Hochschule für angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Segmentierte Faltung -

Analyse und Implementierungsansatz für mobile Endgeräte

Master-Thesis

Eingereicht von:

Malte Spiegelberg
(Matrikel-Nr.: 1930156)

Betreuer:

Prof. Dr. Robert Mores

Prof. Dr. Andreas Plaß

Hamburg, August 2012

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Hiermit versichere ich, dass ich die vorliegende Master–Thesis mit dem Titel

Segmentierte Faltung - Analyse und Implementierungsansatz

selbstständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z.B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

Hamburg, im August 2013 Malte Spiegelberg

1 Einführung

Die Umsetzung der Faltung in Echtzeit durch digitale Signalverarbeitung ist mit Hilfe der diskreten Faltung nicht möglich. Rechenwerke sind nicht in der Lage, diese so schnell umzusetzen, dass eine direkte Ausgabe in Echtzeit möglich ist. Die schnellere Berechnung mit Hilfe der zyklischen Faltung im Frequenzbereich ist zwar schneller, jedoch benötigt man zur Ermittlung des Faltungsergebnisses sowohl Eingangssignal als auch Impulsantwort in voller Länge. Die Segmentierung der Faltung durch eine Aufteilung von Eingangssignal und Impulsantwort in Segmente sorgt für eine Unterteilung des Gesamtprozesses in Teilprozesse. Diese können entsprechend schneller berechnet werden und ermöglichen dadurch eine schnellere Umsetzung der Faltung. Willam G. Gardner stellte im Jahr 1995 den zero-delay convolver vor ([Gar, 1995]), eine Technik, welche die diskrete Faltung und die zyklische Faltung kombiniert um so eine latenzfreie Faltung umzusetzen. In der heutigen Zeit berechnen Prozessoren die zyklische Faltung ausreichend schnell, sodass ein Ansatz ohne diskrete Faltung funktionsfähig sein sollte.

Diese Thesis stellt zunächst Grundlagen der digitalen Signalverarbeitung vor, welche zur Ermittlung eines Faltungsergebnisses der zyklischen Faltung mit Hilfe eines digitalen Rechenwerks erforderlich sind. Anschließend wird das Prinzip der Faltung im diskreten, zyklischen und segmentierten Ablauf gezeigt. Ein Schwerpunkt liegt dabei in der Betrachtung der Berechnung des Endergebnisses aus den Teilergebnissen der segmentierten Faltung und den möglichen Ansätzen. Die segmentierte Faltung wird im Folgenden analysiert. Dabei stehen der Arbeitsaufwand für eine Umsetzung sowie der benötigte Speicher für den Prozess bei einer möglichen Umsetzung im Mittelpunkt. Aus der Analyse des Prozesses wird dann eine Umformung in einen Pseudocode durchgeführt, welcher eine Hinführung auf die in den Folgekapiteln vorgestellten Programmiertexte in MATLAB® und C sind. Nachfolgend werden weitere Aspekte einer möglichen Umsetzung der segmentierten Faltung auf Mikroprozessoren beschrieben.

2 Grundlagen

2.1 DFT - Diskrete Fourier Transformation

2.1.1 Berechnung der DFT und IDFT

Die DFT ist eine spezielle Form der Fourier-Transformation für Abtastsignale (FTA). Bei der DFT wird das Intervall, indem die Berechnung durchgeführt wird, im Zeitbereich eingeschränkt. Somit läuft die Summe nicht mehr über ein unendlich langes diskretes Intervall, sondern nur über einen Ausschnitt, welcher begrenzt ist. Die Einschränkung im Zeitbereich hat eine Einschränkung im Frequenzbereich zur Folge: „Aus N komplexen (meistens aber reellen) Abtastwerten können höchstens N komplexe Amplitudenwerte (Spektralwerte) berechnet werden.“ [Mey, 2009, S. 165].

Durch das Betrachten eines Blocks der Länge N des Signals x[n] wird das zugehörige Spektrum diskret, es enthält genau N Spektrallinien. Hier ergibt sich zudem ein Zusammenhang zwischen der Fourier-Reihe und der DFT. Die Fourier-Reihe ist im Zeitbereich ebenfalls begrenzt. Dabei wird genau eine Periode des kontinuierlichen Signals untersucht. In dieser steckt der gesamte Informationsgehalt des Signals, welcher im Frequenzbereich in diskreten Spektrallinien abgebildet wird. Die DFT setzt nun mit einem diskreten und im Zeitbereich begrenzten Signal an. Dadurch wird der Frequenzbereich periodisch. Der Informationsgehalt des untersuchten Signals der Blocklänge N ist folglich in einer Periode des Frequenzbereichs zu finden. Deswegen begrenzt man in der DFT den Frequenzbereich auf eine Periode und erfasst so den gesamten Informationsgehalt des analysierten diskreten Signals. Deswegen kann die DFT sowohl aus der FTA als auch aus der komplexen Fourier-Reihe hergeleitet werden. Betrachtet man die FTA (1), so enthält diese als Ergebnis ein kontinuierliches Spektrum.

$$\begin{aligned} X_A\left(e^{j\Omega}\right) &= \sum_{n=-\infty}^{\infty} x\left[n\right] \cdot e^{-jn\Omega} \\ \Omega &= \omega \cdot t = \frac{\omega}{f_A} \end{aligned} \tag{1}$$

Wird das Intervall der FTA nun auf die Blocklänge N eingeschränkt, kann der Frequenzbereich umgeschrieben werden und enthält diskrete Werte (2).

$$\begin{aligned} X\left(\omega\right) &= \sum_{n=0}^{N-1} x\left[n\right] \cdot e^{-jn\omega T} \\ \omega &= 2\pi \cdot f \rightarrow \frac{2\pi \cdot k}{NT}; k = 0, 1, 2, ..., N-1 \end{aligned} \tag{2}$$

Beim Einsetzen der diskreten Frequenzvariable für den Wert ω fällt der Faktor T heraus. Die Frequenzvariable kann nun auf den Wert k reduziert werden. Alle anderen Werte sind bei der Berechnung konstant. Somit ergibt sich die Formel für die

Berechnung der DFT (3).

$$DFT\left(x\left[n\right]\right) = X\left[k\right] = \sum_{n=0}^{N-1} x\left[n\right] \cdot e^{-j2\pi\frac{kn}{N}} \tag{3}$$

Die Position der Spektrallinien im Frequenzbereich ist nach der DFT im Bereich von 0 bis f_A. Die DFT hat neben der Periodizität im Frequenzbereich noch weitere Eigenschaften. Im Bereich Audio ist dabei insbesondere das Spektrum von reellen Zeitsignalen wichtig. Dieses ist bei der DFT konjugiert komplex. Für Anwendungen der DFT kann dadurch fast die Hälfte der Berechnungen eingespart werden. Die Rechnung läuft dann nur über die Hälfte der Blocklänge, die restlichen Ergebnisse können durch komplexes Konjugieren ermittelt werden. Das Beispiel in (4) zeigt diesen Zusammenhang. Ebenso wird deutlich, dass der Gleichanteil ($k = 0$) als einziger Wert nicht konjugiert komplex auftaucht.

$$\begin{aligned} x\left[n\right] &= \left[10, 10, 10, 10, 0, 0, 0, 0\right], N = 8 \\ X\left[k\right] &= \left[40;\ 10 - j24, 14;\ 0, 10 - j4, 14;\ 0, 10 + j4, 14;\ 0, 10 + 24, 14\right] \end{aligned} \tag{4}$$

Die Rücktransformation in den Zeitbereich erfolgt mit der IDFT. Auch hier ergeben sich in der Formel Gemeinsamkeiten zwischen der Fourier-Reihe und der DFT. Um das Ergebnis zu erhalten, muss mit dem Wert $\frac{1}{N}$ normiert werden, ebenso ist der Exponent in der Exponentialfunktion positiv (5).

$$IDFT\left(X\left[k\right]\right) = x\left[n\right] = \frac{1}{N}\sum_{n=0}^{N-1} X\left[k\right] \cdot e^{j2\pi\frac{kn}{N}} \tag{5}$$

In Prozessen der digitalen Signalverarbeitung ist es von Vorteil, Speicher zu sparen. Die IDFT als eigene Formel zu speichern und zu berechnen kann zu einem solchen erhöhten Speicheraufwand führen. Dieser kann jedoch verringert werden, indem man die IDFT mit Hilfe der DFT berechnet.

2.1.2 Berechnung der IDFT mit der DFT

Durch Umformungen lässt sich die IDFT mit der Formel der DFT berechnen. Diese Umformung sorgt für eine Ersparnis, weil die IDFT Formel nicht bekannt sein muss, um in den Zeitbereich zurück zu transformieren. Die Umformungen der IDFT zur Berechnung mit der DFT sind aus [Mey, 2009] und [Zöl, 2005] entnommen.

Zur Berechnung der IDFT mit dem DFT-Algorithmus wird zunächst komplex konjugiert. Zur einfacheren Darstellung wird zusätzlich die Exponentialfunktion um-

geschrieben (6).

$$\begin{aligned} x[n] &= \frac{1}{N}\sum_{n=0}^{N-1} X[k]\cdot e^{j2\pi\frac{kn}{N}} \\ W_N^{nk} &= e^{-j2\pi\frac{kn}{N}} \\ x^*[n] &= \frac{1}{N}\sum_{n=0}^{N-1} X^*[k]\cdot W_N^{nk} \end{aligned} \tag{6}$$

Der Ausdruck in der Summe entspricht nun einer DFT. Die linke Seite des Ausdrucks muss entsprechend umgeformt werden, um $x[n]$ zu erhalten. Um dies zu erreichen, wird zunächst mit der imaginären Einheit j multipliziert (8). Die Berechnung orientiert sich dabei am Zusammenhang in Ausdruck (7).

$$z = \mathrm{Re}(z) + j\cdot\mathrm{Im}(z) \Rightarrow z^* = \mathrm{Re}(z) - j\cdot\mathrm{Im}(z) \Rightarrow j\cdot z^* = j\cdot\mathrm{Re}(z) + \mathrm{Im}(z) \tag{7}$$

$$x^*[n]\cdot j = \frac{j}{N}\sum_{n=0}^{N-1} j\cdot X^*[k]\cdot W_N^{nk} \tag{8}$$

Dieser Zusammenhang ermöglicht bereits die Berechnung der DFT mit der IDFT anhand einer in der Praxis der Signalverarbeitung verwendbaren Form. Diese wird in [Mey, 2009, S. 171] wie folgt angegeben:

1. Tauschen von Real- und Imaginärteil von $X[k]$
2. Berechnen der DFT
3. Tauschen von Real- und Imaginärteil des Ergebnisses
4. Normieren mit dem Faktor $\frac{1}{N}$

Die Formel für die IDFT kann mit Hilfe einer weiteren Umformung ermittelt werden, welche dafür sorgt, dass die linke Seite wieder $x[n]$ zeigt. Diese wird durch erneutes konjugieren und multiplizieren mit der imaginären Einheit erreicht (9).

$$IDFT(X[k]) = \frac{j}{N}\left[DFT\{j\cdot X[k]^*\}\right]^* \tag{9}$$

2.1.3 Rechenaufwand der DFT

Die DFT benötigt zur Berechnung einer Spektrallinie N Multiplikationen und N Additionen. Folglich ist ihr Aufwand für ein Spektrum der Länge N mit N^2 Multiplikationen und N^2 Additionen beschrieben. Für reelle Eingangsfolgen reduziert sich der Aufwand auf $\frac{N^2}{2}$ Multiplikationen und $\frac{N^2}{2}$ Additionen, hinzu kommen dann die komplexen Konjugationen, deren Anzahl N^2 beträgt. Je kleiner die Blocklänge N ist, desto mehr lohnt sich die DFT im Gegensatz zur im Folgeabschnitt vorgestellte FFT. „Wenn also nur ein Spektralwert interessiert, so wendet man besser die DFT an.“ [Mey, 2009, S. 175]

2.2 FFT – Fast Fourier Transformation

2.2.1 Von der DFT zur FFT

In praktischen Anwendungen findet die DFT in der zuvor vorgestellten Variante der Berechnung fast keine Verwendung. In fast allen Fällen wird die FFT genutzt. FFT ist die Bezeichnung für einen Algorithmus, der die DFT optimiert und dadurch besonders bei größeren Blocklängen erhebliche Einsparungen bei Arbeitsaufwand und Speicherbedarf mit sich bringt. Entwickelt wurde die FFT im Jahr 1965 von J.W. Cooley und John Tukey, die ersten Ansätze sind bereits 1805 in Werken von Carl Friedrich Gauss zu finden.

Bei der Berechnung der FFT spielen insbesondere die Gewichtungsfaktoren W_N eine zentrale Rolle. Diese sind komplexe Vektoren der Länge eins, deren Winkel sich, je nach der zu berechnenden Spektrallinie $\frac{X[k]}{k}$ verändert (10).

$$\begin{aligned} W_N^{nk} &= e^{-j2\pi\frac{nk}{N}} = \left(e^{-j\frac{2\pi}{N}}\right)^{nk} = \left(\sqrt[N]{e^{-j2\pi}}\right)^{nk} \\ \left|W_N^{nk}\right| &= 1 \end{aligned} \tag{10}$$

Durch die Blocklänge N, welche im Nenner des Exponenten steht, ergeben sich genau N verschiedene Werte, also auch N verschiedene Winkel (11). Diese verteilen sich mit gleichem Abstand über einen Kreisdurchlauf, sie sind daher für ein bestimmtes N im Voraus bekannt.

$$\arg\left(W_N^{nk}\right) = \frac{\arg\left(e^{-j2\pi nk}\right) + a \cdot 2\pi}{N}; a = 0, 1, 2, ..., N-1 \tag{11}$$

Aus diesem Grund kann die FFT schon vor der Rechnung Aufwand einsparen, indem die Gewichtungsfaktoren vorberechnet werden und im Speicher abgelegt sind. Der FFT Algorithmus teilt die DFT als Gesamtberechnung in Teilberechnungen auf, welche einfacher zu lösen sind und somit Rechenaufwand einsparen. Dabei sind Aufteilungen auf mit einer Blocklänge von $N = 2^n$ sinnvoll, weil sich dann die Blöcke für eine Teilberechnung auf die halbe Länge verringern. Diese Teilberechnungen werden anschließend auf das Gesamtproblem zurückgeführt. Dazu werden zwei DFTs halber Länge berechnet, deren Eingangswerte zum Beispiel aus den geraden und ungeraden Werten von $x[n]$ bestehen. Die Gewichtungsfaktoren werden entsprechend umgeformt, sodass die Rechenersparnis deutlich wird (12).

$$\begin{aligned} X[k] &= \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi\frac{kn}{N}} = \sum_{m=0}^{M-1} x[2m] \cdot W_N^{2mk} + \sum_{m=0}^{M-1} x[2m+1] \cdot W_N^{(2m+1)k} \\ W_N^{2mk} &= e^{-j2\pi\frac{2mk}{N}} = e^{-j2\pi\frac{2mk}{N}} = e^{-j2\pi\frac{mk}{\frac{N}{2}}} = e^{-j2\pi\frac{mk}{M}} = W_M^{mk} \\ W_N^{(2m+1)k} &= W_N^k \cdot W_M^{mk} \\ X[k] &= \sum_{m=0}^{M-1} x[2m] \cdot W_M^{mk} + W_N^k \sum_{m=0}^{M-1} x[2m+1] \cdot W_M^{mk} \end{aligned} \tag{12}$$

Der Gewichtungsfaktor W_M^{mk} wird jetzt in beiden Teil-DFTs benötigt, muss daher nur einmal berechnet werden. Der Gewichtungsfaktor W_N^k wird pro Berechnungsebene nur einmal benötigt. Wird das Gesamtproblem nun auf DFTs der Länge eins herunterskaliert, enthält der Gewichtungsfaktor W_M^{mk} den Winkel $0°$ und somit den Wert $W_M^{mk} = 1$. Es ist daher keine Multiplikation nötig. Dies ist die wesentliche Ersparnis des FFT-Algorithmus. Um den Algorithmus in der Praxis möglichst effektiv einsetzen zu können, wurden speizelle Algorithmen entwickelt. Der bekannteste ist der Radix-2 Algorithmus, welcher auch im Rahmen dieser Thesis zu Analysen und Berechnungen genutzt wurde.

2.2.2 Radix-2 FFT-Algorithmus

Zur Berechnung mit dem Radix-2 FFT-Algorithmus muss die Gesamtblocklänge 2^n entsprechen. Diese Länge kann auch durch zero-padding erreicht werden, also eine Erweiterung des Eingangssignals mit Nullen, bis die entsprechende Länge erreicht ist. Das Eingangssignal $x[n]$ muss vor Beginn der Berechnung der FFT komplett bekannt sein. Es wird aufgeteilt in Blöcke, welche für die Teilberechnungen genutzt werden. Dazu wird entweder anhand der fortlaufenden Funktionswerte aufgeteilt (z.B. erster Block $x[0]-x[7]$, zweiter Block $x[8]-x[15]$, usw.), oder es wird sortiert nach geraden und ungeraden Werten (z.B. erster Block $x[0,2,4,6]$, zweiter Block $x[1,3,5,7]$). Die Sortierung nach geraden und ungeraden Werten hat den Hintergrund, dass am Ausgang nicht die gleiche Reihenfolge der Spektralwerte anliegt, wie am Eingang. Um diese herzustellen, muss entweder am Eingang oder am Ausgang sortiert werden. Man spricht dann von Bitumkehr am Ein- oder Ausgang (in der Literatur meistens mit engl. decimation in time/frequency bezeichnet), weil in binärer Darstellung das erste und letzte Bit vertauscht wird, um dies zu realisieren (Tab. 1).

n	000	001	010	011	100	101	110	111
Bitumkehr	000	100	010	110	001	101	011	111

Tabelle 1: Bitumkehr beim Radix-2 Algorithmus

Die Gewichtungsfaktoren W_N sind auf der ersten Ebene $W_N^0 = 1$ und $W_N^1 = -1$, sie enthalten die Winkel $0°$ und $180°$. Die Ersparnis bei den Berechnungen innerhalb der FFT beim Radix-2 Algorithmus lässt sich am einfachsten mit einem Schmetterlingsgraph abbilden (Abb. 1). Dieser zeigt den Signalverlauf einer Teil-DFT mit zwei Eingangswerten $x_1[n]$ und $x_2[n]$. Ohne Optimierung würden bei der Berechnung dieses Teilproblems beide Gewichtungsfaktoren W_N^0 und W_N^1 verwendet. Anhand des Zusammenhangs $W_N^0 = (W_N^1)^*$ kann man den Schmetterlingsgraphen optimieren und dadurch eine Multiplikation einsparen.

Somit fallen pro Berechnungsebene nur die Hälfte an Multiplikationen an. Bei den Additionen werden pro Berechnungsebene die Hälfte zu Subtraktionen. Die Zahl der Additionen und Subtraktionen bleibt durch die Optimierung hingegen gleich. Die Berechnungsebenen des Algorithmus können aus der Gesamtblocklänge N berechnet werden. Bei $N = 2^n$ ergeben sich $\log_2(N)$ Ebenen, in denen die Teilberechnungen

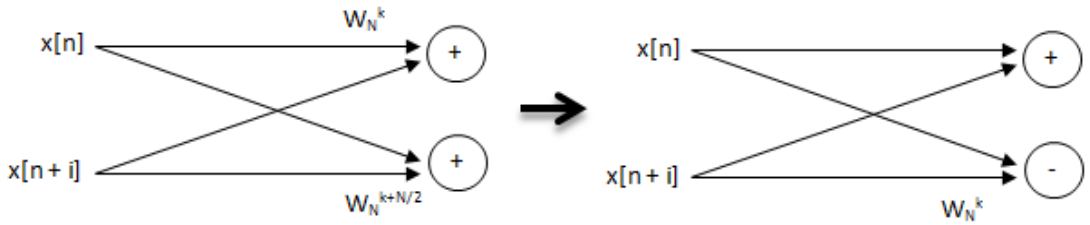

Abbildung 1: Schmetterlingsgraph des Radix-2 Algorithmus: Die Optimierung geschieht durch die Ausnutzung der konjugiert komplexen Gewichtungsfaktoren

durchgeführt werden. Ein Ablaufplan für $N = 8$ mit Bitumkehr am Eingang ist in (Abb. 2) zu sehen. Darin ist auch erkennbar, dass die Gewichtungsfaktoren bei Bitumkehr am Eingang von $360°$ in Richtung $0°$ laufen. Deswegen ist $W_N^2 = -j$ und nicht etwa j, was einem Umlauf von $0°$ in Richtung $360°$ entsprechen würde.

