Nachrichtenübertragung - Softwarepraktikum
Termin 14:00 Uhr - 15:45 Uhr
Protokoll Nr. 5 (Projekt)

Gruppe “sw06c”

Tilman Knebel (Matr. Nr: 195298)
<tilman@cs.tu-berlin.de>

Christian Richter (Matr. Nr: 192548)
<richterc@cs.tu-berlin.de>

18. Juli 2003

Inhaltsverzeichnis

1 Tiefpaßfilterung eines eindimensionalen Signals 1

1.1 Die Funktion 1

1.2 Anwendungsbeispiele 3

2 Auflösungsänderung von Bildsignalen 5

2.1 Vorstellung verschiedener Methoden 5

2.1.1 Simples vervielfachen von Pixeln 8

Implementierung 8

Bewertung 9

2.1.2 Lineare Interpolation 11

Implementierung 11

Bewertung Vergrößerung 12

Bewertung Verkleinerung 13

2.1.3 Polynomielle Interpolation 13

Algorithmus 13

Implementierung 13

Bewertung 15

2.1.4 Faltung mit Bartlettfenster 15

Algorithmus 15

Implementierung 17

Bewertung 19

2.1.5 Faltung mit Gaussfenster 19

Algorithmus 19

Implementierung 20
Bewertung 21
2.1.6 Faltung mit Si-Funktion 21
Algorithmus 21
Implementierung 21
Bewertung 21
2.1.7 Spektrale Erweiterung durch Zero-Padding 23
Algorithmus 24
Implementierung 24
Bewertung 25
2.2 Vergleich der vorgestellten Methoden 27
2.2.1 Skalierungsgüte 27
Implementierung 27
Ergebnis 28
2.2.2 Zusammenfassung der Ergebnisse 29
2.3 Quellen 29

Zusammenfassung

Funktionen stellen in OCTAVE, genau wie in jeder anderen höheren Programmiersprache, eines der wichtigsten Mittel zur Stukturierung und Modularisierung von Programmen dar.
In der Funktion benutzte Variablen bleiben lokal, beeinflussen also den Rest des Programmes nicht. Auf diese Weise lassen sich Programmteile ausgliedern. Die Kommunikation mit dem Programm geschieht dann nur noch über eine streng definierte Schnittstelle. Code der in Funktionen ausgelagert wurde, kann an beliebigen Stellen im Programm beliebig oft aufgerufen werden.

Das letzte Protokoll dieser Veranstaltung widmet sich den beiden Funktionen, die im Rahmen des Programmierprojektes von uns zu erstellen waren.
In der ersten Aufgabe (Teil A in Kapitel 1) wird eine Funktion für die Tiefpaßfilterung eines eindimensionalen Signals beschrieben, während in der zweiten Aufgabe (Teil B in Kapitel 2) Algorithmen zur Auflösungsänderung von Bildern vorgestellt werden.

Kapitel 1

Tiefpaßfilterung eines eindimensionalen Signals

Es gibt verschiedene Ansätze, für die Filterung von Signalen. Einerseits kann man ein Filter im *Zeitbereich* erstellen, andererseits ist es auch möglich das Signal im *Frequenzbereich* zu bearbeiten.

Ein Filter im Zeitbereich ließe sich in Octave leicht mit dem Kommando ***filter*** realisieren, das als Parameter neben dem zu filternden Signalverlauf noch die Vektoren **a** und **b** erwartet, die die Filterkoeffizienten enthalten, durch die das Filter definiert ist. Enthält nur der Vektor **b** von null verschiedene Werte, spricht man von einem *nichtrekursiven* Filter (FIR), enthält auch **a** Koeffizienten ungleich null nennt man das Filter *rekursiv*. Weitere Ausführungen zu Filtern finden sich unter anderem im 2. Protokoll.

Im Gegensatz arbeitet eine Tiefpaßfilterung im Frequenzbereich direkt mit dem Spektrum des Signals. Dieses muß so manipuliert werden, das die hohen Frequenzen abgeschnitten bzw. entsprechend gedämpft werden.

1.1 Die Funktion

In MATLAB darf immer nur eine einzige Funktion in einer Datei stehen, die dann auch noch den selben Namen wie die Funktion tragen muß. In OCTAVE besteht diese Einschränkung glücklicherweise nicht, trotzdem sollte man sich daran halten um Kompatibilität zum “Quasi-Standard” zu wahren. Die untenstehende Funktion befindet sich also in einer Datei **tp.m**.

```
function y = tp(x, omega_g)
  % x       : Eingangssignal
  % omega_g : normierte Grenzfrequenz [0..pi]
```

```
% Eingagngssignal in Freq.-Bereich transformieren
spec_x = fft(x);
%Länge des Eingangsvektors bestimmen
nx = length(x);
% Filterfrequenzgang (Rechteck) erzeugen
rect = zeros(1, nx);
% "Knickkoeffizient" bestimmen
n = round(omega_g * nx / (2*pi));
% Tiefe Koeffizienten bleiben unverändert
rect(1:n+1) = 1;
offset = 1;
% Symmetriepaare erzeugen, damit
  Rücktransformierte real ist
rect(nx-n+offset:nx) = 1;
% Filtern des Spektrums
spec_y = spec_x .* rect;
% Rücktransformieren und Imaginärteil
  vernachlässigen
y = real(ifft(spec_y));
```

Das Schlüsselwort **function** leitet die Funktion ein. Es werden die Parameter **x** (Signal) und **omega_g** (normierte Grenzfrequenz ω_g) erwartet und das tiefpaßgefilterte Signal (**y**) zurückgegeben.

Zunächst wird das Eingangssignal mittels **fft** in den Frequenzbereich transformiert. Als nächstes wird ein Frequenzgang mit der gleichen Länge (Elementanzahl N) wie das Signalspektrum zusammengesetzt, der die gewünschte Tiefpaßcharakteristik besitzt. Dafür wird als erstes ein Vektor durch **zeros** mit der entsprechenden Anzahl Nullen gefüllt.

Der resultierende Frequenzgang wäre im Moment noch konstant Null, es würde also gar keine Frequenzen durchgelassen werden. Aus diesem Grund werden in einem zweiten Schritt alle Koeffizienten für Frequenzen zwischen $-\omega_g$ und ω_g auf Eins angehoben. Da das diskrete Spektrum periodisch fortgesetzt ist, entspricht $X(-j\omega_g) = X(-j(\omega_g + 2\pi))$, und die entsprechenden Arrayindizes betragen $\frac{\omega_g N}{2\pi} + 1$ und $N - \frac{\omega_g N}{2\pi} + 1$. Die Addition von 1 ist nötig, da das erste Element den ersten Fourierkoeffizienten, also den Mittelwert des Signals im Zeitbereich darstellt, welcher durch die Filterung nicht verändert wird.

Gefiltert wird das Signal $x(t)$ dann, indem man sein Spektrum $X(j\omega)$ mit dem erstellten Frequenzgang $H(j\omega)$ multipliziert.

$$H(j\omega) = \frac{Y(j\omega)}{Xj\omega)} \Rightarrow Y(j\omega) = X(j\omega) \cdot H(j\omega)$$

Das entstehende Spektrum $Y(j\omega)$ wird dann noch mittels der inversen Fouriertransformation **ifft** rücktransformiert um das Ausgangssignal $y(t)$ zu erhalten.

