

Mathematische Optimierung im Entwurf cosinus-modulierter Filterbänke

Diplomarbeit am Mathematischen Seminar
der Christian-Albrechts-Universität zu Kiel
unter der Leitung von Herrn Prof. Dr. A. Srivastav

In Kooperation mit Herrn Prof. Dr.-Ing. U. Heute
und Herrn Dr.-Ing. J. Kliewer,
Lehrstuhl für Netzwerk- und Systemtheorie

Vorgelegt von
Volkmar Sauerland
aus Wattenbek

Kiel, Februar 2003

Inhaltsverzeichnis

1	Einleitung	3
2	Theoretische Grundlagen	6
2.1	Analoge und digitale Signale	6
2.2	Transformationen zeitdiskreter Signale	7
2.2.1	Faltung von Signalen	7
2.2.2	Die z -Transformation	7
2.2.3	Die Fouriertransformation	8
2.3	Digitale Filter	9
3	M-Kanal-Filterbänke	12
3.1	Taktreduktion und Takterhöhung	12
3.2	Perfekte Rekonstruktion	15
3.3	Polyphasenzerlegung von Filtern	17
3.4	PR-Beschreibung durch Polyphasenmatrizen	19
4	Cosinus-modulierte Filterbänke	23
4.1	PR-Bedingungen an den Prototypfilter	25
4.2	Der Prototypentwurf als QCQP	27
4.3	Entwurf reeller Prototypen	29
5	Diskretisierung von Prototypfiltern	33
5.1	Canonical Signed Digits	34
5.2	Direkte Diskretisierung der Filterkoeffizienten	34
5.3	Die Liftingfaktorisierung	35
5.4	Diskretisierung der Liftingkoeffizienten	39
5.4.1	Algorithmus 1	40
5.4.2	Algorithmus 2	40
5.5	Entwurfsbeispiele	42

<i>INHALTSVERZEICHNIS</i>	2
6 Zusammenfassung	48
A Die Module <i>linalg</i>, <i>filter</i> und <i>lifting</i>	50
A.1 <i>linalg.h</i>	50
A.2 <i>linalg.c</i>	51
A.3 <i>filter.h</i>	55
A.4 <i>filter.c</i>	55
A.5 <i>lifting.h</i>	60
A.6 <i>lifting.c</i>	61
B Das Programm <i>cosbank</i>	67
C Das Programm <i>quant</i>	72
Danksagung	81

Kapitel 1

Einleitung

Der Entwurf digitaler Filter- und Filterbänke ist eine wichtige Aufgabe in der digitalen Signalverarbeitung, die in der Informationstechnologie von großer Bedeutung ist.

Ein Signal ist immer als eine Überlagerung von Schwingungen verschiedener Frequenzen gegeben. Die Aufgabe eines Filters ist es, bestimmte Frequenzbereiche von besonderem Interesse vom Rest eines Signals zu separieren.

Eine Filterbank besteht aus mehreren parallelen Filtern, d.h. sie vollzieht eine Zerlegung eines Signals in verschiedene Frequenzbereiche, die *sogenannten Teilbänder*. Eine solche Zerlegung ist in einigen Anwendungen wichtig, wie zum Beispiel in der Codierung von Audiosignalen (Stichwort *MP3*). Unter Ausnutzung der Wahrnehmungseigenschaften des menschlichen Gehörs kann hier eine besonders starke Reduktion der Datenmenge stattfinden, wenn man die verschiedenen Teilbänder einzeln behandelt.

Bei modulierten Filterbänken gehen alle Teilbandfilter durch Multiplikation mit gewissen *Modulationsfolgen* aus einem einzigen sogenannten *Prototypfilter* hervor, was eine erhebliche Verminderung des Entwurfsaufwands bedeutet.

Zur Zerlegung reeller digitaler Signale (im Allgemeinen sind digitale Signale komplexe Zahlenfolgen) sind cosinus-modulierte Filterbänke geeignet. So sind zum Beispiel cosinus-modulierte Filterbänke Bestandteil von *MP3-Encodern*.

Im zweiten Kapitel der Arbeit werden kurz die wesentlichen und für das Verständnis der weiteren Kapitel notwendigen Grundlagen der digitalen Signalverarbeitung bis hin zur Definition digitaler Filter behandelt.

Kapitel 3 beschäftigt sich mit Filterbänken im Allgemeinen. Wir stellen die Begriffe der Taktoperation und der Polyphasenzerlegung von Teilbandfiltern vor, die zusammen eine effiziente Implementierung von Filterbänken erlauben. Die wichtige Eigenschaft der perfekten Rekonstruktion

wird eingeführt und in Beziehung zu den Polyphasenkomponenten der Teilbandfilter gestellt.

Im vierten Kapitel kommen wir dann zu den cosinus-modulierten Filterbänken, bei denen alle Teilbandfilter aus einem einzigen Prototypfilter hervorgehen. Der Entwurf cosinus-modulierter Filterbänke mit perfekter Rekonstruktion wird äquivalent als quadratisches Programm mit nicht-konvexen quadratischen Nebenbedingungen formuliert. Ein hierauf basierendes C-Programm, das lokale Optima des Entwurfproblems liefert, befindet sich im Anhang. Globale Optima des gegebenen Problemtyps können im Allgemeinen nicht effizient gefunden werden.

Das quadratische Programm besitzt eine semidefinite Relaxierung, deren Lösung in bestimmten Fällen von der Form \mathbf{xx}^T ist. Dabei handelt es sich um die beiden Fälle, für die in der Signalverarbeitung die Entwurfsverfahren der MLT- und ELT-Filter populär sind. Die Resultate dieser Verfahren gleichen denen der semidefiniten Relaxierung. Dies zeigt implizit die Optimalität von MLT- und ELT-Prototypfiltern.

Im fünften Kapitel sollen anstelle reellwertiger Prototypfilter, solche mit diskreten Koeffizienten gewonnen werden. Die Koeffizienten sollen hier eine duale Darstellung mit möglichst wenigen Bits (kurze Wortlänge) und Nicht-Null-Bits haben, um die Rechenzeit der Filterbänke klein zu halten. Dies geschieht durch nachträgliches diskretisieren reellwertiger Lösungen. Zunächst runden wir die Filterkoeffizienten direkt. Dies zieht kleine Verletzungen der Nebenbedingungen nach sich, d.h. man muss eine gewisse Aufweichung der perfekten Rekonstruktion in Kauf nehmen.

In den Abschnitten 5.3 und 5.4 wird dieses Problem umgangen, indem eine Faktorisierung von Prototypen in sogenannte Liftingkoeffizienten vorgenommen wird. Hier entfallen die Nebenbedingungen, sodass die Resultate stets perfekte Rekonstruktion ermöglichen. Einem in [KM1] vorgeschlagenem Algorithmus stellen wir ein neues iteratives Verfahren gegenüber, das jeweils die noch nicht diskretisierten Liftingkoeffizienten reoptimiert und dadurch die Zahl der Nicht-Null-Bits um durchschnittlich 15% reduziert. Für das Verfahren befindet sich ein C-Programm im Anhang.

Zur Illustration kommen in der Arbeit einige technische Schaltbilder vor. Die Elementaren Schaltelemente sind Verzögerungsglieder, Filter, und Taktoperatoren, deren Wirkung im Text erklärt wird. Die zugehörigen Symbole sind in Abbildung 1.1 dargestellt.

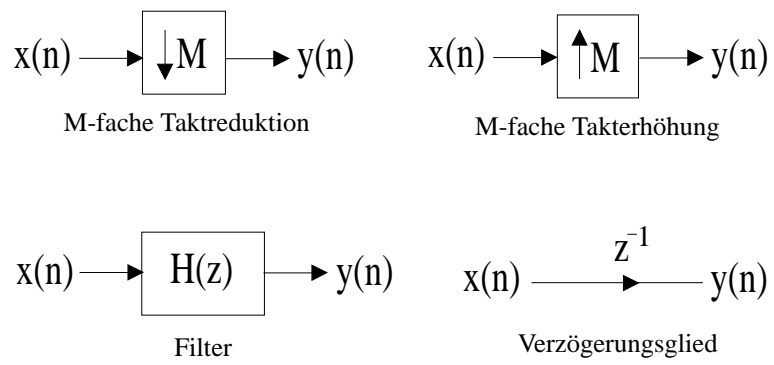


Abbildung 1.1: Elementare Schaltsymbole

Kapitel 2

Theoretische Grundlagen

2.1 Analoge und digitale Signale

Ein *zeitkontinuierliches Signal* ist eine Funktion $x : \mathbb{R} \rightarrow \mathbb{C}$. In dieser Arbeit wird ein solches Signal auch als *analoges Signal* bezeichnet. Ein *zeitdiskretes Signal* ist eine Folge $x : \mathbb{Z} \rightarrow \mathbb{C}$. Ein zeitdiskretes Signal soll *digitales Signal* heißen, falls sein Bildbereich eine endliche Teilmenge von \mathbb{C} ist, z.B. (hauptsächlich) eine Menge reeller Zahlen, die eine duale Darstellung gewisser Länge besitzen.

Ein analoges Signal x heißt endlich, falls $t_1, t_2 \in \mathbb{R}$ mit $t_1 < t_2$ existieren, derart dass $x(t) = 0$ für alle $t \in \mathbb{R} \setminus [t_1, t_2]$ gilt. Entsprechend ist ein endliches zeitdiskretes Signal eine Folge, bei der nur endlich viele Glieder von Null verschieden sind. Seien $n_{min}, n_{max} \in \mathbb{Z}$ der kleinste und der größte Index, der von Null verschiedenen Glieder eines zeitdiskreten Signals x . Dann ist $L := n_{max} - n_{min} + 1$ die Länge von x .

Natürliche Vorgänge (Elektrische Signale, Musik, Erdbeben, Temperaturschwankungen, u.s.w) sind analoge Signale, bzw. lassen sich als analoge Signale beschreiben. Mit analoger Technik lassen sich solche Vorgänge direkt verarbeiten, d.h. sie werden in physikalisch andere analoge Größen umgewandelt, die der Mensch direkt verwerten kann.

Soll die Verarbeitung (Speicherung, Analyse, Komprimierung, ...) physikalischer Abläufe aber mit Unterstützung digitaler Computer geschehen, braucht man offenbar endliche digitale Signale.

Ein analoges Signal x_a kann dazu durch äquidistante Abtastung (englisch: sampling) in ein zeitdiskretes Signal x_d verwandelt werden: $x_d(n) := x_a(nT)$, $n \in \mathbb{Z}$, $T > 0$. Durch Runden der Werte entsteht aus dem zeitdiskreten Signal dann ein digitales Signal.

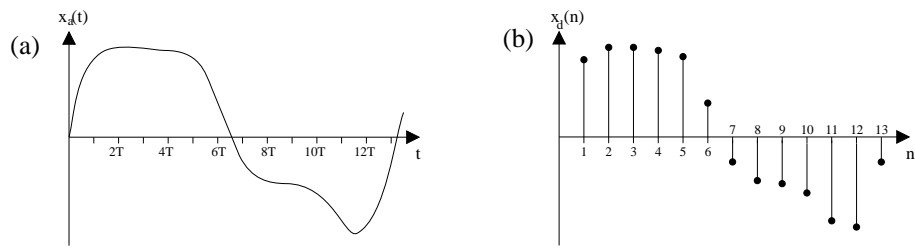


Abbildung 2.1: Ein analoges Signal (a) und seine diskrete Abtastung (b)

2.2 Transformationen zeitdiskreter Signale

2.2.1 Faltung von Signalen

Analog zur kontinuierlichen Faltung $x * y$ zweier Funktionen $x, y : \mathbb{R} \mapsto \mathbb{C}$, gegeben durch (falls existent) $(x * y)(t) = \int_{s \in \mathbb{R}} x(s)y(t-s)dt$, wird die Faltung zweier Folgen $x, y \in \mathbb{C}^{\mathbb{Z}}$ definiert durch

$$(x * y)(n) := \sum_{k \in \mathbb{Z}} x(k)y(n - k), \quad n \in \mathbb{Z},$$

falls die Summe konvergiert. Insbesondere ist die Faltung definiert, wenn eine der Folgen endlich ist.

2.2.2 Die z -Transformation

Die Umwandlung einer Folge $x \in \mathbb{C}^{\mathbb{Z}}$ in die Laurentreihe

$$z\{x\} = X(z) := \sum_{n \in \mathbb{Z}} x(n)z^{-n}$$

wird als die z -Transformation von x bezeichnet, falls es ein $z \in \mathbb{C}$ gibt, für das die Reihe konvergiert. Bekanntlich ist der Konvergenzbereich einer Laurentreihe im wesentlichen ein offener Kreisring um den Ursprung (eventuell leer). D.h. es gibt reelle Zahlen $0 \leq r \leq R$, derart dass $X(z)$ auf

$$K_{r,R}(0) := \{z \in \mathbb{C} \mid r < |z| < R\}$$

konvergiert und auf $\mathbb{C} \setminus \overline{K_{r,R}(0)}$ divergiert. Ist die Folge x endlich, so ist $X(z)$ eine endliche Summe, dann auch *Laurentpolynom* genannt, die auf ganz $\mathbb{C} \setminus \{0\}$ (ggf. ganz \mathbb{C}) konvergiert.

Die z -Transformation hat unter anderem die folgenden elementaren Eigenschaften. Es seien $x, y \in \mathbb{C}^{\mathbb{Z}}$ und $R_x, R_y \subseteq \mathbb{C}$ offene Kreisringe, auf denen $X(z)$ und $Y(z)$ konvergieren. Ferner sei $D \in \mathbb{Z}$ und $x_D \in \mathbb{C}^{\mathbb{Z}}$ eine um D Glieder verschobene Version von x , d.h. $x_D(n) = x(n - D)$, $n \in \mathbb{N}$. Dann gelten

- *Linearität*: $z\{ax + by\} = aX(z) + bY(z)$, wobei der Konvergenzbereich $R_x \cap R_y$ enthält,
- *Zeitverschiebung*: $X_D(z) = z^{-D}X(z)$, mit Konvergenzbereich R_x ,
- *Faltungseigenschaft*: $z\{x * y\} = X(z)Y(z)$, wobei der Konvergenzbereich $R_x \cap R_y$ enthält.

2.2.3 Die Fouriertransformation

Ist der Einheitskreis im Konvergenzbereich der z -Transformierten $X(z)$ eines zeitdiskreten Signals enthalten, so ist durch

$$\omega \mapsto X(\omega) := X(e^{i\omega}) = \sum_{n \in \mathbb{Z}} x(n)e^{-in\omega}$$

eine Abbildung von \mathbb{R} nach \mathbb{C} definiert, die *zeitdiskrete Fouriertransformation*. Sie ist 2π -periodisch ($\forall k, n \in \mathbb{Z} : e^{-in\omega} = e^{-in(\omega+2\pi k)}$), daher ist nur ihre Einschränkung auf das Intervall $[-\pi, \pi]$ interessant (im Unterschied zur zeitkontinuierlichen Fouriertransformation $X(\omega) = \int_{\mathbb{R}} x(t)e^{-i\omega t} dt$). Linearität und Faltungseigenschaft übertragen sich offenbar von der z -Transformation auf die Fouriertransformation.

Die Fouriertransformation gibt Auskunft darüber, wie ein Signal aus einfachen komplexen Schwingungen, also aus Folgen der Form $e^{in\omega}$, $n \in \mathbb{Z}$, zusammengesetzt ist. Bei zeitdiskreten Signalen werden Frequenzen von Schwingungen in Winkel pro Folgenglied gemessen (statt Winkel/Zeit im kontinuierlichen Fall). Man spricht dann von der *diskreten Frequenz*.

Beispiel: Das endliche Signal $x \in \mathbb{C}^{\mathbb{Z}}$, gegeben durch

$$x(n) := \begin{cases} e^{in\pi/4} & \text{falls } n = 0, \dots, 7 \\ 0 & \text{sonst} \end{cases},$$

ist eine einfache komplexe Schwingung der Frequenz $\frac{\pi}{4}$ über eine Periode. Die Fouriertransformierte in den Frequenzen $\omega_k := \frac{k}{4}\pi$, $k = -4, -3, \dots, 3, 4$, ist

$$X(\omega_k) = \sum_{n=0}^7 x(n)e^{-in\omega_k} = \sum_{n=0}^7 e^{in(\frac{\pi}{4} - \frac{k}{4}\pi)} = \begin{cases} 8 & \text{falls } k = 1 \\ 0 & \text{sonst} \end{cases}.$$

Abbildung 2.2 zeigt den Betrag von $X(\omega)$ auf ganz $[-\pi, \pi]$. Kompliziertere zeitdiskrete Signale $x \in \mathbb{C}^{\mathbb{Z}}$ können als Summen von Folgen, die einfache komplexe Schwingungen darstellen, angesehen werden. Periodische zeitdiskrete Signale sind sogar exakt durch solche Summen darstellbar (zeitdiskrete Fourierreihe). Die Transformation $X(\omega)$, $\omega \in [-\pi, \pi]$, eines solchen Signals

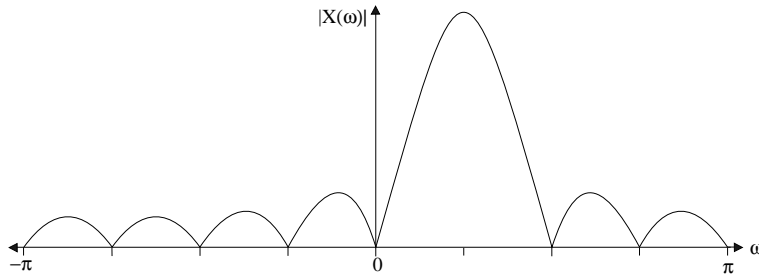


Abbildung 2.2: zeitdiskrete Fouriertransformation einer einfachen Schwingung

ist dann die Summe der Transformaten der einfachen Schwingungen (Linearität). Wegen der Information, die die Fouriertransformation eines Signals über dessen Zusammensetzung aus komplexen Schwingungen verschiedener Frequenzen liefert, wird diese auch als dessen *Spektrum* bezeichnet. Näheres zum Thema Fourieranalyse (Fourierreihen, zeitkontinuierliche- und zeitdiskrete Fouriertransformation) findet man z.B. in [Op].

2.3 Digitale Filter

Ein zu analysierendes Signal wird in der Praxis physikalische Vorgänge beschreiben, die sich in Abhängigkeit der Anwendung aus mehr oder weniger interessanten Größen zusammensetzen. Die wesentlichen Größen des Signals sollen dann vom Rest separiert werden. Das bedeutet, dass nur bestimmte Frequenzen aus dem Spektrum des Signals erhalten werden. Die interessanten Frequenzen sollen zu diesem Zweck mit Zahlen vom Betrag 1 und der Rest mit 0 multipliziert werden.

Definition 2.1 Sei $h \in \mathbb{C}^{\mathbb{Z}}$ ein (digitales) Signal. Die Faltung eines beliebigen (digitalen) Signals $x \in \mathbb{C}^{\mathbb{Z}}$ mit h heißt Filterung im Zeitbereich. Entsprechend wird die Multiplikation der Spektren $X(\omega)$ und $H(\omega)$ als Filterung im Frequenzbereich bezeichnet. Sowohl h als auch $H(z)$ werden (digitales) Filter genannt. Die Koeffizienten von h heißen Filterkoeffizienten und die Länge von h (falls endlich) Filterlänge. Das Spektrum eines Filters wird auch als dessen Frequenzgang bezeichnet. Betrag und Phase des Frequenzgangs heißen Betragsfrequenzgang und Phasengang.

Als Prototypen für cosinus-modulierte Filterbänke werden später nur endliche *kausale* Filter betrachtet. Ein Filter $H(z)$ ist kausal, wenn seine Koeffi-

zienten für alle negativen Indizes $n \in \mathbb{Z}_{<0}$ verschwinden:

$$H(z) = \sum_{n=0}^N h(n)z^{-n}.$$

Ein kausales digitales Filter benutzt zur Berechnung eines Ausgangswertes also nur den gegenwärtigen Eingangswert und vergangene Eingangswerte, sodass es prinzipiell auch für Echtzeitanwendungen geeignet ist.

Im obigen Sinne ist das Filter h so zu wählen, dass $|H(\omega)| = 0$ gilt für Frequenzen $\omega \in [-\pi, \pi]$, die gelöscht werden sollen. Anderenfalls sollte $|H(\omega)| = 1$ gelten. Eine sehr geläufige Anwendung ist z.B. die eines *Tiefpassfilters*. Im Idealfall soll dieser alle Frequenzen erhalten, die unterhalb einer gegebenen *Grenzfrequenz* ω_S liegen (*Durchlassbereich*). Oberhalb von ω_S sollen dagegen alle Frequenzen gelöscht werden (*Sperrbereich*).

Beispiel: Ein Filter $h \in \mathbb{R}^{\mathbb{Z}}$ soll ein Tiefpassfilter mit Grenzfrequenz $\omega_S = \frac{\pi}{4}$ sein. Im Idealfall hätte h den folgenden Betragsfrequenzgang:

$$|H(\omega)| = \begin{cases} 1 & \text{falls } |\omega| \leq \frac{\pi}{4} \\ 0 & \text{sonst} \end{cases}.$$

Mit den in der Praxis nur endlich vielen Filterkoeffizienten, kann ein solcher idealer Betragsfrequenzgang aber nur näherungsweise erreicht werden. Die Güte der Näherung hängt von der Filterlänge ab. Die Filterlänge ist wiederum proportional zur Berechnungszeit eines Koeffizienten der Faltung, also der Dauer für das Verarbeiten eines Koeffizienten des Eingangssignals (*Taktperiode*).

Abbildung 2.3.a zeigt den idealen Betragsfrequenzgang aus obigem Beispiel. Abbildungen 2.3.b und 2.3.c zeigen den entsprechenden Frequenzgang eines realen Filters $H(z)$, wobei die zweite Darstellung logarithmisch ist, nämlich in Dezibel (*dB*). An ihr lässt sich der Faktor, um den die Frequenzen im Sperrbereich abgeschwächt werden, die sogenannte *Dämpfung*, besser ablesen, als an der linearen Darstellung. Die Angabe des Betragsfrequenzganges in *dB* ist deshalb in der Signalverarbeitung üblich. Die Transformation einer reellen Zahl r in ihre *dB*-Darstellung ist durch $r \mapsto 20 \log_{10}(r)$ gegeben.

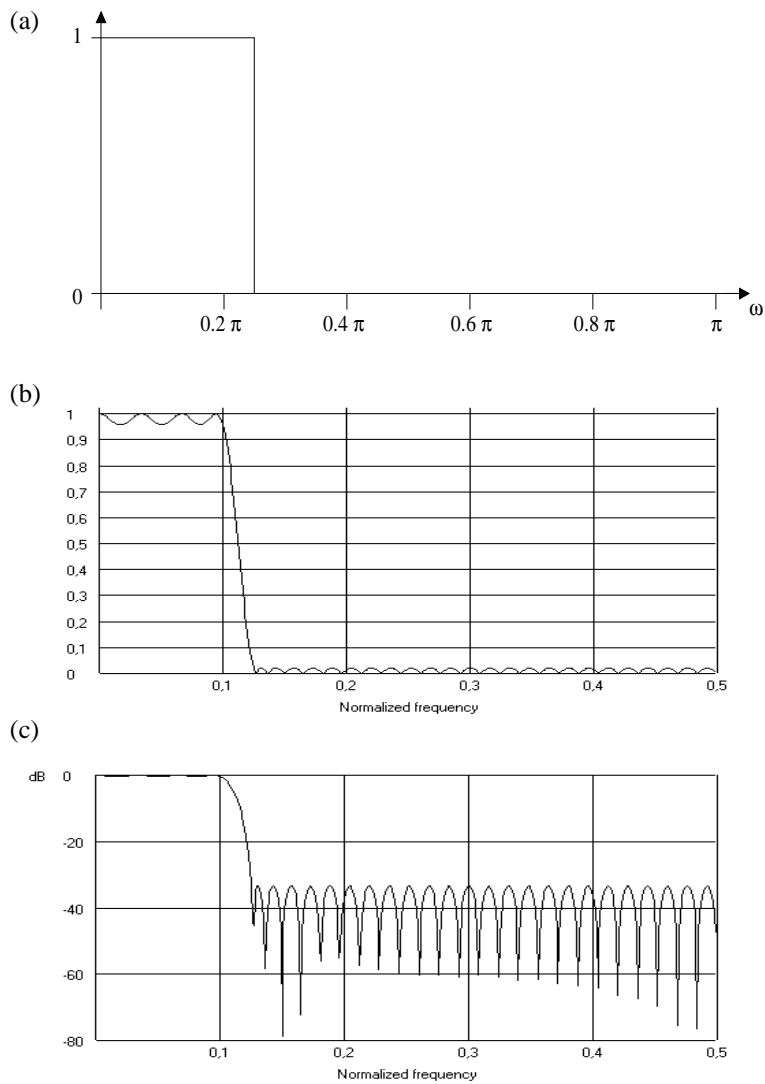


Abbildung 2.3: Darstellungen des Betragsfrequenzgangs eines Tiefpassfilters:
 (a) ideal; (b) real, linear; (c) real, logarithmisch

Kapitel 3

M -Kanal-Filterbänke

Eine M -Kanal-Filterbank ist eine Anordnung von M Filtern ($M \in \mathbb{N}$), die einen gemeinsamen Eingang (*Analysefilterbank*) oder einen gemeinsamen Ausgang (*Synthesefilterbank*) haben. Sie dient der Zerlegung eines Signals in mehrere Frequenzbänder (Teilbänder) bzw. deren Rekombinierung. Die M Teilbandsignale können dazwischen einzeln weiterverarbeitet werden, z.B. zwecks Komprimierung von Audiosignalen (MP3).

Wenn die Durchlassbereiche der einzelnen Filter die gleiche spektrale Breite haben, heißt eine Filterbank *gleichmäßig auflösend*, anderenfalls *ungleichmäßig auflösend*. Eine cosinus-modulierte Filterbank ist z.B. gleichmäßig auflösend (siehe Kapitel 4).

