Grundlagen der Informatik III

Wintersemester 2015/2016

Stefan Roth, Jochen Gast

TECHNISCHE UNIVERSITÄT DARMSTADT

2. Praktikum

Ausgabe: 01.12.2015; Abgabe: 14.12.2015 23:55 Uhr

Abgabe der Programme: Ihre Abgabe soll sämtliche zum Kompilieren und Ausführen Ihres Programms benötigten Dateien in einer .zip- oder .gz-Datei komprimiert enthalten und im Moodle-Kurs unter der entsprechenden Abgabe hochgeladen werden.

Hinweis: Alle folgenden Aufgaben beziehen sich auf den **ANSI C - Standard (bis C11)**. Beachten Sie, dass für die Bearbeitung der Aufgaben keine externen Bibliotheken mit Ausnahme der C Standard Bibliotheken benutzt werden dürfen.

Lineares und Nichtlineares Filtern von Bildern

Die Verwendung von Filtern ist eine grundlegende Operation der digitalen Bildverarbeitung. Sie werden vor allem als Vorverarbeitungsschritt und zur Visualisierung angewandt. Prominente Beispiele sind das Blurring (Weichzeichnen) und Schärfen von Bildern (Abbildung 0.1).

Abbildung 0.1: Verwendung von linearen Filtern zum Blurren/Schärfen von Bildern.

Diskrete Faltung

Zunächst konzentrieren wir uns in diesem Praktikum auf einen linearen Filter in Form einer *diskreten Faltung*. Dabei ist das Eingangsbild durch ein zweidimensionales Array aus Grauwerten gegeben, das die Farbintensitäten der Pixel im Wertebereich [0,255] speichert, z.B. im Falle des in Abbildung 0.1 gezeigten Bildes

$$f = \begin{pmatrix} 45 & 46 & 56 & \dots \\ 43 & 44 & 50 & \dots \\ 39 & 43 & 46 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}. \tag{0.1}$$

Nun wird ein sogenannter Kernel h auf die Bildmatrix f angewandt, indem jeder Pixel durch eine Linearkombination seiner Nachbarn (und sich selbst) ersetzt wird. Der Kernel beschreibt dabei die Gewichte der Linearkombination. Die Faltungsoperation wird durch das Symbol $*$ gekennzeichnet:

$$g = f * h, \qquad g(i,j) = \sum_{k,l} f(i-k, j-l)\, h(k,l) = \sum_{k,l} f(k,l)\, h(i-k, j-l), \tag{0.2}$$

45	60	98	127	132	133	137	133
46	65	98	123	126	128	131	133
47	65	96	115	119	123	135	137
47	63	91	107	113	122	138	134
50	59	80	97	110	123	133	134
49	53	68	83	97	113	128	133
50	50	58	70	84	102	116	126
50	50	52	58	69	86	101	120

Bild $f(i,j)$

*

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

Kernel $h(i,j)$

=

69	95	116	125	129	132
68	92	110	120	126	132
66	86	104	114	124	132
62	78	94	108	120	129
57	69	83	98	112	124
53	60	71	85	100	114

Gefiltertes Bild $g(i,j)$

Das Ausgangsbild ist je nach Größe des angewandten Filters kleiner als das Eingangsbild. Um ein Ausgangsbild der gleichen Größe zu erhalten müssen sogenannte Randbehandlungsstrategien eingesetzt werden. Hier benutzen wir für alle linearen Filter die sogenannte Replikation:

- **Replikation.** Werte außerhalb des Bildrandes gleichen dem Wert des (örtlich) nächsten Nachbarn auf dem Bildrand.