1.2 Anwendungsbeispiele

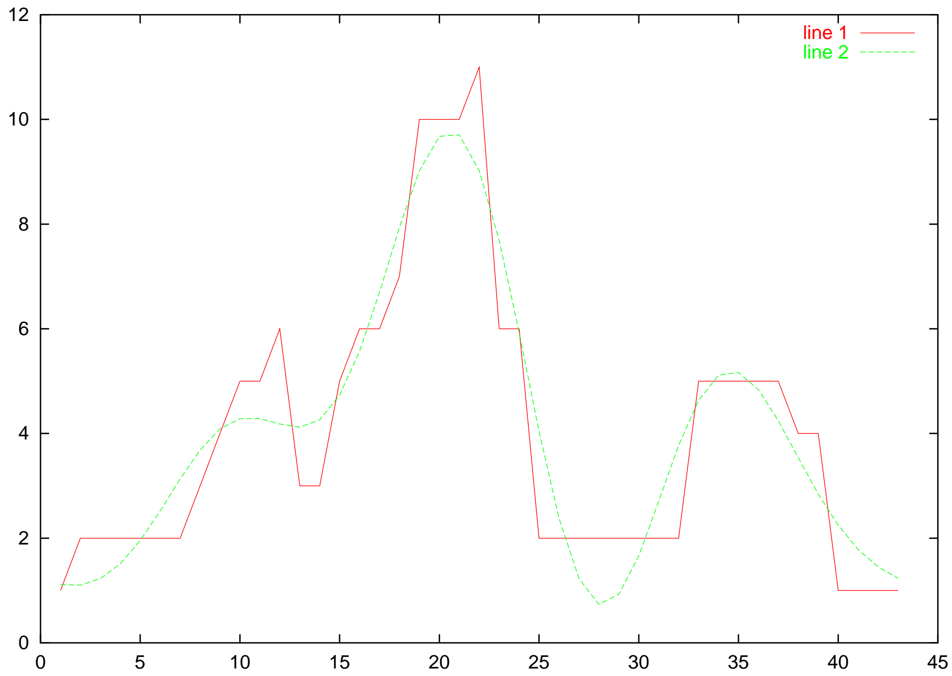
Um die Funktion zu testen, wird ein synthetisch erzeugter Signalverlauf tiefpaßgefiltert. Dabei wird einmal eine hohe und einmal eine niedrige Grenzfrequenz angesetzt. Folgender Code erzeugt das Signal und ruft die in Abschnitt 1.1 erläuterte Funktion **tp** zur Tiefpaßfilterung auf.

```
% Signal erzeugen
x = [1 2 2 2 2 2 2 3 4 5 5 6 3 3 5 \
     6 6 7 10 10 10 11 6 6 2 2 2 2 \
     2 2 2 2 5 5 5 5 5 4 4 1 1 1 1];
% Filtern und plotten (omega_g = 0.8)
t08=tp(x,0.8); % Filtern
clg;
hold on;
plot(x);
plot(t08);
hold off;
toFile("tp_08-plot.ps");
% Filtern und plotten (omega_g = 0.4)
t04=tp(x,0.4); % Filtern
clg;
hold on;
plot(x);
plot(t04);
hold off;
toFile("tp_04-plot.ps");
```

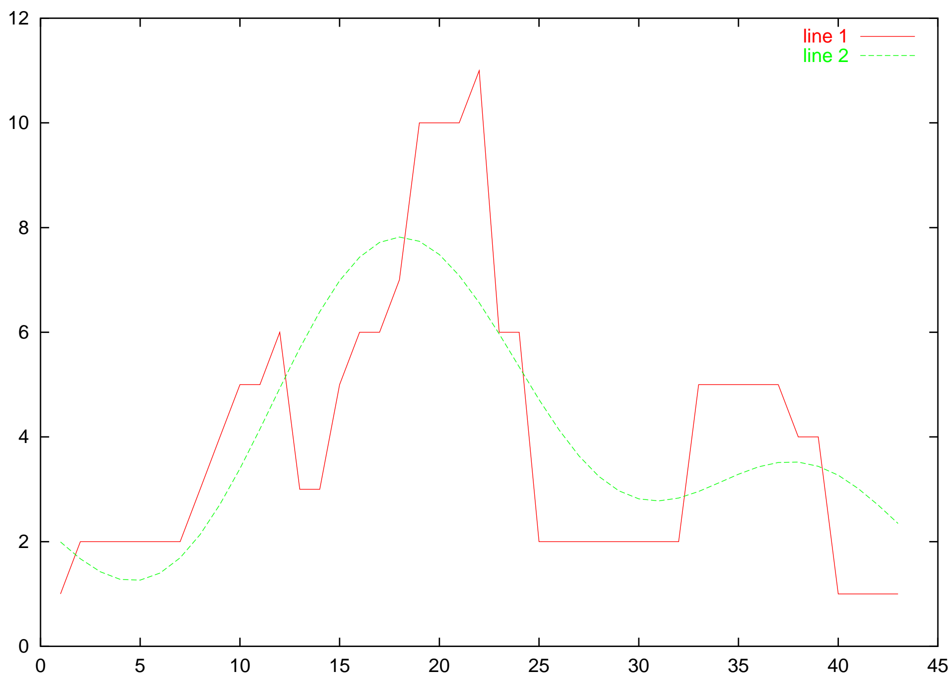
In diesem Beispiel wird das Signal einmal mit einer Grenzfrequenz von $\omega_g = 0,8$ und einmal mit $\omega_g = 0,4$ gefiltert. Die Ergebnisse sind in Bild zu sehen. Die roten Kurven stellen dabei jeweils das Originalsignal dar, während die grünen das tiefpaßgefilterte Signal anzeigen.

Man erkennt deutlich, daß das gefilterte Signal dem Original um so genauer folgt, je höher die Grenzfrequenz ist.

Abbildung 1.1: Tiefpaßfilterung mit verschiedenen Grenzfrequenzen



(a) Grenzfrequenz 0.8



(b) Grenzfrequenz 0.4

Kapitel 2

Auflösungsänderung von Bildsignalen

Es sollen Methoden erarbeitet werden, mit denen sich ein gegebenenes Bild um einen ganzzahligen Faktor s in beiden Dimensionen skalieren läßt. Dazu werden Pixel entfernt oder neue hinzugefügt, so daß sich die Aulösung des Bildes ändert. Im eindimensionalen Fall entspricht dies einer Änderung der Samplingrate, also einem Up- oder Downsampling. Dabei ist die ideale Interpolation so definiert, daß das Bild vor dem Verkleinern bzw. nach dem Vergrößern mit $\omega_g = \frac{\omega_t}{2s}$ tiefpaßgefiltert wird. Nur so ist gewährleistet, daß die Frequenzkomponenten möglichst originalgetreu erhalten werden.