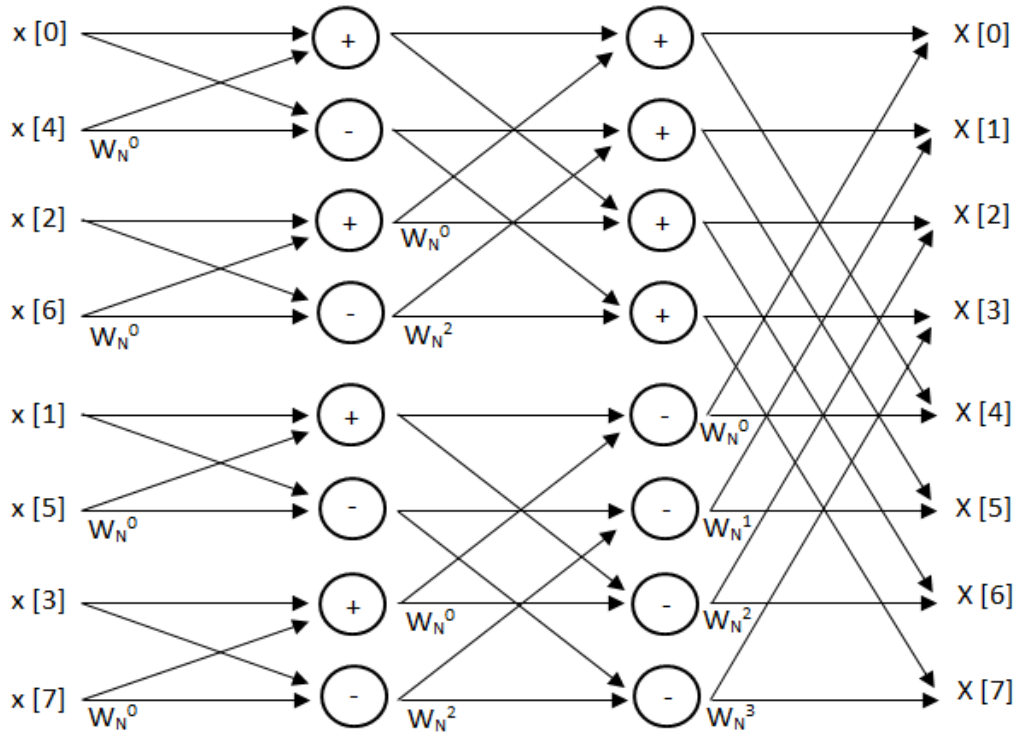

Abbildung 2: Ablaufplan des Radix-2 Algorithmus für $N = 8$

2.2.3 Rechenaufwand des Radix-2 Algorithmus

Der Rechenaufwand für die FFT mit dem Radix-2 Algorithmus wird mit steigendem N im Gegensatz zur DFT gering. Pro Berechnungsebene werden ohne Optimierung N Multiplikationen nötig. Durch die Optimierung der Schmetterlingsgraphen spart man die Hälfte ein, es bleiben $\frac{N}{2}$ Multiplikationen. Die Zahl der Additionen bleibt pro Ebene bei N. Die Zahl der Berechnungsebenen kann aus der Blocklänge N berechnet werden. Bei $N = 2^n$ sind bei der Aufteilung bis auf Einzelwerte genau n Ebenen nötig. Folglich ist die Anzahl der Ebenen $\log_2(N)$. Der Rechenaufwand für den Radix-2 Algorithmus beträgt demnach:

- $\frac{N}{2} \cdot \log_2(N)$ Multiplikationen
- $N \cdot \log_2(N)$ Additionen

2.2.4 Radix-4 FFT Algorithmus

Bei größeren Blocklängen kann statt einer Aufteilung auf einen Schmetterlingsgraphen mit zwei Eingängen auch ein Graph mit vier Eingängen verwendet werden (Abb. 3). Dieses Verfahren wird als Radix-4 FFT bezeichnet und spart weitere Berechnungen ein. Voraussetzung für die Verwendung des Radix-4 Algorithmus ist eine Blocklänge von $N = 4^n$. Der Radix-4 Algorithmus ist in Anwendungen mit größeren Blocklängen

oft der Algorithmus der Wahl, weil dieser im Gegensatz zum Radix-2 noch weitere Berechnungen einsparen kann. Eine entsprechende Blocklänge von 4^n wird wie beim Radix-2 durch zero-padding erreicht, also das Auffüllen des Eingangs mit Nullen bis zur gewünschten Länge.

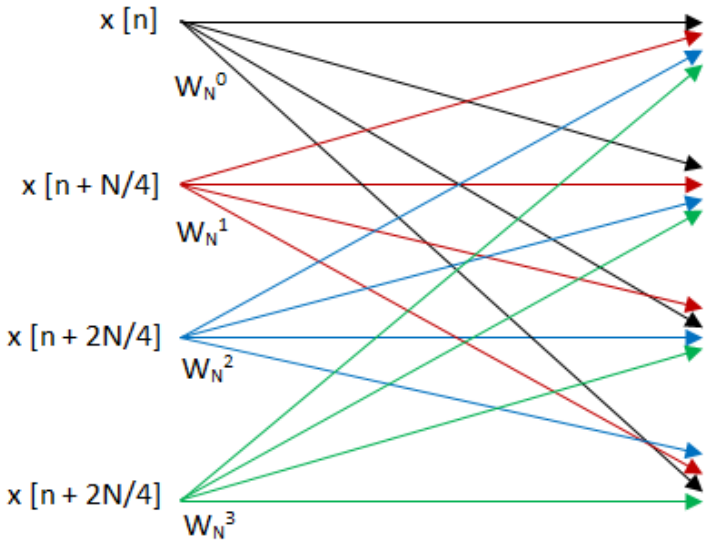

Abbildung 3: Schmetterlingsgraph des Radix-4 Algorithmus

Der Rechenaufwand für den Radix-4 Algorithmus beträgt bei den Multiplikationen 75% des Rechenaufwands des Radix-2 Algorithmus. Es sind pro Rechenebene $\frac{N}{4}$ Schmetterlingsgraphen nötig, jeder enthält drei Multiplikationen. Die Berechnungsebenen sind bei $N = 4^n$ genau halb so viele, wie beim Radix-2 Algorithmus. Dieser Zusammenhang kann auch direkt deutlich gemacht werden, wenn man 4^n als 2^{2^n} schreibt. Es gibt folglich $\log_4(N) = \frac{\log_2(N)}{2}$ Berechnungsebenen, in denen die Operationen durchgeführt werden. Im Bezug auf die Additionen ändert sich beim Radix-4 Algorithmus im Vergleich mit Radix-2 nichts, weil die Anzahl der Knotenpunkte, an denen addiert wird, gleich bleibt. Der Rechenaufwand für den Radix-4 Algorithmus beträgt demnach:

- $\frac{3N}{4} \cdot \frac{\log_2(N)}{2} = \frac{3N}{8} \cdot \log_2(N)$ Multiplikation
- $\frac{8N}{4} \cdot \frac{\log_2(N)}{2} = N \cdot \log_2(N)$ Additionen

2.3 Faltung

2.3.1 Diskrete Faltung

Im diskreten Zeitbereich wird aus dem Faltungsintegral eine Summe, welche das Eingangssignal bei der Berechnung durchläuft (13). Die Rechenoperationen für einen Wert ändern sich dadurch nicht. Der Unterschied besteht darin, dass im analogen, kontinuierlichen Bereich zu jeder Zeit ein Wert vor liegt, der Abstand zwischen zwei Werten ist daher unendlich klein. Im digitalen, diskreten Bereich treffen die Werte des Eingangssignals im Abstand $T_A = \frac{1}{f_A}$ ein. Die Faltungssumme liefert daher für jedes Abtastintervall einen Ausgangswert. Die Problematik des hohen Rechenaufwands einer Faltung eines beliebigen Signals mit einem weiteren beliebigen Signal bleibt auch im diskreten Zeitbereich erhalten. Der Rechenaufwand steigt mit der Blocklänge um N^2 bei Multiplikationen und Additionen. Deswegen ist die diskrete Faltung auch nur bis zu einer Blocklänge von $N = 64$ effektiver als die nachfolgend vorgestellte zyklische Faltung. Dieser Wert hat sich mit der Zeit verändert, was mit steigenden Taktgeschwindigkeiten von Prozessoren zu begründen ist. Gardner nennt in [Gar, 1995] einen Wert von $N = 32$ als Übergang zwischen diskreter Faltung und der zyklischen Faltung, bei [Mey, 2009, S. 204] sind die genannten 64 zu finden.

$$x(t) * h(t) = \int_{-\infty}^{\infty} x(\tau) \cdot h(t-\tau)\, d\tau \longrightarrow x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot h[n-k] \quad (13)$$

Um die diskrete Faltung ausführen zu können, müssen Eingangssignal und Impulsantwort die gleiche Länge haben. Ist dies nicht der Fall, wird durch ein zero-padding eines der Signale auf die Länge des anderen gebracht, bevor die Faltung ausgeführt wird. Das Faltungsergebnis hat bei der diskreten Faltung eine Länge von $L_y = L_h + L_x - 1$ (L_x = Länge Eingangssignal, L_h = Länge Impulsantwort)(14). Durch das zero-padding auf eine gleiche Länge N findet man in der Literatur meistens den Wert $L_y = 2N - 1$.

$$\begin{aligned} x[n] &= [1,2,3]\,; h[n] = [1,2,1]\,; N = 3; L_y = 2N - 1 = 5 \quad (14) \\ y[n] &= x[n] * h[n] = \sum_{k=0}^{2} x[k] \cdot h[n-k] \\ &= [(1\cdot 1), (1\cdot 2 + 2\cdot 1), (1\cdot 1 + 2\cdot 2 + 3\cdot 1), (2\cdot 1 + 3\cdot 2), (3\cdot 1)] \\ &= [1,4,8,8,3] \end{aligned}$$

Der mit N^2 ansteigende Rechenaufwand der diskreten Faltung macht diese für längere Sequenzen unbrauchbar. Wie auch im kontinuierlichen Bereich kann auch im diskreten Bereich der Rechenweg der Faltung durch eine Transformation in den Frequenzbereich erweitert werden. Dieses Verfahren heißt zyklische Faltung.

2.3.2 Zyklische Faltung

Transformiert man das Eingangssignal und die Impulsantwort in den Frequenzbereich, so wird aus der Faltung eine Multiplikation (15). Dieser Umweg erscheint zunächst komplex und rechenintensiv, er ist aber bereits ab den im vorigen Abschnitt erwähnten Blocklängen effizienter und schneller.

$$x[n] * h[n] \circ - \circ X[k] \cdot H[k] \tag{15}$$

Die zyklische Faltung wird im einfachsten Fall direkt über die DFT durchgeführt. Dies ist jedoch auf Grund des Rechenaufwands der DFT nur für kleine Blocklängen sinnvoll. Daher wird fast ausschließlich die FFT verwendet, um die Transformation durchzuführen. Im Bezug auf das Faltungsergebnis muss an dieser Stelle die Transformation in ihrer Länge erweitert werden. Aus den Überlegungen zur FFT geht schon hervor, dass diese nur so viele Spektrallinien liefert wie Eingangswerte übergeben werden. Ein Eingangssignal der Länge L_x würde folglich nur L_x Spektrallinien durch die FFT erhalten. Eine Multiplikation mit einem diskreten Spektrum der gleichen Länge und eine anschließende Rücktransformation in den Zeitbereich liefert ein Ergebnis der Länge L_x, was nicht jenem der diskreten Faltung entspricht. Daher sollte die FFT Blocklänge so gewählt werden, dass genug Spektrallinien vorliegen, um das Faltungsergebnis der Länge $L_y = 2N - 1$ abzubilden. Im Bezug auf den Radix-2 und den ebenfalls vorgestellten Radix-4 Algorithmus sind daher FFT-Blocklängen der nächstgrößeren Potenzen 2^n bzw. 4^n zu wählen (16).

$$\begin{aligned} x[n] &= [1,2,3] \\ h[n] &= [1,2,1] \\ L_{FFT} &= 1 \Longrightarrow y[n] = 1 \\ L_{FFT} &= 2 \Longrightarrow y[n] = [5,4] \\ L_{FFT} &= 3 \Longrightarrow y[n] = [9,7,8] \\ L_{FFT} &= 4 \Longrightarrow y[n] = [4,4,8,8] \\ L_{FFT} &= 5 \Longrightarrow y[n] = [1,4,8,8,3] \end{aligned} \tag{16}$$

Die Rechenersparnis durch die Nutzung der zyklischen Faltung lässt sich sowohl durch praktische Beispiele zeigen, etwa durch Berechnung der Faltung in beiden Varianten auf dem gleichen Computer und Bestimmung der Rechenzeit. In der Theorie kann der Rechenaufwand auch im Voraus ermittelt werden. Sind bei der diskreten Faltung N^2 Multiplikationen und Additionen notwendig, so entstehen bei der zyklischen Faltung mit dem Radix-2 Algorithmus entsprechend die doppelte Anzahl der Operationen für die FFT-Berechnung (s. Abschnitt 2.1.3) und die Multiplikationen im Frequenzbereich. Der Vergleich der benötigten Multiplikationen (als die rechenintensivste Operation) zeigt die Ersparnis bei der zyklischen Faltung (Tab. 2).

Im Bezug auf Echtzeitanwendungen wie zum Beispiel einem Faltungshall in der Audiotechnik entsteht bei der zyklischen Faltung eine Problematik, welche im Folge-

N	lineare Faltung	zyklische Faltung
2	4	24
4	16	56
8	64	128
16	256	288
32	1.024	640
64	4.096	1.408
128	16.384	3.072
256	65.535	6.656
512	262.144	14.336
1024	1.048.576	30.720

Tabelle 2: Vergleich des Rechenaufwands von linearer und zyklischer Faltung

abschnitt gelöst wird. Die FFT kann nur berechnet werden, wenn das gesamte Signal bekannt ist, weil sie auf der gesamten Blocklänge entsprechende Eingangssamples benötigt. Ein Signal mit der Länge von zehn Sekunden kann daher auf normalen Wege mit der zyklischen Faltung erst dann berechnet werden, wenn die gesamten zehn Sekunden Signal vorliegen. Die Latenz des Systems wird hauptsächlich durch die Länge der Signale beeinflusst, nicht etwa durch die Berechnungen. Somit ist es mit der zyklischen Faltung bei Beibehaltung der Gesamtsignale nicht möglich, Echtzeitanwendungen zu realisieren, weil ein als unendlich lang angenommenes Eingangssignal dem System zur Berechnung komplett vorliegen müsste.

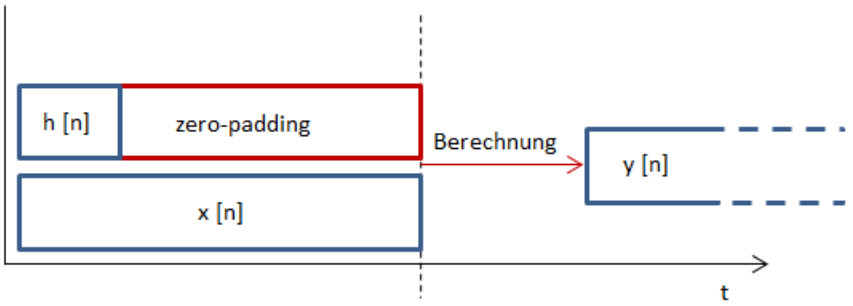

Abbildung 4: Latenz bei der zyklischen Faltung: Das Eingangssignal muss komplett bekannt sein, bevor die Berechnung beginnt

2.3.3 Segmentierte Faltung

Durch die Linearität der Faltungsoperation lässt sich die Berechnung der Faltung in mehrere Berechnungen aufteilen. Dabei muss anschließend das Ergebnis aus den Teilergebnissen durch eine Addition der Einzelwerte berechnet werden. Dies wird erforderlich, weil alle Teilfaltungen die Ergebnislänge $2N-1$ haben und somit überlappen. Zur Berechnung des Faltungsergebnisses gibt es zwei Verfahren, overlap-add (Additionen) und overlap-save (Filterung). Welches Verfahren genutzt wird, hängt von der Anwendung und in vielen Fällen auch der Implementierung ab. Im Rahmen dieser

Thesis werden beide Verfahren vorgestellt, für die Analysen und Programmiertexte wird overlap-add genutzt.

Wird das Eingangssignal einer Faltung unendlich lang, kann es segmentiert werden, um anschließend die bereits erwähnten Teilfaltungen zu betrachten (17). Wird die Impulsantwort nicht segmentiert, so wird diese durch zero-padding auf die Segmentlänge N der Eingangssegmente $x_k[n]$ erweitert [Zöl, 2005, S.171f.].

$$\begin{aligned} x[n] &= [1,2,3,4,4,3,2,1]\,; h[n] = [1,4]\ \ N = 2 \\ x_k[n] &= \left\{ \begin{matrix} x[n]\,, wenn\,(k-1)\,N \leq n \leq kN-1 \\ 0\ sonst \end{matrix} \right\} \\ x_1[n] &= [1,2] \\ x_2[n] &= [3,4] \\ x_3[n] &= [4,3] \\ x_4[n] &= [2,1] \end{aligned} \tag{17}$$

Die Eingangssegmente $x_k[n]$ werden anschließend mit einer FFT der Länge $2N$ in den Frequenzbereich transformiert. Diese Länge ist erforderlich, damit genug Spektrallinien im Spektrum vorliegen, um das Faltungsergebnis der Länge $2N-1$ im Zeitbereich abbilden zu können. N wird je nach verwendetem FFT-Algorithmus gewählt, im Rahmen dieser Thesis mit 2^n zur Verwendung des Radix-2 Algorithmus. Neben den Eingangssegmenten wird auch die Impulsantwort in den Frequenzbereich transformiert (18). Dabei wird die gleiche Blocklänge verwendet. Dies wirkt sich bei einer Optimierung der FFT positiv aus, weil die benötigten Gewichtungsfaktoren gleich sind und somit nur einmal berechnet werden müssen.

$$\begin{aligned} H[f] &= \mathcal{F}\{h[n]\} \\ &= [5; 1-j4; -3; 1+j4] \end{aligned} \tag{18}$$

$$\begin{aligned} X_k[f] &= \mathcal{F}\{x_k[n]\} \\ X_1[f] &= [3; 1-j2; -1; 1+j2] \\ X_2[f] &= [7; 3-j4; -1; 3+j4] \\ X_3[f] &= [7; 4-j3; 1; 4+j3] \\ X_4[f] &= [3; 2-j; 1; 2+j] \end{aligned} \tag{19}$$

Im Frequenzbereich wird nun jedes Eingangssegment mit der Impulsantwort multipliziert, wodurch Teilfaltungsergebnisse im Frequenzbereich entstehen. Diese werden nun per IFFT in den Zeitbereich transformiert. Es liegen nun $\frac{L_x}{N}$ Ergebnissegmente vor. Diese haben alle die Länge $2N$, wobei der letzte Wert Null ist. Somit ist an dieser Stelle wieder die Länge $L_y = 2N-1$ eines Faltungsergebnisses zu sehen (20)

$$
\begin{aligned}
y_1[n] &= x_1[n] * h[n] = [1, 6, 8, 0] \\
y_2[n] &= x_2[n] * h[n] = [3, 16, 16, 0] \\
y_3[n] &= x_3[n] * h[n] = [4, 19, 12, 0] \\
y_4[n] &= x_4[n] * h[n] = [2, 9, 4, 0]
\end{aligned} \tag{20}
$$

Vergleicht man das Ausgangssignal einer diskreten Faltung mit den berechneten Ausgangssegmenten, so ist zu erkennen, dass die ersten Werte des ersten Segments auch im Ausgangssignal wieder auftauchen. Das hängt damit zusammen, dass dieses zunächst nur ein Segment verarbeitet und dessen Faltungsergebnis mit der Impulsantwort ausgibt. Ab dem Zeitpunkt N ist dann auch das zweite Segment des Eingangssignals an der Berechnung beteiligt, somit müssen die Werte des Ausgangssignals aus den Werten mehrerer Ausgangssegmente berechnet werden. An dieser Stelle wird dann eine überlappende Addition (overlap-add) oder Filterung (overlap-save) verwendet, um das Ausgangssignal zu bestimmen (s. Abschnitt 2.3.4 und 2.3.5). In (Abb. 5) ist die Berechnung des Ausgangssignals $y[n]$ mit Hilfe des Overlap-Add Verfahrens zu sehen.

