Realisierung digitaler Filter in C

Begleitmaterial zum Buch

Grundlagen der digitalen Kommunikationstechnik

Übertragungstechnik – Signalverarbeitung – Netze

Carsten Roppel

E-Mail: c.roppel@fh-sm.de

Fachbuchverlag Leipzig, 2006

Stand: 19.11.2009

Inhaltsverzeichnis

1 Finite Impulse Response (FIR)-Filter 2
1.1 Realisierung mit Gleitkomma-Zahlen 3
1.2 Circular Buffer 5
1.3 Realisierung mit Festkomma-Zahlen 7
1.4 Optimierungen 12
2 Infinite Impulse Response (IIR)-Filter 13
2.1 Realisierung mit Gleitkomma-Zahlen 14
2.2 Realisierung mit Festkomma-Zahlen 15
Literaturverzeichnis 16

1 Finite Impulse Response (FIR)-Filter

Ein FIR-Filter mit den Koeffizienten $b_0, \ldots, b_N$, also insgesamt $N+1$ Koeffizienten, zeigt Bild 1 (vgl. Bild 8-19 in [1]). Das Filter besteht aus N Verzögerungsgliedern oder Speicherelementen, die die Eingangswerte $x(n), \ldots, x(n-N)$ enthalten. Diese Werte werden mit den Filterkoeffizienten multipliziert und die Ergebnisse aufaddiert:

$$y(n) = \sum_{i=0}^{N} b_i \; x(n-i) \,. \tag{1}$$

Wenn ein neuer Eingangswert bereit steht, werden die alten Eingangswerte in den Speicherelementen nach rechts geschoben. Der älteste Wert fällt rechts aus dem Speicher heraus und der neue Wert wird links hinein geschrieben.

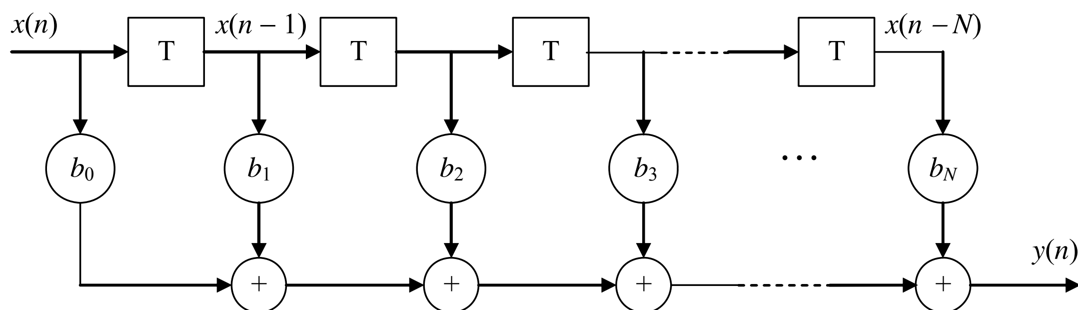

Bild 1: Ein FIR-Filter mit N + 1 Koeffizienten

Wir betrachten als Beispiel ein Tiefpassfilter der Bandbreite $B = f_A/4$ mit 11 Koeffizienten, gegeben durch

$$b_i = \begin{cases} \frac{1}{2}\,\mathrm{si}\left(\frac{\pi}{2}(i-5)\right) & \text{für} \quad 0 \le n \le 10\,, \\ 0 & \text{sonst} \end{cases} \tag{2}$$

(vgl. Kapitel 8.2.1, Seite 262 in [1]). Die numerischen Werte der Filterkoeffizienten sind in Tabelle 1 zusammengestellt und Bild 2 zeigt die Übertragungsfunktion des Filters.

Tabelle 1: Numerische Werte der Koeffizienten des FIR-Tiefpassfilters

b_0	0.0637
b_1	0
b_2	−0.1061
b_3	0
b_4	0.3183
b_5	0.5
b_6	0.3183
b_7	0
b_8	−0.1061
b_9	0
b_{10}	0.0637

