Bildverarbeitung in R

- Ausarbeitung zum Vortrag vom 13.07.2016 -

Tobias Klinke

Proseminar R
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Betreuer: Jakob Lüttgau

Sommersemester 2016

Abstract

Die Programmiersprache R ist auf statistische Berechnungen und verschiedene Anwendungen in der Wissenschaft spezialisiert. Statistische Daten liegen häufig nicht nur als Text- sondern auch als Bilddaten vor. R bietet daher auch Pakete zur Bildverarbeitung an. Von denen sind viele auf spezielle Anwendungsgebiete - wie etwa Medizinische Bildverarbeitung - zugeschnitten und abstrahieren stark von der eigentlich zugrunde liegenden Bildverarbeitung.
Diese Arbeit behandelt die Grundlagen der Bildverarbeitung anhand einiger häufig verwendeter Algorithmen. Das Vorgehen besteht dabei immer aus zwei Teilen: Zunächst wird der zu betrachtende Algorithmus theoretisch mathematisch fundiert. Im Anschluss wird die konkrete Anwendung anhand eines Beispiels in R mit Hilfe des Paketes `EBImage` demonstriert.
Inhaltlich wird wie folgt verfahren: Nach einer kurzen Einführung in die Bildverarbeitung und Repräsentation digitaler Bilder werden Histogramme als wichtiges Werkzeug der Analyse von Bildern vorgestellt und im Anschluss wichtige Algorithmen der Bildverarbeitung. Diese Algorithmen werden getrennt in die Kategorien „Punktoperation“ und „Filter“ vorgestellt: Solche, die jedes einzelne Element eines Bildes ohne Betrachtung der Umgebung („Punktoperationen“) verändern und solche, die auch umgebende Elemente mit in die Berechnung einbeziehen („Filter“). Von letzteren werden strukturverändernde und nicht strukturverändernde betrachtet.

Inhaltsverzeichnis

1 Einleitung

1.1 Was ist Bildverarbeitung?

Für das Gebiet der Bildverarbeitung existiert keine einheitliche Definition, sondern es lässt sich am leichtesten durch Abgrenzung zu nahe verwandten Gebieten charakterisieren[1].

- *Computergrafik* beschäftigt sich mit der Erzeugung von Bildern, während Bildverarbeitung mit bereits vorhandenen Bildern arbeitet.
- *Bildbearbeitung* verwendet fertige Programme zur Veränderung von Bildern, Bildverarbeitung liefert die Algorithmen zur Entwicklung solcher Programme.
- *Bildanalyse* mit den Teildisziplinen *Computer vision* und *Pattern recognition* versucht aus Bildern Informationen zu gewinnen, um sich im 3-dimensionalen Raum orientieren zu können bzw. vorhandene Muster in Bildern wiederzuerkennen; Bildverarbeitung dient nicht primär der Informationgewinnung, sondern erzeugt aus Eingabebildern als Ausgabe wieder Bilder.

Zusammenfassend lässt sich sagen, dass Bildverarbeitung Algorithmen zur Verarbeitung von Pixeldaten entwickelt, die als Eingabe Bilder nehmen und als Ausgabe daraus wieder Bilder (oder einfache Metriken) erzeugen, ohne dabei Kontextwissen über die zu verarbeitenden Bilder mit einzubeziehen oder die Bild*inhalte* zu untersuchen.

1.2 Repräsentation digitaler Bilder

Es gibt verschiedene Möglichkeiten Bilder zu repräsentieren:
Zunächst ist ein Bild eine zweidimensionale Abbildung[2]:

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

bzw.

$$I : \mathbb{N} \times \mathbb{N} \to \mathbb{P}$$

Jedem Koordinatenpaar (u,v) wird ein Element aus einer Menge (bzw. Intervall) $\mathbb{P}$ zugeordnet. Die Auswahl der Menge $\mathbb{P}$ bestimmt die Farbtiefe des Bildes. Eine äquivalente Darstellung eines Bild ist eine Matrix von Zahlen.

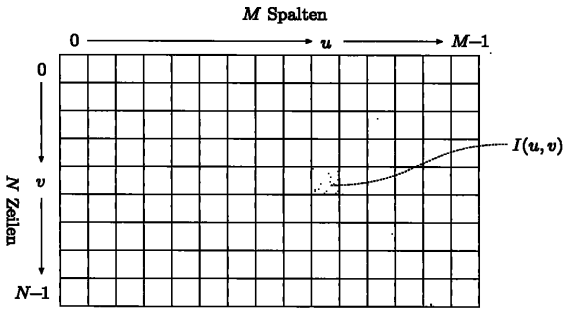

Abbildung 1: Matrix-Darstellung eines Bildes
Quelle: [Burge, 2006, S. 12]

Die Anzahl an Spalten bestimmt die Breite, die Anzahl an Zeilen die Höhe des Bildes.
Der Eintrag in der u-ten Spalte und v-ten Zeile ist dann $I(u,v)$ gemäß obiger Darstellung. Einen einzelnen Eintrag in dieser Matrix bezeichnet man als Pixel. Abbildung 1 zeigt diese Darstellung schematisch.
Für die Farbtiefe (also die Menge an möglichen Werte für Pixel) sind Farb-, Graustufen- und Binärbilder üblich:
Bei Farbbildern besteht jeder Pixel aus drei Farbkomponenten (Rot, Grün und Blau) zu je 8 Bit, d.h. $\mathbb{P} = [0,255]^3$. Graustufenbilder[3] haben nur eine Farbkomponente für Helligkeitswerte mit üblicherweise 8 Bit: $\mathbb{P} = [0,255]$, wobei 0 Schwarz und 255 Weiß darstellt. Binärbilder benötigen nur ein Bit pro Pixel ($\mathbb{P} = \{0,1\}$), wobei hier 0 Schwarz und 1 Weiß darstellt.

