im Referenzbild übereinstimmen, sondern nach Möglichkeit auch ein Maß dafür zu erhalten, wie weit man von diesem Ziel geometrisch entfernt ist.



(a) (b) (c)

Abbildung 17.7
Direkter Vergleich von Binärbildern. Gegeben ist ein binäres Originalbild (a) und ein binäres Template (b). Der lokale Ähnlichkeitswert an einer bestimmten Template-Position entspricht der Anzahl der übereinstimmenden (schwarzen) Vordergrundpixel. Im Ergebnis (c) sind hohe Ähnlichkeitswerte hell dargestellt. Obwohl die Vergleichsfunktion naturgemäß den Maximalwert an der korrekten Position (im Zentrum des Buchstabens 'B') aufweist, ist die eindeutige Bestimmung der korrekten Match-Position durch die vielen weiteren, lokalen Maxima schwierig.

17.2.2 Die Distanztransformation

Eine mögliche Lösung dieses Problems besteht darin, zunächst für jede Bildposition zu bestimmen, wie weit sie geometrisch vom nächsten Vordergrundpixel entfernt ist. Damit erhalten wir ein Maß für die minimale Verschiebung, die notwendig wäre, um ein bestimmtes Pixel mit einem Vordergrundpixel zur Überlappung zu bringen. Ausgehend von einem Binärbild $I(u,v) = I(\boldsymbol{p})$ bezeichnen wir zunächst

$$FG(I) = \{\boldsymbol{p} \mid I(\boldsymbol{p}) = 1\} \tag{17.11}$$
$$BG(I) = \{\boldsymbol{p} \mid I(\boldsymbol{p}) = 0\} \tag{17.12}$$

als die Menge der Koordinaten aller Vordergrund- bzw. Hintergrundpixel. Die *Distanztransformation* von I, $D(\boldsymbol{p}) \in \mathbb{R}$, ist definiert als

$$D(\boldsymbol{p}) = \min_{\boldsymbol{p}' \in FG(I)} \text{dist}(\boldsymbol{p}, \boldsymbol{p}') \tag{17.13}$$

für alle $\boldsymbol{p} = (u,v)$, wobei $u = 0 \ldots M-1$, $v = 0 \ldots N-1$ (Bildgröße $M \times N$). Falls ein Bildpunkt $\boldsymbol{p}$ selbst ein Vordergrundpixel ist (d. h. $p \in FG$), dann ist $D(\boldsymbol{p}) = 0$, da keine Verschiebung notwendig ist, um diesen Punkt mit einem Vordergrundpixel zur Überdeckung zu bringen.

Die Funktion $\text{dist}(\boldsymbol{p}, \boldsymbol{p}')$ in Gl. 17.13 misst den geometrischen Abstand zwischen zwei Koordinaten $\boldsymbol{p} = (u,v)$ und $\boldsymbol{p}' = (u',v')$. Beispiele für geeignete Distanzfunktionen sind die *euklidische* Distanz

$$\mathrm{d}_E(\boldsymbol{p}, \boldsymbol{p}') = \|\boldsymbol{p} - \boldsymbol{p}'\| = \sqrt{(u-u')^2 + (v-v')^2} \quad \in \mathbb{R}^+ \tag{17.14}$$

oder die *Manhattan*-Distanz[1]

$$\mathrm{d}_M(\boldsymbol{p}, \boldsymbol{p}') = |u-u'| + |v-v'| \quad \in \mathbb{N}_0. \tag{17.15}$$

Abb. 17.8 zeigt ein einfaches Beispiel für die Distanztransformation unter Verwendung der Manhattan-Distanz $\mathrm{d}_M()$.

[1] Auch „city block distance" genannt.

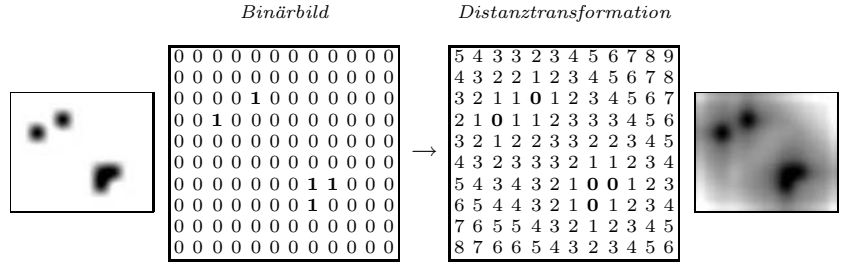


Abbildung 17.8
Beispiel für die Distanztransformation eines Binärbilds mit der Manhattan-Distanz $\mathrm{d}_M()$.

Die direkte Berechnung der Distanztransformation aus der Definition in Gl. 17.13 wäre allerdings ein relativ aufwendiges Unterfangen, da man für jede Bildkoordinate $\boldsymbol{p}$ das nächstgelegene aller Vordergrundpixel finden müsste (außer $\boldsymbol{p}$ ist selbst bereits ein Vordergrundpixel).

Der *Chamfer*-Algorithmus

Der so genannte *Chamfer*-Algorithmus [8] ist ein effizientes Verfahren zur Berechnung der Distanztransformation. Er verwendet, ähnlich wie das sequentielle Verfahren zur Regionenmarkierung (Abschn. 11.1.2), zwei aufeinander folgende Bilddurchläufe, in denen sich die berechneten Abstandswerte wellenförmig über das Bild fortpflanzen. Der erste Durchlauf startet an der linken oberen Bildecke und pflanzt die Abstandswerte in diagonaler Richtung nach unten fort, der zweite Durchlauf erfolgt in umgekehrter Richtung, jeweils unter Verwendung der „Distanzmasken“

$$M^L = \begin{bmatrix} m_2^L & m_3^L & m_4^L \\ m_1^L & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \text{und} \quad M^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & m_1^R \\ m_4^R & m_3^R & m_2^R \end{bmatrix} \tag{17.16}$$

für den ersten bzw. zweiten Durchlauf. Die Werte in M^L und M^R bezeichnen die geometrische Distanz zwischen dem aktuellen Bildpunkt (mit $\times$ markiert) und seinen relevanten Nachbarn. Sie sind abhängig von der gewählten Abstandsfunktion $\mathrm{dist}(\boldsymbol{p}, \boldsymbol{p}')$. Alg. 17.1 beschreibt die Berechnung der Distanztransformation $D(u, v)$ für ein Binärbild $I(u, v)$ mithilfe dieser Distanzmasken.

Die Distanztransformation für die *Manhattan*-Distanz (Gl. 17.15) kann mit dem Chamfer-Algorithmus unter Verwendung der Masken

$$M_M^L = \begin{bmatrix} 2 & 1 & 2 \\ 1 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \text{und} \quad M_M^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & 1 \\ 2 & 1 & 2 \end{bmatrix} \tag{17.17}$$

exakt berechnet werden. In ähnlicher Weise wird die *euklidische* Distanz (Gl. 17.14) mithilfe der Masken

Algorithmus 17.1
Chamfer-Algorithmus zur Berechnung der Distanztransformation. Aus einem Binärbild I wird unter Verwendung der Distanzmasken M^L und M^R (Gl. 17.16) die Distanztransformation D (Gl. 17.13) berechnet. Für die Bildränder ist eine gesonderte Behandlung vorzusehen.

```
1:  DistanceTransform (I)
        I: binary image of size M × N
    Step 1 – initialize:
2:  Set up a distance map D(u,v) ∈ ℝ of size M × N
3:  for all image coordinates (u,v) do
4:      if I(u,v) = 1 then
5:          D(u,v) ← 0                    ▷ foreground pixel (zero distance)
6:      else
7:          D(u,v) ← ∞                    ▷ background pixel (infinite distance)
    Step 2 – L→R pass (using distance mask M^L = m_i^L):
8:  for v ← 1, 2, ..., N−1 do             ▷ top → bottom
9:      for u ← 1, 2, ..., M−2 do         ▷ left → right
10:         if D(u,v) > 0 then
11:             d1 ← m_1^L + D(u−1, v)
12:             d2 ← m_2^L + D(u−1, v−1)
13:             d3 ← m_3^L + D(u, v−1)
14:             d4 ← m_4^L + D(u+1, v−1)
15:             D(u,v) ← min(d1, d2, d3, d4)
    Step 3 – R→L pass (using distance mask M^R = m_i^R):
16: for v ← N−2, ..., 1, 0 do             ▷ bottom → top
17:     for u ← M−2, ..., 2, 1 do         ▷ right → left
18:         if D(u,v) > 0 then
19:             d1 ← m_1^R + D(u+1, v)
20:             d2 ← m_2^R + D(u+1, v+1)
21:             d3 ← m_3^R + D(u, v+1)
22:             d4 ← m_4^R + D(u−1, v+1)
23:             D(u,v) ← min(D(u,v), d1, d2, d3, d4)
24: return D
```

$$M_E^L = \begin{bmatrix} \sqrt{2} & 1 & \sqrt{2} \\ 1 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \text{und} \quad M_E^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & 1 \\ \sqrt{2} & 1 & \sqrt{2} \end{bmatrix} \tag{17.18}$$

realisiert, wobei jedoch über die Fortpflanzung der lokalen Distanzen nur eine *Approximation* des tatsächlichen Minimalabstands möglich ist. Diese ist allerdings immer noch genauer als die Schätzung auf Basis der Manhattan-Distanz. Wie in Abb. 17.9 dargestellt, werden in diesem Fall die Abstände in Richtung der Koordinatenachsen und der Diagonalen zwar exakt berechnet, für die dazwischenliegenden Richtungen sind die geschätzten Distanzwerte jedoch zu hoch. Eine genauere Approximation ist mithilfe größerer Distanzmasken (z. B. 5×5, siehe Aufg. 17.4) möglich, mit denen die exakten Abstände zu Bildpunkten in einer größeren Umgebung einbezogen werden [8]. Darüber hinaus kann man Gleitkommaoperationen durch Verwendung von skalierten, ganzzahligen Distanzmasken vermeiden, beispielsweise mit den Masken

$$M_{E'}^L = \begin{bmatrix} 4 & 3 & 4 \\ 3 & \times & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \text{und} \quad M_{E'}^R = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \times & 3 \\ 4 & 3 & 4 \end{bmatrix} \tag{17.19}$$

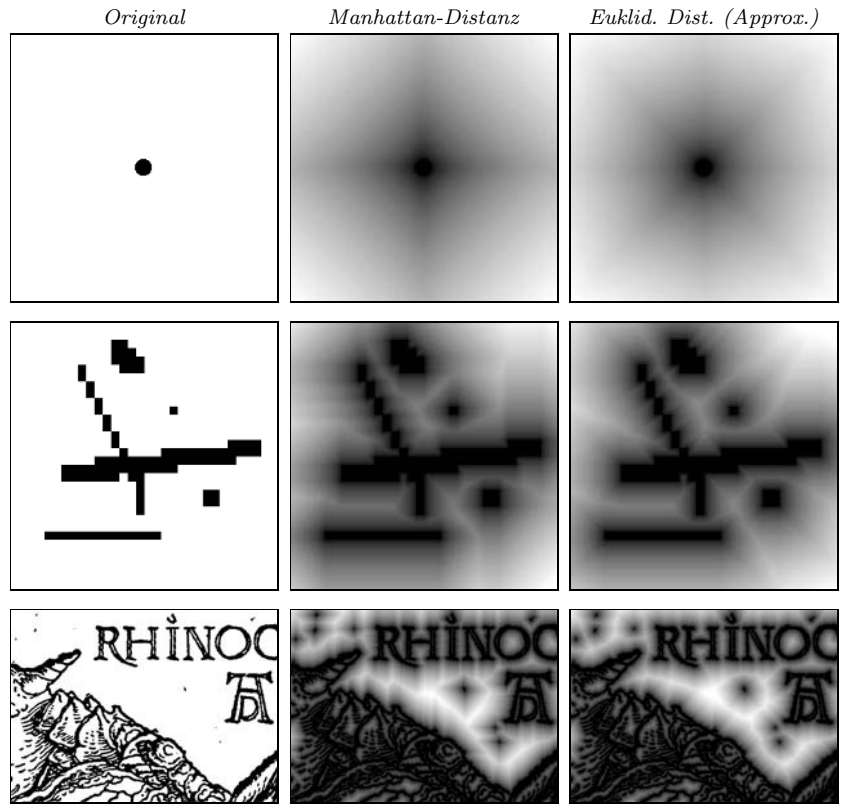


Abbildung 17.9
Distanztransformation mit dem *Chamfer*-Algorithmus. Ursprüngliches Binärbild mit schwarzem Vordergrund (links). Ergebnis der Distanztransformation für die Manhattan-Distanz (Mitte) und die euklidische Distanz (rechts). Die Helligkeitswerte (skaliert auf vollen Kontrastumfang) entsprechen dem geschätzten Abstand zum nächstliegenden Vordergrundpixel (hell = großer Abstand).

für die euklidische Distanz, wobei sich gegenüber den Masken in Gl. 17.18 etwa die dreifachen Werte ergeben.

17.2.3 *Chamfer Matching*

Nachdem wir in der Lage sind, für jedes Binärbild in effizienter Weise die Distanztransformation zu berechnen, werden wir diese nun für den Bildvergleich einsetzen. *Chamfer Matching* (erstmals in [6] beschrieben) verwendet die Distanzverteilung, um die maximale Übereinstimmung zwischen einem Binärbild I und einem (ebenfalls binären) Template R zu lokalisieren. Anstatt, wie beim direkten Vergleich (Abschn. 17.2.1), die überlappenden Vordergrundpixel zu zählen, verwendet Chamfer Matching die summierten Werte der Distanzverteilung als Maß Q für die Übereinstimmung. Das Template R wird über das Bild bewegt und für jedes Vordergrundpixel innerhalb des Templates, $(i,j) \in FG(R)$, wird der zugehörige Wert der Distanzverteilung D addiert, d. h.

$$Q(r,s) = \frac{1}{K} \sum_{(i,j)\in FG(R)} D(r+i, s+i)\,, \tag{17.20}$$

wobei $K = |FG(R)|$ die Anzahl der Vordergrundpixel im Template R bezeichnet.

```
1:  ChamferMatch (I, R)
        I: binary image of size w_I × h_I
        R: binary template of size w_R × h_R

2:      Step 1 – initialize:
3:      D ← DistanceTransform(I)                          ▷ Alg. 17.1
4:      K ← number of foreground pixels in R
5:      Set up a match map Q of size (w_I − w_R + 1) × (h_I − h_R + 1)

6:      Step 2 – compute match function:
7:      for r ← 0 ... (w_I − w_R) do          ▷ set origin of template to (r, s)
8:          for s ← 0 ... (h_I − h_R) do
9:              Evaluate match for template positioned at (r, s):
10:             q ← 0
11:             for i ← 0 ... (w_R − 1) do
12:                 for j ← 0 ... (h_R − 1) do
13:                     if R(i, j) = 1 then      ▷ foreground pixel in template
14:                         q ← q + D(r + i, s + j)
15:             Q(r, s) ← q/K
16:     return Q
```

Algorithmus 17.2
Chamfer Matching – Berechnung der Match-Funktion. Gegeben sind ein binäres Binärbild I und ein binäres Template R. Im ersten Schritt wird die Distanztransformation D für das Bild I berechnet (s. Alg. 17.1). Anschließend wird für jede Position des Templates R die entsprechende Summe der Werte in der Distanzverteilung ermittelt. Die Ergebnisse werden in der zweidimensionalen Match-Funktion Q abgelegt und zurückgegeben.

Der gesamte Ablauf zur Berechnung der Match-Funktion Q ist in Alg. 17.2 zusammengefasst. Wenn an einer Position alle Vordergrundpixel des Templates R eine Entsprechung im Bild I finden, dann ist die Summe der Distanzwerte null und es liegt eine perfekte Übereinstimmung (*match*) vor. Je mehr Vordergrundpixel des Templates Distanzwerte in D „vorfinden“, die größer als null sind, umso höher wird die Summe der Distanzen Q bzw. umso schlechter die Übereinstimmung. Die beste Übereinstimmung ergibt sich dort, wo Q ein Minimum aufweist, d. h.

$$\boldsymbol{p}_{\text{opt}} = (r_{\text{opt}}, s_{\text{opt}}) = \underset{(r,s)}{\operatorname{argmin}}\, Q(r, s). \tag{17.21}$$

Das Beispiel in Abb. 17.10 demonstriert den Unterschied zwischen dem direkten Bildvergleich und *Chamfer Matching* anhand des Binärbilds aus Abb. 17.7. Wie deutlich zu erkennen ist, erscheint die Match-Funktion des Chamfer-Verfahrens wesentlich glatter und weist nur wenige, klar ausgeprägte lokale Maxima auf. Dies ermöglicht das effektive Auffinden der Stellen optimaler Übereinstimmung mittels einfacher, lokaler Suchmethoden und ist damit ein wichtiger Vorteil. Abb. 17.11 zeigt ein weiteres Beispiel mit Kreisen und Quadraten, wobei die Kreise verschiedene Durchmesser aufweisen. Das verwendete Template besteht aus dem Kreis mittlerer Größe. Wie das Beispiel zeigt, toleriert Chamfer Matching auch geringfügige Größenabweichungen zwischen dem Bild und dem Vergleichsmuster und erzeugt auch in diesem Fall eine relativ glatte Match-Funktion mit deutlichen Spitzenwerten.

Chamfer Matching ist zwar kein „silver bullet“, funktioniert aber unter eingeschränkten Bedingungen und in passenden Anwendungen durch-

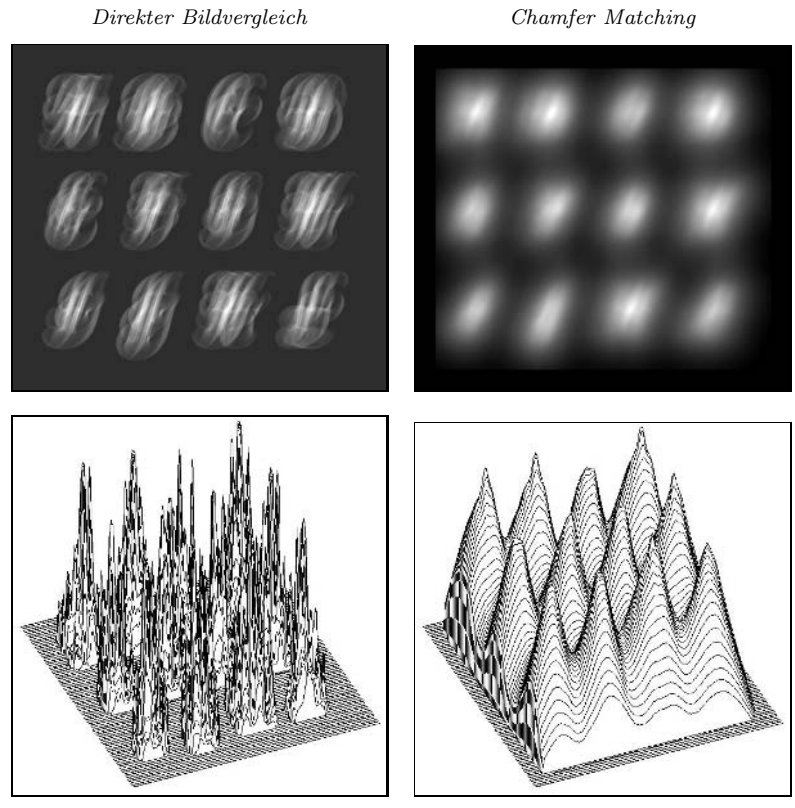
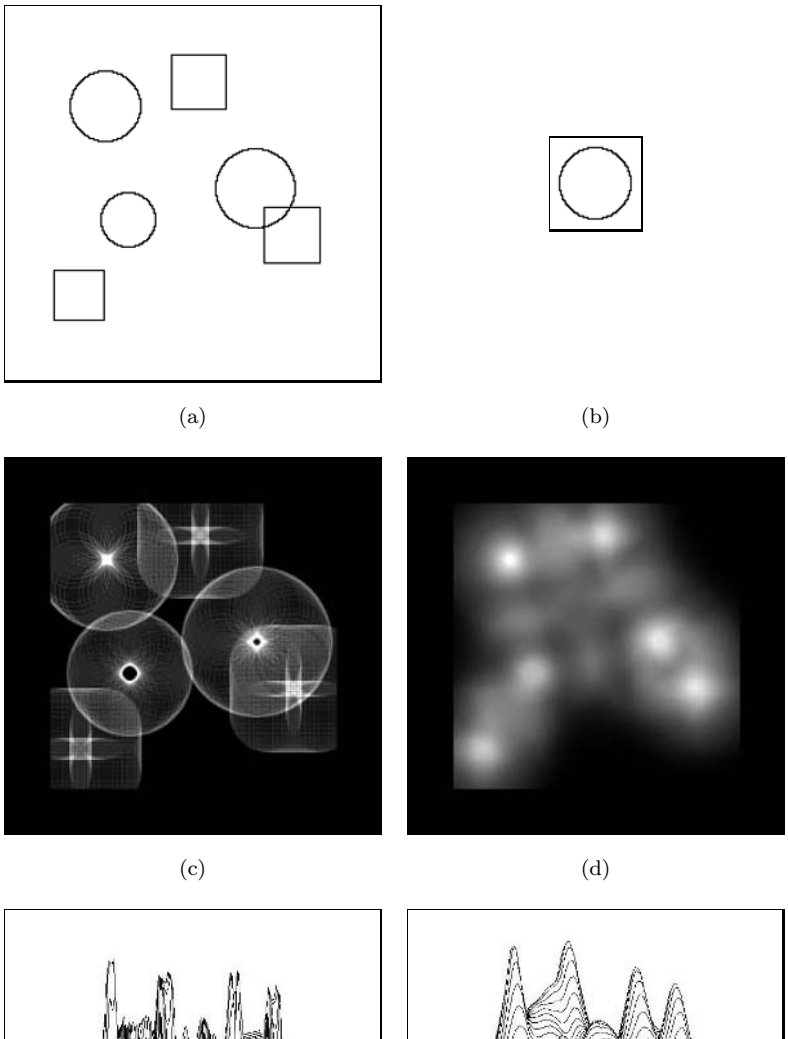


Abbildung 17.10
Direkter Bildvergleich vs. Chamfer Matching (Originalbilder s. Abb. 17.7). Gegenüber dem Ergebnis des direkten Bildvergleichs (a) ist die Match-Funktion Q aus dem *Chamfer Matching* (b) wesentlich glatter. Sie weist an den Stellen hoher Übereinstimmung deutliche Spitzenwerte auf, die mit lokalen Suchmethoden leicht aufzufinden sind. Zum besseren Vergleich ist die Match-Funktion Q (s. Gl. 17.20) in (b) invertiert.

aus zufriedenstellend und effizient. Problematisch sind natürlich Unterschiede in Form, Lage und Größe der gesuchten Bildmuster, denn die Match-Funktion ist nicht invariant gegenüber Skalierung, Rotation oder Verformungen. Da die Methode auf der minimalen Distanz der Vordergrundpixel basiert, verschlechtert sich überdies das Ergebnis rasch, wenn die Bilder mit zufälligen Störungen (*clutter*) versehen sind oder großflächige Vordergrundregionen enthalten. Eine Möglichkeit zur Reduktion der Wahrscheinlichkeit falscher Match-Ergebnisse besteht darin, anstelle der einfachen (linearen) Summe (Gl. 17.20) den quadratischen Durchschnitt der Distanzwerte, also

$$Q_{rms}(r,s) = \sqrt{\frac{1}{K} \sum_{(i,j)\in FG(R)} \big(D(r+i,s+i)\big)^2} \tag{17.22}$$

als Messwert für die Übereinstimmung zwischen dem aktuellen Bildausschnitt und dem Template R zu verwenden [8]. Auch hierarchische Versionen des Chamfer-Verfahrens wurden vorgeschlagen [9], insbesondere um die Robustheit zu verbessern und den Suchaufwand zu reduzieren.



(a) (b)

(c) (d)

(e) (f)

Abbildung 17.11
Chamfer Matching bei Größenänderung. Binärbild mit geometrischen Formen unterschiedlicher Größe (a) und Template (b), das zum mittleren der drei Kreise identisch ist. Gegenüber dem direkten Bildvergleich (c, e) ergibt das Ergebnis des Chamfer-Verfahrens (d, f) eine glatte Funktion mit leicht zu lokalisierenden Spitzenwerten. Man beachte, dass sowohl die drei unterschiedlich großen Kreise wie auch die ähnlich großen Quadrate in (f) ausgeprägt hohe Match-Werte zeigen.

17.3 Aufgaben

Aufg. 17.1. Adaptieren Sie das in Abschn. 17.1 beschriebene Template-Matching-Verfahren für den Vergleich von RGB-Farbbildern.

Aufg. 17.2. Implementieren Sie das *Chamfer*-Verfahren (Alg. 17.1) für Binärbilder mit der *euklidischen* Distanz und der *Manhattan*-Distanz.

