

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 lesen⁶ 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 Farbtabelle unterschiedlicher Größe mit 2...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. Mittlerweise sind die entsprechenden Patente jedoch abgelaufen und damit dürfte auch die Zukunft von GIF gesichert sein.

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 bzw. 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

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

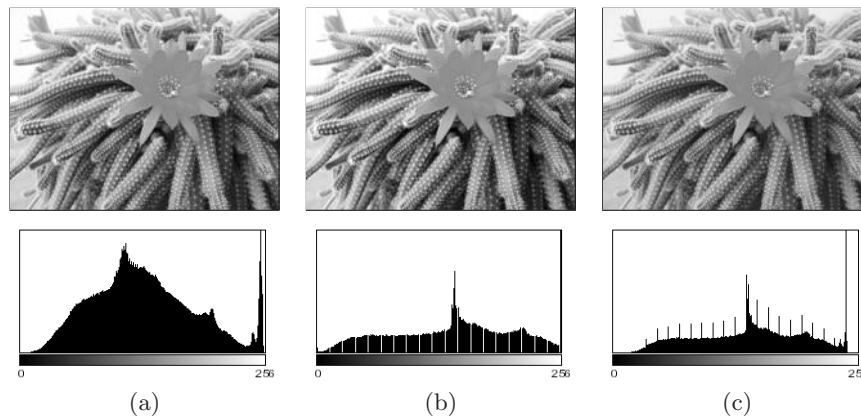


Abb. 4.9. Auswirkungen von Bildfehlern im Histogramm: Sättigungseffekt im Bereich der hohen Intensitäten (a), Histogrammlöcher verursacht durch eine geringfügige Kontrasterhöhung (b) und Histogrammspitzen aufgrund einer Kontrastreduktion (c).

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)).¹

Auswirkungen von Bildkompression

Bildveränderungen aufgrund von Bildkompression hinterlassen ebenfalls oft deutliche 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 ist also über das Histogramm relativ 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 Vollfarbgebild 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

¹ Leider erzeugen auch manche Aufnahmegeräte (vor allem einfache Scanner) derartige Fehler durch interne Kontrastanpassung („Optimierung“) der Bildqualität.

if ($p < 0$) $p = 0$;

auf den Minimalwert 0 begrenzen und damit verhindern, dass Pixelwerte negativ werden. Dieser Vorgang wird häufig als „Clamping“ bezeichnet.

5.1.3 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, q_{\min} und q_{\max} ist der aktuell kleinste bzw. größte Pixelwert in einem Bild $I(u, v)$, das über einen maximalen Intensitätsbereich $[p_{\min}, p_{\max}]$ verfügt. Um den gesamten Intensitätsbereich abzudecken, wird zunächst der kleinste Pixelwert q_{\min} auf den Minimalwert abgebildet und nachfolgend der Bildkontrast um den Faktor $(p_{\max} - p_{\min}) / (q_{\max} - q_{\min})$ erhöht (Abb. 5.1). Die Auto-Kontrast-Funktion ist daher definiert als

$$I'(u, v) \leftarrow (I(u, v) - q_{\min}) \cdot \frac{p_{\max} - p_{\min}}{q_{\max} - q_{\min}}, \quad (5.4)$$

vorausgesetzt natürlich $q_{\max} \neq q_{\min}$, d. h., das Bild muss mindestens zwei unterschiedliche Pixelwerte aufweisen. Für ein 8-Bit-Grauwertbild mit $p_{\max} = 255$ und $p_{\min} = 0$ vereinfacht sich diese Abbildung zu

$$I'(u, v) \leftarrow (I(u, v) - q_{\min}) \cdot \frac{255}{q_{\max} - q_{\min}}. \quad (5.5)$$