	$\overbrace{}^{N}$								
	1	6	8	0					
+			3	16	16	0			
+					4	19	12	0	
+							2	9	4	0
$y[n]$	1	6	11	16	20	19	14	9	4	0

Abbildung 5: Berechnung des Ausgangssignals $y[n]$ mit dem Overlap-Add Verfahren

Der Vergleich mit dem Ergebnis der diskreten Faltung zeigt, dass lediglich die letzte Stelle nicht vorhanden ist (21). Diese entsteht durch die FFT-Länge, welche mehr Spektrallinien als erforderlich ausgibt. Da diese Stelle aber immer eine Null enthält, wird das eigentliche Ergebnis nicht beeinflusst. Ebenso könnte die Null durch eine entsprechende Anweisung in Programmiertexten abgeschnitten werden.

$$
y_{conv}[n] = [1, 6, 11, 16, 20, 19, 14, 9, 4] \tag{21}
$$

Im Bereich der Audiotechnik erreichen Impulsantworten für Faltungshall meist Längen von über einer Sekunde. Mit dem bereits vorgestellten Mögichkeiten wäre ein Faltungshall für einen Prozess mit einer solchen Impulsantwort nicht echtzeitfähig, weil die Segmentlänge mindestens die Länge der Impulsantwort haben müsste. Somit würde die Latenz des Systems mindestens jener Länge entsprechen, weil zur Berechnung des ersten Ergebnissegments ein Eingangssegment bekannt sein muss. Um die

hohe Latenz zu verringern, kann die Impulsantwort ebenfalls segmentiert werden. Dadurch sind zwar mehr Ausgangssegmente vorhanden und mehr Berechnungen nötig, die Latenz des Systems wird allerdings verringert, weil die Länge eines Eingangssegments sich nun an der Länge eines Segments der Impulsantwort orientiert und nicht mehr an der Gesamtlänge der Impulsantwort. Das Segmentieren der Impulsantwort funktioniert analog zu der Segmentierung des Eingangssignals (22).

$$\begin{aligned} h[n] &= [1,1,2,2,4,4,2,1]; N = 4 \\ h_i[n-(i-1)N] &= \left\{ \begin{matrix} h[n], wenn\,(i-1)N \leq n \leq iN-1 \\ 0\ sonst \end{matrix} \right\} \\ h_1[n] &= [1,1,2,2] \\ h_2[n] &= [4,4,2,1] \end{aligned} \tag{22}$$

Die Impulsantwort wird nun in den Segmenten transformiert. In Folge dessen liegen nun $\frac{L_h}{N}$ Segmente vor, wenn davon ausgegangen wird, dass die Segmentlänge von Eingangssignal und Impulsantwort gleich gewählt wird. Dies muss nicht der Fall sein, es lässt sich allerdings zeigen, dass dieser Fall den effizientesten darstellt. Wenn die Länge des Segments der Impulsantwort kürzer als die Länge eines Segments des Eingangssignals ist, steigt die Anzahl der Berechnungen im Gesamtablauf. Liegen längere Segmente bei der Impulsantwort vor, steigt der Speicheraufwand an.

Nach der FFT eines Eingangssegments wird es nun nicht mehr mit der gesamten Impulsantwort multipliziert, sondern einzeln mit allen Segmenten der Impulsantwort. Anschließend wird per IFFT wieder in den Zeitbereich transformiert. Es liegen nun $L_h \cdot (\frac{L_x}{N})$ Ergebnissegmente vor, welche durch die Überlagerung zum Ausgangssignal $y[n]$ führen. Der Prozess der segmentieren Faltung für k Eingangssegmente ist in Abb.6 zu sehen, ein Rechenbeispiel in Gleichung (23). Dabei ist schon zu erkennen, dass die Impulsantwort nur einmal transformiert werden muss und im gesamten Prozess verwendet werden kann. Dieser Aspekt wird später bei der Umsetzung in Programmiertext wieder aufgegriffen, weil er die Berechnungszeit verringern kann.

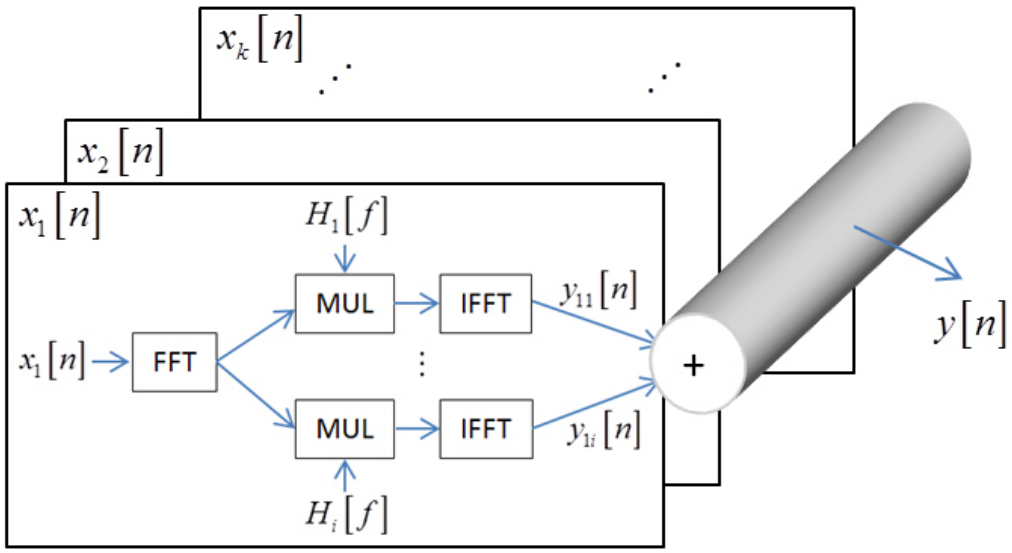

Abbildung 6: Der Prozess der Segmentierten Faltung für k Eingangssegmente

$$
\begin{aligned}
x[n] &= [1,2,3,...,31,32] \\
h[n] &= [1,1,2,2,4,4,2,1] \\
N &= 4
\end{aligned}
\tag{23}
$$

$$
\begin{aligned}
x_k[n] &= \left\{ \begin{matrix} x[n], wenn\ 4(k-1) \leq n \leq 4k-1 \\ 0\ sonst \end{matrix} \right\} \\
x_1[n] &= [1,2,3,4] \\
x_2[n] &= [5,6,7,8] \\
&\vdots \\
x_8[n] &= [29,30,31,32]
\end{aligned}
$$

$$
\begin{aligned}
h_i[n-4(i-1)] &= \left\{ \begin{matrix} h[n], wenn\ 4(i-1) \leq n \leq 4i-1 \\ 0\ sonst \end{matrix} \right\} \\
h_1[n] &= [1,1,2,2] \\
h_2[n] &= [4,4,2,1]
\end{aligned}
$$

$$
\begin{aligned}
y_{ki}[n] &= x_k[n] * h_i[n] = \mathcal{F}^{-1}\{\mathcal{F}(x_k[n]) \cdot \mathcal{F}(h_i[n])\} \\
y_{11}[n] &= [1,3,7,13,14,14,8] \\
y_{12}[n] &= [4,12,22,33,24,11,4] \\
&\vdots \\
y_{82}[n] &= [116,236,302,341,220,95,32]
\end{aligned}
$$

Um aus den Ergebnissegmenten $y_{ki}[n]$ das Ausgangssignal $y_k[n]$ zu berechnen, werden anschließend die bereits erwähnten Verfahren zur Überlagerung dieser Segmente durchgeführt. Diese werden nun im Detail betrachtet und verglichen. Die Grafik in Abb. 7 zeigt die Überlagerung mit Overlap-Add, so wie sie auch im Rahmen der Analysen verwendet werden wird.

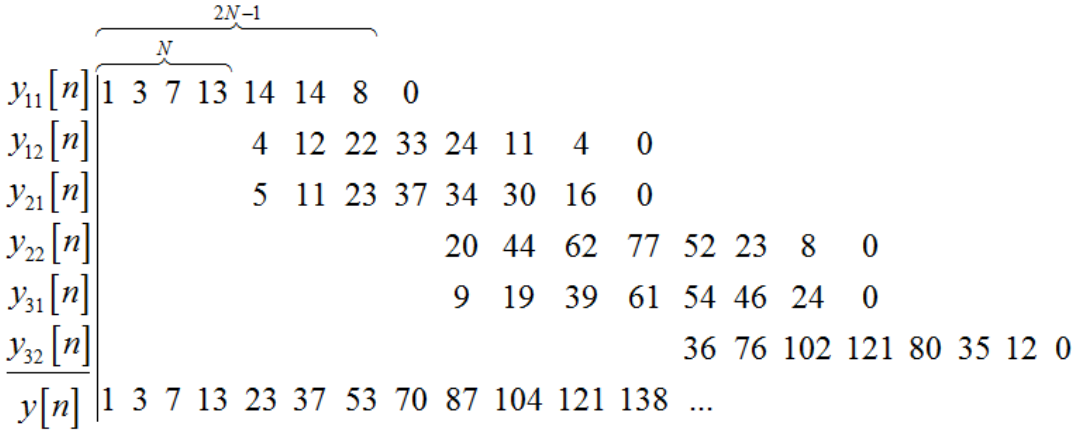

Abbildung 7: Overlap-Add der Ausgangssegmente aus dem Beispiel in (23) zur Berechnung des Ausgangssignals $y[n]$

2.3.4 Overlap-Add

Das Overlap-Add Verfahren ist das einfachere der Überlagerungsverfahren zur Ermittlung des Ausgangssignals. Es beschreibt eine Additionsreihe, deren Elemente aus den Ergebnissegmenten bestehen. Das Eingangssignal wird dabei in Segmente der Länge N aufgeteilt, welche nacheinander in den Prozess laufen. Diese Segmente überlappen einander nicht, somit trifft kein Wert des Eingangssignals doppelt ein. Bei der Impulsantwort wird nach gleichem Prinzip verfahren (Abb. 8). Die Ergebnisse der

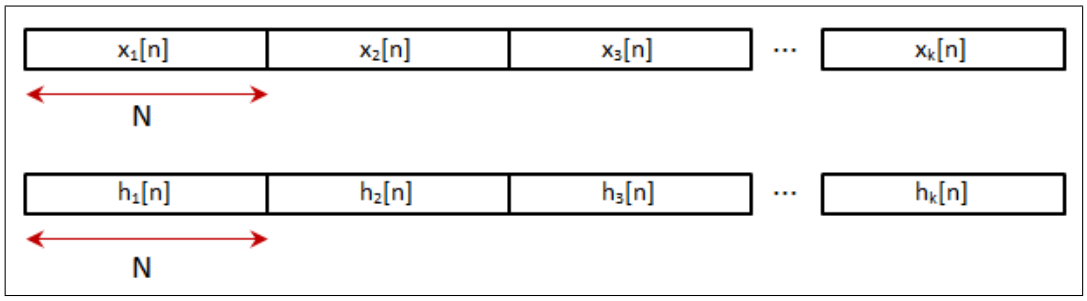

Abbildung 8: Segmentieren von Eingangssignal und Impulsantwort beim Overlap-Add Verfahren

Additionen sind die Werte des Ausgangssignals $y[n]$. Die Ausgangssegmente $y_{ki}[n]$ werden verschachtelt, anschließend erfolgt die Addition der Einzelwerte zu den Werten von $y[n]$. Auf dem ersten Ausgabeabschnitt von $y[1]$ bis $y[N]$ ist keine Addition erforderlich. Auf diesem Abschnitt wird das Faltungsergebnis $y_{11}[n] = x_1[n] * h_1[n]$

ausgegeben. Zu diesem Zeitpunkt ist das erste Segment $x_1[n]$ verarbeitet worden. Dieses wurde mit dem ersten Segment $h_1[n]$ gefaltet und wird direkt ausgegeben. An dieser Stelle ist auch die Latenz des Systems zusehen. Die beschriebene Berechnung kann erst durchgeführt werden, wenn $x_1[n]$ komplett vorliegt. Somit beträgt die Latenz (im Idealfall) die zeitliche Länge von N (je nach Abtastfrequenz), zusätzlich wird ein Rechenaufwand nötig, der mit Fortschreiten der Berechnung ansteigt, bis die maximale Überlagerung erreicht ist. Dies ist genau dann der Fall, wenn genauso viele Segmente des Eingangssignals verarbeitet worden sind, wie es Segmente in der Impulsantwort gibt. Jener Wert $\frac{L_h}{N}$ hat den Hintergrund, dass jedes Segment $x_k[n]$ mit Fortschreiten der Berechnung mit jedem Segment $h_i[n]$ gefaltet werden muss. Ein Segment $x_k[n]$ ist folglich auch an $\frac{L_h}{N}$ Abschnitten der Berechnung beteiligt.

Das jeweils folgende Segment $x_{k+1}[n]$ wird nach dem Eintreffen und Transformieren genauso behandelt wie das Segment $x_k[n]$. Zunächst wird daher wieder mit dem transformierten Impulsantwortsegment $H_1[f]$ multipliziert und in den Zeitbereich zurück transformiert. Gleichzeitig muss noch das Segment $x_k[n]$ aus dem vorigen Durchlauf mit $h_{i+1}[n]$ gefaltet und ausgegeben werden. Es kommt daher zu einer Überlagerung, welche ab dem zweiten Ausgabeabschnitt beginnt, also nach genau N Samples. Die Folgesamples $N+1-2N$ bestehen aus der Summe der Ausgangssegmente $y_{11}[n]$, $y_{12}[n]$ und $y_{21}[n]$. Es folgen weitere N Samples, an deren Werten $y_{12}[n]$, $y_{21}[n]$, $y_{13}[n]$, $y_{22}[n]$ und $y_{31}[n]$ beteiligt sind. Dies setzt sich fort, sodass nach jeweils N Samples immer zwei weitere Segmente hinzukommen, deren Werte in das Ausgangssignal mit eingehen (Abb. 9).

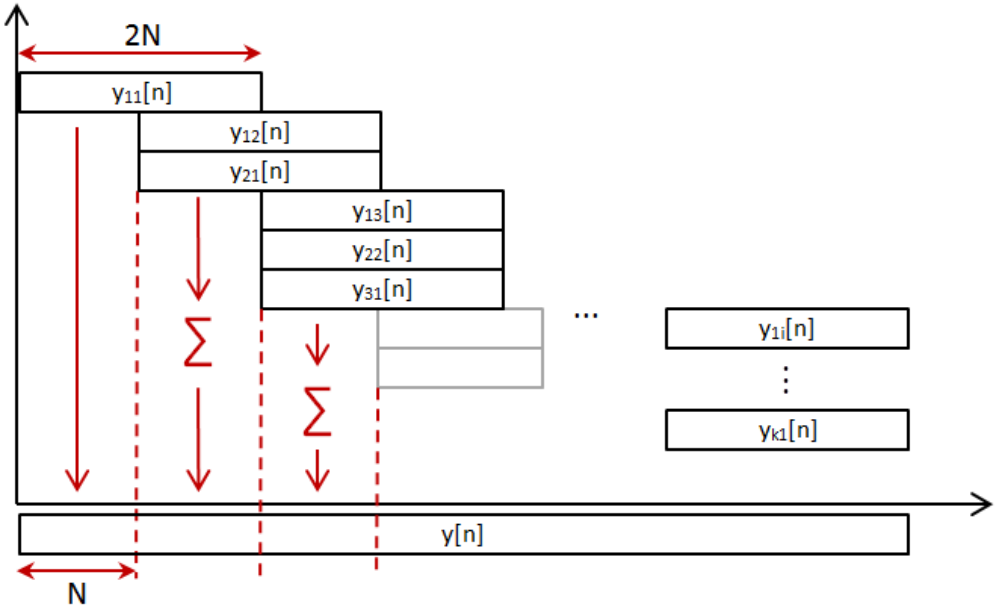

Abbildung 9: Ausgabe des Faltungsergebnisses beim Overlap-Add Verfahren

Auf Grund der begrenzten Impulsantwort kommt es, wie bereits erwähnt, im Verlauf des Prozesses zu einer maximalen Überlagerung von Segmenten. Ab diesem Zeitpunkt sind immer gleich viele Ausgangssegmente an der Berechnung des Ausgangssignals beteiligt. Die Zahl der Additionen zur Berechnung eines Ausgangswertes kann

anhand der Elemente der Impulsantwort, also an der Segmentlänge festgelegt werden. Je kleiner N, desto mehr Elemente in der Impulsantwort und desto höher die Zahl der Berechnungen im Overlap-Add Verfahren. Bei großen Werten für N gibt es entsprechend weniger Segmente der Impulsantwort, also auch weniger überlagernde Segmente und Berechnungen im Overlap-Add Verfahren.

2.3.5 Overlap-Save

Im Gegensatz zum Overlap-Add-Verfahren wird bei Overlap-Save am Eingang überlagert, es treffen daher, im Gegensatz zum Overlap-Add Verfahren, Eingangswerte mehrfach zur Berechnung ein. Dies sorgt dafür, dass die Ausgangssegmente ab einem bestimmten Wert dem Signal $y[n]$ entsprechen und zur Ausgabe des korrekten Ergebnisses an einander gereiht werden können. Durch ein Abschneiden der Ausgangssegmente entstehen direkt die Segmente des Ausgangssignals $y[n]$. Dies kann mit einem Schwingungsvorgang verglichen werden, bei dem Ein- und Ausschwingen ausgeblendet werden. Der relevante Teil des Segments ist jeweils von der Länge der Segmente der Impulsantwort abhängig.

Zum Vergleich der beiden Verfahren wird bei dieser Betrachtung weiterhin davon ausgegangen, dass sowohl Impulsantwort als auch Eingangssignal mit der gleichen Blocklänge N segmentiert werden. Dies ist beim Overlap-Add Verfahren der effizienteste Fall, welcher auch in den weiteren Betrachtungen und Analysen dieser Thesis genutzt wird. Deswegen wird mit den gleichen Werten das Overlap-Save Verfahren gezeigt, um einen direkten Vergleich möglich zu machen.