Aufg. 17.3. Implementieren Sie die exakte euklidische Distanztransformation durch „brute-force"-Suche nach dem jeweils nächstgelegenen Vordergrundpixel (das könnte einiges an Rechenzeit benötigen). Vergleichen Sie das Ergebnis mit der Approximation durch das Chamfer-Verfahren (Alg. 17.1) und bestimmen Sie die maximale Abweichung (in %).

Aufg. 17.4. Modifizieren Sie den Chamfer-Algorithmus (Alg. 17.1) unter Verwendung folgender 5×5-Distanzmasken anstelle der Masken in Gl. 17.18 zur Schätzung der euklidischen Distanz:

$$M^L = \begin{bmatrix} \cdot & 2.236 & \cdot & 2.236 & \cdot \\ 2.236 & 1.414 & 1.000 & 1.414 & 2.236 \\ \cdot & 1.000 & \times & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}, \quad M^R = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \times & 1.000 & \cdot \\ 2.236 & 1.414 & 1.000 & 1.414 & 2.236 \\ \cdot & 2.236 & \cdot & 2.236 & \cdot \end{bmatrix}.$$

Vergleichen Sie die Ergebnisse mit dem Standardverfahren (mit 3×3-Masken). Begründen Sie, warum zusätzliche Maskenelemente in Richtung der Hauptachsen und der Diagonalen überflüssig sind.

Aufg. 17.5. Implementieren Sie das Chamfer Matching unter Verwendung des linearen Durchschnitts (Gl. 17.20) und des quadratischen Durchschnitts (Gl. 17.22) für die Match-Funktion. Vergleichen Sie die beiden Varianten in Bezug auf Robustheit der Ergebnisse.

A

Mathematische Notation

A.1 Häufig verwendete Symbole

Die folgenden Symbole werden im Haupttext vorwiegend in der angegebenen Bedeutung verwendet, jedoch bei Bedarf auch in anderem Zusammenhang eingesetzt. Die Bedeutung sollte aber in jedem Fall eindeutig sein.

$*$ ..... Linearer Faltungsoperator (Abschn. 6.3.1).

∂ ..... Partieller Ableitungsoperator (Abschn. 7.2.1).

∇ ..... Gradient. ∇f ist der Vektor der partiellen Ableitungen einer mehrdimensionalen Funktion f (Abschn. 7.2.1).

$\lfloor x \rfloor$ ..... „Floor“ von x, das ist die nächste ganze Zahl z, die kleiner ist als x, d. h. $z = \lfloor x \rfloor \leq x$.

$\arctan_2(y, x)$ .. Inverse Tangensfunktion $\tan^{-1}\left(\frac{y}{x}\right)$, entsprechend der Java-Methode `Math.atan2(y,x)` (Abschn. 7.3, B.1.6).

$\mathrm{card}\{\ldots\}$ ..... Kardinalität (Mächtigkeit, Anzahl der Elemente) einer Menge, $\mathrm{card}\, A \equiv |A|$ (Abschn. 4.1).

DFT ..... Diskrete Fouriertransformation (Abschn. 13.3).

$\mathcal{F}$ ..... Kontinuierliche Fouriertransformation (Abschnitt 13.1.4).

$g(x)$, $g(x, y)$ .. Ein- und zweidimensionale kontinuierliche Funktionen ($x, y \in \mathbb{R}$).

$g(u)$, $g(u, v)$ .. Ein- und zweidimensionale diskrete Funktionen ($u, v \in \mathbb{Z}$).

$G(m)$, $G(m,n)$ Ein- und zweidimensionale diskrete Spektra ($m, n \in \mathbb{Z}$).

$\mathsf{h}(i)$ ..... Histogramm eines Bilds für den Pixelwert i (Abschn. 4.1).

$\mathsf{H}(i)$ ..... Kumulatives Histogramm für den Pixelwert i (Abschn. 4.6).

$I(u,v)$ ..... Der Wert des Bilds I an der (ganzzahligen) Position (u,v).

i ..... Pixelwert ($0 \le i < K$).

i ..... Imaginäre Einheit (s. Abschn. 1.2).

K ..... Anzahl der möglichen Pixelwerte eines Bilds.

M, N ..... Anzahl der Spalten (Breite) bzw. Zeilen (Höhe) eines Bilds ($0 \le u < M$, $0 \le v < N$).

mod ..... Modulo-Operator: $a \bmod b$ ist der Rest der ganzzahligen Division a/b (Abschn. 13.4).

$\mathsf{p}(i)$ ..... Wahrscheinlichkeitsdichtefunktion (Abschn. 5.5.1).

$\mathsf{P}(i)$ ..... Verteilungsfunktion oder kumulative Wahrscheinlichkeitsdichte (Abschn. 5.5.1).

$\mathcal{Q}$ ..... Viereck (Abschn. 16.1.4).

$\mathrm{Round}(x)$ ..... Rundungsfunktion: $\mathrm{Round}(x) = \lfloor x + 0.5 \rfloor$ (Abschn. 16.3.1).

$\mathcal{S}_1$ ..... Einheitsquadrat (Abschn. 16.1.4).

A.2 Komplexe Zahlen $\mathbb{C}$

Definitionen:

$$z = a + \mathrm{i}b, \qquad z, \mathrm{i} \in \mathbb{C},\ a, b \in \mathbb{R},\ i^2 = -1 \tag{A.1}$$

$$z^* = a - \mathrm{i}b \qquad \text{(konjugiert-komplexe Zahl)} \tag{A.2}$$

$$sz = sa + \mathrm{i}sb, \qquad s \in \mathbb{R} \tag{A.3}$$

$$|z| = \sqrt{a^2 + b^2}, \quad |sz| = s|z| \tag{A.4}$$

$$\begin{aligned} z &= a + \mathrm{i}b = |z| \cdot (\cos\psi + \mathrm{i}\sin\psi) \\ &= |z| \cdot e^{\mathrm{i}\psi},\ \text{wobei}\ \ \psi = \tan^{-1}(b/a) \end{aligned} \tag{A.5}$$

$$\mathrm{Re}\big(a + \mathrm{i}b\big) = a, \quad \mathrm{Re}\big(e^{\mathrm{i}\varphi}\big) = \cos\varphi \tag{A.6}$$

$$\mathrm{Im}\big(a + \mathrm{i}b\big) = b, \quad \mathrm{Im}\big(e^{\mathrm{i}\varphi}\big) = \sin\varphi \tag{A.7}$$

$$e^{\mathrm{i}\varphi} = \cos\varphi + \mathrm{i} \cdot \sin\varphi \tag{A.8}$$

$$e^{-\mathrm{i}\varphi} = \cos\varphi - \mathrm{i} \cdot \sin\varphi \tag{A.9}$$

$$\cos(\varphi) = \tfrac{1}{2} \cdot \big(e^{\mathrm{i}\varphi} + e^{-\mathrm{i}\varphi}\big) \tag{A.10}$$

$$\sin(\varphi) = \tfrac{1}{2i} \cdot \big(e^{\mathrm{i}\varphi} - e^{-\mathrm{i}\varphi}\big) \tag{A.11}$$

Rechenoperationen:

$$z_1 = (a_1 + \mathrm{i}b_1) = |z_1|\, e^{\mathrm{i}\varphi_1} \tag{A.12}$$

$$z_2 = (a_2 + \mathrm{i}b_2) = |z_2|\, e^{\mathrm{i}\varphi_2} \tag{A.13}$$

$$z_1 + z_2 = (a_1 + b_1) + \mathrm{i}(b_1 + b_2) \tag{A.14}$$

$$\begin{aligned} z_1 \cdot z_2 &= (a_1 a_2 - b_1 b_2) + \mathrm{i}(a_1 b_2 + b_1 a_2) \\ &= |z_1| \cdot |z_2| \cdot e^{\mathrm{i}(\varphi_1 + \varphi_2)} \end{aligned} \tag{A.15}$$

$$\begin{aligned} \frac{z_1}{z_2} &= \frac{a_1 a_2 + b_1 b_2}{a_2^2 + b_2^2} + \mathrm{i}\frac{a_2 b_1 - a_1 b_2}{a_2^2 + b_2^2} \\ &= \frac{|z_1|}{|z_2|} \cdot e^{\mathrm{i}(\varphi_1 - \varphi_2)} \end{aligned} \tag{A.16}$$

A.3 Algorithmische Komplexität und $\mathcal{O}$-Notation

Unter „Komplexität“ versteht man den Aufwand, den ein Algorithmus zur Lösung eines Problems benötigt, in Abhängigkeit von der so genannten „Problemgröße“ N. In der Bildverarbeitung ist dies üblicherweise die Bildgröße oder auch beispielsweise die Anzahl der Bildregionen. Man unterscheidet üblicherweise zwischen der *Speicher*komplexität und der *Zeit*komplexität, also dem Speicher- bzw. Zeitaufwand eines Verfahrens. Ausgedrückt wird die Komplexität in der Form $\mathcal{O}(N)$, was auch als „big Oh“-Notation bezeichnet wird.

Möchte man beispielsweise die Summe aller Pixelwerte eines Bilds der Größe $M \times N$ berechnen, so sind dafür i. Allg. $M \cdot N$ Schritte (Additionen) erforderlich, das Verfahren hat also eine Zeitkomplexität „der Ordnung MN“, oder

$$\mathcal{O}(MN).$$

Da die Zahl der Bildzeilen und -spalten eine ähnliche Größenordung aufweist, werden sie üblicherweise als identisch (N) angenommen und die Komplexität beträgt in diesem Fall

$$\mathcal{O}(N^2).$$

Die direkte Berechnung der linearen *Faltung* (Abschn. 6.3.1) für ein Bild der Größe $N \times N$ und einer Filtermatrix der Größe $K \times K$ hätte beispielweise die Zeitkomplexität $\mathcal{O}(N^2K^2)$. Die *Fast Fourier Transform* (FFT, s. Abschn. 13.4.2) berechnet das Spektrum eines Signalvektors der Länge $N = 2^k$ in der Zeit $\mathcal{O}(N \log_2 N)$.

Dabei wird eine konstante Anzahl zusätzlicher Schritte, etwa für die Initialisierung, nicht eingerechnet. Auch multiplikative Faktoren, beispielsweise wenn pro Pixel jeweils 5 Schritte erforderlich wären, werden in der $\mathcal{O}$-Notation nicht berücksichtigt. Mithilfe der $\mathcal{O}$-Notation können daher Algorithmen in Bezug auf ihre Effizienz klassifiziert und verglichen werden. Details dazu finden sich in jedem Buch über Computeralgorithmen, wie z. B. [2].

B

Java-Notizen

Als Text für den ersten Abschnitt einer technischen Studienrichtung setzt dieses Buch gewisse Grundkenntnisse in der Programmierung voraus. Anhand eines der vielen verfügbaren Java-Tutorials oder eines einführenden Buchs sollten alle Beispiele im Text leicht zu verstehen sein. Die Erfahrung zeigt allerdings, dass viele Studierende auch nach mehreren Semestern noch Schwierigkeiten mit einigen grundlegenden Konzepten in Java haben und einzelne Details sind regelmäßig Anlass für Komplikationen. Im folgenden Abschnitt sind daher einige typische Problempunkte zusammengefasst.

B.1 Arithmetik

Java ist eine Programmiersprache mit einem strengen Typenkonzept und ermöglicht insbesondere nicht, dass eine Variable dynamisch ihren Typ ändert. Auch ist das Ergebnis eines Ausdrucks im Allgemeinen durch die Typen der beteiligten Operanden bestimmt und – im Fall einer Wertzuweisung – *nicht* durch die „aufnehmende“ Variable.

B.1.1 Ganzzahlige Division

Die Division von ganzzahligen Operanden ist eine häufige Fehlerquelle. Angenommen, `a` und `b` sind beide vom Typ `int`, dann folgt auch der Ausdruck `(a / b)` den Regeln der ganzzahligen Division und berechnet, wie oft `b` in `a` *enthalten* ist. Auch das Ergebnis ist daher wiederum vom Typ `int`. Zum Beispiel ist nach Ausführung der Anweisungen

```
int a = 2;
int b = 5;
double c = a / b;
```

der Wert von `c` *nicht* 0.4, sondern 0.0, weil der Ausdruck `a / b` auf der rechten Seite den `int`-Wert 0 ergibt, der bei der nachfolgenden Zuweisung auf `c` automatisch auf den `double`-Wert 0.0 konvertiert wird.

Wollten wir `a / b` als Gleitkommaoperation berechnen, so müssen wir zunächst mindestens einen der Operanden in einen Gleitkommawert umwandeln, beispielsweise durch einen expliziten *type cast* `(double)`:

```
double c = (double) a / b;
```

Dabei ist zu beachten, dass `(double)` nur den unmittelbar nachfolgenden Wert `a` betrifft und nicht den gesamten Ausdruck `a / b`, d. h. der Wert `b` behält den Typ `int`.

Beispiel

Nehmen wir z. B. an, wir möchten die Pixelwerte p_i eines Bilds so skalieren, dass der momentan größte Pixelwert $p_{\max}$ auf 255 abgebildet wird (s. auch Kap. 5). Mathematisch werden die Pixelwerte einfach in der Form

$$q \leftarrow \frac{p_i}{p_{\max}} \cdot 255$$

skaliert und man ist leicht versucht, dies 1:1 in Java-Anweisungen umzusetzen, etwa so:

```
int p_max = ip.getMaxValue();
...
int p = ip.getPixel(u,v);
int q = (p / p_max) * 255;
...
```

Wie wir leicht vorhersagen können, bleibt das Bild schwarz, mit Ausnahme der Bildpunkte mit dem ursprünglichen Pixelwert `p_max` (was wird mit diesen?). Der Grund liegt wiederum in der Division `(p/p_max)` mit zwei `int`-Operanden, wobei der Divisor (`p_max`) in den meisten Fällen größer ist als der Dividend (`p`) und die Division daher null ergibt.

Natürlich könnte man die gesamte Operation auch (wie oben gezeigt) mit Gleitkommawerten durchführen, aber das ist in diesem Fall gar nicht notwendig. Wir können stattdessen die Reihenfolge der Operationen vertauschen und die Multiplikation zuerst durchführen:

```
int q = p * 255 / p_max;
```

Die Multiplikation `p * 255` erzeugt zunächst große Zwischenwerte, die für die nachfolgende (ganzzahlige) Division nunmehr kein Problem darstellen. In Java werden übrigens arithmetische Ausdrücke auf der gleichen Ebene immer von links nach rechts berechnet, deshalb sind hier auch keine zusätzlichen Klammern notwendig (diese würden aber auch nicht schaden).

B.1.2 Modulo-Operator

Der *Modulo*-Operator $a \bmod b$ erzeugt der Rest der ganzzahligen Division a/b. mod ist in Java selbst nicht direkt verfügbar, für ganzzahlige, positive Operanden a, b liefert jedoch der *Rest*-Operator[1]

```
a % b
```

die richtigen Ergebnisse. Ist jedoch einer der Operanden negativ, so kann man sich mit folgender Methode behelfen, die der mathematischen Definition entspricht:

```
static int Mod(int m, int n) {
    int q = m / n;
    if (m * n >= 0)
        return m - n * q;
    else
        return m - n * q + n;
}
```

B.1.3 Unsigned Bytes

Die meisten Grauwert- und Indexbilder in Java und ImageJ bestehen aus Bildelementen vom Datentyp `byte`, wie auch die einzelnen Komponenten von Farbbildern. Ein einzelnes Byte hat acht Bit und kann daher $2^8 = 256$ verschiedene Bitmuster oder Werte darstellen, für Bildwerte idealerweise den Wertebereich $0 \ldots 255$. Leider gibt es in Java (etwa im Unterschied zu C/C++) keinen 8-Bit-Datentyp mit diesem Wertebereich, denn der Typ `byte` besitzt ein Vorzeichen und damit in Wirklichkeit den Wertebereich $-128 \ldots 127$.

Man kann die 8 Bits eines `byte`-Werts dennoch zur Darstellung der Werte von $0 \ldots 255$ nutzen, allerdings muss man zu Tricks greifen, wenn man mit diesen Werten *rechnen* möchte. Wenn wir beispielsweise die Anweisungen

```
int  p = 200;
byte b = (byte) p;
```

ausführen, dann weisen die Variablen `p` und `b` folgende Bitmuster auf:

```
p = 00000000000000000000000011001000
b = ........................11001000
```

Als `byte` (mit dem obersten Bit als Vorzeichen)[2] interpretiert, hat die Variable `b` aus Sicht von Java den Dezimalwert -56. Daher hat etwa nach der Anweisung

```
int  p1 = b;                // p1 == -56
```

[1] Für das Ergebnis `c = a % b` gilt: `(a / b) * b + c = a`.

[2] Die Darstellung von negativen Zahlen erfolgt in Java wie üblich durch 2er-Komplement.

Tabelle B.1
Methoden und Konstanten der `Math`-Klassse in Java.

`double abs(double a)`	`double max(double a, double b)`
`int abs(int a)`	`float max(float a, float b)`
`float abs(float a)`	`int max(int a, int b)`
`long abs(long a)`	`long max(long a, long b)`
`double ceil(double a)`	`double min(double a, double b)`
`double floor(double a)`	`float min(float a, float b)`
`double rint(double a)`	`int min(int a, int b)`
`long round(double a)`	`long min(long a, long b)`
`int round(float a)`	`double random()`
`double toDegrees(double rad)`	`double toRadians(double deg)`
`double sin(double a)`	`double asin(double a)`
`double cos(double a)`	`double acos(double a)`
`double tan(double a)`	`double atan(double a)`
`double atan2(double y, double x)`	
`double log(double a)`	`double exp(double a)`
`double sqrt(double a)`	`double pow(double a, double b)`
`double E`	`double PI`

die Variable `p1` ebenfalls den Wert -56! Um dennoch mit dem vollen 8-Bit-Wert in `b` rechnen zu können, müssen wir Javas Arithmetik umgehen, indem wir den Inhalt von `b` als *Bitmuster* verkleiden in der Form

```
int  p2 = (0xff & b);      // p2 == 200
```

Nun weist `p2` tatsächlich den gewünschten Wert 200 auf und wir haben damit einen Weg, Daten vom Typ `byte` auch in Java als `unsigned byte` zu verwenden. Praktisch alle Zugriffe auf einzelne Pixel sind in ImageJ selbst in dieser Form implementiert und damit wesentlich schneller als bei Verwendung der `getPixel()`-Zugriffsmethoden.

B.1.4 Mathematische Funktionen (`Math`-Klasse)

In Java sind die wichtigsten mathematischen Funktionen als statische Methoden in der Klasse `Math` verfügbar (Tabelle B.1).

`Math` ist Teil des `java.lang`-Package und muss daher nicht explizit importiert werden. Die meisten `Math`-Methoden arbeiten mit Argumenten vom Typ `double` und erzeugen auch Rückgabewerte vom Typ `double`. Zum Beispiel wird die Kosinusfunktion $y = \cos(x)$ folgendermaßen aufgerufen:

```
double x;
double y = Math.cos(x);
```

Numerische Konstanten, wie beispielsweise π, erhält man in der Form

```
double x = Math.PI;
```

B.1.5 Runden

Für das *Runden* von Gleitkommwerten stellt `Math` (verwirrenderweise) gleich *drei* Methoden zur Verfügung:

```
double rint(double a)
long   round(double a)
int    round(float a)
```

Um beispielsweise einen `double`-Wert `x` auf `int` zu runden, gibt es daher folgende Möglichkeiten:

```
double x; int k;
k = (int) Math.rint(x);
k = (int) Math.round(x);
k = Math.round((float)x);
```

B.1.6 Inverse Tangensfunktion

Die inverse Tangensfunktion $\varphi = \tan^{-1}(a)$ bzw. $\varphi = \arctan(a)$ findet sich im Text an mehreren Stellen und kann in dieser Form mit der Methode `atan(double a)` aus der `Math`-Klasse direkt berechnet werden (Tabelle B.1). Der damit berechnete Winkel ist allerdings auf zwei Quadranten beschränkt und daher ohne zusätzliche Bedingungen mehrdeutig. Häufig ist jedoch a ohnehin durch das Seitenverhältnis zweier Katheten angegeben, also in der Form

$$\varphi = \arctan\left(\tfrac{y}{x}\right),$$

wofür wir im Text die (selbst definierte) Funktion

$$\varphi = \arctan_2(y, x)$$

verwenden. Die Funktion $\arctan_2(y, x)$ entspricht der Methode `atan2(y, x)` in der `Math`-Klasse und liefert einen Winkel φ im Intervall $-\pi \ldots \pi$, also über den vollen Kreisbogen.[3]

B.1.7 `Float` und `Double` (Klassen)

Java verwendet intern eine Gleitkommadarstellung nach IEEE-Standard. Es gibt daher für die Typen `float` und `double` auch folgende Werte:

[3] Die Funktion `atan2(y,x)` ist in den meisten Programmiersprachen (u. a. in C/ C++) verfügbar.

`POSITIVE_INFINITY`
`NEGATIVE_INFINITY`
`NaN` („not a number“)

Diese Werte sind in den zugehörigen Wrapper-Klassen `Float` bzw. `Double` als Konstanten definiert. Falls ein solcher Wert auftritt (beispielsweise `POSITIVE_ INFINITY` bei einer Division durch 0[4]), rechnet Java ohne Fehlermeldung mit dem Ergebnis weiter.

B.2 Arrays in Java

B.2.1 Arrays erzeugen

Im Unterschied zu den meisten traditionellen Programmiersprachen (wie FORTRAN oder C) können in Java Arrays *dynamisch* angelegt werden, d. h., die Größe eines Arrays kann durch eine Variable oder einen arithmetischen Ausdruck spezifiziert werden, zum Beispiel:

```
int N = 20;
int[] A = new int[N];
int[] B = new int[N*N];
```

Einmal angelegt, ist aber auch in Java die Größe eines Arrays fix und kann nachträglich nicht mehr geändert werden. Java stellte allerdings eine Reihe sehr flexibler *Container*-Klassen (z. B. die Klasse `Vector`) für verschiedenste Anwendungszwecke zur Verfügung. Einer Array-Variablen kann nach ihrer Definition jederzeit ein anderes Array geeigneten Typs (oder der Wert `null`) zugewiesen werden:

```
A = B;     // A now points to B's data
B = null;
```

Die obige Anweisung `A = B` führt übrigens dazu, dass das usprünglich an `A` gebundene Array nicht mehr zugreifbar ist und daher zu *garbage* wird. Im Unterschied zu C und C++ ist jedoch die explizite Freigabe von Speicherplatz in Java nicht erforderlich – dies erledigt der eingebaute „Garbage Collector“.

Angenehmerweise ist in Java auch sichergestellt, dass neu angelegte Arrays mit numerischen Datentypen (`int`, `float`, `double` etc.) automatisch auf null initalisiert werden.

B.2.2 Größe von Arrays

Da ein Array dynamisch erzeugt werden kann, ist es wichtig, dass seine Größe auch zu Laufzeit festgestellt werden kann. Dies geschieht durch Zugriff auf das `length`-Attribut des Arrays:[5]

[4] Das gilt nur für die Division mit Gleitkommawerten. Die Division durch einen ganzzahligen Wert 0 führt auch in Java zu einem Fehler (*exception*).

[5] Man beachte, dass `length` bei Arrays keine Methode ist!

```
int k = A.length;  // number of elements in A
```

Es mag dabei überraschen, dass in Java die Anzahl der Elemente eines Array-Objekts auch 0 (nicht `null`) sein kann! Ist ein Array mehrdimensional, so muss die Größe jeder Dimension einzeln abgefragt werden (s. unten). Die Größe ist eine Eigenschaft des Arrays selbst und kann daher auch von Array-Argumenten innerhalb einer Methode abgefragt werden. Anders als etwa in C ist es daher nicht notwendig, die Größe eines Arrays als zusätzliches Funktionsargument zu übergeben.

B.2.3 Zugriff auf Array-Elemente

In Java ist der Index des ersten Elements in einem Array immer 0 und das letzte Element liegt an der Stelle $N-1$ für ein Array mit N Elementen. Um ein eindimensionales Array `A` beliebiger Größe zu durchlaufen, würde man typischerweise folgendes Konstrukt verwenden:

```
for (int i = 0; i < A.length; i++) {
    // do something with A[i]
}
```

B.2.4 Zweidimensionale Arrays

Mehrdimensionale Arrays sind eine häufige Ursache von Missverständnissen. In Java sind eigentlich alle Arrays *ein*dimensional und mehrdimensionale Arrays werden als Arrays von Arrays realisiert. Wenn wir beispielsweise die 3×3-Filtermatrix

$$H(i,j) = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

als zweidimensionales Array

```
double[][] h1 = {{1,2,3},
                 {4,5,6},
                 {7,8,9}};
```

darstellen, dann ist `h1` tatsächlich ein eindimensionales Array mit drei Elementen, die selbst wiederum eindimensionale Arrays vom Typ `double` sind.