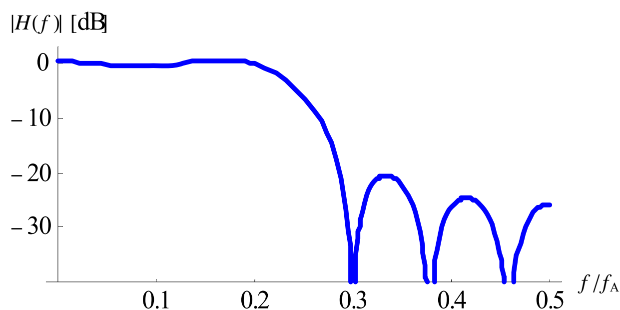

Bild 2: Übertragungsfunktion (Betrag) des FIR-Tiefpassfilters

Das Filter hat bei $B = f_A/4$ eine Dämpfung von $|H(f)| = 1/2$ bzw. $20\log|H(f)| = -6$ dB. Arbeitet das Filter beispielsweise mit einer Abtastrate von $f_A = 1$ MHz, so ist $B = 250$ kHz. Die Zeit zwischen zwei Abtastwerten beträgt dann $T_A = 1/f_A = 1$ µs. Bei einer Filterung in Echtzeit ist dies die Zeit, die für die Berechnung eines neuen Ausgangswertes gemäß Gl. (1) zur Verfügung steht.

1.1 Realisierung mit Gleitkomma-Zahlen

Bild 3 zeigt eine direkte Umsetzung von Gl. (1) in C-Code unter Verwendung von Gleitkommazahlen vom Typ float. Gezeigt ist nur der für die Filterfunktion relevante Teil eines C-Programms. Der für die Ein- und Ausgabe erforderliche Code hängt von der Testumgebung ab (z. B. PC oder DSP-Entwicklungssystem) und ist daher nicht enthalten.

Die Filterkoeffizienten werden in einem Array b[nc] abgelegt. Dabei ist nc $= N + 1$ die Anzahl der Koeffizienten. Die Speicherelemente von Bild 1 werden durch das Array input_buffer[nc] realisiert. Immer, wenn ein neuer Eingangswert bereit steht, erfolgt der oben beschriebene Schiebevorgang und anschließend die Berechnung des Ausgangswertes $y(n)$.

Der Nachteil dieses Verfahrens besteht darin, dass für den Schiebevorgang N Werte im input_buffer bewegt werden müssen. Gerade bei aufwändigen Filtern mit vielen Koeffizienten ist damit ein unnötiger zeitlicher Aufwand verbunden. Dieser Nachteil wird dadurch umgangen, dass die Speicherelemente als ein Ringspeicher aufgefasst werden, der kontinuierlich beschrieben wird. Dieser wird als "Circular Buffer" bezeichnet.

```
#define nc 11    // Anzahl der Filterkoeffizienten

int i = 0;
float new_sample, y, b[nc], input_buffer[nc];

// Filterkoeffizienten b[0] = b_0, ..., b[nc - 1] = b_N
b[0] = 0.0637;
b[1] = 0;
b[2] = -0.1061;
b[3] = 0;
b[4] = 0.3183;
b[5] = 0.5;
b[6] = 0.3183;
b[7] = 0;
b[8] = -0.1061;
b[9] = 0;
b[10] = 0.0637;

for(i = 0; i < nc; i++)
   input_buffer[i] = 0;

// Der folgende Code wird jedes Mal ausgeführt, wenn ein
// neuer Eingangswert (new_sample) zur Verfügung steht

// Schiebe Werte im input_buffer nach rechts
for (i = nc - 1; i > 0; i--)
  input_buffer[i] = input_buffer[i - 1];

// Schreibe neuen Eingangswert in Buffer
input_buffer[0] = new_sample;

// Berechne neuen Ausgangswert
y = 0;
for (i = 0; i < nc; i++)
  y += (b[i] * input_buffer[i]);
```

Bild 3: C-Code für FIR-Filter

1.2 Circular Buffer

Beim Circular Buffer werden die neuen Eingangswerte fortlaufend in den Speicher geschrieben. Ein Zeiger zeigt auf die Stelle, in die der neue Wert geschrieben wird. Bild 4 zeigt eine grafische Darstellung des Vorgangs. Wenn das letzte Element des Speichers erreicht wird, wird der Schreibvorgang bei dem ersten Speicherelement fortgesetzt. Der neue Wert $x(n)$ überschreibt den ältesten Wert $x(n - N)$ im Speicher und der Schiebevorgang entfällt.