Am Eingang werden zunächst $N-1$ Samples mit Nullen dem Eingangssignal vorausgeschickt. Dies ist nötig, damit im Ergebnis eben jener Teil des ersten Segments keine Ergebnisse für $y[n]$ enthält, das Aliasing zu Beginn der Ausgabe nicht entsteht. Es folgen die Samples $x_1[1] - x_1[N-1]$. Für die Folgesegmente wird entsprechend vorgegangen (24) (Abb.10).

$$\begin{aligned} x[n] &= [x_1, x_2, x_3, x_4, ..., x_n] \\ N &= 4 \\ x_1[n] &= [0, 0, 0, 0, x_1, x_2, x_3, x_4] \\ x_2[n] &= [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8] \\ &\vdots \\ x_k[n] &= [x_{k-7}, x_{k-6}, x_{k-5}, x_{k-4}, x_{k-3}, x_{k-2}, x_{k-1}, x_k] \end{aligned} \tag{24}$$

Die Segmentlänge N entscheidet in diesem Fall nicht über die Segmentlänge der Berechnung, sondern um den relevanten Teil eines Segments. Werden Impulsantwort und Eingangssignal mit der gleichen Länge N segmentiert, entstehen wie in (24) zu sehen, Segmente $x_k[n]$ der Länge $2N$. Im Bezug auf die Segmentierung der Impulsantwort wird in diesem Beispiel zunächst davon ausgegangen, dass diese nur aus einem Segment besteht, welches durch zero-padding auf die Länge $2N$ der Segmente $x_k[n]$ zu bringen. Um das Ausgangssignal $y[n]$ zu bestimmen, werden die Berechnungen

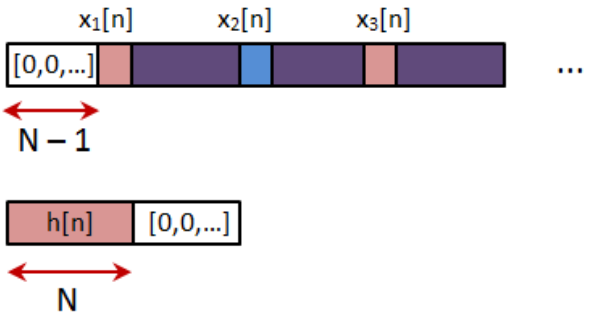

Abbildung 10: Segmentieren des Eingangssignals beim Overlap-Save Verfahren (die Impulsantwort besitzt hier nur ein Segment)

der Ausgangssegmente auf die selbe Art und Weise wie auch beim Overlap-Add-Verfahren durchgeführt. Der Kern des Verfahrens der segmentierten Faltung bleibt somit erhalten, nur die abschließende Berechnung des Endergebnisses ändert sich. Die Veränderung der Eingabesegmente $x_k[n]$ beim Overlap-Save-Verfahren führt bei den Berechnungen innerhalb der segmentierten Faltung zu keinerlei Veränderungen. Lediglich die Segmentierungslänge ist verändert, dabei sollte anhand der genutzten FFT-Algorithmen die Vorgabe der möglichen Blocklängen eingehalten werden.

Die Ausgangssegmente des Overlap-Save-Verfahrens besitzen zu viele Informationen. Aus Grund der Überlappung müssen diese zur Ermittlung des Ausgangssignals wieder entfernt werden. Das Beispiel in (25) zeigt, welche Abschnitte der Ausgangssegmente verwendet werden. Dabei ist das Faltungsergebnis zuvor bereits mit diskreter Faltung berechnet worden, um einen Vergleich mit dem Overlap-Save Verfahren zu ermöglichen. Die in den Ergebnissegmenten relevanten Werte sind im Beispiel fett markiert.

$$
\begin{aligned}
x[n] &= [1,2,1,2,1,2,1,2,0,1,0,1,2,1,0] \\
h[n] &= [1,4] \\
y[n] &= x[n] * h[n] = [1,6,9,6,9,6,9,6,8,1,4,1,6,9,4,1,4]
\end{aligned}
\tag{25}
$$

$$
\begin{aligned}
x_1[n] &= [0,0,0,0,1,2,1,2] \\
x_2[n] &= [1,2,1,2,1,2,1,2] \\
x_3[n] &= [1,2,1,2,0,1,0,1]
\end{aligned}
$$

$$
\begin{aligned}
y_1[n] &= [8,0,0,0,\mathbf{1},\mathbf{6},\mathbf{9},\mathbf{6}] \\
y_2[n] &= [9,6,9,6,\mathbf{9},\mathbf{6},\mathbf{9},\mathbf{6}] \\
y_3[n] &= [5,6,9,6,\mathbf{8},\mathbf{1},\mathbf{4},\mathbf{1}]
\end{aligned}
$$

Das Entfernen der Samples in den Segmenten kommt einer Fensterung nah, deswegen wird Overlap-Save auch oft als Filterung bezeichnet. Welches Verfahren genutzt wird, hängt in den meisten Fällen von der Anwendung ab. Viele Verfahren nutzen

mittlerweile Overlap-Save, weil die Filterung einfach umzusetzen ist und schnell zu berechnen ist. Dabei werden Additionen eingespart und der Rechenaufwand sinkt. Bei Segmentierung der Impulsantwort wird die Anwendung von Overlap-Save jedoch problematisch. Zwar können hier die Überlagerungsbereiche eingespart werden, es müssen aber die durch die Segmentierung der Impulsantwort entstandenen Segmente dennoch addiert werden. Dieses Verfahren kann jene Additionen nicht ersetzen, somit müsste

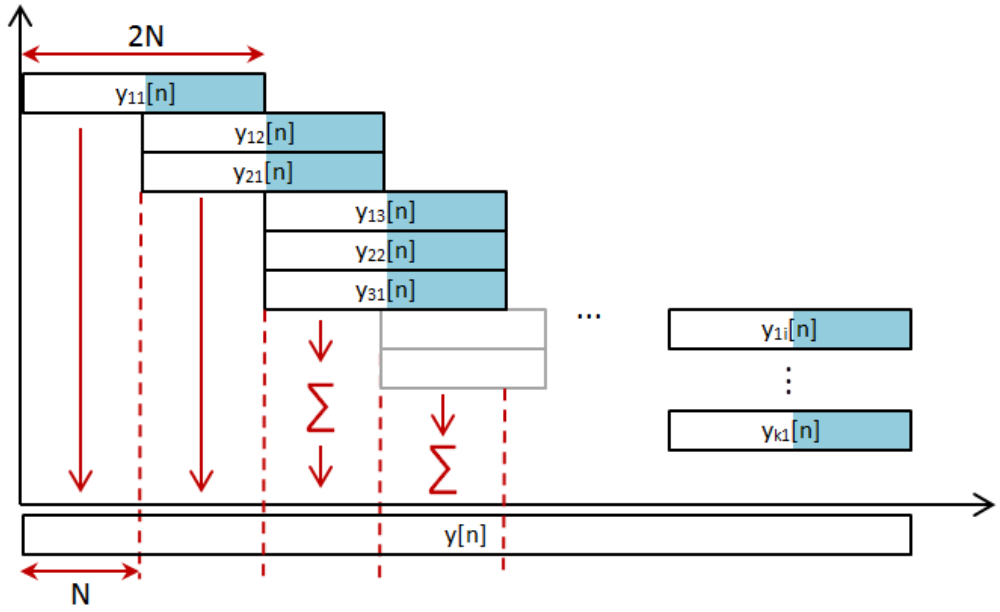

Abbildung 11: Overlap-Save bei Segmentierung von Impulsantwort und Eingangssignal. Die jeweils revelanten Segmentanteile sind in blau eingefärbt.

eine Kombination aus beiden Verfahren genutzt werden (Abb. 11), welche einen höheren Aufwand bei der Programmierung des Prozesses zur Folge hätte. Im Rahmen dieser Thesis und den zugehörigen Analysen wird daher Overlap-Add genutzt.

3 Arbeitsaufwand der segmentierten Faltung

Die Anzahl der Berechnungen des Prozesses für eine vorgegebene Segmentlänge N ermöglicht eine Analyse, welche für spätere Untersuchungen als Anhaltspunkt dient. Zusätzlich ist der Arbeitsaufwand, wenn er in FLOPS (floating point operations) berechnet wird, mit nahezu allen Prozessoren vergleichbar. Dadurch lassen sich bereits im Voraus mögliche Implementierungen auf Prozessoren abschätzen. Ein entscheidendes Kriterium im Bezug auf die segmentierte Faltung ist dabei die genutzte FFT. Abgesehen vom Algorithmus kann, durch eine Vorberechnung der Gewichtungsfaktoren, Rechenleistung eingespart werden. Bei der Programmierung des Prozesses ist diese Vorberechnung in den meisten Fällen in einer Bibliothek ausgelagert, dem Programmierer daher normalerweise nicht sichtbar. Es ist daher erforderlich, die FFT-Bibliothek vor dem Einbinden in den Prozess zu überprüfen, ob eine entsprechende Vorberechnung der Gewichtungsfaktoren vorgesehen ist oder nicht.

3.1 Variablenbenennung

Um in der Thesis ein Durcheinander von Variablen zu vermeiden, werden an dieser Stelle einheitliche Variablen für die Abläufe eingeführt, welche für die gesamte weitere Thesis verwendet werden. Die Zuordnung in Formeln ist dann anhand der hier eingeführten Variablen möglich (Tab. 3).

$x\,[n]$	Eingangssignal
L_x	Länge von $x\,[n]$
$h\,[n]$	Impulsantwort
L_h	Länge von $h\,[n]$
$y\,[n]$	Ausgangssignal
L_y	Länge von $y\,[n]$
N	Segmentlänge für $x\,[n]$ und $h\,[n]$
k	Index der Segmente von $x\,[n]$
i	Index der Segmente von $h\,[n]$

Tabelle 3: Variablenzuordnung der Thesis

Bei Transformation in den Frequenzbereich werden Großbuchstaben sowie die Variable f für die diskreten Frequenzen verwendet.

3.2 Umrechnung in FLOPS

Grundsätzlich wird eine reelle Addition als eine FLOP gezählt, ebenso eine reelle Multiplikation. Bei der FFT werden allerdings komplexe Werte entstehen, somit sind komplexe Berechnungen nötig, welche auch entsprechend betrachtet werden müssen. Dazu werden die Addition und die Multiplikation einer komplexen Zahl im Detail betrachtet [Chu, S. 15].

Addiert man zwei komplexe Zahlen, so werden die Real- und Imaginärteile addiert (26), es entstehen dabei zwei reelle Additionen.

$$z_1 + z_2 = (a + jb) + (c + jd) = (a + c) + j\,(b + d) \tag{26}$$

Die Multiplikation zweier komplexer Zahlen besteht aus der Multiplikation aller Real- und Imaginärteile mit einander und anschließender Additionen der neu entstandenen Real- und Imaginärteile (27).

$$z_1 \cdot z_2 = (a + jb) \cdot (c + jd) = (a \cdot c - b \cdot d) + j\,(a \cdot d + b \cdot c) \tag{27}$$

Es entstehen zwei reelle Additionen und vier reelle Multiplikationen, welche als eine komplexe Multiplikation gezählt werden. Umformungen der Gleichung 27 können eine komplexe Multiplikation auch als drei reelle Multiplikationen und sechs reelle Additionen darstellen. Diese Umformung entstand für ältere Rechner, bei denen die Rechenzeit für eine Multiplikation noch wesentlich höher war als jene für eine Addition. In der heutigen Zeit liefern Benchmarks für Prozessoren oft Werte, welche sich im Mikrosekundenbereich für Additionen und Multiplikationen unterscheiden. In [Chu, S. 16] findet sich eine Umformung auf drei Multiplikationen und drei Additionen, welche möglich ist, wenn die Gewichtungsfaktoren vorberechnet sind (28).

$$\begin{aligned}
\lambda &= z_1 \cdot z_2 = (a + jb) \cdot (c + jd) \\
W_N^k &= c + jd \\
\delta &= d + c \\
\gamma &= d - c \\
\\
u_1 &= (a + b) \cdot c \\
u_2 &= \delta \cdot b \\
u_3 &= \gamma \cdot a \\
\\
\Re(\lambda) &= u_1 - u_2 = (a + b) \cdot c - \delta \cdot b = (a + b) \cdot c - (d + c) \cdot b \\
&= a \cdot c - b \cdot d \\
\Im(\lambda) &= u_1 + u_3 = (a + b) \cdot c + \gamma \cdot a = (a + b) \cdot c + (d - c) \cdot a \\
&= b \cdot c + a \cdot d
\end{aligned} \tag{28}$$

Diese Umformung sorgt aber auf Grund der bereits genannten Berechnungszeiten heutiger Systeme für so kleine Veränderung, dass diese eher einem Detail in einer FFT-Bibliothek entspricht. Die Zahl der FLOPS ändert sich durch diese Umformung nicht, sie bleibt bei sechs FLOPS, weswegen dieser Wert auch als jener für die komplexe Multiplikation angenommen wird.

3.3 Aufteilung in Teilprozesse zur Analyse des Arbeitsaufwands

Der Prozess der segmentierten Faltung lässt sich in mehrere Teilprozesse aufteilen, welche getrennt voneinander analysiert werden können. Das Eingangssignal wird dabei als endlich angenommen, um Abschätzungen zum Arbeitsaufwand durchführen zu können. Zur Aufteilung wird als Grundlage der Ablauf der Berechnungen für ein Segment $x_k[n]$ untersucht, welches zu einem beliebigen Zeitpunkt im System eintrifft und bearbeitet werden soll. Ein solches Segment wird zunächst in den Frequenzbereich transformiert, was eine FFT der Länge $2N$ benötigt. Eine Einschränkung des Wertes für N wird anhand von FFT-Algorithmen nötig. Im Falle des Radix-2 Algorithmus wird $N = 2^n$ angenommen, um den Vorgaben des Algorithmus zu entsprechen. Die Blocklänge der FFT wiederum hängt mit der Faltung zusammen. Die FFT muss mindestens so aufgelöst sein, dass die Blocklänge mindestens der Länge des Faltungsergebnisses $x[n] * h[n]$ entspricht. Diese Länge von $y[n]$ beträgt $L_y = L_x + L_h - 1$, für ein Segment $y_{ki}[n]$ beträgt diese $L_{yki} = N + N - 1 = 2N - 1$. Mit der Blocklänge $2N$ wird also sowohl den Vorgaben des Algorithmus als auch denen der Faltung entsprochen.

Ist das Segment $x_k[n]$ in den Frequenzbereich transformiert, liegt das Segment $X_k[f]$ vor, welches anschließend mit allen Segmenten $H_i[f]$ multipliziert wird. Diese Multiplikationen sind komplex, deshalb gelten die bereits vorgestellten Prinzipen zur Analyse. Als Ergebnisse dieser Multiplikationen liegen die Segmente $Y_{ki}[f]$ vor. Diese müssen anschließend wieder in den Zeitbereich transformiert werden. Dazu verwendet werden wieder die FFTs der Blocklänge $2N$, welche auch schon zu Beginn der Verarbeitung des Segments verwendet wurden. Als Ergebnis erhält man die Ausgangssegmente $y_{ki}[n]$, aus denen per Overlap-Add das Faltungsergebnis $y[n]$ berechnet wird. Der Prozess teilt sich somit in vier einzelne Analyseschritte auf:

- FFT des Eingangssegments $x_k[n]$
- Multiplikation der Segments $X_k[f]$ mit den Segmenten $H_i[f]$
- IFFT der Segmente $Y_{ki}[f]$
- Overlap-Add der Segmente $y_{ki}[n]$

3.3.1 FFT der Eingangssegmente

Die FFT der Eingangssegmente $x_k[n]$ mit dem Radix-2 Algorithmus mit der Blocklänge N kann beim Rechenaufwand für ein Segment bestimmt werden. Anschließend kann die Analyse des Arbeitsaufwands mit einem Faktor $\frac{L_x}{N}$ auf alle Segmente des untersuchten Signals $x[n]$ zurückgeführt werden. Die FFT hat mit dem Radix-2 Algorithmus einen Rechenaufwand von $\frac{N}{2} \cdot \log_2(N)$ Multiplikationen und $N \cdot \log_2(N)$ Additionen. Dieser wird nun auf die Blocklänge $2N$ bezogen und in FLOPS überführt.

$$\begin{aligned} \text{Multiplikationen} \quad &: \quad \frac{N}{2} \cdot \log_2(N) \longrightarrow N \cdot \log_2(2N) \\ \text{Additionen} \quad &: \quad N \cdot \log_2(N) \longrightarrow 2N \cdot \log_2(2N) \end{aligned} \tag{29}$$

Die Logarithmusfunktion bestimmt die Anzahl der Ebenen des Verfahrens, in diesem Fall wird pro Ebene um den Faktor zwei bei den Berechnungslängen der Teil-DFTs verringert. Die Umrechnung in FLOPS kann daher anhand der Faktoren vor der Logarithmusfunktion erfolgen. Auch wenn im Audiobereich nur reelle Eingangswerte zu erwarten sind, wird in der Analyse von komplexen Werten ausgegangen. Innerhalb der Berechnung sind bereits ab der zweiten Ebene des Radix-2 Algorithmus komplexe Werte beteiligt, bedingt durch die Gewichtungsfaktoren, welche ab dieser Ebene die imaginäre Einheit aufweisen. Dies wird in der Analyse auf alle Werte bezogen. Der dadurch entstehende Mehraufwand sorgt zwar für höhere Endergebnisse bei der Analyse, gleichen dadurch aber einen möglichen Overhead aus, welchen ein Prozessor mit sich bringen könnte. Die Ergebnisse der Analyse nähern daher eher ein worst-case Szenario an, als den optimierten Weg. Somit sind diese anschließend auch besser mit den Auswertungen eines Programmiertextes zu vergleichen.