3.1 Taktreduktion und Takterhöhung

Das Anlegen des gleichen Eingangssignals an die M Analysefilter bedeutet, dass die zu verarbeitende Datenmenge, trotz gleichen Informationsgehalts, auf das M -fache steigt. Diese Redundanz bleibt auch nach der Analysefilterung erhalten. Man kann sich deshalb vorstellen, dass eine Rekonstruktion des Signals durch die Synthesefilterbank auch dann noch möglich ist, wenn in jedem Teilbandsignal nur ein Teil der Daten (Folgeglieder) zur weiteren Verwendung „ausgesiebt“ wird. Diese Aufgabe übernimmt in der digitalen Signalverarbeitung der Operator der sogenannten Taktreduktion. Dual dazu hat man zur Rekombination des Signals die Takterhöhung. Diese wichtigen Operatoren werden jetzt formal eingeführt.

Definition 3.1 Sei $N \in \mathbb{N}$. Die Taktoperatoren

$$[\downarrow N] : \mathbb{C}^{\mathbb{Z}} \rightarrow \mathbb{C}^{\mathbb{Z}},$$

$$[\uparrow N] : \mathbb{C}^{\mathbb{Z}} \rightarrow \mathbb{C}^{\mathbb{Z}}$$

werden definiert durch

$$([\downarrow N](x))(n) := x(Nn), \quad n \in \mathbb{Z}$$

und

$$([\uparrow N](x))(n) := \begin{cases} x(n/N) & \text{falls } N|n \\ 0 & \text{sonst} \end{cases}.$$

Sie heißen Taktreduktion bzw. Takterhöhung um den ganzzahligen Faktor N .

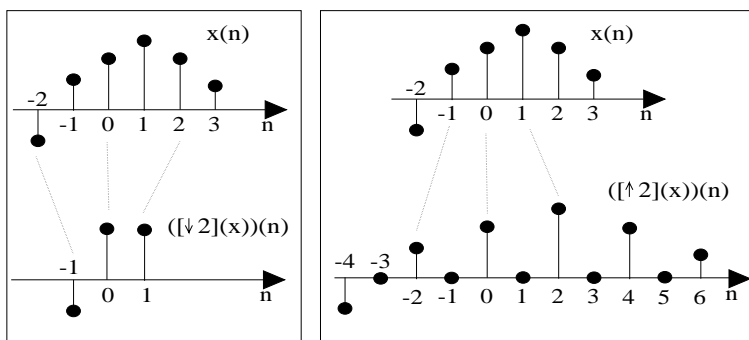


Abbildung 3.1: Veranschaulichung der Taktoperatoren (Faktor 2)

Von einem zeitdiskreten Signal wird bei der Taktreduktion also nur jedes N -te Folgeglied „abgegriffen“, während bei der Takterhöhung der „Zwischenraum“ je zweier Folgeglieder mit $N - 1$ Nullen aufgefüllt wird.

Für die Theorie von Filterbänken ist die Darstellung der z -Transformierten $z\{[\downarrow N](x)\}$ bzw. $z\{[\uparrow N](x)\}$ eines Signals $x \in \mathbb{C}^{\mathbb{Z}}$ mit Hilfe von $X(z)$ wichtig. Die z -Transformierten von $[\downarrow N](x)$ und $[\uparrow N](x)$ seien dazu auch mit $[\downarrow N](X(z))$ und $[\uparrow N](X(z))$ bezeichnet, wodurch $[\downarrow N]$ und $[\uparrow N]$ auch als Operatoren im z -Bereich erklärt sind.

Lemma 3.1 Sei $x \in \mathbb{C}^{\mathbb{Z}}$. Dann gelten

$$[\downarrow N](X(z)) = \frac{1}{N} \sum_{k=0}^{N-1} X(z^{1/N} W_N^k), \quad (3.1)$$

$$[\uparrow N](X(z)) = X(z^N), \quad (3.2)$$

wobei durch W_N , $N \in \mathbb{N}$, die N -te Einheitswurzel

$$W_N = e^{-2\pi i/N}$$

bezeichnet wird.

Beweis. Die zweite Gleichung ergibt sich mit der Definition von $[\uparrow N](x)$:

$$\begin{aligned} [\uparrow N](X(z)) &= \sum_{n \in \mathbb{Z}} ([\uparrow N](x))(n)z^{-n} = \sum_{k \in \mathbb{Z}} ([\uparrow N](x))(kN)z^{-kN} \\ &= \sum_{k \in \mathbb{Z}} x(k)z^{-kN} = X(z^N). \end{aligned}$$

Für die Herleitung der ersten Gleichung definiere $y \in \mathbb{C}^{\mathbb{Z}}$ durch

$$y(n) := \begin{cases} x(n), & \text{falls } N|n \\ 0, & \text{sonst} \end{cases}.$$

Dann ist $([\downarrow N](x))(n) = x(Nn) = y(Nn)$ und

$$\begin{aligned} [\downarrow N](X(z)) &= \sum_{n \in \mathbb{Z}} ([\downarrow N](x))(n)z^{-n} = \sum_{n \in \mathbb{Z}} y(Nn)z^{-n} \\ &= \sum_{k \in \mathbb{Z}} y(k)z^{-k/N} = Y(z^{1/N}). \end{aligned}$$

Bleibt $Y(z)$ durch $X(z)$ auszudrücken. Dazu sei $\alpha_N \in \mathbb{C}^{\mathbb{Z}}$ gegeben durch

$$\alpha_N(n) := \begin{cases} 1 & \text{falls } N|n \\ 0 & \text{sonst} \end{cases},$$

sodass $y(n) = \alpha_N(n)x(n)$, $n \in \mathbb{N}$, gilt. Man kann α_N schreiben als

$$\alpha_N(n) = \frac{1}{N} \sum_{k=0}^{N-1} W_N^{-kn}.$$

Hiermit bekommt man

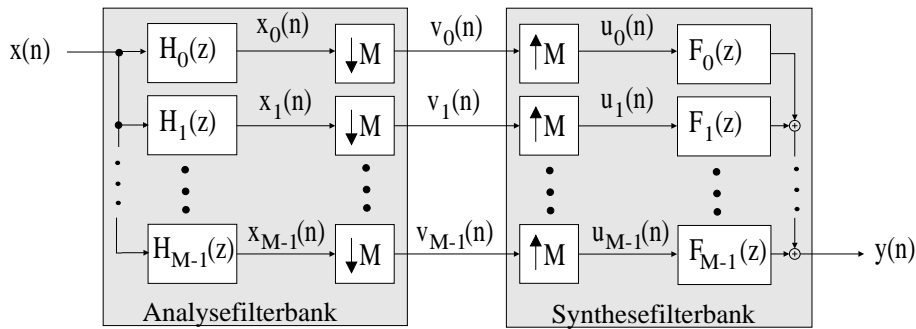
$$Y(z) = \sum_{n \in \mathbb{Z}} \frac{1}{N} \sum_{k=0}^{N-1} W_N^{-kn} x(n) z^{-n} = \frac{1}{N} \sum_{k=0}^{N-1} \sum_{n \in \mathbb{Z}} x(n) (zW_N^k)^{-n}.$$

Die innere Summe auf der rechten Seite ist gleich $X(zW_N^k)$, womit Gleichung (3.1)

$$[\downarrow N](X(z)) = Y(z^{1/N}) = \frac{1}{N} \sum_{k=0}^{N-1} X(z^{1/N} W_N^k)$$

folgt. □

Bei gleichmäßig auflösenden M -Kanal-Filterbänken, zu denen die cosinusmodulierte Filterbank gehört, kann nun bestenfalls jedem Analysefilter eine Taktreduktion um den Faktor M folgen. In Abbildung 3.2 ist ein entsprechendes Analyse/Synthese-System als Schaltbild dargestellt. Die Teilbandsignale werden vor der Synthesefilterbank wieder einer Takterhöhung um den Faktor M unterzogen und danach zum Ausgangssignal y aufaddiert. Ein konkretes (sehr simples) Beispiel einer solchen Filterbank wird am Ende des folgenden Abschnitts gegeben.


 Abbildung 3.2: Allgemeines Schema einer M -Kanal-Filterbank

3.2 Perfekte Rekonstruktion

Eine wichtige Eigenschaft einer Filterbank ist die perfekte Rekonstruktion, PR-Eigenschaft genannt. Sie bedeutet, dass das Ausgangssignal y , bis auf eine Verschiebung der Folgeglieder (Systemverzögerung) und evtl. einen skalaren Faktor $r \in \mathbb{R} \setminus \{0\}$, dem Eingangssignal x gleicht.

Definition 3.2 Sei $\Phi : \mathbb{C}^{\mathbb{Z}} \rightarrow \mathbb{C}^{\mathbb{Z}}$ die Übertragungsfunktion einer Filterbank. Die Filterbank heißt perfekt rekonstruierend (PR), wenn es ein $r \in \mathbb{R} \setminus \{0\}$ und ein $D \in \mathbb{N}_0$ gibt, sodass

$$y(n) := (\Phi(x))(n) = rx(n - D)$$

für alle $x \in \mathbb{C}^{\mathbb{Z}}$ und alle $n \in \mathbb{Z}$ gilt. Bei geringfügiger Verletzung der PR-Eigenschaft ($y(n) = rx(n - D) + \epsilon(n)$) spricht man von fast perfekter Rekonstruktion.

Die Forderung nach perfekter Rekonstruktion erklärt auch die Notwendigkeit der Synthesefilterung. Die Frequenzspektren der Teilbandsignale müssen sich offenbar überlappen, um eine Wiederherstellung des Eingangs zu erlauben. Denn nur ideale Frequenzgänge ließen sich lückenlos aneinander reihen, ohne zu überlappen. Eine schlichte Addition der Teilbandsignale kann dann aber nicht mehr exakt das ursprüngliche Signal ergeben. Dieser Fehler muss durch eine Synthesefilterung korrigiert werden.

Für die allgemeine M -Kanal-Filterbank in Abbildung 3.2 lässt sich mit Hilfe von (3.1) und (3.2) eine Beschreibung des Ausgangssignals im z -Bereich herleiten [Kl]:

$$Y(z) = \frac{1}{M} \sum_{l=0}^{M-1} X(zW_M^l) \sum_{k=0}^{M-1} H_k(zW_M^l) F_k(z). \quad (3.3)$$

Denn nach den Analysefilterungen hat man die Signale $X_k(z) = X(z)H_k(z)$, woraus nach Taktreduktion um den Faktor M die Teilbandsignale

$V_k(z) = \frac{1}{M} \sum_{l=0}^{M-1} H_k(z^{\frac{1}{M}} W_M^l) X(z^{\frac{1}{M}} W_M^l)$ entstehen. Die anschließende Takterhöhung in der Synthesefilterbank ergibt die Signale

$$U_k(z) = V_k(z^M) = \frac{1}{M} \sum_{l=0}^{M-1} H_k(z W_M^l) X(z W_M^l),$$

die, multipliziert mit den $F_k(z)$ und aufaddiert, das Ausgangssignal $Y(z)$ entsprechend (3.3) ergeben.

Mit den Abkürzungen

$$A_l(z) := \frac{1}{M} \sum_{k=0}^{M-1} H_k(z W_M^l) F_k(z), \quad l = 0, \dots, M-1, \quad T(z) := A_0(z),$$

kann (3.3) geschrieben werden als

$$Y(z) = T(z)X(z) + \sum_{l=1}^{M-1} X(z W_M^l) A_l(z).$$

Perfekte Rekonstruktion ist hier also gegeben, wenn $T(z) = r z^{-D}$ für $D \in \mathbb{N}_0$ und $r \in \mathbb{R} - \{0\}$ gilt und die Summe auf der rechten Seite verschwindet.

$T(z)$ wird in der Signalverarbeitung auch *Verzerrungsfunktion* genannt. Die anderen $A_l(z)$, $l = 1, \dots, M-1$, mit denen die unerwünschten Terme $X(z W_M^l)$ gewichtet werden, heißen *Aliasfunktionen*.

Das einfachste PR-System liegt vor, wenn man die Analyse- und Synthesefilter der allgemeinen M -Kanal-Filterbank als simple Verzögerungen definiert durch

$$H_k(z) := z^{-k}, \quad F_k(z) := z^{-(M-1-k)}, \quad k = 0, \dots, M-1.$$

In diesem Fall gilt

$$A_l(z) = \frac{1}{M} \sum_{k=0}^{M-1} z^{-k} W_M^l z^{-(M-1-k)} = z^{-(M-1)} \frac{1}{M} \sum_{k=0}^{M-1} W_M^l.$$

Im Falle $l = 0$ ist die Summe ganz rechts gleich M , d.h. $T(z) = z^{-M-1}$. Dagegen verschwinden für $l = 1, \dots, M-1$ mit dieser Summe die Aliasfunktionen, sodass $Y(z) = z^{-(M-1)} X(z)$ folgt. Dass das System die verzögerte Ausgangsfolge $y(n) = x(n - M + 1)$ ergibt, kann man auch auf direktem Wege leicht nachrechnen. Abbildung 3.3 zeigt eine Implementierung dieses PR-Systems mit Hilfe von Verzögerungsketten.

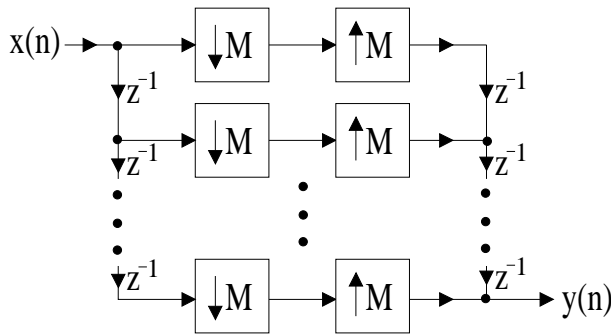


Abbildung 3.3: Einfaches PR-System mit Verzögerungsketten

3.3 Polyphasenzerlegung von Filtern

Die Implementierung der allgemeinen M -Kanal-Filterbank, wie sie in Abbildung 3.2 (Seite 15) gezeigt wird, ist aufgrund der Reihenfolge von Filtern und Taktoperatoren (Taktreduktion und Takterhöhung) ineffizient. So wird bei der Analysefilterung jedes Glied eines Teilbandsignals berechnet, obwohl anschließend $M - 1$ von M Koeffizienten „weggeschmissen“ werden. Vor der Synthese werden die Signale wieder gestreckt, sodass auch hier unnötig viel gerechnet werden muss. Günstig wäre es, wenn man die Reihenfolge von Filtern und Taktoperatoren einfach umkehren könnte. Dazu müssten für jeden Filter $G(z)$ und alle Signale $X(z)$ die Beziehungen $[\downarrow N](G(z)X(z)) = G(z)([\downarrow N](X(z)))$ und $G(z)([\uparrow N](X(z))) = [\uparrow N](G(z)X(z))$, $N \in \mathbb{N}$, gelten. Das stimmt aber nicht ganz. Stattdessen bekommt man vermöge (3.1) und (3.2) das

Lemma 3.2 Für alle Signale $x \in \mathbb{C}^{\mathbb{Z}}$ und Filter $g \in \mathbb{C}^{\mathbb{Z}}$ gelten die Identitäten

$$[\downarrow N](G(z^N)X(z)) = G(z)([\downarrow N](X(z))) \quad (3.4)$$

und

$$G(z^N)([\uparrow N](X(z))) = [\uparrow N](G(z)X(z)). \quad (3.5)$$

Im Englischen werden die Identitäten (3.4) und (3.5) als *Noble Identities* bezeichnet. Ein Tauschen der Reihenfolge von Filtern und Taktoperatoren ist also nicht ohne weiteres erlaubt. Es gibt aber dennoch eine Möglichkeit, die Kombinationen von Filtern und Taktoperatoren effizient zu implementieren. Sie beruht auf der Darstellung der Filter durch ihre sogenannten *Polyphasenkomponenten*, die im Folgenden eingeführt wird. Ferner wird sich mit den Polyphasenkomponenten der Filter einer M -Kanal-Filterbank eine sehr griffige Beschreibung der PR-Eigenschaft ergeben (Abschnitt 3.4).

Sei $H(z) = \sum_{n \in \mathbb{Z}} h(n)z^{-n}$ ein Filter. Unterteilt man die Koeffizienten $h(n)$ nach geraden und ungeraden $n \in \mathbb{Z}$, so kann man

$$H(z) = \sum_{n \in \mathbb{Z}} h(2n)z^{-2n} + z^{-1} \sum_{n \in \mathbb{Z}} h(2n+1)z^{-2n}$$

schreiben. Mit

$$E_0(z) = \sum_{n \in \mathbb{Z}} h(2n)z^{-n}, \quad E_1(z) = \sum_{n \in \mathbb{Z}} h(2n+1)z^{-n},$$

hat $H(z)$ dann die Darstellung

$$H(z) = E_0(z^2) + z^{-1}E_1(z^2).$$

Sie wird Typ1-Polyphasendarstellung zum Faktor 2 genannt. Beim Filter $H(z) = 1 + 2z^{-1} + 3z^{-2} + 4z^{-3}$ wäre z.B.

$$E_0(z) = 1 + 3z^{-1}, \quad E_1(z) = 2 + 4z^{-1}.$$

Das Konzept kann analog von 2 auf jede ganze Zahl erweitert werden.

Sei $M \in \mathbb{Z}$. Mit einer Unterteilung der Koeffizienten $h(n)$ nach Zugehörigkeit ihrer Indizes zu den Restklassen von \mathbb{Z} modulo M ($n \in M\mathbb{Z} + l$, $l = 0, \dots, M-1$) schreibe

$$H(z) = \sum_{l=0}^{M-1} z^{-l} \sum_{n \in \mathbb{Z}} h(nM+l)z^{-nM}.$$

Kompakt geschrieben ist dies

$$H(z) = \sum_{l=0}^{M-1} z^{-l} E_l(z^M) \quad (\text{Typ1 - Polyphasendarstellung}), \quad (3.6)$$

wobei die *Polyphasenkomponenten* $E_l(z)$ durch

$$E_l(z) = \sum_{n \in \mathbb{Z}} e_l(n)z^{-n}$$

mit

$$e_l(n) = h(Mn+l), \quad l = 0, \dots, M-1, \quad n \in \mathbb{Z}.$$

gegeben sind. Die durch

$$R_l(z) := E_{M-1-l}(z), \quad l = 0, \dots, M-1$$

definierten permutierten Versionen der $E_l(z)$ heißen Typ2-Polyphasenkomponenten und die Darstellung

$$H(z) = \sum_{l=0}^{M-1} z^{-(M-1-l)} R_l(z^M) \quad (3.7)$$

wird auch *Typ2-Polyphasendarstellung* von $H(z)$ genannt.

Mit Hilfe von (3.6) und (3.7) bekommt man eine effiziente Implementierung von $[\downarrow M](H(z)X(z))$ und $H(z)([\uparrow M](X(z)))$:

Lemma 3.3 *Sei $H(z)$ ein Filter mit den Polyphasendarstellungen $H(z) = \sum_{l=0}^{M-1} z^{-l} E_l(z^M)$ und $H(z) = \sum_{l=0}^{M-1} z^{-(M-1-l)} R_l(z^M)$ vom Typ1 bzw. Typ2. Für alle Signale $X(z)$ gelten dann*

$$[\downarrow M](H(z)X(z)) = \sum_{l=0}^{M-1} E_l(z)([\downarrow M](z^{-l}X(z))) \quad (3.8)$$

und

$$H(z)([\uparrow M](X(z))) = \sum_{l=0}^{M-1} z^{-(M-1-l)} [\uparrow M](R_l(z)X(z)). \quad (3.9)$$

Beweis. Offenbar ist

$$[\downarrow M](H(z)X(z)) = [\downarrow M] \left(\sum_{l=0}^{M-1} z^{-l} E_l(z^M) X(z) \right).$$

Vertauschen von Summation und $[\downarrow M]$ (die Taktoperatoren sind linear) und Anwenden der Identität (3.4) ergibt (3.8). Analog ergibt sich (3.9) mit Hilfe der Identität (3.5). \square

Abbildung 3.4 veranschaulicht die Äquivalenzen aus Lemma 3.3 schalttechnisch.

3.4 Beschreibung perfekter Rekonstruktion durch Polyphasenmatrizen

Wir betrachten wieder die allgemeine M -Kanal-Filterbank mit Analysefiltern $H_0(z), \dots, H_{M-1}(z)$ und Synthesefiltern $F_0(z), \dots, F_{M-1}(z)$, wie sie in Abbildung 3.2 gezeigt wird. Analysefilterbank und Synthesefilterbank sind durch die Vektoren

$$\mathbf{h}(z) = \begin{bmatrix} H_0(z) \\ H_1(z) \\ \vdots \\ H_{M-1}(z) \end{bmatrix}, \quad \mathbf{f}(z) = \begin{bmatrix} F_0(z) \\ F_1(z) \\ \vdots \\ F_{M-1}(z) \end{bmatrix},$$

beschrieben. Mit $\mathbf{h}(z)$, $\mathbf{f}(z)$, dem Eingangssignal $X(z)$ und dem Ausgangssignal $Y(z)$ bekommt man einen vektorwertige Ausdruck für die Übertragungsfunktion der Filterbank:

$$Y(z) = \mathbf{f}^T(z) \left([\uparrow M] \circ [\downarrow M](\mathbf{h}(z)X(z)) \right). \quad (3.10)$$

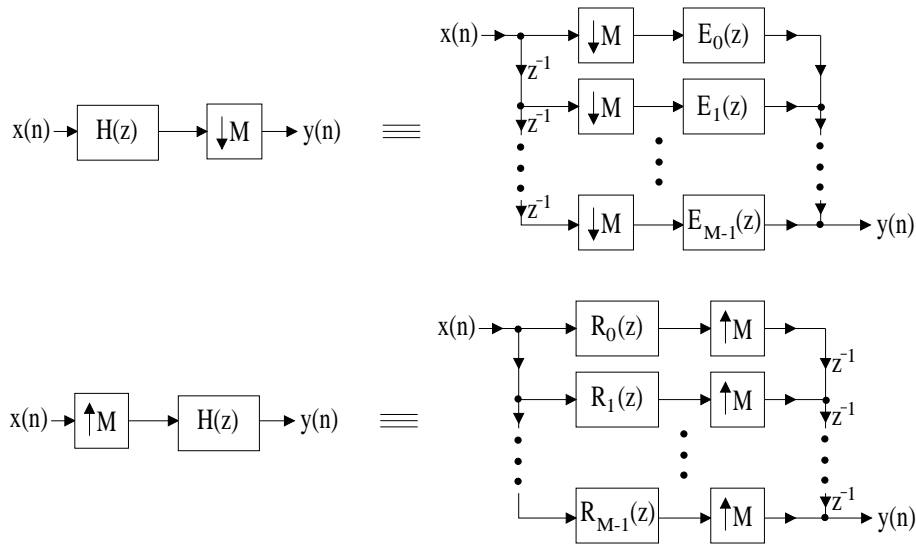


Abbildung 3.4: Polyphasen-Implementierungen der Filter/Taktoperator-Kombinationen

Wobei das Anwenden eines Taktoperators auf einen Vektor bedeuten soll, dass dieser auf jede Komponente des Vektors angewendet wird. Diese vektorwertige Gleichung soll nun mit Hilfe sogenannter *Polyphasenmatrizen* in eine Gleichung umformuliert werden, die eine sehr übersichtliche Beschreibung der PR-Bedingungen der Filterbank erlaubt.

Die Analyse- und Synthesefilter seien gemäß (3.6) und (3.7) in Polyphasenkomponenten $E_{k,l}(z)$ bzw. $R_{l,k}(z)$ zerlegt:

$$H_k(z) = \sum_{l=0}^{M-1} z^{-l} E_{k,l}(z^M),$$

$$F_k(z) = \sum_{l=0}^{M-1} z^{-(M-1-l)} R_{l,k}(z^M).$$

Definition 3.3 Die Matrix

$$\mathbf{E}(z) = \begin{bmatrix} E_{0,0}(z) & \dots & E_{0,M-1}(z) \\ \vdots & \ddots & \vdots \\ E_{M-1,0}(z) & \dots & E_{M-1,M-1}(z) \end{bmatrix}$$

über dem Ring aller Laurentpolynome wird *Polyphasenmatrix der Analysefilterbank* genannt. Entsprechend heißt die Matrix

$$\mathbf{R}(z) = \begin{bmatrix} R_{0,0}(z) & \dots & R_{0,M-1}(z) \\ \vdots & \ddots & \vdots \\ R_{M-1,0}(z) & \dots & R_{M-1,M-1}(z) \end{bmatrix}$$

Polyphasenmatrix der Synthesefilterbank. Ferner seien die Vektoren $\mathbf{e}(z)$ und $\bar{\mathbf{e}}(z)$ definiert durch

$$\mathbf{e}(z) = \begin{bmatrix} 1 \\ z^{-1} \\ \vdots \\ z^{-(M-1)} \end{bmatrix}, \quad \bar{\mathbf{e}}(z) = \begin{bmatrix} z^{-(M-1)} \\ z^{-(M-2)} \\ \vdots \\ 1 \end{bmatrix}.$$

Die Vektoren $\mathbf{e}(z)$ und $\bar{\mathbf{e}}(z)$ stehen für Verzögerungsketten (siehe Abbildung 3.3, Seite 17). Mit Hilfe von (3.8) und (3.9) kann nachgerechnet werden, dass für alle Signale x und v_0, \dots, v_{M-1} aus $\mathbb{C}^{\mathbb{Z}}$ die Beziehungen

$$[\downarrow M](\mathbf{h}(z)X(z)) = \mathbf{E}(z)[\downarrow M](\mathbf{e}(z)X(z))$$

und

$$\mathbf{f}^T(z) \left([\uparrow M] \begin{bmatrix} V_0(z) \\ V_1(z) \\ \vdots \\ V_{M-1}(z) \end{bmatrix} \right) = \bar{\mathbf{e}}^T(z) [\uparrow M] \left(\mathbf{R}(z) \begin{bmatrix} V_0(z) \\ V_1(z) \\ \vdots \\ V_{M-1}(z) \end{bmatrix} \right)$$

gelten. Mit (3.10) ergibt sich daraus die Gleichung

$$Y(z) = \bar{\mathbf{e}}^T(z) [\uparrow M] \left(\mathbf{R}(z) \mathbf{E}(z) [\downarrow M] (\mathbf{e}(z)X(z)) \right). \quad (3.11)$$

Das Schaltbild in Abbildung 3.5 veranschaulicht Gleichung (3.11). Es entspricht also der Darstellung in Abbildung 3.2 auf Seite 15. Wenn $\mathbf{R}(z)\mathbf{E}(z) =$

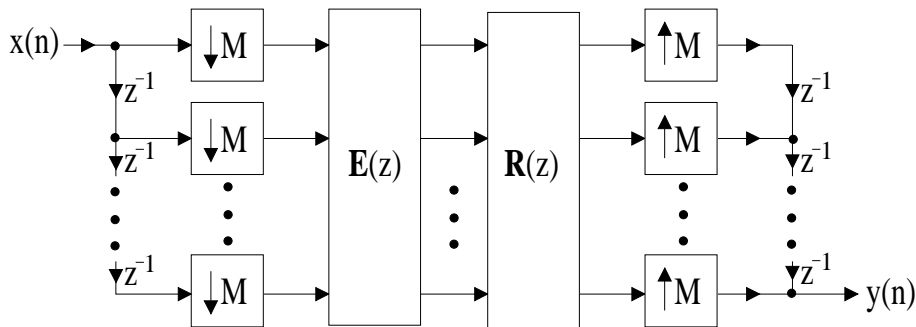


Abbildung 3.5: Darstellung einer M -Kanal-Filterbank mit Polyphasenmatrizen

I gilt, wird (3.11) zu $Y(z) = \bar{\mathbf{e}}^T(z) ([\uparrow M] \circ [\downarrow M]) (\mathbf{e}(z)X(z))$. Dies entspricht genau dem Ausgang der einfachen PR-Filterbank, die in Abschnitt 3.2 (Abbildung 3.3) betrachtet wurde. Dort wurde $Y(z) = z^{-(M-1)}X(z)$ errechnet. Eine M -Kanal-Filterbank, deren Polyphasenmatrizen $\mathbf{E}(z)$ und $\mathbf{R}(z)$ der

Bedingung $\mathbf{R}(z)\mathbf{E}(z) = \mathbf{I}$ genügen, ist demnach perfekt rekonstruierend mit einer Verzögerung von $M - 1$ Takten.