Um diese Ziele anzunähern oder durch Vorwissen oder Abschätzungen sogar zu übertreffen, sind verschiedene Algorithmen denkbar, von denen einige im Folgenden vorgestellt werden.

2.1 Vorstellung verschiedener Methoden

Allen Methoden gemein ist, daß zunächst die Größe des Bildes ermittelt werden muß. Außerdem werden einige temporäre Variablen immer wieder verwendet. Aus diesen praktischen Erwägungen heraus haben wir uns schließlich dafür entschieden, alle im Folgenden vorgestellten Methoden zur Größenänderung eines Bildes in einer einzigen Funktion namens ***resample*** zu implementieren.

Der grundlegende Aufbau sieht dabei folgendermaßen aus.

```
function bild_neu = resample(bild, scale, method)
   % bild   = Matrix mit Bildpunkten
   % scale  = ganzzahliger Skalierungsfaktor
   % method = Algorithmus
   [x,y] = size(bild); % Groesse bestimmen
```

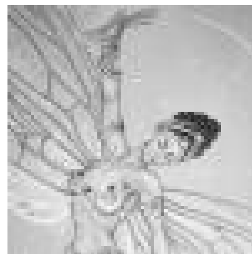
```
bild_neu = []; % skaliertes Bild (Rueckgabewert)
bild_tmp = []; % Speicher für Zwischenschritt

if (scale > 0)  % vergroessern
   if (method == 0) % Methode 0 (simple)
      % ... siehe Abschnitt 2.1.1
   end;
   if (method == 1) % Methode 1 (Lineare Interpol.)
      % ... siehe Abschnitt2.1.2
   end;
   if (method == 2) % Methode 2 (Polynom. Interpol.)
      % ... siehe Abschnitt 2.1.3
   end;
   if (method == 3) % Methode 3 (Bartlettfilterung)
      % ... siehe Abschnitt 2.1.4
   end;
   if (method == 4) % Methode 4 (Gauss-Filterung)
      % ... siehe Abschnitt 2.1.5
   end;
   if (method == 5) % Methode 5 (Si-Filterung )
      % ... siehe Abschnitt 2.1.6
   end;
   if (method == 6) % Methode 6 (Spektrale Erweit.)
      % ... siehe Abschnitt 2.1.7
   end;
else  % verkleinern
   scale = abs(scale);
   if (method == 0) % Methode 0 (simple)
      % ... siehe Abschnitt 2.1.1
   end;
   if (method == 1) % Methode 1 (Mittelwertbildung)
      % ... siehe Abschnitt 2.1.2
   end;
end;
```

Alle Verfahren sind also durchnumeriert und lassen sich durch Übergabe eines entsprechenden Parameters ([0..6] vergrößern, [0..1] verkleinern) auswählen. Dieses Vorgehen hat den großen Vorteil, daß der Benutzer sich nicht etliche verschiedene Funktionsnamen merken muß. Um ein Bild zu skalieren gibt man einfach ***resample*** ein, der konkrete Algorithmus wird dann, genau wie die Skalierung, nur noch durch die Parameter bestimmt.

Abbildung 2.1: Testbild

(a) Originalgröße (300x300)



(b) Verkleinerung (75x75)

Folgender OCTAVE-Code wurde von uns dazu benutzt ein Testbild zu laden, die jeweiligen Resampling-Algorithmen darauf anzuwenden und die Ergebnismatrix als Bild darzustellen:

```
scale = 4;  % Skalierungsfaktor (Beispiel)
method = 0; % Algorithmus (Beispiel)
% Bild laden
fid = fopen("frau300.byt", "r");
frau = fread(fid, [300,300], "uchar")';
fclose(fid);
% Originalbild darstellen
colormap(gray(256));
image(frau);
% Groesse aendern
frau_neu = resample(frau, scale, method);
% Ergebnisbild darstellen
image(lena_neu);
```

Die Skalierung muß, entsprechend der Vorgabe, ein ganzzahliger Faktor mit einem Betrag größer Null sein. Ein positiver Faktor signalisiert Vergrößerung, ein negativer Verkleinerung.

Das Original des Testbildes (dargestellt in Abbildung 2.1 a) hat eine Auflösung von 300x300 Bildpunkten. Es dient damit quasi als Referenz, an der sich alle im Folgenden gezeigten Vergrößerungalgorithmen messen lassen müssen. Diese werden mit einer um den Faktor 4 auf 75x75 Pixel verkleinerten Version dieses Bildes arbeiten, das in Abbildung 2.1b zu sehen ist

2.1.1 Simples vervielfachen von Pixeln

Zur Vergrößerung wird jeder Bildpunkt in einen quadratischen Bereich von s^2 vervielfacht. Dabei wird jedes Pixel für sich betrachtet, es findet keine Interpolation statt. Die Verkleinerung geschieht durch Selektion der Pixel, die in einem Raster der Kantenlänge s liegen. Alle anderen Bildpunkte werden ignoriert.

Implementierung

Um den Algorithmus kurz zu halten und die Abarbeitung in OCTAVE zu beschleunigen wurden nicht einzelne Pixel, sondern ganze Zeilen bzw. Spalten betrachtet. Für die Vergrößerung um den Faktor n werden also zunächst alle Zeilenvektoren der Bildmatrix ver-n-facht. In der resultierende Matrix werden dann alle Spaltenvektoren ebenfalls ver-n-facht, so daß sich das fertige Bild ergibt.

```
% Zeilen vervielfachen
 for i=1:x
    for j=1:scale
       bild_tmp = [bild_tmp ; bild(i,:) ];
    end;
 end;
% Spalten vervielfachen
 for i=1:y
   for j=1:scale
      bild_neu = [bild_neu , bild_tmp(:,i) ];
   end;
 end;
```

Die Verkleinerung verläuft analog. Es wird jede n-te Zeile in eine temporäre Matrix kopiert, von der dann wiederum jede n-te Spalte als skaliertes Bild zurückgegeben wird.

```
% Zeilen selektieren
 for i=1:scale:x
    bild_tmp = [bild_tmp ; bild(i,:) ];
 end;
% Spalten selektieren
 for i=1:scale:y
    bild_neu = [bild_neu , bild_tmp(:,i) ];
 end;
```

Bewertung

Das vergrößerte Bild (zu sehen in Abbildung 2.2 a) erinnert an die üblichen Skalierungen, die in Bildbearbeitungsprogrammen als Lupenfunktion verwendet werden. Jedes Originalpixel ist als solches identifizierbar. Der Nachteil dabei ist, daß das Auge beim Betrachten die horizontalen und vertikalen Pixelgrenzen als zusätzliche Konturen interpretiert und die wahren Konturen etwas untergehen. Im Vergleich zur idealen Interpolation wird hier statt der Tiefpaßfilterung ein ideales Halteglied angewendet, wodurch zusätzliche Frequenzen auch oberhalb der ursprünglichen Nyquistfrequenz entstehen.

Die Verkleinerung (Abbildung 2.1 b) funktioniert vor allem bei Fotos tadellos, es ist jedoch leicht vorstellbar das es Bilder geben könnte, die Informationen ausgerechnet an den zu verwerfenden Pixeln enthalten, womit diese Informationen also komplett verlorengehen. Im Vergleich zum idealen Downsampling sind hier zu viele Frequenzkomponenten verlorengegangen.