[1] vgl. [Burge, 2006, S. 2-4]
[2] Quelle Formel: [Burge, 2006, S. 10]
[3] im Folgenden kurz *Graubilder* genannt

1.3 Packages zur Bildverarbeitung

Die Pakete, die R für die Bildverarbeitung zur Verfügung stellt, lassen sich in drei Kategorien einteilen: Low-Level, allgemeine Bildverarbeitung und Spezialpakete.
Das Low-Level-Paket `readbitmap` liest Bilddaten ein und stellt sie als 2D-Array zur Vefügung und bietet keinerlei Bildverarbeitung. Das gleiche gilt für die Pakete `jpg`, `png`, `bmp` für die entsprechenden Formate. Diese können jedoch auch Bilder in den Formaten schreiben.
Pakete zur allgemeinen Bildverarbeitung sind `EBImage` und `adimpro`, wobei `adimpro` auf Filter (insbesondere Glättung) spezialisiert ist, aber auch allgemeine Funktionen anbietet.
Daneben existieren noch diverse Spezialpakete, etwa zur medizinischen Bildverarbeitung[4].
Alle nachfolgenden Beispiele verwenden ausschließlich `EBImage`, da es das am leichtesten allgemein anwendbare der vorgestellten Pakete ist. `EBImage` ist nicht auf CRAN verfügbar, sondern muss mit einem Skript von *Bioconductor*[5] installiert werden.

```
1 # Installation
2 source("https://bioconductor.org/biocLite.R")
3 biocLite("EBImage")
4
5 # Einbinden
6 library(EBImage)
```

Listing 1: Installation und Einbinden von EBImage

1.4 Einführung in `EBImage`

In `EBImage` werden Bilder durch Objekte vom Typ `Image` repräsentiert und die meisten der Bildverarbeitungsfunktionen erwarten Objekte dieses Typs als Argument. Image-Objekte sind im Prinzip 2-dimensionale Arrays, die die Pixeldaten enthalten. Die einzelnen Pixelwerte sind entgegen obiger Darstellung nicht wie üblich als Integer kodiert, sondern als `double` im Intervall $[0.0; 1.0]$, wobei im Fall von Graubildern 0.0 Schwarz und 1.0 Weiß darstellt. Farbbilder bestehen aus mehreren „Frames“. Da im folgenden fast ausschließlich Graubilder behandelt werden, wird auf die interne Darstellung von Farbbildern in `EBImage` nicht näher eingegangen.
Der allgemeine Ablauf, der allen weiteren Beispielen zugrunde liegt, besteht aus folgenden Teilen:

1. Einlesen des Eingabebildes mit `readImage()`
2. Verarbeitung des Bildes und Erzeugung des Ausgabebildes
3. Speichern/Anzeigen des Ausgabebildes mit `writeImage()` bzw. `display()`

1. und 3. ist dabei immer gleich und es wird im folgenden nur noch der 2. Schritt gezeigt. Die mitgelieferten R -Skripte sind jedoch vollständig und mit den Beispieldaten ausführbar.
Als vollständiges Beispiel für die grundlegende Struktur der Skripte dient ein Beispiel zur Konvertierung eines Farbbildes in ein Graubild. Diese Konvertierung ist daher wichtig, weil wir uns im Folgenden nur noch mit Algorithmen, die auf Graubildern arbeiten, beschäftigen. Die Eingabebilder liegen jedoch oft als Farbbilder vor und müssen vor der Verarbeitung konvertiert werden.

```
1 # 1. Bild aus Datei lesen
2 img <- readImage("inputimage.png")
3
4 # 2. Bild verarbeiten
5 outputImg <- channel(img, "luminance") # in Graustufenbild umwandeln
6
7 # 3. Bild speichern oder anzeigen
8 writeImage(outputImg, "outputimage.png") # bzw. display(outputImg)
```

Listing 2: Grundgerüst für die weiteren Beispiele

[4] siehe CRAN `https://cloud.r-project.org/web/views/MedicalImaging.html`
[5] `http://bioconductor.org/packages/release/bioc/html/EBImage.html`

Wie die Konvertierung erfolgt, ist später im Abschnitt zu den Punktoperationen beschrieben.