Der Bereich $[p_{\min}, p_{\max}]$ muss nicht dem maximalen Wertebereich entsprechen, sondern kann grundsätzlich ein beliebiger Kontrastbereich sein, den das Ergebnisbild abdecken soll. Natürlich funktioniert die Methode auch dann, wenn der Kontrast auf einen kleineren Bereich reduziert werden soll. Abb. 5.2 (b) zeigt die Auswirkungen einer Auto-Kontrast-Operation auf das zugehörige

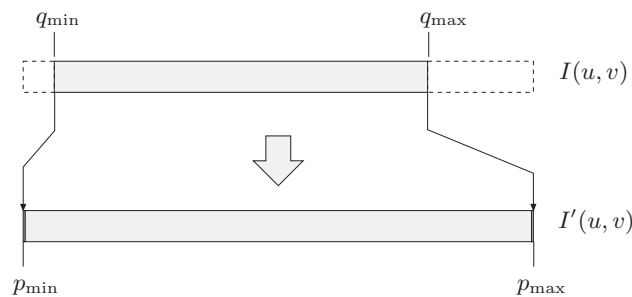


Abb. 5.1. Auto-Kontrast-Operation (Gl. 5.4).

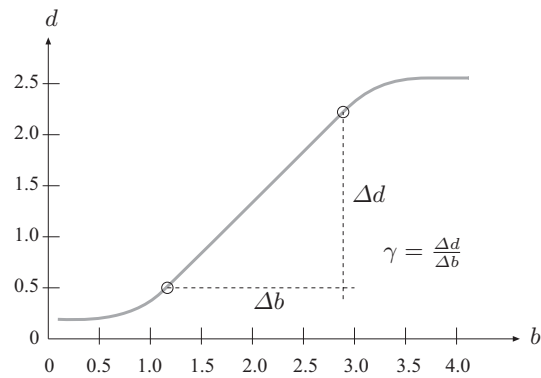


Abb. 5.11. Belichtungskurve von fotografischem Film. Bezogen auf die logarithmische Beleuchtungsstärke b verläuft die resultierende Dichte d in einem weiten Bereich annähernd als Gerade. Die Steilheit dieses linearen Anstiegs bezeichnet man als „Gamma“ (γ) des Filmmaterials.

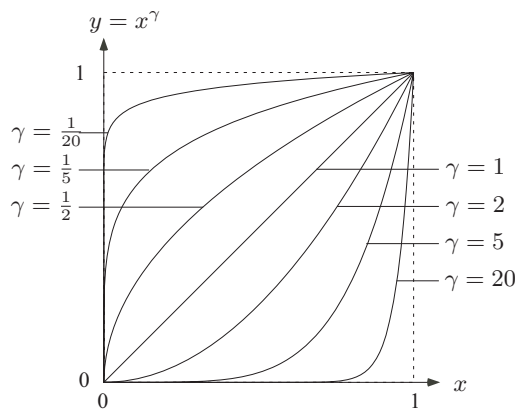


Abb. 5.12. Gammafunktion $y = x^\gamma$ im Bereich $x = 0 \dots 1$ für verschiedene Gammawerte.

$$y = f_\gamma(x) = x^\gamma \quad \text{für } x \in \mathbb{R}, \gamma > 0, \tag{5.10}$$

mit dem Parameter γ , dem so genannten *Gammawert*. Verwenden wir die Gammafunktion nur innerhalb des Bereichs $x = 0 \dots 1$, dann bleibt auch – unabhängig von γ – der Funktionswert x^γ im Bereich $0 \dots 1$ und die Funktion verläuft immer durch die Punkte $(0, 0)$ und $(1, 1)$. Wie Abb. 5.12 zeigt, ergibt sich für $\gamma = 1$ die identische Funktion $f_\gamma(x) = x$, 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

Tabelle 5.1. Parameter für einzelne Standard-Korrekturfunktionen auf Basis der Gammafunktion mit Offset gemäß Gl. 5.17–5.18. γ bezeichnet den nominellen und γ_{eff} den effektiven Gammawert.