Abbildung 2.2: Vergrößerungen (300x300 Pixel)

(a) Methode Pixelvervielfachung

(b) Methode Lineare Interpolation

Der Vorteil ist natürlich, daß der Algorithmus extrem einfach und damit schnell berechenbar ist.

2.1.2 Lineare Interpolation

Bei der linearen Interpolation werden, genau wie bei der Pixelvervielfachung Zwischenbildpunkte eingesetzt. Allerdings werden diese Zwischenpunkte nicht durch einfaches Duplizieren sondern durch (lineare) Interpolation der Farbwerte zweier benachbarter Pixel gewonnen. Ein Problem dabei sind die Randpixel, da hier jeweils nur ein Wert zur Verfügung steht. Dieser Fall wird gesondert behandelt und die Pixel aus der letzten originalen Bildzeile werden einfach dupliziert, so daß um das interpolierte Bild quasi ein Rahmen entsteht, der aus den anliegenden Farbwerten besteht.

Für die Verkleinerung wird das Bild in quadratische Blöcke der Kantenlänge s geteilt und jeweils der mittlere Farbwert gebildet. Der für diesen Block repräsentative Pixel im Zielbild erhält dann den errechneten Mittelwert.

Implementierung

Praktisch geschieht dies für beide Dimensionen nacheinander.

```
% Zeilen einfügen
for i=1:x-1
  delta = ( bild(i+1,:)-bild(i,:) ) / scale;
  for j=0:scale-1
    bild_tmp = [bild_tmp ; bild(i,:)+(j*delta) ];
  end;
end;
%gleichmäßiges Erweitern auf Zielgröße (oben+unten)
for j = 0:scale-1
  if ( rem(j,2)==0 ) % jede zweite einzufügende
                       Zeile oben anfügen
    bild_tmp = [bild(1,:) ; bild_tmp];
  else  % jede zweite einzufügende
            Zeile unten anfügen
    bild_tmp = [bild_tmp ; bild(x,:) ];
  end
end;
% Spalten vervielfachen
for i=1:y-1
  delta = ( bild_tmp(:,i+1)-bild_tmp(:,i) ) / scale;
  for j = 0:scale-1
```

```
    bild_neu = [bild_neu, bild_tmp(:,i)+(j*delta)];
  end;
end;
%gleichmäßiges Erweitern auf Zielgröße (oben+unten)
for j=0:scale-1
   if ( rem(j,2)==0 )  % jede zweite einzufügende
                         Spalte vorn anfügen
     bild_neu = [bild_tmp(:,1), bild_neu];
   else  % jede zweite einzufügende Zeile
           hinten anfügen
     bild_neu = [bild_neu , bild_tmp(:,y) ];
   end
end;
```

Die Verkleinerung geschieht für diese Methode für jedes Pixel im Zielbild durch Zusammenfassen jeweils eines quadratischen Pixelblocks und anschliessender Mittelwertbildung. Auch hier wird wieder Zeilen- bzw. Spaltenweise vorgegangen.

```
% Zeilen zusammenfassen (mitteln)
rest = rem(x,scale);
for i = 1:scale:x-rest
  bild_tmp = [ bild_tmp;
               mean(bild(i:i+scale-1, :)) ];
end;
if ( rest>0 )
  bild_tmp = [ bild_tmp;  % Restzeilen mitteln
               mean(bild(x-rest+1:x, :)) ];
end
% Spalten zusammenfassen (mitteln)
rest=rem(y,scale);
for i = 1:scale:y-rest
  bild_neu = [ bild_neu,
               mean(bild_tmp(:,i:i+scale-1)')' ];
end;
if(rest>0)
  bild_neu = [ bild_neu,
               mean(bild_tmp(:, x-rest+1:x)')' ];
end
```

Bewertung Vergrößerung

Die durch die in Abschnitt 2.1.1 beschriebene einfache Vervielfachung der Pixel entstehenden Konturen (Pixelgrenzen) sind hier bei der Linearen Interpolation (Abbil-

dung 2.2 b) stark gedämpft. Bei starker Vergrößerung lassen sich jedoch immer noch horizontale und vertikale Strukturen erkennen, die durch die Skalierung entstehen.

Im Verhältnis zur idealen Interpolation sind auch hier zusätzliche Frequenzkomponenten oberhalb der ursprüngl. Nyquistfrequenz entstanden.Vorteilhaft ist aber die trotz allem weiterhin halbwegs schnelle Berechenbarkeit bei recht guter Qualität.

Bewertung Verkleinerung

Die Verkleinerung durch Mittelwertbildung verwertet – im Gegensatz zur Verkleinerung durch Auslassen von Bildpunkten – jedes Pixel und ist damit relativ gut.
Im Ergebnis läßt sich nichts Negatives entdecken. Grundsätzlich ist die Methode für ganzzahlige Verkleinerungen optimal, da jedes Pixel in seinen örtlich nächsten Representanten im Zielbild einfließt. Die Mittelwertbildung entspricht einer Mittelwertfilterung mit rechteckförmiger Impulsantwort, womit die für ideale Interpolation erforderliche ideale Tiefpaßfilterung angenähert wird.

2.1.3 Polynomielle Interpolation

Es ist möglich, k beieinanderliegende Pixel in einem Bild durch ein Polynom k-ten Grades funktional exakt darzustellen. Die Zwischenwerte können dann durch die Auswertung des Polynoms an den gewünschten Koordinaten errechnet werden. Dies ist für den zweidimensionalen Fall durch Trennung in horizontale und vertikale Pixelgruppen möglich.

Algorithmus

Konkret wurde ein Beispiel programmiert, indem fünf horizontale oder vertikale Nachbarpixel zur Generierung eines Polynoms herangezogen werden, das dann zwischen dem zweiten und dritten Pixel Interpolationswerte erzeugt. Im Randbereich treten – genau wie bei der Linearen Interpolation (Abschnitt 2.1.2) – Schwierigkeiten auf, da Pixelwerte fehlen. Konkret wurde hier das Problem durch Anwendung der Linearen Interpolation im Randbereich umgangen.