Die Umformung in FLOPS wird anhand der Betrachtungen in Abschnitt 3.2 durchgeführt, wonach eine komplexe Multiplikation sechs FLOPS entspricht und eine komplexe Addition mit zwei FLOPS angegeben wird.

$$\begin{aligned}
\text{Multiplikationen} \quad &: \quad \frac{N}{2} \cdot \log_2(N) \longrightarrow N \cdot \log_2(2N) \\
&\longrightarrow \quad 6N \cdot \log_2(2N)\, FLOPS \\
\\
\text{Additionen} \quad &: \quad N \cdot \log_2(N) \longrightarrow 2N \cdot \log_2(2N) \\
&\longrightarrow \quad 4N \cdot \log_2(2N)\, FLOPS \\
\\
W_{FFT} \quad &= \quad 10N \cdot \log_2(2N)\, FLOPS
\end{aligned} \tag{30}$$

3.3.2 Multiplikation der Segmente $X_k[f]$ mit den Segmenten $H_i[f]$

Für jedes Eingangssegment muss nach der Transformation eine Multiplikation mit allen Segmenten $H_i[f]$ berechnet werden. Diese ist im Audiobereich auf Grund der zuvor sowohl im Eingangssignal als auch in der Impulsantwort durchgeführten Transformation in den Frequenzbereich als komplex anzunehmen. Reelle Multiplikationen könnten bei anderen Anwendungen auftreten, diese würden dann aber, genau wie zuvor bei der FFT, einen Schritt weg vom worst-case Fall bedeuten, welcher hier ermittelt werden soll. Die Anzahl der durchzuführenden Multiplikationen hängt von der Segmentlänge N ab, welche darüber entscheidet, wie viele Segmente die Impulsantwort hat. Die Anzahl der Segmente der Impulsantwort ist $\frac{L_h}{N}$, somit sind pro Segment $X_k[f]$ genau $\frac{L_h}{N}$ Multiplikationsabläufe nötig. Da es sich um eine elementweise Multiplikation handelt, hat jeder der angesprochenen Abläufe eine Länge von $2N$, entsprechend der Länge eines transformierten Segments $X_k[f]$. Es müssen pro Ablauf somit $2N$ komplexe Multiplikationen durchgeführt werden, für den gesamten

Ablauf ergeben sich die Werte in (31).

$$\begin{aligned} \text{Multiplikationen} \quad &: \quad 2N \cdot \frac{L_h}{N} \longrightarrow 12 \cdot \frac{L_h}{N} FLOPS \\ W_{MULT} \quad &= \quad 12 \cdot \frac{L_h}{N} FLOPS \end{aligned} \tag{31}$$

3.3.3 IFFT der Segmente $Y_{ki}[f]$

Die Rücktransformation der Segmente $Y_{ki}[f]$ in den Zeitbereich verhält sich wie die FFT zur Transformation der Segmente $x_k[n]$ in den Frequenzbereich (vgl. Abschnitt 1.1.2). Die Umformungen zur Berechnung der IFFT mit Hilfe der FFT erfordern ebenfalls Prozessorzeit, diese kann aber nicht, wie im Falle der Rechenoperationen, direkt in FLOPS übersetzt werden. Deswegen wird in dieser Analyse die Vertauschung von Real- und Imaginärteil nicht als zusätzlicher Rechenaufwand mit aufgefasst. Erreicht wird die Vertauschung im Normalfall durch entsprechende Programmierung. Die Werte des Real- und Imaginärteils liegen getrennt im Speicher, somit kann durch entsprechende Anweisungen entweder Realteil oder Imaginärteil der genutzten Werte in die Berechnung einbezogen werden. Das Vertauschen spielt sich daher eher im Programmiertext, weniger in der Prozessorarchitektur ab und ist somit in der benötigten Prozessorzeit verschwindend gering. Die IFFT zur Rücktransformation der Segmente $Y_{ki}[f]$ in den Zeitbereich wird daher mit den gleichen Werten in die Analyse einbezogen wie die FFT der Segmente $x_k[n]$. Pro Segment xk[n] entstehen Lh/N Ausgangssegmente. Der Rechenaufwand für die IFFT muss daher um den entsprechenden Faktor erweitert werden (32).

$$\begin{aligned} \text{Multiplikationen} \quad &: \quad \frac{L_h}{\not{N}} \cdot \frac{\not{N}}{2} \cdot \log_2(N) \longrightarrow L_h \cdot \log_2(2N) \\ &\longrightarrow \quad 6 \cdot L_h \cdot \log_2(2N)\, FLOPS \\ \text{Additionen} \quad &: \quad \frac{L_h}{\not{N}} \cdot \not{N} \cdot \log_2(N) \longrightarrow 2L_h \cdot \log_2(2N) \\ &\longrightarrow \quad 4 \cdot L_h \cdot \log_2(2N)\, FLOPS \\ W_{IFFT} \quad &= \quad 10 \cdot L_h \cdot \log_2(2N)\, FLOPS \end{aligned} \tag{32}$$

3.3.4 Overlap-Add Verfahren

Die Rechenoperationen des Overlap-Add-Verfahrens sind Additionen, welche im Audiobereich reell sind. Genau wie bei den vorigen Betrachtungen wird aber komplex gerechnet, um eine möglichst allgemeine und nicht zu fokussierte Analyse zu erhalten. Die Anzahl der Summen des Overlap-Add Verfahrens kann bereits im Voraus ermittelt werden, wenn die Länge des Eingangssignals und die Länge der Impulsantwort bekannt sind. Diese entspricht dann dem Faltungsergebnis $L_y = L_x + L_h - 1$,

von welchem sowohl am Anfang und am Ende jeweils ein halbes Ergebnissegment, also die Segmentlänge abgezogen wird. Die Anzahl der zu berechnenden Summen kann somit angegeben werden (33).

$$A_{Sum} = L_x + L_h - 2N - 1 \tag{33}$$

Die Summen enthalten jedoch nicht immer gleich viele Summanden. Dies könnte zwar umgesetzt werden, jedoch müsste man dann den Speicher immer an allen nicht verwendeten Stellen mit Nullen füllen, was zusätzlichen Aufwand bedeuten würde. Deswegen geht die Analyse davon aus, dass immer nur jene Segmente $y_{ki}[n]$ am Overlap-Add beteiligt sind, welche auch Teil des aktuell berechneten Ergebnissegments sind. Zur genauen Analyse wurde im Rahmen der Thesis eine Formel entwickelt, mit der sich die Anzahl der Additionen genau berechnen lässt, ebenso ist der zunächst geschilderte Fall mit immer gleich vielen Summanden beschrieben und lässt sich annähern.

Das Overlap-Add Verfahren teilt sich in zwei Bereiche, welche getrennt voneinander untersucht wurden. Zum einen gibt es den Bereich mit maximaler Überlagerung. Dieser ist genau dann vorhanden, wenn es mehr Segmente $x_k[n]$ des Eingangssignals als Segmente $h_i[n]$ der Impulsantwort gibt. Es entsteht dann ein Bereich im Overlap-Add Verfahren, in dem $2 \cdot \frac{L_h}{N}$ Segmente überlagert sind. Größer als dieser Wert kann die Zahl der überlagerten Segmente nicht werden, deswegen bleibt der Wert ab diesem Zeitpunkt gleich, bis das Ende der Berechnung den Wert der überlagerten Segmente wieder senkt. Der Bereich mit maximaler Überlagerung wird an beiden Seiten durch einen gleichen Ablauf begrenzt. In diesem sind Beginn und Ende der Berechnung enthalten, also jene Bereiche, in denen die Zahl der überlagerten Segmente nicht konstant bleibt. Durch den Aufbau des Verfahrens lassen sich diese Bereiche ebenfalls genau erfassen. Im Abb. 12 ist ein schematischer Ablauf einer Ergebnisberechnung mit Overlap-Add-Verfahren gezeigt. Die Blöcke sind dabei bewusst kurz gewählt, um das Verfahren komplett abbilden zu können.

Sieht man sich nun einen Bereich mit maximaler Überlagerung an, enthält dieser $2 \cdot \frac{L_h}{N}$ Segmente. Um ein Ergebnissample bei dieser Zahl von Segmenten zu berechnen, sind entsprechend $2 \cdot \frac{L_h}{N} - 1$ Additionen nötig. Ein Überlappungsabschnitt enthält genau N Ergebnissamples, somit kann die Zahl der Additionen für einen Abschnitt bereits angegeben werden (34).

$$W_{OV1} = \left(\frac{2 \cdot L_h}{N} - 1 \right) \cdot N = 2 \cdot L_h - N \tag{34}$$

Die Anzahl der Bereiche, in denen die maximale Überlagerung vorliegt, kann aus der Gesamtzahl aller Additionsbereiche hergeleitet werden. Da diese Herleitung an den Beginn und das Ende der Berechnung angelehnt ist, wird jener Ablauf zunächst analysiert, bevor die zugehörigen Mengen an Additionsbereichen pro Ablauf hergeleitet werden. Die erste Ausgabe des Overlap-Add Verfahrens im Bereich $y[0] - y[N-1]$ ist ohne Berechnungen möglich. In diesem Bereich wird das Segment $y_{11}[n]$ ausgegeben. Anschließend werden zwei weitere Segmente aufgerufen und es beginnt der zweite

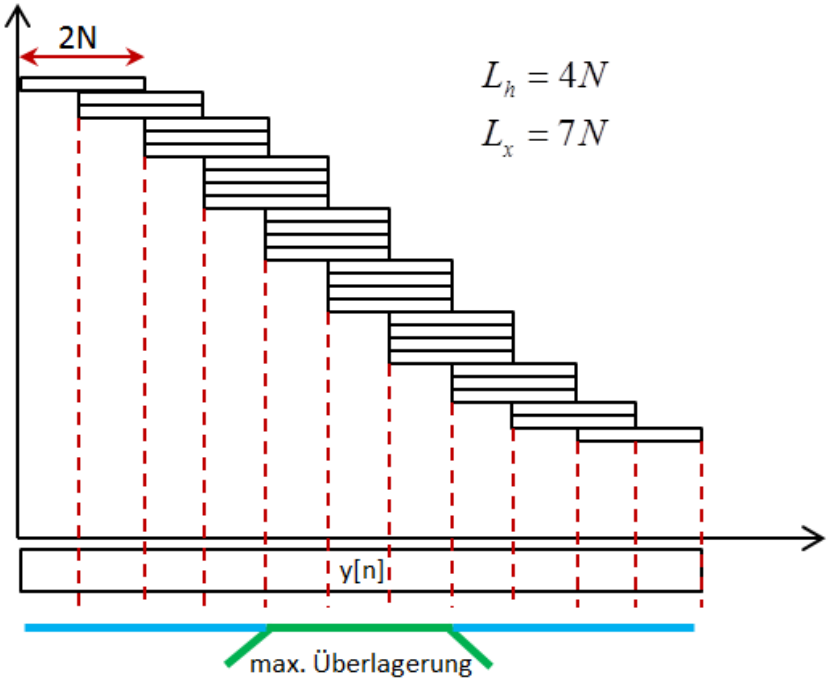

Abbildung 12: Schematischer Komplettablauf einer Overlap-Add Berechnung

Abschnitt, welcher die Werte $y\,[N]$ bis $y\,[2N-1]$ berechnet. Im dritten Abschnitt ist die Ausgabe von Segment $y_{11}\,[n]$ beendet und es werden drei neue Segmente geladen. Pro Additionsabschnitt wird immer ein Segment mehr geladen, bis die neu geladenen Segmente den Wert $\frac{L_h}{N}$ erreichen und somit die maximale Überlagerung vorliegt.

In einem Additionsbereich liegen immer genau N Ergebnissamples, unabhängig davon, wo sich dieser in der Ausgabe befindet. Somit ist nur noch wichtig, wie viele Additionen eine Summe in einem der Bereiche enthält. Anhand der vorigen Betrachtungen kann für die Abschnitte ein Zusammenhang hergestellt werden. Der zweite Abschnitt enthält drei Segmente, somit müssen zwei Additionen pro Ausgangssample berechnet werden (Abb. 12).

Im dritten Additionsbereich sind fünf Segmente beteiligt, entsprechend sind vier Additionen notwendig, um die Ergebnissamples zu berechnen. Die Zahl der Additionen steigt pro Bereich um zwei je Sample an, somit bietet sich eine Summe zur Darstellung aller Bereiche an, welche bis zu jenem mit maximaler Überlagerung zählt. Der Laufindex der Summe entspricht der aktuellen Bereichsnummer. Die obere Grenze lässt sich durch den Zusammenhang zwischen der Zahl der Segmente der Impulsantwort $\frac{L_h}{N}$ und der maximalen Überlagerung ermitteln. Wird der $\frac{L_h}{N}$-te Bereich ausgegeben, wird darin das Segment $y_{1i}\,[n]$einbezogen. Dies ist das letzte Segment in der Ausgabe, an dessen Berechnung das erste Segment $x_1\,[n]$ des Eingangssignals beteiligt ist. Im nächsten Abschnitt wird daher nur ein Segment $y_{k1}\,[n]$ mit einbezogen, durch das weggefallene $y_{1i}\,[n]$ stagniert so die Zahl der Segmente im Overlap-Add Verfahren, die maximale Überlagerung ist erreicht (Abb. 12). Die Summe für den Anfangs- und Endbereich muss daher bis $\frac{L_h}{N}$ laufen. Um den Index der Summe mit einem eher typischen Wert von $m = 1$ zu initialisieren, wird entsprechend die obere Grenze angepasst. Der Inhalt

der Summe erschließt sich aus den bisherigen Untersuchungen. In einem Bereich m gibt es genau $2m$ Additionen pro Ausgangssample. Die Bereichslänge entspricht der Segmentlänge N, somit ist die Zahl der Additionen für den Beginn und das Ende der Berechnung mit der Summe in (35) erfasst.

$$W_{OV2} = 2 \cdot \sum_{m=1}^{\frac{L_h}{N}-1} 2m \tag{35}$$

$$\begin{aligned} \frac{L_h}{N} &= 4;\ 1 \leq m \leq 3 \\ W_{OV2} &= 2 \cdot (2+4+6) = 24 \end{aligned}$$

Die Gesamtzahl der Bereiche, in denen Additionen berechnet werden müssen, muss jetzt aufgeteilt werden, woraus die Faktoren für die einzelnen Bereiche ermittelt werden können. Insgesamt gibt es $\frac{L_x+L_h}{N}$ Bereiche, wovon zwei Bereiche abgezogen werden, in denen jeweils die Direktausgabe eines einzelnen Segments erfolgt. Somit verbleiben $\frac{L_x+L_h-2N}{N}$ Bereiche mit Additionen. Anhand der aufgestellten Summe lässt sich über den Laufindex die Zahl der Bereiche ohne maximale Überlagerung feststellen, welche $2 \cdot \left(\frac{L_h}{N}-1\right)$ beträgt. Die Bereiche mit maximaler Überlagerung können nun ermittelt werden, woraus sich die Gesamtformel zur Ermittlung aller Additionen des Overlap-Add Verfahrens ergibt (36 & 37).

Bereiche mit max. Overlap:

$$\frac{L_x+L_h-2N}{N} - 2 \cdot \left(\frac{L_h}{N}-1\right) = \frac{L_x+L_h-2N-2 \cdot L_h+2N}{N} = \frac{L_x-L_h}{N} \tag{36}$$

Formel für Additionen des Overlap-Add Verfahrens (für $L_x \geq L_h$):

$$W_{OV} = W_{OV1} + W_{OV2} = 2 \cdot \sum_{m=1}^{\frac{L_h}{N}-1} 2m + \left(\frac{L_x-L_h}{N}\right) \cdot (2 \cdot L_h - N) \tag{37}$$

Ein System, für das $L_x = L_h$ gilt, erreicht folglich nie die maximale Überlagerung. Der zweite Teil der Formel fällt in diesem Fall durch den Vorfaktor null weg.

Für ein unendlich langes Eingangssignal, wie es bei einer technischen Umsetzung des Prozesses erwartet wird, ergeben sich entsprechend unendlich viele Abschnitte mit maximaler Überlagerung. Deswegen ist eine Annäherung an die tatsächlichen Additionen mit der Annahme von maximaler Überlagerung über den gesamten Prozess möglich, wenn $L_h << L_x$. In Tabelle 4 sind sowohl exakte Berechnung als auch die Näherung für den Fall $L_x = 2\ Sek., L_h = 1\ Sek.$ und $f_a = 44,1\ kHz.$

N	exakte Berechnung	Näherung
2	$2,92 \cdot 10^9$	$5,83 \cdot 10^9$
4	$1,22 \cdot 10^9$	$2,92 \cdot 10^9$
8	$5,47 \cdot 10^8$	$1,46 \cdot 10^9$
16	$2,58 \cdot 10^8$	$7,29 \cdot 10^8$
32	$1,25 \cdot 10^8$	$3,64 \cdot 10^8$
64	$6,17 \cdot 10^7$	$1,82 \cdot 10^8$
128	$3,06 \cdot 10^7$	$9,09 \cdot 10^7$
256	$1,52 \cdot 10^7$	$4,53 \cdot 10^7$
512	$7,57 \cdot 10^6$	$2,25 \cdot 10^7$
1024	$3,76 \cdot 10^6$	$1,11 \cdot 10^7$
2048	$1,86 \cdot 10^6$	$5,39 \cdot 10^6$
4096	$9,06 \cdot 10^5$	$2,55 \cdot 10^6$

Tabelle 4: Näherung und exakte Berechnung der Additionen bei Overlap-Add

Um für die folgenden Betrachtungen eine einheitliche Einheit zu haben, wird auch die Formel für das Overlap-Add Verfahren auf eine Ausgabe in FLOPS umgerechnet. Dazu werden die Additionen als komplex betrachtet, um wieder eine möglichst universelle Analyse durchführen zu können. Im Bereich von Audiosignalen wären an dieser Stelle wieder reelle Werte zu erwarten (38).

$$W_{OV} = 2 \cdot \left(2 \cdot \sum_{m=1}^{\frac{L_h}{N}-1} 2m + \left(\frac{L_x - L_h}{N} \right) \cdot (2 \cdot L_h - N) \right) \; FLOPS \qquad (38)$$

3.3.5 Betrachtung des Gesamtprozesses

Um die Berechnungen des gesamten Prozesses zu analysieren, werden die Formeln für die Teilbetrachtung zusammengeführt. Dadurch entsteht eine Kurvenschaar, welche verschiedene Fälle abbildet, welche auf reelle Prozesse zurückgeführt werden können. An dieser Stelle führt ein Blick voraus zu einem weiteren Zusammenhang, der im Verlauf einer möglichen Implementierung wichtig wird. Eine Programmiersprache wie C durchläuft den Programmablauf einmal und kann durch Dauerschleifen so beeinflusst werden, dass ein kontinuierlicher Ablauf realisierbar wird. Die hohen Prozessorgeschwindigkeiten sorgen zudem für eine Annäherung an Echtzeit. Innerhalb der Schleife ergeben sich je nach den Anweisungen und Verzweigungen verschieden große Anzahlen an durchzuführenden Berechnungen. Eine Analyse ist demnach genau dann brauchbar, wenn der maximale Rechenaufwand gezeigt wird. Ist dieser für eine Zeitspanne erfasst, kann durch entsprechende Skalierung auf Prozesse in jeglicher Länge geschlossen werden. Dieser Zusammenhang wird spätestens dann deutlich, wenn die Länge des Eingangssignals in Bezug auf die bereits durchgeführten Analyseschritte betrachtet wird. Diese steht grundsätzlich als Faktor außerhalb der gezeigten Berechnungen und kann so relativ einfach verändert werden.

Die Formel zur Ermittlung des gesamten Rechenaufwands wird durch Addition der Teilergebnisse aufgestellt.

$$\begin{aligned} W &= \frac{L_x}{N} \cdot (2 \cdot W_{FFT} + W_{MULT}) + W_{OV} \\ &= \frac{L_x}{N} \cdot \left(\left(1 + \frac{L_h}{N} \right) \cdot 10N \cdot \log_2 (2N) + 12 \cdot \frac{L_h}{N} \right) \\ &\quad + 2 \cdot \left(2 \cdot \sum_{m=1}^{\frac{L_h}{N}-1} 2m + \left(\frac{L_x - L_h}{N} \right) \cdot (2 \cdot L_h - N) \right) \; FLOPS \end{aligned} \tag{39}$$

Der Overlap-Add-Teil der Formel kann nun noch weiter vereinfacht werden, um den Fall der maximalen Berechnungen darzustellen. Dazu wird der erste Teil der Berechnung, welcher Start und Ende enthält, vernachlässigt. Ein vorauszusehendes Ende wird in diesem Prozess nach der Implementierung so oder so nicht geben, wenn man sich vorstellt, dass ein Gerät, welches diesen Prozess umsetzt, vom Benutzer zu jeder Zeit abgeschaltet werden könnte und dann wieder neu gestartet wird. Der Beginn kann insofern vernachlässigt werden, als dass dieser auch mit maximaler Überlagerung der Segmente realisiert werden könnte. Als Vorfaktor werden nun wieder alle Bereiche mit Additionen gesetzt, sodass die Betrachtung des Overlap-Add Verfahrens nun alle Bereiche als jene mit maximaler Überlagerung darstellt und entsprechend berechnet wird (40).

$$\begin{aligned} W &= \frac{L_x}{N} \cdot \left(\left(1 + \frac{L_h}{N} \right) \cdot 10N \cdot \log_2 (2N) + 12 \cdot \frac{L_h}{N} \right) \\ &\quad + 2 \cdot \left(\frac{L_x - L_h}{N} \right) \cdot (2 \cdot L_h - N) \; FLOPS \end{aligned} \tag{40}$$

Die Variablen dieser Formel sind L_x, L_h und N. Die Segmentlänge N ist dabei die entscheidende Variable, wenn es um die Veränderung des Arbeitsaufwands geht. Sie wird deswegen in den Betrachtungen auf die x-Achse gesetzt, sodass in einer Kurve mit festen Werten für L_x und L_h der Arbeitsaufwand abhängig von der Segmentlänge zu sehen ist. Der Wert für L_h ergibt sich aus dem Anwendungszusammenhang. So ist zum Beispiel eine Raumimpulsantwort bis zu mehrere Sekunden lang, ein einfaches Filter kann in wesentlich geringeren Zeitfenstern umgesetzt werden. Der Wert L_h kann dem entsprechend für verschiedene Anwendungsfälle eingesetzt werden und bleibt für diese konstant. L_x ist nicht vorauszusehen, deswegen wird ein Wert von einer Sekunde eingesetzt, sodass durch anschließendes Multiplizieren der Wert für andere Längen L_x berechnet wird.