Standard	γ	x_0	s	d	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	$1/2.200 \approx 0.455$

sein müssen, um eine kontinuierliche Gesamtfunktion zu erzeugen. Abb. 5.15 zeigt zur Illustration zwei Beispiele für die Funktion $\bar{f}_{(\gamma, x_0)}(x)$ mit den Werten $\gamma = 0.5$ bzw. $\gamma = 2.0$ und jeweils $x_0 = 0.2$.

In der Praxis sind für x_0 kleinere Werte üblich und γ muss so gewählt werden, dass die ideale Korrekturfunktion optimal angenähert wird. Beispielsweise gibt die in Abschn. 5.3.3 bereits erwähnte Spezifikation ITU-BT.709 [41] die Werte

$$\gamma = \frac{1}{2.222} \approx 0.45 \quad \text{und} \quad x_0 = 0.018$$

vor, woraus sich gemäß Gl. 5.18 die Werte $s = 4.50681$ bzw. $d = 0.0991499$ ergeben. Diese Korrekturfunktion $\bar{f}_{\text{ITU}}(x)$ mit dem nominellen Gammawert 0.45 entspricht einem *effektiven* Gammawert $\gamma_{\text{eff}} = 1/1.956 \approx 0.511$. Auch im sRGB-Standard [77] (siehe auch Abschn. 12.3.3) ist die Intensitätskorrektur auf dieser Basis spezifiziert. Die zugehörigen Parameter sind in Tabelle 5.1 zusammengefasst. Abb. 5.16 zeigt beide Korrekturfunktionen im Vergleich mit der entsprechenden gewöhnlichen Gammafunktion für den ITU- bzw. sRGB-Standard.

Inverse Korrektur

Um eine modifizierte Gammakorrektur der Form $y = \bar{f}_{(\gamma, x_0)}(x)$ (Gl. 5.17) rückgängig zu machen, benötigen wir die zugehörige inverse Funktion, d. h. $x = \bar{f}_{(\gamma, x_0)}^{-1}(y)$, die wiederum stückweise definiert ist:

$$\bar{f}_{(\gamma, x_0)}^{-1}(y) = \begin{cases} \frac{y}{s} & \text{für } 0 \leq y \leq s \cdot x_0 \\ \left(\frac{y+d}{1+d}\right)^{\frac{1}{\gamma}} & \text{für } s \cdot x_0 < y \leq 1 \end{cases} \quad (5.19)$$

Dabei sind s und d die Werte aus Gl. 5.18 und es gilt

$$x = \bar{f}_{(\gamma, x_0)}^{-1}(\bar{f}_{(\gamma, x_0)}(x)) \quad \text{für } x \in [0, 1], \quad (5.20)$$

wobei zu beachten ist, dass in beiden Funktionen der *gleiche* 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).

Tabelle 5.3. Modus-Konstanten für arithmetische Verknüpfungsoperationen der Interface-Klasse *Blitter* zur Anwendung mit der *ImageProcessor*-Methode *copyBits()*.

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

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:

```
ByteProcessor ip1 = ... //some byte image
ByteProcessor ip2 = ... //some other byte image
...
ip1.copyBits(ip2, 0, 0, Blitter.ADD); // ip1 = ip1 + ip2
...
```

Das Zielbild *dst* wird durch diese Operation modifiziert, das andere Bild *src* bleibt unverändert. Die Konstante *ADD* für den Modus ist – neben weiteren arithmetischen Operationen – in der Klasse *Blitter* definiert (Tabelle 5.3). Diese Operationen führen implizit auch eine Begrenzung der Werte (clamping) auf den maximalen Wertebereich durch. Bei allen Bildern – mit Ausnahme von Gleitkommabildern – werden die Ergebnisse nicht gerundet, sondern auf ganzzahlige Werte abgeschnitten. Daneben sind auch (bitweise) logische Operationen wie *OR* und *AND* vorgesehen (siehe auch Anhang C).