Implementierung

```
% Zeilen einfügen
gu=2;
for i = 1:x-1
  if ( i>gu & i<x-gu )
    po=zeros(y,scale);
```

```
    for i1 = 1:y
      po(i1,:) = polyval( polyfit(
                           -scale*gu:scale:scale*(gu+1),
                          0:scale-1);
      end
      po=po';
      bild_tmp=[bild_tmp ; po];
  else
    delta = ( bild(i+1,:)-bild(i,:) ) / scale;
    for j = 0:scale-1
      bild_tmp = [bild_tmp ; bild(i,:)+(j*delta) ];
    end;
  end
end;
%gleichmäßiges Erweitern auf Zielgröße (oben+unten)
for j = 0:scale-1
  if (rem(j,2)==0) % Index über scale gerade
    bild_tmp = [bild(1,:) ; bild_tmp];
  else  % Index über Skalierungsfaktor ungerade
    bild_tmp = [bild_tmp ; bild(x,:) ];
  end
end;
% Spalten einfügen
[x,y]=size(bild_tmp);
for i = 1:y-1
  if ( i>gu & i<y-gu )
    po=zeros(x,scale);
    for i1 = 1:x
      po(i1,:)=polyval(polyfit(
                          -scale*gu:scale:scale*(gu+1),
                          bild_tmp(i1,i-gu:i+gu+1),
                          2*gu+1),
                         0:scale-1);
    end
    bild_neu=[bild_neu,po];
  else
    delta = ( bild_tmp(:,i+1)-bild_tmp(:,i) ) / scale;
    for j = 0:scale-1
      bild_neu=[bild_neu, bild_tmp(:,i)+(j*delta)];
    end;
  end
end;
%gleichmäßiges Erweitern auf Zielgröße (oben+unten)
```

```
for j = 0:scale-1
  if rem(j,2)==0  % Index über scale gerade
    bild_neu = [bild_tmp(:,1) , bild_neu];
  else  % Index über Skalierungsfaktor ungerade
    bild_neu = [bild_neu , bild_tmp(:,y) ];
  end
end;
```

Bewertung

Das Ergebnis, welches in Abbildung 2.3 a dargestellt ist, sieht sehr sauber aus. Lediglich in den Randbereichen sind horizontale und vertikale Strukturen zu erkennen, die von der Linearen Interpolation stammen. Ansonsten sind die Pixel im Gegensatz zur dieser nicht rautenförmig sondern rund, so daß schräge Linien keine Zacken, sondern nur ganz leiche Wellen zeigen.

Prinzipiell unterscheidet sich die Methode von der Linearen Interpolation dadurch, daß die neu eingefügten Farbwerte einen glatten, weicheren Verlauf haben. Die zusätzlich eingefügten Frequenzen sind schwächer als für die ersten beiden Verfahren. Für diesen Algorithmus wird eine statistische Abhängigkeit der einzufügenden Pixel von weiter entfernten Nachbarpixeln als Modellannahme vorausgesetzt. Das ist plausibel, denn um eine Linie oder Kontur formerhaltend zu vergrößern, muß die Kontur streckenweise verfolgt werden.

2.1.4 Faltung mit Bartlettfenster

Eine verallgemeinerte Skalierungsmethode basiert auf einfacher Matrixskalierung durch zero-padding mit anschliessender Tiefpaßfilterung. Die Filterung soll die zusätzlich entstandenen Frequenzen oberhalb der ursprünglichen Nyquistfrequenz verringern.

Im Folgenden wird die Filterung im Zeitbereich durch Faltung mit einer im Idealfall si-förmigen, zweidimensionalen Fensterfunktion (Abbildung 2.4) durchgeführt.
Die örtliche Größe der Faltungsfunktion bestimmt, wieviele Nachbarpixel k zur Berechnung der Interpolation beitragen sollen. Sie läßt sich berechnen durch $k \cdot s - 1$. Je größer die Faltungsmatrix ist, desto stärker tritt eine allgemeine Unscharfmaskierung auf.

Algorithmus

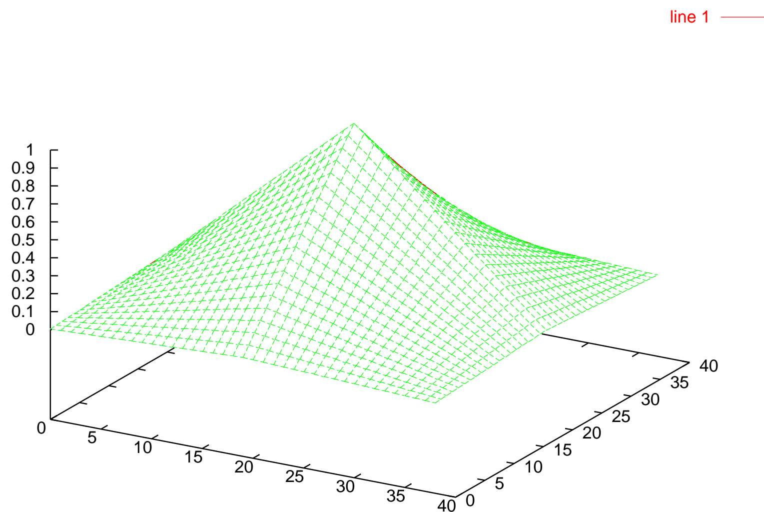
Es wird zunächst eine Null-Matrix der Zielbildgröße erzeugt. Dann wird die Bildinformation genau an den skalierten Koordinaten eingefügt. Dieses Bild wird mit der

Abbildung 2.3: Vergrößerungen (300x300 Pixel)

(a) Methode Polynomielle Interpolation

(b) Methode Bartlett-Filterung

Abbildung 2.4: Bartlet-Fensterfunktion



Fensterfunktion gefaltet, die im Fall der Bartlettfilterung durch eine pyramidenförmige Struktur mit einer maximalen Intensität von 1 (Abbildung 2.4) angenähert wird. Im Folgenden wird die Größe der Faltungsmatrix für $k = 2$ bestimmt. Die Kantenlänge beträgt also $2 \cdot s - 1$.

Implementierung

```
bild_neu = conv2(padding(bild, scale, scale),
                    bartlett(scale, scale) );
bild_neu = bild_neu(scale+1:end-scale+1,
                       scale+1:end-scale+1 );
```

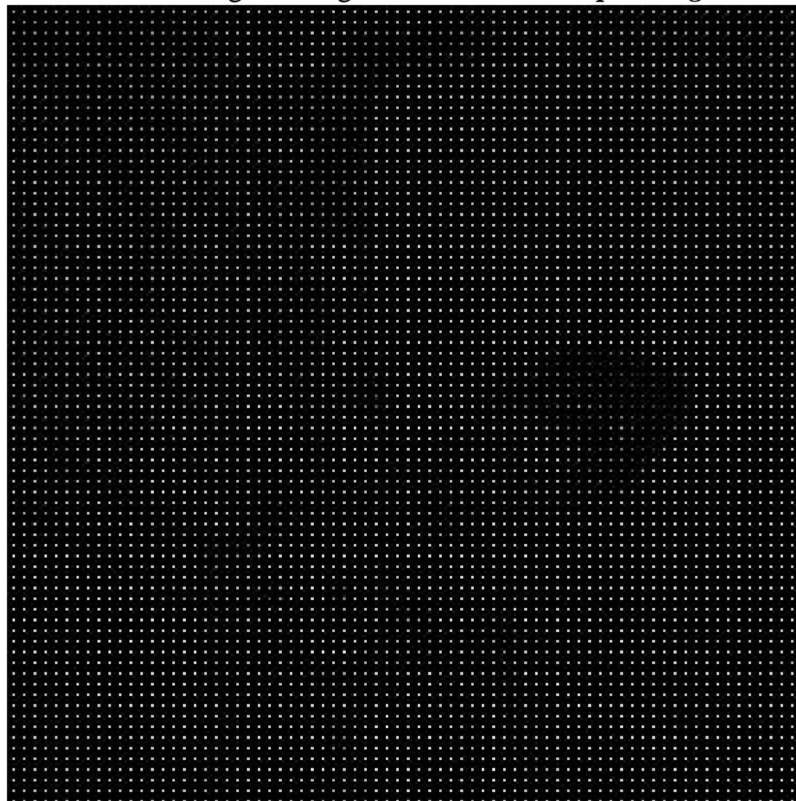
Die eigentliche Implementierung der Bartlett-Fensterfunktion ist hier der Übersichtlichkeit halber in eine externe Funktion Namens ***bartlett*** verlagert worden, die folgendermaßen aussieht:

```
function result = bartlett(x, y)
  row_vect = zeros(1, x*2-1);
  col_vect = zeros(y*2-1, 1);
  x_len = length(row_vect);
  for i=1:x_len
    if (i <= x)
      row_vect(i) = i;
```

```
  else
    row_vect(i) = x_len-i+1;
  end
end
y_len = length(col_vect);
for i=1:y_len
  if (i <= y)
    col_vect(i) = i;
  else
    col_vect(i) = y_len-i+1;
  end
end
result = 1/(x*y) .* (col_vect * row_vect);
```