Zeilenweise Anordnung

Die übliche Annahme ist, dass Array-Elemente zeilenweise angeordnet sind (Abb. B.1 (a)). Der erste Index entspricht der Zeilennummer j, der zweite Index der Spaltennummer i, sodass

$$H(i,j) \equiv \texttt{h1[j][i]}.$$

In diesem Fall erscheint die Initialisierung der Matrix genau in der richtigen Anordnung, anderseits ist die Reihenfolge der Indizes beim Array-Zugriff vertauscht.

Spaltenweise Anordung

Alternativ kann man sich die Anordnung der Array-Elemente spaltenweise vorstellen (Abb. B.1 (b)). In diesem Fall werden bei der Initialisierung Spaltenvektoren angegeben, d. h. für die obenstehende Matrix

```
double[][] h2 = {{1,4,7},{2,5,8},{3,6,9}};
```

Der Zugriff auf die Elemente erfolgt nun allerdings Indizes in der natürlichen Reihenfolge:

$$H(i,j) \equiv \texttt{h2[i][j]}$$

Beide Varianten sind grundsätzlich äquivalent und die Anordnung eine Frage der Konvention. Große Arrays (z. B. Bilder) sollten allerdings aus Effizienzgründen möglichst entlang aufeinander folgender Speicheradressen bearbeitet werden. In Java sind die Elemente eines mehrdimensionalen Arrays immer in der Reihenfolge des *letzten* Index im Speicher abgelegt, d. h., das Element `Arr[][]...[k]` liegt im Speicher neben dem Element `Arr[][]...[k+1]`.

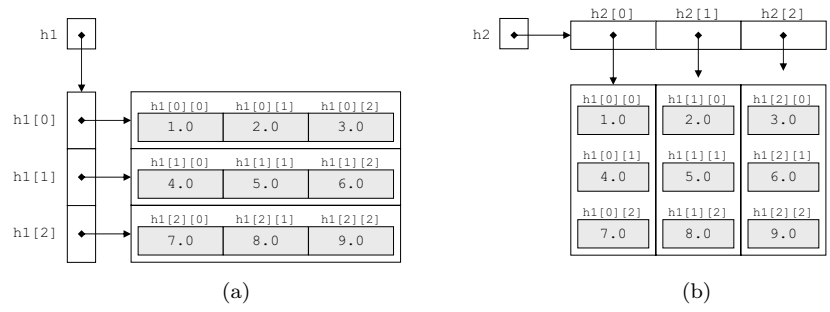


Abbildung B.1
Zweidimensionale Arrays. In Java werden mehrdimensionale Arrays als eindimensionale Arrays implementiert, deren Elemente wiederum eindimensionale Arrays sind. Ein zweidimensionales Array kann entweder als Folge von *Zeilen*vektoren (links) oder *Spalten*vektoren (rechts) interpretiert werden.

Größe mehrdimensionaler Arrays

Die Größe eines mehrdimensionalen Arrays kann dynamisch durch Abfrage der Größe seiner Sub-Arrays bestimmt werden. Zum Beispiel werden für folgendes dreidimensionale Array mit der Dimension $P \times Q \times R$

```
int A[][][] = new int[P][Q][R];
```

die einzelnen Größen ermittelt durch:

```
int p = A.length;          // = P
int q = A[0].length;       // = Q
int r = A[0][0].length;    // = R
```

Dies gilt zumindest für „rechteckige“ Arrays, bei denen alle Sub-Arrays auf einer Ebene gleiche Länge aufweisen. Andernfalls muss die Länge jedes Sub-Arrays einzeln bestimmt werden, was aus Sicherheitsgründen aber ohnehin zu empfehlen ist.

C

ImageJ-Kurzreferenz

C.1 Installation und Setup

Alle aktuellen Informationen zum Download und zur Installation finden sich auf der ImageJ-Homepage

http://rsb.info.nih.gov/ij/

Derzeit sind dort fertige Installationspakete für Linux (x86), Macintosh, Macintosh OS9/OSX und Windows verfügbar. Die nachfolgenden Informationen beziehen sich überwiegend auf die Windows-Installation (Version 1.33), die Installation ist aber für alle anderen Umgebungen ähnlich.

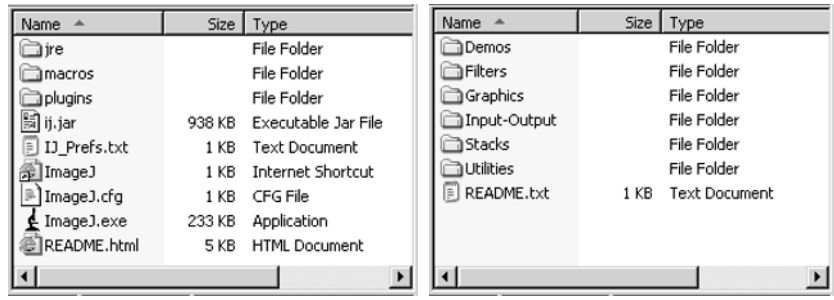
ImageJ kann in einem beliebigen Dateiordner (wir bezeichnen ihn mit `<ij>`) installiert werden und ist ohne Installation weiterer Software funktionsfähig. Abb. C.1 (a) zeigt den Inhalt des Installationsordners unter Windows, dessen wichtigste Inhalte folgende sind:

`jre`
: Die vollständige Java-Laufzeitumgebung (Java Runtime Environment), also die „Java Virtual Machine“ (JVM). Diese ist für die eigentliche Ausführung von Java-Programmen erforderlich.

`macros`
: Unterordner für ImageJ-Makros (sind hier nicht behandelt).

`plugins`
: In diesem Ordner werden die eigenen ImageJ-Plugins gespeichert. Er enthält bereits einige andere Unterordner mit Beispiel-Plugins (Abb. C.1 (b)). Eigene Plugins dürfen nicht tiefer als eine Verzeichnisebene unter diesem Ordner liegen, da sie ansonsten von ImageJ nicht akzeptiert werden.

`ij.jar`
: Eine „Java Archive“-Datei, in der die gesamte Basisfunktionalität von ImageJ enthalten ist. Bei einem Update auf eine neuere



(a) (b)

Abbildung C.1
ImageJ-Installation unter Windows. Inhalt des Installationsordners `<ij>` (a) und des Unterordners `plugins` (b).

Version von ImageJ muss i. Allg. nur diese eine Datei ersetzt werden. JAR-Dateien enthalten Sammlungen von binären Java-Files (`.class`) und können wie ZIP-Files geöffnet werden.

`IJpreff.txt`
: Eine Textdatei, in der diverse Optionen für ImageJ eingestellt werden können.

`ImageJ.cfg`
: Enthält die Startparameter für die Java-Laufzeitumgebung, im Normalfall die Zeile
`"jre\bin\javaw.exe -Xmx300m -cp ij.jar ij.ImageJ"`.
Die Option `-Xmx300m` bestimmt dabei, dass anfangs 300 MB Speicherplatz für Java angefordert werden. Diese Größe ist für manche Anwendungen zu klein und kann an dieser Stelle modifiziert werden.

`ImageJ.exe`
: Ein kleines Launch-Programm, das über Windows wie andere Programme gestartet wird und das anschließend selbst Java und ImageJ startet.

Um eigene Plugin-Programme zu erstellen, ist außerdem ein Texteditor zum Editieren der Java-Files und ein Java-Compiler erforderlich. Das in ImageJ verwendete Java Runtime Environment enthält beides, also auch bereits einen Compiler, sodass grundsätzlich keine weitere Software notwendig ist. Die so verfügbare Programmierumgebung ist aber selbst für kleinere Experimente unzureichend und es empfiehlt sich die Verwendung einer integrierten Java-Programmierumgebung, wie beispielsweise *Eclipse*[1], *NetBeans*[2] oder *Borland JBuilder*[3]. Damit ist insbesondere bei umfangreicheren Plugin-Projekten eine saube Projektverwaltung und Programmanalyse möglich, mit der viele Programmierfehler bereits im Vorfeld zu vermeiden sind, die andernfalls erst während der Programmausführung auftreten.

[1] www.eclipse.org
[2] www.netbeans.org
[3] www.borland.com/jbuilder

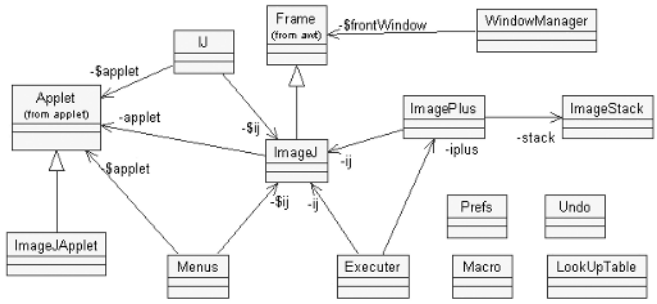


Abbildung C.2
ImageJ-Klassendiagramm für das Packacke `ij`.

C.2 ImageJ-API

Für das ImageJ-API[4] ist die vollständige Dokumentation und der gesamte Quellcode von ImageJ unter

http://rsb.info.nih.gov/ij/developer/

online bzw. zum Download verfügbar. Beide Quellen sind bei der Entwicklung eigener ImageJ-Plugins äußerst hilfreich, wie auch das ImageJ-Plugin-Tutorial von Werner Bailer [3].[5] Zusätzlich empfiehlt sich die Verwendung der Standard-Java-Dokumentation in der jeweils aktuellen Version, die man auf der Java-Homepage von Sun findet.[6] Nachfolgend ein Auszug der wichtigsten Packages und zugehörigen Klassen des ImageJ-API.[7]

C.2.1 Bilder (Package `ij`)

`ImagePlus` (Klasse)
: Eine erweiterte Variante der Standard-Java-Klasse `java.awt.Image` zur Repräsentation von Bildern. Ein Objekt der Klasse `ImagePlus` enthält ein Objekt der Klasse `ImageProcessor`, das die Funktionalität für die Verarbeitung des Bilds zur Verfügung stellt.

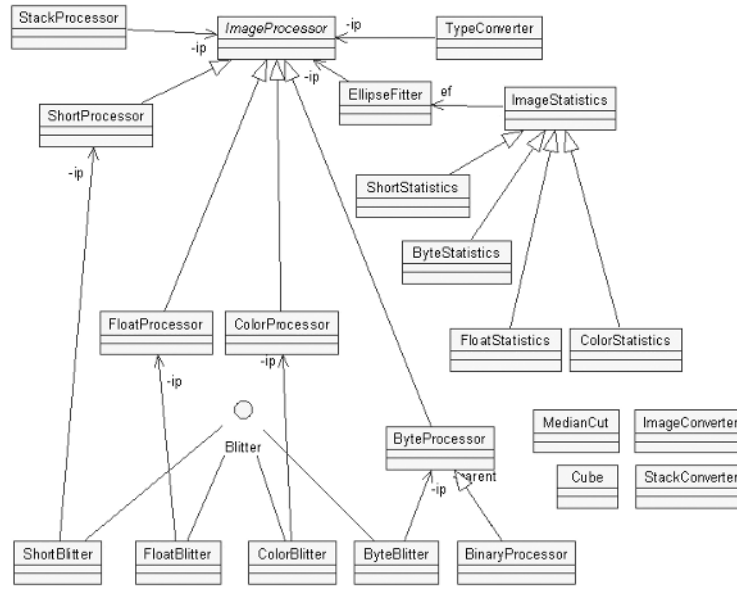
`ImageStack` (Klasse)
: Ein erweiterbarer „Stapel“ von Bildern.

C.2.2 Bildprozessoren (Package `ij.process`)

`ImageProcessor` (Klasse)
: Die (abstrakte) Überklasse für die vier in ImageJ verfügbaren Bildprozessor-Klassen `ByteProcessor`, `ShortProcessor`, `FloatProcessor` und `ColorProcessor`. Bei der Programmierung von

[4] *Application Programming Interface*
[5] www.fh-hagenberg.at/mtd/depot/imaging/imagej/
[6] http://java.sun.com/reference/api/
[7] Die UML-Klassendiagramme in Abb. C.2–C.5 entstammen der ImageJ-Site http://rsb.info.nih.gov/ij/developer/.

Abbildung C.3
ImageJ-Klassendiagramm für das Packacke `ij.process`.



Plugins hat man es meistens mit Bildern in der Form von Objekten der Klasse `ImageProcessor` bzw. deren Subklassen zu tun. `ImagePlus`-Objekte (s. oben) benötigt man vorwiegend zu Darstellung von Bildern.

`ByteProcessor` (Klasse)
: Prozessor für 8-Bit-Grauwert- und Indexfarbbilder. Die davon abgeleitete Subklasse `BinaryProcessor` implementiert Binärbilder, die nur die Werte 0 und 255 enthalten.

`ShortProcessor` (Klasse)
: Prozessor für 16-Bit-Grauwertbilder.

`FloatProcessor` (Klasse)
: Prozessor für Bilder mit 32-Bit-Gleitkommawerten.

`ColorProcessor` (Klasse)
: Prozessor für 32-Bit-RGB-Farbbilder.

C.2.3 Plugins (Packages `ij.plugin`, `ij.plugin.filter`)

`PlugIn` (Interface)
: Interface-Definition für Plugins, die Bilder importieren oder darstellen, jedoch nicht verarbeiten, oder für Plugins, die überhaupt keine Bilder verwenden.

`PlugInFilter` (Interface)
: Interface-Definition für Plugins, die Bilder verarbeiten.

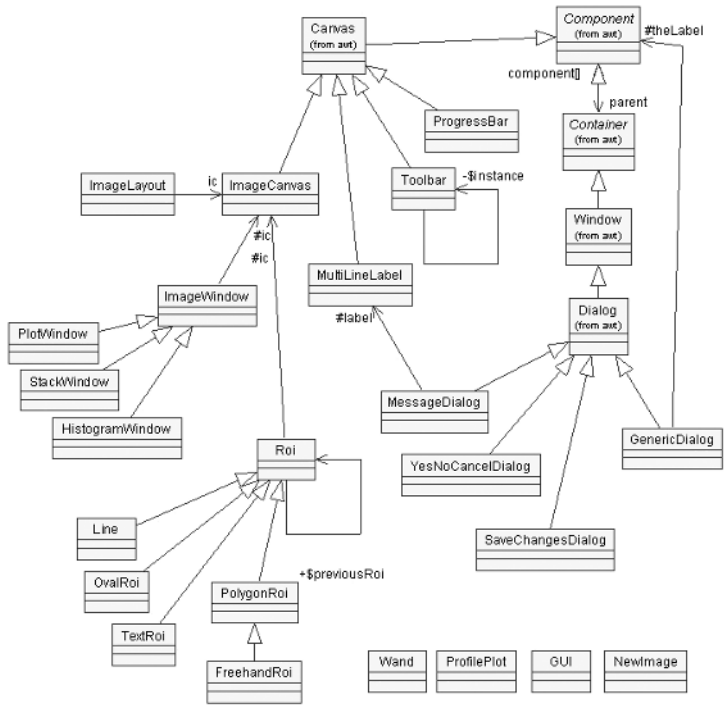


Abbildung C.4
ImageJ-Klassendiagramm für das Package `ij.gui`.

C.2.4 GUI-Klassen (Package `ij.gui`)

Die GUI[8]-Klassen von ImageJ stellen die Funktionalität zur Bildschirmdarstellung von Bildern und zu Interaktion zur Verfügung.

`ColorChooser` (Klasse)
: Repräsentiert ein Dialogfeld zur interaktiven Farbauswahl.

`NewImage` (Klasse)
: Stellt die Funktionalität zur Erzeugung neuer Bilder zur Verfügung (interaktiv und durch statische Methoden).

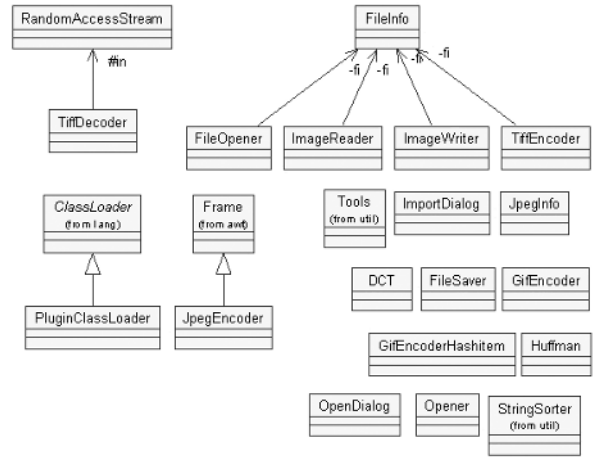
`GenericDialog` (Klasse)
: Stellt Interaktionsboxen mit frei spezifizierbaren Dialogfeldern zur Verfügung.

`ImageCanvas` (Klasse)
: `ImageCanvas` ist eine Subklasse der AWT-Klasse `Canvas` und beschreibt die Bildschirmfläche zur Darstellung von Bildern innerhalb eines Bildschirmfensters.

`ImageWindow` (Klasse)
: `ImageWindow` ist eine Subklasse der AWT-Klasse `Frame` und beschreibt ein Bildschirmfenster zur Darstellung von Bildern der Klasse `ImagePlus`. Ein `ImageWindow` enthält wiederum ein Objekt der Klasse `ImageCanvas` (s. oben) zur eigentlichen Darstellung des Bilds.

[8] Graphical User Interface.

Abbildung C.5
ImageJ-Klassendiagramm für das Package `ij.io`.



`Roi` (Klasse)
: Definiert eine rechteckige „Region of Interest“ (ROI) und bildet die Überklasse für die übrigen ROI-Klassen `Line`, `OvalRoi`, `PolygonRoi` (mit Subklasse `FreehandRoi`) und `TextRoi`.

C.2.5 Window-Management (Package `ij`)

`WindowManager` (Klasse)
: Diese Klasse stellt statische Methoden zur Verwaltung von ImageJs Bildschirmfenstern zur Verfügung.

C.2.6 Utility-Klassen (Package `ij`)

`IJ` (Klasse)
: Diese Klasse stellt statische Utility-Methoden in ImageJ zur Verfügung.

C.2.7 Input-Output (Package `ij.io`)

Das `ij.io`-Package enthält Klassen zum Öffnen (Laden) und Schreiben von Bildern von bzw. auf Dateien in verschiedenen Bildformaten.

C.3 Bilder und Bildfolgen erzeugen

C.3.1 `ImagePlus` (Klasse)

Zur Erzeugung von Bildobjekten bietet die Klasse `ImagePlus` folgende Konstruktor-Methoden:

`ImagePlus ()`
: Konstruktor-Methode: Erzeugt ein neues `ImagePlus`-Objekt ohne Initialisierung.

`ImagePlus (String pathOrURL)`
: Konstruktor-Methode: Öffnet die mit `pathOrURL` angegebene Bilddatei (TIFF, BMP, DICOM, FITS, PGM, GIF oder JPRG) oder URL (TIFF, DICOM, GIF oder JPEG) und erzeugt dafür ein neues `ImagePlus`-Objekt.

`ImagePlus (String title, Image img)`
: Konstruktor-Methode: Erzeugt ein neues `ImagePlus`-Objekt aus einem bestehenden Bild `img` vom Standard-Java-Typ `java.awt.Image`.

`ImagePlus (String title, ImageProcessor ip)`
: Konstruktor-Methode: Erzeugt ein neues `ImagePlus`-Objekt aus einem bestehenden `ImageProcessor`-Objekt `ip` mit dem Titel `title`.

`ImagePlus (String title, ImageStack stack)`
: Konstruktor-Methode: Erzeugt ein neues `ImagePlus`-Objekt für ein bestehendes `ImageStack`-Objekt `stack` mit dem Titel `title`.

`ImageStack createEmptyStack ()`
: Erzeugt einen leeren Stack mit derselben Breite, Höhe und Farbtabelle wie das aktuelle Bild (`this`).

C.3.2 `ImageStack` (Klasse)

Zur Erzeugung von Bildfolgen (Image-Stacks) bietet die Klasse `ImageStack` folgende Konstruktor-Methoden:

`ImageStack (int width, int height)`
: Konstruktor-Methode: Erzeugt ein leeres `ImageStack`-Objekt mit der Bildgröße `width` × `height`.

`ImageStack (int width, int height, ColorModel cm)`
: Konstruktor-Methode: Erzeugt ein leeres `ImageStack`-Objekt mit der Bildgröße `width` × `height` und dem Farbmodell `cm` vom Typ `java.awt. image.ColorModel`.

C.3.3 `NewImage` (Klasse)

Die Klasse `NewImage` bietet einige statische Methoden zur Erzeugung von Bildobjekten vom Typ `ImagePlus` und Image-Stacks:

`static ImagePlus createByteImage (String title, int width, int height, int slices, int fill)`
: Erzeugt ein Bild oder eine Bildfolge (wenn `slices` > 1) der Dimension `width` × `height` und dem Titel `title`. Zulässige Füllwerte (`fill`) sind `NewImage.FILL_BLACK`, `NewImage.FILL_WHITE` und `NewImage.FILL_RAMP`.

`static ImagePlus createShortImage (String title, int width, int height, int slices, int fill)`
: Erzeugt ein 16-Bit- Grauwertbild. Sonst wie oben.

`static ImagePlus createFloatImage (String title, int width, int height, int slices, int fill)`
Erzeugt ein 32-Bit-Float-Bild. Sonst wie oben.

`static ImagePlus createRGBImage (String title, int width, int height, int slices, int fill)`
Erzeugt ein 32-Bit-RGB-Bild. Sonst wie oben.

C.3.4 `ImageProcessor` (Klasse)

`Image createImage ()`
Erzeugt eine Kopie des Bilds (`ImageProcessor`) als gewöhnliches Java-AWT-Image.

C.4 Bildprozessoren erzeugen

C.4.1 `ImageProcessor` (Klasse)

`ImageProcessor createProcessor (int width, int height)`
Erzeugt ein neues `ImageProcessor`-Objekt der angegebenen Größe vom gleichen Typ wie das aktuelle Bild (`this`). Diese Methode ist für alle Subklassen von `ImageProcessor` definiert.

`ImageProcessor duplicate ()`
Erzeugt eine Kopie des bestehenden `ImageProcessor`-Objekts (`this`). Diese Methode ist für alle Subklassen von `ImageProcessor` definiert.

C.4.2 `ByteProcessor` (Klasse)

`ByteProcessor (Image img)`
Konstruktor-Methode: Erzeugt ein neues `ByteProcessor`-Objekt aus einem bestehenden Bild vom Typ `java.awt.Image`.

`ByteProcessor (int width, int height)`
Konstruktor-Methode: Erzeugt ein neues `ByteProcessor`-Objekt mit der Größe `width` × `height`.

`ByteProcessor (int width, int height, byte[] pixels, ColorModel cm)`
Konstruktor-Methode: Erzeugt ein neues `ByteProcessor`-Objekt mit der Größe `width` × `height` aus einem eindimensionalen `byte`-Array (mit Pixelwerten) und dem Farbmodell `cm` vom Typ `java.awt.image. ColorModel`.

C.4.3 `ColorProcessor` (Klasse)

`ColorProcessor (Image img)`
Konstruktor-Methode: Erzeugt ein neues `ColorProcessor`-Objekt aus einem bestehenden Bild vom Typ `java.awt.Image`.

`ColorProcessor(int width, int height)`
Konstruktor-Methode: Erzeugt ein neues `ColorProcessor`-Objekt mit der Größe `width` × `height`.

`ColorProcessor(int width, int height, int[] pixels)`
Konstruktor-Methode: Erzeugt ein neues `ColorProcessor`-Objekt mit der Größe `width` × `height` aus einem eindimensionalen `int`-Array (mit RGB-Pixelwerten).

C.4.4 `FloatProcessor` (Klasse)

`FLoatProcessor(int width, int height)`
Konstruktor-Methode: Erzeugt ein neues `FloatProcessor`-Objekt mit der Größe `width` × `height`.

`FloatProcessor(int width, int height, double[] pixels, ColorModel cm)`
Konstruktor-Methode: Erzeugt ein neues `FloatProcessor`-Objekt mit der Größe `width` × `height` aus einem eindimensionalen `double`-Array (mit Pixelwerten).

`FloatProcessor(int width, int height, float[] pixels, ColorModel cm)`
Konstruktor-Methode: Erzeugt ein neues `FloatProcessor`-Objekt mit der Größe `width` × `height` aus einem eindimensionalen `float`-Array (mit Pixelwerten) und dem Farbmodell `cm` vom Typ `java.awt.image. ColorModel`.

`FloatProcessor(int width, int height, int[] pixels)`
Konstruktor-Methode: Erzeugt ein neues `FloatProcessor`-Objekt mit der Größe `width` × `height` aus einem eindimensionalen `int`-Array (mit Pixelwerten).

C.4.5 `ShortProcessor` (Klasse)

`ShortProcessor(int width, int height)`
Konstruktor-Methode: Erzeugt ein neues `ShortProcessor`-Objekt mit der Größe `width` × `height`. Das Bild verwendet die Standard-Lookup-Tabelle für Grauwerte, die den Pixelwert 0 auf Schwarz abbildet.