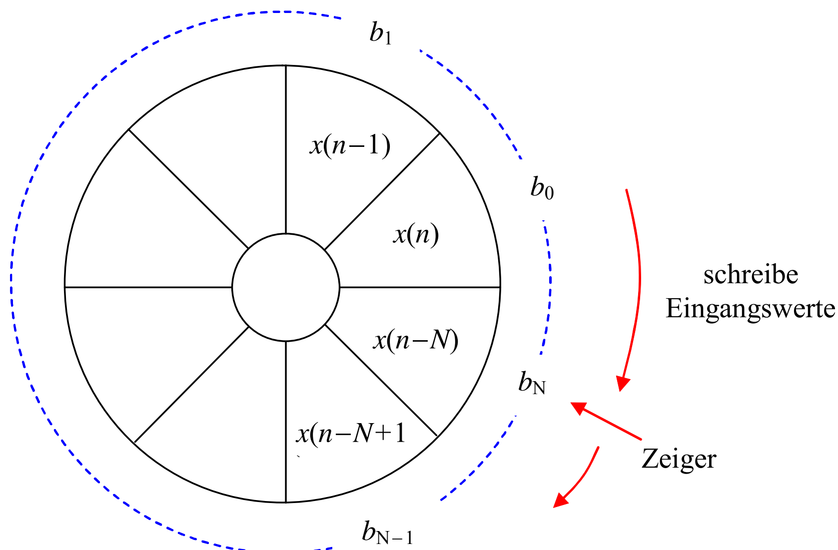

Bild 4: Schematische Darstellung mit Circular Buffer

Im zugehörigen C-Code (Bild 5) wird der Zeiger als eine Integer-Variable zeiger definiert. Diese Variable wird modulo nc inkrementiert, d. h. zeiger nimmt nacheinander die Werte 0, 1, …, (nc – 1), 0, 1, … an. Ein neuer Eingangswert wird an die Stelle des Circular Buffer geschrieben, an der der Zeiger gerade steht. Anschließend wird der Zeiger inkrementiert; er zeigt nun auf die Stelle des Circular Buffer, in dem jetzt der älteste Eingangswert $x(n - N)$ abgespeichert ist. Nun erfolgt wieder die Berechnung des neuen Ausgangswertes. Dabei ist zu beachten, dass b[0] mit dem ältesten Wert $x(n - N)$ und b[nc – 1] mit dem jüngsten Wert $x(n)$ multipliziert wird. Daher müssen im Array b die Filterkoeffizienten in der Reihenfolge b[0] = b_N, b[1] = b_{N-1}, …, b[nc – 1] = b_0 abgelegt werden. Dies ist die umgekehrte Reihenfolge wie in Bild 3, wobei sich in diesem Beispiel aufgrund der Symmetrie der Impulsantwort des Filters zahlenmäßig kein Unterschied ergibt.

```
#define nc 11    // Anzahl der Filterkoeffizienten

int i = 0, zeiger = 0;
float new_sample, y, b[nc], circular_buffer[nc];

// Filterkoeffizienten b[0] = b_N, ..., b[nc - 1] = b_0
b[0] = 0.0637;
b[1] = 0;
b[2] = -0.1061;
b[3] = 0;
b[4] = 0.3183;
b[5] = 0.5;
b[6] = 0.3183;
b[7] = 0;
b[8] = -0.1061;
b[9] = 0;
b[10] = 0.0637;

for(i = 0; i < nc; i++)
   circular_buffer[i] = 0;

// Der folgende Code wird jedes Mal ausgeführt, wenn ein
// neuer Eingangswert (new_sample) zur Verfügung steht

// Schreibe neuen Eingangswert in Buffer
circular_buffer[zeiger] = new_sample;

// Inkrementiere Zeiger modulo nc
zeiger = (zeiger + 1) % nc;

// Berechne neuen Ausgangswert
y = 0;
for(i = 0; i < nc; i++)
  y += (b[i] * circular_buffer[(zeiger + i) % nc]);
```

Bild 5: C-Code für FIR-Filter mit Circular Buffer

1.3 Realisierung mit Festkomma-Zahlen

Oft handelt es sich bei digitalen Signalprozessoren (Digital Signal Processor, DSP) um Festkomma-Prozessoren, die in der Regel mit 16-bit-Zahlen arbeiten. Wird ein digitales Filter auf einem solchen Prozessor implementiert, müssen die Filterkoeffizienten auf Festkomma-Zahlen abgebildet werden. Ferner muss bei der Berechnung des Ausgangswertes darauf geachtet werden, dass der darstellbare Wertebereich nicht überschritten wird. Andernfalls würde es zu einem Überlauf kommen.