Die Funktion ***padding*** skaliert die Matrix auf Zielgröße, wobei alle Werte gleich null sind, nur die Pixel aus dem Originalbild werden an den proportional gestreckten Koordinaten Rasterförmig eingefügt, wobei sich dann das in Abbildung 2.5 gezeigte Bild ergibt.

Abbildung 2.5: Ergebnis der Funktion ***padding***



```
function padded_im = padding(image, x, y)
   padded_im = zeros( size(image, 1)*y+1,
                      size(image, 2)*x+1 );
   for i = 1:size(image, 1)
     for j = 1:size(image, 2)
        padded_im(y*i, x*j) = image(i, j);
     end
   end
```

Bewertung

Das verwendete Bartlettfenster erzeugt genau das gleiche Bild wie die Lineare Interpolation. Lediglich in den Randbereichen ist hier eine Tendenz zu Schwarz zu erkennen (Abbildung 2.3 b). Das liegt daran, daß für die Faltung Pixelwerte außerhalb des Bildes verwendet werden müssen, für die hier der Farbwert Null angenommen wird. Da die Fensterfunktion linear ist, ergibt sich für die interpolierten Werte das gleiche Ergebnis wie bei linearer Interpolation und da die Fenstergröße gerade bis zu den Nachbarpixeln aus dem Originalbild heranreicht, ergeben sich die interpolierten Werte nur durch die beiden nächsten Nachbarpixel horizontal und vertikal.

2.1.5 Faltung mit Gaussfenster

Als Näherung für die Si-Funktion aus Abschnitt 2.1.4 kann auch die zweidimensionale Gaussfunktion (Abbildung 2.6)

$$g(x, y) = exp(-\frac{1 \cdot \sigma^2}{2} \cdot ((x - \mu_x)^2 + (y - \mu_y)^2)$$

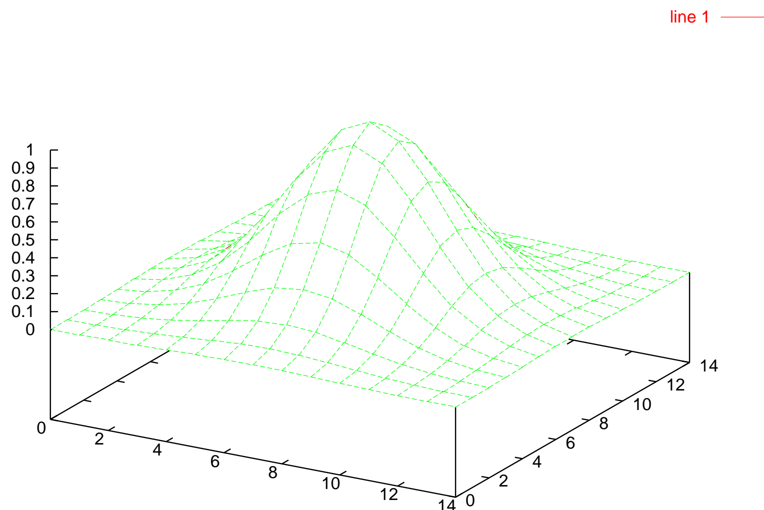
genutzt werden.

Algorithmus

Die Gaussglocke wird so generiert, daß sie für die Berechnung der Interpolationswerte jeweils sechs Nachbarpixel horizontal und vertikal nutzt, in jede Richtung also drei Pixel in die Rechnung eingehen. Im Bereich darüberhinaus wird sie abgeschnitten, so daß zur Erhaltung der Gesamtintensität ein Faktor eingeführt wird.

Die Varianz σ der Glocke wird so gewählt, daß jedes Pixel durch eine Gaussglocke repräsentiert wird, die gerade den mittleren Bereich zwischen den Originalpixeln ausfüllt.

Abbildung 2.6: Gauss-Fensterfunktion



Implementierung

```
bild_neu = conv2( padding(bild, scale, scale),
                  gauss(scale, scale, 0.4,s) );
bild_neu = bild_neu(scale*sh+1:end-scale*sh+1,
                    scale*sh+1:end-scale*sh+1)/2.425;
```

Auch hier wird zum Erzeugen der Fensterfunktion eine extra Funktion – in diesem Fall mit Namen ***gauss*** – benutzt, deren Quelltext im Folgenden dargestellt ist.

```
function result = gauss(x, y, o, s)
  sh=s/2;
  result = zeros(x*s-1, y*s-1);
  [x_len,y_len] = size(result);

  for xi=1:x_len
     for yi=1:y_len
       result(xi,yi) = exp(-o*o*(
                          (xi-sh*x)*
                          (xi-sh*x)+(yi-sh*y)*
                          (yi-sh*y)) /2);

     end
  end
```

Bewertung

Da die entstehende Gaussglocke symmetrisch zum Ursprung ist steht zu erwarten, daß jedes Pixel im Zielbild (Abbildung 2.7 a) nicht eckig, sondern möglichst rund repräsentiert ist und das Ergebnis entspricht dieser Erwartung.

Durch die große Kantenlänge der Faltungsmatrix ist eine leichte Unscharfmaskierung zu erwarten. Ein Vergleich mit dem durch Lineare Interpolation (Abschnitt 2.1.2) entstandenen Bild bestätigt diese Vermutung.

2.1.6 Faltung mit Si-Funktion

Die ideale Interpolation ist aus der Nachrichtenübertragung bekannt als:

$$u(t) = \sum u(kT) \cdot \delta(t - kT) * si(\frac{\omega_T t}{2})$$

Dies soll hier anhand einer über das gesamte Bild ausgedehnten Si-Funktion (Abbildung 2.8) ausprobiert werden.

Algorithmus

Es wird wieder die in den Abschnitten 2.1.4 und 2.1.5 erwähnte Faltungsoperation angewendet, diesmal aber für eine Si-Funktion, deren Grenzfrequenz auf den Vergrößerungsfaktor eingestellt wird, sich also zu $\omega_T = \frac{2\pi}{s}$ ergibt.