2 Histogramme

2.1 Grundlagen

Ein Histogramm eines (Graustufen-)Bildes zeigt die Häufigkeitsverteilung der einzelnen Intensitätswerte. Es lässt sich mit folgender Formel beschreiben[6]:

$$h(i) = \text{card}\{(u,v)|I(u,v) = i\}$$
$$\text{für alle } 0 \leq i < K \text{ mit } K = \text{card}\mathbb{P}$$

Es ist also eine Tabelle, die für jeden möglichen Intensitätswert i angibt, wie oft dieser im Bild vorkommt, also wie viele Pixel diese Farbe haben.
Grafisch werden Histogramme als 2-dimensionaler Plot dargestellt, wobei die untere Achse alle möglichen Helligkeitswerte und die linke Achse die Anzahl der Pixel mit dem jeweiligen Helligkeitswert darstellt.
In `R` kann ein Histogramm eines Bildes mit der Funktion `hist()` angezeigt werden. Dabei werden jedoch zunächst mehrere Grauwerte jeweils zu Gruppen zusammengefasst und zusammen im Histogramm dargestellt. Durch Angabe des optionalen Parameters `breaks=256` kann eine feinere Einteilung (hier: 8 Bit Genauigkeit, d.h. 256 verschiedene Werte) erreicht werden.
Aufgrund der internen Darstellung von `EBImage` zeigt die x-Achse Fließkommawerte im Bereich von 0.0 bis 1.0. Zur gewohnten Darstellung im Intervall von $[0, 255]$ kann das Bild vorher mit dem Faktor 256 skaliert werden (Beispiel dafür in Abbildung 2).

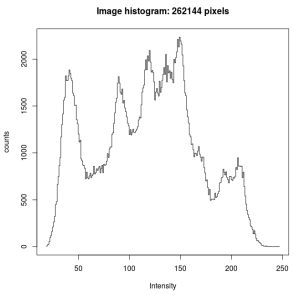

Abbildung 2: Beispiel-Histogramm von `hist()` erzeugt

```
1 hist(img) # Standard-Darstellung
2 hist(img, breaks=256) # feiner Einteilung
3 hist(img * 256, breaks=256) # feinere Einteilung und skaliert
```

Listing 3: Histogramme anzeigen

2.2 Belichtung im Histogramm ablesen

Im Histogramm lassen sich einige globale Eigenschaften eines Bildes ablesen, zum Beispiel die Belichtung (s. Beispielbilder in Abbildung 3):
So hat ein *unterbelichtetes* Bild auffällig hohe Werte im dunklen Bereich des Histogramms, während die rein weißen Anteile fast gar nicht vorhanden sind.
Histogramme *normal belichteter* Bilder folgen häufig einer Glockenkurve, bei der der gesamte Bereich ausgenutzt wird.
Bei *überbelichteten* Bildern findet man extreme Spitzen am rechten (= weißen) Rand des Histogramms, während die dunklen Anteile eher gering sind.

[6] nach [Burge, 2006, S. 40]

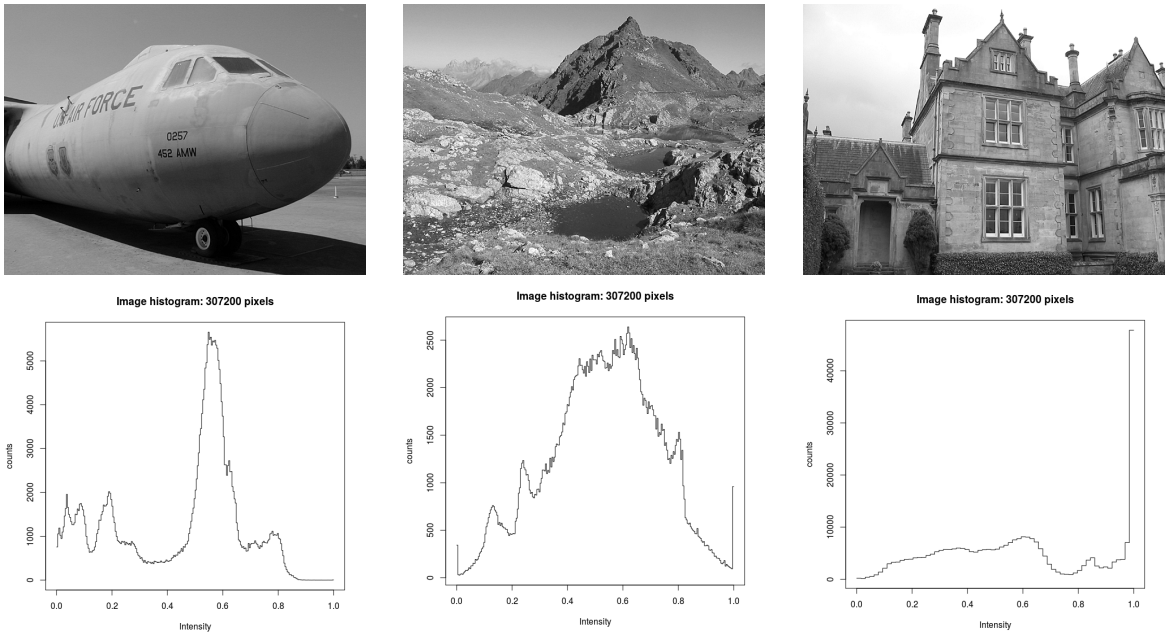

Abbildung 3: Unterbelichtetes, normal belichtetes und überbelichtetes Bild
Bildquelle: [Burge, 2006, S. 42]