5.4.4 ImageJ-Plugins für mehrere Bilder

Plugins in ImageJ sind primär für die Bearbeitung einzelner Bilder ausgelegt, wobei das aktuelle (vom Benutzer ausgewählte) Bildobjekt I_1 vom Typ *ImageProcessor* (bzw. einer Subklasse) als Argument an die *run()*-Methode übergeben wird (s. Abschn. 3.2.3).

Sollen zwei (oder mehr) Bilder $I_1, I_2 \dots I_k$ miteinander verküpft werden, müssen die zusätzlichen Bilder $I_2 \dots I_k$ ebenfalls spezifiziert werden. Die übliche Vorgangsweise besteht darin, innerhalb des Plugins eine interaktive Auswahlmöglichkeit vorzusehen. Wir zeigen dies nachfolgend anhand eines Beispiel-Plugins, das zwei Bilder transparent überblendet.

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

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).³ 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  }

```

³ 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.

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.

```

1: REGIONLABELING( $I$ )
2:   Initialize  $m \leftarrow 2$  (the value of the next label to be assigned).
3:   Iterate over all image coordinates  $\langle u, v \rangle$ .
4:     if  $I(u, v) = 1$  then
5:       FLOODFILL( $I, u, v, m$ )           ▷ one of the versions below
6:       Increment  $m$ .
7:   return

8: FLOODFILL( $I, u, v, label$ )           ▷ Recursive Version
9:   if coordinate  $\langle u, v \rangle$  is within image boundaries and  $I(u, v) = 1$  then
10:    Set  $I(u, v) \leftarrow label$ 
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  $\langle u, v \rangle$  onto the stack: PUSH( $S, \langle u, v \rangle$ )
19:   while  $S$  is not empty do
20:     Get the next coordinate from the top of the stack:
21:      $\langle x, y \rangle \leftarrow POP(S)$ 
22:     if coordinate  $\langle x, y \rangle$  is within image boundaries and  $I(x, y) = 1$  then
23:       Set  $I(x, y) \leftarrow label$ 
24:       PUSH( $S, \langle x+1, y \rangle$ )
25:       PUSH( $S, \langle x, y+1 \rangle$ )
26:       PUSH( $S, \langle x, y-1 \rangle$ )
27:       PUSH( $S, \langle x-1, y \rangle$ )
28:   return

29: FLOODFILL( $I, u, v, label$ )           ▷ Breadth-First Version
30:   Create an empty queue  $Q$ 
31:   Insert the seed coordinate  $\langle u, v \rangle$  into the queue: ENQUEUE( $Q, \langle u, v \rangle$ )
32:   while  $Q$  is not empty do
33:     Get the next coordinate from the front of the queue:
34:      $\langle x, y \rangle \leftarrow DEQUEUE(Q)$ 
35:     if coordinate  $\langle x, y \rangle$  is within image boundaries and  $I(x, y) = 1$  then
36:       Set  $I(x, y) \leftarrow label$ 
37:       ENQUEUE( $Q, \langle x+1, y \rangle$ )
38:       ENQUEUE( $Q, \langle x, y+1 \rangle$ )
39:       ENQUEUE( $Q, \langle x, y-1 \rangle$ )
40:       ENQUEUE( $Q, \langle x-1, y \rangle$ )
41:   return

```

```

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 s = new Stack();
9   s.push(new Node(x,y));
10  while (!s.isEmpty()){
11    Node n = (Node) 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 q = new LinkedList(); // Queue
24   q.addFirst(new Node(x,y));
25   while (!q.isEmpty()) {
26     Node n = (Node) 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()`.