Die binäre Darstellung von Festkomma-Zahlen erfolgt meist im Zweierkomplement. Beim Zweierkomplement hat das MSB (most significant bit) einer n-bit-Zahl die Wertigkeit -2^{n-1}:

	MSB				LSB
Bit	n-1	n-2		1	0
Wertigkeit	-2^{n-1}	2^{n-2}		2^1	2^0

Die größte (positive) darstellbare Zahl ist $2^{n-1} - 1$ und die kleinste (negative) Zahl ist -2^{n-1}. Damit ergibt sich für 16-bit-Zahlen ein Wertebereich von 32767 (7FFF hex) bis −32768 (8000 hex). Bei der Darstellung nach Vorzeichen und Betrag ergäbe sich ein Wertebereich von ±32767. Der Vorteil der Zweierkomplement-Darstellung liegt darin, dass für die Subtraktion zweier Zahlen lediglich deren Zweierkomplemente addiert werden müssen.

Die bei der Quantisierung der Filterkoeffizienten entstehenden Rundungsfehler sind dafür verantwortlich, dass sich die Übertragungsfunktion des Filters ändert. Bei FIR-Filtern betrifft dies den Amplitudengang, d. h. den Betrag der Übertragungsfunktion. Dagegen ändert sich bei einem FIR-Filter mit symmetrischer Impulsantwort (vgl. Kapitel 8.2.1, Seite 263 in [1]) der Phasengang nicht, da auch nach der Rundung die Symmetrie erhalten bleibt. Nach der Umwandlung der Koeffizienten in Festkomma-Zahlen ist zu prüfen, ob das Filter die Spezifikation noch erfüllt.

Bei der Quantisierung der Filterkoeffizienten kann nicht der volle Wertebereich der 16-bit-Zahlen ausgeschöpft werden, da es sonst bei der Multiplikation mit einem großen Eingangswert sofort zu einem Überlauf kommen würde. Wir betrachten am Beispiel unseres FIR-Filters die Rundung auf 6-bit-Integerwerte. Damit steht ein Wertebereich von $2^5 - 1 = 31$ bis $-2^5 = -32$ für die Koeffizienten zur Verfügung. Der betragsmäßig größte Koeffizient wird auf ±31 abgebildet. Dies geschieht mit der Umrechnung

$$\tilde{b}_n = \text{Round}\left[31 \cdot \frac{b_n}{|b_{\max}|}\right] \tag{3}$$

der Koeffizienten b_n in die entsprechenden gerundeten Werte $\tilde{b}_n$. Die Funktion Round[] beschreibt die Rundung auf den nächsten ganzzahligen Wert. In unserem Beispiel ist $b_{\max} = 0{,}5$; dieser Wert wird auf 31 abgebildet. Entsprechend wird beispielsweise $b_2 = b_8 = -0{,}1061$ auf −7 abgebildet.

Den Einfluss der Quantisierung auf die Übertragungsfunktion zeigt Bild 6. Die gestrichelte Linie gibt den Verlauf mit idealen Koeffizienten wieder. Im Beispiel verringert sich die minimale Dämpfung im Sperrbereich um ca. 1,5 dB.

Damit es bei der Berechnung der Ausgangswerte nicht zum Überlauf kommt, muss das Eingangssignal skaliert werden. Mit dem Skalierungsfaktor

$$g = b_{\max} / \tilde{b}_{\max} \tag{4}$$

erhält man ein Filter mit $|H_{\max}(f)| \approx 1$, d. h. die Verstärkung im Durchlassbereich ist etwa 1. Dazu müssen die Eingangswerte, die im Bereich 32767 bis −32768 liegen, mit g multipliziert (und auf Integerwerte gerundet) werden.

Im Beispiel ist $b_{\max} = 0{,}5$ und $\tilde{b}_{\max} = 31$, der Skalierungsfaktor beträgt also $g = 1/62$. Multiplikation mit g bedeutet Division durch 62. Um eine zeitaufwändige Division zu vermeiden, sollte g eine Potenz von 2 sein. In diesem Fall kann die Division einschließlich der Rundung durch bitweises Verschieben nach rechts ersetzt werden. Im Beispiel wählen wir $g = 2^{-6} = 1/64$. Der Multiplikation mit g entspricht dann die Rechtsverschiebung um 6 bit. Den zugehörigen C-Code enthält Bild 7.