Die PR-Eigenschaft ist offenbar auch dann erfüllt, wenn $\mathbf{R}(z)\mathbf{E}(z) = z^{-D}\mathbf{I}$, $D \in \mathbb{Z}$, gilt, denn $z^{-D}\mathbf{I}$ kann als $z^{-MD}\mathbf{I}$ vor den Operator der Takterhöhung gezogen werden. Man hat dann perfekte Rekonstruktion mit einer Verzögerung von $MD + M - 1$ Takten.

Satz 3.1 *Eine allgemeine M -Kanal-Filterbank mit Analyse-Polyphasenmatrix $\mathbf{E}(z)$ und Synthese-Polyphasenmatrix $\mathbf{R}(z)$ ist perfekt rekonstruierend, falls ein $D \in \mathbb{Z}$ existiert, derart dass $\mathbf{R}(z)\mathbf{E}(z) = z^{-D}\mathbf{I}$ gilt. Die Ein/Ausgangs-Beziehung der Filterbank ist dann durch $Y(z) = z^{-(MD+M-1)}X(z)$ gegeben.*

Eine Hinreichende und Notwendige Bedingung für perfekte Rekonstruktion, die an $\mathbf{R}(z)\mathbf{E}(z)$ gestellt werden kann, wird in [Vai] bewiesen. Sie ergibt sich dort als Folgerung einer entsprechenden Bedingung für die Freiheit von Aliaskomponenten:

Satz 3.2 *Eine allgemeine M -Kanal-Filterbank mit Analyse-Polyphasenmatrix $\mathbf{E}(z)$ und Synthese-Polyphasenmatrix $\mathbf{R}(z)$ ist perfekt rekonstruierend genau dann, wenn $\mathbf{R}(z)\mathbf{E}(z)$ die Form*

$$\mathbf{R}(z)\mathbf{E}(z) = rz^{-D} \begin{bmatrix} \mathbf{0} & \mathbf{I}_{M-d} \\ z^{-1}\mathbf{I}_d & \mathbf{0} \end{bmatrix},$$

$$r \in \mathbb{R} \setminus \{0\}, \quad D \in \mathbb{Z}, \quad 0 \leq d \leq M - 1,$$

hat. Die Ein/Ausgangs-Beziehung der Filterbank ist dann durch

$$Y(z) = rz^{-(MD+d+M-1)}X(z)$$

gegeben.

Damit steht für jede mögliche Systemverzögerung eine PR-Bedingung zur Verfügung. Diese Arbeit wird sich jedoch mit Verzögerungen der Form $MD + M - 1$ begnügen und auch auf den skalaren Faktor verzichten, also nur die Bedingung

$$\mathbf{R}(z)\mathbf{E}(z) = z^{-D}\mathbf{I}, \quad D \in \mathbb{Z}, \quad (3.12)$$

benutzen.

Kapitel 4

Cosinus-modulierte Filterbänke

Im vorherigen Kapitel waren Analyse- und Synthesefilter einer Filterbank unabhängig voneinander wählbar. Bei der Klasse der *modulierten* Filterbänke ergeben sich die Teilbandfilter stattdessen als frequenzverschobene Versionen eines einzigen *Tiefpassprototyps* $P(z)$ bzw. $Q(z)$. Dazu werden die Koeffizienten der Prototypfilter mit den Koeffizienten bestimmter *Modulationsfolgen* $w_k^{(a)}$ bzw. $w_k^{(s)}$, $k = 0, 1, \dots, M - 1$, multipliziert:

$$\begin{aligned}h_k(n) &= p(n)w_k^{(a)}(n) \quad n \in \mathbb{Z} \\f_k(n) &= q(n)w_k^{(s)}(n) \quad n \in \mathbb{Z}.\end{aligned}$$

Eine derartige Konstruktion bedeutet im Vergleich zu Filterbänken mit unabhängig entworfenen Teilbandfiltern unter anderem eine erhebliche Reduktion des Entwurfsaufwands, da jeweils nur ein Analyse- und Syntheseprototyp entworfen werden muss.

Der Betragsfrequenzgang eines Filters $P(z)$ kann um die Frequenz $\phi \in [-\pi, \pi]$ verschoben werden, wenn er mit einer komplexen Folge der Form $e^{i(\phi n + \alpha)}$ moduliert wird: Setzt man

$$h(n) = p(n)e^{i(\phi n + \alpha)} = e^{i\alpha}(p(n)e^{i\phi n}), \quad \phi, \alpha \in [-\pi, \pi],$$

so gilt für $\omega \in [-\pi, \pi]$

$$H(\omega) = \sum_{n \in \mathbb{Z}} h(n)e^{-in\omega} = e^{i\alpha} \sum_{n \in \mathbb{Z}} p(n)e^{-in(\omega - \phi)} = e^{i\alpha} P(\omega - \phi),$$

also

$$|H(\omega)| = |P(\omega - \phi)|.$$

Durch $h_k(n) = p(n)e^{i(\frac{2\pi}{M}kn + \alpha_k)}$, $k = 0, 1, \dots, M - 1$, werden z.B. M zu $P(z)$ gleichmäßig frequenzverschobene Teilbandfilter $H_k(z)$ erzeugt. Abbildung

4.1 zeigt die Lage der entstehenden Teilbänder bei geeignetem Prototypfilter $P(z)$.

Für ein reelles Signal $x \in \mathbb{R}^{\mathbb{Z}}$ gilt $X(\omega) = \overline{X(-\omega)}$, $\omega \in [0, \pi]$, d.h. $|X(\omega)|$ ist

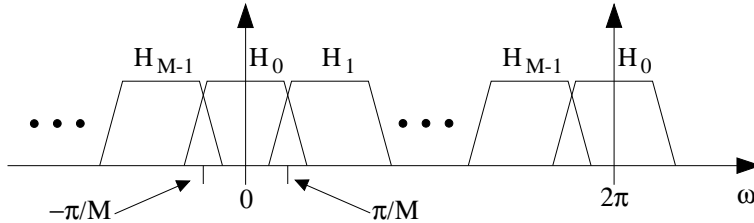


Abbildung 4.1: Lage der Teilbänder $|H_k(\omega)|$ bei komplexer Modulation $h_k(n) = p(n)e^{i(\frac{2\pi}{M}kn + \alpha_k)}$, $k = 0, 1, \dots, M - 1$.

symmetrisch um 0. Will man zu einem reellen Signal auch reelle Teilbandsignale erhalten, so braucht man Filter, deren Betragsfrequenzgänge ebenfalls symmetrisch sind, d.h. Filter mit reellen Koeffizienten. Diesem Zweck dient die *Cosinus-Modulation*. Eine modulierte Version $H(z)$ des reellwertigen Prototypen $P(z)$ wird dabei durch eine reelle Modulationsfolge der Form $2 \cos(\phi n + \alpha)$ erzeugt:

$$h(n) = 2p(n) \cos(\phi n + \alpha), \quad \phi, \alpha \in [-\pi, \pi], \quad \text{Cosinus - Modulation.}$$

Wegen $2 \cos(\phi n + \alpha) = e^{i(\phi n + \alpha)} + e^{-i(\phi n + \alpha)}$ entsteht $|H(\omega)|$ durch Verschieben von $|P(\omega)|$ um ϕ in beide Richtungen. Beispielsweise bekommt man mit

$$h_k(n) = 2p(n) \cos\left(\frac{\pi}{M}\left(k + \frac{1}{2}\right)n + \alpha_k\right), \quad k = 0, 1, \dots, M - 1,$$

gleichmäßig frequenzverschobene Filter $H_k(z)$, wie in Abbildung 4.2 skizziert.

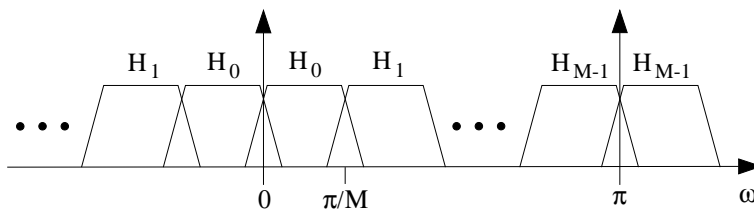


Abbildung 4.2: Lage der Teilbänder $|H_k(\omega)|$ bei Cosinus-Modulation $h_k(n) = 2p(n) \cos(\frac{\pi}{M}(k + \frac{1}{2})n + \alpha_k)$, $k = 0, 1, \dots, M - 1$.

4.1 PR-Bedingungen an den Prototypfilter

Wir betrachten cosinus-modulierte Filterbänke, bei denen Analyse- und Synthesefilter aus einem einzigen Prototypfilter hervorgehen. Der Faktor aller Taktoperatoren soll, wie bisher stets angenommen wurde, gleich der Zahl M der Teilbänder sein, die der Einfachheit halber gerade sei. Die Systemverzögerung D_{Sys} soll von der Form $2MD + 2M - 1$ (vgl. Satz 3.1) und die Länge der Prototypfilter ein Vielfaches von $2M$ sein. Ein gutes Modulationsschema für diesen Fall ist [HKN]

$$\boxed{\begin{aligned} h_k(n) &= 2p(n) \cos\left(\frac{\pi}{M}\left(k + \frac{1}{2}\right)\left(n - \frac{D_{Sys}}{2}\right) + \phi_k\right), \\ f_k(n) &= 2p(n) \cos\left(\frac{\pi}{M}\left(k + \frac{1}{2}\right)\left(n - \frac{D_{Sys}}{2}\right) - \phi_k\right), \end{aligned}} \quad (4.1)$$

mit $n = 0, \dots, L_p - 1$, $k = 0, \dots, M - 1$, $\phi_k = (-1)^k \pi/4$ und der Systemverzögerung $D_{Sys} = 2MD + 2M - 1$.

Satz 4.1 Sei $M \in \mathbb{N}$ gerade, $m \in \mathbb{N}$, $D \in \mathbb{N}_0$ und setze $L_p = 2mM$ und $D_{Sys} = 2MD + 2M - 1$. Sei $P(z) = \sum_{n=0}^{L_p-1} p(n)z^{-n}$ ein Prototypfilter einer cosinus-modulierten Filterbank mit M Kanälen (nach Schema (4.1)), die eine Systemverzögerung von D_{Sys} Takten habe. Seien $P_l(z) = \sum_{n=0}^{m-1} p(2Mn + l)z^{-n}$, $l = 0, \dots, 2M - 1$, die $2M$ Typ1-Polyphasenkomponenten der Länge m von $P(z)$. Die Filterbank ist genau dann perfekt rekonstruierend, wenn der Prototypfilter die folgenden Bedingungen erfüllt:

$$\boxed{\begin{aligned} P_l(z)P_{2M-1-l}(z) + P_{M+l}(z)P_{M-1-l}(z) &= \frac{1}{2M}z^{-D}, \\ l &= 0, \dots, \frac{M}{2} - 1 \end{aligned}} \quad (4.2)$$

Beweis. Mit den Matrizen

$$\begin{aligned} \mathbf{P}_0(z^2) &= \text{diag}[P_0(-z^2), \dots, P_{M-1}(-z^2)], \\ \mathbf{P}_1(z^2) &= \text{diag}[P_M(-z^2), \dots, P_{2M-1}(-z^2)] \end{aligned}$$

und der Modulationsmatrix \mathbf{T}_1 , definiert durch

$$\begin{aligned} [\mathbf{T}_1]_{k,n} &= 2 \cos\left(\frac{\pi}{M}\left(k + \frac{1}{2}\right)\left(n - \frac{D_{Sys}}{2}\right) + \phi_k\right), \\ k &= 0, \dots, M - 1, j = 0, \dots, 2M - 1, \end{aligned}$$

kann für die Polyphasenmatrix $\mathbf{E}(z)$ der durch (4.1) definierten Analysefilterbank die Beziehung

$$\mathbf{E}(z) = \mathbf{T}_1 \begin{bmatrix} \mathbf{P}_0(z^2) \\ z^{-1}\mathbf{P}_1(z^2) \end{bmatrix} \quad (4.3)$$

nachgerechnet werden [HKN]. Analog gilt für die Polyphasenmatrix $\mathbf{R}(z)$ der Synthesefilterbank:

$$\mathbf{R}(z) = [z^{-1}\mathbf{Q}_1(z^2), \mathbf{Q}_0(z^2)]\mathbf{T}_2^T, \quad (4.4)$$

mit

$$\begin{aligned} \mathbf{Q}_0(z^2) &= \text{diag}[P_{M-1}(-z^2), \dots, P_0(-z^2)], \\ \mathbf{Q}_1(z^2) &= \text{diag}[P_{2M-1}(-z^2), \dots, P_M(-z^2)] \end{aligned}$$

und

$$\begin{aligned} [\mathbf{T}_2]_{k,n} &= 2 \cos \left(\frac{\pi}{M} \left(k + \frac{1}{2} \right) \left(2M - 1 - n - \frac{D_{Sys}}{2} \right) - \phi_k \right), \\ k &= 0, \dots, M-1, j = 0, \dots, 2M-1. \end{aligned}$$

Ferner gilt für die Modulationsmatrizen [HKN]:

$$\mathbf{T}_2^T \mathbf{T}_1 = 2M \left((-1)^D \mathbf{I}_{2M} + \begin{bmatrix} \mathbf{J}_M & \mathbf{0} \\ \mathbf{0} & -\mathbf{J}_M \end{bmatrix} \right), \quad (4.5)$$

wobei mit \mathbf{J}_M die $M \times M$ -Antidiagonalmatrix bezeichnet ist:

$$[\mathbf{J}_M]_{i,j} = \begin{cases} 1 & \text{falls } i = M + 1 - j \\ 0 & \text{sonst} \end{cases}.$$

Ist \mathbf{A} eine $M \times M$ -Matrix, so erhält man $\mathbf{J}_M \mathbf{A}$ (bzw. $\mathbf{A} \mathbf{J}_M$) durch Umkehren der Reihenfolge der Zeilen (bzw. Spalten) von \mathbf{A} . Insbesondere ist $\mathbf{J}_M^2 = \mathbf{I}_M$ und für die Diagonalmatrizen $\mathbf{Q}_i(z^2)$, $\mathbf{P}_i(z^2)$ gilt:

$$\mathbf{Q}_i(z^2) = \mathbf{J}_M \mathbf{P}_i(z^2) \mathbf{J}_M, \quad i = 0, 1.$$

Das Einsetzen von (4.3) und (4.4) in die PR-Bedingung der Polyphasenmatrizen (3.12) ergibt mit (4.5) die Gleichung

$$\begin{aligned} \frac{z^{-2D-1}}{2M} \mathbf{I} &= (-1)^D [z^{-1}\mathbf{Q}_1(z^2), \mathbf{Q}_0(z^2)] \begin{bmatrix} \mathbf{P}_0(z^2) \\ z^{-1}\mathbf{P}_1(z^2) \end{bmatrix} \\ &+ [z^{-1}\mathbf{Q}_1(z^2), \mathbf{Q}_0(z^2)] \begin{bmatrix} \mathbf{J}_M & \mathbf{0} \\ \mathbf{0} & -\mathbf{J}_M \end{bmatrix} \begin{bmatrix} \mathbf{P}_0(z^2) \\ z^{-1}\mathbf{P}_1(z^2) \end{bmatrix}. \end{aligned} \quad (4.6)$$

Der zweite Summand hierin verschwindet:

$$\begin{aligned} & [z^{-1}\mathbf{Q}_1(z^2), \mathbf{Q}_0(z^2)] \begin{bmatrix} \mathbf{J}_M & \mathbf{0} \\ \mathbf{0} & -\mathbf{J}_M \end{bmatrix} \begin{bmatrix} \mathbf{P}_0(z^2) \\ z^{-1}\mathbf{P}_1(z^2) \end{bmatrix} \\ &= z^{-1} [\mathbf{Q}_1(z^2) \mathbf{J}_M \mathbf{P}_0(z^2) - \mathbf{Q}_0(z^2) \mathbf{J}_M \mathbf{P}_1(z^2)] \\ &= z^{-1} [\mathbf{J}_M \mathbf{P}_1(z^2) \mathbf{J}_M^2 \mathbf{P}_0(z^2) - \mathbf{J}_M \mathbf{P}_0(z^2) \mathbf{J}_M^2 \mathbf{P}_1(z^2)] \\ &= z^{-1} \mathbf{J}_M [\mathbf{P}_1(z^2) \mathbf{P}_0(z^2) - \mathbf{P}_0(z^2) \mathbf{P}_1(z^2)] \\ &= \mathbf{0} \end{aligned}$$

Es folgt, dass (4.6), und damit die PR-Eigenschaft der Filterbank, äquivalent zu den Bedingungen

$$(-1)^D z^{-1} [P_l(-z^2)P_{2M-1-l}(-z^2) + P_{M+l}(-z^2)P_{M-1-l}(-z^2)] = \frac{z^{-2D-1}}{2^M},$$

$$l = 0, \dots, \frac{M}{2} - 1,$$

an den Prototypfilter ist. Hieraus erhält man mit $z^{-2D} = (-1)^{-D}(-z^2)^{-D}$ und nach Substitution von $-z^2$ durch z die Bedingungen (4.2). \square

4.2 Der Prototypentwurf als quadratisches Minimierungsproblem

Die Ausdrücke in Formel (4.2) sind Polynome vom Grad $2m - 2$, da die Polyphasenkomponenten $P_l(z)$, $l = 0, \dots, 2M - 1$, vom Grad $m - 1$ sind. Um die PR-Bedingungen zu prüfen sind alle Koeffizienten dieser Polynome nachzurechnen. Es muss dann gelten:

$$(p_l * p_{2M-1-l} + p_{M+l} * p_{M-1-l})(n) = \begin{cases} \frac{1}{2^M} & \text{falls } n = D \\ 0 & \text{sonst} \end{cases}, \quad (4.7)$$

$$l = 0, \dots, \frac{M}{2} - 1, \quad n = 0, \dots, 2m - 2.$$

Wobei die $(p_l * p_{2M-1-l} + p_{M+l} * p_{M-1-l})(n)$, elementar gerechnet, gleich

$$\begin{aligned} & \sum_{k=\max(0, n+1-m)}^{\min(m-1, n)} p(2Mk + 2M - 1 - l) p(2M(n - k) + l) \\ & + \sum_{k=\max(0, n+1-m)}^{\min(m-1, n)} p(2Mk + M - 1 - l) p(2M(n - k) + M + l) \end{aligned} \quad (4.8)$$

sind.

Ein zu entwerfender reellwertiger Tiefpassprototyp $P(z)$ mit Grenzfrequenz Ω_S soll neben der PR-Eigenschaft vor allem eine gute Dämpfung im Sperrbereich besitzen. Die beste Dämpfung kann z.B. als Minimierung des maximalen Wertes von $|P(\omega)|$ im Sperrbereich aufgefasst werden (*Tschebycheff-Approximation*). Hier soll $|P(\omega)|$ im quadratischen Mittel minimiert werden, also das Integral

$$E_s = \int_{\omega=\Omega_S}^{\pi} |P(\omega)|^2 d\omega, \quad (4.9)$$

das auch als *Sperrenergie* eines Filters bezeichnet wird [Kl].

$P(\omega) = \sum_{n=0}^{L_p-1} p(n)e^{-in\omega}$ kann mit

$$\mathbf{p} = \begin{bmatrix} p(0) \\ p(1) \\ \vdots \\ p(L_p - 1) \end{bmatrix}, \quad \mathbf{c}(\omega) = \begin{bmatrix} 1 \\ e^{-i\omega} \\ \vdots \\ e^{-i(L_p-1)\omega} \end{bmatrix},$$

geschrieben werden als

$$P(\omega) = \mathbf{p}^T \mathbf{c}(\omega).$$

Dann ist

$$|P(\omega)|^2 = \mathbf{p}^T \mathbf{c}(\omega) \overline{\mathbf{p}^T \mathbf{c}(\omega)} = \mathbf{p}^T \mathbf{c}(\omega) \mathbf{c}^H(\omega) \mathbf{p},$$

wobei die Matrix $\mathbf{c}(\omega) \mathbf{c}^H(\omega)$ durch ihren Realteil $Re\{\mathbf{c}(\omega) \mathbf{c}^H(\omega)\}$ (Realteil komponentenweise gebildet) ersetzt werden kann, da \mathbf{p} ein reeller Vektor ist.

Durch

$$\mathbf{S} = \int_{\omega=\Omega_s}^{\pi} Re\{\mathbf{c}(\omega) \mathbf{c}^H(\omega)\} d\omega \quad (4.10)$$

wird eine Matrix $\mathbf{S} \in \mathbb{R}^{L_p \times L_p}$ definiert, mit der die Sperrenergie E_s als quadratische Form ausgedrückt werden kann:

$$E_s = \int_{\omega=\Omega_s}^{\pi} \mathbf{p}^T Re\{\mathbf{c}(\omega) \mathbf{c}^H(\omega)\} \mathbf{p} d\omega = \mathbf{p}^T \mathbf{S} \mathbf{p}. \quad (4.11)$$

Die Matrix \mathbf{S} ist positiv definit, da der Ausdruck $\mathbf{p}^T \mathbf{S} \mathbf{p}$ in (4.11) ja gerade die für alle Filter p positive Sperrenergie (siehe (4.9)) darstellt. Die Einträge von \mathbf{S} lassen sich berechnen zu

$$[\mathbf{S}]_{i,j} = \begin{cases} \pi - \Omega_s & \text{falls } i = j \\ -\frac{\sin((i-j)\Omega_s)}{i-j} & \text{sonst} \end{cases}. \quad (4.12)$$

Die Gleichungen für die PR-Eigenschaft aus Formel (4.7) bzw. die linken Seiten von (4.8) lassen sich ebenfalls mit Hilfe quadratischer Formen schreiben:

$$\mathbf{p}^T \mathbf{Q}_{l,n} \mathbf{p} = \begin{cases} \frac{1}{2M} & \text{falls } n = D \\ 0 & \text{sonst} \end{cases},$$

$$l = 0, \dots, \frac{M}{2} - 1, \quad n = 0, \dots, 2m - 2.$$

Offenbar sind die geeignet gewählten Matrizen $\mathbf{Q}_{l,n}$ schwach besetzt aber indefinit.

Insgesamt ist die Aufgabe, einen optimalen PR-Prototypen zu entwerfen,

durch die quadratische Minimierungsaufgabe

$$\begin{array}{ll}
 \min_{\mathbf{p}} & \mathbf{p}^T \mathbf{S} \mathbf{p}, & \text{(Zielfunktion)} \\
 \text{NB} & \mathbf{p}^T \mathbf{Q}_{l,n} \mathbf{p} = \frac{1}{2M} \delta_{n,D}, & \text{(Nebenbedingungen)} \\
 & l = 0, \dots, \frac{M}{2} - 1, \quad n = 0, \dots, 2m - 2
 \end{array} \tag{4.13}$$

formuliert.

4.3 Entwurf reeller Prototypen

Die Aufgabe (4.13) ist ein quadratisches Minimierungsproblem mit quadratischen Nebenbedingungen (QCQP), die nicht konvex sind. Aufgaben dieses Typs sind im Allgemeinen *NP*-schwer, wovon ich auch bei der speziellen Struktur der Matrizen in Aufgabe (4.13) ausgehe. Es kann also nicht davon ausgegangen werden, dass eine optimale Lösung unseres Problems in polynomieller Zeit gefunden werden kann.

(4.13) besitzt die semidefinite Relaxierung (SDP):

$$\begin{array}{ll}
 \min_{\mathbf{X}} & \text{Spur}(\mathbf{S}\mathbf{X}), \\
 \text{NB} & \text{Spur}(\mathbf{Q}_{l,n}\mathbf{X}) = \frac{1}{2M} \delta_{n,D}, \\
 & l = 0, \dots, \frac{M}{2-1}, \quad n = 0, \dots, 2r - 2, \\
 & \mathbf{X} \text{ positiv semidefinit.}
 \end{array} \tag{4.14}$$

Dies ist ein konvexes Problem, dessen globales Minimum in polynomieller Zeit gefunden werden kann [Al]. Besitzt die Lösung \mathbf{X} von (4.14) die Form $\mathbf{X} = \mathbf{p}\mathbf{p}^T$, so ist \mathbf{p} globale Lösung von (4.13).

Für symmetrische Prototypen sind die Lösungen der SDP-Relaxierung in den Fällen $m = 1$ und $m = 2$ von der genannten Form.