Besonders im Audiobereich ist noch die Betrachtung der Abtastfrequenz wichtig. Diese entscheidet schließlich darüber, wie lang eine Sekunde in Abtastwerten ausgedrückt ist, also über den Wert für L_x und L_h, der eingesetzt wird.

3.4 Ergebnisse

Die Auswertung der Formel für verschieden lange Impulsantworten und Abtastfre-

quenzen wird schnell unübersichtlich, wenn zu viele Fälle erfasst werden. Deswegen werden in diesem Abschnitt der Thesis drei Beispielfälle betrachtet, welche ebenfalls in der Speicheranalyse genutzt werden. Weitere Fälle können entsprechend der Formel analysiert werden, indem die zugehörigen Werte eingesetzt werden. Als jene drei Fälle werden folgende untersucht, dabei jeweils für $L_x = 1\ Sek.$:

- $L_h = 2\ Sek.\ bei\ f_a = 22\ kHz$
- $L_h = 0.5\ Sek.\ bei\ f_a = 44,1\ kHz$
- $L_h = 1\ Sek.\ bei\ f_a = 48\ kHz$

N	**2 Sek.@ 22 kHz**	**0.5 Sek.@ 44,1 kHz**	**1 Sek.@ 48 kHz**
1	$3,89 \cdot 10^9$	$9,72 \cdot 10^8$	$4,61 \cdot 10^9$
2	$1,56 \cdot 10^{10}$	$1,41 \cdot 10^{10}$	$3,46 \cdot 10^{10}$
4	$9,84 \cdot 10^9$	$8,75 \cdot 10^9$	$2,13 \cdot 10^{10}$
8	$5,77 \cdot 10^9$	$5,41 \cdot 10^9$	$1,31 \cdot 10^{10}$
16	$3,45 \cdot 10^9$	$3,27 \cdot 10^9$	$7,89 \cdot 10^9$
32	$2,02 \cdot 10^9$	$1,93 \cdot 10^9$	$4,64 \cdot 10^9$
64	$1,16 \cdot 10^9$	$1,11 \cdot 10^9$	$2,67 \cdot 10^9$
128	$6,56 \cdot 10^8$	$6,35 \cdot 10^8$	$1,52 \cdot 10^9$
256	$3,67 \cdot 10^8$	$3,57 \cdot 10^8$	$8,50 \cdot 10^8$
512	$2,03 \cdot 10^8$	$2,00 \cdot 10^8$	$4,73 \cdot 10^8$
1024	$1,12 \cdot 10^8$	$1,12 \cdot 10^8$	$2,62 \cdot 10^8$
2048	$6,22 \cdot 10^7$	$6,35 \cdot 10^7$	$1,45 \cdot 10^8$
4096	$3,49 \cdot 10^7$	$3,72 \cdot 10^7$	$8,13 \cdot 10^7$
22050	$1,03 \cdot 10^7$	$---$	$---$
44100	$---$	$1,09 \cdot 10^7$	$---$
48000	$---$	$---$	$2,40 \cdot 10^7$

Tabelle 5: Analyse des Arbeitsaufwands für die drei zuvor beschriebenen Fälle

Die Segmentlänge N ist bei den Auswertungen in einen Bereich eingegrenzt worden, indem eine Nutzung als Echtzeitprozess möglich wäre. Dies hängt neben der Segmentlänge auch von der Abtastfrequenz ab, welche ebenfalls darüber entscheidet, wie hoch die Latenz des Systems zu erwarten ist. So entspricht eine Segmentlänge von $N = 128$ bei $f_a = 22kHz$ einer Verzögerung von 5.81 ms, bei $f_a = 48kHz$ wären es nur 2.67 ms. In der Auswertung macht es aus diesem Grund Sinn, die verschiedenen Segmentlängen mit Hilfe der Abtastfrequenzen zu verrechnen und darzustellen, sodass eine Angabe einer Minimallatenz für die Systeme verschiedener Abtastfrequenzen möglich ist.

In Tabelle 5 sind die drei Fälle jeweils für $N = 1$ (diskrete Faltung) bis $N = 2048$ dargestellt, zusätzlich die Fälle, in denen, je nach Abtastfrequenz f_a der gesamte Block des Eingangssignals verarbeitet wird. Durch das setzen von $L_x = 1\ Sek.$ ist aus den gezeigten Werten eine einfache Umrechnung für längere Zeitabschnitte durch entsprechende Faktoren vor der Gesamtgleichung möglich. In Abbildung 13 ist eine

Kurvenschaar für weitere Fälle zu sehen, um einen Vergleich in Bezug auf Abtastfrequenz und Impulsantwortlängen zu ermöglichen.

Deutlich erkennbar ist, dass es einige Fälle gibt, in denen der Ansatz der segmentierten Faltung keinen Sinn macht, weil er rechenintensiver als die diskrete Faltung ist. In der Auswertung sind dies, über alle gezeigten Auswertungen gesehen, die Fälle $2 \leq N < 64$ (für $L_h = 1\ Sek.$). Würden diese realisiert, wären die Systeme, welche diese Berechnungen zeitlich umsetzen könnten, die Ideallösung für eine Implementierung. Die Latenz wäre sehr gering, sodass eine Echtzeitumgebung geschaffen werden könnte. Die Fälle, in denen im Implementierungsansatz in späteren Abschnitten der Thesis gearbeitet wird, sind Bereiche um $N = 64$ bis maximal $N = 256$. Diese liegen im Bereich einer, zumindest in der Theorie, akzeptablen Latenz und liegen im Bereich mit einer entsprechenden Ersparnis beim Rechenaufwand. Es ist auffällig, dass die Länge der Impulsantwort darüber entscheidet, ab wann einer der Prozesse im Vergleich zur diskreten Faltung weniger Rechenoperationen benötigt.Die Abhängigkeit

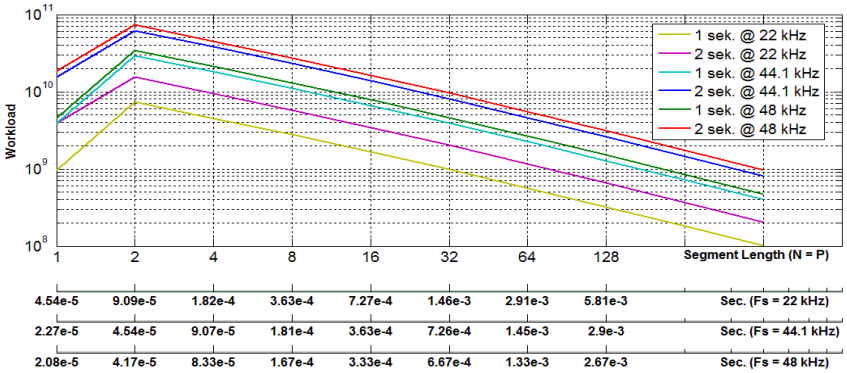

Abbildung 13: Auswertung der Analyse des Arbeitsaufwands für verschiedene Abtastfrequenzen und Impulsantwortlängen

von der Länge L_h der Impulsantwort wird auch in den Folgeabschnitten noch eine wichtige Rolle spielen. Je weniger die Länge L_x des Eingangssignals im Mittelpunkt steht, desto einfacher funktioniert die Umsetzung in einen Programmiertext, welcher durchgehend für ein beliebiges L_x und ein zuvor festgelegtes L_h das Ausgangssignal berechnet. In der Analyse des Arbeitsaufwands muss der Wert L_x mit betrachtet werden, um den Arbeitsaufwand über der Zeit voraussagen zu können und diesen anschließend mit den Möglichkeiten der Prozessortechnik zu vergleichen.

4 Speicheraufwand der segmentierten Faltung

Neben dem Arbeitsaufwand ist für eine Implementierung eines Prozesses die Frage nach dem benötigten Speicher eine wesentliche. Prozessoren haben über Zeit zwar mehr Speicher erhalten und verarbeiten auch Daten aus externen Speichern mit hoher Geschwindigkeit, jedoch muss insbesondere bei externem zusätzlichen Speicher oft mehr Geld ausgegeben werden, welches durch Optimierung vielleicht hätte gespart werden können. Deswegen macht eine Analyse des Speicherbedarfs eines Prozesses vor der Implementierung und der Prozessorwahl Sinn. Die Speicherauslastung kann dann bereits ungefähr vorausgesagt werden und externer zusätzlicher Speicher kann vermieden werden. Prozessor-interner Speicher sollte dabei nicht komplett ausgelastet sein, damit für den Fall, dass ein noch nicht ausreichend optimierter Prozess getestet werden soll, kein Speicherüberlauf passieren kann. Daten würden in diesem Fall nicht korrekt geschrieben werden können, weil kein weiterer Speicher mehr verfügbar ist. Von einer zu knappen Speicherbemessung sollte deswegen abgesehen werden.

Zur Analyse des Speicheraufwands der segmentierten Faltung wird zunächst wieder unterteilt und es werden einzelne Aspekte betrachtet, welche anschließend zusammengeführt werden und den Gesamtwert bilden. Das Zerlegen in diese Teilaspekte wird im Folgekapitel noch einmal aufgegriffen, wenn die Umsetzung in einen Prozess und die Abbildung in Programmiersprache erfolgt. Eine Wortbreite wird bei den Betrachtungen zwar in Form einer Variable mitgeführt, aber zunächst, außer in speziellen Beispielen, nicht eingesetzt, sodass die Analyse sich noch nicht zu sehr auf Systeme mit entsprechenden Wortbreiten fokussiert.

4.1 Betrachtung der Impulsantwort

Die Impulsantwort ist in Bezug auf den benötigten Speicher am einfachsten und genausten vorherzusagen. Diese wird schließlich nur einmal in Segmenten transformiert und muss anschließend während des gesamten Prozesses verfügbar sein. Somit belegt diese den Speicher dauerhaft und kann entsprechend betrachtet und analysiert werden. Die Länge der Impulsantwort L_h wird durch die Segmentierung zunächst nicht verändert, jedoch wird anschließend nicht die gesamte Impulsantwort, sondern die einzelnen Segmente in den Frequenzbereich transformiert. So entstehen eine Reihe von transformierten Impulsantwortsegmenten $H_i\left[f\right]$, welche im Speicher verbleiben müssen, weil jedes Segment $x_k\left[n\right]$ nach der FFT mit allen Segmenten $H_i\left[f\right]$ multipliziert wird. Der Speicher darf deswegen auch an dieser Stelle niemals überschrieben werden, es sei denn, die Impulsantwort soll sich während des Betriebs verändern.

Wird ein Segment $h_i\left[n\right]$ in den Frequenzbereich transformiert, so entsteht das Segment $H_i\left[f\right]$, welches die Spektrallinien von $h_i\left[n\right]$ enthält. Auf Grund der Faltung und ihrer Ergebnislänge $L_y = L_h + L_x - 1$ muss die FFT mindestens diese Anzahl an Spektrallinien abbilden. Der genutzte Radix-2 Algorithmus legt zudem fest, dass ein Wert $N = 2^n$ als FFT-Länge genutzt werden muss. Bezieht man das Faltungsergebnis auf die FFT-Länge, so wäre dies $L_y = 2N - 1$, was kein gerades Ergebnis zur Folge haben kann. Deswegen wird $2N$ als FFT-Länge genutzt, was zur Folge hat, dass die Segmente $H_i\left[f\right]$ ebenfalls die Länge $2N$ haben. Außerdem kann davon aus-

gegangen werden, dass komplexe Werte vorliegen, welche normalerweise im Speicher getrennt mit Real- und Imaginärteil abgelegt werden. Grundsätzlich braucht man, unabhängig von der Darstellungsart, zwei Informationen bei einer komplexen Zahl, daher muss die doppelte Menge an Speicher verfügbar sein, um ein Segment $H_i[f]$ zu speichern, wenn es komplett aus komplexen Werten besteht. Der Speicheraufwand für ein transformiertes Segmente der Impulsantwort ist somit ermittelt (41).

$$M_{H_i} = 2 \cdot 2N \cdot S \; Bit \tag{41}$$

Die Wortbreite würde pro Wert die entsprechende Anzahl an Bit zuweisen. Würde der Wert S aus der Gleichung gestrichen, läge folglich ein 1-Bit System vor, welches für ein System, welches unkomprimierte Audiodaten speichern soll, unbrauchbar wäre.

Die Impulsantwort wird der Segmentierung mit Länge N aufgeteilt in $\frac{L_h}{N}$ Segmente. Mit diesem Faktor wird aus der Betrachtung eines Einzelsegments im Prozess eine Betrachtung für den Gesamtprozess der segmentierten Faltung. Diese Erweiterung ist deswegen möglich, weil die Segmente alle auf die selbe Art und Weise den Prozess der Transformation in den Frequenzbereich durchlaufen. Die gesamte in Segmenten transformierte Impulsantwort benötigt im Speicher folglich $\frac{L_h}{N}$ mal den Wert eines Segments (42)

$$M_{H_{ges}} = 2 \cdot 2N \cdot \frac{L_h}{N} \cdot S \; Bit = 4 \cdot L_h \cdot S \; Bit \tag{42}$$

Dadurch ist auch gezeigt, dass seine Segmentierung keinen zusätzlichen Speicheraufwand bei der Betrachtung des Ergebnisses erzeugt. Die Berechnung und der dazu benötigte Speicher sind natürlich abhängig von der Segmentlänge N. Die in Segmenten transformierte Impulsantwort besitzt diese Anhängigkeit nicht und nutzt den gleichen Anteil am Speicher wie eine am Stück transformierte Impulsantwort. Dabei sei allerdings erwähnt, dass die Ergebnisse keineswegs in beiden Fällen gleich sind (43).

$$\begin{aligned} h[n] &= 0.01 \cdot n; \; 1 \leq n \leq 100; \; N = 10 \\ H[f] &= 50.5, -19.76 - j32.47, -0.5 + j15.91, -1.74 - j10.82, ... \\ H_1[f] &= 0.55, -0.14 - j0.38, -0.05 + j0.15, 0.04 - j0.12, ... \end{aligned} \tag{43}$$

Der Speicher, in welchem die Impulsantwort abgelegt wird, darf wie bereits erwähnt, nicht überschrieben werden. Auf diesen erfolgen während des gesamten laufenden Prozesses immer wieder Zugriffe. Der Speicherbereich sollte deswegen bei einer möglichen Implementierung je nach Impulsantwort entsprechend gesichert sein, sodass es zu keinem unerwünschten Überschreiben kommt. Ebenso kann die Transformation der Impulsantwort in den Frequenzbereich bereits zu Beginn des Prozesses erfolgen. Benötigt werden während des Prozesses nur die transformierten Segmente, daher sollten auch nur diese im Speicher liegen. Die Impulsantwort im Zeitbereich könnte im Voraus bekannt sein und in einem Zwischenspeicher liegen, der sich überschreiben

lässt. So wäre ein Nutzer in der Lage, eine andere Impulsantwort einzusetzen, welche dann beim Start des Systems in Segmenten transformiert im Hauptspeicher abgelegt wird und anschließend nur noch gelesen wird. Diese Überlegung wird auch bei der Umsetzung des Prozesses in Programmiertext im weiteren Verlauf noch eine wichtige Rolle spielen, weil dadurch eine entscheidende Optimierung möglich ist.

4.2 Speicheranalyse des Ablaufs für ein Segment

Läuft ein Segment $x_k[n]$ in den Prozess, so wird zunächst in den Frequenzbereich transformiert. Der Speicher, den diese Transformation zusätzlich nutzen würde, ist so gering, dass er an dieser Stelle vernachlässigt wird. Hinzu kommt, dass es mittlerweile immer mehr so genannter In-Place Verfahren gibt, welche eine Berechnung der FFT ohne zusätzlichen Speicheraufwand ermöglichen sollen. Diese zu analysieren und zu testen ist jedoch nicht Bestandteil dieser Thesis, deshalb wird davon ausgegangen, dass zusätzlicher Speicher in geringem Maße erforderlich ist. Dieser variiert zudem je nach der verwendeten Prozessortechnologie und ist somit auch nicht allgemein bestimmbar, weswegen die Analyse ohne diesen durchgeführt wird. Durch die gleich bleibende Segmentlänge N für den gesamten Prozess kann für den erforderlichen Speicher für ein transformiertes Segment $X_k[f]$ der gleiche Wert wie bei den transformierten Segmenten der Impulsantwort angenommen werden (44)

$$M_{X_k} = 2 \cdot 2N \cdot S\ Bit = 4N \cdot S\ Bit \tag{44}$$

Der Speicher für die Segmente $X_k[f]$ wird nun aber nicht mit der Länge des Eingangssignals verrechnet. Dies wäre auch im Hinblick auf ein unendliches Eingangssignal nicht realisierbar und ist auch nicht notwendig, wenn es um eine optimale Speichernutzung geht, bei der alle nicht mehr benötigten Daten überschrieben werden können. Die Analyse läuft zunächst weiter für ein nun transformiertes Segment $X_k[f]$ und wird anschließend auf den Gesamtprozess bezogen, wodurch bei der Optimierung ersichtlich werden wird, dass die Länge des Eingangssignals kein entscheidendes Kriterium für die Analyse ist. Zunächst soll der Prozess in einer nicht-optimierten Form betrachtet werden, sodass an dieser Stelle keine Optimierungen erfolgen.

Die transformierten Segmente des Eingangssignals $X_k[f]$ werden nun jeweils mit allen transformierten Segmenten der Impulsantwort $H_i[f]$ multipliziert. Dies passiert jedoch für ein Segment $X_k[f]$ nicht direkt für alle Segmente der Impulsantwort. Aus den Betrachtungen des Overlap-Add Verfahrens (Abb. 12) ist bereits ersichtlich, dass nicht direkt alle Ergebnisse $Y_{k1}[f] - Y_{ki}[f]$ für ein Segment $X_k[f]$ vorliegen müssen, sondern pro Berechnungsschritt, also alle N Samples, jeweils ein neues Ergebnissegment benötigt wird. Deswegen ist für eine Betrachtung des Speicheraufwands die Größe eines Ergebnissegments wesentlich wichtiger als die Größe aller Ergebnissegmente, welche aus einem Eingangssegment $x_k[n]$ beim Durchlaufen des Prozesses berechnet werden.

Die Multiplikation eines Segments $X_k[f]$ mit einem Segment $H_i[f]$ wird komponentenweise durchgeführt. Das Ergebnissegment hat daher die gleiche Größe wie der

längere der beiden verrechneten Vektoren. In diesem Fall sind beide gleich lang, deswegen ist die Größe des Ergebnisses gleich der Größe einer der beiden multiplizierten Vektoren $X_k[f]$ und $H_i[f]$ (45).

$$M_{Y_{ki}} = 2 \cdot 2N \cdot S\ Bit = 4N \cdot S\ Bit \tag{45}$$

Die berechneten Ergebnissegmente $Y_{ki}[f]$ werden anschließend wieder per IFFT in den Zeitbereich transformiert. Dies passiert, je nach Möglichkeit und verwendetem FFT-Algorithmus, entweder direkt per In-Place Verfahren oder mit einem entsprechenden zusätzlichen Speicher, welcher in diesen Betrachtungen nicht erfasst ist. Ein Segment $y_{ki}[n]$, welches durch diese Transformation entsteht, hat genau jene Menge an Werten wie die IFFT vorgibt. Dieser Wert entspricht $2N$, somit bleibt auch nach der Rücktransformation die Länge von $2N \cdot S\ Bit$ erhalten.