Es sei noch darauf hingewiesen, dass die obige Skalierung nicht die hundertprozentige Vermeidung eines Überlaufs garantiert. Für das maximale Ausgangssignal gilt

$$y_{\max} = x_{\max}\, g \sum_{k=0}^{N} |b_k|\,. \tag{5}$$

Man erhält den maximalen Ausgangswert $y_{\max}$, wenn die Eingangswerte $x(n)$ ihre maximalen Werte $\pm x_{\max}$ annehmen und ihr Vorzeichen so gewählt wird, dass jeweils die Beträge der Koeffizienten b_k addiert werden. Damit $y_{\max}$ innerhalb des darstellbaren Zahlenbereichs liegt, muss $y_{\max} \le x_{\max}$ sein und damit

$$g = \frac{1}{\sum_{k=0}^{N} |b_k|} \tag{6}$$

gelten. Im Beispiel wäre ein Skalierungsfaktor von $g = 93$ erforderlich, um eine Vermeidung eines Überlaufs auch für den oben geschilderten (aber unwahrscheinlichen) Fall zu garantieren.

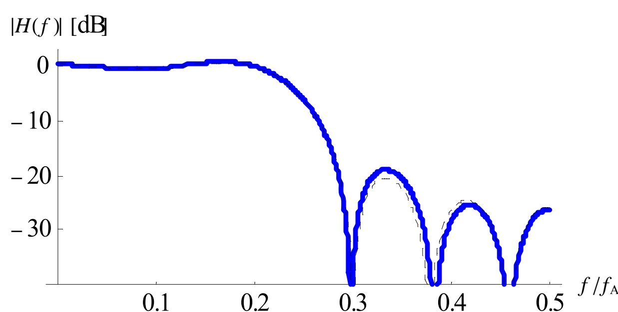

Bild 6: Übertragungsfunktion des FIR-Filters mit gerundeten Koeffizienten

```
#define nc 11    // Anzahl der Filterkoeffizienten

int i = 0, zeiger = 0;
int new_sample, y, b[nc], circular_buffer[nc];

// Filterkoeffizienten b[0] = b_N, ..., b[nc - 1] = b_0
b[0] = 4;
b[1] = 0;
b[2] = -7;
b[3] = 0;
b[4] = 20;
b[5] = 31;
b[6] = 20;
b[7] = 0;
b[8] = -7;
b[9] = 0;
b[10] = 4;

for(i = 0; i < nc; i++)
   circular_buffer[i] = 0;

// Der folgende Code wird jedes Mal ausgeführt, wenn ein
// neuer Eingangswert (new_sample) zur Verfügung steht

// Schreibe neuen Eingangswert in Buffer
// Eingangswerte werden um 6 bit nach rechts verschoben
// (entspricht Multiplikation mit 1/64)
circular_buffer[zeiger] = new_sample >> 6;

// Inkrementiere Zeiger modulo nc
zeiger = (zeiger + 1) % nc;

// Berechne neuen Ausgangswert
y = 0;
for(i = 0; i < nc; i++)
  y += (b[i] * circular_buffer[(zeiger + i) % nc]);
```

Bild 7: C-Code für FIR-Filter mit Circular Buffer und Integer-Zahlen

Bei der Quantisierung der Filterkoeffizienten gilt es, einen Kompromiss zwischen den Filtereigenschaften und der Quantisierung der Eingangswerte zu finden. Im obigen Beispiel sind ursprünglich mit 16 bit quantisierte Eingangswerte nach der Skalierung nur noch mit einer Auflösung von etwa 10 bit quantisiert. Dadurch verringert sich der auf das Quantisierungsrauschen bezogene Störabstand für das Eingangssignal um 6·6,02 dB = 36,12 dB ([1], Kapitel 3.3). Stellt man die Koeffizienten beispielsweise mit 8-bit-Integerwerten dar, so wird zwar die Übertragungscharakteristik des Filters besser mit der des idealen Filters übereinstimmen. Andererseits müssen dann aber die Eingangswerte mit $g = 1/254$ skaliert werden. Die Eingangswerte würden nach der Skalierung nur noch mit einer Auflösung von etwa 8 bit quantisiert sein.

Günstiger sind in dieser Hinsicht gebrochene Zweierkomplement-Zahlen. Man bezeichnet diese Zahlendarstellung auch als Q.*m*-Format, wobei *m* die Anzahl der Nachkommastellen bezeichnet (Q: quantity of fractional bits). Gebräuchlich ist beispielsweise bei 16-bit-Zahlen das Q.15-Format. Dann hat das MSB die Wertigkeit $-2^0 = -1$, während die restlichen 15 Bits den gebrochenen Teil der Zahl angeben:

	MSB				LSB
Bit	$n-1$	$n-2$	…	1	0
Wertigkeit	-2^0	2^{-1}	…	$2^{-(n-2)}$	$2^{-(n-1)}$

Damit ergibt sich für 16-bit-Zahlen im Q.15-Format ein Wertebereich von $1 - 2^{-15}$ (7FFF hex) bis -1 (8000 hex). Das Quantisierungsintervall zwischen zwei aufeinander folgenden Q.15-Zahlen beträgt $2^{-15} = 3{,}05 \cdot 10^{-5}$. Man beachte, dass die Q.15-Zahlen und die zuvor beschriebenen Integerwerte im Zweierkomplement die gleiche Binärdarstellung haben.