Dabei handelt es sich gerade um die Fälle, für die in der Signalverarbeitung die Entwurfsverfahren der MLT- und ELT-Filter [Ma] bekannt sind. Die Resultate dieser Verfahren gleichen denen der semidefiniten Relaxierung. Damit ist implizit gezeigt, dass MLT- und ELT-Filter optimal sind.

Die Lösungen der Relaxierung sind für größere m jedoch weit von Matrizen der Form $\mathbf{X} = \mathbf{p}\mathbf{p}^T$ entfernt. Dann ist das Finden einer guten Näherung einer optimalen Lösung via (SDP)-Relaxierung schwierig.

Lokale Minima unseres Problems können mit Hilfe allgemeiner Gradientenverfahren für nicht-konvexe Programmierung (NLP) gefunden werden. Solche Verfahren nähern sich einem lokalen Minimum von einem gegebenen

Startpunkt aus. Zu den klassischen Verfahren dieser Art gehören die *Penalty- und Barriere-Methoden*. Heute zählt die sogenannte *sequentielle quadratische Programmierung* (SQP) zu den wichtigsten Verfahren zur Lösung allgemeiner NLP-Probleme. Beim SQP-Verfahren wird in jedem Schritt die Zielfunktion durch eine quadratische Approximation an der aktuellen Iterierten ersetzt. Die Karush-Kuhn-Tucker-Bedingung des Ersatzproblems stellt dann ein quadratisches Optimierungsproblem dar, dessen Lösung als neue Iterierte bzw. als neue Abstiegsrichtung genommen wird. Eine gute Einführung zu den genannten und anderen Verfahren ist in [GK] zu finden.

Ein spezielles Verfahren zum Lösen nicht-konvexer Probleme vom Typ (QCQP), dem ein signifikanter Vorteil im Vergleich zu den genannten Verfahren nachgewiesen werden könnte, konnte ich dagegen nicht ausfindig machen.

Die Anzahl der lokalen Minima unserer Aufgabe steigt mit deren Größe (Filterlänge). Bei größeren Problemen und zufälligem Startvektor wird die Wahrscheinlichkeit der Konvergenz gegen ein gutes lokales (ggf. globales) Minimum gering.

Das Lösen von (4.13) per SQP-Verfahren hat sich aber als erfolgreich erwiesen, wenn ein Fast-PR-Prototyp mit guter Dämpfung als Startvektor benutzt wird. Diese Vorgehensweise ist implementiert im C-Programm *cosbank*, dessen Quellcode im Anhang zu finden ist. Als NLP-Solver wurde die SQP-Software *CFSQP Version 2.5d* von *AEM-Design* benutzt. Unser Programm beschränkt sich auf den Entwurf symmetrischer Prototypen ($p(n) = p(L_p - 1 - n)$, $n = 0 \dots L_p - 1$). Die Polynome vom Grad $2m - 2$ in (4.2) sind dann ebenfalls symmetrisch. Sowohl die Dimension der Minimierungsaufgabe als auch die Anzahl der Nebenbedingungen werden dadurch halbiert. Der benötigte Fast-PR-Prototyp wird zuvor nach einem Entwurfsverfahren von Xu, Lu und Antoniou [XLA] erzeugt. Das „Xu-Verfahren“ minimiert eine Summe, die aus der Sperrenergie des Prototypen und einem gewichteten Term besteht, der die durch Verzerrung von $T(z)$ (siehe Abschnitt 3.2) verursachte Verletzung der PR-Eigenschaft darstellt. Die verbleibende Verletzung der PR-Eigenschaft ist von der Größe der Sperrenergie abhängig. Tabelle 4.1 enthält Kennzahlen von Prototypfiltern, die mit dem Programm *cosbank* und dem Xu-Verfahren entworfen wurden. In Abbildung 4.3 ist die Sperrbereichsverstärkung der Prototypen als Funktion der Länge ihrer Polyphasenkomponenten für verschiedene Teilbandanzahlen dargestellt. Abbildungen 4.4 bis 4.6 zeigen die Frequenzgänge einiger dieser PR-Prototypen.

M	m	L_p	Verstärkung (dB)		mcv(Xu)
			PR-Filter	Xu-Filter	
8	3	48	-46.9	-55.3	3.06e-5
8	4	64	-52.3	-58.5	5.50e-6
8	5	80	-61.2	-67.5	3.60e-6
8	6	96	-68.0	-82.1	1.47e-6
8	7	112	-72.6	-89.1	2.82e-7
16	3	96	-49.9	-58.4	9.41e-6
16	4	128	-55.5	-61.6	2.06e-6
16	5	160	-64.5	-70.5	1.21e-6
16	6	192	-71.0	-84.5	4.75e-7
16	7	224	-75.5	-92.3	8.71e-8
32	3	192	-52.9	-61.3	2.70e-6
32	4	256	-58.6	-64.6	7.00e-7
32	5	320	-67.5	-73.4	3.62e-7
32	6	384	-74.2	-86.5	1.42e-7
32	7	448	-79.7	-95.4	2.31e-8

Tabelle 4.1: Einige cosinus-modulierte Filterbänke mit reellwertigen Prototypen. Verglichen wird die Sperrbereichsverstärkung (quadratisch gemittelt) der Prototypen, die mit dem Programm *cosbank* (PR-Filter) und dem „Xu-Verfahren“ (Xu-Filter) entworfen wurden. Zu letzterem ist auch die maximale Verletzung (mcv(Xu)) der PR-Bedingungen (4.2) angegeben.

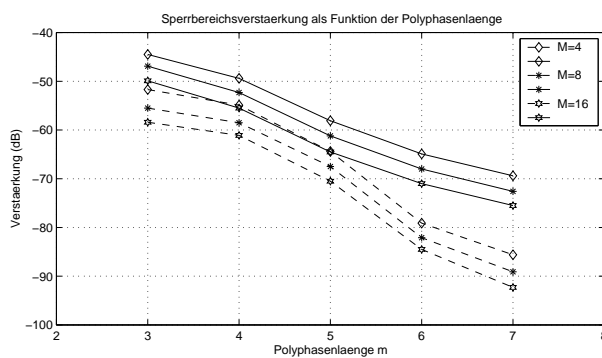


Abbildung 4.3: Darstellung der Sperrbereichsverstärkung (quadratisch gemittelt) der Prototypfilter aus Tabelle 4.1 in Abhängigkeit der Länge ihrer Polyphasenkomponenten (Xu-Filter gestrichelt).

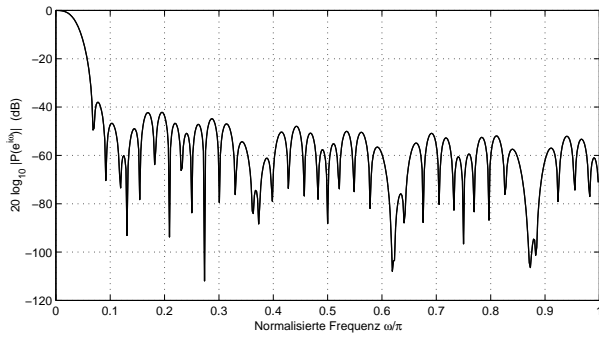


Abbildung 4.4: Prototypfilter $M = 16$, $m = 3$, $L_p = 96$

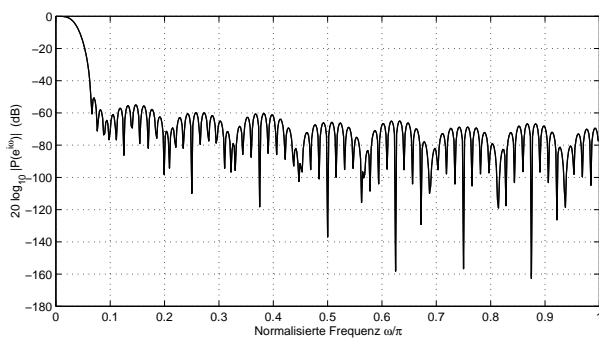


Abbildung 4.5: Prototypfilter $M = 16$, $m = 5$, $L_p = 160$

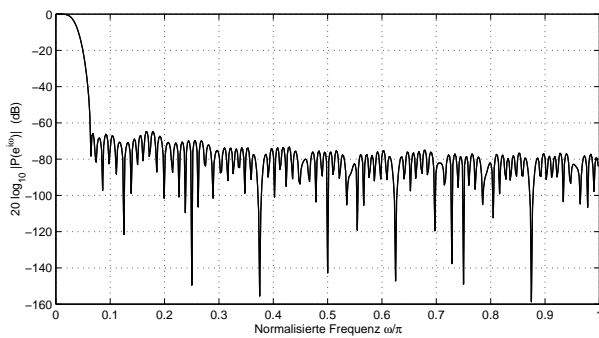


Abbildung 4.6: Prototypfilter $M = 16$, $m = 7$, $L_p = 224$

Kapitel 5

Diskretisierung von Prototypfiltern

Im letzten Kapitel wurde der Entwurf reeller Prototypfilter für cosinusmodulierte Filterbänke diskutiert. Bei einer Filterbank, die mit digitaler Technik realisiert wird, sind aber nicht nur die Signale digital, sondern natürlich auch die Koeffizienten des Prototypen selbst. Zwar sind die konkreten Ergebnisse eines Computerprogramms wie *cosbank* Maschinenzahlen, die eine binäre Darstellung endlicher Länge besitzen. Im Interesse einer effizienten Implementierung der Filterbank sollen die Koeffizienten $p(n)$, $n = 0, \dots, L_p - 1$ aber eine binäre Darstellung besitzen, die kürzer ist und aus möglichst wenigen von Null verschiedenen Bits besteht:

$$p(n) = \sum_{i=a_n}^{b_n} q_{n,i} 2^i, \quad a_n, b_n \in \mathbb{Z}, \quad q_{n,i} \in \{0, 1\},$$

$$n = 0, \dots, L_p - 1, \quad i = a_n, \dots, b_n, \quad \sum_{n=0}^{L_p} \sum_{i=a_n}^{b_n} |q_{n,i}| \quad \text{klein.}$$

Der Rechenaufwand, der benötigt wird, um einen Filterkoeffizienten mit einer zweiten binären Zahl (Eingangssignal) zu multiplizieren, ist nämlich zur Anzahl seiner Nicht-Null-Bits proportional (# Additionen).

Es bleibt anzumerken, dass die Koeffizienten der durch die Cosinus-Modulation entstehenden Teilbandfilter im obigen Sinne schlecht (nicht mit wenigen Nicht-Null-Bits darstellbar) sind, selbst wenn der Prototyp gut ist. Aber diese werden nicht direkt mit den Signalen multipliziert, sondern die Teilbandsignale entstehen, indem zunächst die Polyphasenkomponenten des Prototypen und anschließend ein Modulationsblock durchlaufen wird. In [F1] wurde gezeigt, dass der Hauptrechenaufwand durch die Filteroperation des Prototypen gegeben ist, wenn die Modulation mit Hilfe einer sogenannten schnellen *diskreten Cosinus-Transformation (DCT)* realisiert wird. Um

weiteren Aufwand bei der Modulation zu sparen, wurden in [MKK] zwei Möglichkeiten angegeben, die in Satz 4.1 definierten Modulationsmatrizen \mathbf{T}_1 und \mathbf{T}_2 durch solche mit ganzzahligen Koeffizienten zu ersetzen. Die Aussage des Satzes über den Prototypfilter bleibt nämlich gültig, solange nur die Gleichung (4.5) erfüllt ist. Die Vorschläge in [MKK] für den Entwurf solcher Matrizen beruhen auf einer Householder-Faktorisierung bzw. auf der Verwendung von Projektions-Matrizen. Allerdings sind die Teilbandfilter dann nicht mehr die genau gleichförmig frequenzverschobenen Versionen des Prototypen, wie es bei der Cosinus-Modulation der Fall ist.

5.1 Canonical Signed Digits

Anstelle „normaler“ Binärzahlen werden zur Realisierung digitaler Filter- und Filterbänke gern Zahlen in einer Darstellung durch sogenannte *Canonical Signed Digits* (kurz *CSD*-Darstellung) benutzt.

Eine reelle Zahl r sei durch

$$r \approx \sum_{i=a}^b q_i 2^i, \quad a \leq b \in \mathbb{Z}, \quad q_i \in \{-1, 0, 1\}, \quad i = a, \dots, b.$$

approximiert. Das Tupel $[q_i]_{i=a}^b$ heißt dann *CSD*-Darstellung von r . Insbesondere ist eine normale binäre Darstellung einer Zahl auch eine *CSD*-Darstellung. Da die Komponenten einer *CSD*-Zahl auch den Wert -1 annehmen können ist diese Art der Darstellung redundant. Zum Beispiel sind $[0, 1, 1, 1]$ und $[1, 0, 0, -1]$ zwei Darstellungen der Zahl 7. Das Beispiel zeigt, dass die Zahl der Nicht-Null-Bits im Mittel kleiner gehalten werden kann, als bei reinen Binärzahlen. Wie sich leicht numerisch ermitteln lässt, beträgt die mittlere Einsparung unter allen mit einer festen Wortlänge darstellbaren Zahlen etwa ein Viertel der Nicht-Null-Bits. Dies ist ein Vorteil, da der Rechenaufwand einer Multiplikation mit einer *CSD*-Zahl, genauso wie bei reinen Binärzahlen, zur Anzahl ihrer Nicht-Null-Bits proportional ist.

Im Folgenden wird die nächste Näherung einer reellen Zahl r durch eine *CSD*-Zahl mit N Bits und k Nicht-Null-Bits mit

$$CSD_{N,k}(r)$$

bezeichnet.

5.2 Direkte Diskretisierung der Filterkoeffizienten

Wenn man die Koeffizienten eines gemäß Abschnitt 4.3 entworfenen Prototypfilters $p \in \mathbb{R}^{L_p}$ diskretisiert, führt dies, neben einer Erhöhung der Sperrenergie, zu einer Verletzung der PR-Eigenschaft. Auf diese Weise sind also

nur Fast-PR-Filterbänke realisierbar. Es genügt deshalb, einen reellen Fast-PR-Prototypen mit entsprechend besserer Dämpfung zur Diskretisierung zu benutzen. Die Bedingung, die wir dabei stellen wollen, ist, dass die Verletzungen der PR-Bedingungen (4.2) in der Größenordnung der reellen Lösung bleiben. Dabei zeigt sich, dass die Sperrenergie des Filters ebenfalls in ihrer ursprünglichen Größenordnung bleibt, da sich die Koeffizienten nicht zu stark ändern. Wir können die Sperrenergie deshalb beim Quantisieren außer Acht lassen.

Da die PR-Bedingungen auf disjunkten Teilmengen der Koeffizienten von p formuliert sind, ist der Suchraum nicht so groß, wie er bei Einbeziehung der Zielfunktion (Sperrenergie) wäre. Seien $N, K \in \mathbb{N}$. Beschränkt man sich bei der Diskretisierung von p auf die Zahlen $CSD_{N,k}(p(n))$, $k \leq K$, so ist die Größe des Suchraums gleich $\frac{M}{2}K^{4r}$, bei symmetrischem Prototypfilter sogar nur $\frac{M}{2}K^{2r}$, denn die $\frac{M}{2}$ PR-Bedingungen beziehen sich auf je $4r$ bzw. $2r$ Koeffizienten von p .

Tabelle 5.1 in Abschnitt 5.5 enthält Kennzahlen einiger reeller Fast-PR-Prototypfilter, die nach dem Verfahren von Xu, Lu und Antoniou [XLA] entworfen wurden, und Kennzahlen zugehöriger diskreter Versionen, die auf die oben beschriebene Weise entstanden sind.

Ich möchte noch anmerken, dass die PR-Bedingungen (4.2) zwar prinzipiell unabhängig von der Sperrenergie des Prototypen sind. Bei den verwendeten Fast-PR-Prototypen nach Xu ist die Verletzung der PR-Eigenschaft der Filterbank, die sich als Summe ihrer linearen Verzerrung (Verzerrung von $T(z) = A_0(z)$, siehe Abschnitt 3.2) und der Aliasverzerrungen (verursacht durch die anderen $A_l(z)$) ergibt [Kl], aber auch stark von der Energie im Sperrbereich des Prototypen abhängig. Dies ist ein theoretisches Argument dafür, dass die Sperrbereichsdämpfung gut bleibt, wenn die Verletzungen der PR-Bedingungen beim Quantisieren nicht wesentlich verstärkt werden.

5.3 Die Liftingfaktorisierung

Im vorigen Abschnitt wurde darauf hingewiesen, dass ein reeller, perfekt rekonstruierender Prototypfilter $p \in \mathbb{R}^{L_p}$ nicht direkt diskretisiert werden kann, ohne dass die PR-Eigenschaft verloren geht. Man kann einen solchen Prototypen jedoch in einer Weise implementieren, bei der die PR-Eigenschaft gegen Diskretisierung „immun“ ist. Die Implementation beruht auf einer Faktorisierung des Prototypen (seiner Polyphasenkomponenten) in sogenannte *Liftingkoeffizienten*. Die Faktorisierung kann nach verschiedenen Schemen stattfinden, die zu unterschiedlichen Liftingkoeffizienten führen. Bei festem Schema ist sie durch eine Bijektion

$$f : \mathbb{R}^{L_c} \mapsto \{p \in \mathbb{R}^{L_p} : p \text{ erfüllt (4.2)}\} \quad (5.1)$$

gegeben, wobei L_c die Anzahl der Liftingkoeffizienten ist. Allgemein wird *Lifting* in [DS] erklärt. Wir benutzen ein Schema aus [KM1][KM2], das nachstehend detailliert beschrieben wird.

Seien $H_0(z)$, $H_1(z)$, $F_0(z)$, $F_1(z)$ vier kausale FIR-Filter, die der Bedingung

$$[H_0(z), H_1(z)] \begin{bmatrix} F_0(z) \\ F_1(z) \end{bmatrix} = rz^{-D}, \quad D \in \mathbb{N}, \quad (5.2)$$

genügen. Dies können die vier Filter einer einfachen Zweikanal-Filterbank mit perfekter Rekonstruktion (Abbildung 5.1) sein (daher die Bezeichnungen der Filter). Die Bedingung wird aber auch durch je vier Polyphasenkompo-

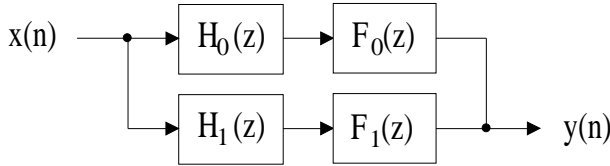


Abbildung 5.1: Eine einfache Zweikanal-Filterbank

nenten $P_l(z)$, $P_{2M-1-l}(z)$, P_{M+l} , P_{M-1-l} (siehe (4.2)) von Prototypfiltern cosinus-modulierter Filterbänke erfüllt, die die PR-Eigenschaft besitzen. Sind \mathbf{M} , \mathbf{M}' zwei (2×2) -Matrizen über dem Ring aller Laurentpolynome und $\mathbf{M}\mathbf{M}' = z^{-D_2}\mathbf{I}$, $D_2 \in \mathbb{N}_0$, so werden durch

$$[H_0^{neu}(z), H_1^{neu}(z)] = [H_0(z), H_1(z)]\mathbf{M}, \quad (5.3)$$

$$\begin{bmatrix} F_0^{neu}(z) \\ F_1^{neu}(z) \end{bmatrix} = \mathbf{M}' \begin{bmatrix} F_0(z) \\ F_1(z) \end{bmatrix}, \quad (5.4)$$

vier neue Filter $H_0^{neu}(z)$, $H_1^{neu}(z)$, $F_0^{neu}(z)$, $F_1^{neu}(z)$ definiert mit

$$[H_0^{neu}(z), H_1^{neu}(z)] \begin{bmatrix} F_0^{neu}(z) \\ F_1^{neu}(z) \end{bmatrix} = rz^{-(D+D_2)}. \quad (5.5)$$

Der durch (5.3) und (5.4) beschriebene Vorgang heißt *Lifting* bzw. *duales Lifting*.

Zu zwei Zahlen $a, b \in \mathbb{R} \setminus \{0\}$ seien die Matrizen

$$\begin{aligned} \mathbf{A}_0 &= \begin{bmatrix} 1 & 0 \\ az^{-1} & 1 \end{bmatrix}, & \mathbf{A}'_0 &= \begin{bmatrix} 1 & 0 \\ -az^{-1} & 1 \end{bmatrix}, \\ \mathbf{A}_1 &= \begin{bmatrix} z^{-1} & 0 \\ az^{-1} & 1 \end{bmatrix}, & \mathbf{A}'_1 &= \begin{bmatrix} z^{-1} & 0 \\ -az^{-1} & 1 \end{bmatrix}, \\ \mathbf{B}_0 &= \begin{bmatrix} 1 & b \\ 0 & 1 \end{bmatrix}, & \mathbf{B}'_0 &= \begin{bmatrix} 1 & -b \\ 0 & 1 \end{bmatrix}, \end{aligned}$$

$$\mathbf{B}_1 = \begin{bmatrix} 1 & b \\ 0 & z^{-1} \end{bmatrix}, \quad \mathbf{B}'_1 = \begin{bmatrix} 1 & -b \\ 0 & z^{-1} \end{bmatrix},$$

definiert. Dann ist

$$\mathbf{A}_i \mathbf{A}'_i = \mathbf{B}_i \mathbf{B}'_i = z^{-i} \mathbf{I}, \quad i \in \{0, 1\}.$$

Mit

$$\mathbf{M} = \mathbf{A}_i \mathbf{B}_j, \quad \mathbf{M}' = \mathbf{B}'_j \mathbf{A}'_i, \quad i, j \in \{0, 1\}, \quad (5.6)$$

gilt also $\mathbf{M} \mathbf{M}' = z^{-(i+j)} \mathbf{I}$, d.h. für \mathbf{M}, \mathbf{M}' bekommt man gemäß (5.3) und (5.4) vier neue Filter die (5.5) mit $D_2 = i + j \in \{0, 1, 2\}$ erfüllen. Die Vorgänge (5.3) und (5.4) werden dann auch passend als (englisch) *zero-delay lifting* ($i=j=0$), *type-1 single-delay lifting* ($i=1, j=0$), *type-2 single-delay lifting* ($i=0, j=1$) oder *maximum-delay lifting* ($i=j=1$) bezeichnet. Die Zahlen a, b in den Matrizen sind die *Lifting-Koeffizienten*. Je nach Wahl von i und j bekommt man die neuen Filter:

$$i=0: \quad H_0^{neu}(z) = H_0(z) + az^{-1}H_1(z), \quad F_1^{neu}(z) = F_1(z) - az^{-1}F_0(z), \quad (5.7)$$

$$i=1: \quad H_0^{neu}(z) = z^{-1}H_0(z) + aH_1(z), \quad F_1^{neu}(z) = z^{-1}F_1(z) - aF_0(z), \quad (5.8)$$

$$j=0: \quad H_1^{neu}(z) = H_1(z) + bH_0^{neu}(z), \quad F_0^{neu}(z) = F_0(z) - bF_1^{neu}(z), \quad (5.9)$$

$$j=1: \quad H_1^{neu}(z) = z^{-1}H_1(z) + bH_0^{neu}(z), \quad F_0^{neu}(z) = z^{-1}F_0(z) - bF_1^{neu}(z). \quad (5.10)$$

Alle neuen Filter sind um genau einen Koeffizienten länger, als die ursprünglichen Filter.

Umgekehrt lassen sich zu Filtern $H_0^{neu}(z), H_1^{neu}(z), F_0^{neu}(z), F_1^{neu}(z)$ der Länge $m \geq 2$, die Gleichung (5.5) genügen, Liftingkoeffizienten a, b und verkürzte Filter $H_0(z), H_1(z), F_0(z), F_1(z)$ bestimmen, aus denen die gegebenen Filter gemäß obiger Gleichungen hervorgehen. (5.9) ist zum Beispiel äquivalent zu

$$H_1(z) = H_1^{neu}(z) - bH_0^{neu}(z), \quad F_0(z) = F_0^{neu}(z) + bF_1^{neu}(z).$$

Da $H_1(z)$ und $F_0(z)$ die Länge $m - 1$ haben sollen, müssen $h_1(m - 1)$ und $f_0(m - 1)$ verschwinden. Dazu muss b die zwei Gleichungen

$$b = \frac{h_1^{neu}(m - 1)}{h_0^{neu}(m - 1)}, \quad b = -\frac{f_0^{neu}(m - 1)}{f_1^{neu}(m - 1)} \quad (5.11)$$

erfüllen. Die Gleichheit der beiden Terme für b folgt aus (5.5) im Zeitbereich, $D + D_2 \neq 2m - 2$ vorausgesetzt:

$$\begin{aligned} 0 &= (h_0^{neu} * f_0^{neu} + h_1^{neu} * f_1^{neu})(2m - 2) \\ &= h_0^{neu}(m - 1)f_0^{neu}(m - 1) + h_1^{neu}(m - 1)f_1^{neu}(m - 1). \end{aligned} \quad (5.12)$$

Analog bekommt man Umkehrungen der Gleichungen (5.10), (5.7) und (5.8). Darüber hinaus lassen sich skalare Filter $H_0(z) = h_0$, $H_1(z) = h_1$, $F_0(z) = f_0$ und $F_1(z) = f_1$, die (5.2) erfüllen ($h_0 f_0 + h_1 f_1 = r$) als

$$[H_0(z), H_1(z)] = [1, 1] \sqrt{r/2} \mathbf{M}, \quad \begin{bmatrix} F_0(z) \\ F_1(z) \end{bmatrix} = \mathbf{M}^{-1} \sqrt{r/2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

schreiben, wobei man \mathbf{M} , \mathbf{M}^{-1} als

$$\mathbf{M} = \begin{bmatrix} 1 & 0 \\ a & 1 \end{bmatrix} \begin{bmatrix} 1 & b \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ c & 1 \end{bmatrix}, \quad \mathbf{M}^{-1} = \begin{bmatrix} 1 & 0 \\ -c & 1 \end{bmatrix} \begin{bmatrix} 1 & -b \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -a & 1 \end{bmatrix} \quad (5.13)$$

mit den eindeutig bestimmten Werten

$$b = \frac{h_1 - f_0}{2\sqrt{r/2}}, \quad a = \frac{\frac{1}{\sqrt{r/2}} h_1 - 1 - b}{b}, \quad c = \frac{\frac{1}{2\sqrt{r/2}} (h_0 - f_1) - a}{1 + ab}, \quad (5.14)$$

wählen kann.