2.3 Kontrastanpassung

Der *Kontrast* eines Bildes bezeichnet die Differenz zwischen minimalem und maximalem vorkommenden Grauwert. Wenn der Kontrast sehr niedrig ist, wird nur ein geringer Teil des möglichen Wertebereichs genutzt. Für optisch bessere Ergebnisse kann eine *Kontrastanpassung* vorgenommen werden. Dabei wird der tatsächlich genutzte Bereich auf den eigentlich möglichen skaliert, was einer Verbreiterung des Histogramms entspricht (s. Abbildung 4).
Die einzelnen Grauwerte a werden wie folgt abgebildet[7]:

$$f_{ac}(a) = (a - a_{low}) \cdot \frac{a_{max} - a_{min}}{a_{high} - a_{low}}$$

Dabei ist $a_{min/max}$ der entsprechend minimal / maximal *mögliche* Grauwert und $a_{low/high}$ der entsprechend minimal / maximal *vorkommende* Grauwert.

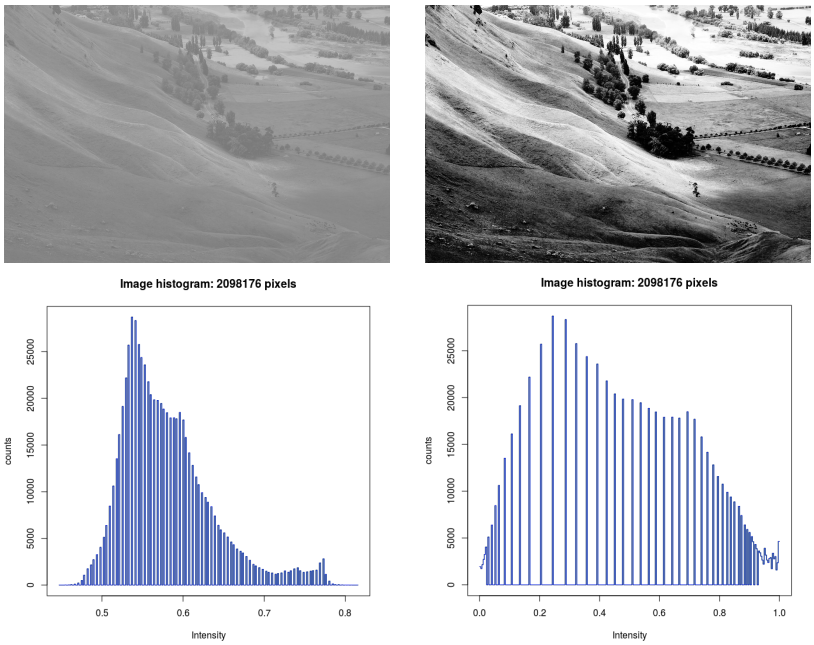

Abbildung 4: Bild und Histogramm vor (links) und nach (rechts) Kontrastanpassung
Quelle Originalbild: `https://en.wikipedia.org/wiki/File:Unequalized_Hawkes_Bay_NZ.jpg`

`EBImage` bietet für die Kontrastanpassung die Funktion `equalize`, die ein `Image`-Objekt als Argument hat und ein `Image` mit angepasstem Kontrast zurückgibt.

[7] Formel aus [Burge, 2006, S. 59]

```
1 equimg <- equalize(img)
```

Listing 4: Kontrastanpassung mit EBImage

Die Kontrastanpassung ist ein Beispiel für eine Punktoperation. Diese werden ausführlicher im nächsten Abschnitt behandelt.

3 Punktoperationen

3.1 Grundlagen

In diesem Abschnitt betrachten wir Abbildungen der folgenden Art[8]:

$$I'(u,v) = f(I(u,v))$$

wobei $f : \mathbb{P} \to \mathbb{P}$ ist.
Die Funktion f beschreibt eine Punktoperation, das heißt aus dem Bild I wird unter f das Bild I', wobei der neue Wert eines Pixel in I' nur vom vorherigen Wert des Pixels in I abhängt und *nicht* von benachbarten Pixeln.
Ferner ist diese Punktoperation - wie alle hier betrachteten Punktoperationen - *homogen*, d.h. f ist von den Koordinaten unabhängig.
In R können einige einfache Punktoperationen dadurch implementiert werden, dass man mit `Image`-Objekten fast genau so rechnen kann wie mit anderen 2-dimensionalen Arrays auch. Dabei muss man jedoch einige Besonderheiten beachten: Es findet kein automatisches Clamping der Werte auf den gültigen Wertebereich statt. D.h. es können beim Rechnen Werte außerhalb des Intervalls $[0.0; 1.0]$ entstehen, die also „mehr als weiß“ oder negativ sind und daher keine gültigen Farbwerte darstellen. Erst beim Anzeigen oder Speichern findet ein Clamping statt: Alle Werte größer als 1.0 werden zu 1.0 und alle Werte kleiner als 0.0 werden zu 0 begrenzt.

3.2 Einfache Punktoperationen

Folgende einfache Punktoperationen nutzen die Möglichkeit mit `Image`-Objekten wie mit Arrays zu rechnen.