`ShortProcessor(int width, int height, short[] pixels, ColorModel cm)`
Konstruktor-Methode: Erzeugt ein neues `ShortProcessor`-Objekt mit der Größe `width` × `height` aus einem eindimensionalen `short`-Array (mit Pixelwerten) und dem Farbmodell `cm` vom Typ `java.awt.image. ColorModel`.

C.5 Bildparameter

C.5.1 `ImageProcessor` (Klasse)

`int getHeight ()`
Liefert die Höhe (Anzahl der Zeilen) des Bilds.

`int getWidth ()`
Liefert die Breite (Anzahl der Spalten) des Bilds.

`java.awt.image.ColorModel getColorModel ()`
Liefert das Farbmodell dieses Bilds (z. B. `IndexColorModel` für Grauwert- und Indexfarbbilder, `DirectColorModel` für Vollfarbbilder).

C.6 Zugriff auf Pixel

C.6.1 `ImageProcessor` (Klasse)

Methoden zum Lesen von Pixelwerten

`int getPixel (int x, int y)`
Liefert den Wert des Pixels an der Position (`x`, `y`) bzw. den Wert 0 für alle Positionen außerhalb des Bildbereichs. Für Koordinaten außerhalb des Bildbereichs wird der Wert 0 retourniert (kein Fehler). Angewandt auf `ByteProcessor` oder `ShortProcessor` entspricht der Rückgabewert dem numerischen Pixelwert.

Für `ColorProcessor` sind die RGB-Farbwerte in der Standardform als `int` angeordnet. Für `FloatProcessor` enthält der 32-Bit-`int`-Rückgabewert das Bitmuster des entsprechenden `float`-Werts. Die Umwandlung in einen `float`-Wert erfolgt in diesem Fall mit der Methode `Float.intBitsToFloat()`.

`int[] getPixel (int x, int y, int[] iArray)`
Liefert den Wert des Pixels an der Position (`x`, `y`) als `int`-Array mit *einem* Element bzw. mit *drei* Elementen für `ColorProcessor` (RGB-Pixelwerte). Ist `iArray` ein entsprechendes Array (ungleich `null`), dann werden die Komponentenwerte darin abgelegt und `iArray` wird zurückgegeben. Ansonsten wird ein neues Array erzeugt.

`float getPixelValue (int x, int y)`
Liefert den Inhalt des Pixels an der Position (`x`, `y`) als `float`-Wert. Für Bilder vom Typ `ByteProcessor` und `ShortProcessor` wird ein kalibrierter Wert erzeugt, der durch die optionale Kalibrierungstabelle des Prozessors bestimmt wird. Für `FloatProcessor` wird der tatsächliche Pixelwert, für `ColorProzessor` der Luminanzwert des RGB-Pixels geliefert.

`double getInterpolatedPixel(double x, double y)`
Liefert den durch bilineare Interpolation geschätzten Wert an der (kontinuierlichen) Bildposition (`x`, `y`).

`Object getPixels()`
Liefert einen Verweis auf das Pixel-Array des `ImageProcessor`-Objekts (keine Kopie). Der Elementtyp des zugehörigen Arrays ist vom Typ des Prozessors abhängig:

`ByteProcessor` → `byte[]`

`ShortProcessor` → `short[]`

`FloatProcessor` → `float[]`

`ColorProcessor` → `int[]`

Der Rückgabewert dieser Methode ist allerdings vom generischen Typ `Object`, daher ist ein entsprechender Typecast erforderlich, z. B.

```
ByteProcessor ip = new ByteProcessor(200,300);
byte[] pixels = (byte[]) ip.getPixels();
```

`Object getPixelsCopy()`
Liefert einen Verweis auf das *Snapshot*-Array (UNDO-Kopie) des `ImageProcessor`-Objekts falls vorhanden, ansonsten eine neue Kopie des Bildinhalts als Pixel-Array. Das Ergebnis ist gleich wie bei `getPixels()` zu behandeln.

`void getRow(int x, int y, int[] data, int length)`
Liefert `length` Pixelwerte aus der Zeile `y`, beginnend an der Stelle (`x`, `y`) im Array `data`.

`void getColumn(int x, int y, int[] data, int length)`
Liefert `length` Pixelwerte aus der Spalte `x`, beginnend an der Stelle (`x`, `y`) im Array `data`.

`double[] getLine(double x1,double y1,double x2,double y2)`
Liefert ein eindimensionales Array von Pixelwerten entlang der Geraden zwischen dem Startpunkt (`x1`, `y1`) und dem Endpunkt (`x2`, `y2`).

Methoden zum Schreiben von Pixelwerten

`void putPixel(int x, int y, int value)`
Setzt den Wert des Pixels an der Position (`x`, `y`) auf `value`. Koordinaten außerhalb des Bildbereichs werden ignoriert (kein Fehler). Bei Bildern vom Typ `ByteProcessor` (8-Bit-Pixelwerte) und `ShortProcessor` (16-Bit-Pixelwerte) wird `value` durch Clamping auf den zulässigen Wertebereich beschränkt. Für `ColorProcessor` sind die RGB-Farbwerte in `value` in der Standardform angeordnet. Für `FloatProcessor` enthält `value` das Bitmuster des ent-

sprechenden `float`-Werts. Die Umwandlung aus einem `float`-Wert erfolgt in diesem Fall mit der Methode `Float.floatToIntBits()`.

`void putPixel (int x, int y, int[] iArray)`
Setzt den Wert des Pixels an der Position (`x`, `y`) auf den durch das Array `iArray` spezifizierten Wert. `iArray` besteht aus *einem* Element bzw. aus *drei* Elementen für `ColorProcessor` (RGB-Pixelwerte).

`void putPixelValue (int x, int y, double value)`
Setzt den Wert des Pixels an der Position (`x`, `y`) auf `value`.

`void setPixels (Object pixels)`
Ersetzt das bestehende Pixel-Array des Prozessors durch `pixels`. Typ und Größe des eindimensionalen Arrays `pixels` müssen der Spezifikation des Prozessors entsprechen (s. `getpixels()`). Das *Snapshot*-Array des Prozessors wird zurückgesetzt.

`void putRow (int x, int y, int[] data, int length)`
Ersetzt `length` Pixelwerte in Zeile `y`, beginnend an der Stelle (`x`, `y`) mit den Werten des Arrays `data`.

`void putColumn (int x, int y, int[] data, int length)`
Ersetzt `length` Pixelwerte in Spalte `x`, beginnend an der Stelle (`x`, `y`) mit den Werten des Arrays `data`.

`void insert (ImageProcessor ip, int xloc, int yloc)`
Setzt das Bild `ip` im Prozessorbild an der Position (`xloc`, `yloc`) ein.

Direkter Zugriff auf Pixel-Arrays

Die Verwendung von Methoden für den Zugriff auf Pixelwerte ist mit einem relativ hohen Zeitaufwand verbunden. Schneller ist der direkte Zugriff auf die Zellen des Pixel-Arrays des `ImageProcessor`-Objekts. Dabei ist zu beachten, dass Pixel-Arrays von Bildern in Java bzw. ImageJ *eindimensional* und zeilenweise angeordnet sind (Abb. B.1 (a)).

Eine Referenz auf das eindimensionale Pixel-Array *pixels* erhält man mithilfe der Methode `getPixels()`. Für jedes Pixel an der Position (u, v) muss der eindimensionale Index i innerhalb des Arrays berechnet werden, wobei die Breite w (Länge der Zeilen) des Bilds bekannt sein muss:

$$I(u, v) \equiv pixels[v \cdot w + u]$$

Das Beispiel in Prog. C.1 zeigt den direkten Pixel-Zugriff für ein Bild vom Typ `ByteProcessor` in der `run`-Methode eines ImageJ-Plugins. Die bitweise Maskierung `"0xFF & pixels[]"` (Zeile 8) bzw. `"0xFF & p"` (Zeile 10) ist notwendig, um den Bytewert des Pixels ohne Vorzeichen (im Bereich $0 \ldots 255$) zu erhalten (s. auch Abschn. B.1.3). Dasselbe gilt auch für 16-Bit-Bilder vom Typ `ShortProcessor`, wobei in diesem Fall eine Bitmaske mit dem Wert `0xFFFF` und der Typecast `(short)` zu verwenden ist.

Programm C.1
Direkter Pixel-Zugriff für ein Bild vom Typ `ByteProcessor`.

```
1  public void run (ImageProcessor ip) {
2    int w = ip.getWidth();
3    int h = ip.getHeight();
4    byte[] pixels = (byte[]) ip.getPixels();
5
6    for (int v = 0; v < h; v++) {
7      for (int u = 0; u < w; u++) {
8        int p = 0xFF & pixels[v * w + u];
9        p = p + 1;
10       pixels[v * w + u] = (byte) (0xFF & p);
11     }
12   }
13 }
```

Falls (wie in obigem Beispiel) die Koordinatenwerte (u, v) für die Berechnung nicht benötigt werden und die Reihenfolge des Zugriffs auf die Pixelwerte irrelevant ist, kann man natürlich auch mit nur einer Schleife über alle Elemente des eindimensionalen Pixel-Arrays (der Länge $w \cdot h$) iterieren. Diese Möglichkeit wird beispielsweise in Prog. 12.1 (S. 240) für die Iteration über alle Pixelwerte eines Farbbilds genutzt.

C.7 Konvertieren von Bildern

C.7.1 `ImageProcessor` (Klasse)

Die Klasse `ImageProcessor` stellt folgende Methoden zur Konvertierung zwischen Prozessor-Objekten zur Verfügung, die jeweils eine *Kopie* des bestehenden Prozessors erzeugen, der selbst unverändert bleibt. Falls der bestehende Prozessor bereits vom gewünschten Zieltyp ist, wird nur eine Kopie angelegt.

`ImageProcessor convertToByte (boolean doScaling)`
: Kopiert den Inhalt des bestehenden Prozessors (`this`) in ein neues Objekt vom Typ `ByteProcessor`.

`ImageProcessor convertToShort (boolean doScaling)`
: Kopiert den Inhalt des bestehenden Prozessors (`this`) in ein neues Objekt vom Typ `ShortProcessor`.

`ImageProcessor convertToFloat ()`
: Kopiert den Inhalt des bestehenden Prozessors (`this`) in ein neues Objekt vom Typ `FloatProcessor`.

`ImageProcessor convertToRGB ()`
: Kopiert den Inhalt des bestehenden Prozessors (`this`) in ein neues Objekt vom Typ `RGBProcessor`.

C.7.2 `ImagePlus`, `ImageConverter` (Klassen)

Zur Konvertierung von Bildern der Klasse `ImagePlus` ist die Klasse `ImageConverter` vorgesehen. Um ein `ImagePlus`-Bild `imp` zu konvertieren, erzeugt man zunächst ein Objekt der Klasse `ImageConverter` durch

```
ImageConverter iConv = new ImageConverter(imp);
```

Auf das `ImageConverter`-Objekt `iConv` können folgende Methoden angewandt werden:

`void convertToGray8 ()`
: Konvertiert das Bild in ein 8-Bit-Grauwertbild.

`void convertToGray16 ()`
: Konvertiert das Bild in ein 16-Bit-Grauwertbild.

`void convertToGray32 ()`
: Konvertiert das Bild in ein 32-Bit-Grauwertbild.

`void convertToRGB ()`
: Konvertiert das Bild in ein RGB-Farbbild.

`void convertToHSB ()`
: Konvertiert das bestehende RGB-Bild in einen HSV-Image-Stack.[9]

`void convertHSBToRGB ()`
: Konvertiert einen HSB-Image-Stack nach RGB.

`void convertRGBStackToRGB ()`
: Konvertiert einen 8-Bit-Image-Stack in ein RGB-Bild.

`void convertToRGBStack ()`
: Konvertiert das bestehende RGB-Farbbild in einen Image-Stack, bestehend aus 3 Einzelbildern.

`void convertRGBtoIndexedColor (int nColors)`
: Konvertiert ein RGB-Bild in ein Indexfarbbild mit `nColors` Farben.

`void setDoScaling (boolean scaleConversions)`
: Wenn `scaleConversions` = *true*, dann werden erzeugte 8-Bit-Bilder auf 0 . . . 255 skaliert und 16-Bit-Bilder auf 0 . . . 65535. Ansonsten erfolgt keine Skalierung.

C.8 Histogramme und Bildstatistiken

C.8.1 `ImageProcessor` (Klasse)

`int[] getHistogram ()`
: Berechnet das Histogramm des gesamten Bilds bzw. der ausgewählten *Region of Interest* (ROI).

[9] HSB ist identisch zum HSV-Farbraum (s. Abschn. 12.2.3).

Weitere Bildstatistiken können über die Klasse `ImageStatistics` und die daraus abgeleiteten Klassen `ByteStatistics`, `ShortStatistics` etc. berechnet werden.

C.9 Punktoperationen

C.9.1 `ImageProcessor` (Klasse)

Die nachfolgenden Methoden für die Klasse `ImageProcessor` dienen zur arithmetischen oder bitweisen Verknüpfungen eines Bilds mit einem *skalaren* Wert. Die Operationen werden jeweils auf alle Pixel des Bilds bzw. auf alle Pixel innerhalb der *Region of Interest* (ROI) angewandt.

`void add (int value)`
: Addiert `value` zu jedem Pixel.

`void add (double value)`
: Addiert `value` zu jedem Pixel.

`void and (int value)`
: Binäre AND-Operation der Pixelwerte mit `value`.

`void applyTable (int[] lut)`
: Anwendung der Abbildung Lookup-Table `lut` auf alle Pixel des Bilds bzw. innerhalb der ausgewählten ROI.

`void autoThreshold ()`
: Schwellwertoperation mit einem automatisch aus dem Histogramm bestimmten Schwellwert p_{th}.

`void gamma (double value)`
: Gammakorrektur mit dem Gammawert `value`.

`void log ()`
: Logarithmus (Basis 10).

`void max (double value)`
: Pixel mit Wert größer als `value` werden auf den Wert `value` gesetzt.

`void min (double value)`
: Pixel mit Wert kleiner als `value` werden auf den Wert `value` gesetzt.

`void multiply (double value)`
: Pixel werden mit `value` multipliziert.

`void noise (double range)`
: Zu jedem Pixel wird ein normalverteilter Zufallswert im Bereich $\pm$`range` addiert.

`void or (int value)`
: Binäre OR-Operation der Pixelwerte mit `value`.

`void sqr ()`
: Pixel werden durch den Wert ihres Quadrats ersetzt.

`void sqrt ()`
Pixel werden durch den Wert ihrer Quadratwurzel ersetzt.

`void threshold (int level)`
Schwellwertoperation mit $p_{\text{th}} =$`level`, Ergebnis ist 0 oder 255.

`void xor (int value)`
Binäre EXCLUSIVE-OR-Operation der Pixelwerte mit `value`.

Die Klasse `ImageProcessor` stellt folgende Methode zur Verknüpfung von *zwei* Bildern zur Verfügung:

`void copyBits (ImageProcessor src, int x, int y, int mode)`
Kopiert das Bild `src` in das aktuelle Bild (`this`) an die Position (`x`, `y`) mit dem Kopiermodus `mode`. Konstanten für `mode` sind in `Blitter` (s. unten) definiert. Für ein Beispiel s. auch Abschn. 5.7.3.

C.9.2 `Blitter` (Interface)

Folgende `mode`-Werte für die Methode `copyBits()` sind als Konstanten in `Blitter` definiert (A bezeichnet das Zielbild, B das Quellbild):

`ADD` (Konstante)
$A(u,v) \leftarrow A(u,v) + B(u,v)$

`AND` (Konstante)
$A(u,v) \leftarrow A(u,v) \wedge B(u,v)$ (bitweise UND-Operation)

`AVERAGE` (Konstante)
$A(u,v) \leftarrow (A(u,v) + B(u,v))/2$

`COPY` (Konstante)
$A(u,v) \leftarrow B(u,v)$

`COPY_INVERTED` (Konstante)
$A(u,v) \leftarrow 255 - B(u,v)$ (nur für 8-Bit-Grauwert- und RGB-Bilder)

`DIFFERENCE` (Konstante)
$A(u,v) \leftarrow |A(u,v) - B(u,v)|$

`DIVIDE` (Konstante)
$A(u,v) \leftarrow A(u,v)/B(u,v)$

`MAX` (Konstante)
$A(u,v) \leftarrow \max(A(u,v), B(u,v))$

`MIN` (Konstante)
$A(u,v) \leftarrow \min(A(u,v), B(u,v))$

`MULTIPLY` (Konstante)
$A(u,v) \leftarrow A(u,v) \cdot B(u,v)$

`OR` (Konstante)
$A(u,v) \leftarrow A(u,v) \vee B(u,v)$ (bitweise OR-Operation)

`SUBTRACT` (Konstante)
$A(u,v) \leftarrow A(u,v) - B(u,v)$

`XOR` (Konstante)
$A(u,v) \leftarrow A(u,v)$ xor $B(u,v)$ (bitweise XOR-Operation)

C.10 Filter

C.10.1 `ImageProcessor` (Klasse)

`void convolve (float[] kernel, int kernelWidth, int kernelHeight)`
Lineare Faltung mit dem angegebenen Filterkern beliebiger Größe.

`void convolve3x3 (int[] kernel)`
Lineare Faltung mit einem beliebigen Filterkern der Größe 3×3.

`void dilate ()`
Dilation durch ein 3×3-Minimum-Filter.

`void erode ()`
Erosion durch ein 3×3-Maximum-Filter.

`void findEdges ()`
3×3-Kantenfilter (Sobel-Operator).

`void medianFilter ()`
3×3-Medianfilter.

`void smooth ()`
3×3-Boxfilter (Glättungsfilter).

`void sharpen ()`
Schärft das Bild mit einem einfachen 3×3-Laplace-Filter.

C.11 Geometrische Operationen

C.11.1 `ImageProcessor` (Klasse)

`ImageProcessor crop ()`
Erzeugt ein neues `ImageProcessor`-Objekt aus dem Inhalt der aktuellen *Region of Interest* (ROI).

`ImageProcessor flipHorizontal ()`
Spiegelt das Bild in horizontaler Richtung.

`ImageProcessor flipVertical ()`
Spiegelt das Bild in vertikaler Richtung.

`ImageProcessor resize (int dstWidth, int dstHeight)`
Erzeugt ein neues `ImageProcessor`-Objekt, das auf die angegebene Größe skaliert ist.

`ImageProcessor rotateLeft ()`
Erzeugt ein neues, um 90° im Uhrzeigersinn gedrehtes Bild.

`ImageProcessor rotateRight ()`
Erzeugt ein neues, um 90° gegen den Uhrzeigersinn gedrehtes Bild.

`void scale (double xScale, double yScale)`
Skaliert das Bild in x- und y-Richtung mit den angegebenen Faktoren.

`void setInterpolate (boolean doInterpolate)`
Wenn `doInterpolate`=*true*, dann wird bei den geometrischen Operationen `scale()`, `resize()` und `rotate()` die bilineare Interpolation verwendet, ansonsten die Nearest-Neighbor-Interpolation.

C.12 Grafische Operationen in Bildern

C.12.1 `ImageProcessor` (Klasse)

`void drawDot (int xcenter, int ycenter)`
Zeichnet einen Punkt mit der aktuellen Strichbreite und dem aktuellen Farbwert.

`void drawLine (int x1, int y1, int x2, int y2)`
Zeichnet eine Gerade von (`x1`, `y1`) nach (`x2`, `y2`).

`void drawPixel (int x, int y)`
Setzt das Pixel an der Position (`x`, `y`) auf den aktuellen Farbwert.

`void drawRect (int x, int y, int width, int height)`
Zeichnet ein achsenparalleles Rechteck an der Position (`x`, `y`) mit der Breite `width` und der Höhe `height`.

`void drawString (String s)`
Fügt den Text `s` an der aktuellen Position ein.

`void drawString (String s, int x, int y)`
Fügt den Text `s` an der Position (`x`, `y`) ein.

`void fill ()`
Füllt das gesamte Bild bzw. die *Region of Interest* (ROI) mit dem aktuellen Farbwert.

`void fill (int[] mask)`
Füllt alle Pixel innerhalb der *Region of Interest* (ROI), wenn die zugehörige Position im Array `mask` den Wert `ImageProcessor.BLACK` enthält. Das Array `mask` muss exakt gleich groß sein wie die ROI.

`void getStringWidth (String s)`
Liefert die Breite des Texts `s` in Pixel.

`void insert (ImageProcessor src, int xloc, int yloc)`
Setzt den Inhalt des Bilds `src` im aktuellen Bild (`this`) an der Position (`xloc`, `yloc`) ein.

`void lineTo (int x2, int y2)`
Zeichnet eine Gerade von der aktuellen Position nach (`x2`, `y2`). Die aktuelle Position wird danach auf (`x2`, `y2`) gesetzt.

`void moveTo (int x, int y)`
Die aktuelle Position wird auf (`x`, `y`) gesetzt.

`void setAntialiasedText (boolean antialiasedText)`
Spezifiziert, ob beim Rendern von Text Anti-Aliasing verwendet wird oder nicht.

`void setClipRect (Rectangle clipRect)`
Setzt den Zeichenbereich (*clipping rectangle*) für die Methoden `lineTo()`, `drawLine()`, `drawDot()` und `drawPixel()`.

`void setColor (java.awt.Color color)`
Spezifiziert den Farbwert für nachfolgende Zeichenoperationen. Dabei wird (abhängig vom Bildtyp) der dem angegebenen Farbwert ähnlichste Pixelwert gesucht.

`void setFont (java.awt.Font font)`
Das `Font`-Objekt `font` spezifiziert die Schriftart für die Methode `drawString()`.

`void setJustification (int justification)`
Spezifiziert die Art der Textausrichtung für die Methode `drawString()`. Zulässige Werte für `justification` sind `CENTER_JUSTIFY`, `RIGHT_JUSTIFY` und `LEFT_JUSTIFY` (Konstanten in der Klasse `ImageProcessor`).

`void setLineWidth (int width)`
Spezifiziert die Strichbreite für die Methoden `lineTo()` und `drawDot()`.

`void setValue (double value)`
Spezifiziert den Farbwert für nachfolgende Zeichenoperationen. Der `double`-Wert `value` wird je nach Bildtyp unterschiedlich interpretiert.

C.13 Bilder darstellen

C.13.1 `ImagePlus` (Klasse)

`String getShortTitle ()`
Liefert den Titeltext in verkürzter Form für dieses Bild.

`String getTitle ()`
Liefert den Titeltext für dieses Bild.

`void hide ()`
Schließt das Fenster für dieses `ImagePlus`-Bild, sofern eines vorhanden ist.

`void show ()`
Öffnet ein Fenster zur Anzeige dieses `ImagePlus`-Bilds und löscht die Statusanzeige des ImageJ-Hauptfensters.

`void show (String statusMessage)`
Öffnet ein neues Fenster zur Anzeige dieses `ImagePlus`-Bilds und

zeigt den Text `statusMessage` in der Statusanzeige des ImageJ-Hauptfensters.

`void setTitle (String title)`
Ersetzt den Titeltext für dieses Bild.

`void updateAndDraw ()`
Aktualisiert dieses Bild aus den Pixeldaten des zugehörigen `ImageProcessor`-Objekts und erneuert die Anzeige des Bildinhalts.

`void updateAndRepaintWindow ()`
Verwendet `updateAndDraw()` und zeichnet anschließend das gesamte Bildschirmfenster neu, um auch Informationen außerhalb des Bildbereichs (wie Dimension, Typ und Größe) zu aktualisieren.

C.14 Operationen auf Bildfolgen (Stacks)

C.14.1 `ImagePlus` (Klasse)

`ImageStack getStack ()`
Erzeugt ein `ImageStack`-Objekt für dieses Bild.

`int getStackSize ()`
Liefert die Anzahl der Bilder (Slices) des Stacks oder 1, wenn es sich um ein Einzelbild handelt.

C.14.2 `ImageStack` (Klasse)

Zur Erzeugung von Stacks siehe die Konstruktor-Methoden in Abschn. C.3.2.

`void addSlice (String sliceLabel, ImageProcessor ip)`
Fügt das Bild `ip` mit dem Bezeichnungstext `sliceLabel` am Ende dieses Stacks ein.

`void addSlice (String sliceLabel,ImageProcessor ip,int n)`
Fügt das Bild `ip` mit dem Bezeichnungstext `sliceLabel` nach dem `n`-ten Bild ein bzw. am Anfang des Stacks, wenn `n`=0.

`void addSlice (String sliceLabel, Object pixels)`
Fügt das als Pixel-Array `pixels` übergebene Bild am Ende dieses Stacks ein.

`void deleteLastSlice ()`
Löscht das letzte Bild des Stacks.

`void deleteSlice (int n)`
Löscht das `n`-te Bild des Stacks, mit $1 \leq$ `n` $\leq$ `getsize()`.

`int getHeight ()`
Liefert die Höhe der Bilder im Stack.

`Object[] getImageArray ()`
Erzeugt ein eindimensionales Array mit den Bildern des Stacks.

`Object getPixels (int n)`
Liefert das (eindimensionale) Pixel-Array des `n`-ten Bilds im Stack, mit $1 \le$ `n` $\le$ `getsize()`.