Insgesamt können vier Filter der Länge m , die Gleichung (5.2) erfüllen, komplett durch

$$[H_0(z), H_1(z)] = [1, 1] \sqrt{r/2} \mathbf{M}_0 \cdot \prod_{k=1}^{k_0} \mathbf{M}_k \cdot \prod_{k=k_0+1}^{k_1} \mathbf{M}_k \cdot \prod_{k=k_1+1}^{m-1} \mathbf{M}_k,$$

$$\begin{bmatrix} F_0(z) \\ F_1(z) \end{bmatrix} = \prod_{k=m-1}^{k_1+1} \mathbf{M}'_k \cdot \prod_{k=k_1}^{k_0+1} \mathbf{M}'_k \cdot \prod_{k=k_0}^1 \mathbf{M}'_k \cdot \mathbf{M}_0^{-1} \cdot \sqrt{r/2} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad (5.15)$$

faktorisiert werden, wobei \mathbf{M}_0 , \mathbf{M}_0^{-1} wie in (5.13) und \mathbf{M}_k , \mathbf{M}'_k wie in (5.6) als $\mathbf{A}_{i_k} \mathbf{B}_{j_k}$, $\mathbf{B}'_{j_k} \mathbf{A}'_{i_k}$, mit

$$(i_k, j_k) = \begin{cases} (0, 0) & \text{für } 1 \leq k \leq k_0 \\ (1, 0) \vee (0, 1) & \text{für } k_0 < k \leq k_1 \\ (1, 1) & \text{sonst} \end{cases}$$

gewählt seien. Die Grenzen k_0, k_1 sind so zu wählen, dass $(k_1 - k_0) + 2(m - 1 - k_1)$ gleich der Verzögerung D in (5.2) ist.

Die Matrizen in (5.15) sind durch $2(m - 1) + 3$ Liftingkoeffizienten gegeben. Die Faktorisierung eines Prototypen $p \in \mathbb{R}^{L_p}$, $L_p = 2Mm$, liefert demnach $L_c = \frac{M}{2}(2m + 1)$ Liftingkoeffizienten, da Lifting wegen Formel (4.2) auf je vier der $2M$ Polyphasenkomponenten von p angewendet wird. Durch die Zuordnung der Liftingkoeffizienten zu p ist, entsprechend (5.1), eine Bijektion f erklärt.

Bemerkung (zur Numerik). Anhand Formel (5.11) ist die Notwendigkeit der PR-Eigenschaft eines Prototypen für die Faktorisierung seiner

Polyphasenkomponenten in Liftingkoeffizienten deutlich geworden. Ein mit einem numerischen Verfahren entworfener Prototyp erfüllt die PR-Bedingungen (4.2) jedoch nur näherungsweise. Zum Beispiel haben wir bei den mit *cosbank* entworfenen Filtern einen Fehler ε zwischen 10^{-18} und 10^{-15} . Auf der linken Seite von (5.12) hätte man dann diesen Fehler anstelle der Null. Die beiden Berechnungen des Liftingkoeffizienten b in (5.11) würden sich um den Betrag $\frac{\varepsilon}{h_0^{neu(m-1)} f_1^{neu(m-1)}}$ unterscheiden. Diese Differenz bleibt gering, wenn das Produkt im Nenner betragsmäßig deutlich größer als ε ist. Insbesondere hat man gute Aussichten auf eine erfolgreiche Liftingfaktorisation, falls der Betrag des kleinsten Filterkoeffizienten deutlich größer als $\sqrt{\varepsilon}$ ist, etwa 10^{-7} bei $\varepsilon = 10^{-16}$.

5.4 Diskretisierung der Liftingkoeffizienten

Wenn die Liftingkoeffizienten $c \in \mathbb{R}^{L_c}$ eines reellen Prototypen $p \in \mathbb{R}^{L_p}$ gegeben sind, können diese, ohne Einfluss auf die PR-Eigenschaft des Filters, diskretisiert werden.

Wir gewinnen die Liftingkoeffizienten, indem wir den zuvor erzeugten PR-Prototypen p komplett faktorisieren, also durch $c = f^{-1}(p)$ (f gemäß (5.1)). Alternativ könnte man die Liftingkoeffizienten auch direkt bestimmen. D.h. man könnte versuchen die Minimierungsaufgabe

$$\min_c f(c)^T \mathbf{S} f(c), \quad \mathbf{S} \text{ nach (4.2), } f \text{ nach (5.1),}$$

mit einem *NLP*-Verfahren zu lösen. Diese Vorgehensweise hat sich in der Praxis aber als wenig erfolgreich erwiesen. Das Problem hierbei ist, dass für die „hochgradig unlineare“ Zielfunktion kein vernünftiger Startpunkt für das Optimierungsverfahren angegeben werden kann. Es sei daran erinnert, dass wir zum Lösen der Aufgabe (4.13) (Entwurf des reellen Prototypen p) einen Fast-PR-Filter als Startpunkt verwendet haben.

Es sollen nun *CSD*-Näherungen $\tilde{c}(n)$ der Liftingkoeffizienten $c(n)$ bestimmt werden. Wichtig dabei ist, dass die Sperrenergie des zum diskreten Liftingvektor gehörenden Prototypen nicht allzu groß wird.

Für zwei Liftingvektoren $c1, c2 \in \mathbb{R}^{L_c}$ sei die Ratio der Sperrenergien der aus ihnen resultierenden Prototypen mit $R(c1, c2)$ bezeichnet:

$$R(c1, c2) = \frac{E_s(f(c1))}{E_s(f(c2))}$$

Die Aufgabe ist dann, sowohl die Summe der Nicht-Null-Bits in den Darstellungen der $\tilde{c}(n)$ als auch die Ratio $R(\tilde{c}, c)$ klein zu halten.

5.4.1 Algorithmus 1

Ein einfacher Ansatz ist [KM1], zunächst eine feine Diskretisierung von c vorzunehmen und dann iterativ das jeweils unempfindlichste ($R(\tilde{c}, c)$ am kleinsten haltende) Nicht-Null-Bit in den CSD-Darstellungen zu entfernen. Der Algorithmus wird abgebrochen, wenn die Summe der Nicht-Null-Bits klein genug ist oder $R(\tilde{c}, c)$ eine vorgegebene Ratio R übersteigt.

Das Verfahren hat, eine feste Wortlänge der Liftingkoeffizienten vorausgesetzt, einen Aufwand von $O(L_c^4)$. Denn es werden $O(L_c)$ Bits entfernt, und für jede Entfernung werden $O(L_c)$ Bits mit einem Aufwand von $O(L_c^2)$ (Berechnung der Sperrenergie) getestet.

5.4.2 Algorithmus 2

Bessere Ergebnisse, d.h. eine kleinere Bitzahl bei vorgegebener Ratio R , bekommt man, wenn man die Komponenten des Lifting-Vektors c iterativ diskretisiert, und zwischendurch die noch nicht diskretisierten Komponenten reoptimiert. Dabei soll jeweils diejenige Komponente gewählt werden, die unter der Bedingung $R(\tilde{c}, c) < R$ mit den wenigsten Nicht-Null-Bits quantisiert werden kann. Die Reoptimierung wird dann mit einem NLP-Verfahren durchgeführt (ggf. mit einer begrenzten Anzahl von Iterationen), wobei die schon diskretisierten Koeffizienten fixiert werden. Im nachstehenden Listing ist der Algorithmus durch Pseudo-Code beschrieben.

Dazu einige Erläuterungen:

In Schritt 1 wird der reelle Liftingvektor per Faktorisierung bestimmt. Es wird eine Wortlänge N für seine quantisierten Komponenten vorgegeben und eine obere Schranke R für die Güteratio aus reeller und diskreter Lösung.

In Schritt 2 initialisieren wir die diskrete Lösung \tilde{c} , indem wir sie mit c gleichsetzen. Die Indexmenge J wird verwendet, um alle Komponenten, die im Verfahren noch nicht endgültig diskretisiert (nicht fixiert) wurden, zu kennzeichnen. Sie wird also mit den Zahlen 1 bis L_c initialisiert.

Im dritten Schritt werden nun iterativ alle Komponenten von \tilde{c} quantisiert. Dabei wird in jeder Iteration wie folgt vorgegangen:

Unter allen noch nicht diskretisierten Liftingkomponenten wird jeweils nach derjenigen gesucht, die sich mit der geringsten Anzahl k von Nicht-Null-Bits diskretisieren lässt, ohne dass dadurch die Güteratio $R(\tilde{c}, c)$ die vorgegebene Schranke R übersteigt (Schritt 3.(b)). Um die Zahl k der benötigten Nicht-Null-Bits für eine Komponente zu ermitteln (Schritt 3.(b)ii.), zählen wir k einfach solange hoch (beginnend mit $k = 0$), bis das Ersetzen der Komponente durch ihre nächste Näherung mit N Bits und k Nicht-Null-Bits zu einer Güteratio unterhalb der gegebenen Schranke R führt. Sinnvollerweise wird dabei stets die Wirkung einer Reoptimierung in die Berechnung mit einbezogen (Schritt 3.(b)ii.D.). Können zwei Komponenten mit der gleichen Anzahl von Nicht-Null-Bits quantisiert werden, so wählen wir diejenige, bei

der die Quantisierung zur besseren Güteratio führt (in Schritt 3.(b)iii.).

In Schritt 3.(c) diskretisieren wir die ermittelte Liftingkomponente tatsächlich und fixieren diese für den Rest des Algorithmus (ihr Index wird dazu aus der Menge J entfernt).

Es folgt jeweils eine Reoptimierung aller noch freien Liftingkoeffizienten in Schritt 3.(d).

Listing: Algorithmus 2

1. Bestimme $c \in \mathbb{R}^{L_c}$ durch $c = f^{-1}(p)$, eine maximale Länge N für die CSD -Zahlen und eine maximal zulässige Energie-Ratio R .
2. Setze $\tilde{c} = c$ und $J = \{1, \dots, L_c\}$.
3. Wiederhole (a) bis (d), solange $J \neq \emptyset$ gilt.
 - (a) Setze $k_{best} = \infty$, $R_{best} = \infty$, und wähle $j_{best} \in J$.
 - (b) Wiederhole i. bis iii. für alle $j \in J$.
 - i. Setze $k = -1$ und $R1 = R$.
 - ii. Wiederhole A. bis E., bis $R1 < R$ gilt
 - A. Setze $k = k + 1$.
 - B. Setze $\tilde{\tilde{c}} = \tilde{c}$.
 - C. Setze $\tilde{\tilde{c}}(j) = CSD_{N,k}(\tilde{c}(j))$.
 - D. Optimiere $\tilde{\tilde{c}}$ bei Fixierung aller $\tilde{\tilde{c}}(i)$ mit $i \notin J \setminus \{j\}$.
 - E. Setze $R1 = R(\tilde{\tilde{c}}, c)$.
 - iii. Falls $k < k_{best}$ oder ($k = k_{best}$ und $R1 < R_{best}$) gilt, setze $j_{best} = j$, $k_{best} = k$ und $R_{best} = R1$.
 - (c) Setze $\tilde{c}(j_{best}) = CSD_{N,k_{best}}(\tilde{\tilde{c}}(j_{best}))$ und $J = J \setminus \{j_{best}\}$.
 - (d) Reoptimiere \tilde{c} bei Fixierung aller $\tilde{c}(j)$ mit $j \notin J$.
4. Gib \tilde{c} als Lösung aus.

Der beschriebene Algorithmus ist im Programm *quant.c* implementiert (Anhang). Das verwendete NLP-Verfahren ist wieder der SQP-Solver *CFSQP Version 2.5d*. Für die gegebene Zielfunktion ist der Rechenaufwand $O(L_c^2)$, womit das SQP-Verfahren bei vorgegebener maximaler Anzahl von Iterationen einen Aufwand von $O(L_c^3)$ hat. Die Laufzeit unseres Algorithmus ist dann von der Größenordnung $O(L_c^5)$. Bei Verzicht auf die Optimierung innerhalb der Suchschleife (3.(b)ii.D.) ist der Aufwand $O(L_c^4)$. Im ersten Fall braucht es auf einem aktuell handelsüblichen Prozessor etwa einen Tag, um einen Prototypen der Länge 250 (seine Liftingkoeffizienten) zu diskretisieren. Bei einem Prototypen der Länge 500 müsste man hingegen schon einen Monat warten. Durch Aufteilung der Berechnungen auf parallele Prozessoren

könnten aber immerhin Filterlängen von über 1000 erreicht werden.

5.5 Entwurfsbeispiele

In Abschnitt 5.2 haben wir eine einfache Methode beschrieben, mit der Prototypfilter direkt quantisiert werden können. Wir haben diese Methode auf einige Fast-PR-Prototypen angewendet, die wir zuvor mit dem „Xu-Verfahren“ entworfen hatten. Die Verletzung der einzelnen PR-Bedingungen (4.2) durfte dabei jeweils maximal um den Faktor 2 steigen. Den in Abschnitt 5.2 erwähnten Suchraum haben wir allerdings nicht total sondern zufällig abgesucht. Ergebnisse dieser Vorgehensweise sind in Tabelle 5.1 festgehalten. Beispielhaft sind in Abbildung 5.2 die Frequenzgänge eines Fast-PR-Filters und seiner diskreten Version zu sehen.

Als numerische Tests für die beiden Algorithmen des Abschnitts 5.4 haben wir Prototypfilter mit 4 bis 32 reellen (8 bis 64 komplexen) Teilbändern und unterschiedlich langen Polyphasenkomponenten quantisiert. Die reellwertigen Prototypen hierfür sind zuvor gemäß Abschnitt 4.3 (Programm *cosbank* im Anhang) erzeugt worden.

Als Abbruchkriterium für Algorithmus 1 wurde das Überschreiten der gleichen Energie-Ratio R gewählt, die wir auch als Schranke in Algorithmus 2 verwendet haben. Daraus folgt, dass die quadratisch gemittelte Dämpfung der diskretisierten Prototypen bei beiden Verfahren in etwa gleich ist. Die Ratio R wurde im Übrigen so gewählt, dass die Dämpfung in Dezibel, verglichen mit den reellen Filtern, um maximal 5% schwächer werden sollte. Sie schwankt deshalb zwischen 1,7 bei geringerer Dämpfung und 2,2 bei stärkerer Dämpfung. Das Reoptimieren in Algorithmus 2 wurde stets mit 20 Schritten des verwendeten *SQP*-Verfahrens durchgeführt.

In Tabelle 5.2 sind Kennzahlen zu den Ergebnissen unserer Testläufe aufgeführt. Aus dem oben genannten Grund wird die Verstärkung der Prototypfilter für beide Algorithmen gemeinsam angegeben.

Es zeigt sich, dass die mittlere Anzahl der Nicht-Null-Bits pro Filterkoeffizient bei Algorithmus 2 um durchschnittlich 15% unterhalb derer von Algorithmus 1 liegt. Unser Verfahren ist, wie schon erwähnt, wesentlich zeitaufwendiger als Algorithmus 1, was aber für den einmaligen Entwurf eines guten Prototypfilters bis zu einer gewissen Gesamtlaufzeit (sagen wir einen Monat) keine Rolle spielen sollte. Dies erlaubt es die schon erwähnten Filterlängen von 500 (ein Prozessor) bis über 1000 (bei Verwendung paralleler Prozessoren) mit Algorithmus 2 anzugehen.

Für die beiden Verfahren sind in Abbildungen 5.3 und 5.4 die Summe der Nicht-Null-Bits und die Sperrbereichsverstärkung von Prototypfiltern in Abhängigkeit der Polyphasenlänge graphisch dargestellt.

M	m	L_p	reeller Prototyp		diskreter Prototyp	
			dB	MCV	dB	Bits
8	4	64	-58.5	5.50e-6	-57.8	3.06
8	6	96	-82.1	1.47e-6	-80.9	4.79
8	8	128	-98.7	3.15e-8	-98.5	6.03
16	3	96	-58.4	9.41e-6	-57.3	3.12
16	5	160	-70.5	1.21e-6	-69.4	3.88
16	7	224	-92.3	8.71e-8	-92.0	4.84

Tabelle 5.1: Reelle Fast-PR-Prototypen (nach Xu, Lu, Antoniou) und ihre diskreten Versionen nach dem Verfahren aus Abschnitt 5.2. Angegeben sind jeweils die Parameter M (Kanäle der Filterbank), m (Länge der Polyphasenkomponenten) und L_p (Filterlänge), sowie die Sperrbereichsverstärkung im quadratischen Mittel. MCV ist die maximale Abweichung von den PR-Bedingungen (4.2) beim reellen Prototypen und Bits die durchschnittliche Anzahl der Nicht-Null-Bits pro diskreten Filterkoeffizienten.

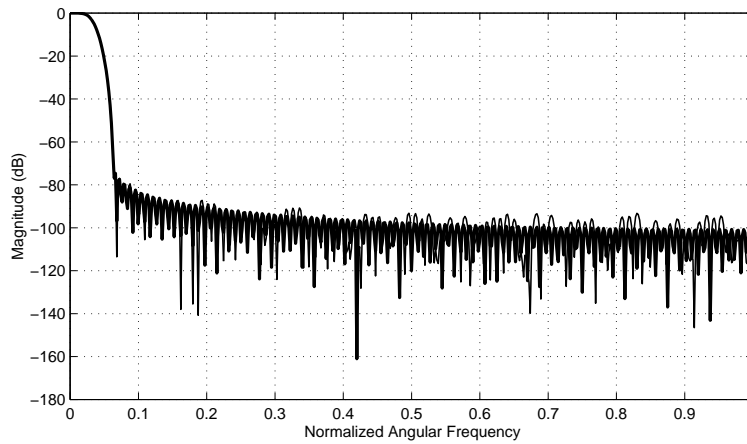


Abbildung 5.2: Betragsfrequenzgänge der beiden Prototypfilter aus der letzten Zeile von Tabelle 5.1 ($M=16$, $m=7$). Die schmale Linie gehört zur diskreten Version.

M	m	L _p	dB	Bits/Filterkoeff.			Zeiten	
				Alg.1	Alg.2	Relation	Alg.1	Alg.2
4	3	24	-43.4	1.08	0.96	89%	0.05 Sek.	3 Sek.
4	4	32	-46.7	1.44	1.19	83%	0.1 Sek.	9 Sek.
4	5	40	-55.4	1.55	1.20	77%	0.3 Sek.	25 Sek.
4	6	48	-61.9	1.83	1.58	86%	0.9 Sek.	66 Sek.
4	7	56	-66.2	1.77	1.57	89%	1.3 Sek.	2 Min.
6	3	36	-43.6	1.14	1.03	90%	0.15 Sek.	13 Sek.
6	4	48	-48.7	1.58	1.33	84%	0.5 Sek.	1 Min.
6	5	60	-57.3	1.53	1.32	86%	1.1 Sek.	2 Min.
6	6	72	-63.5	1.86	1.63	88%	2.9 Sek.	6 Min.
6	7	84	-68.0	1.68	1.46	87%	4.7 Sek.	11 Min.
8	3	48	-44.7	1.21	1.06	88%	0.2 Sek.	44 Sek.
8	4	64	-50.0	1.56	1.14	73%	3.3 Sek.	3 Min.
8	5	80	-58.4	1.53	1.24	81%	2.5 Sek.	7 Min.
8	6	96	-64.8	1.94	1.50	77%	7.2 Sek.	20 Min.
8	7	112	-69.2	1.83	1.57	86%	10.5 Sek.	37 Min.
12	3	72	-46.4	1.08	1.01	94%	0.8 Sek.	4 Min.
12	4	96	-49.9	1.41	1.20	85%	4.2 Sek.	17 Min.
12	5	120	-58.9	1.41	1.19	84%	7.6 Sek.	40 Min.
12	6	144	-65.8	1.85	1.48	80%	20 Sek.	2 Std.
12	7	168	-70.5	1.80	1.52	84%	32 Sek.	4 Std.
16	3	96	-47.6	1.14	1.06	93%	3 Sek.	19 Min.
16	4	128	-52.9	1.48	1.20	81%	10 Sek.	70 Min.
16	5	160	-61.5	1.51	1.27	84%	23 Sek.	3 Std.
16	6	192	-67.8	1.78	1.57	88%	80 Sek.	7 Std.
16	7	224	-71.8	1.78	1.52	85%	2 Min.	15 Std.
24	3	144	-49.4	1.12	1.02	91%	12 Sek.	83 Min.
24	4	192	-54.8	1.39	1.22	88%	41 Sek.	6 Std.
24	5	240	-63.1	1.40	1.28	91%	100 Sek.	16 Std.
24	6	288	-68.9	1.80	1.56	87%	6 Min.	48 Std.
32	3	192	-50.7	1.14	1.01	89%	31 Sek.	7 Std.
32	4	256	-56.1	1.40	1.17	84%	2 Min.	26 Std.

Tabelle 5.2: Vergleich einiger cosinus-modulierte Filterbänke mit perfekter Rekonstruktion. Angegeben sind die Parameter M (Kanäle der Filterbank), m (Länge der Polyphasenkomponenten) und L_p (Filterlänge), sowie die Sperrbereichsverstärkung im quadratischen Mittel in dB, die bei beiden Algorithmen etwa gleich ist. Zu beiden Algorithmen sind die mittlere Anzahl der Nicht-Null-Bits pro Filterkoeffizient sowie deren Verhältnis zueinander und die benötigte Rechenzeit angegeben (Sun Ultra 2, 450 MHz).

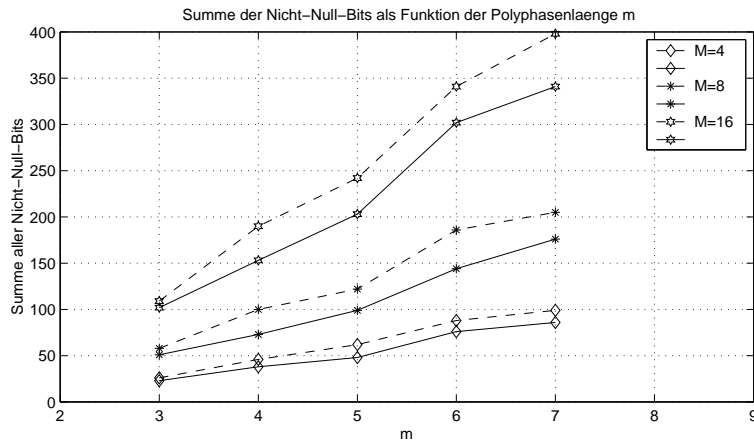


Abbildung 5.3: Darstellung der Summe aller Nicht-Null-Bits in Abhängigkeit der Polyphasenlänge m für Prototypen aus beiden Algorithmen (Algorithmus 1 gestrichelt).

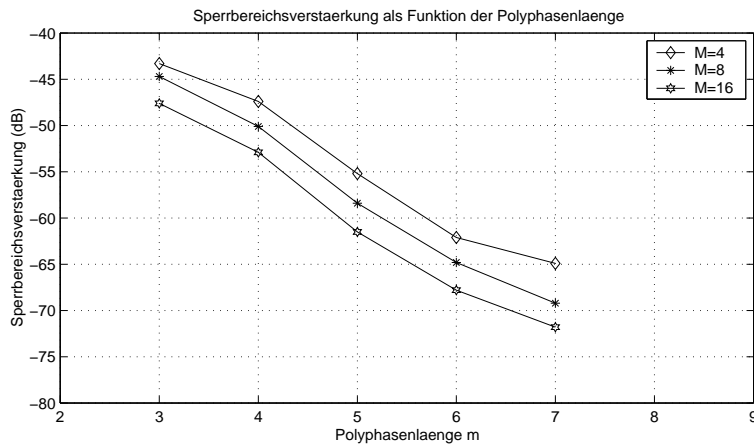


Abbildung 5.4: Darstellung der Sperrbereichsverstärkung (quadratisch gemittelt) in Abhängigkeit der Polyphasenlänge m für Prototypen aus beiden Algorithmen.

Man erkennt in Abbildung 5.3, dass die mit Algorithmus 2 erzeugten Prototypen mit Polyphasenkomponenten der Länge 5 bzw. 7 ungefähr die gleiche Anzahl von Nicht-Null-Bits haben, wie jene mit Polyphasenkomponenten der Länge 4 bzw. 6, die mit Algorithmus 1 entworfen wurden. Erstere haben also etwa den gleichen Aufwand beim Filtern. Ihre Dämpfung, die man an Abbildung 5.4 ablesen kann, ist aber um einiges höher.

Abschließend zeigen wir noch einige Frequenzgangplots von Filtern, die mit Algorithmus 1 und Algorithmus 2 quantisiert wurden.

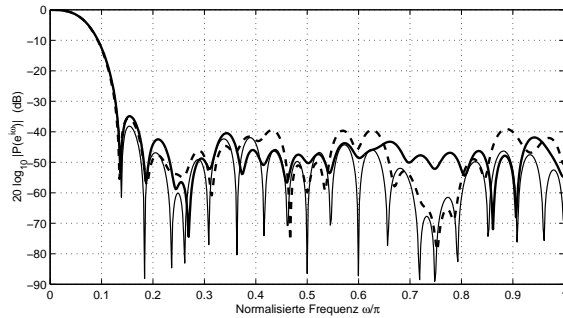


Abbildung 5.5: Betragsspektren von PR-Prototypen einer cosinus-modulierten 8-Kanal-Filterbank der Länge 48 (aus Tabelle 5.2). Die dünne Linie gehört zum reellwertigen Prototypen, die dicken Linien zu den diskreten Versionen (Algorithmus 1 gestrichelt, Algorithmus 2 durchgezogen). Die diskreten Versionen haben im Mittel 1.21 bzw. 1.06 Nicht-Null-Bits pro Filterkoeffizient.

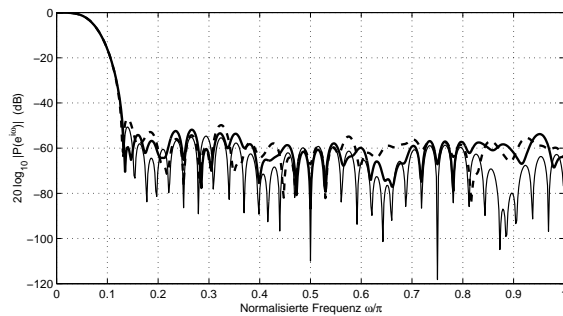


Abbildung 5.6: Betragsspektren von PR-Prototypen einer cosinus-modulierten 8-Kanal-Filterbank der Länge 80. (aus Tabelle 5.2). Die diskreten Versionen haben im Mittel 1.53 bzw. 1.24 Nicht-Null-Bits pro Filterkoeffizient.