In Bezug auf das Overlap-Add Verfahren und den benötigten Speicher ist ein Aspekt wesentlich. Ein Ausgabesegment muss nach der Berechnung abgelegt werden, bevor die Ausgabe erfolgt. Geht man von einem kontinuierlich ablaufenden Prozess ab, so wie er auch im Rahmen dieser Thesis in Programmcode entwickelt werden soll, laufen die Additionen der Reihe nach ab und je ein Ergebnis entsteht, welches dann zwischengespeichert werden muss, bis alle restlichen Ergebnisse dieses Ausgabesegments berechnet sind. Ebenso sollte das Ausgabesegment nicht sofort überschrieben werden, um eine mögliche Fehlerquelle auszuschließen. Daher wird von zwei Ausgangsvektoren ausgegangen, welche abwechselnd beschrieben und ausgelesen werden (Abb. 14). Der zusätzliche Speicheraufwand für diese zwei Ausgangsvektoren kann anhand

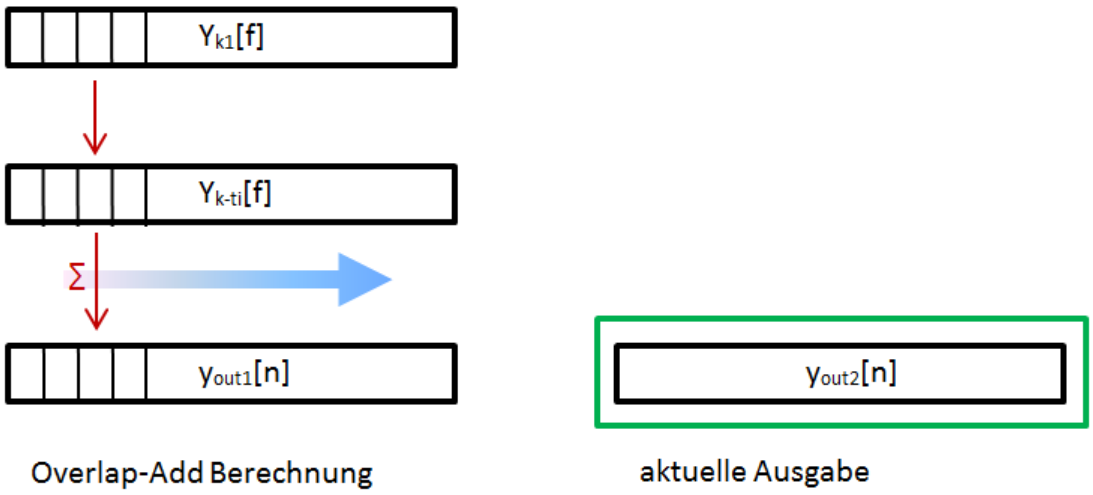

Abbildung 14: Die Ausgangsvektoren y_{out1} und y_{out2} werden abwechselnd mit den Overlap-Add Ergebnissen beschrieben und ausgelesen

der vorigen Betrachtungen direkt angegeben werden. Ein Ausgabeabschnitt enthält genau N Werte, der zugehörige Ausgabevektor erhält entsprechend auch die Länge N. Die Verdopplung führt zu der benötigten Speichermenge für die Ausgabevektoren (46).

$$\begin{aligned} M_y &= 2N \cdot S \; Bit \\ M_{y,kompl} &= 4N \cdot S \; Bit \end{aligned} \tag{46}$$

Daran ist auch schon zu erkennen, dass das Overlap-Add Verfahren im Vergleich zum Rest des Prozesses den geringsten Speicherbedarf hat. Dieser entspricht genau dem Bedarf eines Segments $y_{ki}[n]$. Aus diesen Segmenten werden wiederum die Ergebnisse für die Ausgabevektoren berechnet, der Speicher wird durch Overlap-Add im Vergleich zu diesen Segmenten nur wenig beansprucht.

4.3 Betrachtung des nicht-optimierten Gesamtablaufs

Aus den Einzelschritten kann jetzt, ähnlich wie beim Arbeitsaufwand, auf den gesamten Prozess geschlossen werden. Dazu wird zunächst ermittelt, wie der Speicherbedarf ohne jegliche Optimierung ist, um dann entsprechende Optimierungen durchzuführen. In Abb. 15 ist dargestellt, wie sich der Speicherbedarf für ein Segment $x_k[n]$ verteilt, wenn keine Optimierung durchgeführt wird. In diesem Fall verbleibt alles, was im Prozess gespeichert wird im Speicher und es kann nichts überschrieben werden.

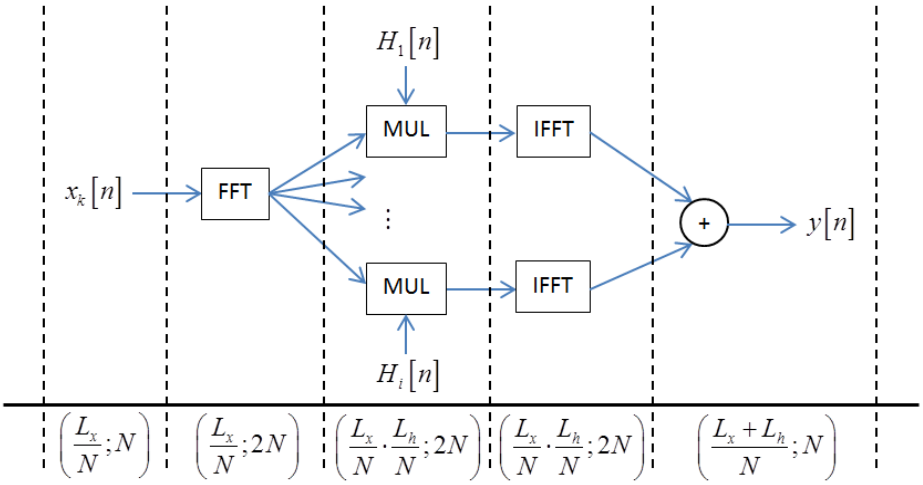

Abbildung 15: Speicherbelegung bei nicht-optimiertem Prozess: In Klammern sind jeweils die Zahl der Segmente und die Länge dieser angegeben

Dieser worst-case Fall kann anschließend mit einem optimierten verglichen werden, um die Ersparnis darzustellen. Dabei wurde beachtet, dass die optimierte Lösung auch prozessor-unabhängig bleibt und eine Portierbarkeit nicht beeinflusst wird. Ein Segment $x_k[n]$ benötigt bei dieser nicht-optimierten Lösung Speicher für das Segment selbst, für das transformierte Segment $X_k[f]$, alle Multiplikationsergebnisse $Y_{ki}[f]$ und die rücktransformierten Segmente $y_{ki}[n]$ (47).

$$
\begin{aligned}
M_{x_k} &= \frac{L_x}{N} \cdot N \cdot S\ Bit = L_x \cdot S\ Bit \qquad (47)\\
M_{x_k,kompl} &= 2 \cdot \frac{L_x}{N} \cdot N \cdot S\ Bit = 2 \cdot L_x \cdot S\ Bit\\
M_{X_k} &= \frac{L_x}{N} \cdot 4N \cdot S\ Bit = 4 \cdot L_x \cdot S\ Bit\\
M_{Y_{ki}} &= \frac{L_x}{N} \cdot \left(\frac{L_h}{N} \cdot 4N \cdot S \right)\ Bit = \frac{L_x}{N} \cdot 4 \cdot L_h \cdot S\ Bit\\
M_{y_{ki}} &= \frac{L_x}{N} \cdot 4 \cdot L_h \cdot S\ Bit
\end{aligned}
$$

Für eine Betrachtung des Gesamtprozesses kann jetzt anhand des Faktors Lx auf den Gesamtbedarf des Speichers für verschieden lange Eingangssignale geschlossen werden. Ebenso können die Impulsantwort und die Ausgabevektoren des Overlap-Add Verfahren mit ihren Speicheranforderungen hinzugefügt werden. Die in (48) gezeigte Formel gibt den Speicheraufwand für diesen nicht-optimierten Prozess an, in Tab. 6 finden sich Werte für die drei Vergleichsfälle

- $L_h = 2\ Sek.\ bei\ f_a = 22\ kHz$
- $L_h = 0.5\ Sek.\ bei\ f_a = 44.1\ kHz$
- $L_h = 1\ Sek.\ bei\ f_a = 48\ kHz$

und einer Wortbreite von 16 Bit.

$$
\begin{aligned}
M_{ges} &= \left(4 \cdot L_h + 4N + \frac{L_x}{N} \cdot (N + 4N + 4 \cdot L_h + 4 \cdot L_h) \right) \cdot S\ Bit \qquad (48)\\
&= \left(4 \cdot L_h + 4N + \frac{L_x}{N} \cdot (5N + 8 \cdot L_h) \right) \cdot S\ Bit
\end{aligned}
$$

Alle anderen wesentlichen Wortbreiten können durch Multiplizieren der Tabellenwerte mit 2^n berechnet werden. Die 16 Bit werden verwendet, um an das typische CD-Format von 44 kHz bei 16 Bit anzuknüpfen. 32 Bit sind typisch für Floating-Point Prozessoren, welche im Verlauf der Thesis noch genauer betrachtet werden. Im Bereich der GPPs (general purpose processors), also jenen Prozessoren, die im Heimcomputer genutzt werden, sind 64 Bit als Wortbreite etabliert. Auf Mikrocontrollern findet man jene allerdings noch recht selten, so zumindest der Recherchestand zu Beginn des Monats Mai des Jahres 2013.

Die nicht-optimierte Fassung des Prozesses ist besonders durch zwei Gegebenheiten sehr unvorteilhaft. Zum einen ist der Speicherbedarf von der Länge L_x abhängig. Diese Abhängigkeit sorgt dafür, dass der Prozess je nach Länge des Eingangssignals einen anderen Speicheraufwand hat. Dies zeigt schon die wesentliche Problematik, der

N	**2 Sek.@ 22 kHz**	**0.5 Sek.@ 44,1 kHz**	**1 Sek.@ 48 kHz**
1	0	0	0
2	$6.22 \cdot 10^{10}$	$6.22 \cdot 10^{10}$	$1.47 \cdot 10^{11}$
4	$3.11 \cdot 10^{10}$	$3.11 \cdot 10^{10}$	$7.37 \cdot 10^{10}$
8	$1.56 \cdot 10^{10}$	$1.56 \cdot 10^{10}$	$3.69 \cdot 10^{10}$
16	$7.78 \cdot 10^{9}$	$7.78 \cdot 10^{9}$	$1.84 \cdot 10^{10}$
32	$3.89 \cdot 10^{9}$	$3.89 \cdot 10^{9}$	$9.22 \cdot 10^{9}$
64	$1.95 \cdot 10^{9}$	$1.95 \cdot 10^{9}$	$4.61 \cdot 10^{9}$
128	$9.74 \cdot 10^{8}$	$9.76 \cdot 10^{8}$	$2.31 \cdot 10^{9}$
256	$4.88 \cdot 10^{8}$	$4.90 \cdot 10^{8}$	$1.16 \cdot 10^{9}$
512	$2.45 \cdot 10^{8}$	$2.47 \cdot 10^{8}$	$5.80 \cdot 10^{8}$
1024	$1.23 \cdot 10^{8}$	$1.25 \cdot 10^{8}$	$2.92 \cdot 10^{8}$
2048	$6.27 \cdot 10^{7}$	$6.44 \cdot 10^{7}$	$1.48 \cdot 10^{8}$
4096	$3.23 \cdot 10^{7}$	$3.40 \cdot 10^{7}$	$7.60 \cdot 10^{7}$

Tabelle 6: Speicheranalyse des nicht-optimierten Prozesses

Prozess soll schließlich am Ende ein endloses Signal verarbeiten können. Dies wird mit einem Speicheraufwand von $M_{ges} = 232\ MB$ für $L_h = 0.5\ Sek.$ bei $f_a = 44.1\ kHz$ bei einem Eingangssignal mit $L_x = 1\ Sek.$ bei 16 Bit Wortbreite auf Dauer nicht realisierbar. Zwar steigt der Speicheraufwand bei Verlängerung auf $L_x = 2\ Sek.$ nicht genau auf das Doppelte, jedoch ist der Teil der Formel, der durch L_x mitbestimmt wird, wesentlich größer als der Rest. Der zweite Aspekt, an dem die anschließende Optimierung ansetzt, ist jener des Überschreibens im Speicher. Durch ein Überschreiben kann ein großes Maß an Speicherbelegungen eingespart werden. Der Speicher läuft dann mit der Berechnung und wird entsprechend der benötigten Segmente immer dann überschrieben, wenn der Inhalt nicht mehr für Berechnungen benötigt wird.

4.4 Optimierung der Speichernutzung

Um Speicher einzusparen wird der Ablauf der Ergebnisberechnung noch einmal betrachtet. Dieses Mal allerdings vom Ausgang aus, was im Wesentlichen eine Betrachtung des Overlap-Add Verfahrens und der Ausgabesegmente $y_{ki}[n]$ ist. Die Ausgangsvektoren des Overlap-Add Verfahrens können nicht optimiert werden, sie sind aber auch nicht entscheidend für eine Ersparnis beim Speicher, weil ihr Anteil an der Gesamtbetrachtung des Speicheraufwands entsprechend gering ist. Die Eingangssegmente und ihre zugehörigen Ausgangssegmente hingegen bieten eine Menge Einsparpotential, durch welches eine Möglichkeit entsteht, die Analyse von der Länge des Eingangssignals L_x loszulösen und unabhängig durchzuführen. Um die wesentliche Ersparnis sichtbar zu machen, sind in Abb. 16 die Ergebnissegmente $y_{ki}[n]$ farbig anhand ihrer Zugehörigkeit zu den Segmenten $x_k[n]$ markiert. Dabei fällt auf, dass das Segment $x_k[n]$ nicht im gesamten Prozess beteiligt ist, sondern nach $\frac{L_h}{N}$ Ausgabeabschnitten nicht mehr Teil der Ausgabe ist. Für den Speicher heißt dies, dass der Speicher nach $\frac{L_h}{N}$ Ausgaben auf der Position des ersten Segments wieder überschrieben werden kann. Nach $\frac{L_h}{N}$ Ausgaben beginnt bei der Ausgabe mit Overlap-Add der

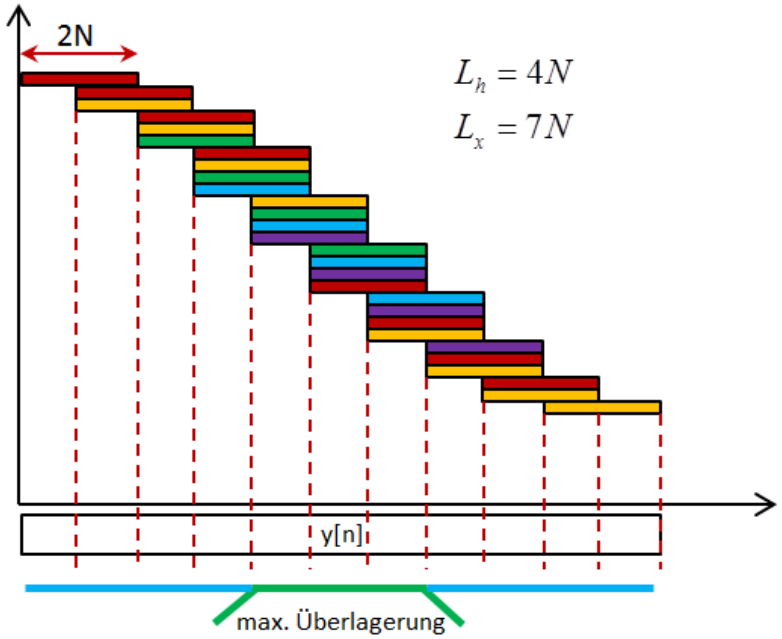

Abbildung 16: Optimierung durch Überschreiben der transformierten Eingangssegmente. Eine gleiche Färbung steht für die Nutzung desselben Speicherbereichs

Bereich mit maximaler Überlagerung. Ab diesem Zeitpunkt endet alle $2N$ die Beteiligung eines Segments $x_k[n]$ am Ausgangssignal und es wird ein neues Segment $x_k[n]$ geladen, welches direkt an den Berechnungen beteiligt ist. Daraus kann geschlossen werden, dass genau $\frac{L_h}{N}$ Eingangssegmente im Speicher gehalten werden müssen, dieser also $\frac{L_h}{N}$-mal den Bereich eines Eingangssegments umfassen muss, um diese Segmente zu speichern. Ab dem Beginn der maximalen Überlagerung kann dann jeweils das zuletzt geladene Segment $x_k[n]$ das nicht mehr beteiligte Segment $x_{k-t}[n]$ im Speicher überschreiben.

Werden die Segmente $x_k[n]$ transformiert, benötigen sie entsprechend die doppelte Größe auf Grund der doppelten Segmentlänge welche durch die FFT mit $2N$ entsteht. Somit benötigt man weitere $\frac{L_h}{N}$ Speicherbereiche, in diesem Fall mit der Länge $2N$, in denen die Segmente $X_k[f]$ nach der Transformation abgelegt werden. An dieser Stelle kann allerdings noch weiter Speicher gespart werden, wenn man sich die Bedeutung der Eingangssegmente und deren Verwendung in der Berechnung genauer ansieht. Verwendet wird nach der Transformation eines Eingangssegments $x_k[n]$ nur das transformierte Eingangssegment $X_k[f]$, nicht jedoch das Segment im Zeitbereich. Es muss daher auch nicht mehr im Prozess vorliegen, woraus folgt, dass es nur einen Eingangsvektor geben muss, welcher jeweils mit dem aktuellen Segment des Eingangssignals beschrieben und anschließend transformiert wird. Da direkt nach dem Beschreiben dieses Vektors das nächste Segment eintrifft und natürlich das vorige Segement nicht überschrieben werden soll, werden zwei Eingangsvektoren genutzt, welche abwechselnd beschrieben und ausgelesen werden. Benötigt werden daher am Eingang des Prozesses die beiden Eingangsvektoren der Länge N, anschließend ein Speicher, in dem $\frac{L_h}{N}$ Segmente $X_k[f]$ der Länge $2N$ Platz finden. Ein Faktor 2 wird

in die Berechnung ab diesem Punkt fest eingerechnet, weil die Ergebnisse nach der FFT als komplex angenommen werden.

Eine Multiplikation eines Segments $X_k[f]$ mit einem Segment $H_i[f]$, welche elementweise durchgeführt wird, hat im Ergebnis die gleiche Länge $2N$ wie die beiden Segmente. Die Speichermenge für die Ergebnissegmente im Frequenzbereich $Y_{ki}[f]$ kann anhand der Ausgabe des Ausgangssignals festgestellt werden. Dazu wird der Prozess bei maximaler Überlagerung betrachtet. Die Ausgabe benötigt zu diesem Zeitpunkt $\frac{2 \cdot L_h}{N}$ Segmente, aus denen die Ergebniswerte berechnet werden. Deswegen müssen auch im Speicher genauso viele Segmente verfügbar sein, daher ist die Speicherauslastung für die Segmente $Y_{ki}[f]$ genau doppelt so hoch wie für die Segmente $X_k[f]$ (49).

$$M_{Y_{ki}} = 2 \cdot \frac{L_h}{N} \cdot 4N \cdot S\ Bit = 8 \cdot L_h \cdot S\ Bit \tag{49}$$

Eine weitere Optimierung kann beim Übergang aus dem Frequenzbereich (Segmente $Y_{ki}[f]$) zurück in den Zeitbereich (Segmente $y_{ki}[n]$) durchgeführt werden. Das Zwischenergebnis $Y_{ki}[f]$ wird im Prozess nur als Segment für die IFFT benötigt, hat aber ansonsten in der Berechnung keinerlei weiteren Nutzen. Daher kann ein dauerhaftes Speichern der Segmente $Y_{ki}[f]$ verhindert werden und somit weiterer Speicher gespart werden. Die Berechnung wird wie in Abb. 17 gezeigt für jedes Ergebnissegment $y_{ki}[n]$ einzeln durchgeführt.