Ein Unterschied ergibt sich jedoch bei der Multiplikation. Wenn man zwei Zahlen mit einem Betrag kleiner oder gleich 1 multipliziert, kommt es prinzipiell nicht zum Überlauf, da das Ergebnis immer im Wertebereich von ±1 liegt. Allerdings entsteht bei der Multiplikation zweier Q.*m*-Zahlen ein Zwischenergebnis im Q.2*m*-Format:

$$\frac{x}{2^m} \cdot \frac{y}{2^m} = \frac{x \cdot y}{2^{2m}}. \tag{7}$$

Dabei entspricht der Integerzahl x die Q.*m*-Zahl $x/2^m$. Bei der Multiplikation von zwei Q.15-Zahlen entsteht also eine Q.30-Zahl der Länge 32 bit. Diese muss für die weitere Verarbeitung wieder in eine Q.15-Zahl zurückgewandelt werden. Dies geschieht, indem das 32-bit-Zwischenergbnis um 15 bit nach rechts verschoben wird. Die entsprechenden Nachkommastellen gehen dabei verloren, d. h. das Ergebnis wird wieder auf eine Q.15-Zahl gerundet.

Bild 8 zeigt den entsprechenden C-Code. Die Filterkoeffizienten erhält man durch Multiplikation der Werte aus Tabelle 1 mit 2^{15} (und anschließendes Runden). Die Multiplikation der Q.15-Zahlen geschieht in der Funktion mult_q15().

```
#define nc 11     // Anzahl der Filterkoeffizienten

int i = 0, k = 0, zeiger = 0;
short new_sample, y, b[nc], circular_buffer[nc];   // 16 bit

// Filterkoeffizienten b[0] = b_N, ..., b[nc - 1] = b_0
b[0] = 2087;
b[1] = 0;
b[2] = -3477;
b[3] = 0;
b[4] = 10430;
b[5] = 16384;
b[6] = 10430;
b[7] = 0;
b[8] = -3477;
b[9] = 0;
b[10] = 2087;

for(i = 0; i < nc; i++)
   circular_buffer[i] = 0;

// Der folgende Code wird jedes Mal ausgeführt, wenn ein
// neuer Eingangswert (new_sample) zur Verfügung steht

// Schreibe neuen Eingangswert in Buffer
circular_buffer[zeiger] = new_sample;

// Inkrementiere Zeiger modulo nc
zeiger = (zeiger + 1) % nc;

// Berechne neuen Ausgangswert
y = 0;
for(i = 0; i < nc; i++)
{
  k = (zeiger + i) % nc;
  y += mult_q15(b[i], circular_buffer[k]);
}


// Multiplikation zweier Q15-Zahlen
short mult_q15(short factor1, short factor2)
{
   return (short)(((long)factor1 * (long)factor2) >> 15);
}
```

Bild 8: C-Code für FIR-Filter mit Circular Buffer und Q.15-Zahlen

1.4 Optimierungen

Unter Ausnutzung bestimmter Eigenschaften eines FIR-Filters sind eine Reihe von Optimierungen möglich, mit denen sich die für die Berechnung erforderliche Zeit weiter reduzieren lässt.

Zunächst fällt auf, dass in unserem Beispiel-Filter jeder zweite Koeffizient null ist. In der Programmschleife zur Berechnung des Ausgangswertes können daher die Multiplikationen mit den Null-Koeffizienten weggelassen werden. Es handelt sich um ein so genanntes Halbbandfilter. Der nutzbare Frequenzbereich eines digitalen Filters, das mit der Abtastrate f_A arbeitet, liegt im Frequenzbereich $0 \leq f \leq f_A/2$. Bei einem Halbbandfilter liegt die Grenzfrequenz genau in der Mitte dieses Bereichs, also bei $f_A/4$.