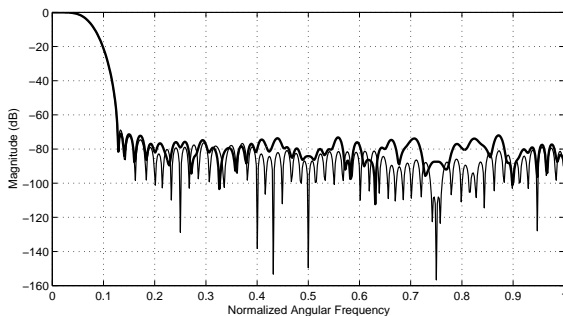


Abbildung 5.7: Betragsspektren von PR-Prototypen einer cosinus-modulierten 8-Kanal-Filterbank der Länge 128 (nicht in Tabelle 5.2 enthalten). Die diskrete Version nach Algorithmus 2 (dicke Linie) hat im Mittel 1.71 Nicht-Null-Bits pro Filterkoeffizient.

Kapitel 6

Zusammenfassung

Das Hauptziel dieser Arbeit war der Entwurf von Prototypfiltern für cosinus-modulierte Filterbänke, die perfekte oder fast perfekte Rekonstruktion des Eingangssignals mit einer effizienten Realisierung der Filterbank verbinden. Dabei ist mit Effizienz der Rechenaufwand und damit der Zeitaufwand gemeint, den die Filterbank benötigt, um ein gegebenes digitales Signal in Teilbänder zu zerlegen und zu rekonstruieren. Von Seiten des Prototypfilters ist dieser Aufwand durch die Wortlänge und die Anzahl der von Null verschiedenen Bits in den binären Darstellungen der Filterkoeffizienten bestimmt (der Aufwand ist zu beiden Größen proportional). Der zusätzliche Aufwand, der durch die Cosinus-Modulation entsteht ist geringer [Fl]. Es bestehen aber auch Ansätze, bei denen die Cosinus-Modulation durch eine effizienter realisierbare ganzzahlige Modulation ersetzt wird [MKK].

Reellwertige Lösungen des Problems haben wir im Falle von Fast-PR-Prototypen mit dem bekannten Verfahren von Xu, Lu und Antoniou [XLA] erzeugt, den Entwurf reeller PR-Prototypen sind wir als quadratisches Programm mit quadratischen Nebenbedingungen angegangen (Abschnitt 4.3).

Um diskrete Filterrealisierungen zu erhalten wurde hier stets so vorgegangen, dass Lösungen des kontinuierlichen Problems im Nachhinein quantisiert wurden. Aus Sicht der mathematischen Programmierung gestaltet sich der direkte Entwurf diskreter Lösungen denn auch als schwierig. Es sei erwähnt, dass sich bei einem Kollegen von mir, der mit dem Entwurf von Tiefpassfiltern als solchen (nicht als Prototypen von Filterbänken) betraut war, die Verwendung von ganzzahliger linearer Programmierung für längere Filter als inpraktikabel erwiesen hat. Allerdings wurde von Alfred Mertins [Me] ein iteratives Entwurfsverfahren für symmetrische (linearphasige) Prototypen vorgestellt, das für reelle und ganzzahlige Lösungen gleichermaßen geeignet ist.

Die von uns verwendete nachträgliche Quantisierung führt, auf die Filterkoeffizienten direkt angewendet, zu einer Verletzung der PR-Eigenschaft der Prototypen. Eine einfache und effizient realisierbare Methode (Abschnitt 5.2) haben wir deshalb bevorzugt auf Fast-PR-Filter angewendet, da diese eine bessere Dämpfung haben als jene mit perfekter Rekonstruktion.

Um auch diskrete Lösungen zu verwirklichen, die echt PR sind, wurde auf die sogenannte Liftingfaktorisierung zurückgegriffen. Diese gestattet es, PR-Prototypen in einer Weise zu implementieren, bei der die PR-Eigenschaft gegen jede Änderung der Koeffizienten (jetzt Liftingkoeffizienten) immun ist. Die reellen Liftingkoeffizienten können entweder direkt per *NLP*-Verfahren gewonnen werden, oder durch komplette Faktorisierung eines zuvor entworfenen PR-Prototypen. Hier zeigt sich die zweite Vorgehensweise als geeigneter. Wie wir gezeigt haben, muss der Prototyp die quadratischen Nebenbedingungen des quadratischen Programms aber mit hoher Genauigkeit erfüllen, um eine Faktorisierung numerisch möglich zu machen.

In [KM1] wurde ein einfaches Verfahren vorgestellt, die Liftingkoeffizienten eines Prototypfilters mit einer geringen Anzahl von Nicht-Null-Bits zu diskretisieren. Diesem haben wir ein neues, iteratives Verfahren gegenübergestellt, dessen wesentliche Eigenschaft eine in jedem Schritt stattfindende Reoptimierung noch nicht diskretisierter Liftingkomponenten ist. Die Zahl der Nicht-Null-Bits konnte hiermit im Mittel um 15% gesenkt werden. Allerdings ist der Zeitaufwand unserer Methode erheblich größer. Für Filterlängen von bis zu 500 (bei Verwendung paralleler Prozessoren auch bis zu über 1000) ist die Methode aber praktikabel.

In dieser Arbeit haben wir uns auf den Entwurf von cosinus-modulierten Filterbänken mit symmetrischen Prototypfiltern beschränkt. Es sei aber darauf hingewiesen, dass sich für *DFT*-modulierte Filterbänke bei entsprechender Taktreduktion die gleichen PR-Bedingungen an den Prototypfilter ergeben. Eine schöne Formulierung für modulierte Filterbänke, die für unterschiedliche Systemverzögerungen und auch unterschiedliche Taktreduktionsverhältnisse gilt, findet man in [K1]. Die Formulierung ähnelt (4.2) im Kern und lässt sich analog als quadratisches Optimierungsproblem formulieren.

Die hier erzielten Ergebnisse, die die diskreten Lösungen betreffen, lassen sich somit allgemein auf modulierte Filterbänke übertragen.

Der von uns vorgestellte Algorithmus 2 in Abschnitt 5.4 ist im Übrigen generell zur Diskretisierung kontinuierlicher Lösungen von *NLP*-Aufgaben ohne Nebenbedingungen geeignet.

Anhang A

Die Module *linalg*, *filter* und *lifting*

Für die C-Programme *cosbank* und *quant*, die dem Entwurf reellwertiger Prototypfilter gemäß Abschnitt 4.3 und deren nachträglicher Diskretisierung gemäß Abschnitt 5.4 dienen, habe ich drei separate Hilfsmodule programmiert. Diese sind hier als erstes abgedruckt. Die eigentlichen Programme folgen in Anhang B (*cosbank*) und Anhang C (*quant*).

Die Hilfsmodule bestehen aus den Header -und Quellcode-Dateien *linalg.h/c*, *filter.h/c* und *lifting.h/c* (A.1 bis A.6).

In *linalg.h/c* (A.1 und A.2) sind einige Funktionen für Matrix -und Vektorrechnung implementiert. Diese werden als elementare Operationen für das Rechnen mit Filtern benötigt.

Die Dateien *filter.h/c* (A.3 und A.4) enthalten das „Xu-Verfahren“ zum Entwurf linearphasiger Fast-PR-Prototypfilter (siehe Abschnitt 4.3), sowie Funktionen zur Berechnung der Sperrenergie von Filtern und zur Berechnung der Verletzung von PR-Bedingungen (4.2) durch Prototypen cosinusmodulierter Filterbänke. Ferner sind zwei Funktionen enthalten, mit denen Polyphasenkomponenten von Filtern ausgelesen bzw. geändert werden können. Diese werden für die Liftingfaktorisierung (siehe Abschnitt 5.3) benötigt.

Die Funktionen zur Liftingfaktorisierung von Prototypfiltern sind in den Dateien *lifting.h/c* (A.5 und A.6) implementiert.

A.1 *linalg.h*

```
// Deklarationen von Funktionen fuer Vektor -und Matrixrechnung
void matout(double *A, int n1, int n2);
// gibt eine (n1 x n2)-Matrix A aus
void plus(double *A, double *B, double *C, int n1, int n2);
```

```

// C=A+B, A,B,C (n1 x n2)-Matrizen

void minus(double *A, double *B, double *C, int n1, int n2);
// C=A-B, A,B,C (n1 x n2)-Matrizen

void mult(double r, double *A, double *B, int n1, int n2);
// B=r*A, r skalar, A,B (n1 x n2)-Matrizen

void matmult(double *A, double *B, double *C, int n1, int n2, int n3);
// C=AB, A (n1 x n2)-Matrix, B (n2 x n3)-Matrix

void transpose(double *A, double *B, int n1, int n2);
// B=A (B ist transponierte der (n1 x n2)-Matrix A)

void linsolve(double *A, double *b, double *x, int n);
// Loesung eines linearen Gleichungssystems per Gauss-Elimination: Ax=b

void inv(double *A, double *B, int n);
// A=B-1, Inverse per Gauss-Jordan-Algorithmus

void delay(double *x, double *y, int n);
// y = rechtsverschobene Version von x

void advance(double *x, double *y, int n);
// y = linksverschobene Version von x

void conv(double *x, double *y, double *v, int n);
// v ist lineare Faltung von x und y

```

A.2 linalg.c

```

// Funktionen fuer Matrix -und Vektorrechnung
// Uebergabe per Zeiger, Matrizen sind Zeilenweise im Speicher abzulegen

#include <math.h>
#include <stdio.h>

void matout(double *A, int n1, int n2)
// gibt die (n1 x n2)-Matrix A aus
{
    int i,j;
    printf("\n");
    for (i=0; i<n1; i++)
    {
        for (j=0; j<n2; j++) printf(" %.3e",*(A+n2*i+j));
        printf("\n");
    }
    printf("\n");
}

void plus(double *A, double *B, double *C, int n1, int n2)
// C=A+B, A,B,C (n1 x n2)-Matrizen
{
    int i,j;

    for (i=0; i<n1; i++) for (j=0; j<n2; j++)
        *(C+n2*i+j)=*(A+n2*i+j)**(B+n2*i+j);
}

void minus(double *A, double *B, double *C, int n1, int n2)
// C=A-B, A,B,C (n1 x n2)-Matrizen

```

```

{
    int i,j;

    for (i=0; i<n1; i++) for (j=0; j<n2; j++)
        *(C+n2*i+j)=*(A+n2*i+j)-*(B+n2*i+j);
}

void mult(double r, double *A, double *B, int n1, int n2)
// B=r*A, r skalar, A,B (n1 x n2)-Matrizen
{
    int i,j;

    for (i=0; i<n1; i++) for (j=0; j<n2; j++) *(B+n2*i+j)=r***(A+n2*i+j);
}

void matmult(double *A, double *B, double *C, int n1, int n2, int n3)
// C=AB, A (n1 x n2)-Matrix, B (n2 x n3)-Matrix
{
    int i,j,k;
    double *M;

    M=(double *)calloc(n1*n3,sizeof(double));

    for (i=0; i<n1; i++) for (j=0; j<n3; j++)
    {
        *(M+n3*i+j)=0;
        for (k=0; k<n2; k++)
            *(M+n3*i+j)+=*(A+n2*i+k)**(B+n3*k+j);
    }
    for (i=0; i<n1; i++) for (j=0; j<n3; j++)
        *(C+n3*i+j)=*(M+n3*i+j);
    free(M);
}

void transpose(double *A, double *B, int n1, int n2)
// B=A (B ist transponierte der (n1 x n2)-Matrix A)
{
    int i,j;
    double *M;

    M=(double *)calloc(n1*n2,sizeof(double));

    for (i=0; i<n1; i++) for (j=0; j<n2; j++)
        *(M+n1*j+i)=*(A+n2*i+j);
    for (i=0; i<n1; i++) for (j=0; j<n2; j++)
        *(B+n1*j+i)=*(M+n1*j+i);
    free(M);
}

void linsolve(double *A, double *b, double *x, int n)
// Loesung eines linearen Gleichungssystems per Gauss-Elemination: Ax=b
{
    int i,j,k,r,regular=1;
    double l,dummy;
    double *A2,*b2;

    A2=(double *)calloc(n*n,sizeof(double)); // Kopien von A und b
    b2=(double *)calloc(n,sizeof(double));
    for (i=0; i<n; i++) for (j=0; j<n; j++) *(A2+n*i+j)=*(A+n*i+j);
    for (i=0; i<n; i++) *(b2+i)=*(b+i);

    for (j=0; j<n; j++)
    {
        r=j; // Pivotwahl
        for (i=j+1; i<n; i++)

```

```

        if (fabs(*(A2+n*i+j))>fabs(*(A2+n*r+j))) r=i;

    if (*(A2+n*r+j)==0) regular=0;
    if (regular==1)
    {
        dummy=(b2+j); // tausche j-te gegen r-te Zeile
        *(b2+j)=*(b2+r);
        *(b2+r)=dummy;
        for (k=j; k<n; k++)
        {
            dummy=(A2+n*j+k);
            *(A2+n*j+k)=(A2+n*r+k);
            *(A2+n*r+k)=dummy;
        }

        for (i=j+1; i<n; i++) // vielfache Zeilen addieren
        {
            l=(A2+n*i+j)/(A2+n*j+j);
            *(b2+i)-=l*(b2+j);
            for (k=j; k<n; k++) *(A2+n*i+k)-=l*(A2+n*j+k);
        }
    }
}
if (regular==0) printf("die Matrix ist singulaer\n");
else // Transformiertes System loesen
{
    for (i=0; i<n; i++) *(x+i)=(b2+i);
    for (i=n-1; i>=0; i--)
    {
        for (k=i+1; k<n; k++) *(x+i)-=(A2+n*i+k)*(x+k);
        *(x+i)/(A2+n*i+i);
    }
}
free(A2);
free(b2);
}

void inv(double *A, double *B, int n)
// A=B^(-1), Inverse per Gauss-Jordan-Algorithmus
{
    int i,j,k,r,dum,regular=1;
    int p[n];
    double v[n];
    double l,dummy;

    for (i=0; i<n; i++) for (j=0; j<n; j++) *(B+n*i+j)=(A+n*i+j);
    for (j=0; j<n; j++) p[j]=j;
    for (j=0; j<n; j++)
    {
        r=j; // Pivotwahl
        for (i=j+1; i<n; i++)
            if (fabs(*(B+n*i+j))>fabs(*(B+n*r+j))) r=i;

        if (*(B+n*r+j)==0) regular=0;
        if (regular==1)
        {
            if (r>j) // Zeilentausch
            {
                for (k=0; k<n; k++)
                {
                    dummy=(B+n*j+k);
                    *(B+n*j+k)=(B+n*r+k);
                    *(B+n*r+k)=dummy;
                }
                dum=p[j]; p[j]=p[r]; p[r]=dum;
            }

            l=1/(B+n*j+j); // Transformation

```

```

    for (i=0; i<n; i++) *(B+n*i+j)*=1;
    *(B+n*j+j)=1;
    for (k=0; k<j; k++)
    {
        for (i=0; i<j; i++) *(B+n*i+k)-=*(B+n*i+j)**(B+n*j+k);
        for (i=j+1; i<n; i++) *(B+n*i+k)-=*(B+n*i+j)**(B+n*j+k);
        *(B+n*j+k)*=-1;
    }
    for (k=j+1; k<n; k++)
    {
        for (i=0; i<j; i++) *(B+n*i+k)-=*(B+n*i+j)**(B+n*j+k);
        for (i=j+1; i<n; i++) *(B+n*i+k)-=*(B+n*i+j)**(B+n*j+k);
        *(B+n*j+k)*=-1;
    }
}
if (regular==0) printf("die Matrix ist singulaer\n");

for (i=0; i<n; i++) // Spaltentausch
{
    for (k=0; k<n; k++) v[p[k]]=*(B+n*i+k);
    for (k=0; k<n; k++) *(B+n*i+k)=v[k];
}
}

void delay(double *x, double *y, int n)
// y = rechtsverschobene Version von x
{
    int i;

    for (i=0; i<n-1; i++) *(y+n-1-i)=*(x+n-2-i);
    *y=0;
}

void advance(double *x, double *y, int n)
// y = linksverschobene Version von x
{
    int i;

    for (i=0; i<n-1; i++) *(y+i)=*(x+i+1);
    *(y+n-1)=0;
}

void conv(double *x, double *y, double *v, int n)
// v ist lineare Faltung von x und y
{
    int i,j,a,b;
    double *x2,*y2;

    x2=(double *)calloc(n,sizeof(double));
    y2=(double *)calloc(n,sizeof(double));

    for (i=0; i<n; i++) *(x2+i)=*(x+i); // kopien von x,y
    for (i=0; i<n; i++) *(y2+i)=*(y+i); // falls v=x oder v=y

    for (i=0; i<2*n-1; i++) *(v+i)=0;
    for (i=0; i<2*n-1; i++)
    {
        a=i+1-n; if (a<0) a=0;
        b=i+1; if (b>n) b=n;
        for (j=a; j<b; j++) *(v+i)+=*(x+j)**(y+i-j);
    }
    free(x2);
    free(y2);
}

```

A.3 filter.h

```
// Deklarationen einiger Funktionen fuer Filter und Filterbaenke

void einlesen(double *x, int Lp, char fname[]);
// liest Filterkoeffizienten aus einer Datei

void schreiben(double *x, int Lp, char fname[]);
// schreibt Filterkoeffizienten in eine Datei

void fir(double *h, int N, double wp, double ws);
// Entwurf eines einfachen FIR-Filters

void xu_lin(double *p, int K, int r,
            int samps, double alpha, double epsilon);
// Entwurf eines fast-PR Prototypfilters nach Xu, Lu, Antoniou

void e_mat(int N, double *S, double ws);
// erzeugt Matrix fuer die Berechnung der Sperrenergie eines Filters

void e_mat_sym(int N, double *S, double ws);
// wie e_mat, aber fuer symmetrische Filter

double energy(int N, double *h, double *S);
// berechnet die Sperrenergie eines Filters

void get_polycomp(double *h, double *p, int K, int r, int l);
// liefert eine bestimmte Polyphasenkomponente eines Filters

void set_polycomp(double *h, double *p, int K, int r, int l);
// setzt eine bestimmte Polyphasenkomponente eines Filters

double pr_vio(double *h, int K, int r, int d, int l, int n);
// ergibt Verletzungen der PR-Bedingungen eines Filters als
// Prototyp fuer kosinus-modulierte Filterbaenke
```

A.4 filter.c

```
// einige Funktionen fuer Filter und Filterbaenke

#include <math.h>
#include <stdio.h>
#include "linalg.h"

void einlesen(double *x, int Lp, char fname[])
/*****
/* liest (Filter-)Koeffizienten aus einer ASCII-Datei */
/*
/* *x      Koeffizienten werden ab Adresse x abgelegt */
/* Lp     Anzahl der Koeffizienten */
/* fname[] Quellpfad der Datei */
*****/
{
    FILE *fp1;
    int i,j;
    float zahl;

    fp1=fopen(fname,"r");

    for (i=0; i<Lp; i++)
```

```

    {
        fscanf(fp1,"%e",&zahl);
        *(x+i)=zahl;
    }
    fclose(fp1);
}

void schreiben(double *x, int Lp, char fname[])
/*****
/* schreibt (Filter-)Koeffizienten in eine ASCII-Datei */
/*
/* Koeffizienten sind ab Adresse x abgelegt */
/* Lp Anzahl der Koeffizienten */
/* fname[] Zielpfad der Datei */
*****/
{
    FILE *fp1;
    int i;

    fp1=fopen(fname,"w");
    for (i=0; i<Lp; i++)
    {
        if (*(x+i)>0) putc(' ',fp1);
        fprintf(fp1," %.16e",*(x+i));
    }
    fclose(fp1);
}

void fir(double *h, int N, double wp, double ws)
/*****
/* Berechnung eines linearphasigen FIR Tiefpassfilters */
/* ( per Initial-Schritt des Remez-Exchange-Algorithmus [Mi] ) */
/* keine Stuetzstellenanpassung, also kein Equiripple-Filter */
/*
/* *h Filterkoeffizienten */
/* N Filterordnung ( Filter hat N+1 Koeffizienten ) */
/* wp Grenze des Durchlassbereichs */
/* ws Grenze des Sperrbereichs */
*****/
{
    int L,i,j,anz1,anz2;
    double pi=3.14159;
    double *w; // Frequenz-Stuetzstellen
    double *D; // gewuenschte Amplituden an den Stuetzstellen: D[i]=D(w[i])
    double *a; // Koeffizienten der Amplitude, (7.96) bzw.(7.100) in [Mi]
    double *M; // Matrix ( erhalte a aus M*a=D )

    wp=pi*wp; ws=pi*ws;
    if (N%2==0) L=N/2; else L=(N-1)/2;

    w=(double *)calloc(L+2,sizeof(double));
    D=(double *)calloc(L+2,sizeof(double));
    a=(double *)calloc(L+2,sizeof(double));
    M=(double *)calloc((L+2)*(L+2),sizeof(double));

/***** Stuetzstellen berechnen *****/

    anz1=(int)(wp/(wp+pi-ws)*(L+2));
    if (anz1==0) anz1=1;
    anz2=L+2-anz1;
    for (i=0; i<anz1; i++) *(w+i)=(i+0.5)*wp/(anz1-0.5);
    for (i=0; i<anz2; i++) *(w+i+anz1)=ws+i*(pi-ws)/(anz2-0.5);

/**** Gleichungssystem erstellen und loesen *****/
/**** (7.118) in [Mi] ****/

    for (i=0; i<anz1; i++) if (N%2==1) *(D+i)=1;

```

```

        else *(D+i)=1/cos(*(w+i)/2);
    for (i=0; i<anz2; i++) *(D+anz1+i)=0;
    for (i=0; i<L+2; i++) for (j=0; j<L+1; j++)
        *(M+(L+2)*i+j)=cos(j*w[i]);
    for (i=0; i<L+2; i++) *(M+(L+2)*i+L+1)=pow(-1,i);
    linsolve(M,D,a,L+2);

/**** Filterkoeffizienten ermitteln *****/

    if (N%2==0) // linearphasiger FIR-Filter vom Typ 1
    {
        *(h+L)=*a; // (7.97) in [Mi]
        for (i=1; i<L+1; i++) *(h+L-i)=*(a+i)/2;

        for (i=L+1; i<N+1; i++) *(h+i)=*(h+N-i); // spiegeln
    }
    else // linearphasiger FIR-Filter vom Typ 2
    {
        *h=(a+L)/4; // (7.101) und (7.99) in [Mi]
        *(h+L)=*(a+1)+2*a)/4;
        for (i=2; i<L+1; i++) *(h+L+1-i)=*(a+i)+*(a+i-1))/4;

        for (i=L+1; i<N+1; i++) *(h+i)=*(h+N-i); // spiegeln
    }

    free(w);
    free(D);
    free(a);
    free(M);
}

void xu_lin(double *p, int K, int r, int samps, double alpha, double epsilon)
/*****
/* Berechnung eines linearphasigen fast PR Prototypfilters fuer */
/* eine kosinus-modulierte Filterbank nach Xu,Lu und Antoniou [Xu] */
/* */
/* *p Filterkoeffizienten */
/* K Anzahl der komplexen Teilbaender (M=K/2 reelle Teilbaender) */
/* r Laenge der Polyphasenkomponenten => Prototyp-Laenge=K*r */
/* samps Anzahl der Stuetzstellen (k in [Xu]) */
/* alpha Gewichtung der Sperr-Energie gegenueber Amplitudenverz. */
/* epsilon Toleranz (Abbruchskriterium der Iteration z.B. 1e-10) */
/* */
/*****
{
    double *q; // Updater
    double *wp; // Stuetzpunkte
    double *Us; // Matrix zur Berechnung der Sperr-Energie
    double *U; // Matrix zur Approximation der Aplidudenverzerrung
    double *UT; // Transponierte von U (Speicherverschwendung, jaja)
    double *H1,*H2; // Diagonalmatrizen H(wp), H(wp-pi/M) in [Xu]
    double *Ut1,*Ut2; // Matrizen Ut(wp), Ut(wp-pi/M) in [Xu]
    double *A; // Hilfsmatrix
    double *c,*d; // Hilfsvektoren
    double pi=3.14159;
    double ws=2*pi/K; // Grenzfrequenz
    double delta,zahl;
    int N=K*r; // Prototypplaenge
    int M=K/2;
    int i,j;

    q=(double *)calloc(N/2,sizeof(double));
    wp=(double *)calloc(samps,sizeof(double));
    Us=(double *)calloc(N/2*N/2,sizeof(double));
    U=(double *)calloc(samps*N/2,sizeof(double));
    UT=(double *)calloc(samps*N/2,sizeof(double));
    Ut1=(double *)calloc(samps*N/2,sizeof(double));
    Ut2=(double *)calloc(samps*N/2,sizeof(double));