`ImageProcessor getProcessor (int n)`
Liefert das `ImageProcessor`-Objekt des `n`-ten Bilds im Stack, mit $1 \le$ `n` $\le$ `getsize()`.

`int getSize ()`
Liefert die Anzahl der Bilder (Slices) im Stack.

`String getSliceLabel (int n)`
Liefert den Bezeichnungstext des `n`-ten Bilds im Stack mit $1 \le$ `n` $\le$ `getsize()`.

`int getWidth ()`
Liefert die Breite der Bilder im Stack.

`void setPixels (Object pixels, int n)`
Ersetzt das Pixel-Array des `n`-ten Bilds im Stack, mit $1 \le$ `n` $\le$ `getsize()`.

`void setSliceLabel (String label, int n)`
Ersetzt den Bezeichnungstext des `n`-ten Bilds im Stack, mit $1 \le$ `n` $\le$ `getsize()`.

C.14.3 Stack-Beispiel

Prog. C.2–C.3 zeigt ein Beispiel für den Umgang mit Image-Stacks, in dem ein Bild durch *Alpha Blending* in ein zweites Bild überblendet wird (analog zu Prog. 5.5–5.6 in Abschn. 5.7.4).

Das Hintergrundbild (`bgIp`) ist das aktuelle Bild, das bei der Ausführung des Plugin an die `run()`-Methode übergeben wird. Das Vordergrundbild wird über eine Dialogbox (`GenericDialog`) ausgewählt, ebenso die Länge (Anzahl der *Slices*) der zu erzeugenden Bildfolge (Prog. C.2).

In Prog. C.3 wird zunächst mit `NewImage.createByteImage()` ein Stack mit der erforderlichen Zahl von Bildern erzeugt. Anschließend wird in einer Schleife für jedes Bild im Stack der Transparenzwert α (s. Gl. 5.41) berechnet und das zugehörige Bild durch eine gewichtete Summe der beiden Ausgangsbilder ersetzt. Man beachte, dass in einer Folge von N Bildern – im Unterschied zur sonst üblichen Nummerierung – der Frame-Index von $1 \ldots N$ läuft (`getProcessor()` in Zeile 70). Das entspechende Ergebnis und die Dialogbox sind in Abb. C.6 dargestellt.

Programm C.2
Alpha Blending Stack (Teil 1).

```
1  import ij.IJ;
2  import ij.ImagePlus;
3  import ij.ImageStack;
4  import ij.WindowManager;
5  import ij.gui.*;
6  import ij.plugin.filter.PlugInFilter;
7  import ij.process.*;
8
9  public class AlphaBlendStack_ implements PlugInFilter {
10   static int nFrames = 10;
11   ImagePlus fgIm;  // fgIm = foreground image
12
13   public int setup(String arg, ImagePlus imp) {
14     return DOES_8G;}
15
16   boolean runDialog() {
17     // get list of open images
18     int[] windowList = WindowManager.getIDList();
19     if(windowList==null){
20       IJ.noImage();
21       return false;
22     }
23     String[] windowTitles = new String[windowList.length];
24     for (int i = 0; i < windowList.length; i++) {
25       ImagePlus imp = WindowManager.getImage(windowList[i]);
26       if (imp != null)
27         windowTitles[i] = imp.getShortTitle();
28       else
29         windowTitles[i] = "untitled";
30     }
31     GenericDialog gd = new GenericDialog("Alpha Blending");
32     gd.addChoice("Foreground image:",
33         windowTitles, windowTitles[0]);
34     gd.addNumericField("Frames:", nFrames, 0);
35     gd.showDialog();
36     if (gd.wasCanceled())
37       return false;
38     else {
39       int img2Index = gd.getNextChoiceIndex();
40       fgIm = WindowManager.getImage(windowList[img2Index]);
41       nFrames = (int) gd.getNextNumber();
42       if (nFrames < 2)
43         nFrames = 2;
44       return true;
45     }
46   } // continued...
```

Programm C.3
Alpha Blending (Teil 2).

```
47  // class AlphaBlendStack_ (continued)
48
49  public void run(ImageProcessor bgIp) {
50    // bgIp = background image
51
52    if(runDialog()) { //open dialog box (returns false if cancelled)
53      int w = bgIp.getWidth();
54      int h = bgIp.getHeight();
55
56      // prepare foreground image
57      ImageProcessor fgIp =
58          fgIm.getProcessor().convertToByte(false);
59      ImageProcessor fgTmpIp = bgIp.duplicate();
60
61      // create image stack
62      ImagePlus movie =
63        NewImage.createByteImage("Movie",w,h,nFrames,0);
64      ImageStack stack = movie.getStack();
65
66      // loop over stack frames
67      for (int i=0; i<nFrames; i++) {
68        // transparency of foreground image
69        double iAlpha = 1.0 - (double)i/(nFrames-1);
70        ImageProcessor iFrame = stack.getProcessor(i+1);
71
72        // copy background image to frame i
73        iFrame.insert(bgIp,0,0);
74        iFrame.multiply(iAlpha);
75
76        // copy foreground image and make transparent
77        fgTmpIp.insert(fgIp,0,0);
78        fgTmpIp.multiply(1-iAlpha);
79
80        // add foreground image frame i
81        ByteBlitter blitter =
82            new ByteBlitter((ByteProcessor)iFrame);
83        blitter.copyBits(fgTmpIp,0,0,Blitter.ADD);
84      }
85
86      // display movie (image stack)
87      movie.show();
88    }
89  }
90
91 } // end of class AlphaBlendStack_
```

Abbildung C.6
Alpha Blending in eine Bildfolge (Ergebnis zu Prog. C.2–C.3). Ausgangsbilder: Hintergrundbild (a) und Vordergrundbild (b). Anzeige des erzeugten Stacks (horizontaler „Slider“ am unteren Rand des Fensters) in zwei verschiedenen Positionen für Frame 3 (c) und Frame 8 (d). Dialogfenster zur Auswahl des Vordergrundbilds und der Stackgröße (e).



(a)



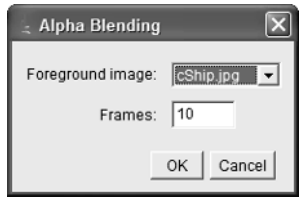
(b)



(c)



(d)



(e)

C.15 *Region of Interest* (ROI)

Die *Region of Interest* dient zur Selektion eines Bildbereichs für die nachfolgende Bearbeitung. Sie wird üblicherweise interaktiv durch den Benutzer spezifiziert. ImageJ unterstützt mehrere Formen von ROIs:

- Rechteckige ROIs (Klasse `Roi`)
- Elliptische ROIs (Klasse `OvalRoi`)
- Geradenförmige ROIs (Klasse `Line`)
- Polygonale ROIs (Klasse `PolygonRoi` und Subklasse `FreehandRoi`)
- Text-ROIs (Klasse `TextRoi`)

Die zugehörigen Klassen sind im Package `ij.gui` definiert. ROI-Objekte dieser Form sind nur über Objekte der Klasse `ImagePlus` zugänglich (s. Abschn. C.15.3).

C.15.1 `ImageProcessor` (Klasse)

Bei der Bearbeitung von Bildern der Klasse `ImageProcessor` manifestiert sich die ROI nur durch ihr begrenzendes Rechteck (Bounding Box), im Fall einer nichtrechteckigen ROI durch eine zusätzliche Bitmaske (eindimensionales `int`-Array) in der Größe des ROI-Rechtecks.

`Rectangle getRoi ()`
: Liefert das Rechteck (vom Typ `java.awt.Rectangle`) der aktuellen *Region of Interest* (ROI) dieses Bilds.

`void setRoi (Rectangle roi)`
: Ersetzt die *Region of Interest* (ROI) dieses Bilds mit dem angegebenen Rechteck und löscht die zugehörige Bitmaske (`mask`), falls die Größe von `roi` sich gegenüber der vorherigen ROI ändert.

`void setRoi (int x, int y, int rwidth, int rheight)`
: Ersetzt die *Region of Interest* (ROI) dieses Bilds mit dem angegebenen Rechteck und löscht die zugehörige Bitmaske (`mask`), falls die Größe von `roi` sich gegenüber der vorherigen ROI ändert.

`int[] getMask ()`
: Liefert die Bitmaske einer nichtrechteckigen ROI bzw. `null`, wenn die ROI rechteckig ist.

`void setMask (int[] mask)`
: Ersetzt die Bitmaske zur Spezifikation einer nichtrechteckigen ROI. Die Anzahl der Elemente in `mask` muss der Größe des ROI-Rechtecks entsprechen.

C.15.2 `ImageStack` (Klasse)

`Rectangle getRoi ()`
: Liefert das Rechteck (vom Typ `java.awt.Rectangle`) der aktuellen *Region of Interest* (ROI) des Stacks.

`void setRoi (Rectangle roi)`
Setzt die *Region of Interest* (ROI) für den gesamten Stack. `roi` ist vom Typ `java.awt.Rectangle`, das beispielsweise durch
`new Rectangle(x, y, rwidth, rheight)`
erzeugt werden kann.

C.15.3 `ImagePlus` (Klasse)

`Roi getRoi ()`
Liefert das ROI-Objekt (vom Typ `ij.gui.Roi` bzw. einer der Subklassen `Line`, `OvalRoi`, `PolygonRoi`, `TextRoi`) der aktuellen *Region of Interest* (ROI) für dieses Bild.

`void killRoi ()`
Löscht die aktuelle *Region of Interest* (ROI).

`void setRoi (Rectangle roi)`
Ersetzt die *Region of Interest* (ROI) dieses Bilds mit dem angegebenen Rechteck.

`void setRoi (int x, int y, int rwidth, int rheight)`
Ersetzt die *Region of Interest* (ROI) dieses Bilds mit dem angegebenen Rechteck.

`void setRoi (Roi roi)`
Ersetzt die *Region of Interest* (ROI) mit dem angegebenen Objekt der Klasse `Roi` (bzw. einer Subklasse).

`int[] getMask ()`
Liefert die Bitmaske einer nichtrechteckigen ROI bzw. `null`, wenn die ROI rechteckig ist.

C.15.4 `Roi`, `Line`, `OvalRoi`, `PolygonRoi` (Klassen)

`Roi (int x, int y, int width, int height)`
Konstruktor-Methode: Erzeugt ein `Roi`-Objekt für eine rechteckige *Region of Interest*.

`Line (int x1, int y1, int x2, int y2)`
Konstruktor-Methode: Erzeugt ein `Line`-Objekt für eine geradenförmige *Region of Interest*.

`OvalRoi (int x, int y, int width, int height)`
Konstruktor-Methode: Erzeugt ein `OvalRoi`-Objekt für eine ellipsenförmige *Region of Interest*.

`PolygonRoi (int[] xPnts, int[] yPnts, int nPnts, int type)`
Konstruktor-Methode: Erzeugt aus den Koordinatenwerten `xPnts` und `yPnts` ein `PolygonRoi`-Objekt für eine polygonale *Region of Interest* (zulässige Werte für `type` sind `Roi.POLYGON`, `Roi.FREEROI`, `Roi. TRACED_ROI`, `Roi.POLYLINE`, `Roi.FREELINE` und `Roi.ANGLE`).

`boolean contains (int x, int y)`
Liefert *true*, wenn (`x`, `y`) innerhalb dieser ROI liegt.

C.16 *Image Properties*

Manchmal ist es notwendig, die Ergebnisse eines Plugins an ein weiteres Plugin zu übergeben. Die `run()`-Methode eines ImageJ-Plugins sieht jedoch keinen Rückgabewert vor. Eine Möglichkeit besteht darin, Ergebnisse aus einem Plugin als *property* im zugehörigen Bild abzulegen. Properties sind paarweise Einträge eines Schlüssels (*key*) und eines zugehörigen Werts (*value*), der ein beliebiges Java-Objekt sein kann. ImageJ unterstützt diesen Mechanismus, der auf Basis einer Hash-Tabelle implementiert ist, mit folgenden Methoden:

C.16.1 `ImagePlus` (Klasse)

`java.util.Properties getProperties ()`
Liefert das `Properties`-Objekt (eine Hash-Tabelle) mit allen Property-Einträgen für dieses Bild oder `null`.

`Object getProperty (String key)`
Liefert die zum Schlüssel `key` gehörige Property dieses Bilds bzw. `null`, wenn diese nicht definiert ist.

`void setProperty (String key, Object value)`
Trägt das Paar (`key`, `value`) in die Property-Tabelle dieses Bilds ein. Falls bereits eine Property für `key` definiert war, wird diese durch `value` ersetzt.

Beispiel

Prog. C.4 zeigt ein einfaches Beispiel zur Verwendung von Properties, bestehend aus zwei getrennten ImageJ-Plugins. Im ersten Plugin (`Plugin1_`) wird das Histogramm des Bilds berechnet und das Ergebnis als Property mit dem Schlüssel `"Plugin1"` eingefügt (Zeile 16). Das zweite Plugin (`Plugin2_`) holt das Ergebnis des Histogramms aus den Properties des übergebenen Bilds (Zeile 33) und könnte es anschließend weiter verarbeiten. Der dafür erforderliche Schlüssel wird hier über die statischen Variable `KEY` der Klasse `Plugin1_` ermittelt (Zeile 32).

C.17 Interaktion

C.17.1 `IJ` (Klasse)

`static void beep ()`
Erzeugt ein Tonsignal.

Programm C.4
Beispiel zur Verwendung von *Image Properties*. Im ersten Plugin (`Plugin1_`) wird in der `run()`-Methode das Histogramm berechnet und als Property an das zweite Plugin (`Plugin2_`) übergeben.

File `Plugin1_.java`:

```
1 import ij.ImagePlus;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.ImageProcessor;
4
5 public class Plugin1_ implements PlugInFilter {
6   ImagePlus imp;
7   public static final String KEY = "Plugin1";
8
9   public int setup(String arg, ImagePlus imp) {
10    this.imp = imp;
11    return DOES_ALL + NO_CHANGES;}
12
13  public void run(ImageProcessor ip) {
14    int[] hist = ip.getHistogram();
15    // add histogram to image properties:
16    imp.setProperty(KEY,hist);
17  }
18 }
```

File `Plugin2_.java`:

```
19 import ij.IJ;
20 import ij.ImagePlus;
21 import ij.plugin.filter.PlugInFilter;
22 import ij.process.ImageProcessor;
23
24 public class Plugin2_ implements PlugInFilter {
25   ImagePlus imp;
26
27   public int setup(String arg, ImagePlus imp) {
28     this.imp = imp;
29     return DOES_ALL;}
30
31   public void run(ImageProcessor ip) {
32     String key = Plugin1_.KEY;
33     int[] hist = (int[]) imp.getProperty(key);
34     if (hist == null){
35       IJ.error("This image has no histogram");
36     }
37     else {
38       // process histogram ...
39     }
40   }
41 }
```

`static void error (String s)`
Zeigt die Fehlermeldung `s` in einer Dialogbox mit dem Titel „ImageJ“.

`static ImagePlus getImage ()`
Liefert das aktuelle (vom Benutzer ausgewählte) Bild vom Typ `ImagePlus`.

`static double getNumber (String prompt,double defaultVal)`
Ermöglicht die Eingabe eines numerischen Werts durch den Benutzer.

`static String getString (String prompt, String defaultStr)`
Ermöglicht die Eingabe einer Textzeile durch den Benutzer.

`static void log (String s)`
Schreibt den Text `s` in das „Log“-Fenster von ImageJ.

`void showMessage (String msg)`
Zeigt den Text `msg` in einer Dialogbox mit dem Titel „Message“.

`void showMessage (String title, String msg)`
Zeigt den Text `msg` in einer Dialogbox mit dem Titel `title`.

`boolean showMessageWithCancel (String title, String msg)`
Zeigt den Text `msg` in einer Dialogbox mit dem Titel `title` mit der Möglichkeit zum Abbruch des Vorgangs.

`void showStatus (String s)`
Zeigt den Text `s` im Statusbalken von ImageJ.

`static void write (String s)`
Schreibt den Text `s` auf ein Konsolenfenster.

C.17.2 `ImageProcessor` (Klasse)

`void showProgress (double percentDone)`
Setzt die Balkenanzeige für den Bearbeitungsfortschritt auf den Wert `percentDone`.

`void hideProgress ()`
Blendet die Balkenanzeige für den Bearbeitungsfortschritt aus.

C.17.3 `GenericDialog` (Klasse)

Die Klasse `GenericDialog` bietet eine einfache Möglichkeit zur Erstellung von Dialogfenstern mit mehreren Feldern unterschiedlichen Typs. Das Layout des Dialogfensters wird automatisch erstellt. Ein Anwendungsbeispiel und das zugehörige Ergebnis ist in Prog. C.5 gezeigt (s. auch Abschn. 5.7.4 und C.14.3), weitere Details finden sich in der ImageJ-Online-Dokumentation und in [3].

Programm C.5
Beispiel für die Verwendung der Klasse `GenericDialog` und zugehöriges Dialogfenster.




```
1 import ij.ImagePlus;
2 import ij.gui.GenericDialog;
3 import ij.gui.NewImage;
4 import ij.plugin.PlugIn;
5
6 public class GenericDialogExample implements PlugIn {
7   static String title = "New Image";
8   static int width = 512;
9   static int height = 512;
10
11   public void run(String arg) {
12     GenericDialog gd = new GenericDialog("New Image");
13     gd.addStringField("Title:", title);
14     gd.addNumericField("Width:", width, 0);
15     gd.addNumericField("Height:", height, 0);
16     gd.showDialog();
17     if (gd.wasCanceled())
18       return;
19     title = gd.getNextString();
20     width = (int) gd.getNextNumber();
21     height = (int) gd.getNextNumber();
22
23     ImagePlus imp = NewImage.createByteImage(
24       title, width, height, 1, NewImage.FILL_WHITE);
25     imp.show();
26   }
27 }
```

C.18 Plugins

ImageJ-Plugins gibt es in zwei unterschiedlichen Formen, die jeweils als Java-Interface implementiert sind:

- `PlugIn`: arbeitet unabhängig von bestehenden Bildern.
- `PlugInFilter`: wird auf ein bestehendes Bild angewandt.

C.18.1 `PlugIn` (Interface)

Das `PlugIn`-Interface schreibt nur die Implementierung der `run`-Methode vor:

`void run (String arg)`
: Startet das Plugin. Das Argument `arg` kann auch eine leere Zeichenkette sein.

C.18.2 `PlugInFilter` (Interface)

Das `PlugInFilter`-Interface schreibt die Implementierung folgender Methoden vor:

`void run (ImageProcessor ip)`
: Startet das Plugin. Das übergebene `ImageProcessor`-Objekt `ip` ist das aktuelle Ausgangsbild.

`int setup (String arg, ImagePlus imp)`
: Wird bei der Ausführung eines Plugins durch ImageJ *vor* der `run()`-Methode aufgerufen. Das übergebene `ImagePlus`-Objekt `imp` ist das aktuelle Ausgangsbild (nicht der Bildprozessor). Falls das aktuelle `ImagePlus`-Objekt in der nachfolgend ausgeführten `run`-Methode benötigt wird, kann man es in der `setup()`-Methode an eine statische Variable der Plugin-Klasse binden oder mit `IJ.getImage()` ermitteln. Rückgabewert der `setup()`-Methode ist ein kodiertes `int`-Bitmuster, das die Möglichkeiten des Plugins beschreibt und aus einer Kombination der untenstehenden Konstanten gebildet wird. Ist der Rückgabewert `DONE`, dann wird die `run()`-Methode des Plugin nicht ausgeführt.

Konstanten für Rückgabewerte der `setup()`-Methode der Klasse `PlugInFilter` (alle vom Typ `int`):

`DOES_8G` (Konstante)
: Das Plugin akzeptiert 8-Bit Grauwertbilder.

`DOES_8C` (Konstante)
: Das Plugin akzeptiert 8-Bit Indexfarbbilder.

`DOES_16` (Konstante)
: Das Plugin akzeptiert 16-Bit Grauwertbilder.

`DOES_32` (Konstante)
: Das Plugin akzeptiert 32-Bit `float`-Bilder.

`DOES_RGB` (Konstante)
: Das Plugin akzeptiert 3×8-Bit Vollfarbbilder.

`DOES_ALL` (Konstante)
: Das Plugin akzeptiert alle Arten von ImageJ-Bildern.

`DOES_STACKS` (Konstante)
: Die `run`-Methode des Plugin soll für alle Bilder eines Stacks ausgeführt werden.

`DONE` (Konstante)
: Die `run`-Methode des Plugin soll nicht ausgeführt werden.

`NO_CHANGES` (Konstante)
: Das Plugin modifiziert die Pixeldaten des übergebenen Bilds nicht.

`NO_IMAGE_REQUIRED` (Konstante)
: Das Plugin benötigt kein Bild zur Durchführung. In diesem Fall hat `ip` in der `run`-Methode den Wert `null`.

`NO_UNDO` (Konstante)
: Das Plugin erfordert keine UNDO-Möglichkeit.

`ROI_REQUIRED` (Konstante)
: Das Plugin erfordet ein Bild, in dem die *Region of Interest* (ROI) explizit spezifiziert ist.

`STACK_REQUIRED` (Konstante)
Das Plugin erfordert eine Bildfolge (Stack).

`SUPPORTS_MASKING` (Konstante)
Dies vereinfacht die Bearbeitung nichtrechteckiger ROIs. ImageJ soll nach der Anwendung jene Pixel, die nicht außerhalb der ROI, jedoch innerhalb ihrer Bounding Box liegen, wiederherstellen.

Beispielsweise wäre für ein Plugin, das 8- und 16-Bit Grauwertbilder bearbeiten kann und diese Bilder nicht verändert, der Rückgabewert der `setup()`-Methode

```
DOES_8G + DOES_16G + NO_CHANGES
```

C.18.3 Plugins ausführen – `IJ` (Klasse)

`Object runPlugIn (String className, String arg)`
Erzeugt ein Plugin-Objekt der Klasse `className` und führt die `run`-Methode mit dem Argument `arg` aus. Ist `className` vom Typ `PlugInFilter`, dann wird das Plugin auf das aktuelle Bild angewandt und zuvor die `setup()`-Methode ausgeführt. Rückgabwert ist das Plugin-Objekt.

C.19 Window-Management

C.19.1 `WindowManager` (Klasse)

Diese Klasse stellt statische Methoden zur Manipulation der Bildschirmfenster in ImageJ zur Verfügung.

`static boolean closeAllWindows ()`
Schließt alle offenen Fenster.

`static ImagePlus getCurrentImage ()`
Liefert das aktuell angezeigte Bildobjekt vom Typ `ImagePlus`.

`static ImageWindow getCurrentWindow ()`
Liefert das aktuelle Fenster vom Typ `ImageWindow`.

`static int[] getIDList ()`
Liefert ein Array mit den ID-Nummern der angezeigten Bilder bzw. `null`, wenn kein Bild angezeigt wird. Die Indizes sind ganzzahlige, negative Werte.

`static ImagePlus getImage (int imageID)`
Liefert eine Referenz auf ein Bildobjekt vom Typ `ImagePlus`, wobei folgende Fälle zu unterscheiden sind:
Für `imageID` < 0 wird das Bild mit der angegebenen ID-Nummer geliefert. Für `imageID` > 0 wird jenes Bild geliefert, dass im Ergebnis (Array) von `getIDList()` an der Stelle `imageID` liegt. Für `imageID` $= 0$ oder wenn keine Bilder geöffnet sind ist das Ergebnis `null`.

`static int getWindowCount ()`
Liefert die Anzahl der geöffneten Bilder.

`static void putBehind ()`
Schiebt die Bildschirmanzeige des aktuellen Bilds nach hinten und zeigt das nächste Bild in der Liste (zyklisch) als aktuelles Bild.

`static void setTempCurrentImage (ImagePlus imp)`
Macht das angegebene Bild `imp` vorübergehend zum aktuellen Bild und erlaubt damit die Bearbeitung von Bildern, die nicht in einem Fenster angezeigt werden. Durch erneuten Aufruf mit dem Argument `null` wird zum vorherigen aktuellen Bild zurückgekehrt.

C.20 Weitere Funktionen

C.20.1 `ImagePlus` (Klasse)

`boolean lock ()`
Sperrt dieses Bild für den Zugriff durch andere Threads. Liefert *true*, wenn das Bild erfolgreich gesperrt wurde, und *false*, wenn das Bild bereits gesperrt war.

`boolean lockSilently ()`
Wie `lock()`, jedoch ohne Tonsignal.

`void unlock ()`
Hebt die Sperrung dieses Bild auf.

`FileInfo getOriginalFileInfo ()`
Liefert Informationen über die Datei, aus der das Bild geöffnet wurde. Das resultierende Objekt (vom Typ `ij.io.FileInfo`) enthält u. a. Felder wie `fileName` (`String`), `directory` (`String`) und `description` (`String`).

`ImageProcessor getProcessor ()`
Liefert eine Referenz auf das zugehörige `ImageProcessor`-Objekt.