Implementierung

```
s = x;
bild_neu = conv2( padding(bild, scale, scale),
                  si(s,s,scale) );
bild_neu = bild_neu(s+1:end-s-1, s+1:end-s-1);
```

Bewertung

Das Ergebnis (Bild 2.7 b) enthält starke Wellenstrukturen am Randbereich, da die Faltung sich auch auf Bereiche außerhalb des eigentlichen Bildes bezieht, die hier als schwarz angenommen werden. Der dabei entstehende hohen Kontrast ist für diese starke Welligkeit, die auch in Bildteilen mit auffälligen Kanten zu erkennen ist, verantwortlich. Ansonsten ist das Bild schärfer als in den anderen Verfahren.

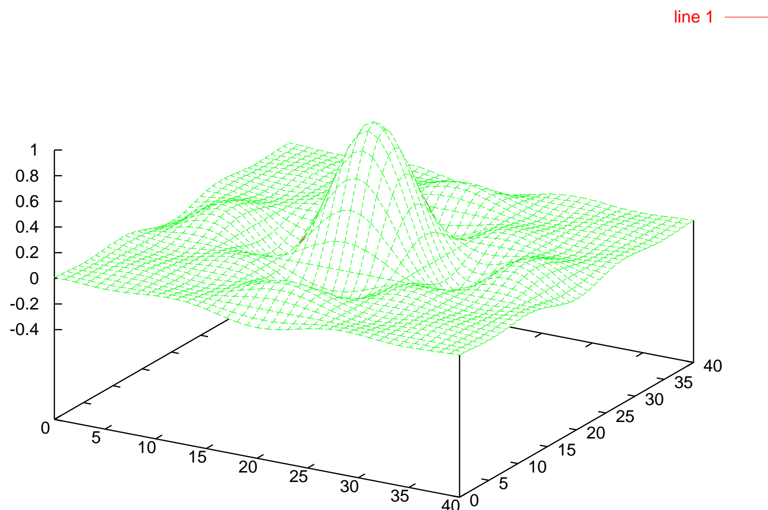
Die Wellenbildung an Kanten ist offensichtlich für die Bildverarbeitung im Gegensatz zu Audiosignalen ein großes Problem, dennoch ist das Verfahren korrekt: Eine

Abbildung 2.7: Vergrößerungen (300x300 Pixel)

(a) Methode Gauss-Faltung

(b) Methode Si-Faltung

Abbildung 2.8: Si-Fensterfunktion



Sprungfunktion im Audiobereich hat Energien für Frequenzen von 0 bis zur Nyquistfrequenz. Wird ein ideales Upsampling durchgeführt, dürfen im Zielsignal keine Energien oberhalb der ursprünglichen Nyquistfrequenz auftreten, also wird die Funktion etwas geglättet und damit wellig.

Um die Welligkeit zu vermeiden müßte das Spektrum oberhalb der ursprünglichen Nyquistfrequenz Si-förmig fortgesetzt werden, was der Up-Sampler aber nicht wissen kann. Demnach ist die Ideale Interpolation für Bildsignale ungeeignet.

2.1.7 Spektrale Erweiterung durch Zero-Padding

Ein alternativer Ansatz zur Bildskalierung wäre, das Bild im Spektralbereich so zu manipulieren, daß alle Komponenten aus dem Originalbild ins skalierte Bild übertragen werden und den zusätzlich entstandenen hohen Spektralbereich mit einem geschätzten Spektrum aufzufüllen. Das neue, vergrößerte Spektrum kann dann in den Ortsbereich zurücktransformiert werden.
Problematisch dabei ist, daß die Fouriertransformation für periodische Signale gilt, das Bild also als periodisch angenommen wird, was sich auf das Zielbild entsprechend auswirkt. Sofern das Spektrum mit Nullen aufgefüllt wird, entspricht die Methode abgesehen von der Periodizität der in Abschnitt 2.1.6 beschriebenen Faltung mit Si-Funktion.

Algorithmus

Um die oben erwähnten Wellenstrukturen am Rand zu vermeiden, soll hier das Bild mit mehreren Spiegelungen seiner selbst umgeben werden, so daß in den Randbereichen keine starken Kontraste auftreten. Weiterhin wird das Spektrum in den durch die Vergrößerung entstandenen neuen hohen Frequenzbereichen mit Nullen gefüllt.

Implementierung

```
%Bild ganz oft vervielfachen und dabei spiegeln,
 bis Zielbild quadratisch und doppelt gross
xt = x;
yt = y;
sx = 1; % obere linke Ecke des Originalbildes
sy = 1;
if ( xt == yt ) % Bild quadratisch
  sx = sx+xt;
  sy = sy+yt;
  xt = 3*xt;
  yt = 3*yt;
  bild = [flipud(bild); bild; flipud(bild)];
  bild = [fliplr(bild), bild, fliplr(bild)];
else % Bild nicht quadratisch
  % schrittweise vergrößern
  while ( xt < yt )
    sx = sx+xt;
    xt = 3*xt;
    bild = [flipud(bild); bild; flipud(bild)];
  end
  while ( yt < xt )
    sy = sy+yt;
    yt = 3*yt;
    bild = [fliplr(bild), bild, fliplr(bild)];
  end
  if( xt < yt )
    delta=(yt-xt)/2;
    bild=bild(:,floor(delta):yt-ceil(delta));
    yt=yt-2*delta;
    sy=sy-floor(delta);
  end
  if ( yt < xt )
    delta = (xt-yt)/2;
    bild = bild(floor(delta):xt-ceil(delta), :);
```

```
    xt = xt-delta;
    xs = sx-floor(delta);
  end
end
%Bild muss unbedingt ungeradzahlige Breite
 und Hoehe haben
if ( rem(xt,2) == 0 )
  bild = bild(1:xt-1,:);
  xt = xt-1;
end
if ( rem(yt,2) == 0 )
  bild = bild(:, 1:yt-1);
  yt = yt-1;
end
%Berechnung des Spektrums und Auffüllen mit Nullen
rein = (scale-1)*xt;
fbild = fft2(bild);
xt1 = (xt+1)/2;
f2bild = [ fbild(1:xt1,1:xt1),
           zeros(xt1,rein),
           fbild(1:xt1,xt1+1:xt),
           zeros(rein,xt+rein),
           fbild(xt1+1:xt,1:xt1),
           zeros(xt1-1,rein),
           fbild(xt1+1:xt,xt1+1:xt) ];
bild_neu = real(ifft2(f2bild));
bild_neu = bild_neu*(scale*scale);
% Heraussuchen des Originalbildes in der Mitte
  des Ergebnissbildes
sx = (sx-1)*scale;
sy = (sy-1)*scale;
x = x*scale;
y = y*scale;
bild_neu = bild_neu(sx-1:sx+x-2, sy-1:sy+y-2);
```

Bewertung

Im Ergebnis in Abbildung 2.9 a erkennt man – im Gegensatz zu den vorangegangenen Methoden – sehr scharfe Konturen in beliebigen Richtungen. Die ursprünglichen Pixel sind nicht mehr zu erkennen. Ein Nachteil sind jedoch kleine Flecken und Wellen in Bereichen mit hohem Kontrast, die wegen der Rechteckfensterung und den Nullstellen der dazugehörigen Si-Funktion entstanden sind. Ein weiterer Nachteil dieser Methode

Abbildung 2.9: Vergrößerung (300x300 Pixel)

(a) Methode Zero-Padding

ist der sehr hoher Rechenaufwand, da das gesamte Bild fouriertransformiert werden muß.