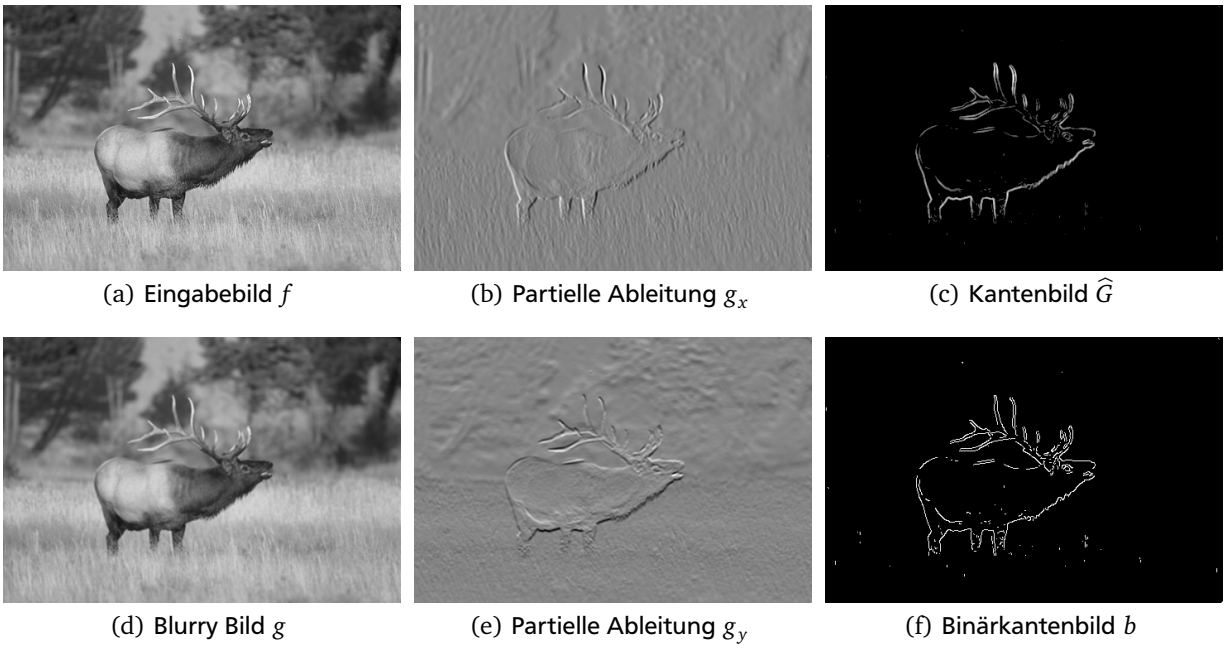
Nichtlineares Filtern

Der bisher betrachtete Filter war linear; eine allgemeinere Variante des Filterns stellt das nichtlineare Filtern dar. Hier wird, anstelle einer linearen Operation durch einen diskreten Filterkernel, eine nichtlineare Operation auf lokale Nachbarschaften angewandt. In diesem Praktikum betrachten wir eine spezielle Untergruppe nichtlinearer Filter, sogenannte *Rangordnungsfilter*, welche Pixel-Intensitäten lokaler Nachbarschaften durch Sortieren in eine *Rangordnung* bringen und anschließend jeweils eine Intensität aus dieser Rangordnung auswählen. Nach der Anwendung eines Rangordnungsfilters erhält man also in der Regel ein Bild, bei dem jeder Pixelwert durch einen Wert innerhalb seiner lokalen Nachbarschaft ersetzt worden ist. Hier betrachten wir drei Beispiele:

- **Minimumfilter.** Wähle den minimalen Wert der Rangordnung aus. Im Beispiel oben würde aus der blau markierten Nachbarschaft der Wert 63 ausgewählt werden.
- **Medianfilter.** Wähle den Median der Rangordnung aus. Im Beispiel oben würde aus der blau markierten Nachbarschaft der Wert 96 ausgewählt werden.
- **Maximumfilter.** Wähle den maximalen Wert der Rangordnung aus. Im Beispiel oben würde aus der blau markierten Nachbarschaft der Wert 123 ausgewählt werden.

Abbildung 0.2 zeigt beispielhaft die Anwendung eines Medianfilters für einen 5x5 und einen 7x7 Medianfilter.
Hinweis: Beachten Sie, dass für Rangordnungsfilter in der Regel keine Randbehandlungsstrategien vorgegeben sind, d.h. am Rand oder der Ecke eines Bildes gibt es ggf. weniger Pixel (als in der Mitte des Bildes), die in eine Rangordnung gebracht werden müssen. Weitere Details zum linearen und nichtlinearen Filtern von Bildern finden Sie beispielsweise auf `http://szeliski.org/Book` (frei erhältliches Ebook).

Abbildung 0.2: Verwendung eines nichtlinearen Median Filters. Links: Inputbild. Mitte: 5x5 Median. Rechts: 7x7 Median.