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*)

$$Circularity(\mathcal{R}) = 4\pi \cdot \frac{Area(\mathcal{R})}{Perimeter^2(\mathcal{R})}, \quad (11.9)$$

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 Umfangswerts aus Gl. 11.6, also

$$Circularity_{\text{corr}}(\mathcal{R}) = 4\pi \cdot \frac{Area(\mathcal{R})}{Perimeter_{\text{corr}}^2(\mathcal{R})}. \quad (11.10)$$

In Abb. 11.15 sind die Werte für die Kreisförmigkeit nach Gl. 11.9 bzw. 11.10 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:

$$BoundingBox(\mathcal{R}) = (u_{\min}, u_{\max}, v_{\min}, v_{\max}), \quad (11.11)$$

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)).

$I(u, v)$ – grundsätzlich auch für Regionen in Grauwertbildern anwendbar. Für zusammenhängende, binäre Regionen können Momente auch direkt aus den Koordinaten der Konturpunkte berechnet werden [71, 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. 11.13 reduziert auf

$$m_{pq} = \sum_{(u,v) \in \mathcal{R}} u^p v^q. \quad (11.14)$$

So kann etwa die **Fläche** einer binären Region als Moment nullter Ordnung in der Form

$$Area(\mathcal{R}) = |\mathcal{R}| = \sum_{(u,v) \in \mathcal{R}} 1 = \sum_{(u,v) \in \mathcal{R}} u^0 v^0 = m_{00}(\mathcal{R}) \quad (11.15)$$

ausgedrückt werden bzw. der **Schwerpunkt** \bar{x} (Gl. 11.12) als

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v) \in \mathcal{R}} u^1 v^0 = \frac{m_{10}(\mathcal{R})}{m_{00}(\mathcal{R})} \quad (11.16)$$

$$\bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v) \in \mathcal{R}} u^0 v^1 = \frac{m_{01}(\mathcal{R})}{m_{00}(\mathcal{R})} \quad (11.17)$$

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 $\bar{\mathbf{x}} = (\bar{x}, \bar{y})$ der Region und erhält dadurch die so genannten *zentralen* Momente der Ordnung p, q :

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v) \in \mathcal{R}} I(u, v) \cdot (u - \bar{x})^p \cdot (v - \bar{y})^q \quad (11.18)$$

Für Binärbilder (mit $I(u, v) = 1$ innerhalb der Region \mathcal{R}) reduziert sich Gl. 11.18 auf

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v) \in \mathcal{R}} (u - \bar{x})^p \cdot (v - \bar{y})^q. \quad (11.19)$$

```

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
18    static double centralMoment(ImageProcessor ip, int p, int q) {
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
35    static double normalCentralMoment
36        (ImageProcessor ip, int p, int q) {
37        double m00 = moment(ip, 0, 0);
38        double norm = Math.pow(m00, (double)(p + q + 2) / 2);
39        return centralMoment(ip, p, q) / norm;
40    }
41 }

```

Programm 11.2. 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. 11.14, 11.19, 11.21).

wobei meist die aus der Kodierung von analogen TV-Farbsignalen (s. auch Abschn. 12.2.4) bekannten Gewichte

$$w_R = 0.299 \quad w_G = 0.587 \quad w_B = 0.114 \quad (12.6)$$

bzw. die in ITU-BT.709 [41] für die digitale Farbkodierung empfohlenen Werte

$$w_R = 0.2125 \quad w_G = 0.7154 \quad w_B = 0.072 \quad (12.7)$$

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 [62] 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 (nichtlinearen) 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.

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. $Y = \text{Lum}(R, G, B)$ aus Gl. 12.5–12.6) zu ersetzen, d. h.

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} \leftarrow \begin{pmatrix} Y \\ Y \\ Y \end{pmatrix}. \quad (12.8)$$

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. 250). Die in ImageJ verwendete Gewichtung der RGB -Komponenten entspricht mit $(0.299, 0.587, 0.114)$ der in Gl. 12.6.

