

Prof. Dr. Thomas Vetter
Departement Mathematik und Informatik
Spiegelgasse 1, CH – 4051 Basel

Patrick Kahr (patrick.kahr@unibas.ch)
Renato Farruggio (renato.farruggio@unibas.ch)
Martin Ramm (martin.ramm@unibas.ch)

Übungsblatt 2

Ausgabe: 18.03.2021

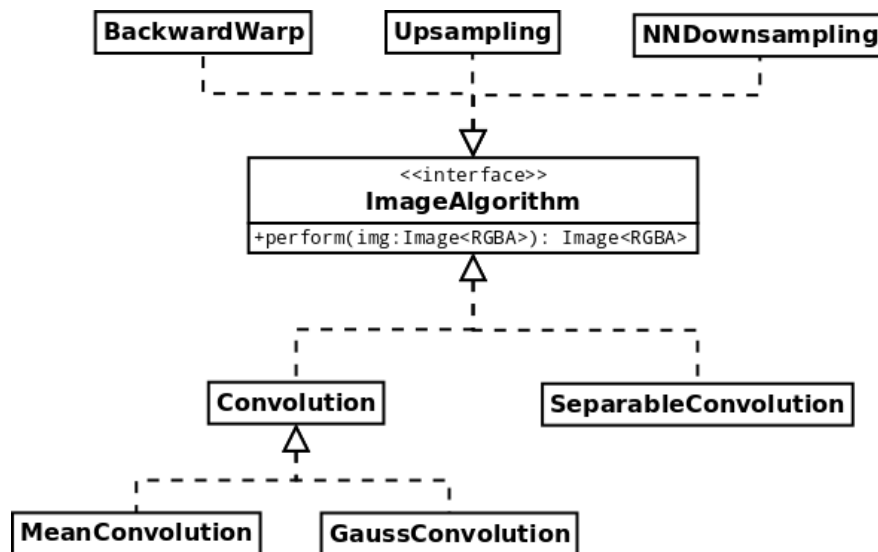
Abgabe: 08.04.2021

Zu erreichende Punktzahl: 30 (Programmieraufgaben + Theoriefragen)

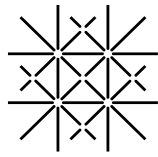
Bevor Sie die Übungen lösen, lesen Sie bitte dieses Übungsblatt sowie das Infoblatt aufmerksam durch. Es ist wichtig dass die Abgabe rechtzeitig und wie im Infoblatt beschrieben erfolgt.

In diesem Übungsblatt werden Sie einige wichtige und häufig verwendete Algorithmen der digitalen Bildbearbeitung implementieren.

Die Klassenstruktur dieses Aufgabenblattes ist wie folgt aufgebaut:



Die Klassen befinden sich alle in (Sub-) Packages von `image.processing`.



Aufgabe 1 – Faltung

6 Punkte

In dieser Aufgabe sollen Sie das Faltungsprodukt mit verschiedenen Kernels implementieren. Ein Kernel wird in unserem Fall als `Image<Float>` dargestellt.

Unit-Tests: `ex2.KernelTest.java`

Hinweis 1: Falls Sie beim Testen die Fehlermeldung `Failed loading resources.` erhalten, so ist vermutlich das working directory der JUnit run-configurations falsch eingestellt. Passen Sie es an zu `out/test/` und probieren Sie es dann erneut. Ansonsten helfen wir gerne weiter.

(a) Mean und Gauss Kernel (1½ Punkte)

Zuerst bauen wir uns ein paar Kernel: einen Mittelwerts- und einen Gauss-Filter. Vervollständigen Sie dazu die Methode `getKernel()` in den Klassen `MeanConvolution` und `GaussConvolution`.

Hinweis 1: Der Gauss Kernel sollte mit der Summe seiner Einträge normiert werden.

Benötigte Dateien: `image.processing.convolution.MeanConvolution.java`,
`image.processing.convolution.GaussConvolution.java`

(b) Convolution (1½ Punkte)

Nun können wir die eigentliche Faltung, die Sie in der Vorlesung besprochen haben, implementieren. Vervollständigen Sie dazu die Methode `perform(..)` der Klasse `Convolution`. Beachten Sie, dass eine Faltung am Rand des Bildes immer auf Werte ausserhalb des Bildes zugreift. Normalerweise müsste man deshalb eine spezielle Randbehandlung einführen. Dies wurde für Sie in der Methode `get(..)` der `Image` Klasse übernommen (bei Zugriff ausserhalb des Bildes wird der nächste Randwert zurückgegeben). Sie brauchen sich also nicht darum zu kümmern.

Hinweis 1: Die Methode `printKernel()` könnte für das Debugging hilfreich sein...