(a) Eingabebild f (b) Partielle Ableitung g_x (c) Kantenbild $\widehat{G}$

(d) Blurry Bild g (e) Partielle Ableitung g_y (f) Binärkantenbild b

Abbildung 0.3: Visualisierung der Teilergebnisse im Canny-Algorithmus.

Vereinfachter Canny-Algorithmus zur Erkennung von Kanten

Ein weiterer Anwendungsfall für Filter ist die Erkennung von Kanten in Bildern. In diesem Praktikum werden Sie eine vereinfachte Variante des sogenannten Canny-Algorithmus implementieren, welcher basierend auf Faltungsoperationen ein Bild mit den Kanten des Eingabebildes erzeugt. Um sicher zu stellen, dass Mehrfachdetektionen Kanten nicht zu breit erscheinen lassen, verfeinert der Algorithmus dieses Kantenbild in einem weiteren Verarbeitungsschritt, der sogenannten *non-maximum suppression*. Wir betrachten den Algorithmus anhand eines Beispiels, welches in Abbildung 0.3 gezeigt ist. Der Algorithmus geht wie folgt vor um aus einem Eingabebild f (Abbildung 0.3(a)) ein Kantenbild (Abbildung 0.3(f)) zu erzeugen

1. **Blurren.** Um den Einfluss von Bildrauschen zu reduzieren, wird das Eingabebild f in einem Vorprozess geblurrt. Hierzu wenden Sie den Filterkernel

$$h_{\text{blur}} = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \tag{0.3}$$

   auf das Bild an und erhalten ein geblurrtes Bild g (Abbildung 0.3(d)).

2. **Partielle Ableitungen.** Anschließend müssen die partiellen Ableitungen g_x und g_y des geblurrten Bildes in x-Richtung und y-Richtung berechnet werden. Sie erhalten die partiellen Ableitungen, indem Sie das Bild jeweils mit dem sogenannten Sobeloperator filtern:

$$h_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \quad \text{Sobeloperator in x-Richtung}$$

$$h_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \quad \text{Sobeloperator in y-Richtung}$$

   Abbildung 0.3(b) und Abbildung 0.3(e) zeigen jeweils Visualisierungen der partiellen Ableitungen.

3. **Stärke der Kanten.** Nun berechnen Sie ein Bild G mit der absoluten Stärke der Kanten. Dieser ergibt sich pixelweise aus der euklidischen Norm der partiellen Ableitungen

$$G(i,j) = \sqrt{g_x(i,j)^2 + g_y(i,j)^2}$$

4. **Kantendetektion.** Basierend auf der Kantenstärke G müssen wir nun einen Schwellenwert τ festlegen, der besagt, ab welcher Stärke eine Kante als solche erkannt wird. Alle Werte kleiner als τ müssen demzufolge auf 0 gesetzt werden. Wir erhalten somit das Kantenbild $\widehat{G}$, welches in 0.3(c) visualisiert ist.

5. **Richtung der Kanten.** Für die *non-maximum suppression* benötigen wir zusätzlich die Richtung θ des Gradienten. Diese kann pixelweise über den Arkustangens berechnet werden:

$$\theta(i,j) = \text{atan}(g_y(i,j)/g_x(i,j))$$

6. **Non-maximum suppression.** Die absolute Stärke der Kanten $G(i,j)$ liefert zunächst ein initiales Kantenbild, welches Kanten aufgrund von Mehrfachdetektionen nicht gut lokalisiert. Dies hat den Effekt, dass viele Kanten zu breit erscheinen. Um die Kanten weiter zu verfeinern, wenden wir eine vereinfachte *non-maximum suppression* an: Diese vermeidet Mehrfachdetektionen von Kanten, indem Nichtmaxima entlang des Gradienten unterdrückt werden. [1] Implementieren Sie die *non-maximum suppression* folgendermaßen:

   - Diskretisiere θ in vier Richtungen: vertikal (Nord-Süd), horizontal (West-Ost) und zwei diagonale Richtungen (Nordwest-Südost und Nordost-Südwest). Die folgende Tabelle veranschaulicht die möglichen Richtungen:

θ Intervalle (in Grad)	**Richtung der *non-maximum suppression***
$(67.5°, 90°]$ oder $[-90°, -67.5°]$	Nord-Süd
$(-22.5°, 22.5°]$	West-Ost
$(22.5°, 67.5°]$	Nordwest-Südost
$(-67.5°, -22.5°]$	Nordost-Südwest

   - Bestimme nun für jeden Kandidatpixel eine (diskretisierte) Richtung der korrespondierenden Kante. Wenn nun einer der beiden Nachbarpixeln, die in dieser Richtung liegen, einen größeren Wert hat, unterdrücke den Kandidatpixel, indem dessen Wert auf 0 gesetzt wird. Abbildung 0.4 zeigt dazu ein kurzes Beispiel: Der betrachtete Pixel in grün hat die Gradientenmagnitude $\widehat{G} = 18$. Da die dazugehörige Kantenrichtung gegeben ist durch $\theta = 80°$, müssen die Nachbarn in Nord-Süd Richtung (blau) verglichen werden. Der obere Nachbar hat einen größere Magnitude (22), weshalb der betrachtete Pixel unterdrückt werden muss.

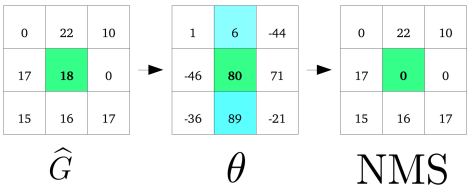


Abbildung 0.4: *non-maximum suppression*

7. **Binärkantenbild.** Nach der *non-maximum suppression* aus dem letzten Schritt, entsprechen alle übrig gebliebenen Pixel, dessen Werte größer als 0 sind, den Kanten im Bild. Erzeugen Sie aus diesem Bild nun ein Binärbild, welches ausschließlich aus 0/1 Pixeln besteht (1 für eine Kante im Bild, sonst 0). Das finale Ergebnis ist in Abbildung 0.3(f) gezeigt.

[1] Für Interessierte: Dies funktioniert, da der Bildgradient in die Richtung orthogonal zur Bildkante zeigt.

Portable Graymaps

Wir werden Gebrauch von Graustufenbildern im PGM (Portable Graymap) Format machen. Hier nehmen wir vereinfacht an, dass der Dateikopf einer PGM Datei folgendermaßen aufgebaut ist:

1. Magischer Wert “P2”: Beschreibt das Format der Bilddaten.
2. Zeilenvorschub (Carriage return)
3. Optional: Mehrzeiliges Kommentar zeilenweise startend mit “#”.
4. Breite des Bilds (dezimal in ASCII kodiert).
5. Leerzeichen
6. Höhe des Bilds (dezimal in ASCII kodiert).
7. Zeilenvorschub (Carriage return)
8. Maximalwert für die Helligkeit (dezimal in ASCII kodiert).
9. Zeilenvorschub (Carriage return)

Darauf folgen die Bilddaten in Form von Pixeln kodiert als ASCII. Beispielsweise wird durch die PGM Datei mit dem Inhalt

```
P2
# Das Wort "FEEP" in verschiedenen Graustufen (Beispiel von der Netpbm-Man-Page)
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

das folgende Bild kodiert:

Abbildung 0.5: Visualisierung der PGM-Datei von oben.

Aufgabenstellung

Für die Teilaufgaben 1–4 implementieren Sie jeweils ein ANSI C Programm, welches von der Kommandozeile aus mit verschiedenen Eingabe- und Ausgabedateinamen aufgerufen werden soll.

Laden Sie zunächst die Datei **Praktikum2-Coderahmen.zip** von der Moodle-Website herunter, welche Bilder sowie einen leeren Coderahmen enthält. Basierend auf diesem Rahmen bearbeiten Sie bitte folgende Aufgaben:

Aufgabe 0: CMake – Plattformunabhängiges Kompilieren

Installieren Sie, falls sie nicht im IST-Pool arbeiten, zunächst das Programm **CMake**, welches Sie auf `https://cmake.org` herunterladen können. Machen Sie sich mit dem Coderahmen sowie der Entwicklungsumgebung `CMake` vertraut. Der Coderahmen enthält drei Unterordner: Einen Unterordner **bin** für Binärdateien und ausführbare Dateien, einen Unterordner **build** für alle CMake-Projektdateien sowie einen Unterordner **src** für den Sourcecode. Im Hauptordner finden Sie die Datei **CMakeLists.txt**, welche alle benötigten Informationen enthält, um das Projekt zu kompilieren. Für Unix-basierte Betriebssysteme kann der Code nun folgendermaßen kompiliert werden:

```
1) Generiere Projektdateien:
   $ cd build
   $ cmake ..