Invertieren

Beim Invertieren wird ein Negativ des Bildes erzeugt. Die entsprechende Abbildung, die jeder Farbe ihr Negativ zuordnet, lautet[9]:

$$f(a) = a_{max} - a$$

Dabei ist a_{max} der maximale Intensitätswert (also Weiß).
Das Invertieren bedeutet eine vertikale Spiegelung des Histogramms in der Mitte[10]. In R wird zum Invertieren einfach das Bild von der Konstanten $1.0 = a_{max}$ abgezogen:

```
1 img_inverted <- 1.0 - img
```

Listing 5: Invertieren eines Bildes

[8] für folgenden Absatz vgl. [Burge, 2006, S. 55]
[9] Quelle: [Burge, 2006, S. 57]
[10] vgl. [Burge, 2006, S.58]

Aufhellen, Verdunkeln und Kontrast ändern

Ähnlich leicht wie das Invertieren umzusetzen sind die Punktoperationen zum Aufhellen/Verdunkeln eines Bildes sowie die Änderung des Kontrastes:[11]
Aufhellen erreicht die Funktion $f_{lighter}(a) = a + c$ mit $c > 0$, also die Addition einer Konstanten. Dabei werden alle entstehenden Werte größer als weiß zu weiß.
Verdunkeln ist entsprechend die Subtraktion einer Konstanten (wobei alle Werte kleiner als schwarz zu schwarz werden): $f_{darker}(a) = a - c$ mit $c > 0$
Der Kontrast eines Bildes lässt sich durch Multiplikation mit einer Konstanten ändern: $f_{contrast}(a) = a \cdot c$. Wenn $c > 1$ erhöht sich der Kontrast; wenn $0 < c < 1$ gilt, so wird der Kontrast veringert.

```
1 img_lighter <- img + 0.5       # Aufhellen
2 img_darker <- img - 0.4        # Verdunkeln
3 img_higher_contrast <- img * 1.5 # Kontrast erhoehen
```

Listing 6: Aufhellen / Verdunkeln und Kontrast ändern

3.3 Konvertierung Farbbild zu Graubild

Bei der bereits in der Einleitung als Beispiel erwähnten Konvertierung von Farb- zu Graubildern handelt es sich ebenfalls um eine Punktoperation.
Bei dieser Konvertierung wird aus den drei Farbwerten R, G, B (für Rot-, Grün-, und Blaukomponente) eines Pixels ein Grauwert für diesen Pixel berechnet.
Dafür bietet `EBImage` zwei Varianten an, die sich lediglich in der Gewichtung der einzelnen Farbkomponenten unterscheiden[12]:

- Die erste Methode (von `EBImage "gray"` genannt) gewichtet R, G, B im Durchschnitt: $gray = \frac{1}{3} \cdot (R + G + B)$
- Die Methode `"luminance"` gewichtet die Farben der menschlichen Wahrnehmung entsprechend unterschiedlich - Grün trägt am meisten zur wahrgenommenen Helligkeit bei, Blau am wenigsten: $gray = 0.2126 * R + 0.7152 * G + 0.0722 * B$

`"luminance"` erzeugt subjektiv ein wenig besser aussehende Ergebnisse.
Die entsprechende Funktion in `EBImage` heißt `channel`. Als ersten Parameter erwartet sie das Bild, als zweites den Modus als String. Dann wird das Graubild zurückgegeben.

```
1 gray_img <- channel(img, "gray")
2 luminance_img <- channel(img, "luminance")
```

Listing 7: Farbbild in Graustufen konvertieren

3.4 Thresholding - Graubild zu Binärbild

Thresholding (auch *Schwellenwertverfahren*) ist eine Punktoperation, die dazu dient, aus einem Graubild ein Binärbild zu machen.
Die Abbildungsfunktion ist dabei wie folgt definiert[13]:

$$f_{th}(a) = a_{min} \quad \text{für } a < a_{thr}$$
$$f_{th}(a) = a_{max} \quad \text{für } a \geq a_{thr}$$

Es wird ein *Schwellenwert* a_{thr} gewählt und alle Werte, die kleiner als dieser Schwellenwert sind, werden zu Schwarz, alle anderen zu Weiß.
In `R` wendet man dazu auf ein `Image` den `>`-Operator an:

[11] vgl. [Burge, 2006, S. 56f]
[12] Quelle der Formeln: [EBImage, 2016, Dokumentation der `channel()`-Funktion]
[13] [Burge, 2006, S. 57]

```
1 thresholded_image <- img > 0.5
2 # alle Werte > 0.5 werden zu weiss, alle anderen zu schwarz
```

Listing 8: Thresholding mit Schwellenwert von 0.5

Die Qualität des gewonnenenen Binärbildes hängt stark von der Wahl des Schwellenwertes ab. Zum einen kann man einen Schwellenwert anhand des Histogramms abschätzen, indem man ihn im „Tal“ zwischen zwei Spitzen order am „Fuß“ eines „Berges“ ansetzt. Eine genauere Bestimmung ist mit der nach ihrem Erfinder benannten „Otsu“-Methode möglich[14]. Die Idee des Otsu-Algorithmus besteht darin, die Pixel in zwei Klassen (für später zu weiß bzw. schwarz werdende Pixel) aufzuteilen. Die Aufteilung geschieht so, dass gleichzeitig die Varianz innerhalb der Klassen jeweils möglichst gering und die Varianz zwischen den Klassen maximal wird. Damit wird der optimale Schwellenwert erzielt.