```

```

A=(double *)calloc(N/2*N/2,sizeof(double));
H1=(double *)calloc(samps,sizeof(double));
H2=(double *)calloc(samps,sizeof(double));
c=(double *)calloc(N/2,sizeof(double));
d=(double *)calloc(samps,sizeof(double));

fir(p,K*r-1,0.3/M,1.0/M);
for (i=0; i<samps; i++) *(wp+i)=i*pi/((samps-1)*M);

/***** Us mit Formel [Xu] (15d) *****/
for (i=0; i<N/2; i++) for (j=0; j<N/2; j++)
{
  if (i==j) *(Us+N/2*i+j)
    =2*(pi-ws -sin((2*i-N+1)*ws)/(2*i-N+1));
  else *(Us+N/2*i+j)
    =2*(sin((i-j)*ws)/(j-i)-sin((i+j-N+1)*ws)/(i+j-N+1));
}
mult(alpha,Us,Us,N/2,N/2);

delta=2*epsilon;
while (delta>epsilon)
{
  /***** H1=H(wp) mit Formel [Xu] (15b) *****/
  for (i=0; i<samps; i++)
  {
    for (j=0; j<N/2; j++)
      *(c+j)=cos((N-1-2*j)*(wp+i)/2);
    matmult(p,c,&zahl,1,N/2,1);
    *(H1+i)=2*zahl;
  }

  /***** H2=H(wp-pi/M) mit Formel [Xu] (15b) *****/
  for (i=0; i<samps; i++)
  {
    for (j=0; j<N/2; j++)
      *(c+j)=cos((N-1-2*j)*(wp+i)-pi/M)/2);
    matmult(p,c,&zahl,1,N/2,1);
    *(H2+i)=2*zahl;
  }

  /***** Ut1=Ut(wp) mit Formel [Xu] (15c) *****/
  for (i=0; i<samps; i++) for (j=0; j<N/2; j++)
    *(Ut1+N/2*i+j)=2*cos((N-1-2*j)*(wp+i)/2);

  /***** Ut2=Ut(wp-pi/M) mit Formel [Xu] (15c) *****/
  for (i=0; i<samps; i++) for (j=0; j<N/2; j++)
    *(Ut2+N/2*i+j)=2*cos((N-1-2*j)*(wp+i)-pi/M)/2);

  /***** U laut Formel [Xu] (15a) *****/
  for (i=0; i<samps; i++) for (j=0; j<N/2; j++)
    *(U+N/2*i+j)=*(H1+i)**(Ut1+N/2*i+j)
      +*(H2+i)**(Ut2+N/2*i+j);

  for (i=0; i<samps; i++) *(d+i)=1;
  transpose(U,UT,samps,N/2);
  matmult(UT,d,c,N/2,samps,1); // c=Ud

  matmult(UT,U,A,N/2,samps,N/2); // A=UU
  plus(A,Us,A,N/2,N/2); // A=UU + alpha*Us
  linsolve(A,c,q,N/2); // Formel [Xu] (16)
  plus(p,q,p,N/2,1);
  mult(0.5,p,p,N/2,1); // [Xu] (17) mit tau=0.5
  minus(p,q,q,N/2,1);
  matmult(q,q,&delta,1,N/2,1);
  printf("delta=%.16e\n",delta);
}

for (i=0; i<N/2; i++) *(p+N-1-i)=*(p+i);

```

```

    free(q);
    free(wp);
    free(Us);
    free(U);
    free(UT);
    free(Ut1);
    free(Ut2);
    free(A);
    free(H1);
    free(H2);
    free(c);
    free(d);
}

void e_mat(int N, double *S, double ws)
// erzeugt Matrix S (gespeichert als Vektor) zur Berechnung der
// Sperrenergie eines Filters der Laenge N mit Grenzfrequenz ws
// siehe (4.10),(4.12)
{
    int i,j;
    double pi=3.14159265358979323846;
    S[0]=pi-ws;
    for (i=1; i<N ; i++)
    {
        S[i]=-sin(i*ws)/i;
    }
}

void e_mat_sym(int N, double *S, double ws)
// erzeugt (N/2 x N/2)-Matrix S zur Berechnung der Sperrenergie eines
// symmetrischen Filters der Laenge N (N gerade) mit Grenzfrequenz ws
// nach Formel (15d) in [Xu]
{
    int i,j;
    double pi=3.14159265358979323846;
    for (i=0; i<N/2; i++) for (j=0; j<N/2; j++)
    {
        if (i==j) *(S+N/2*i+j)
            =2*(pi-ws -sin((2*i-N+1)*ws)/(2*i-N+1));
        else *(S+N/2*i+j)
            =2*(sin((i-j)*ws)/(j-i)-sin((i+j-N+1)*ws)/(i+j-N+1));
    }
}

double energy(int N, double *h, double *S)
// Berechnet die Sperrenergie eines Filters h der Laenge N mit Hilfe
// der "Energie-Matrix" S, die mit e_mat erzeugt wurde
// siehe (4.11)
{
    int i,j;
    double result, teilsum;
    result=0;

    for (i=1; i<N; i++)
    {
        teilsum=0;
        for (j=0; j<N-i; j++) teilsum+=h[j]*h[j+i];
        result+=S[i]*teilsum;
    }
    result*=2;
    teilsum=0;
    for (j=0; j<N; j++) teilsum+=h[j]*h[j];
    result+=S[0]*teilsum;
    return result;
}

```

```

void get_polycomp(double *h, double *p, int K, int r, int l)
/*****
/* Bestimmung einer von K Polyphasenkomponenten eines Filters */
/* *h Filter (der Laenge K*r) (input) */
/* *p Polyphasenkomponente (output) */
/* K Anzahl der Komponenten (input) */
/* r Laenge der Komponenten (input) */
/* l Nummer der zu bestimmenden Komponente (input) */
*****/
{
    int j;
    for (j=0; j<r; j++) *(p+j)=*(h+K*j+l); // siehe bei (3.6)
}

void set_polycomp(double *h, double *p, int K, int r, int l)
/*****
/* Aendern einer Polyphasenkomponente eines Filters */
/* Parameter wie in set_polycomp, aber *h als output, *p als input */
*****/
{
    int j;
    for (j=0; j<r; j++) *(h+K*j+l)=*(p+j); // siehe bei (3.6)
}

double pr_vio(double *h, int K, int r, int d, int l, int n)
/*****
/* Bestimmung der Verletzung von PR-Bedingungen eines Filters, */
/* als Prototyp einer kosinusmodulierten Filterbank */
/* *h Filter (der Laenge K*r) */
/* K #komplexe Teilbaender der zug. Filterbank (4|K muss gelten) */
/* r Laenge der Polyphasenkomponenten von h */
/* d Parameter fuer die Verzoegerung der Filterbank: K*d+K+1 */
/* l,n Auswahl der PR-Bedingung (aus insgesamt K/4*(2*r-1)) */
/* l=0,...,K/4-1, n=0,...,2*n-2 */
*****/
{
    double *x,*y,*v1,*v2;
    x=(double *)calloc(r,sizeof(double));
    y=(double *)calloc(r,sizeof(double));
    v1=(double *)calloc(2*r-1,sizeof(double));
    v2=(double *)calloc(2*r-1,sizeof(double));

    get_polycomp(h,x,K,r,l);
    get_polycomp(h,y,K,r,K-1-l);
    conv(x,y,v1,r);
    get_polycomp(h,x,K,r,K/2+1);
    get_polycomp(h,y,K,r,K/2-1-l);
    conv(x,y,v2,r);
    // v1+v2 = linke Seite in (4.2) (l-te Bedingung)

    if (n==d) return fabs(*(v1+n)+*(v2+n)-(double)2/(K*K));
    else return fabs(*(v1+n)+*(v2+n));
    // gibt Verletzungen der Bedingungen in (4.2)

    free(x);
    free(y);
    free(v1);
    free(v2);
}

```

A.5 lifting.h

```
// Deklarationen der Liftingfunktionen (nach Kapitel 5)
```

```

void fsl(double a, double b, double d, // erster Liftingschritt
        double *h0, double *h1,
        double *f0, double *f1);

void zdl(double a, double b, // Liftingschritt: zero-delay
        double *h0, double *h1,
        double *f0, double *f1, int r);

void sdl(double a, double b, // Liftingschritt: single delay
        double *h0, double *h1,
        double *f0, double *f1, int r);

void mdl(double a, double b, // Liftingschritt: maximum delay
        double *h0, double *h1,
        double *f0, double *f1, int r);

void lifting(double *h, double *c, int K, int r);
// komplettes Lifting (Berechnung der Filterkoeffizienten)

void rev_fsl(double *a, double *b, double *d, // Umkehrung von fsl
            double *h0, double *h1,
            double *f0, double *f1, int K);

void rev_zdl(double *a, double *b, // Umkehrung von zdl
            double *h0, double *h1,
            double *f0, double *f1, int r, int n);

void rev_sdl(double *a, double *b, // Umkehrung von sdl
            double *h0, double *h1,
            double *f0, double *f1, int r, int n);

void rev_mdl(double *a, double *b, // Umkehrung von mdl
            double *h0, double *h1,
            double *f0, double *f1, int r);

void factorize(double *h, double *c, int K, int r);
// Umkehrung von lifting (Filterfaktorisierung in Liftingkoeffizienten)

```

A.6 lifting.c

```

// Funktionen fuer Lifting nach Kapitel 5 der Diplomarbeit
// hier fuer Prototypfilter mit Verzoeigerung Lp-1

#include <math.h>
#include <stdio.h>
#include "linalg.h"
#include "filter.h"

void fsl(double a, double b, double d,
        double *h0, double *h1,
        double *f0, double *f1)
// Erster Liftingschritt (neue Filter nach (5.13))
{
    *h0=*h0*(1+a+d+b*d+a*b*d);
    *h1=*h1*(1+b+a*b);
    *f0=*f0*(1-b+a*b);
    *f1=*f1*(1-a-d+b*d-a*b*d);
}

void zdl(double a, double b,
        double *h0, double *h1,
        double *f0, double *f1, int r)

```

```

// Liftingschritt: zero-delay
// neue filter nach (5.7) und (5.9)
{
    double *x;
    x=(double *)calloc(r,sizeof(double));

    mult(a,h1,x,r,1);
    delay(x,x,r);
    plus(h0,x,h0,r,1);           //  $H_0(z)=H_0(z)+a*z^{-1}*H_1(z)$ 

    mult(a,f0,x,r,1);
    delay(x,x,r);
    minus(f1,x,f1,r,1);        //  $F_1(z)=F_1(z)-a*z^{-1}*F_0(z)$ 

    mult(b,h0,x,r,1);
    plus(h1,x,h1,r,1);         //  $H_1(z)=H_1(z)+b*H_0(z)$ 

    mult(b,f1,x,r,1);
    minus(f0,x,f0,r,1);        //  $F_0(z)=F_0(z)-b*F_1(z)$ 

    free(x);
}

void sdl(double a, double b,
         double *h0, double *h1,
         double *f0, double *f1, int r)
// Liftingschritt: type-1 single-delay
// neue Filter nach (5.8) und (5.9)
{
    double *x,*y;
    x=(double *)calloc(r,sizeof(double));
    y=(double *)calloc(r,sizeof(double));

    delay(h0,x,r); mult(a,h1,y,r,1);
    plus(x,y,h0,r,1);           //  $H_0(z)=z^{-1}*H_0(z)+a*H_1(z)$ 

    delay(f1,x,r); mult(a,f0,y,r,1);
    minus(x,y,f1,r,1);         //  $F_1(z)=z^{-1}*F_1(z)-a*F_0(z)$ 

    mult(b,h0,x,r,1);
    plus(h1,x,h1,r,1);         //  $H_1(z)=H_1(z)+b*H_0(z)$ 

    mult(b,f1,x,r,1);
    minus(f0,x,f0,r,1);        //  $F_0(z)=F_0(z)-b*F_1(z)$ 

    free(x);
    free(y);
}

void mdl(double a, double b,
         double *h0, double *h1,
         double *f0, double *f1, int r)
// Liftingschritt: maximum-delay
// neue Filter nach (5.8) und (5.10)
{
    double *x,*y;
    x=(double *)calloc(r,sizeof(double));
    y=(double *)calloc(r,sizeof(double));

    delay(h0,x,r); mult(a,h1,y,r,1);
    plus(x,y,h0,r,1);           //  $H_0(z)=z^{-1}*H_0(z)+a*H_1(z)$ 

    delay(f1,x,r); mult(a,f0,y,r,1);
    minus(x,y,f1,r,1);         //  $F_1(z)=z^{-1}*F_1(z)-a*F_0(z)$ 

    delay(h1,x,r); mult(b,h0,y,r,1);
    plus(x,y,h1,r,1);           //  $H_1(z)=z^{-1}*H_1(z)+b*H_0(z)$ 
}

```

```

delay(f0,x,r); mult(b,f1,y,r,1);
minus(x,y,f0,r,1); // F0(z)=z-1*F0(z)-b*F1(z)

free(x);
free(y);
}

void lifting(double *h, double *c, int K, int r)
/*****
/* Berechnung der Prototypfilter-Koeffizienten einer per */
/* Liftingschema implementierten kosinus-modulierten Filterbank */
/* */
/* *h (Output) resultierende Filterkoeffizienten */
/* *c (Input) Vektor der Liftingkoeffizienten */
/* K,r (Input) Anzahl und Laenge der Polyphasenkomponenten des Filters */
/*****
{
int Lc=K/4*(2*r+1); // Anzahl der Liftingkoeffizienten
int i,j,l,n1,n2,n3; // Zaehlvariablen
double a,b,d; // Variablen fuer Liftingkoeffizienten
double *h0,*h1,*f0,*f1; // Variablen fuer Polyphasenkomponenten

h0=(double *)calloc(r,sizeof(double));
h1=(double *)calloc(r,sizeof(double));
f0=(double *)calloc(r,sizeof(double));
f1=(double *)calloc(r,sizeof(double));

n3=(r-1)/2; // # maximum-delay Liftingschritte pro Block
n2=(r-1)%2; // # single-delay ...
n1=(r-1)-n3-n2; // # zero-delay ...
// ergibt Verzoeigerung D=r-1 pro Block (PR-Bedingung in (4.2))

for (i=0; i<K; i++) for (j=0; j<r; j++) *(h+K*j+i)=0;
for (i=0; i<K; i++) *(h+i)=pow(K,-1);
// alle K Polyphasenkomponenten sind vor dem Lifting skalar: 1/K

for (l=0; l<K/4; l++) // die K/4 PR-Bedingungen nach (4.2)
{
// Auswahl der zur l-ten PR-Bedingung gehoerenden
// 4 Polyphasenkomponenten mit get_polycomp:
get_polycomp(h,h0,K,r,l); get_polycomp(h,f0,K,r,K-1-l);
get_polycomp(h,h1,K,r,K/2+1); get_polycomp(h,f1,K,r,K/2-1-l);
a=c[(2*r+1)*l]; b=c[(2*r+1)*l+1]; d=c[(2*r+1)*l+2];
fsl(a,b,d,h0,h1,f0,f1); // Erster Liftingschritt
for (i=0; i<n1; i++) // Liftingschritte vom Typ zero-delay
{
a=c[(2*r+1)*l + 2*i + 3];
b=c[(2*r+1)*l + 2*i + 4];
zdl(a,b,h0,h1,f0,f1,r);
}
for (i=n1; i<n1+n2; i++) // Liftingschritte vom Typ single-delay
{
a=c[(2*r+1)*l + 2*i + 3];
b=c[(2*r+1)*l + 2*i + 4];
sdl(a,b,h0,h1,f0,f1,r);
}
for (i=n1+n2; i<n1+n2+n3; i++) // Liftingschritte vom Typ maximum-delay
{
a=c[(2*r+1)*l + 2*i + 3];
b=c[(2*r+1)*l + 2*i + 4];
mdl(a,b,h0,h1,f0,f1,r);
}
// Polyphasenkomponenten von h mit set_polycomp ueberschreiben:
set_polycomp(h,h0,K,r,l); set_polycomp(h,f0,K,r,K-1-l);
set_polycomp(h,h1,K,r,K/2+1); set_polycomp(h,f1,K,r,K/2-1-l);
}
free(h0);

```

```

    free(h1);
    free(f0);
    free(f1);
}

/***** Inverse Liftingfunktionen (Faktorisierung) *****/

void rev_fsl(double *a, double *b, double *d,
             double *h0, double *h1,
             double *f0, double *f1, int K)
{
    *b=K/2*(h1-f0);
    *a=(K*h1-1*b)/(b);
    *d=(K/2*(h0-f1)-a)/(1+(a)*(b));
}

void rev_zdl(double *a, double *b,
             double *h0, double *h1,
             double *f0, double *f1, int r, int n)
{
    double *x;
    x=(double *)calloc(r,sizeof(double));

    *b=h1[n-1]/h0[n-1];           // b nach (5.11)

    mult(*b,h0,x,r,1);
    minus(h1,x,h1,r,1);          // H1(z)=H1(z)-b*H0(z)

    mult(*b,f1,x,r,1);
    plus(f0,x,f0,r,1);           // F0(z)=F0(z)+b*F1(z)

    *a=h0[n-1]/h1[n-2];          // a entsprechend (5.11)

    mult(*a,h1,x,r,1); delay(x,x,r);
    minus(h0,x,h0,r,1);          // H0(z)=H0(z)-a*z^(-1)*H1(z)

    mult(*a,f0,x,r,1); delay(x,x,r);
    plus(f1,x,f1,r,1);          // F1(z)=F1(z)+a*z^(-1)*F0(z)

    free(x);
}

void rev_sdl(double *a, double *b,
             double *h0, double *h1,
             double *f0, double *f1, int r, int n)
{
    double *x;
    x=(double *)calloc(r,sizeof(double));

    *b=h1[n-1]/h0[n-1];           // b nach (5.11)

    mult(*b,h0,x,r,1);
    minus(h1,x,h1,r,1);          // H1(z)=H1(z)-b*H0(z)

    mult(*b,f1,x,r,1);
    plus(f0,x,f0,r,1);           // F0(z)=F0(z)+b*F1(z)

    *a=h0[0]/h1[0];              // a entsprechend (5.11)

    mult(*a,h1,x,r,1);
    minus(h0,x,h0,r,1);
    advance(h0,h0,r);            // H0(z)=z*(H0(z)-a*H1(z))

    mult(*a,f0,x,r,1);
    plus(f1,x,f1,r,1);
    advance(f1,f1,r);            // F1(z)=z*(F1(z)+a*F0(z))
}

```

```

    free(x);
}

void rev_mdl(double *a, double *b,
            double *h0, double *h1,
            double *f0, double *f1, int r)
{
    double *x;
    x=(double *)calloc(r,sizeof(double));

    *b=h1[0]/h0[0];                // b entsprechend (5.11)

    mult(*b,h0,x,r,1);
    minus(h1,x,h1,r,1);
    advance(h1,h1,r);                // H1(z)=z*(H1(z)-b*H0(z))

    mult(*b,f1,x,r,1);
    plus(f0,x,f0,r,1);
    advance(f0,f0,r);                // F0(z)=z*(F0(z)+b*F1(z))

    *a=h0[0]/h1[0];                // a entsprechend (5.11)

    mult(*a,h1,x,r,1);
    minus(h0,x,h0,r,1);
    advance(h0,h0,r);                // H0(z)=z*(H0(z)-a*H1(z))

    mult(*a,f0,x,r,1);
    plus(f1,x,f1,r,1);
    advance(f1,f1,r);                // F1(z)=z*(F1(z)+a*F0(z))

    free(x);
}

void factorize(double *h, double *c, int K, int r)
/*****
/* Faktorisierung des Prototypfilters einer kosinus-modulierten      */
/* Filterbank mit PR-Eigenschaft in Liftingkoeffizienten          */
/*                                                                    */
/* *h (Input) resultierende Filterkoeffizienten                  */
/* *c (Output) Vektor der Liftingkoeffizienten                    */
/* *K,r (Input) Anzahl und Laenge der Polyphasenkomponenten des Filters */
*****/
{
    int Lc=K/4*(2*r+1);          // Anzahl der Liftingkoeffizienten
    int i,ii,j,l,n1,n2,n3,n;     // Zaehlvariablen
    double a,b,d;                // Variablen fuer Liftingkoeffizienten
    double *h0,*h1,*f0,*f1;     // Variablen fuer Polyphasenkomponenten

    h0=(double *)calloc(r,sizeof(double));
    h1=(double *)calloc(r,sizeof(double));
    f0=(double *)calloc(r,sizeof(double));
    f1=(double *)calloc(r,sizeof(double));

    // Berechnung der Anzahlen der einzelnen Liftingschritte
    n3=(r-1)/2;                 // # maximum-delay Liftingschritte pro Block
    n2=(r-1)%2;                 // # single-delay ...
    n1=(r-1)-n3-n2;             // # zero-delay ...
    // Verzoeigerung pro Block (PR-Bedingung in (4.2)) ist D=r-1

    for (l=0; l<K/4; l++) // die K/4 PR-Bedingungen nach (4.2)
    {
        // Auswahl der zur l-ten PR-Bedingung gehoerenden
        // 4 Polyphasenkomponenten mit get_polycomp
        get_polycomp(h,h0,K,r,l); get_polycomp(h,f0,K,r,K-1-l);
        get_polycomp(h,h1,K,r,K/2+1); get_polycomp(h,f1,K,r,K/2-1-l);
        for (i=0; i<n3; i++) // Liftingschritte vom Typ maximum-delay,
        {                     // Bestimmung der Liftingkoeffizienten

```

```

    rev_md1(&a,&b,h0,h1,f0,f1,r);
    ii=n1+n2+n3-1-i;
    c[(2*r+1)*l + 2*ii + 3]=a;
    c[(2*r+1)*l + 2*ii + 4]=b;
}
for (i=0; i<n2; i++) // Liftingschritte vom Typ single-delay,
{ // Bestimmung der Liftingkoeffizienten
    n=r-n3-i;
    rev_sd1(&a,&b,h0,h1,f0,f1,r,n);
    ii=n1+n2-1-i;
    c[(2*r+1)*l + 2*ii + 3]=a;
    c[(2*r+1)*l + 2*ii + 4]=b;
}
for (i=0; i<n1; i++) // Liftingschritte vom Typ zero-delay,
{ // Bestimmung der Liftingkoeffizienten
    n=r-n3-n2-i;
    rev_zd1(&a,&b,h0,h1,f0,f1,r,n);
    ii=n1-1-i;
    c[(2*r+1)*l + 2*ii + 3]=a;
    c[(2*r+1)*l + 2*ii + 4]=b;
}
rev_fsl(&a,&b,&d,h0,h1,f0,f1,K); // Erster Liftingschritt
c[(2*r+1)*l]=a; c[(2*r+1)*l+1]=b;
c[(2*r+1)*l+2]=d; // Koeffizienten Bestimmt
// Polyphasenkomponenten von h mit set_polycomp ueberschreiben:
set_polycomp(h,h0,K,r,l); set_polycomp(h,f0,K,r,K-1-l);
set_polycomp(h,h1,K,r,K/2+1); set_polycomp(h,f1,K,r,K/2-1-l);
}
free(h0);
free(h1);
free(f0);
free(f1);
}

```

Anhang B

Das Programm *cosbank*

An dieser Stelle ist der Quellcode des Programms *cosbank* abgedruckt, mit dem ich alle reellwertigen PR-Prototypfilter gemäß Abschnitt 4.3 entworfen habe.

```
/* **** */
/* Programm zum Entwurf von linear-phasigen Prototyp- */
/* filtern fuer perfekt rekonstruierende kosinusmodulierte */
/* Filterbaenke (reelle Loesungen, nicht diskret) */
/* */
/* Benutzt SQP-Software "CFSQP Version 2.5d" */
/* */
/* (c) Volkmar Sauerland, Mai 2002 */
/* **** */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "linalg.h"
#include "filter.h"
#include "cfsqpusr.h"

#define pi 3.1415926535897932846

void obj(); // Berechnung der quadratischen Zielfunktion
// zur Uebergabe an die "cfsqp"-Routine
void gradob(); // Gradient von obj
void cntr(); // Berechnung der quadratischen Nebenbedingungen
// (Perfekte Rekonstruktion) zur Uebergabe an "cfsqp"
void gradcn(); // Gradient von cntr
void cosbank(); // Entwurf einer kosinusmodulierten PR Filterbank

int main()
{
    int K,r,i,j;
    double *p,*S,cv,mcv;

    printf("Hallo, dieses Programm entwirft linearphasige Prototypen\n");
    printf("fuer perfekt rekonstruierende kosinusmodulierte Filterbaenke.\n");
    printf("Bitte die Anzahl K der komplexen Teilbaender und die Laenge r\n");
    printf("der K Polyphasenkomponenten eingeben (Prototypplaenge Lp=K*r).\n");
    printf("K="); scanf("%i",&K);
    printf("r="); scanf("%i",&r);
}
```

```

p=(double *)calloc(K*r,sizeof(double));
S=(double *)calloc(K*r,sizeof(double));

e_mat(K*r,S,2*pi/K);
cosbank(p,K,r); // Entwurf des PR Prototypen
for (i=0; i<K*r/2; i++) p[K*r-1-i]=p[i]; // Koeffizienten spiegeln
printf("energie=%.3e\n",energy(K*r,p,S));
mcv=0;
for (i=0; i<K/4; i++) for (j=0; j<2*r-1; j++)
{
    cv=pr_vio(p,K,r,r-1,i,j);
    if (cv>mcv) mcv=cv;
}
printf("mcv=%.3e\n",mcv);
printf("schreibe Prototypen p nach ../filter/p_pr.asc\n");
schreiben(p,K*r,"../filter/p_pr.asc"); // Filter speichern

free(p);
}

```

```

void cosbank(double *p, int K, int r)
/*****
/* Entwurf eines linearphasigen Prototypfilters fuer eine */
/* kosinusmodulierte, perfekt rekonstruierende (PR) Filterbank */
/* in 2 Schritten: */
/* - Entwurf eines fast PR (NPR) Prototypen */
/* (Funktion "xu_lin" in der Unit "filter.c") */
/* - Mittels SQP-Verfahren ( hier "CFSQP V. 2.5d" ) aus dem */
/* NPR Prototypen einen PR Prototypen machen */
/* */
/* *p Prototypfilter */
/* K Anzahl der komplexen Teilbaender, 4|K soll gelten */
/* r Laenge der K Polyphasenkomponenten des Prototypen */
*****/
{
    int i,samps=100;
    char choice;
    double alpha=1.0e-3;

    int nparam,nf,nineq,neq,mode,iprint,miter,neqn,nineqn,
        ncsrl,ncsrn,nfsr,mesh_pts[1],inform;
    double bigbnd,eps,epsneq,udelta;
    double *bl,*bu,*f,*g,*lambda;
    double *cd; // Parameter fuer die CFSQP-Routine

    mode=100; // Initialisierung der cfsqp-Parameter
    iprint=12;
    miter=10000;
    bigbnd=1.e10;
    eps=1.e-7;
    epsneq=1.e-13;
    udelta=0.e0;
    nparam=K*r/2;
    nf=1;
    neqn=K*r/4;
    nineqn=0;
    nineq=0;
    neq=K*r/4;
    ncsrl=ncsrn=nfsr=mesh_pts[0]=0;
    bl=(double *)calloc(nparam,sizeof(double));
    bu=(double *)calloc(nparam,sizeof(double));
    f=(double *)calloc(nf+1,sizeof(double));
    g=(double *)calloc(nineq+neq+1,sizeof(double));
    lambda=(double *)calloc(nineq+neq+nf+nparam+1,sizeof(double));
    cd=(double *)calloc(K*r*K*r/4+2,sizeof(double));

    for (i=0; i<nparam; i++) { *(bl+i)=-1.0; *(bu+i)=1.0; }
}