2) Kompiliere und Linke:
   $ make
```

Diese Befehle erzeugen nun, jeweils für alle Teilaufgaben, Executables im **bin** Ordner. Für andere Betriebssysteme (z.B. unter Windows) müssen ggf. Anpassungen mit dem graphischen CMake Editor gemacht werden, um Projektdateien, zu generieren, die anschließend beispielsweise mit Visual Studio kompiliert werden können. **Hinweis**: Es ist erwünscht, dass Sie weitere Header- und Sourcedateien anlegen und die CMake Projektdatei `CMakeLists.txt` um diese erweitern.

Aufgabe 1: Lesen/Schreiben von Portable Graymaps

Implementieren Sie das Einlesen- und Schreiben von PGM-Dateien in ANSI C, indem Sie das Programm **p1.c** vervollständigen. Dieses soll beispielhaft für den Aufruf "`./p1 src.pgm dst.pgm`"

- die PGM-Datei **src.pgm** einlesen und als PGM in die Ausgangsdatei **dst.pgm** exportieren.

Beachten Sie außerdem:

- Obwohl ihr Programm in der Lage sein sollte, mit den Kommentaren einer PGM Datei umzugehen, müssen Sie diese nicht abspeichern.
- Wir geben Ihnen nicht vor, welche Datenstruktur und welche Datentypen Sie verwenden. Allerdings können Schwierigkeiten bei den folgenden Aufgaben vermieden werden, wenn Sie intern einen Gleitkomma-Datentypen wählen und diesen ausschließlich beim Speichern von PGM Dateien wieder in ASCIII konvertieren.
- PGM-Dateien können Sie mit einem Bildeditor ihrer Wahl visualisieren (z.B. GIMP).

Aufgabe 2: Lineares Filtern

Implementieren Sie nun Funktionen, die das lineare Filtern von Bildern in Ihrer gewählten Datenstruktur ermöglichen. Um Ihre Implementierung zu testen, schreiben Sie das Programm **p2.c** so um, dass es (beispielhaft) für den Aufruf "`./p2 src.pgm blur.pgm sharp.pgm`"

- das Bild `src.pgm` einliest,

- mit Hilfe des Filterkernels

$$h_{\text{blur}} = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

  weichzeichnet und nach `blur.pgm` exportiert,

- mit Hilfe des Filterkernels

$$h_{\text{sharp}} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

  schärft und nach `sharp.pgm` exportiert.

Aufgabe 3: Nichtlineares Filtern

Implementieren Sie nun Funktionen für das nichtlineare Filtern von Bildern. Um Ihre Implementierung zu testen, schreiben Sie das Programm **p3.c** so um, dass es (beispielhaft) für den Aufruf "`./p3 src.pgm N min.pgm med.pgm max.pgm`"

- das Bild `src.pgm` einliest,
- mit Hilfe eines NxN-Minimum-Filters nichtlinear filtert und nach `min.pgm` exportiert,
- mit Hilfe eines NxN-Median-Filters nichtlinear filtert und nach `med.pgm` exportiert,
- mit Hilfe eines NxN-Maximum-Filters nichtlinear filtert und nach `max.pgm` exportiert.

Hinweis: Sie können davon ausgehen, dass die Eingabezahl N ungerade ist.

Aufgabe 4: Canny-Algorithmus

Implementieren Sie den vorgestellten Canny-Algorithmus für die Erkennung von Bildkanten. Vervollständigen Sie dazu Programm **p4.c**, sodass es (beispielhaft) für den Aufruf "`./p4 elk.pgm edges.pgm`"

- das Bild `elk.pgm` einliest,
- und ein Binärkantenbild nach `edges.pgm` exportiert.

Beachten Sie außerdem:

- Passen Sie den gewählten Schwellenwert τ so an, dass er für das Testbild `elk.pgm` ein Ergebnis liefert, das in etwa dem der Abbildung 0.3(f) entspricht.