`EBImage` bietet für den Otsu-Algorithmus die Funktion `otsu`, die als Parameter ein Bild erwartet und dafür den optimalen Schwellenwert zurück gibt.

```
1 thresholded_image <- img > otsu(img)
```

Listing 9: Thresholding mit Otsu

Ein Beispiel zeigt Abbildung 5: Aus dem oberen Graubild wurde das untere Binärbild. Als Schwellenwert berechnete `otsu` ca. $0,82$.

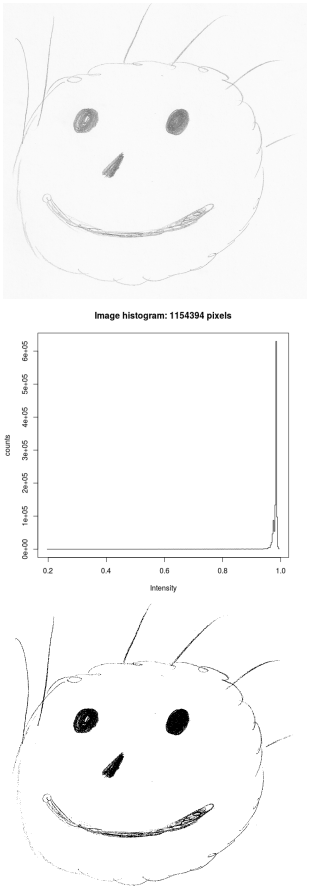

Abbildung 5: Anwendung von Thresholding und Otsu: Oben: Original, unten: nach Thresholding, in der Mitte: das Histogramm des Originalbildes

3.5 Ausblick - weitere Punktoperationen

Neben den hier behandelten homogenen Punktoperationen gibt es noch solche, bei denen der neue Farbwert auch zusätzlich von den Koordinaten des Pixels abhängt, das enstpräche $I'(u,v) = f(I(u,v),u,v)$ in unserer Notation.
Außerdem kann man Punktoperationen auf das Punkt-für-Punkt-Verknüpfen mehrerer Bilder verallgemeinern[15]: $I'(u,v) = f(I_1(u,v), I_2(u,v), ..., I_n(u,v))$. Nur kurz als Beispiel dafür erwähnt sei Alpha-Blending, das zwei Bilder mit einer Gewichtung a überlagert[16].

```
1 img <- a * img1 + (1.0 - a) * img2
```

Listing 10: Alpha-Blending mit zwei Bildern

Ausgelassen haben wir insbesondere auch Punktoperationen auf Farbbildern.

4 Filter

Neben den Punktoperationen, bei denen der neue Wert eines Pixels nur allein vom alten abhängt, gibt es noch die Filter, bei denen der neue Wert eines Pixels zusätzlich von den Pixeln in seiner Umgebung abhängt.

[14] für folgende Erläuterung vgl. [wikipedia, 2016]
[15] Quelle: [Burge, 2006, S. 83]
[16] vgl. [Burge, 2006, S. 84]

4.1 Lineare Filter

Zunächst folgende Definitionen[17]:
Filterregion: $R \subseteq \mathbb{Z} \times \mathbb{Z}$
Filterfunktion $H : R \to \mathbb{R}$
Die Berechnung des Pixels mit den Koordinaten u, v im Bild I', das durch Anwendung des Filters mit der Filterfunktion H auf dem Bild I hervorgeht, erfolgt nach folgender Vorschrift:
$I'(u,v) = \sum_{(i,j)\in R} I(u+i, v+j) \cdot H(i,j)$

Mit anderen Worten: Lineare Filter berechnen eine gewichtete Summe der umgebenden Pixel. Die umgebenden Pixel werden durch die Filterregion R bestimmt und die Filterfunktion weist jedem Pixel aus R ein Gewicht zu.
Daher lässt sich ein linearer Filter (mit rechteckiger Filterregion) auch einfach als Matrix der Gewichte darstellen (Beispiel s. Abbildung 6).
Bei der Implementierung des Filter-Algorithmus muss man darauf achten, dass man immer eine Kopie des zu filternden Bildes anlegen muss, auch wenn man das Originalbild gar nicht mehr benötigt. Sonst würde man schon Teile des Bildes überschreiben und die benachbarten Pixel falsch beeinflussen.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Abbildung 6: Matrix des 3x3 - Boxfilters

Man kann sich den Algorithmus zur Anwendung des Filters so vorstellen:
Man setzt den Ursprung der Filtermatrix auf den gerade zu berechnenden Pixel an der Stelle (u, v), multipliziert die umgebenden Pixel mit den Gewichten aus der Matrix und summiert die gewichteten Werte auf. Das Ergebnis wird in das neue Bild ebenfalls an Position (u, v) geschrieben.