Benötigte Dateien: `image.processing.convolution.Convolution.java`

(c) Separable Convolution (1½ Punkte)

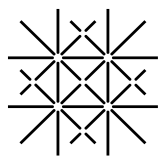
Separierbare Filter haben die Eigenschaft, dass sie sich als Multiplikation eines Vektors mit sich selbst darstellen lassen. Beispiel:

$$\frac{1}{16} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = \frac{1}{4} \begin{pmatrix} 1 & 2 & 1 \end{pmatrix} \cdot \frac{1}{4} \begin{pmatrix} 1 & 2 & 1 \end{pmatrix}^T$$

Diese Eigenschaft hat zur Folge, dass eine Faltung realisiert werden kann, indem zuerst alle Zeilen des Bildes und danach alle Spalten des *Ergebnisbildes* mit dem 1D-Kernel gefaltet werden. Dies ist effizienter als eine volle 2D-Faltung. Der Gauss Kernel ist ein Beispiel für einen separierbaren Filter. In dieser Aufgabe sollen Sie:

- Einen separierbaren Gauss Kernel in der Klasse `GaussSeparableConvolution` implementieren.
- Die separable Faltung in der Klasse `SeparableConvolution` realisieren. Beachten Sie hierbei, dass wirklich zuerst alle Zeilen und **danach** alle Spalten mit einem 1D Kernel gefiltert werden.

Benötigte Dateien: `image.processing.convolution.SeparableConvolution.java`,
`image.processing.convolution.GaussSeparableConvolution.java`



(d) Weitere Kernel

(1½ Punkte)

In dieser Teilaufgabe werden wir ein paar weitere Kernel generieren. Oftmals ist es von Interesse, in einem Bild die Kanten zu detektieren. Dazu gibt es verschiedene Möglichkeiten:

- Mit einem Laplace Kernel: $-\frac{1}{8} \begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
- Mit einem Sobel Kernel (in horizontaler Richtung). Der Sobel Kernel ist nichts weiteres als eine Approximation an den Gradienten an einer bestimmten Stelle.

Implementieren Sie diese zwei Kernel in der `getKernel()` Methode der jeweiligen Klassen. Beachten Sie, dass bei diesen Kernel negative Farbwerte entstehen können. Dieser Umstand wird von uns abgefangen, indem das resultierende Bild normalisiert, d.h. alle Farbwerte ins Intervall $[0, 1]$ skaliert werden.

Benötigte Dateien: `image.processing.convolution.LaplaceConvolution.java`,
`image.processing.convolution.SobelHConvolution.java`

Aufgabe 2 – Skalierung

5 Punkte

In dieser Aufgabe geht es darum, verschiedene Verfahren zur Bildvergrößerung und Bildverkleinerung kennenzulernen. Der Einfachheit halber beschränken wir uns dabei auf eine Skalierung um den Faktor 2.

Unit-Tests: `ex2.ScalingTest.java`

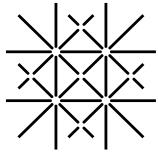
(a) Downsampling

(2 Punkte)

Die einfachste Methode um ein Bild zu verkleinern, bezeichnet man als Nearest-Neighbour Downsampling oder auch Subsampling. Hierbei wird einfach jedes zweite Pixel ausgelassen. Implementieren Sie dieses Verfahren in der Methode `perform(..)` der Klasse `NNDownsampling`.

Dieses Verfahren führt allerdings zu Artefakten. Ein besseres Resultat erhält man, wenn man die hohen Frequenzen aus dem Bild herausschneidet, bevor man es verkleinert. Implementieren Sie dieses Verfahren in der Klasse `GaussianDownsampling` indem Sie das zu verkleinernde Bild vor der Skalierungsoperation mit einem Gauss Kernel filtern. Verwenden Sie dazu Ihren Gauss Filter aus der vorigen Aufgabe ($size = 5, \sigma = 1$). Sehen Sie die Verbesserung?

Benötigte Dateien: `image.processing.scaling.NNDownsampling.java`,
`image.processing.scaling.GaussianDownsampling.java`



(b) Upsampling

(1 Punkt)

Das Vergrössern von Bildern gestaltet sich ein bisschen schwieriger. Beim Downsampling konnten wir einfach Informationen weglassen, jetzt müssen wir neue Informationen aus bestehenden interpolieren. Auch dafür gibt es wiederum verschiedene Verfahren. Das einfachste - die Nearest Neighbour Interpolation - wurde für Sie implementiert. Hierbei wird die Farbe eines Pixels auf den Farbwert des nächsten umliegenden Pixels gesetzt. Schauen Sie sich diese Implementierung in der Klasse `NNInterpolation` an.