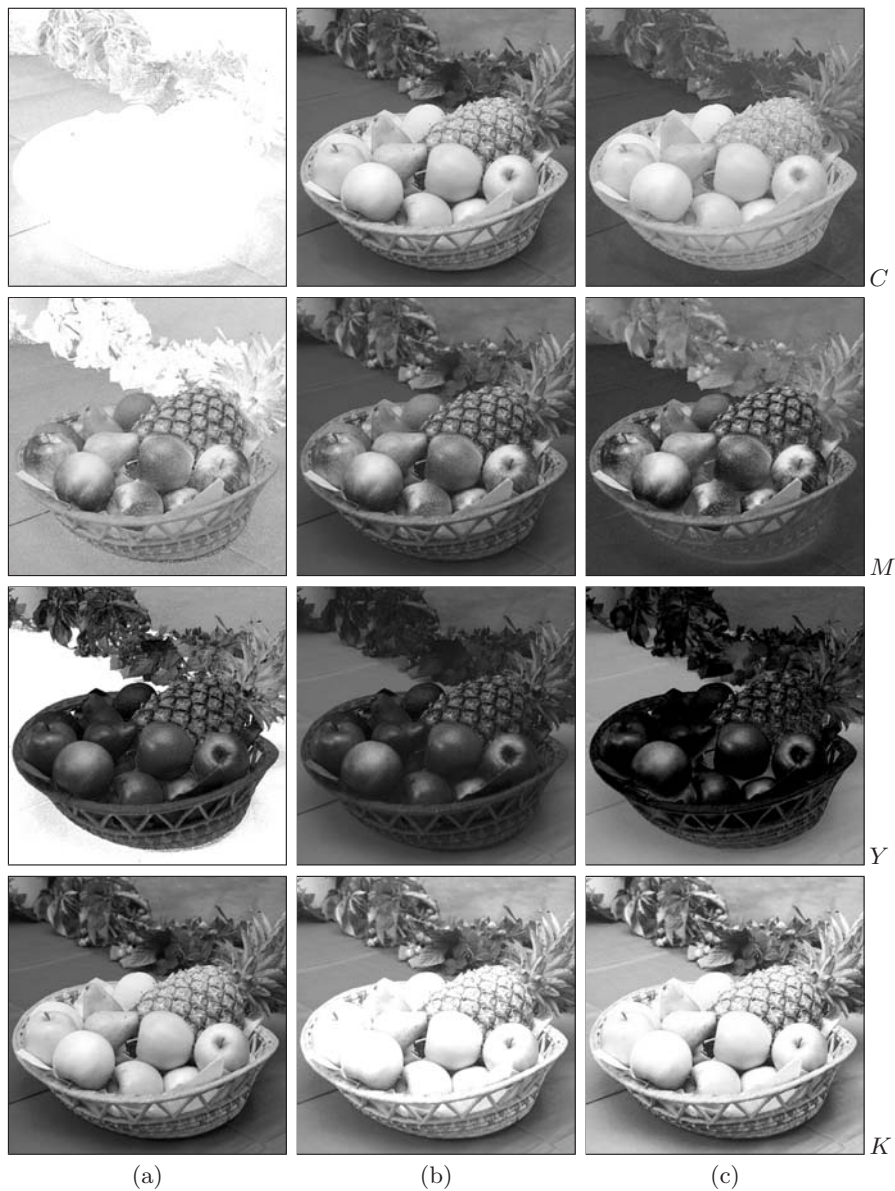


Abb. 12.19. RGB-CMYK-Konvertierung im Vergleich. Einfache Konvertierung nach Gl. 12.39 (a), Verwendung von *undercolor-removal*- und *black-generation*-Funktionen nach Gl. 12.40 (b), Ergebnis aus Adobe Photoshop (c). Die Farbinintensitäten sind invertiert dargestellt, dunkle Bildstellen entsprechen daher 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.

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).