Der Box-Filter ist der einfachste lineare Filter. Seine Matrix (am Beispiel eines 3×3-Box-Filters) zeigt Abbildung 6. Der Ursprung ist gelb markiert.
Er bildet also den Durschnitt der umgebenden Pixel. Dadurch erzeugt er einen einfachen „Weichzeichnen"-Effekt, der das Bild unscharf macht und Bildrauschen unterdrückt - wie in Abbildung 7 zu sehen.
Um beispielsweise den Box-Filter in `R` anwenden zu können, sind zwei Funktionsaufrufe nötig. Zunächst muss man eine Filtermatrix mit der Funktion `makeBrush(size, shape, [...])` erzeugen: Die wichtigsten Parameter sind: `size` - die Größe des Filters in Pixeln und `shape` - die Form des Filters (für Box-Filter ist dies `"box"`).

Abbildung 7: Anwendung eines Box-Filters (links: Original, rechts: nach Box-Filter).
Quelle Original-Farbbild: Lenna-Testbild von `https://en.wikipedia.org/wiki/File:Lenna.png`

Diesen Filter wendet man dann mit der `filter2`-Funktion auf ein Bild an. Der erste Parameter ist das Bild und der zweite der von `makeBrush()` erzeugte Filter. Das gefilterte Bild erhält man als Rückgabewert.
Bei Erstellung eines Box-Filters in `EBImage` ist allerdings eine kleine manuelle Korrektur notwendig. Der Filter ist nicht normiert (also die Summe der Matrixelemente ist nicht gleich 1), weshalb der Filter mit dem Kehrwert seiner Größe skaliert werden muss[18].

```
1 # box braucht manuelle Skalierung
2 brush <- makeBrush(5, "box") * (1/25)
3 smooth <- filter2(img, brush)
```

Listing 11: 5x5-Box-Filter auf Bild anwenden

[17] Die folgenden Formeln sowie der gesamte Abschnitt über lineare Filter basieren auf [Burge, 2006, S. 92f]
[18] warum das so ist, ist mir nicht bekannt

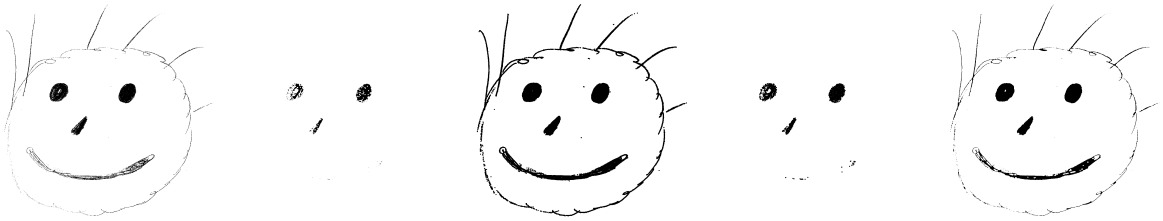

Abbildung 8: Morphologische Operationen mit einem 5x5 - Kreis als Strukturelement, von links nach rechts: Originalbild, Erosion, Dilation, Opening, Closing

4.2 Morphologische Filter

Morphologische Filter werden normalerweise auf Binärbilder angewendet und im Gegensatz zu linearen Filtern sind sie in der Lage, die *Struktur* des Bildinhaltes zu verändern.
Für morphologische Filter werden die Bilder als Mengen aufgefasst: Die zum Bild I gehörende Menge Q_I enthält alle Koordinaten-Paare der (weißen) Vordergrund-Pixel:[19]

$$Q_I = \{(u, v)|I(u, v) = 1\}$$

Der Filter heißt bei morphologischen Operationen Strukturelement und lässt sich ebenfalls nicht nur als Matrix sondern auch als Menge (der Vordergrundpixel) auffassen.

Erosion und Dilation sind die beiden Grundoperationen des morphologischen Filterns. Beide Operationen wenden ein Strukturelement auf ein Bild an und erzeugen damit ein *erodiertes* bzw. *dilatiertes* Bild. Die Mengendarstellung des erzeugten Bildes entsteht wie folgt[20]:

$$I \ominus H = \{(u', v')|\forall(i, j) \in Q_H((u' + i, v' + j) \in Q_I)\} \quad \text{(Erosion)}$$
$$I \oplus H = \{(u', v') = (u + i, v + j)|(u', v') \in Q_I, (i, j) \in Q_H\} \quad \text{(Dilation)}$$

Konkret bedeutet diese Formel für Erosion, dass man H über jedem Pixel von I mit den Koordinaten (u, v) platziert und $I'(u, v)$ genau dann zu einem Vordergrundpixel macht, wenn unter jedem Vordergrundpixel von H auch ein Vordergrundpixel von I liegt. Da diese Bedingung nicht an Rändern von zusammenhängenden Vordergrundbereichen erfüllt ist, schrumpfen Strukturen vom Rand her durch Erosion (Abbildung 8, 2.v.l).
Dilation hingegen lässt Strukturen wachsen, indem H über allen Vordergrundpixeln von I platziert wird und alle Pixel, die dann unter Vordergrundpixeln von H liegen auch in I' zu Vordergrundpixeln werden (Abbildung 8, 3.v.l).