Implementieren Sie dann die Methode `perform(..)` in der Klasse `Upsampling` um die Nearest Neighbour Interpolation zu visualisieren. Verwenden Sie die Methode `access(..)` der Membervariable `interpolation` um auf einen Pixel im zu vergrößernden Bild zuzugreifen.

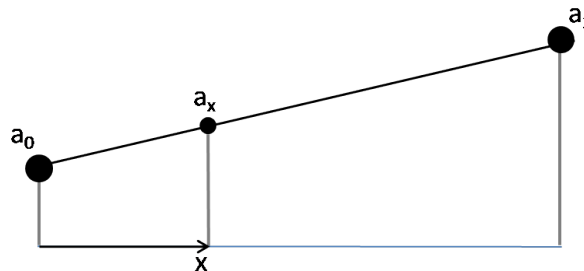
Benötigte Dateien: `image.processing.scaling.NNInterpolation.java`,
`image.processing.scaling.Upsampling.java`

(c) Interpolation

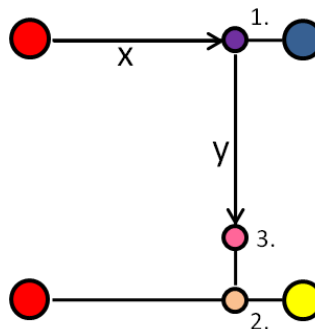
(2 Punkte)

Das Upsampling mit Nearest Neighbour Interpolation kann heftige Artefakte verursachen. Wir wollen deshalb zwei bessere Verfahren ansehen, um Pixelwerte zu interpolieren.

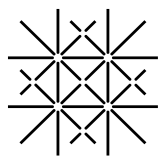
Bilineare Interpolation: Bei der eindimensionalen, linearen Interpolation konstruiert man eine Gerade $g: \mathbb{R} \rightarrow \mathbb{R}$ auf Grund zweier Punkte (auch Stützstellen genannt) und setzt den interpolierten Wert an der Stelle x als: $a_x = xa_1 + (1 - x)a_0$.



Angewandt auf ein Bild können wir einen Punkt einfärben, indem wir die Farbwerte seines linken und rechten Pixelnachbarn als Stützstellen betrachten. Da unser Bild zweidimensional ist, müssen wir die lineare Interpolation sowohl in x wie auch in y -Richtung vornehmen (deshalb auch *bilinear* Interpolation). Wir betrachten im Ganzen also 4 Nachbarpixel:



Implementieren Sie dies in der Methode `access` der Klasse `BiLinearInterpolation`.



Bikubische Interpolation: Bei der kubischen Interpolation wollen wir anstelle einer Geraden ein Polynom $f : \mathbb{R} \rightarrow \mathbb{R}$ dritten Grades interpolieren:

$$f(x) = px^3 + qx^2 + rx + b$$

Um die 4 unbekannten Parameter p, q, r, b zu bestimmen, brauchen wir 4 Stützstellen. Angenommen die Stützstellen seien a, b, c und d , dann können wir einen Punkt $x \in [0, 1]$ interpolieren, indem wir setzen:

$$\begin{aligned} p &= (d - c) - (a - b), \\ q &= 2(a - b) - (d - c), \\ r &= c - a. \end{aligned}$$

Auf ein Bild angewandt, entsprechen die Stützstellen den beiden linken und den beiden rechten Pixelnachbarn eines Punktes. Dehnen Sie dieses Konzept wie schon bei der linearen Interpolation auf den zweidimensionalen Fall aus (Sie müssen also total 16 Pixelwerte betrachten) und vervollständigen Sie die Klasse `BiCubicInterpolation`.

Hinweis 1: Sowohl die bilineare, wie auch die bikubische Interpolation lassen sich als Kombination von eindimensionalen linearen bzw. kubischen Interpolationen schreiben.

Hinweis 2: Bedenken Sie, dass berechnete Farbwerte unter Umständen über oder unter den erlaubten Maximal- bzw. Minimalwert fallen können.

Benötigte Dateien: `image.processing.scaling.BiLinearInterpolation.java`,
`image.processing.scaling.BiCubicInterpolation.java`

Aufgabe 3 – Warp

9 Punkte

In der Vorlesung haben Sie zwei Möglichkeiten kennen gelernt, einen Warp durchzuführen. Bei einem Backward Warp wird für jeden Pixel im *Ziel*-Bild bestimmt, wo der Pixel im alten Bild seinen Ursprung hat (und wenn nötig interpoliert). Dies bedingt, dass man das Inverse derjenigen Transformation, welche man eigentlich darstellen will, kennt.