Pkt.	Farbe	sRGB			CIEXYZ			CIE $L^*a^*b^*$		
		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
R₇₅	75% Rot	0.75	0.00	0.00	0.2155	0.1111	0.0101	39.76	64.52	54.14
R₅₀	50% Rot	0.50	0.00	0.00	0.0883	0.0455	0.0041	25.41	47.92	37.91
R₂₅	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

Transformation $L^*a^*b^* \rightarrow$ CIEXYZ

Die Rücktransformation von $L^*a^*b^*$ den CIEXYZ-Raum ist folgendermaßen definiert:

$$\begin{aligned}
 X &= X_{\text{ref}} \cdot f_2\left(\frac{a^*}{500} + Y'\right) \\
 Y &= Y_{\text{ref}} \cdot f_2(Y') \\
 Z &= Z_{\text{ref}} \cdot f_2\left(Y' - \frac{b^*}{200}\right)
 \end{aligned} \tag{12.52}$$

$$\text{wobei } Y' = \frac{L^*+16}{116}$$

$$\text{und } f_2(c) = \begin{cases} c^3 & \text{wenn } c^3 > 0.008856 \\ \frac{c-16/116}{7.787} & \text{wenn } c^3 \leq 0.008856 \end{cases}$$

Die vollständige Java-Implementierung dieser Konvertierung und einer entsprechenden Farbraum-Klasse (ColorSpace) sind in Prog. 12.10–12.11 (S. 293–294) 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 [29, S. 57]. Konkret ist die Berechnung der Distanz zwischen zwei Farbpunkten C_1 und C_2 hier einfach über den euklidischen Abstand möglich, d. h.

Genauso kann natürlich über den durch das ICC-Profil definierten Farbraum ein sRGB-Pixel in den Farbraum des Scanners oder des Monitors umgerechnet werden.

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 8-Bit-RGB-Farbbild potenziell $2^{24} = 16.777.216$ Farbwerte enthalten kann, wäre ein solches Histogramm-Array (mit immerhin 64 MBytes) 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 Farblöcke entspricht der Anzahl der Farben im Bild. Diese kann, wie in Prog. 12.12 gezeigt, einfach durch Abzählen der Übergänge zwischen den Farblöcken berechnet werden.

Natürlich wird in diesem Fall nicht das ursprüngliche Pixel-Array sortiert (das würde das Bild verändern), sondern eine Kopie des Pixel-Arrays.²² Diese Kopie wird mit der eigenen Methode `duplicateArray()` erzeugt, mit der beliebige, eindimensionale Java-Arrays dupliziert werden können.²³ 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 Farbkanaäle bzw. der Intensitätswerte beschränkt haben. Auch die ImageJ-Methode `getHistogram()` berechnet bei Anwendung auf Objekte der Klasse `ColorProcessor` in der Form

```
ColorProcessor cp;
int[] H = cp.getHistogram();
```

²² Alternativ könnte man auch mit der Methode `duplicate()` eine Kopie des `ImageProcessor`-Objekts anlegen.

²³ Das Duplizieren von Java-Arrays ist auch mit der Standardmethode `clone()` möglich, da die `Array`-Klasse das `Cloneable`-Interface implementiert.

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*

$$\omega = \frac{2\pi}{T} = 1. \quad (13.3)$$

Wenn wir die Funktion modifizieren in der Form

$$f(x) = \cos(3x), \quad (13.4)$$

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

$$T = \frac{2\pi}{\omega}, \quad (13.5)$$

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

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

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

$$f = \frac{1}{T} = \frac{\omega}{2\pi} \quad \text{bzw.} \quad \omega = 2\pi f, \quad (13.6)$$

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

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

dann ändert sich die *Phase* der Kosinusschwingung und φ bezeichnet den *Phasenwinkel* der resultierenden Funktion. Damit ist auch die Sinusfunktion (vgl.

¹ Beispielsweise entspricht die Frequenz $f = 1000$ Zyklen/s (Hertz) einer Periodenlänge von $T = 1/1000$ s und damit einer Kreisfrequenz von $\omega = 2000\pi$. Letztere ist eine einheitslose Größe.