Opening und Closing sind die Operationen die durch Kombination von Erosion und Dilation entstehen[21]:

$$I \circ H = (I \ominus H) \oplus H \quad \text{(Opening)}$$
$$I \cdot H = (I \oplus H) \ominus H \quad \text{(Closing)}$$

Wenn man Erosion gefolgt von Dilation anwendet, schrumpfen die Strukturen erst und wachsen danach wieder. Kleinere Strukturen als das Strukturelement verschwinden dabei. Diese Operation nennt man *Opening*, da sie auch kleine Verbindungen zwischen Objekten entfernt, also *öffnet* (Abbildung 8, 4.v.l).
Wenn man umgekehrt Dilation gefolgt von Erosion anwendet, werden erst alle Strukturen vergrößert und anschließend wieder verkleinert. Dies nennt man *Closing*, da so Lücken in Strukturen geschlossen werden (Abbildung 8, 4.v.l).

[19] Formeln nach [Burge, 2006, S. 174f]
[20] Quelle: [Burge, 2006, s. 175f]
[21] Quelle: [Burge, 2006, S. 179, 182]

EBImage stellt für jede der vorgestellten morphologischen Operationen eine Funktion bereit: `erode()`, `dilate()`, `opening()` und `closing()`. Jede erwartet als erstes Argument das zu verarbeitende Binärbild und als zweites das Strukturelement, das wie bei linearen Filtern mit `makeBrush()` erstellt wurde. Bei `makeBrush()` eigent sich neben dem bekannten `"box"`-Filter auch der kreisförmige `"disc"`-Filter.
Hinweis: In `EBImage` wird immer vom Farbwert 1 (= weiß) als Vordergrund ausgegangen. Oftmals ist aber Schwarz der Vordergrund, dann muss man entweder das Bild vor und nach der Operation jeweils negieren oder die jeweils duale Operation (Erosion statt Dilation, Opening statt Closing) wählen. Beim folgenden `R` -Skript ist Schwarz der Vordergrund.

```
1  img <- 1.0 - img # Schwarz ist Vordergrund, img ist Binaerbild
2
3  # Strukturelement erstellen
4  struct <- makeBrush(5, "disc") # 5x5 Kreis
5
6  # Anwenden
7  eroded <- erode(img, struct)
8  # bzw. dilate(img, struct), opening(img, struct) oder closing(img, struct)
9
10 eroded <- 1.0 - eroded # Schwarz = Vordergrund
```

Listing 12: Morphologische Operationen

5 Zusammenfassung

Die Bildverarbeitung beschäftigt sich mit der Entwicklung von Algorithmen zur Verarbeitung von Pixeldaten, die aus Eingabebildern als Ausgabe wieder Bilder erzeugen.
Digitale Bilder sind zweidimensionale Abbildungen, die Koordinaten Farbwerte zuweisen und auch als rechteckige Matrizen betrachtet werden könne. Üblicherweise unterscheidet man Farb-, Grau(stufen)- und Binärbilder.
In `R` können wir Bilder unter anderem mit Hilfe des Paketes `EBImage` verarbeiten.
Ein wichtiger Analyseschritt vor der Weiterverarbeitung der Bilder ist oft die Erzeugung eines Histogramms, das Informationen über die Verteilung der Helligkeitswerte eines (Grau-)Bildes und damit auch zur Belichtung liefert. Es kann auch zur Anpassung des Kontrastes verwendet werden.
Die zwei wichtigsten Klassen von Bildverarbeitungsoperationen sind die Punktoperationen und Filter: Bei Punktoperationen ist der neue Wert eines Pixels nur vom vorherigen Wert abhängig und bei homogenen Punktoperationen sogar von seinen Koordinaten. Bei Filtern hingegen sind auch die umgebenden Pixel zur Bestimmung des neuen Pixelwertes von Bedeutung. *Lineare* Filter berechnen gewichtete Summen der umgebenden Pixel und verändern die Struktur des Bildinhaltes nicht. *Morphologische* Filter verändern den Inhalt und die Struktur des Bildes.
Wichtige Punktoperationen betreffen neben Änderungen von Helligkeit und Kontrast die Konvertierungen von Farb- nach Graubild und nach Binärbild (*Thresholding*).
Die wichtigsten linearen Filter sind Glättungsfilter wie der Box-Filter.
Die morphologischen Grundoperationen sind das Wachsen- (*Dilation*) und Schrumpfenlassen (*Erosion*) von Bildstrukturen. Die Hintereinanderausführungen daraus ergeben *Opening* und *Closing*.
In `R` stehen mit `EBImage` für alle oben genannten Operationen einfach nutzbare Funktionen bereit.

6 Literatur

Literatur

[Burge, 2006] Burge, W. B. . M. J. (2006). *Digitale Bildverarbeitung : eine Einführung mit Java und ImageJ*. Springer, Berlin, 2., überarb. edition.

[EBImage, 2016] EBImage (2016). EBImage documentation `http://www.bioconductor.org/packages/release/bioc/manuals/EBImage/man/EBImage.pdf`, 26.06.2016.

[wikipedia, 2016] wikipedia (2016). Otsu's method `https://en.wikipedia.org/wiki/Otsu%27s_method`, 26.06.2016 .