`void setProcessor (String title, ImageProcessor ip)`
Macht `ip` zum neuen `ImageProcessor`-Objekt dieses Bilds.

C.20.2 `IJ` (Klasse)

`static String freeMemory ()`
Liefert eine Zeichenkette mit der Angabe des freien Speicherplatzes.

`static ImagePlus getImage ()`
Liefert das akutelle (vom Benutzer ausgewählte) Bild vom Typ `ImagePlus` bzw. `null`, wenn kein Bild geöffnet ist.

`static boolean isMacintosh ()`
Liefert *true*, wenn ImageJ gerade auf einem *Macintosh*-Computer läuft.

Programm C.6
Beispiel für die Registrierung eines Plugin mit `IJ.register()`.

```
1 import ij.IJ;
2 import ij.plugin.PlugIn;
3
4 public class TestRegister_ implements PlugIn {
5   static {
6     IJ.register(TestRegister_.class);
7   }
8   static int memorize = 0;
9
10  public void run(String arg) {
11    memorize =
12      (int) IJ.getNumber("Enter a number", memorize);
13    System.gc(); // call Java's garbage collector
14  }
15 }
```

`static boolean isMacOSX ()`
: Liefert *true*, wenn ImageJ gerade auf einem *Macintosh*-Computer unter *OS X* läuft.

`static boolean isWindows ()`
: Liefert *true*, wenn ImageJ gerade auf einem *Windows*-Rechner läuft.

`static void register (Class c)`
: „Registriert“ die angegebene Klasse (ein Objekt vom Typ `java.lang.Class`), sodass sie von Javas Garbage Collector nicht enfernt wird, da dieser die Werte der statischen Klassenvariablen jeweils neu initialisiert.

  Beispiel: Im Plugin in Prog. C.6 soll der Wert der statischen Variable `memorize` (Zeile 8) von einer Ausführung des Plugin zur nächsten erhalten bleiben. Dazu wird die Methode `IJ.register()` innerhalb eines `static`-Blocks der Plugin-Klasse aufgerufen (Zeile 6). Dieser Block wird nur einmal beim Laden des zugehörigen `class`-Files ausgeführt. Innerhalb der `run()`-Methode wird – *nur* als Test – Javas Garbage Collector mit `System.gc()` angestoßen (Prog. C.6, Zeile 13):

`static void wait (int msecs)`
: Hält das Programm (d. h. den zugehörigen *Thread*) für `msecs` Millisekunden an.

D

Source Code

D.1 Harris Corner Detector

Vollständiger Quellcode als Ergänzung zur Beschreibung in Kap. 8.

D.1.1 File `Corner.java`

```
1  import ij.process.ByteProcessor;
2
3  class Corner implements Comparable {
4    int u;
5    int v;
6    float q;
7
8    Corner (int u, int v, float q){
9      this.u = u;
10     this.v = v;
11     this.q = q;
12   }
13
14   public int compareTo (Object obj) {
15     Corner c2 = (Corner) obj;
16     if (this.q > c2.q) return -1;
17     if (this.q < c2.q) return 1;
18     else return 0;
19   }
20
21   double dist2 (Corner c2){
22     int dx = this.u - c2.u;
23     int dy = this.v - c2.v;
24     return (dx*dx)+(dy*dy);
25   }
26
27   void draw(ByteProcessor ip){
28     //draw this corner as a black cross
29     int paintvalue = 0; //black
30     int size = 2;
31     ip.setValue(paintvalue);
32     ip.drawLine(u-size,v,u+size,v);
33     ip.drawLine(u,v-size,u,v+size);
34   }
35 }
```

D.1.2 File HarrisCornerDetector.java

```
1 import ij.IJ;
2 import ij.ImagePlus;
3 import ij.plugin.filter.Convolver;
4 import ij.process.Blitter;
5 import ij.process.ByteProcessor;
6 import ij.process.FloatProcessor;
7 import ij.process.ImageProcessor;
8
9 import java.util.Arrays;
10 import java.util.Collections;
11 import java.util.Iterator;
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
19   float alpha = DEFAULT_ALPHA;
20   int threshold = DEFAULT_THRESHOLD;
21   double dmin = 10;
22
23   final int border = 20;
24
25   //filter kernels (one-dim. part of separable 2D filters)
26   final float[] pfilt = {0.223755f,0.552490f,0.223755f};
27   final float[] dfilt = {0.453014f,0.0f,-0.453014f};
28   final float[] bfilt = {0.01563f,0.09375f,0.234375f,0.3125f
         ,0.234375f,0.09375f,0.01563f};
29                         // = 1,6,15,20,15,6,1/64
30
31   ImageProcessor ipOrig;
32   FloatProcessor A;
33   FloatProcessor B;
34   FloatProcessor C;
35   FloatProcessor Q;
36   Vector<Corner> corners;
37
38   HarrisCornerDetector(ImageProcessor ip){
39     this.ipOrig = ip;
40   }
41
42   HarrisCornerDetector(ImageProcessor ip, float alpha, int
         threshold){
43     this.ipOrig = ip;
44     this.alpha = alpha;
45     this.threshold = threshold;
46   }
47
```

```
48  void findCorners(){
49    makeDerivatives();
50    makeCrf(); //corner response function (CRF)
51    corners = collectCorners(border);
52    corners = cleanupCorners(corners);
53  }
54
55  void makeDerivatives(){
56    FloatProcessor Ix = (FloatProcessor) ipOrig.convertToFloat
          ();
57    FloatProcessor Iy = (FloatProcessor) ipOrig.convertToFloat
          ();
58
59    Ix = convolve1h(convolve1h(Ix,pfilt),dfilt);
60    Iy = convolve1v(convolve1v(Iy,pfilt),dfilt);
61
62    A = sqr((FloatProcessor) Ix.duplicate());
63    A = convolve2(A,bfilt);
64
65    B = sqr((FloatProcessor) Iy.duplicate());
66    B = convolve2(B,bfilt);
67
68    C = mult((FloatProcessor)Ix.duplicate(),Iy);
69    C = convolve2(C,bfilt);
70  }
71
72  void makeCrf() { //corner response function (CRF)
73    int w = ipOrig.getWidth();
74    int h = ipOrig.getHeight();
75    Q = new FloatProcessor(w,h);
76    float[] Apix = (float[]) A.getPixels();
77    float[] Bpix = (float[]) B.getPixels();
78    float[] Cpix = (float[]) C.getPixels();
79    float[] Qpix = (float[]) Q.getPixels();
80    for (int v=0; v<h; v++) {
81      for (int u=0; u<w; u++) {
82        int i = v*w+u;
83        float a = Apix[i], b = Bpix[i], c = Cpix[i];
84        float det = a*b-c*c;
85        float trace = a+b;
86        Qpix[i] = det - alpha * (trace * trace);
87      }
88    }
89  }
90
91  Vector<Corner> collectCorners(int border) {
92    Vector<Corner> cornerList = new Vector<Corner>(1000);
93    int w = Q.getWidth();
94    int h = Q.getHeight();
95    float[] Qpix = (float[]) Q.getPixels();
96    for (int v=border; v<h-border; v++){
```

```
97      for (int u=border; u<w-border; u++) {
98        float q = Qpix[v*w+u];
99        if (q>threshold && isLocalMax(Q,u,v)) {
100         Corner c = new Corner(u,v,q);
101         cornerList.add(c);
102       }
103     }
104   }
105   Collections.sort(cornerList);
106   return cornerList;
107 }
108
109 Vector<Corner> cleanupCorners(Vector<Corner> corners){
110   double dmin2 = dmin*dmin;
111   Corner[] cornerArray = new Corner[corners.size()];
112   cornerArray = corners.toArray(cornerArray);
113   Vector<Corner> goodCorners = new Vector<Corner>(corners.
          size());
114   for (int i=0; i<cornerArray.length; i++){
115     if (cornerArray[i] != null){
116       Corner c1 = cornerArray[i];
117       goodCorners.add(c1);
118       //delete all remaining corners close to c
119       for (int j=i+1; j<cornerArray.length; j++){
120         if (cornerArray[j] != null){
121           Corner c2 = cornerArray[j];
122           if (c1.dist2(c2)<dmin2)
123             cornerArray[j] = null; //delete corner
124         }
125       }
126     }
127   }
128   return goodCorners;
129 }
130
131 void printCornerPoints(Vector crf){
132   Iterator it = crf.iterator();
133   for (int i=0; it.hasNext(); i++){
134     Corner ipt = (Corner) it.next();
135     IJ.write(i + ": " + (int)ipt.q + " " + ipt.u + " " + ipt.
          v);
136   }
137 }
138
139 ImageProcessor showCornerPoints(ImageProcessor ip){
140   ByteProcessor ipResult = (ByteProcessor)ip.duplicate();
141   //change background image contrast and brightness
142   int[] lookupTable = new int[256];
143   for (int i=0; i<256; i++){
144     lookupTable[i] = 128 + (i/2);
145   }
```

```
146     ipResult.applyTable(lookupTable);
147     //draw corners:
148     Iterator<Corner> it = corners.iterator();
149     for (int i=0; it.hasNext(); i++){
150       Corner c = it.next();
151       c.draw(ipResult);
152     }
153     return ipResult;
154   }
155
156   void showProcessor(ImageProcessor ip, String title){
157     ImagePlus win = new ImagePlus(title,ip);
158     win.show();
159   }
160
161   // utility methods for float processors
162
163   static FloatProcessor convolve1h (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 (FloatProcessor p, float[]
          h) {
171     Convolver conv = new Convolver();
172     conv.setNormalize(false);
173     conv.convolve(p, h, h.length, 1);
174     return p;
175   }
176
177   static FloatProcessor convolve2 (FloatProcessor p, float[] h
          ) {
178     convolve1h(p,h);
179     convolve1v(p,h);
180     return p;
181   }
182
183
184   static FloatProcessor sqr (FloatProcessor fp1) {
185     fp1.sqr();
186     return fp1;
187   }
188
189   static FloatProcessor mult (FloatProcessor fp1,
          FloatProcessor fp2) {
190     int mode = Blitter.MULTIPLY;
191     fp1.copyBits(fp2, 0, 0, mode);
192     return fp1;
```

```
193   }
194
195   static boolean isLocalMax (FloatProcessor fp, int u, int v)
          {
196     int w = fp.getWidth();
197     int h = fp.getHeight();
198     if (u<=0 || u>=w-1 || v<=0 || v>=h-1)
199       return false;
200     else {
201       float[] pix = (float[]) fp.getPixels();
202       int i0 = (v-1)*w+u, i1 = v*w+u, i2 = (v+1)*w+u;
203       float cp = pix[i1];
204       return
205         cp > pix[i0-1] && cp > pix[i0] && cp > pix[i0+1] &&
206         cp > pix[i1-1] &&              cp > pix[i1+1] &&
207         cp > pix[i2-1] && cp > pix[i2] && cp > pix[i2+1] ;
208     }
209   }
210
211 }
```

D.1.3 File HarrisCornerPlugin_.java

```
 1 import ij.IJ;
 2 import ij.ImagePlus;
 3 import ij.gui.GenericDialog;
 4 import ij.plugin.filter.PlugInFilter;
 5 import ij.process.ImageProcessor;
 6
 7 public class HarrisCornerPlugin_ implements PlugInFilter {
 8   ImagePlus imp;
 9   static float alpha = HarrisCornerDet.DEFAULT_ALPHA;
10   static int threshold = HarrisCornerDet.DEFAULT_THRESHOLD;
11   static int nmax = 0; //points to show
12
13     public int setup(String arg, ImagePlus imp) {
14     IJ.register(HarrisCornerPlugin_.class);
15       this.imp = imp;
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;
25         //dialog was cancelled or error occured
26     HarrisCornerDet hcd = new HarrisCornerDet(ip,alpha,
           threshold);
```

```
27    hcd.findCorners();
28    ImageProcessor result = hcd.showCornerPoints(ip);
29    ImagePlus win = new ImagePlus("Corners from " + imp.
          getTitle(),result);
30    win.show();
31    }
32
33    void showAbout() {
34        String cn = getClass().getName();
35        IJ.showMessage("About "+cn+" ...",
36            "Harris Corner Detector"
37        );
38    }
39
40   private boolean showDialog() {
41     // display dialog , return false if cancelled or on error.
42     GenericDialog dlg =
43       new GenericDialog("Harris Corner Detector", IJ.
             getInstance());
44     float def_alpha = HarrisCornerDet.DEFAULT_ALPHA;
45     dlg.addNumericField("Alpha (default: "+def_alpha+")", alpha
           , 3);
46     int def_threshold = HarrisCornerDet.DEFAULT_THRESHOLD;
47     dlg.addNumericField("Threshold (default: "+def_threshold+")
           ", threshold, 0);
48     dlg.addNumericField("Max. points (0 = show all)", nmax, 0);
49     dlg.showDialog();
50     if(dlg.wasCanceled())
51       return false;
52     if(dlg.invalidNumber()) {
53       IJ.showMessage("Error", "Invalid input number");
54       return false;
55     }
56     alpha = (float) dlg.getNextNumber();
57     threshold = (int) dlg.getNextNumber();
58     nmax = (int) dlg.getNextNumber();
59     return true;
60   }
61 }
```

D.2 Kombinierte Regionenmarkierung-Konturverfolgung

Vollständiger Quellcode als Ergänzung zur Beschreibung in Abschn. 11.2.

D.2.1 File `ContourTracingPlugin_.java`

```
1  import ij.IJ;
2  import ij.ImagePlus;
3  import ij.gui.ImageWindow;
4  import ij.plugin.filter.PlugInFilter;
5  import ij.process.ImageProcessor;
6
7  // Uses the ContourTracer class to create an ordered list
8  // of points representing the internal and external contours
9  // of each region in the binary image. Instead of drawing
10 // directly into the image, we make use of ImageJ's ImageCanvas
11 // to draw the contours in a layer ontop of the image.
12 // Illustrates how to use the Java2D api to draw the
13 // polygons and scale and transform them to match ImageJ's
14 // zooming.
15
16 public class ContourTracingPlugin_ implements PlugInFilter {
17
18   public int setup(String arg, ImagePlus imp) {
19     return DOES_8G + NO_CHANGES;
20   }
21
22   public void run(ImageProcessor ip) {
23     ImageProcessor ip2 = ip.duplicate();
24     // trace the contours and get the ArrayList
25     ContourTracer tracer = new ContourTracer(ip2);
26     ContourSet cs = tracer.getContours();
27     //contours.print();
28
29     // change lookup-table to show gray regions
30     ip2.setMinAndMax(0,512);
31
32     ImagePlus imp =
33       new ImagePlus("Contours of " + IJ.getImage().getTitle(),
           ip2);
34
35     ContourOverlay cc = new ContourOverlay(imp, cs);
36     new ImageWindow(imp, cc);
37   }
38 }
```

D.2.2 File Node.java

```
1
2 public class Node {
3   int x;
4   int y;
5
6   Node(int x, int y) {
7     this.x = x;
8     this.y = y;
9   }
10
11  void moveBy (int dx, int dy) {
12    x = x + dx;
13    y = y + dy;
14  }
15 }
```

D.2.3 File Contour.java

```
1 import ij.IJ;
2
3 import java.awt.Polygon;
4 import java.awt.Shape;
5 import java.awt.geom.Ellipse2D;
6 import java.util.ArrayList;
7 import java.util.Iterator;
8
9 abstract class Contour { // generic contour, never instantiated
10  int label;
11  ArrayList<Node> nodes;
12
13  Contour (int label, int initialSize) {
14    this.label = label;
15    nodes = new ArrayList<Node>(initialSize);
16  }
17
18  void addNode (Node n){
19    nodes.add(n);
20  }
21
22  Shape makePolygon() {
23    int m = nodes.size();
24    if (m>1){
25      int[] xPoints = new int[m];
26      int[] yPoints = new int[m];
27      int k = 0;
28      Iterator<Node> itr = nodes.iterator();
29      while (itr.hasNext() && k < m) {
30        Node cn = itr.next();
```

```
31        xPoints[k] = cn.x;
32        yPoints[k] = cn.y;
33        k = k + 1;
34      }
35      return new Polygon(xPoints, yPoints, m);
36    }
37    else { // use circles for isolated pixels
38      Node cn = nodes.get(0);
39      return new Ellipse2D.Double(cn.x-0.1, cn.y-0.1, 0.2, 0.2)
            ;
40    }
41  }
42
43  void moveBy (int dx, int dy) {
44    Iterator<Node> itr = nodes.iterator();
45    while (itr.hasNext()) {
46      Node cn = itr.next();
47      cn.moveBy(dx,dy);
48    }
49  }
50
51  // debug methods:
52
53  abstract void print();
54
55  void printNodes (){
56    Iterator<Node> itr = nodes.iterator();
57    while (itr.hasNext()) {
58      Node n = itr.next();
59      IJ.write("  Node " + n.x + "/" + n.y);
60    }
61  }
62 }
```

D.2.4 File `OuterContour.java`

```
1 import ij.IJ;
2
3 class OuterContour extends Contour {
4
5   OuterContour (int label, int initialSize) {
6     super(label, initialSize);
7   }
8
9   void print() {
10    IJ.write("Outer Contour: " + nodes.size());
11    printNodes();
12  }
13 }
```

D.2.5 File InnerContour.java

```
1 import ij.IJ;
2
3 class InnerContour extends Contour {
4
5   InnerContour (int label, int initialSize) {
6     super(label, initialSize);
7   }
8
9   void print() {
10    IJ.write("Inner Contour: " + nodes.size());
11    printNodes();
12  }
13 }
```

D.2.6 File ContourSet.java

```
1 import java.awt.Shape;
2 import java.util.ArrayList;
3 import java.util.Iterator;
4
5 class ContourSet {
6   ArrayList<Contour> outerContours;
7   ArrayList<Contour> innerContours;
8
9   ContourSet (int initialSize){
10    outerContours = new ArrayList<Contour>(initialSize);
11    innerContours = new ArrayList<Contour>(initialSize);
12  }
13
14  void addContour(OuterContour oc) {
15    outerContours.add(oc);
16  }
17
18  void addContour(InnerContour ic) {
19    innerContours.add(ic);
20  }
21
22  Shape[] getOuterPolygons () {
23    return makePolygons(outerContours);
24  }
25
26  Shape[] getInnerPolygons () {
27    return makePolygons(innerContours);
28  }
29
30  Shape[] makePolygons(ArrayList<Contour> nodes) {
31    if (nodes == null)
32      return null;
```

```
33      else {
34        Shape[] pa = new Shape[nodes.size()];
35        int i = 0;
36        Iterator<Contour> itr = nodes.iterator();
37        while (itr.hasNext()) {
38          Contour c = itr.next();
39          pa[i] = c.makePolygon();
40          i = i + 1;
41        }
42        return pa;
43      }
44    }
45
46    void moveBy (int dx, int dy) {
47      Iterator<Contour> itr;
48      itr = outerContours.iterator();
49      while (itr.hasNext()) {
50        Contour c = itr.next();
51        c.moveBy(dx,dy);
52      }
53      itr = innerContours.iterator();
54      while (itr.hasNext()) {
55        Contour c = itr.next();
56        c.moveBy(dx,dy);
57      }
58    }
59
60    // utility methods:
61
62    void print() {
63      printContours(outerContours);
64      printContours(innerContours);
65    }
66
67    void printContours (ArrayList<Contour> ctrs) {
68      Iterator<Contour> itr = ctrs.iterator();
69      while (itr.hasNext()) {
70        Contour c = itr.next();
71        c.print();
72      }
73    }
74  }
```

D.2.7 File ContourTracer.java

```
1  import ij.process.ImageProcessor;
2
3  public class ContourTracer {
4    static final byte FOREGROUND = 1;
5    static final byte BACKGROUND = 0;
6
7    ImageProcessor ip;
8    byte[][] pixelMap;
9    int[][] labelMap;
10   // label values in labelMap can be:
11   // 0 ... unlabeled
12   // −1 ... previously visited background pixel
13   // > 0 ... valid region label
14
15   public ContourTracer (ImageProcessor ip) {
16     this.ip = ip;
17     int h = ip.getHeight();
18     int w = ip.getWidth();
19     pixelMap  = new byte[h+2][w+2];
20     labelMap = new int[h+2][w+2];
21
22     // create auxil. arrays
23     for (int v = 0; v < h+2; v++) {
24       for (int u = 0; u < w+2; u++) {
25         if (ip.getPixel(u-1,v-1) == 0)
26           pixelMap[v][u] = BACKGROUND;
27         else
28           pixelMap[v][u] = FOREGROUND;
29       }
30     }
31   }
32
33   OuterContour traceOuterContour (int cx, int cy, int label) {
34     OuterContour cont = new OuterContour(label, 50);
35     traceContour(cx, cy, label, 0, cont);
36     return cont;
37   }
38
39   InnerContour traceInnerContour(int cx, int cy, int label) {
40     InnerContour cont = new InnerContour(label, 50);
41     traceContour(cx, cy, label, 1, cont);
42     return cont;
43   }
44
45   // trace one contour starting at xS, yS in direction dir
46   Contour traceContour (int xS, int yS, int label, int dir,
         Contour cont) {
47     int xT, yT; // T = successor of starting point S
48     int xP, yP; // P = „previous“ contour point
```

```
49    int xC, yC; // C = „current“ contour point
50    boolean done;
51
52    Node n = new Node(xS, yS);
53    dir = findNextNode(n, dir);
54    cont.addNode(n); // add node T (may be the ident. to S)
55
56    xP = xS;
57    yP = yS;
58    xC = xT = n.x;
59    yC = yT = n.y;
60    done = (xS==xT && yS==yT); // isolated pixel
61
62    while (!done) {
63      labelMap[yC][xC] = label;
64      n = new Node(xC, yC);
65      dir = findNextNode(n, (dir + 6) % 8);
66      xP = xC; yP = yC; //set „previous“ (P)
67      xC = n.x; yC = n.y; //set „current“ (C)
68      // back to the starting position?
69      done = (xP==xS && yP==yS && xC==xT && yC==yT);
70      if (!done) {
71        cont.addNode(n);
72      }
73    }
74    return cont;
75  }
76
77  int findNextNode (Node Xc, int dir) {
78    // starts at Node Xc in direction dir
79    // returns the final tracing direction
80    final int[][] delta = {
81      { 1,0}, { 1, 1}, {0, 1}, {-1, 1},
82      {-1,0}, {-1,-1}, {0,-1}, { 1,-1}};
83    for (int i = 0; i < 7; i++) {
84      int x = Xc.x + delta[dir][0];
85      int y = Xc.y + delta[dir][1];
86      if (pixelMap[y][x] == BACKGROUND) {
87        // mark surrounding background pixels
88        labelMap[y][x] = -1;
89        dir = (dir + 1) % 8;
90      }
91      else { // found non-background pixel
92        Xc.x = x; Xc.y = y;
93        break;
94      }
95    }
96    return dir;
97  }
98
99  ContourSet getContours() {
```

```
100     ContourSet contours = new ContourSet(50);
101     int region = 0;   // region counter
102     int label = 0;   // current label
103
104     // scan top to bottom, left to right
105     for (int v = 1; v < pixelMap.length-1; v++) {
106       label = 0; // no label
107       for (int u = 1; u < pixelMap[v].length-1; u++) {
108
109         if (pixelMap[v][u] == FOREGROUND) {
110           if (label != 0) { // keep using same label
111             labelMap[v][u] = label;
112           }
113           else {
114             label = labelMap[v][u];
115             if (label == 0) {
116               // unlabeled – new outer contour
117               region = region + 1;
118               label = region;
119               OuterContour co = traceOuterContour(u,v,label);
120               contours.addContour(co);
121               labelMap[v][u] = label;
122             }
123           }
124         }
125         else { // BACKGROUND pixel
126           if (label != 0) {
127             if (labelMap[v][u] == 0) {
128               // unlabeled – new inner contour
129               InnerContour ci = traceInnerContour(u-1,v,label);
130               contours.addContour(ci);
131             }
132             label = 0;
133           }
134         }
135       }
136     }
137     contours.moveBy(-1,-1); // shift back to original coordinates
138     return (contours);
139   }
140 }
```