Ferner ist die Impulsantwort des Filters symmetrisch. Eine Symmetrie ist bei linearphasigen Filtern immer gegeben ([1], Kapitel 8.2.1). Diese Symmetrie kann ebenfalls zur Reduzierung der erforderlichen Multiplikationen genutzt werden. Dazu werden die Eingangswerte im Buffer, die mit den jeweils gleichen Koeffizienten multipliziert werden, zuvor addiert:

$$b_i \, x(n-i) + b_{N-i} \, x(n-N-i) = b_i \left(x(n-i) + x(n-N-i) \right)$$

für $b_i = b_{N-i}$.

Oft arbeitet ein System bei verschiedenen Abtastraten. Das Herabsetzten der Abtastrate bezeichnet man als Dezimieren. Beim Dezimieren um den Faktor 2 wird einfach jeder zweite Abtastwert weggelassen, so dass sich die Abtastrate halbiert. Damit es dabei nicht zu Aliasing kommt, muss das Signal zuvor meist durch ein Tiefpassfilter bandbegrenzt werden. Beide Schritte, Tiefpassfilterung und Dezimierung, können bei einem FIR-Filter zusammengefasst werden. Die Abtastwerte, die beim Dezimieren weggelassen werden, werden dabei gar nicht berechnet. Man bezeichnet dies auch als dezimierendes Filter.

Bei der C-Programmierung sind einfache Maßnahmen das Loop unrolling und das Inlining. Beim Loop unrolling schreibt man anstelle einer for-Schleife, die n-mal durchlaufen wird, n einzelne Anweisungen. Dies ist möglich, wenn n konstant ist und erspart die Abfrage der Bedingung in der for-Anweisung. Beim Inlining ersetzt man Funktionsaufrufe durch die Anweisungen innerhalb der Funktion. Dies erspart die Taktzyklen, die mit dem Overhead bei einem Funktionsaufruf verbunden sind.

Daneben gibt es zahlreiche weitere prozessor- oder compilerspezifische Möglichkeiten der Optimierung [2].

2 Infinite Impulse Response (IIR)-Filter

Ein IIR-Filter wird meist mit Hilfe von Teilsystemen 2. Ordnung in der Direktform II realisiert. Bild 9 zeigt das entsprechende Blockschaltbild (vgl. Bild 8-30 in [1]). Ein Teilsystem besteht aus 2 Verzögerungsgliedern oder Speicherelementen, die die Zwischenwerte $w_j(n)$ enthalten, sowie den zwei Koeffizienten a_{1j}, a_{2j} im rekursiven Teil und den drei Koeffizienten b_{0j}, b_{1j} und b_{2j}. Der zweite Index (j) dient der Unterscheidung bei mehreren Teilsystemen. Ein Teilsystem wird durch die Gleichungen

$$\begin{aligned} w(n) &= x(n) + a_{1j} w_1(n-1) + a_{2j} w_1(n-2) \\ y(n) &= b_{0j} w_1(n) + b_{1j} w_1(n-1) + b_{2j} w_1(n-2) \end{aligned} \tag{8}$$

beschrieben.

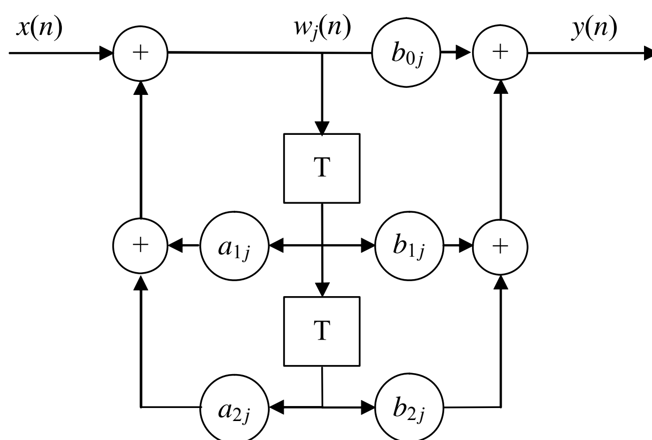

Bild 9: IIR-Filter als Teilsystem 2. Ordnung in der Direktform II

Wir betrachten als Beispiel einen Butterworth-Tiefpass 1. Ordnung mit der −3-dB-Grenzfrequenz von 0,125 f_A (vgl. Kapitel 8.2.2, Seite 269 in [1]). Die numerischen Werte der Filterkoeffizienten sind in Tabelle 2 zusammengestellt[1] und Bild 10 zeigt die Übertragungsfunktion des Filters.