```

```

cd[0]=K; cd[1]=r;
e_mat_sym(K*r,cd+2,2*pi/K);

do
{
xu_lin(p,K,r,samps,alpha,1.0e-10);

cfsqp(nparam,nf,nfsr,nineqn,nineq,neqn,neq,ncsrn,ncsrn,mesh_pts,
mode,iprint,miter,&inform,bigbnd,eps,epsneq,udelta,bl,bu,
p,f,g,lambda,obj,cntr,gradob,gradcn,cd);

choice='n';
if (inform!=0)
{
printf("Die Optimierungsroutine ist nicht norma");
printf("\n1 terminiert. Bitte pruefen Sie (s.o.),\n");
printf("ob Zielfunktion (objectives) und Nebenb");
printf("edingungen (constraints) dennoch in\n");
printf("Ordnung gehen.\n");
printf("Soll ein neuer Versuch mit anderem Fast");
printf("-PR-Prototypen stattfinden (j/n) ?");
scanf("%c",&choice);
}
samps+=(int)(samps/10); alpha*=2;
}
while (choice!='n');

free(bl);
free(bu);
free(f);
free(g);
free(lambda);
free(cd);
}

void
obj(nparam,j,x,fj,cd)
int nparam,j;
double *x,*fj;
double *cd;
/*****
/* Zielfunktion fuer die "cfsqp"-Routine. */
/* Die Sperrbereichs-Energie (4.11) des zu optimierenden Filters */
/* wird uebergeben, Rechnung beruecksichtigt Symmetrie. */
/* */
/* nparam (Input) Problemdimension ( # Liftingkoeffizienten ) */
/* j (Input) Nummer der Zielfunktion (hier gibt es nur eine) */
/* *x (Input) aktuelle Iterierte des Liftingvektors */
/* *fj (Output) Zeiger auf den Wert der Zielfunktion */
/* *cd (Input) Zeiger auf die Zusatzparameter K,r,S */
*****/
{
int i,k,Lp;
double erg;

Lp=(int)(cd[0]*cd[1]);
erg=0;
for (i=0; i<Lp/2 ; i++) for (k=0; k<Lp/2 ; k++)
erg+*(cd+Lp/2*i+k+2)**(x+i)**(x+k);
*fj=erg;
return;
}

void
gradob(nparam,j,x,gradfj,dummy,cd)
int nparam,j;
double *x,*gradfj;

```

```

void (* dummy)();
double *cd;
// Gradient der Zielfunktion (Uebergabe an "cfsqp" per Zeiger *gradfj)
{
    int ii,i,l,Lp;
    double erg;

    Lp=(int)(cd[0]*cd[1]);
    for (ii=1; ii<nparam+1; ii++)
    {
        double erg;
        erg=0;
        for (i=0; i<Lp/2 ; i++) for (l=0; l<Lp/2 ; l++)
            if (i==(ii-1)) erg+=2**((cd+Lp/2*i+l+2)*x[l]);
        gradfj[ii-1]=erg;
    }
    return;
}

```

```

void
cntr(nparam,j,x,gj,cd)
int nparam,j;
double *x,*gj;
double *cd;
/*****
/* Nebenbedingungen fuer die "cfsqp"-Routine. */
/* Die PR-Constraints (4.8) des zu optimierenden Filters werden */
/* uebergeben, Rechnung beruecksichtigt Symmetrie. */
/* */
/* nparam (Input) Problemdimension ( # Liftingkoeffizienten ) */
/* j (Input) Nummer Nebenbedingung */
/* *x (Input) aktuelle Iterierte des Liftingvektors */
/* *gj (Output) Zeiger auf den Wert der j-ten Nebenbedingung */
/* *cd (Input) Zeiger auf die Zusatzparameter K und r */
*****/
{
    int K,r,M,Lp;
    int i,a,b,l,n,i1,i2,i3,i4;
    double erg;
    K=(int)cd[0]; r=(int)cd[1];
    Lp=K*r; M=K/2;
    l=(j-1)/r;
    n=(j-1)%r;
    erg=0;
    a=n+1-r; if (a<0) a=0;
    b=n+1; if (b>r) b=r;
    for (i=a; i<b; i++)
    {
        i1=2*M*i+2*M-1-l;
        i2=2*M*(n-i)+1;
        i3=2*M*i+M-1-l;
        i4=2*M*(n-i)+M+1;
        if (i1>(nparam-1)) i1=Lp-1-i1;
        if (i2>(nparam-1)) i2=Lp-1-i2;
        if (i3>(nparam-1)) i3=Lp-1-i3;
        if (i4>(nparam-1)) i4=Lp-1-i4;
        erg+=(x[i1]*x[i2]+x[i3]*x[i4]);
    }
    erg*=K*K/2;
    if (n==r-1) erg-=1;
    erg/=K*K/2;
    *gj=erg;
    return;
}

```

```

void
gradcn(nparam,j,x,gradgj,dummy,cd)

```

```

int nparam,j;
double *x,*gradgj;
void (* dummy)();
double *cd;
// Gradienten der Nebenbedingungen (Uebergabe an "cfsqp" per Zeiger *gradgj)
{
  int r,M,Lp;
  int ii,i,a,b,l,n,i1,i2,i3,i4;
  double erg;
  M=(int)(cd[0]/2); r=(int)cd[1];
  Lp=2*M*r;
  for (ii=1; ii<nparam+1; ii++)
  {
    l=(j-1)/r;
    n=(j-1)%r;
    erg=0;
    a=n+1-r; if (a<0) a=0;
    b=n+1; if (b>r) b=r;
    for (i=a; i<b; i++)
    {
      i1=2*M*i+2*M-1-l;
      i2=2*M*(n-i)+1;
      i3=2*M*i+M-1-l;
      i4=2*M*(n-i)+M+1;
      if (i1>(nparam-1)) i1=Lp-1-i1;
      if (i2>(nparam-1)) i2=Lp-1-i2;
      if (i3>(nparam-1)) i3=Lp-1-i3;
      if (i4>(nparam-1)) i4=Lp-1-i4;

      if (i1==ii-1) erg+=x[i2];
      if (i2==ii-1) erg+=x[i1];
      if (i3==ii-1) erg+=x[i4];
      if (i4==ii-1) erg+=x[i3];
    }
    gradgj[ii-1]=erg;
  }
  return;
}

```

Anhang C

Das Programm *quant*

Im Programm *quant* ist unser Algorithmus 2 aus Abschnitt 5.4 implementiert, der bei der nachträglichen Diskretisierung der reellen Prototypfilter die besten Resultate lieferte.

```
/* *****  
/* Programm zur Diskretisierung von Prototypfiltern fuer      */  
/* kosinusmodulierte Filterbaenke mit perfekter Rekonstruktion (PR) */  
/* */  
/* (c) Volkmar Sauerland, Sep. 2002 */  
/* *****  
  
// Das Programm liest einen reellen, linearphasigen PR Prototypfilter  
// aus einer ASCII-Datei und faktorisiert ihn gemaess Kapitel 5 in  
// Liftingkoeffizienten.  
// Anschliessend werden die Liftingkoeffizienten zu CSD-Zahlen  
// diskretisiert. Hierfuer wird ein Iteratives Verfahren benutzt, das  
// die noch nicht diskreten Koeffizienten per NLP-Verfahren  
// (hier CFSQP Version 2.5d von AEM-Design) in jedem Schritt reoptimiert:  
// Algorithmus2 in Kapitel 5  
  
#include <math.h>  
#include <time.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include "linalg.h"  
#include "filter.h"  
#include "lifting.h"  
#include "cfsqpusr.h"  
  
void ausgabe(); // gibt die Liftingkoeffizienten aus  
double csd_appr(); // rundet eine reelle Zahl auf eine CSD-Naehierung  
double e_lift(); // berechnet Zielfunktion (Sperr-Energie)  
void obj(); // uebergibt Zielfunktion an "cfsqp"-Routine  
void reopt(); // reoptimiert einen Prototypfilter  
double sensitivity(); // testet Empfindlichkeit einzelner Lifting-  
// koeffizienten auf CSD-Diskretisierungen  
void quantize(); // quantisiert die reellen Liftingkoeffizienten  
// eines Prototypfilters  
void cntr();  
  
double pi=3.14159265358979323846;
```

```

int
main() {

    int K;          // Anzahl der Polyphasenkomponenten des Filters
    int r;          // Laenge der Polyphasenkomponenten
    int Lp;         // Filterlaenge
    int Lc;         // # Liftingkoeffizienten
    double *h;     // Filter
    double *c;     // Liftingvektor
    char dir_name[] = "../filter/"; // Quell -und Zielverzeichnis
    char file_name[30]; // Name fuer die Quelldatei
    char name_in[40];

    int i,j, bits, percent, steps1, steps2;
    double e_faktor;

    printf("Hallo, dieses Programm dient der Diskretisierung von\n");
    printf("Prototypfiltern perfekt rekonstruierender, kosinus-\n");
    printf("modulierter Filterbaenke.\n\n");
    printf("Benoetigt wird eine ASCII-Datei mit den Koeffizienten\n");
    printf("eines reellen Prototypfilters im Verzeichnes\n");
    printf("\n..\filter\\\n\n");
    printf("Zum Laden des Filters bitte den Dateinamen eingeben : ");
    scanf("%s", file_name);
    for (i=0; i<10; i++) name_in[i]=dir_name[i];
    for (i=0; i<30; i++) name_in[i+10]=file_name[i];
    printf("Bitte die durch 4 teilbare Anzahl K und die Laenge r der\n");
    printf("Polyphasenkomponenten des Prototypen eingeben:\n");
    printf("K="); scanf("%i",&K);
    printf("r="); scanf("%i",&r);
    Lp=K*r;
    Lc=K/4*(2*r+1);
    printf("max. Wortlaenge der Liftingkoeffizienten =");
    scanf("%i",&bits);
    printf("max. Sperrenergie-Abweichung in % =");
    scanf("%i",&percent);
    printf("Anzahl der SQP-Schritte in der Such-Schleife =");
    scanf("%i",&steps1);
    printf("Anzahl der SQP-Schritte beim Reoptimieren =");
    scanf("%i",&steps2);
    e_faktor=(double)percent/100+1; // max. Energie-Ratio

    h=(double *)calloc(Lp,sizeof(double));
    c=(double *)calloc(Lc,sizeof(double));
    einlesen(h,Lp,name_in);
    factorize(h,c,K,r);
    quantize(K,r,c, bits, e_faktor, steps1, steps2);

    schreiben(c,K/4*(2*r+1), "../filter/liftco.asc");
    lifting(h,c,K,r);
    printf("\n");
    for (i=0; i<K/4; i++) for (j=0; j<2*r-1; j++)
        printf("PR-Bedingung [%i,%i] = %.16e\n",
            i+1, j+1, pr_vio(h,K,r,r-1,i,j));
    schreiben(h,Lp, "../filter/proto.asc");
    printf("\nFilterkoeffizienten als \"proto.asc\"\n");
    printf("und Liftingkoeffizienten als \"liftco.asc\"\n");
    printf("ins Verzeichnis \"..\filter\\\" geschrieben\n");

    free(h);
    free(c);
}

void quantize(int K, int r, double *c, int bits,
              double e_faktor, int steps1, int steps2)
/* Diskretisierung der Liftingkoeffizienten des Prototypfilters */
/* nach Algorithmus 2 in Kapitel 5 */

```

```

/*
/* K,r: Anzahl und Laenge der Polyphasenkomponenten des Filters
/* c: Vektor der Liftingkoeffizienten ( # = K/4*(2*r+1) )
/* e_faktor: maximale gewuenschte Ratio der Energien im Sperrbereich
/* (reelle Version / diskrete Version)
/* steps1,steps2: maximale Anzahl der SQP-Schritte bei Reoptimierung
/* der noch reellen Koeffizienten in der Suchschleife bzw. in der
/* Diskretisierungsschleife (Schritt 3.(b)ii.D. bzw. Schritt 3.(d)
/* in Algorithmus 2
/*****
{
    int Lp=K*r; // Prototyp-Laenge
    int Lc=K/4*(2*r+1); // Anzahl der Liftingkoeffizienten
    int nzb[Lc]; // Nicht-Null-Bits in c
    double mark[Lc]; // Indexvektor fuer fixierte Koeff.
    int code[bits]; // Darstellung von CSD-Zahlen
    int i,j,nz_bits,best_j,best_nz_bits;
    int bitsum;
    double S[Lp]; // Matrix (als Vektor) der Zielfunktion
    double scale,e1,der,best_der;

    e_mat(K*r,S,2*pi/K);

    for (i=0; i<Lc; i++) mark[i]=0;
    e1=e_lift(K,r,c,S);
    printf("Sperrenergie des reellen Filters : %.16e\n",e1);
    bitsum=0;
    for (i=0; i<Lc; i++)
    {
        best_j=0;
        best_nz_bits=99;
        best_der=1+e_faktor;
        scale=0.5+0.5*pow((double)(1+i)/Lc,2);

        //printf("scale=%.3e\n",scale);

        for (j=0; j<Lc; j++) if (mark[j]==0)
        {
            nz_bits=-1;
            do
            {
                nz_bits++;
                der=sensitivity(K,r,c,mark,S,j,bitsum,nz_bits,steps1)/e1;
                printf("Runden von c[%i] auf %i Bit gibt Gueteratio %.3e\n",
                    j,nz_bits,der);
            }
            while (der>=1+(e_faktor-1)*scale && 2*nz_bits<bits);

            if ( nz_bits < best_nz_bits )
            { best_der=der; best_j=j; best_nz_bits=nz_bits; }
            if ( nz_bits==best_nz_bits && der < best_der )
            { best_der=der; best_j=j; }
        }

        c[best_j]=csd_appr(c[best_j],code,bitsum,best_nz_bits);
        mark[best_j]=1;
        nzb[best_j]=best_nz_bits;
        printf("%i.) nach Runden von c[%i] auf %i bits : %.16e\n",
            i+1,best_j,best_nz_bits,e_lift(K,r,c,S));

        bitsum+=best_nz_bits;
        reopt(K,r,c,mark,S,steps2);
        printf("optimiert auf %.16e\n",e_lift(K,r,c,S));
    }

    printf("Diskretisierung abgeschlossen\n");
    printf("Gueteratio: %.3e\n",e_lift(K,r,c,S)/e1);
    printf("Anzahl der Nicht-Null-Bits : %i\n",bitsum);
    printf("\nDie Liftingkoeffizienten sind:\n\n");

```

```

    for (i=0; i<Lc; i++)
    {
        csd_appr(c[i],code,bits,nzb[i]);
        printf("c%i = %.16e =",i,c[i]);
        for (j=0; j<bits; j++) printf("%i",code[j]);
        printf(" (%i bit)\n",nzb[i]);
    }

    return;
}

double csd_appr(double zahl, int *code, int bits, int nz_bits)
/*****
/* Berechnet wird fuer eine reelle Zahl die naechste Naehung durch */
/* eine CSD-Zahl mit vorgegener Anzahl von Bits und Nicht-Null-Bits */
/* */
/* zahl: reelle Zahl */
/* code: hier wird der Code der Darstellung abgelegt */
/* bits: Anzahl der Bits der CSD-Zahl */
/* nz_bits: Anzahl der Nicht-Null-Bits der CSD-Zahl */
/*****
{
    int i,num_nz,leit,pot,pot1,pot2;
    double erg;

    for (i=0; i<bits; i++) code[i]=0;
    leit=(int)(log(fabs(zahl))/log(2)); // |zahl|>=1 g.d.w. 2^leit<=|zahl|
    if (fabs(zahl)>=1) leit++;
    erg=0;
    num_nz=0; pot=leit;
    while (zahl!=erg && num_nz<nz_bits && pot>=leit-bits+1)
    {
        pot1=(int)(log(fabs(zahl-erg))/log(2)); // |zahl-erg|>=1 g.d.w.
        if (fabs(zahl-erg)>=1) pot2=pot1+1; // (2^pot1)<=|zahl-erg|
        else pot2=pot1-1;
        if ( fabs(pow(2,pot1)-fabs(zahl-erg))
            < fabs(pow(2,pot2)-fabs(zahl-erg)) && pot1>=leit-bits+1 )
            pot=pot1;
        else pot=pot2;
        if (pot>=leit-bits+1)
        {
            num_nz++;
            if (erg>zahl) { erg-=pow(2,pot); code[leit-pot]=-1; }
            else { erg+=pow(2,pot); code[leit-pot]=1; }
        }
    }
    return erg;
}

void reopt(int K, int r, double *c, double *mark,
           double *S, int steps)
/*****
/* Reoptimierung der Liftingkoeffizienten eines Prototypfilters */
/* per SQP-Verfahren */
/* */
/* K,r Anzahl und Laenge der Polyphasenkomponenten des Prototyps */
/* c Vektor der Liftingkoeffizienten ( # = K/4*(2*r+1) ) */
/* mark Vektor (Eintraege 0 oder 1) */
/* 0: die zug. Liftingkoeffizienten werden reoptimiert */
/* 1: die zug. Liftingkoeffizienten sind fest */
/* S Matrix (als Vektor) der quadratischen Zielfunktion */
/* steps Anzahl der Reoptimierungs-Schritte (SQP-Schritte) */
/*****
{
    int nparam,nf,nineq,neq,mode,iprint,neqn,nineqn,
        ncsr1,ncsrn,nfsr,mesh_pts[1],inform,i,j;
    double bigbnd,eps,epsneq,udelta;

```

```

double *bl,*bu,*f,*g,*lambda; // Parameter fuer den SQP-Solver
double *cd; // CFSQP (v. 2.5d von AEM-Design)

mode=100; // Initialisierung der CFSQP-Parameter
iprint=0;
bigbnd=1.e10;
eps=1.e-7;
epsneq=1.e-13;
udelta=0.e0;
nparam=K/4*(2*r+1);
nf=1;
neqn=0;
nineqn=0;
nineq=0;
neq=0;
ncsrl=ncsrn=nfsr=mesh_pts[0]=0;
bl=(double *)calloc(nparam,sizeof(double));
bu=(double *)calloc(nparam,sizeof(double));
f=(double *)calloc(nf+1,sizeof(double));
g=(double *)calloc(nineq+neq+1,sizeof(double));
lambda=(double *)calloc(nineq+neq+nf+nparam+1,sizeof(double));
cd=(double *)calloc(2+2*nparam+K*r,sizeof(double));

for (i=0; i<nparam; i++) { bl[i]=-bigbnd; bu[i]=bigbnd; }
cd[0]=K; cd[1]=r;
for (i=0; i<nparam; i++) { cd[2+i]=c[i]; cd[2+nparam+i]=mark[i]; }
for (i=0; i<K*r; i++) cd[2+2*nparam+i]=S[i];

cfsqp(nparam,nf,nfsr,nineqn,nineq,neqn,neq,ncsrl,ncsrn,mesh_pts,
mode,iprint,steps,&inform,bigbnd,eps,epsneq,udelta,bl,bu,
c,f,g,lambda,obj,cntr,grobfd,grcnfd,cd);

for (i=0; i<nparam; i++) if (mark[i]==1) c[i]=cd[2+i];

free(bl);
free(bu);
free(f);
free(g);
free(lambda);
free(cd);
return;
}

double e_lift(int K, int r, double *c, double *S)
/*****
/* Berechnung der Sperrbereichsenergie eines Prototypfilters */
/*
/* K,r Anzahl und Laenge der Polyphasenkomponenten des Prototyps */
/* c Vektor der Liftingkoeffizienten ( # = K/4*(2*r+1) ), */
/* aus denen der Prototyp resultiert */
/* S Matrix (als Vektor) fuer die Sperr_Energie */
/* (quadratische Zielfunktion) */
*****/
{
int Lp=K*r;
int Lc=K/4*(2*r+1);
double h[Lp];

lifting(h,c,K,r);
return energy(Lp,h,S);
}

void obj(nparam,j,x,fj,cd)
/*****
/* Zielfunktion fuer die "cfsqp"-Routine */
/* Die Sperrbereichs-Energie des zu (Re)Optimierenden Filters */
/* wird Uebergeben. */

```

```

/*
/* nparam (Input) Problemdimension ( # Liftingkoeffizienten ) */
/* j (Input) Nummer der Zielfunktion (hier gibt es nur eine) */
/* x (Input) aktuelle Iterierte des Liftingvektors */
/* fj (Output) Zeiger auf den Wert der j-ten Zielfunktion */
/* cd (Input) Zeiger auf die Zusatzparameter K,r,c,mark,S,
/* der "cfsqp" aufrufenden Funktion "reopt": */
/* -cd[0]=K, cd[1] = r */
/* -cd[2]...cd[nparam+1] = c */
/* -cd[nparam+2]...cd[2*nparam+1] = mark */
/* -cd[2*nparam+2]...cd[2*nparam+2+K*r-1] = S */
/*****
int nparam,j;
double *x,*fj;
double *cd;
{
    int K=(int)cd[0];
    int r=(int)cd[1];
    double c[nparam];
    int i;
    for (i=0; i<nparam; i++)
        if (cd[nparam+2+i]==0) c[i]=x[i]; else c[i]=cd[2+i];
    *fj=e_lift(K,r,c,cd+(2*nparam+2));
    return;
}

double sensitivity(int K, int r, double *c, double *mark, double *S,
    int i, int bits, int nz_bits, int steps)
/*****
/* Berechnet die Sperr-Energie, die sich bei Diskretisierung
/* eines Liftingkoeffizienten ergeben wuerde
/* (ggf. bei Reoptimierung nicht fixierter Koeffizienten)
/*
/* K,r Anzahl und Laenge der Polyphasenkomponenten des Prototyps
/* c Vektor der Liftingkoeffizienten ( # = K/4*(2*r+1) ),
/* aus denen der Prototyp resultiert
/* mark Vektor (Eintraege 0 oder 1)
/* 0: die zug. Liftingkoeffizienten werden reoptimiert
/* 1: die zug. Liftingkoeffizienten sind fest
/* S Matrix (als Vektor) der quadratischen Zielfunktion
/* i Der i-te Liftingkoeffizient wird quantisiert...
/* nz_bits ...als CSD-Zahl mit nz_bits Nicht-Null-Bits
/* steps Anzahl der Reoptimierungsschritte
/*****
{
    int j;
    int Lc=K/4*(2*r+1);
    double erg;
    double c2[Lc];
    int code[bits];

    for (j=0; j<Lc; j++) c2[j]=c[j];
    c2[i]=csd_appr(c[i],code,bits,nz_bits);
    mark[i]=1;
    if (steps>0) reopt(K,r,c2,mark,S,steps);
    erg=e_lift(K,r,c2,S);
    mark[i]=0;
    return erg;
}

void
cntr(nparam,j,x,gj,cd)
int nparam,j;
double *x,*gj;
void *cd;
{
    *gj=0;

```

```
    return;  
}
```

Literaturverzeichnis

- [Al] F. Alizadeh, *Interior point methods in semidefinite programming with application to combinatorial optimization*, SIAM J. Optimization, 5(1995), pp. 13-51.
- [DS] I. Daubechies, W. Sweldens. *Factoring wavelet transforms into lifting steps*. Technical report, Bell Laboratories, Lucent Technologies, 1996.
- [Fl] N. J. Fliege. *Computational efficiency of Modified DFT polyphase filter banks*. In Proc. 27th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, USA, November 1993.
- [GK] C. Geiger, C. Kanzow. *Theorie und Numerik restringierter Optimierungsaufgaben*. Springer, 2002.
- [HKN] P. N. Heller, T. Karp, T. Q. Nguyen. *A general formulation for modulated filter banks*. Submitted to IEEE Trans. on Signal Processing, 1996.
- [Kl] J. Kliewer. *Beiträge zum Entwurf modulierter Filterbänke für verschiedene Teilbandabtastraten*. Nr. 16 in Arbeiten über Digitale Signalverarbeitung, Hrsg. U. Heute, Shaker Verlag, 1999.
- [KM1] T. Karp, A. Mertins. *Efficient prototype filter realizations for cosine-modulated filter banks*. In Proc. 16th GRETSI Symposium on Signal and Image Processing, Grenoble, France, September 1997.
- [KM2] T. Karp, A. Mertins. *Lifting schemes for biorthogonal modulated filter banks*. In Proc. International Conference on Digital Signal Processing, Santorini, Greece, July 1997.
- [Ma] H. S. Malvar. *Extended lapped transforms: properties, applications, and fast algorithms*. IEEE Trans. on Signal Processing, vol. 40, pp. 2703-2714, November 1992.
- [Me] A. Mertins. *Subspace Approach for the Design of Cosine-Modulated Filter Banks with Linear-Phase Prototype Filter*. IEEE Trans. on Signal Processing, vol. 46, no. 10, pp. 2812-2818, October 1998

- [Mi] S.K. Mitra. *Digital signal processing: a computer-based approach*. McGraw-Hill/Irwin, Boston, 2001.
- [MKK] A. Mertins, T. Karp, J. Kliewer. *Design of Perfect Reconstruction Integer-Modulated Filter Banks*. Proc. International Symposium on Signal Processing and its Applications (ISSPA'99), pp. 591-594, Brisbane, Australia, August 1999.
- [Op] A. V. Oppenheim, A. S. Willsky. *Signale und Systeme*. Prentice Hall, 1983.
- [Vai] P.P. Vaidyanathan. *Multirate Systems and Filter Banks*. Prentice Hall, 1993.
- [XLA] H. Xu, W. S. Lu, A. Antoniou. *Efficient Iterative Design Method for Cosine-Modulated QMF Banks*. IEEE Trans. on Signal Processing, Vol. 44, no. 7, pp. 1657-1668, July 1996.

Danksagung

Zunächst möchte ich Herrn Prof. Dr. A. Srivastav für die Ausgabe des interessanten Themas und die freundliche Betreuung meiner Diplomarbeit danken.

Herrn Dr.-Ing. Jörg Kliewer danke ich für seine wertvolle Hilfe bei der Einarbeitung in die Materie der digitalen Filterbänke und diverse Tipps und Anregungen.

Mein besonderer Dank gilt schließlich meinen Eltern für ihre Unterstützung während des gesamten Studiums.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, die vorliegende Arbeit eigenständig angefertigt und keine anderen als die angegebenen Hilfsmittel verwendet zu haben.

Kiel, 14. Februar 2003