D.2.8 File ContourOverlay.java

```
1  import ij.ImagePlus;
2  import ij.gui.ImageCanvas;
3  import java.awt.BasicStroke;
4  import java.awt.Color;
5  import java.awt.Graphics;
6  import java.awt.Graphics2D;
7  import java.awt.Polygon;
8  import java.awt.RenderingHints;
9  import java.awt.Shape;
10 import java.awt.Stroke;
11
12 class ContourOverlay extends ImageCanvas {
13   private static final long serialVersionUID = 1L;
14   static float strokeWidth = 0.5f;
15   static int capsstyle = BasicStroke.CAP_ROUND;
16   static int joinstyle = BasicStroke.JOIN_ROUND;
17   static Color outerColor = Color.black;
18   static Color innerColor = Color.white;
19   static float[] outerDashing = {strokeWidth * 2.0f,
         strokeWidth * 2.5f};
20   static float[] innerDashing = {strokeWidth * 0.5f,
         strokeWidth * 2.5f};
21   static boolean DRAW_CONTOURS = true;
22
23   Shape[] outerContourShapes;
24   Shape[] innerContourShapes;
25
26   ContourOverlay(ImagePlus imp, ContourSet contours) {
27     super(imp);
28     outerContourShapes = contours.getOuterPolygons();
29     innerContourShapes = contours.getInnerPolygons();
30   }
31
32   public void paint(Graphics g) {
33     super.paint(g);
34     drawContours(g);
35   }
36
37   private void drawContours(Graphics g) {
38     Graphics2D g2d = (Graphics2D) g;
39     g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
           RenderingHints.VALUE_ANTIALIAS_ON);
40
41     // scale and move overlay to the pixel centers
42     g2d.scale(this.getMagnification(), this.getMagnification())
           ;
43     g2d.translate(0.5-this.srcRect.x, 0.5-this.srcRect.y);
44
45     if (DRAW_CONTOURS) {
```

```
46      Stroke solidStroke =
47        new BasicStroke(strokeWidth, capsstyle, joinstyle);
48      Stroke dashedStrokeOuter =
49        new BasicStroke(strokeWidth, capsstyle, joinstyle, 1.0f
              , outerDashing, 0.0f);
50      Stroke dashedStrokeInner =
51        new BasicStroke(strokeWidth, capsstyle, joinstyle, 1.0f
              , innerDashing, 0.0f);
52
53      drawShapes(outerContourShapes, g2d, solidStroke,
            dashedStrokeOuter, outerColor);
54      drawShapes(innerContourShapes, g2d, solidStroke,
            dashedStrokeInner, innerColor);
55    }
56  }
57
58  void drawShapes(Shape[] shapes, Graphics2D g2d, Stroke
        solidStrk, Stroke dashedStrk, Color col) {
59    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
          RenderingHints.VALUE_ANTIALIAS_ON);
60    g2d.setColor(col);
61    for (int i = 0; i < shapes.length; i++) {
62      Shape s = shapes[i];
63      if (s instanceof Polygon)
64        g2d.setStroke(dashedStrk);
65      else
66        g2d.setStroke(solidStrk);
67      g2d.draw(shapes[i]);
68    }
69  }
70
71 }
```

Literaturverzeichnis

1. Adobe Systems, http://www.adobe.com/digitalimag/pdfs/AdobeRGB1998.pdf: *Adobe RGB (1998) Color Space Specification*, 2005.
2. Aho, A. V., J. E. Hopcroft und J. D. Ullman: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
3. Bailer, W.: *Writing ImageJ Plugins – A Tutorial*. http://www.fh-hagenberg.at/mtd/depot/imaging/imagej/, 2003.
4. Ballard, D. H. und C. M. Brown: *Computer Vision*. Prentice-Hall, 1982.
5. Barber, C. B., D. P. Dobkin und H. Huhdanpaa: *The quickhull algorithm for convex hulls*. ACM Trans. Math. Softw., 22(4), S. 469–483, 1996.
6. Barrow, H. G., J. M. Tenenbaum, R. C. Bolles und H. C. Wolf: *Parametric correspondence and chamfer matching: two new techniques for image matching*. In: *Proc. International Joint Conf. on Artificial Intelligence*, S. 659–663, Cambridge, MA, 1977.
7. Blahut, R. E.: *Fast Algorithms for Digital Signal Processing*. Addison-Wesley, 1985.
8. Borgefors, G.: *Distance transformations in digital images*. Computer Vision, Graphics and Image Processing, 34, S. 344–371, 1986.
9. Borgefors, G.: *Hierarchical chamfer matching: a parametric edge matching algorithm*. IEEE Trans. Pattern Analysis and Machine Intelligence, 10(6), S. 849–865, 1988.
10. Bresenham, J. E.: *A Linear Algorithm for Incremental Digital Display of Circular Arcs*. Communications of the ACM, 20(2), S. 100–106, 1977.
11. Brigham, E. O.: *The Fast Fourier Transform and Its Applications*. Prentice-Hall, 1988.
12. Bronstein, I. N., K. A. Semendjajew, G. Musiol und H. Mühlig: *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 5. Aufl., 2000.
13. Bunke, H. und P. S. P. Wang (Hrsg.): *Handbook of character recognition and document image analysis*. World Scientific, 2000.
14. Burt, P. J. und E. H. Adelson: *The Laplacian pyramid as a compact image code*. IEEE Trans. Communications, 31(4), S. 532–540, 1983.

15. CANNY, J. F.: *A computational approach to edge detection*. IEEE Trans. on Pattern Analysis and Machine Intelligence, 8(6), S. 679–698, 1986.
16. CASTLEMAN, K. R.: *Digital Image Processing*. Pearson Education, 1995.
17. CHANG, F., C. J. CHEN und C. J. LU: *A linear-time component-labeling algorithm using contour tracing technique*. Computer Vision, Graphics, and Image Processing: Image Understanding, 93(2), S. 206–220, February 2004.
18. COHEN, P. R. und E. A. FEIGENBAUM: *The Handbook of Artificial Intelligence*. William Kaufmann, Inc., 1982.
19. CORMAN, T. H., C. E. LEISERSON, R. L. RIVEST und C. STEIN: *Introduction to Algorithms*. MIT Press, 2. Aufl., 2001.
20. DAVIS, L. S.: *A Survey of Edge Detection Techniques*. Computer Graphics and Image Processing, 4, S. 248–270, 1975.
21. DUDA, R. O., P. E. HART und D. G. STORK: *Pattern Classification*. Wiley, 2001.
22. EFFORD, N.: *Digital Image Processing – A Practical Introduction Using Java*. Pearson Education, 2000.
23. FOLEY, J. D., A. VAN DAM, S. K. FEINER und J. F. HUGHES: *Computer Graphics: Principles and Practice*. Addison-Wesley, 2. Aufl., 1996.
24. FORD, A. und A. ROBERTS: *Colour Space Conversions*. http://www.poynton.com/PDFs/coloureq.pdf, 1998.
25. FÖRSTNER, W. und E. GÜLCH: *A fast operator for detection and precise location of distinct points, corners and centres of circular features*. In: *ISPRS Intercommission Workshop*, S. 149–155, Interlaken, June 1987.
26. FORSYTH, D. A. und J. PONCE: *Computer Vision – A Modern Approach*. Prentice Hall, 2003.
27. FREEMAN, H.: *Computer Processing of Line Drawing Images*. ACM Computing Surveys, 6(1), S. 57–97, March 1974.
28. GERVAUTZ, M. und W. PURGATHOFER: *A simple method for color quantization: octree quantization*. In: GLASSNER, A. (Hrsg.): *Graphics Gems I*, S. 287–293. Academic Press, 1990.
29. GLASSNER, A. S.: *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers, Inc., 1995.
30. GONZALEZ, R. C. und R. E. WOODS: *Digital Image Processing*. Addison-Wesley, 1992.
31. GREEN, P.: *Colorimetry and colour differences*. In: GREEN, P. und L. MACDONALD (Hrsg.): *Colour Engineering*, Kap. 3, S. 40–77. Wiley, 2002.
32. GÜTING, R. H. und S. DIEKER: *Datenstrukturen und Algorithmen*. Teubner, 2. Aufl., 2003.
33. HALL, E. L.: *Computer Image Processing and Recognition*. Academic Press, 1979.
34. HARRIS, C. G. und M. STEPHENS: *A combined corner and edge detector*. In: *4th Alvey Vision Conference*, S. 147–151, 1988.
35. HECKBERT, P.: *Color Image Quantization for Frame Buffer Display*. ACM Transactions on Computer Graphics (SIGGRAPH), S. 297–307, 1982.
36. HOLM, J., I. TASTL, L. HANLON und P. HUBEL: *Color processing for digital photography*. In: GREEN, P. und L. MACDONALD (Hrsg.): *Colour Engineering*, Kap. 9, S. 179–220. Wiley, 2002.
37. HORN, B. K. P.: *Robot Vision*. MIT-Press, 1982.
38. HOUGH, P. V. C.: *Method and means for recognizing complex patterns*. US-Patent 3,069,654, 1962.

39. HU, M. K.: *Visual Pattern Recognition by Moment Invariants*. IEEE Trans. Information Theory, 8, S. 179–187, 1962.
40. HUNT, R. W. G.: *The Reproduction of Colour*. Wiley, 6. Aufl., 2004.
41. 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. ILLINGWORTH, J. und J. KITTLER: *A Survey of the Hough Transform*. Computer Vision, Graphics and Image Processing, 44, S. 87–116, 1988.
43. INTERNATIONAL TELECOMMUNICATIONS UNION: *ITU-R Recommendation BT.709-3: Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange*, 1998.
44. INTERNATIONAL TELECOMMUNICATIONS UNION: *ITU-R Recommendation BT.601-5: Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios*, 1999.
45. ISO 12655: *Graphic technology – spectral measurement and colorimetric computation for graphic arts images*, 1996.
46. JACK, K.: *Video Demystified – A Handbook for the Digital Engineer*. LLH Technology Publishing, 2001.
47. JÄHNE, B.: *Practical Handbook on Image Processing for Scientific Applications*. CRC Press, 1997.
48. JÄHNE, B.: *Digitale Bildverarbeitung*. Springer-Verlag, 5. Aufl., 2002.
49. JAIN, A. K.: *Fundamentals of Digital Image Processing*. Prentice-Hall, 1989.
50. JIANG, X. Y. und H. BUNKE: *Simple and fast computation of moments*. Pattern Recogn., 24(8), S. 801–806, 1991. http://wwwmath.uni-muenster.de/u/xjiang/papers/PR1991.ps.gz.
51. KING, J.: *Engineering color at Adobe*. In: GREEN, P. und L. MACDONALD (Hrsg.): *Colour Engineering*, Kap. 15, S. 341–369. Wiley, 2002.
52. KIRSCH, R. A.: *Computer determination of the constituent structure of biological images*. Computers in Biomedical Research, 4, S. 315–328, 1971.
53. KITCHEN, L. und A. ROSENFELD: *Gray-level corner detection*. Pattern Recognition Letters, 1, S. 95–102, 1982.
54. LUCAS, B. und T. KANADE: *An iterative image registration technique with an application to stereo vision*. In: *Proc. International Joint Conf. on Artificial Intelligence*, S. 674–679, Vancouver, 1981.
55. MALLAT, S.: *A Wavelet Tour of Signal Processing*. Academic Press, 1999.
56. MARR, D. und E. HILDRETH: *Theory of edge detection*. Proc. R. Soc. London, Ser. B, 207, S. 187–217, 1980.
57. MEIJERING, E. H. W., W. J. NIESSEN und M. A. VIERGEVER: *Quantitative Evaluation of Convolution-Based Methods Medical Image Interpolation*. Medical Image Analysis, 5(2), S. 111–126, 2001. http://imagescience.bigr.nl/meijering/software/transformj/.
58. MIANO, J.: *Compressed Image File Formats*. ACM Press, Addison-Wesley, 1999.
59. MLSNA, P. A. und J. J. RODRIGUEZ: *Gradient and Laplacian-Type Edge Detection*. In: BOVIK, A. (Hrsg.): *Handbook of Image and Video Processing*, S. 415–431. Academic Press, 2000.
60. MÖSSENBÖCK, H.: *Sprechen Sie Java*. dpunkt.verlag, 2002.
61. MURRAY, J. D. und W. VANRYPER: *Encyclopedia of Graphics File Formats*. O'Reilly, 2. Aufl., 1996.
62. NADLER, M. und E. P. SMITH: *Pattern Recognition Engineering*. Wiley, 1993.

63. Oppenheim, A. V., R. W. Shafer und J. R. Buck: *Discrete-Time Signal Processing*. Prentice Hall, 2. Aufl., 1999.
64. Pavlidis, T.: *Algorithms for Graphics and Image Processing*. Computer Science Press / Springer-Verlag, 1982.
65. Poynton, C.: *Digital Video and HDTV Algorithms and Interfaces*. Morgan Kaufmann Publishers Inc., 2003.
66. Rasband, W. S.: *ImageJ*. U.S. National Institutes of Health, Bethesda, Maryland, USA, http://rsb.info.nih.gov/ij/, 1997–2006.
67. Reid, C. E. und T. B. Passin: *Signal Processing in C*. Wiley, 1992.
68. Rich, D.: *Instruments and Methods for Colour Measurement*. In: Green, P. und L. MacDonald (Hrsg.): *Colour Engineering*, Kap. 2, S. 19–48. Wiley, 2002.
69. Richardson, I. E. G.: *H.264 and MPEG-4 Video Compression*. Wiley, 2003.
70. Roberts, L. G.: *Machine perception of three-dimensional solids*. In: Tippet, J. T. (Hrsg.): *Optical and Electro-Optical Information Processing*, S. 159–197. MIT Press, Cambridge, MA, 1965.
71. Rosenfeld, A. und P. Pfaltz: *Sequential Operations in Digital Picture Processing*. Journal of the ACM, 12, S. 471–494, 1966.
72. Russ, J. C.: *The Image Processing Handbook*. CRC Press, 3. Aufl., 1998.
73. Schmid, C., R. Mohr und C. Bauckhage: *Evaluation of Interest Point Detectors*. International Journal of Computer Vision, S. 151–172, 2000.
74. Schwarzer, Y. (Hrsg.): *Die Farbenlehre Goethes*. Westerweide Verlag, 2004.
75. Seul, M., L. O'Gorrman und M. J. Sammon: *Practical Algorithms for Image Analysis*. Cambridge University Press, 2000.
76. Shapiro, L. G. und G. C. Stockman: *Computer Vision*. Prentice-Hall, 2001.
77. Silvestrini, N. und E. P. Fischer: *Farbsysteme in Kunst und Wissenschaft*. DuMont, 1998.
78. Sirisathitkul, Y., S. Auwatanamongkol und B. Uyyanonvara: *Color image quantization using distances between adjacent colors along the color axis with highest color variance*. Pattern Recognition Letters, 25, S. 1025–1043, 2004.
79. Smith, S. M. und J. M. Brady: *SUSAN – A New Approach to Low Level Image Processing*. International Journal of Computer Vision, 23(1), S. 45–78, 1997.
80. Sonka, M., V. Hlavac und R. Boyle: *Image Processing, Analysis and Machine Vision*. PWS Publishing, 2. Aufl., 1999.
81. Stokes, M. und M. Anderson: *A Standard Default Color Space for the Internet – sRGB*. Hewlett-Packard, Microsoft, www.w3.org/Graphics/Color/sRGB.html, 1996.
82. Süsstrunk, S.: *Managing color in digital image libraries*. In: Green, P. und L. MacDonald (Hrsg.): *Colour Engineering*, Kap. 17, S. 385–419. Wiley, 2002.
83. Theodoridis, S. und K. Koutroumbas: *Pattern Recognition*. Academic Press, 1999.
84. Trucco, E. und A. Verri: *Introductory Techniques for 3-D Computer Vision*. Prentice Hall, 1998.
85. Turkowski, K.: *Filters for common resampling tasks*. In: Glassner, A. (Hrsg.): *Graphics Gems I*, S. 147–165. Academic Press, 1990.

86. Wallner, D.: *Color management and Transformation through ICC profiles*. In: Green, P. und L. MacDonald (Hrsg.): *Colour Engineering*, Kap. 11, S. 247–261. Wiley, 2002.
87. Watt, A.: *3D-Computergrafik*. Addison-Wesley, 3. Aufl., 2002.
88. Watt, A. und F. Policarpo: *The Computer Image*. Addison-Wesley, 1999.
89. Wolberg, G.: *Digital Image Warping*. IEEE Computer Society Press, 1990.
90. Zeidler, E. (Hrsg.): *Teubner-Taschenbuch der Mathematik*. B. G. Teubner Verlag, 2. Aufl., 2002.
91. Zhang, T. Y. und C. Y. Suen: *A Fast Parallel Algorithm for Thinning Digital Patterns*. Communications of the ACM, 27(3), S. 236–239, 1984.

Sachverzeichnis

Symbole

`&` 238, 438, 457
`*` 101, 431
`<<` 239
`>>` 239
⌊ ⌋ 49, 63, 431
∂ 119, 140, 431
∇ 119, 130, 431
`%` 437
`|` 239

A

Abbildung
  affine 366, 374
  bilineare 372, 374
  lokale 376
  nichtlineare 373
  projektive 364, 367–372, 374
  Ripple 375
  sphärische 375
  Twirl 374
Ableitung
  erste 118, 144
  partielle 119
  zweite 125, 130, 132
`abs` (Methode) 438
Abstand 413, 418, 419
Abtastfrequenz 316, 338
Abtastintervall 313, 314, 338
Abtasttheorem 315, 321, 381
Abtastung 311
  räumlich 9
  zeitlich 9
`acos` (Methode) 438
`ADD` (Konstante) 84, 460
`add` (Methode) 83, 459
`addChoice` (Methode) 87
`addFirst` (Methode) 199
`addNumericField` (Methode) 87
`addSlice` (Methode) 464
Adobe Illustrator 15
Adobe Photoshop 61, 96, 115
Adobe RGB 282
affine Abbildung 366, 374
`AffineMapping` (Klasse) 400, 401
`AffineTransform` (Klasse) 396
Ähnlichkeit 310, 412
Akkumulator-Array 159
Aliasing . 315, 321, 324, 325, 338, 392
Alpha
  Blending 84, 85, 465
  -kanal 17
  -wert 84, 238
Amplitude 301, 302
`AND` (Konstante) 84, 460
`and` (Methode) 459
`applyTable` (Methode) . 82, 151, 459
`applyTo` (Methode) ... 397, 399, 403, 405, 408, 409
`ArrayList` (Klasse) 149
`Arrays` (Klasse) 288
`asin` (Methode) 438

`atan` (Methode) 438
`atan2` (Methode) 431, 438, 439
Auflösung 10
automatische Kontrastanpassung . 59
`autoThreshold` (Methode) 459
`AVERAGE` (Konstante) 84, 460
AWT 238

B

`BACK` (Konstante) 462
background 172
Bandbreite 316
Bartlett-Fenster 343, 345, 346
Basisfunktion 320, 332, 355, 356, 361
Baum 198
`beep` (Methode) 471
Belichtung 41
`BicubicInterpolator` (Klasse) . 407, 408
big-endian 22, 24
bikubische Interpolation 389
Bildanalyse 3
Bildaufnahme 5
Bildebene 7
Bildfehler 43
Bildgröße 10
Bildkompression
  und Histogramm 45
Bildkoordinaten 11, 432
Bildtiefe 12
Bildvergleich 411–430
bilineare Abbildung 372, 374
bilineare Interpolation 387
`BilinearInterpolator` (Klasse) 406, 409
`BilinearMapping` (Klasse) 403
Binärbild 129, 171, 195
`BinaryProcessor` (Klasse) .. 57, 191, 448
`binnedHistogram` (Methode) 50
binning 48–50, 52
Bitmap-Bild 12, 214
Bitmaske 239
Bitoperation 240
Black Box 102
black-generation function 268
`Blitter` (Klasse) 83
BMP 21, 24, 241
bounding box 221, 226
Box-Filter 94
Bradford-Modell 274
breadth-first 198
Brechungsindex 376
Brennweite 8
Byte 22
`byte` (Typ) 437
`ByteBlitter` (Klasse) 85, 467
`ByteProcessor` (Klasse) 83, 246, 448

C

camera obscura 7
Canny-Operator 126
card 40, 431
CCITT 15
`Cdf` (Methode) 71
`ceil` (Methode) 438
`centralMoment` (Methode) 225
CGM-Format 15
chain code 215, 220
chamfer matching 426
Chamfer-Algorithmus 424
Chroma 265
chromatische Adaptierung 273
CIE 271
  Farbdiagramm 271, 272
  L*a*b* 270, 276, 277
  XYZ .. 271, 276, 277, 279–281, 285, 287
circularity 220
city block distance 423
clamping 56, 84, 455
clipping rectangle 463
`clone` (Methode) 288, 398
`close` (Methode) 190
`closeAllWindows` (Methode) 476
Closing 182, 185, 189
CMYK 266–270
`collectCorners` (Methode) 149
`Collections` (Klasse) 149
`Color` (Klasse) 255–257, 284
color keying 262
`COLOR_RGB` (Konstante) 242
`ColorChooser` (Klasse) 449
`ColorModel` (Klasse) .. 243, 284, 454
`ColorProcessor` (Klasse) .. 239, 241, 245, 251, 288, 448
`ColorSpace` (Klasse) 284, 286
`compareTo` (Methode) 150
`Complex` (Klasse) 327
`computeMatch` (Methode) .. 420, 422
Computer Vision 3
Computergrafik 2

concat (Methode) 399
connected components problem ... 205
Container 149
contains (Methode) 471
convertHSBToRGB (Methode) ... 247, 458
convertRGBStackToRGB (Methode) 458
convertRGBtoIndexedColor (Methode) 247, 458
convertToByte (Methode) .. 85, 248, 251, 457, 467
convertToFloat (Methode) 248, 457
convertToGray16 (Methode) ... 247, 458
convertToGray32 (Methode) ... 247, 458
convertToGray8 (Methode) 247, 458
convertToHSB (Methode) .. 247, 458
convertToRGB (Methode) .. 247, 248, 457, 458
convertToRGBStack (Methode) . 458
convertToShort (Methode) 248, 457
convex hull 221
convolution 101
convolve (Methode) 148, 461
convolve3x3 (Methode) 461
Convolver (Klasse) 114, 148
COPY (Konstante) 460
COPY_INVERTED (Konstante) 460
copyBits (Methode) 83, 85, 189, 190, 460, 467
Corner (Klasse) 148, 149
corner detection 139–153
corner response function 141, 145
CorrCoeffMatcher (Klasse) 420, 421
cos (Methode) 438
countColors (Methode) 288
createByteImage (Methode) ... 451, 467
createEmptyStack (Methode) ... 451
createFloatImage (Methode) ... 452
createImage (Methode) 452
createProcessor (Methode) ... 189, 452
createRGBImage (Methode) 452
createShortImage (Methode) ... 451
crop (Methode) 461
CRT 234
CS_CIEXYZ (Konstante) 285
CS_GRAY (Konstante) 285
CS_LINEAR_RGB (Konstante) 285
CS_sRGB (Konstante) 285
cumulative distribution function .. 66

D

Dateiformat 14–24
BMP 21
EXIF 19
GIF 16
JFIF 18
JPEG-2000 19
magic number 24
PBM 21
Photoshop 24
PNG 17
RAS 22
RGB 22
TGA 22
TIFF 15–16
XBM/XPM 22
DCT 355–361
eindimensional 355, 356
zweidimensional 358
DCT (Methode) 358
deconvolution 354
deleteLastSlice (Methode) 464
deleteSlice (Methode) 464
Delta-Funktion 311
depth-first 198
Desaturierung 251
DFT 317–354, 431
eindimensional 317–328, 332
zweidimensional 331–354
DFT (Methode) 327
DICOM 31
DIFFERENCE (Konstante) 84, 460
Differenzfilter 100
Digitalbild 9
dilate (Methode) 188, 190, 461
Dilation 175, 184, 187
Dirac-Funktion 105, 311
DirectColorModel (Klasse) 454
diskrete Fouriertransformation 317–354, 431
diskrete Kosinustransformation 355–361
Distanz 413, 418, 419
euklidische 423
Manhattan 423
-maske 424

-transformation 423
DIVIDE (Konstante) 84, 460
DOES_16 (Konstante) 475
DOES_32 (Konstante) 475
DOES_8C (Konstante) .. 243, 244, 475
DOES_8G (Konstante) 33, 47, 475
DOES_ALL (Konstante) 475
DOES_RGB (Konstante) . 240, 241, 475
DOES_STACKS (Konstante) 475
DONE (Konstante) 475
dots per inch 11, 325
double (Typ) 95, 436
dpi 325
draw (Methode) 151, 152
drawDot (Methode) 462, 463
drawLine (Methode) 152, 462
drawPixel (Methode) 462
drawRect (Methode) 462
drawString (Methode) 462
Druckraster 325, 351
DST 355
dünne Linse 8
duplicate (Methode) ..96, 111, 398, 452, 467
Durchmesser 221
DXF-Format 15
Dynamik 42

E

E (Konstante) 438
Eckpunkte 139–153
Eclipse 34, 446
edge map 129, 155
effektiver Gammawert 80
Eigenwert 141
Einheitskreis 302
Einheitsquadrat 372, 373
elongation 226
EMF-Format 15
Encapsulated PostScript 15
EPS-Format 15
erode (Methode) 189, 190, 461
Erosion 176, 184, 187
error (Methode) 473
euklidischer Abstand ... 413, 419, 423
Euler-Notation 302
Euler-Zahl 229
EXIF 279
exp (Methode) 438
Exzentrizität 226, 231