Bei einem Forward Warp geht man anders vor: Man berechnet für jeden Pixel im *Ursprungs*-Bild, wohin er im Ziel fällt und malt diesen Pixel dann an. Hier wird keine Inverse benötigt, dafür hat es den Nachteil, dass Pixel mehrfach angemalt werden können und es nicht ganz klar ist, welche Pixel überhaupt eingefärbt werden sollen.

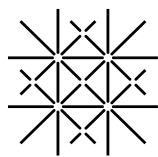
Unit-Tests: `ex2.WarpTest.java`

(a) Backward Warping

(2 Punkte)

Implementieren Sie einen Backward Warp in der Klasse `BackwardWarp`. Die Transformation ist von uns als Vektorfeld vorgegeben und in der Variable `flowField` abgespeichert. Dieses Vektorfeld gibt zu jedem Punkt im Zielbild seine relative Positionsverschiebung als Vektor an.

Benötigte Dateien: `image.processing.warping.BackwardWarp.java`



(b) **Forward Warping**

(3 Punkte)

Implementieren Sie einen Forward Warp in der Klasse `ForwardWarp`. Wiederum ist die Transformation als ein Vektorfeld gegeben. Berechnen Sie die Transformation jeweils für drei Pixel gleichzeitig und verwenden Sie Ihren `SimpleRenderer` vom letzten Übungsblatt um die transformierten Pixel als Dreiecke ins Zielbild zu zeichnen.

Benötigte Dateien: `image.processing.warping.ForwardWarp.java`

(c) **Morphing**

(4 Punkte)

Die Ausgangslage für diese Aufgabe sind zwei Bilder A und B , welche Gesichter zeigen. Ausserdem erhalten Sie zwei Korrespondenzfelder `a2b` und `b2a`. Das Feld `a2b` beschreibt, wie man einen Pixel im Bild A (relativ zu seiner Position) bewegen muss, um zum entsprechenden Punkt im Bild B zu kommen (analog für `b2a`). Dabei handelt es sich um eine *echte* Korrespondenz. Wenn bspw. der Punkt $p \in A$ die Nasenspitze beschreibt, dann zeigt der zugehörige Vektor $v = \text{a2b}(p)$ wo sich die Nasenspitze im Bild B befindet.

Wie können nun beliebige Mischungsverhältnisse

$$\lambda \cdot A + (1 - \lambda) \cdot B$$

bezüglich Form und Farbwerten erreicht werden? Implementieren Sie dies in der Methode `morph()` der Klasse `Morphing`. Für Werte $\lambda < 0$ und $\lambda > 1$ erhält man Karikaturen, die alle Merkmale verstärken, in denen sich das eine Bild vom anderen unterscheidet.

Hinweis 1: Die Aufgabe kann mit einem Backward oder einem Forward Warp gelöst werden.

Benötigte Dateien: `image.processing.warping.Morphing.java`

Aufgabe 4 – Theoriefragen

10 Punkte

Fragen zur Theorie: Die folgenden Fragen sind schriftlich zu beantworten. Bitte legen Sie Ihre Antwort als PDF Dokument (`blatt2-antworten.pdf`) in das dafür vorgesehene Verzeichnis im Übungsframework ab.

(a) **Separierbare Filter**

(5 Punkte)

In den Programmieraufgaben haben Sie die Faltung mit separierbaren Filtern kennengelernt. Anstatt einer 2D Faltung, machen Sie nacheinander zwei 1D Faltungen. Erläutern Sie die Unterschiede dieser zwei Vorgehensweisen und machen Sie eine Komplexitätsanalyse. Versuchen Sie die theoretischen Resultate empirisch nachzuweisen und kommentieren Sie Ihre Beobachtungen.

Hinweis 1: Beschreiben Sie die Komplexität in \mathcal{O} -Notation für ein Bild der Grösse $M \times N$ und eine Filtermaske der Grösse $k \times k$.

(b) **Filtern im Fourierraum**

(5 Punkte)

Das Filtern in Aufgabe 1 wurde mittels Faltung des Bildes und der Filtermaske im Ortsraum durchgeführt. Alternativ dazu kann auch im Fourierraum (Frequenzraum) gefiltert werden. Erklären Sie kurz die Zusammenhänge zwischen beiden Varianten. Welche Schritte müssen jeweils durchgeführt werden um ein Bild ($M \times N$) mit einer Filtermaske ($k \times k$) zu filtern? Können Sie Vor- und Nachteile ausmachen?

Hinweis 1: Die Komplexität der 2D Fast Fouriertransformation (2DFFT) sowie ihrer Inversen für ein Bild der Grösse $M \times N$ ist $\mathcal{O}(MN \log(MN))$.