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

$$z = a + i b \in \mathbb{C},$$

wobei i die imaginäre Einheit bezeichnet ($i^2 = -1$). Dieser Zusammenhang wird noch deutlicher, wenn wir die Euler'sche Notation einer beliebigen komplexen Zahlen z am Einheitskreis betrachten, nämlich

$$z = e^{i\theta} = \cos(\theta) + i \cdot \sin(\theta) \quad (13.10)$$

($e \approx 2.71828$ ist die Euler'sche Zahl). Betrachten wir den Ausdruck $e^{i\theta}$ als Funktion über θ , dann ergibt sich ein „komplexwertiges Sinusoid“, dessen reelle und imaginäre Komponente einer Kosinusfunktion bzw. einer Sinusfunktion entspricht, d. h.

$$\operatorname{Re}\{e^{i\theta}\} = \cos(\theta) \quad (13.11)$$

$$\operatorname{Im}\{e^{i\theta}\} = \sin(\theta)$$

Da $z = e^{i\theta}$ 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.

$$|a \cdot e^{i\theta}| = a \cdot |e^{i\theta}| = a. \quad (13.12)$$

Die *Phase* eines komplexwertigen Sinusoids wird durch Addition eines Phasenwinkels bzw. durch Multiplikation mit einer komplexwertigen Konstante $e^{i\varphi}$ am Einheitskreis verschoben,

$$e^{i(\theta+\varphi)} = e^{i\theta} \cdot e^{i\varphi}. \quad (13.13)$$

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 A.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

$$e^{i\theta} = e^{i\omega x} = \cos(\omega x) + i \cdot \sin(\omega x) \quad (13.14)$$

in *einem* funktionalen Ausdruck zusammenzufassen. Wir kommen auf diese Notation bei der Behandlung der Fouriertransformation in Abschn. 13.1.4 nochmals zurück.

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 ( $I$ )                                ▷ binary image  $I(u, v)$  of size  $M \times N$ 
   STEP 1 – INITIALIZE:
2:   Set up a distance map  $D(u, v) \in \mathbb{R}$  of size  $M \times N$ 
3:   for all image coordinates  $(u, v)$  do
4:     if  $I(u, v) = 1$  then
5:        $D(u, v) \leftarrow 0$                                 ▷ foreground pixel (zero distance)
6:     else
7:        $D(u, v) \leftarrow \infty$                             ▷ background pixel (infinite distance)
   STEP 2 – L→R PASS (using distance mask  $M^L = m_i^L$ ):
8:   for  $v \leftarrow 1, 2, \dots, N-1$  do                    ▷ top → bottom
9:     for  $u \leftarrow 1, 2, \dots, M-2$  do                    ▷ left → right
10:    if  $D(u, v) > 0$  then
11:       $d_1 \leftarrow m_1^L + D(u-1, v)$ 
12:       $d_2 \leftarrow m_2^L + D(u-1, v-1)$ 
13:       $d_3 \leftarrow m_3^L + D(u, v-1)$ 
14:       $d_4 \leftarrow m_4^L + D(u+1, v-1)$ 
15:       $D(u, v) \leftarrow \min(d_1, d_2, d_3, d_4)$ 
   STEP 3 – R→L PASS (using distance mask  $M^R = m_i^R$ ):
16:  for  $v \leftarrow N-2, \dots, 1, 0$  do                    ▷ bottom → top
17:    for  $u \leftarrow M-2, \dots, 2, 1$  do                    ▷ right → left
18:    if  $D(u, v) > 0$  then
19:       $d_1 \leftarrow m_1^R + D(u+1, v)$ 
20:       $d_2 \leftarrow m_2^R + D(u+1, v+1)$ 
21:       $d_3 \leftarrow m_3^R + D(u, v+1)$ 
22:       $d_4 \leftarrow m_4^R + D(u-1, v+1)$ 
23:       $D(u, v) \leftarrow \min(D(u, v), d_1, d_2, d_3, d_4)$ 
24:  return  $D$ 

```

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