F

Faltung . 101, 310, 351, 352, 381, 414, 431, 434, 461
Farbbild 13, 233–298
Farbdifferenz 278
Farbmanagement 285
Farbpixel 236, 238, 239
Farbquantisierung ... 16, 45, 289–297
  3:3:2 292
  Median-Cut 294
  Octree 295
  Populosity 294
Farbraum 248–287
  CMYK 266
  colorimetrischer 270
  HLS 253
  HSB 253
  HSV 253
  Java 283
  L*a*b* 276
  RGB 234
  sRGB 278
  XYZ 271
  YC_bC_r 265
  YIQ 265
  YUV 263
Farbsystem
  additives 233
  subtraktives 266
Farbtabelle 242, 243
Fast Fourier Transform 328, 434
Faxkodierung 215
feature 218
Fensterfunktion 342–345
  Bartlett 343, 345, 346
  elliptische 343, 344
  Gauß 343, 344, 346
  Hanning 343, 345, 346
  Kosinus^2 ..... 345, 346
  Lanczos 385
  Parzen 343, 345, 346
  Supergauß 343, 344
FFT 328, 333, 351, 352, 357, 434
FileInfo (Klasse) 477
fill (Methode) 462
Filter 89–116
  Ableitungs- 120
  Berechnung 93
  Box- 94, 99, 461
  Differenz- 100
  Effizienz 112

Gauß- 100, 104, 115, 140, 144
Glättungs- 90, 94, 96, 99, 134
hot spot 92
im Spektralraum 351
Impulsantwort 105
in ImageJ 113–115, 461
Indexbild 242
inverses 353
isotropes 100
Kanten- 120–125, 461
-koordinaten 92
Laplace- 99, 116, 131, 136, 461
lineares 90–106, 113, 461
-maske 91
-matrix 91
Maximum- 107, 115, 461
Median- ... 108, 109, 115, 172, 461
Minimum- 107, 115, 461
morphologisches 111, 171–193
nichtlineares 106–112, 115
ortsabhängiges 395
Randproblem 93, 113
-region 90
Separierbarkeit 103
unscharfe Maskierung 132
`findCorners` (Methode) 152
`FindCorners_` (Plugin) 152
`findEdges` (Methode) 125, 461
FITS 31
Fläche
Polygon 220
Region 220
`flipHorizontal` (Methode) 461
`flipVertical` (Methode) 461
`FloatProcessor` (Klasse) .. 420, 448
`floatToIntBits` (Methode) 456
flood filling 196–200
`floor` (Methode) 438
Floor-Funktion 432
foreground 172
Formmerkmal 218
Fourier 303
-analyse 304
-deskriptor 217
-integral 304
-reihe 303
-spektrum 305, 317
-transformation 300–354, 431
-Transformationspaar 306, 308, 309
`FreehandRoi` (Klasse) 450, 469
`freeMemory` (Methode) 477
Frequenz 301, 325
zweidimensional 337
`fromCIEXYZ` (Methode) 286
`fromRGB` (Methode) 287

G

`gamma` (Methode) 83, 459
Gammakorrektur ... 74–81, 280, 283, 285, 459
Anwendung 77
inverse 75, 80
modifizierte 78–81
Gammawert 80
Gamut 275, 278
garbage 440
Garbage Collector 478
Gauß
Algorithmus für lineare Gleichungssysteme 369
-Fenster 343, 344, 346
-Filter 115
Flächenformel 220
-funktion 306, 309
`gc` (Methode) 478
`GenericDialog` (Klasse) 87, 449, 466, 473, 474
geometrische Operation 363–410, 461
gepackte Anordnung 236–238
Gerade 156, 159
`get2dHistogram` (Methode) 290
`getBitDepth` (Methode) 242
`getBlues` (Methode) 244, 245
`getColorModel` (Methode) . 244, 454
`getColumn` (Methode) 455
`getCurrentImage` (Methode) ... 244, 476
`getCurrentWindow` (Methode) ... 476
`getGreens` (Methode) 244, 245
`getHeight` (Methode) .. 33, 454, 464
`getHistogram` (Methode) 47, 64, 70, 458
`getIDList` (Methode) 87, 476
`getImage` (Methode) ... 87, 473, 476, 477
`getImageArray` (Methode) 465
`getInstance` (Methode) 285, 287
`getInterpolatedPixel` (Methode) 406, 407, 455
`getInverse` (Methode) 397
`getLine` (Methode) 455
`getMapSize` (Methode) 244

getMask (Methode) 469, 470
getMatchValue (Methode) 422
getNextChoiceIndex (Methode) . 87
getNextNumber (Methode) 87
getNumber (Methode) 473
getOriginalFileInfo (Methode) 477
getPixel (Methode) 33, 56, 64, 111, 112, 239, 438, 454
getPixels (Methode) . 455, 456, 465
getPixelsCopy (Methode) 455
getPixelSize (Methode) 244
getPixelValue (Methode) 454
getProcessor (Methode) .. 465, 467, 477
getProperties (Methode) 471
getProperty (Methode) 471, 472
getReds (Methode) 244, 245
getRoi (Methode) 469, 470
getRow (Methode) 455
getShortTitle (Methode) .. 87, 463
getSize (Methode) 465
getSliceLabel (Methode) 465
getStack (Methode) 464, 467
getStackSize (Methode) 464
getString (Methode) 473
getStringWidth (Methode) 462
getTitle (Methode) 463
getType (Methode) 242
getWidth (Methode) ... 33, 454, 465
getWindowCount (Methode) 477
GIF 16, 24, 31, 45, 215, 237, 242
Glättungsfilter 91, 94
Gleitkomma-Bild 13
Gradient 118, 119, 140, 144
Graph 205
Grauwertbild 12, 17, 249, 281
Grundfrequenz 325

H

Hadamard-Transformation 360
Hanning-Fenster .. 342, 343, 345, 346
Harris-Detektor 140
HarrisCornerDet (Klasse) . 147, 152
Häufigkeitsverteilung 65
Hauptachse 224
HDTV 266
Helligkeit 56
hermitesche Funktion 307
Hertz 301, 325
Hesse'sche Normalform 159
Hexadezimalnotation 239
hide (Methode) 463
hideProgress (Methode) 473
hierarchische Methoden 126
Hintergrund 172
histogram specification 65
Histogramm 39–53, 288–289, 432
-anpassung 65–73
-ausgleich 61–63
Berechnung 46
binning 48
Farbbild 49
gleichverteiltes 62
kumulatives 52, 60, 62, 66
normalisiertes 65
HLS 253, 258–260, 264
HLStoRGB (Methode) 262
homogene Koordinate 365, 398
homogene Punktoperation 55
hot spot 92, 174
Hough-Transformation . 129, 156–170
bias 165
für Ellipsen 168
für Geraden 156, 165
für Kreise 167
hierarchische 166
Kantenstärke 166
verallgemeinerte 169
HSB ..... *siehe* HSV
HSBtoRGB (Methode) 257, 258
HSV 250, 253, 258, 260, 264
Huffman-Kodierung 18

I

i 302, 432
ICC-Profil 285
ICC_ColorSpace (Klasse) 286
ICC_Profile (Klasse) 286
iDCT (Methode) 358
idempotent 182
IJ (Klasse) 450, 477
ij (Package) 450
ij.gui (Package) 449, 469
ij.io (Package) 450
ij.plugin (Package) 448
ij.plugin.filter (Package) ... 448
ij.process (Package) 447
Image (Klasse) 452
ImageCanvas (Klasse) 449
ImageConverter (Klasse) .. 247, 458
ImageJ 27–37

API ... 447–450
Bilder konvertieren ... 457
Filter ... 113–115, 461
geometrische Operation .. 395, 461
grafische Operation ... 462
GUI ... 449
Histogramm ... 458
Homepage ... 35
Installation ... 445
Makros ... 30, 34
Menü ... 30
Plugin ... 31–35, 474
Programmstruktur ... 31
Punktoperationen ... 81–88, 459
ROI ... 469
Stack ... 30, 464
Tutorial ... 35
Undo ... 31
Zugriff auf Pixel ... 454
`ImagePlus` (Klasse) ... 152, 241, 245, 246, 447, 470, 473, 477
`ImageProcessor` (Klasse) ... 32, 190, 240, 241, 243–248, 252, 447
`ImageStack` (Klasse) .. 447, 464, 467, 469
`ImageStatistics` (Klasse) ... 459
`ImageWindow` (Klasse) ... 449
Impulsantwort ... 105, 180
Impulsfunktion ... 105, 311
in place ... 333
Indexbild ... 13, 17, 237, 242
`IndexColorModel` (Klasse) . 244–246, 454
`insert` (Methode) ... 456, 462, 467
`int` (Typ) ... 437
`intBitsToFloat` (Methode) ... 454
Intensitätsbild ... 12
interest point ... 139
Interpolation ... 364, 379–395
bikubische ... 389, 392, 394
bilineare ... 387, 392, 455, 462
durch Faltung ... 383
ideale ... 380
kubische ... 383
Lanczos ... 385, 390, 410
lineare ... 380
Nearest-Neighbor ... 380, 387, 392, 394, 462
zweidimensionale ... 386–392
Interpolationskern ... 383
Invarianz 220, 223, 224, 227, 229, 428
inverses Filter ... 353
`invert` (Methode) ... 83, 397, 399
inverter-Plugin ... 32
Invertieren ... 57
`invertLut` (Methode) ... 187
`isEmpty` (Methode) ... 199
`isLocalMax` (Methode) ... 149
`isMacintosh` (Methode) ... 477
`isMacOSX` (Methode) ... 478
isotrop ... 100, 119, 140, 153, 178
`isWindows` (Methode) ... 478
`iterator` (Methode) ... 151
ITU601 ... 265
ITU709 ... 76, 81, 249, 266, 279

J

`Jampack` (Package) ... 369, 404
Java
Applet ... 29
Array ... 441
AWT ... 31
class File ... 34
Compiler ... 34
Editor ... 446
Integer-Division ... 64
Runtime Environment ... 29, 34, 445, 446
`static`-Block ... 478
JBuilder ... 34, 446
JFIF ... 18, 22, 24
JPEG 15, 17–22, 24, 31, 45, 215, 237, 281, 298, 357
JPEG-2000 ... 19

K

Kammfunktion ... 313
Kantenbild ... 129, 155
Kantenoperator
Canny ... 126, 128
in ImageJ ... 125
Kirsch ... 124
Kompass ... 124
LoG ... 126, 128
Prewitt ... 120, 128
Roberts ... 123, 128
Sobel ... 120, 125, 128
Kantenschärfung ... 129–136
Kardinalität ... 431
`killRoi` (Methode) ... 470
Kirsch-Operator ... 124
Koeffizientenmatrix ... 93

Normalisierung 95
Kollision 203
Kompaktheit 220
komplexe Zahl 302, 433
Komplexität 434
Komponentenanordnung 236
Komponentenhistogramm 50
Kontrast 42, 56
automatische Anpassung 59
Kontur 128, 206–213
konvexe Hülle 221, 230
Konvexität 221, 229
Koordinate
homogene 365, 398
kartesische 366
Koordinatentransformation 364
Korrelation 352, 414
Korrelationskoeffizient 415, 418
Kosinus^2-Fenster 345, 346
Kosinusfunktion 308
eindimensional 300
zweidimensional 334, 335
Kosinustransformation 18, 355
Kreisförmigkeit 220
Kreisfrequenz 301, 325, 356
Kreuzkorrelation 414–416, 418
kumulatives Histogramm . . 52, 60, 66

L

L*a*b* 276
`Lab_ColorSpace` (Klasse) 287
label 196
Lanczos-Interpolation . . 385, 390, 410
Laplace
-Filter 131, 136
-Operator 130
Lauflängenkodierung 214
Leistungsspektrum 326, 333
`Line` (Klasse) 450, 469, 470
lineares Gleichungssystem 369
Linearität 307
`LinearMapping` (Klasse) 398, 401
lines per inch 11
`lineTo` (Methode) 462, 463
`LinkedList` (Klasse) 200
Linse 8
little-endian 22, 24
Lochkamera 5
`lock` (Methode) 477
`lockSilently` (Methode) 477
`log` (Methode) 83, 438, 459, 473
lokale Abbildung 376
lookup table 81
LSB 23
Luminanz 249, 265
-histogramm 49
-wert 454
LZW 15, 16

M

magic number 24
`makeColumnVector` (Methode) . . . 404
`makeIndexColorImage` (Methode) 246
`makeInverseMapping` (Methode) 405
`makeMapping` (Methode) 400, 402
Manhattan-Distanz 423
`Mapping` (Klasse) 397
Maske 214
`matchHistograms` (Methode) . 69, 70
`Math` (Klasse) 438
`MAX` (Konstante) 84, 115, 460
`max` (Methode) 83, 438, 459
Maximalfrequenz 338
`MEDIAN` (Konstante) 115
Median-Cut-Algorithmus 294
Medianfilter 108, 172
gewichtet 109
`medianFilter` (Methode) 461
Mesh-Partitionierung 376
`MIN` (Konstante) 84, 115, 460
`min` (Methode) 83, 438, 459
`Mod` (Methode) 437
mod-Operator 327, 432
Moment 215, 222–227
Hu 227, 231
invariant 227
zentrales 223
`moment` (Methode) 225
moment invariants 227
Morphing 376
Morphologisches Filter 171–193
Closing 182, 185, 189
Dilation 175, 184, 187
Erosion 176, 184, 187
für Grauwertbilder 182
Opening 179, 185, 189
Outline 178, 190
`moveTo` (Methode) 463
MSB 23
`MULTIPLY` (Konstante) 84, 460
`multiply` (Methode) . . . 83, 459, 467

Mustererkennung 3, 218
MyInverter_ (Plugin) 33

N

Nachbarschaft 196, 219
  4er- 174
  8er- 174
NaN (Konstante) 440
NearestNeighborInterpolator (Klasse) 406
NEGATIVE_INFINITY (Konstante) 440
NetBeans 34, 446
NewImage (Klasse) 449, 451, 467
NIH-Image 29
NO_CHANGES (Konstante) 47, 245, 475
NO_IMAGE_REQUIRED (Konstante) 475
NO_UNDO (Konstante) 475
Node (Klasse) 198
noImage (Methode) 87
noise (Methode) 459
nomineller Gammawert 80
non-maximum suppression 164
normalCentralMoment (Methode) 225
Normbeleuchtung 273
NTSC 263, 265
null (Konstante) 440
Nyquist 316

O

$\mathcal{O}$-Notation 434
OCR 218, 229
Octree-Algorithmus 295
opak 84
open (Methode) 190
Opening 179, 185, 189
optische Achse 7
OR (Konstante) 84, 460
or (Methode) 459
Orientierung 224, 337, 338, 349
orthogonal 361
Ortsraum 306, 324, 351, 361
Outline 178, 190
outline (Methode) 190, 191
OvalRoi (Klasse) 450, 469, 470

P

PAL 263
Palettenbild ..... 13, *siehe* Indexbild
Parameterraum 157
Parzen-Fenster 342, 343, 345, 346
Pattern Recognition 3, 218
PDF 15
perimeter 219
Periodenlänge 300
Periodizität 336, 340
perspektivische Abbildung 7, 168
Phase 301, 326
Phasenwinkel 302, 303
Photoshop 24
PI (Konstante) 438, 439
PICT-Format 15
pixel 5
PixelInterpolator (Klasse) 406
Pixelwerte 12, 432
planare Anordnung 236
Plessey-Detektor 140
PlugIn (Interface) 32, 448, 474
PlugInFilter (Interface) 32, 240, 448, 474
PNG 17, 24, 31, 241, 242, 279
Pnt2d (Klasse) 396
point spread function 106
PolygonRoi (Klasse) 450, 469, 470
pop (Methode) 199
Populosity-Algorithmus 294
POSITIVE_INFINITY (Konstante) 440
PostScript 15
pow (Methode) 79, 438
power spectrum 326
Prewitt-Operator 120
Primärfarbe 235
probability density function 65
ProjectiveMapping (Klasse) 401, 409
Projektion 228, 231
projektive Abbildung 364, 367–372, 374
Properties (Klasse) 471
Punktmenge 174
Punktoperation 55–88
  Gammakorrektur 74, 459
  Histogrammausgleich 61
  homogene 55
  in ImageJ 81–88, 459
  Invertieren 57
  Schwellwertoperation 57
  und Histogramm 58
push (Methode) 199
putBehind (Methode) 477
putColumn (Methode) 456

putPixel (Methode) . 33, 34, 56, 64, 96, 111, 112, 239, 455, 456
putPixelValue (Methode) . 422, 456
putRow (Methode) 456
Pyramidentechniken 126

Q

quadrilateral 372
Quantisierung 10, 57, 289–297
  skalare 292
  Vektor- 293

R

random (Methode) 438
rank (Methode) 115
RankFilters (Klasse) 115
RAS-Format 22
Rasterbild 15
RAW-Format 241
Rechteckfenster 344, 346
Rechteckpuls 307
Region 195–231
  Durchmesser 221
  Exzentrizität 226
  Fläche 220, 224, 230
  Hauptachse 224
  konvexe Hülle 221
  Lauflängenkodierung 214
  Markierung 196–206
  Matrix-Repräsentation 214
  Moment 222
  Orientierung 224
  Projektion 228
  Schwerpunkt 222, 231
  Topologie 229
  Umfang 219, 220
region labeling 196, 200
register (Methode) 478
removeLast (Methode) 199
Resampling 377
resize (Methode) 461
resolution 11
RGB 264
RGB-Format 22
RGBtoHLS (Methode) 261
RGBtoHSB (Methode) 255, 256
rint (Methode) 438
Ripple-Abbildung 375
Roberts-Operator 123
Roi (Klasse) 450, 469, 470
ROI_REQUIRED (Konstante) 475
rotateLeft (Methode) 461
rotateRight (Methode) 461
Rotation 227, 349, 363, 365
Rotation (Klasse) 401, 408
round (Methode) 79, 96, 438
Round-Funktion 432
run (Methode) 32, 474, 475
run length encoding 214
runden 84, 439
Rundheit 220
runPlugIn (Methode) 476

S

sampling 311
Sättigung 44
scale (Methode) 462
Scaling (Klasse) 401
Scherung 365
Schwellwertoperation . . . 57, 129, 163, 460
Schwerpunkt 222
Separierbarkeit 103, 332, 359
setAntialiasedText (Methode) 463
setClipRect (Methode) 463
setColor (Methode) 463
setColorModel (Methode) . 244, 246
setDoScaling (Methode) 458
setFont (Methode) 463
setInterpolate (Methode) 462
setJustification (Methode) . . . 463
setLineWidth (Methode) 463
setMask (Methode) 469
setNormalize (Methode) . . 114, 148
setPixels (Methode) 456, 465
setProcessor (Methode) 477
setProperty (Methode) 471, 472
setRoi (Methode) 469, 470
setSliceLabel (Methode) 465
setTempCurrentImage (Methode) 477
setTitle (Methode) 464
setup (Methode) . . . 32, 33, 240, 243, 475
setValue (Methode) 152, 463
Shah-Funktion 313
Shannon 316
shape number 217, 230
sharpen (Methode) 134, 461
Shear (Klasse) 401
short (Typ) 456
ShortProcessor (Klasse) . . 448, 456

`show` (Methode) .. 152, 241, 463, 467
`showMessage` (Methode) 473
`showMessageWithCancel` (Methode) 473
`showProgress` (Methode) 473
`showStatus` (Methode) 473
`sin` (Methode) 438
Sinc-Funktion 307, 381, 386
Sinusfunktion 300, 308
Sinustransformation 355
Skalierung 227, 363–365
`skeletonize` (Methode) 191
slice 464, 465
`smooth` (Methode) 461
Sobel-Operator 120, 461
Software 28
`Solve` (Klasse) 404
`sort` (Methode) 111, 149, 288
source-to-target mapping 378
spatial sampling 9
Spektralraum 306, 324, 351
Spektrum 299–361
Spezialbilder 13
sphärische Abbildung 375
`sqr` (Methode) 83, 459
`sqrt` (Methode) 83, 438, 460
sRGB 80, 81, 278, 280, 281, 283
Stack 198, 240, 464, 467, 469, 476
`Stack` (Klasse) 198, 199
`STACK_REQUIRED` (Konstante) 476
`static`-Block 478
Stauchung 364
Streckung 364
Strukturelement .. 174, 175, 177, 178, 183, 187
`SUBTRACT` (Konstante) 84, 460
Supergauß-Fenster 343, 344
`SUPPORTS_MASKING` (Konstante) . 476

T

`tan` (Methode) 438
target-to-source mapping 373, 378
template matching 411, 412, 421
temporal sampling 9
`TextRoi` (Klasse) 450, 469, 470
TGA-Format 22
thread 477, 478
`threshold` (Methode) 460
thresholding 57
TIFF 15, 19, 22, 24, 31, 215, 240, 242
`toArray` (Methode) 150
`toCIEXYZ` (Methode) 285, 287
`toDegrees` (Methode) 438
topologische Merkmale 229
`toRadians` (Methode) 438
`toRGB` (Methode) 287
tracking 139
Transformationspaar 306
`TransformJ` (Package) 395
Translation 227, 364, 365
`Translation` (Klasse) 401
Transparenz 84, 238, 246
Tristimuluswerte 279
Twirl-Abbildung 374
`TwirlMapping` (Klasse) 404
type cast 436
`TypeConverter` (Klasse) ... 247, 248, 251

U

Umfang 219, 220
undercolor-removal function 268
`unlock` (Methode) 477
unscharfe Maskierung 132–136
`UnsharpMask` (Klasse) 134
`unsigned byte` (Typ) 437
`updateAndDraw` (Methode) .. 36, 244, 464
`updateAndRepaintWindow` (Methode) 464

V

Varianz 416, 421
`Vector` (Klasse) 149, 440
Vektorgrafik 15
Verschiebungseigenschaft 310
Verteilungsfunktion 66
Viereck 372
Vollfarbenbild .. 13, 17, 235, 237, 238
Vordergrund 172

W

Wahrscheinlichkeitsverteilung . 65, 66
`wait` (Methode) 478
Walsh-Transformation 360
`wasCanceled` (Methode) 87
Wavelet 361
Website zu diesem Buch 35
Weißpunkt 253, 273
  D50 ..... 273, 277, 283
  D65 ..... 271, 273, 277, 279
Wellenzahl 332, 338, 356

windowed matching 420
windowing 340
`WindowManager` (Klasse) 87, 244, 450, 476
WMF-Format 15
`write` (Methode) 473

X

XBM/XPM-Format 22
`XOR` (Konstante) 461
`xor` (Methode) 460

Y

YC_bC_r 265
YIQ 265, 267
YUV 250, 263, 265, 267

Z

ZIP 15
Zufallsprozess 66
Zufallsvariable 66

Über die Autoren



Wilhelm Burger absolvierte ein MSc-Studium in *Computer Science* an der University of Utah (Salt Lake City) und erwarb sein Doktorat für Systemwissenschaften an der Johannes Kepler Universität in Linz. Als Postgraduate Researcher arbeitete er am Honeywell Systems & Research Center in Minneapolis und an der University of California in Riverside, vorwiegend in den Bereichen Visual Motion Analysis und autonome Navigation. Er leitete im Rahmen des nationalen Forschungsschwerpunkts „Digitale Bildverarbeitung“ ein Projekt zum Thema „Generische Objekterkennung“ und ist seit 1996 Leiter der Fachhochschul-Studiengänge *Medientechnik und -design* bzw. *Digitale Medien* in Hagenberg (Österreich). Privat schätzt der Autor großvolumige Fahrzeuge, Devotionalien und einen trockenen Veltliner.



Mark J. Burge erwarb einen Abschluss als BA an der Ohio Wesleyan University, ein MSc in Computer Science an der Ohio State University und sein Doktorat an der Johannes Kepler Universität in Linz. Er verbrachte mehrere Jahre als Forscher im Bereich Computer Vision an der ETH Zürich, wo er an einem Projekt zur automatischen Interpretation von Katastralkarten arbeitete. Als *Postdoc* an der Ohio State University war er am „Image Understanding and Interpretation Project“ des NASA Commercial Space Center beteiligt. Derzeit ist der Autor Program Director an der National Science Foundation (NSF) in Washington D.C. (USA) und beschäftigt sich u. a. mit biometrischen Verfahren, Software für Mobiltelefone und maschinellem Lernen. Privat ist Mark Burge Experte für klassische, italienische Espressomaschinen.

Über dieses Buch

Das vollständige Manuskript zu diesem Buch wurde von den Autoren druckfertig in LaTeX unter Verwendung von Donald Knuths Schriftfamilie *Computer Modern* erstellt. Besonders hilfreich waren dabei die Packages algorithmicx (von Szász János) zur Darstellung der Algorithmen, listings (von Carsten Heinz) für die Auflistung des Programmcodes und psfrag (von Michael C. Grant und David Carlisle) zur Textersetzung in Grafiken. Die meisten Illustrationen wurden mit *Macromedia Freehand* erstellt, mathematische Funktionen mit *Mathematica* und die Bilder mit *ImageJ* oder *Adobe Photoshop*. Alle Abbildungen des Buchs, Testbilder in Farbe und voller Auflösung sowie der Quellcode zu den Beispielen sind für Lehrzwecke auf der zugehörigen Website (**www.imagingbook.com**) verfügbar.