2.2 Vergleich der vorgestellten Methoden

In einem weiteren Skript wird ein Originalbild $Obild$ mit 300*300 Pixeln durch einfaches Auslassen von Pixeln (Abschnitt 2.1.1) um dem Faktor 4 verkleinert und anschließend mit allen hier angesprochenen Methoden nach $Sbild$ vergrößert. Auf diese Weise erhält man die Möglichkeit, einen objektiven Vergleich durchführen zu können.

Dafür wird der mittlere quadratische Fehler berechnet. Damit die genannten Randeffekte nicht so stark ins Gewicht fallen, wurde für diese Berechnung ein Randbereich mit einer Dicke von 5 Pixeln entfernt.

$$E = \frac{\sqrt{\sum_{x,y}(Obild_{x,y} - Sbild_{x,y})^2}}{N_x \cdot N_y}$$

2.2.1 Skalierungsgüte

Implementierung

```
clear
% Bild laden
fid = fopen('frau300.byt', 'r');
xfraum = fread(fid, [300,300], 'uint8');
fclose(fid);
xfrau = double(xfraum);
% Verkleinern
x = resample(xfrau,-4,1);
% Vergrößern des verkleinerten Bildes
% mit allen Methoden
% Aufruf: resample(Daten, Faktor, Methode)
y0 = resample(x, 4, 0); %Block
y1 = resample(x, 4, 1); %Linear
y2 = resample(x, 4, 2); %Faltung-Bartlett
y3 = resample(x, 4, 3); %Faltung-Gauss
y4 = resample(x, 4, 4); %Faltung Si
y5 = resample(x, 4, 5); %Spektrale erweiterung
y6 = resample(x, 4, 6); %Polynomielle Interpolation
%Darstellung der Daten
image(real(xfrau));
image(real(x));
```

```
image(real(y0));
image(real(y1));
image(real(y2));
image(real(y3));
image(real(y4));
image(real(y5));
image(real(y6));
%Berechnung der Mittelwerte als Kontrolle
mo = mean(mean(xfrau(5:end-5, 5:end-5)))
m0 = mean(mean(y0(5:end-5, 5:end-5)))
m1 = mean(mean(y1(5:end-5, 5:end-5)))
m2 = mean(mean(y2(5:end-5, 5:end-5)))
m3 = mean(mean(y3(5:end-5, 5:end-5)))
m4 = mean(mean(y4(5:end-5, 5:end-5)))
m5 = mean(mean(y5(5:end-5, 5:end-5)))
m6 = mean(mean(y6(5:end-5, 5:end-5)))
%Berechnung der Fehler Original<->Scaled
delta0 = xfrau(5:end-5,5:end-5)-y0(5:end-5,5:end-5);
delta1 = xfrau(5:end-5,5:end-5)-y1(5:end-5,5:end-5);
delta2 = xfrau(5:end-5,5:end-5)-y2(5:end-5,5:end-5);
delta3 = xfrau(5:end-5,5:end-5)-y3(5:end-5,5:end-5);
delta4 = xfrau(5:end-5,5:end-5)-y4(5:end-5,5:end-5);
delta5 = xfrau(5:end-5,5:end-5)-y5(5:end-5,5:end-5);
delta6 = xfrau(5:end-5,5:end-5)-y6(5:end-5,5:end-5);
[xl1,yl1] = size(delta3);
e0 = sqrt(sum(sum(delta0.*delta0)))/(xl1*yl1)
e1 = sqrt(sum(sum(delta1.*delta1)))/(xl1*yl1)
e2 = sqrt(sum(sum(delta2.*delta2)))/(xl1*yl1)
e3 = sqrt(sum(sum(delta3.*delta3)))/(xl1*yl1)
e4 = sqrt(sum(sum(delta4.*delta4)))/(xl1*yl1)
e5 = sqrt(sum(sum(delta5.*delta5)))/(xl1*yl1)
e6 = sqrt(sum(sum(delta6.*delta6)))/(xl1*yl1)
```

Ergebnis

Der Fehler hat die Einheit des Schwarzwertes und ist die durchschnittliche Abweichung eines Pixels vom Originalwert.

- **0.0519** - Methode 0 - *Block* (Abschnitt 2.1.1)
- **0.0516** - Methode 1 - *Linear* (Abschnitt 2.1.2)
- **0.0489** - Methode 2 - *Polynomiell* (Abschnitt 2.1.3)

- **0.0516** - Methode 3 - *Bartlett* (Abschnitt 2.1.4)
- **0.0581** - Methode 4 - *Gauss* (Abschnitt 2.1.5)
- **0.0620** - Methode 5 - *Si* (Abschnitt 2.1.6)
- **0.0488** - Methode 6 - *Spektrale* Erweiterung (Abschnitt 2.1.7)

2.2.2 Zusammenfassung der Ergebnisse

Es wurde gezeigt das es verschiedene Skalierungsmethoden mit ebenso verschiedenen Vor- und Nachteilen gibt. Je nach Anwendungsgebiet muß man sich das Verfahren heraussuchen, welches einen für diesen Fall vertretbaren Rechenaufwand besitzt.

Es gibt noch eine Reihe weiterer, gut funktionierender Methoden, die hier leider unerwähnt geblieben sind. Sie können Konturen erkennen und haben dadurch die Möglichkeit, noch besser zu Interpolieren.

Von den vorgestellten Verfahren hat daher eigentlich nur Methode Nr. 2, die in Abschnitt 2.1.3 dargestellte *Polynomielle Interpolation*, überzeugt. Die **Spektrale Erweiterung** hat zwar eine niedrige Interpolationsabweichung, dafür sind aber die Kantenechos sehr auffällig und störend.

2.3 Quellen

1. http://www.engineering.uiowa.edu/~gec/248_s00_students/blake_carlson/hw2/adiphw2.html
2. http://www.octave.org/octave-lists/archive/octave-sources.2001/bin00014.bin
   (Bei OCTAVE wurde die Funktion conv2 heruntergeladen und für die zweidimensionale Faltung benutzt.)
3. Skript NÜ1 - Noll

Abbildungsverzeichnis

1.1 Tiefpaßfilterung mit verschiedenen Grenzfrequenzen 4

2.1 Testbild 7
2.2 Vergrößerungen (300x300 Pixel) 10
2.3 Vergrößerungen (300x300 Pixel) 16
2.4 Bartlet-Fensterfunktion 17
2.5 Ergebnis der Funktion ***padding*** 18
2.6 Gauss-Fensterfunktion 20
2.7 Vergrößerungen (300x300 Pixel) 22
2.8 Si-Fensterfunktion 23
2.9 Vergrößerung (300x300 Pixel) 26