Tabelle 2: Numerische Werte der Filterkoeffizienten des IIR-Filters

a_1	0.4142
a_2	0
b_0	0.2929
b_1	0.2929
b_2	0

[1] Das Filterdesigntool von Matlab generiert die Koeffizienten a_{ij} mit gegenüber unserer Festlegung invertierten Vorzeichen

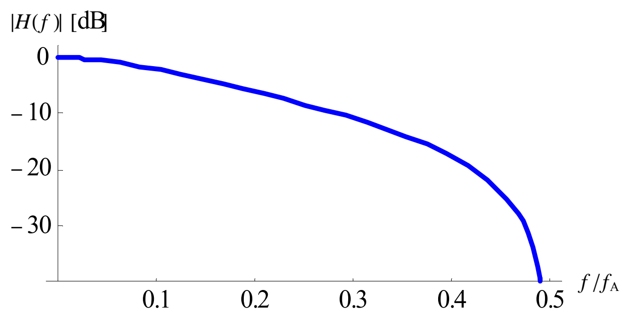

Bild 10: Übertragungsfunktion des IIR-Tiefpassfilters

2.1 Realisierung mit Gleitkomma-Zahlen

Bild 11 zeigt eine direkte Umsetzung von Gl. (8) in C-Code unter Verwendung von Gleitkommazahlen vom Typ float. Gezeigt ist wie zuvor nur der für die Filterfunktion relevante Teil eines C-Programms ohne Ein- und Ausgabe.

Die Filterkoeffizienten werden in den Arrays a[2] und b[3] mit jeweils zwei bzw. drei Elementen abgelegt. Die Speicherelemente von Bild 9 werden durch das Array w[3] realisiert. Nach der Berechnung des Ausgangswertes y werden die Werte in den Speicherelementen weiter geschoben. Im vorliegenden Fall könnte die Berechnung weiter vereinfacht werden, da die Koeffizienten a_2 und b_2 null sind.

```
float a[2], b[3], w[3] = {0, 0, 0};

// Filterkoeffizienten
a[0] = 0.4142;
a[1] = 0;
b[0] = 0.2929;
b[1] = 0.2929;
b[2] = 0;

// Der folgende Code wird jedes Mal ausgeführt, wenn ein
// neuer Eingangswert (new_sample) zur Verfügung steht

w[0] = new_sample + (a[0] * w[1]) + (a[1] * w[2]);
y = (b[0] * w[0]) + (b[1] * w[1]) + (b[2] * w[2]);
w[2] = w[1];
w[1] = w[0];
```

Bild 11: C-Code für IIR-Filter

2.2 Realisierung mit Festkomma-Zahlen

Bild 12 zeigt den Code bei Verwendung von Festkomma-Zahlen im Q.15-Format. Die Filterkoeffizienten erhält man durch Multiplikation der Werte aus Tabelle 2 mit 2^{15} (und anschließendes Runden). Für die Multiplikation der Q.15-Zahlen wird die gleiche Funktion wie in Bild 8 verwendet.

Bei der Realisierung von IIR-Filtern mit Festkomma-Zahlen ist zu beachten, dass diese Filter durch die Rundungsfehler instabil werden können. Ein Filter ist stabil, wenn die Pole der z-Übertragungsfunktion innerhalb des Einheitskreises liegen [1]. Daher sind Filter mit Polen dicht am Einheitskreis kritisch, wenn die Pole nach der Rundung nicht mehr innerhalb des Kreises liegen.

```
short a[2], b[3], w[3] = {0, 0, 0};    // 16 bit Integer

// Filterkoeffizienten
a[0] = 13573;
a[1] = 0;
b[0] = 9598;
b[1] = 9598;
b[2] = 0;

// Der folgende Code wird jedes Mal ausgeführt, wenn ein
// neuer Eingangswert (new_sample) zur Verfügung steht

w[0] = new_sample + mult_q15(a[0], w[1]) +
       mult_q15(a[1], w[2]);
y = mult_q15(b[0], w[0]) + mult_q15(b[1], w[1]) +
    mult_q15(b[2], w[2]);
w[2] = w[1];
w[1] = w[0];


// Multiplikation zweier Q15-Zahlen
short mult_q15(short factor1, short factor2)
{
  return (short)(((long)factor1 * (long)factor2) >> 15);
}
```

Bild 12: C-Code für IIR-Filter mit Q.15-Zahlen

Literaturverzeichnis

[1] Roppel, C.: Grundlagen der digitalen Kommunikationstechnik. Hanser Verlag, 2006.

[2] Oshana, R.: DSP Software Development Techniques for Embedded and Real-Time Systems. Elsevier/Newnes, 2006.