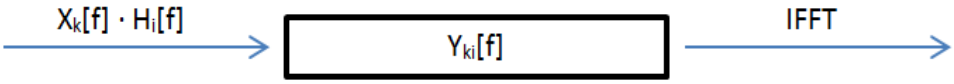

Abbildung 17: Ablauf zur Berechnung der Ergebnissegmente $y_{ki}[n]$. Der Speicher von $Y_{ki}[f]$ wird mit dem nächsten Produkt direkt überschrieben.

Es wird dann nur ein Speicherbereich der Länge $4N$ benötigt, indem ein Zwischenergebnis gespeichert wird. Die Segmente $y_{ki}[n]$ belegen dann die schon gezeigten $\frac{2 \cdot L_h}{N} \cdot 4N \cdot S\ Bit$. Dadurch ergibt sich eine neue, optimierte Speichernutzung, welche in (50) zu sehen ist.

$$
\begin{aligned}
M_{x_k} &= 2N \cdot S\ Bit \\
M_{x_k,kompl} &= 4N \cdot S\ Bit
\end{aligned}
\tag{50}
$$

$$
M_{X_k} = \frac{L_h}{N} \cdot 4N \cdot S\ Bit = 4 \cdot L_h \cdot S\ Bit
$$

$$
M_{Y_{ki}} = 4N \cdot S\ Bit
$$

$$
M_{y_{ki}} = \frac{2 \cdot L_h}{N} \cdot 4N \cdot S\ Bit = 8 \cdot L_h \cdot S\ Bit
$$

$$
\begin{aligned}
M_{ges} &= (4 \cdot L_h + 4N + 4N + 4N + L_h \cdot (4 + 8)) \cdot S\ Bit \\
&= (16 \cdot L_h + 12N) \cdot S\ Bit
\end{aligned}
$$

Aus dieser Darstellung ist schon ersichtlich, dass bei einer optimierten Speicherberechnung keine Abhängigkeit von der Länge des Eingangs L_x vorliegt. Der Prozess orientiert sich beim Speicher nur an der Länge der Impulsantwort L_h und der Segmentlänge N. Der Speicher von $12N$ für die Ausgangsvektoren des Overlap-Add Verfahrens und die Zwischenablage von $Y_{ki}\,[f]$ können bei sehr langen Impulsantworten und kurzen Segmentlängen auch vernachlässigt werden, weil dieser Anteil am Ergebnis nur gering ist. Da in dieser Thesis verschiedenste Varianten untersucht werden, wird diese Vereinfachung nicht angewendet.

4.5 Ergebnisse

Für eine Auswertung werden zunächst die optimierte und die nicht-optimierte Speicherauslastung gegenüber gestellt. In Tab. 7 ist ein Vergleich für den Fall $0.5\ Sek.$ bei $f_a = 44.1\ kHz$ zu sehen, Abb. zeigt den Vergleich im Diagramm. Die Unabhängigkeit von der Länge des Eingangssignals und die Verarbeitung in Zyklen des Ausgangssignals sorgen dabei für eine deutliche Ersparnis beim Speicher. Beträgt dieser bei $N = 64$ und in der nicht-optimierten Version noch $M_{ges} = 232\ MB$ für ein festes L_x von einer Sekunde, so ist bei gleichem Ablauf in der optimierten Version nur noch $M_{ges} = 734\ KB$ an Speicher erforderlich, welcher für ein unbekanntes L_x benötigt wird. Zum Vergleich: Die nicht-optimierte Version verbraucht in diesem Fall fast 325-mal so viel Speicher wie die optimierte.

In Abb. 19 sind mehrere Fälle gezeigt, welche die Speicherauslastung in Abhängigkeit von der Segmentlänge N zeigen.

Es wird sichtbar, wie gering der Einfluss der Addition von $8N$ bei der Berechnung des Speichers ist. Die über lange Zeit gleich bleibenden Werte können mit dem Overlap-Add Verfahren erklärt werden. Bei maximaler Überlagerung liegen immer $2 \cdot \frac{L_h}{N}$ Segmente übereinander. Je kleiner N, desto mehr Segmente $y_{ki}\,[n]$ der Länge $2N$ sind an der Berechnung beteiligt. Die Speicherbereiche für diese Segmente werden daher bei größerem N pro Segment länger, die Anzahl der zu speichernden Segmente

N	nicht-optimiert	optimiert
1	0	0
2	$6,22 \cdot 10^{10}$	$5,65 \cdot 10^{6}$
4	$3,11 \cdot 10^{10}$	$5,65 \cdot 10^{6}$
8	$1,56 \cdot 10^{10}$	$5,65 \cdot 10^{6}$
16	$7,78 \cdot 10^{9}$	$5,65 \cdot 10^{6}$
32	$3,89 \cdot 10^{9}$	$5,65 \cdot 10^{6}$
64	$1,95 \cdot 10^{9}$	$5,66 \cdot 10^{6}$
128	$9,76 \cdot 10^{8}$	$5,67 \cdot 10^{6}$
256	$4,90 \cdot 10^{8}$	$5,69 \cdot 10^{6}$
512	$2,47 \cdot 10^{8}$	$5,74 \cdot 10^{6}$
1024	$1,25 \cdot 10^{8}$	$5,84 \cdot 10^{6}$
2048	$6,44 \cdot 10^{7}$	$6,04 \cdot 10^{6}$
4096	$3,40 \cdot 10^{7}$	$6,43 \cdot 10^{6}$

Tabelle 7: Vergleich des nicht-optimierten und des optimierten Prozesses

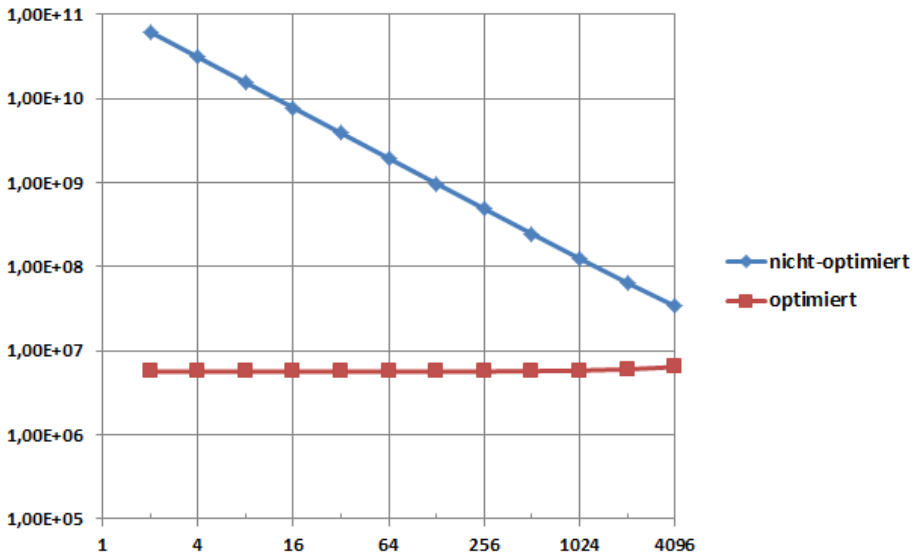

Abbildung 18: Vergleich von optimierter und nicht-optimierter Speicheranalyse

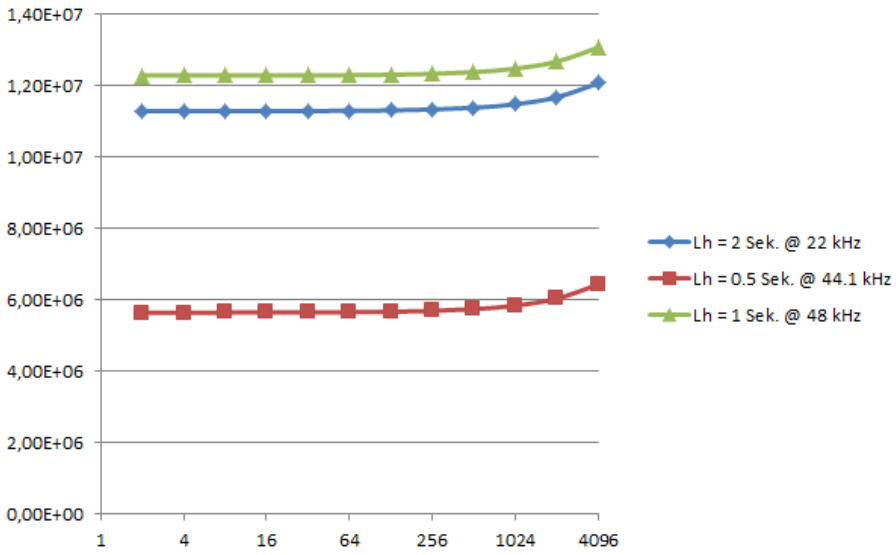

Abbildung 19: Vergleich des Speicherbedarfs für drei verschiedene Anwendungsfälle

nimmt hingegen ab. So bleibt der Speicherbedarf für die meisten Werte für N gleich und ändert sich erst bei großem N, wenn die addierten $8N$ in Bezug auf die Länge der Impulsantwort im Ergebnis erkennbar werden.

Für Anwendungsfälle eines Echtzeitprozesses (Latenz ca. 5 ms) sind Segmentgrößen bis maximal $N = 64$ relevant. Die Speicheranalyse zeigt für diese Fälle einen nahezu gleichen Wert. Dies würde für eine Umsetzung bedeuten, dass ein Prozessor mehrere Segmentlängen umsetzen könnte, ohne dabei einen unterschiedlichen Speicheraufwand zu benötigen.

5 Umsetzung des Gesamtprozesses in Pseudocode

Der Prozess der segmentierten Faltung wird in diesem Abschnitt zunächst mit den vorigen Analysen und weiteren strukturellen Betrachtungen analysiert, um anschließend einen Pseudocode für einen Ablauf eines Computerprogramms zur Berechnung der segmentierten Faltung mit variabler Segmentlänge zu realisieren. Dabei wird noch ohne Hilfe von computerbasierten Werkzeugen gearbeitet, um zunächst eine allgemeine Sicht zu entwickeln, aus der anschließend spezielle Anwendungen wie z.B. MATLAB® eine Umformung in einen lauffähigen Code ermöglichen.

Zur Analyse für den Pseudocode wird auch in diesem Abschnitt mit einer Teilanalyse von Einzelprozessen angesetzt, aus denen dann Programmierabläufe entwickelt werden.

Die Teilanalysen ergeben sich aus dem Ablauf des Prozesses, wenn dieser von Ein- und Ausgang aus betrachtet wird. Dabei ergeben sich bereits durch die Vorbetrachtungen des Arbeits- und Speicheraufwands Teilprozesse, welche an dieser Stelle wieder aufgegriffen werden. Zur Umsetzung als Programmiercode ist besonders wichtig, dass der Prozess als Dauerprozess realisierbar wird, man diesen also mit Hilfe einer Dauerschleife kontinuierlich betreiben kann. Dennoch ändern sich mit jedem Durchlauf bestimmte Parameter, insbesondere in Bezug auf den zu beschreibenden Speicher. Dieser muss anhand der Vorgabe der Speicheroptimierung auch so umgesetzt werden, dass die gezeigte Optimierung auch umgesetzt ist. Zusätzlich müssen beim Overlap-Add je nach Ausgabeabschnitt die korrekten Segmente aus dem Speicher geladen werden. Weiterhin sollte festgelegt sein, wie der Beginn der Berechnung laufen soll. Dabei steht neben der Möglichkeit des Aufbaus des Overlap-Add Verfahrens mit nur einem Segment zu Beginn auch die Berechnung mit dauerhafter maximaler Überlagerung zur Verfügung. Und auch die dazwischen durchgeführten Multiplikationen sollten so organisiert sein, dass nur die Ergebnisse berechnet werden, die für den aktuellen Ausgabevektor erforderlich sind. Die Analyse zur Umformung findet daher in den folgenden Abschnitten statt:

- Transformieren und Speichern der Impulsantwort
- Transformieren und Speichern von Eingangssegmenten
- Multiplikationen mit den Segmenten $H_i\,[f]$ und Speichern der Ergebnisse
- Overlap-Add und Ausgabevektoren
- Realisierung des dauerhaften Ablaufs

5.1 Transformieren und Speichern der Impulsantwort

Die Impulsantwort nimmt auf den eigentlichen Rechenprozess erst bei der Multiplikation im Frequenzbereich Einfluss. An der Stelle werden auch erst die Ergebnisse der Transformation der Segmente der Impulsantwort benötigt. Aus den vorigen Analyse ist bereits hervorgegangen, dass die transformierten Segmente $H_i\,[f]$ dauerhaft bei der Berechnung des Prozesses benötigt werden, es daher sinnvoll ist, diese separat abzuspeichern. Auf Grund der dauerhaften Verwendung wird der dafür vorgesehene Speicher beim Ablauf des Prozesses auch nicht überschrieben. Folglich ist es nicht entscheidend, zu welcher Zeit die Transformation der Segmente in den Frequenzbereich

durchgeführt wird. Deswegen wird diese Transformation außerhalb des kontinuierlichen Rechenablaufs durchgeführt. Dies hat zur Folge, dass zu Beginn zunächst die Impulsantwort transformiert wird, bevor eine erste Berechnung und somit eine erste Ausgabe von $y[n]$ erfolgt. Für eine Umsetzung als Audioeffekt würde dies bedeuten, dass dieser erst nach einer geringen Zeit beginnt, eine Ausgabe zu erzeugen. Dies wäre aber für ein digitales Effektgerät keine Seltenheit, eine Systeminitialisierung würde schon allein durch den Prozessor durchgeführt. Die Verlängerung durch eine Berechnung der Impulsantwort würde zwar zusätzliche Zeit kosten, diese würde dafür aber im laufenden Prozess keine zusätzliche Rechenleistung beanspruchen. Der wesentlich größere Vorteil ist dabei die Ersparnis beim Rechenaufwand, daher wird dieser Weg gewählt.

Die Impulsantwort muss dem Prozess daher zu Beginn komplett vorliegen. Bei einer Implementierung könnte dies durch Dateien realisiert werden, welche im Speicher abgelegt werden. Wird der Prozess gestartet, wird die Impulsantwort Segment für Segment durchlaufen und transformiert. Dies wird im Programmiertext mit einer for-Schleife realisiert. Diese benötigt neben der Impulsantwort im Zeitbereich auch eine Information über dessen Gesamtlänge L_h und die Segmentlänge N. Diese Parameter müssen entsprechend als Variablen vorliegen. Zusätzlich muss ein Speicherbereich definiert werden, in den die transformierten Segmente der Impulsantwort eingefügt werden. Dieser wird als Array ausgelegt, dessen Grenzen sich aus den Betrachtungen bei der Speicheranalyse ergeben. Ein Laufindex für Schleifen während des Programmablaufs muss ebenfalls definiert werden. Der verwendete Pseudocode ist an die später in der Thesis verwendete Programmiersprache C angelehnt. Die Impulsantwort ist in einer zuvor geladenen Datei abgelegt, welche als Array *impulse_res* bereits zugeordnet ist. Das Array zur Ablage der transformierten Impulsantwortsegmente wird mit Nullen initialisiert.

Die Impulsantwort wird vor der Transformation noch durch eine weitere Schleife durchlaufen. Dabei wird überprüft, ob in der Impulsantwort eine Vollaussteuerung vorliegt, welche zu Verzerrungen führen würde, wenn die Faltung durchgeführt wird. Dazu wird *impulse_res* durchlaufen, wobei geprüft wird, ob der Wert eins oder größer erreicht wird, welcher für eine Vollaussteuerung steht. Ist dies der Fall, wird entsprechend gedämpft, sodass Übersteuerung im Ausgangssignal vermieden wird. Dies wird mit einer segmentweisen Division erreicht, bei der jedes Element der Impulsantwort durch den Maximalwert geteilt wird.

```
/*Variablen-Deklaration*/

int i;
int N;
int Lh;
float over;
float HI[2N];
float imp_freq[Lh/N][2N];

/*Variablen-Initialisierung*/
```

```
N = gewaehlte Segmentlaenge
Lh = Laenge von impulse_res;
imp_freq = Null fuer (0,0) bis (Lh/N-1, 2N-1);
over = 0;

/*Suche nach moeglichen Uebersteuerungen und Anpassung*/

for i = 0 bis Lh
    if impulse_res[i] >= 1 UND impulse_res[i] > over
        over = impulse_res[i];
    end
end

if over >= 1
    impulse_res = impulse_res ./ over;
end

/*Transformation und Speichern der Impulsantwort*/

for i = 0 bis (Lh/N)-1
    hi = impulse_res von i*N bis (i+1)*N-1;
HI = FFT von hi mit Laenge 2N;
imp_freq = HI fuer i*N bis (i+1)*2N -- 1;
end
```

Die transformierte Impulsantwort liegt nun segmentweise im Array *imp_freq* und kann von dort unter Angabe der entsprechenden Zeile des Arrays direkt aus dem Speicher geladen werden. Somit liegen die Segmente $H_i[f]$ bei den Multiplikationen mit den Segmenten $X_k[f]$ schon vor und es muss keine zusätzliche Berechnung im laufenden Prozess durchgeführt werden.

Die Segmentlänge N muss durch den Benutzer festgelegt werden. Anhand der Segmentlänge wird, wenn man den Blick von Arbeits- und Speicheraufwand abwendet, auch die Systemlatenz eingestellt.

5.2 Transformieren und Speichern von Eingangssegmenten

Bei den Eingangssegmenten lässt sich eine der wesentlichen Speicheroptimierungen umsetzen, wenn die nicht mehr benötigten Segmente des Eingangssignals überschrieben werden. Dies wird erreicht, wenn ab dem Zeitpunkt der maximalen Überlagerung wieder in den Speicherabschnitt geschrieben wird, welcher zuvor das erste Segment enthielt. In der Praxis wäre dieser Vorgang mit einem hohen Mehraufwand verbunden, weil ständig innerhalb eines Arrays der Eingangswerte gesprungen werden müsste. Das Array für die transformierten Eingangssignale wird daher genau so aufgebaut wie jenes der Impulsantwort. Es ändert sich im Vergleich der Verarbeitung lediglich, dass keine Schleife benötigt wird, weil immer nur ein Segment pro N Abtastwerten eintrifft. Die-

ses wird direkt transformiert und im Array *in_freq* abgelegt. Dabei gibt ein Zeiger, welcher als einfache Integer-Variable ausgelegt ist, den zu belegenden Speicherplatz im Array an. Jenes wird von der ersten bis zur letzten Reihe gefüllt, anschließend wird ab der ersten Reihe wieder überschrieben. Da jenes immer erst im Folgeablauf passiert, kann es hier nicht zu einem verfrühten Überschreiben der Daten kommen. Der Zeiger für das Array muss entsprechend immer beim Erreichen der letzten Reihe des Arrays wieder auf die erste zurückgesetzt werden. Der Kontrollmechanismus zur Abfrage des Zeigerstands und eines eventuellen Rücksetzens des Zeigers wird mit einer if/else - Verzweigung realisiert, welche als Bedingung die Anzahl der Segmente der Impulsantwort abprüft und bei Gleichheit mit dem Zeigerstand entsprechend das Zurücksetzen einleitet.

Das kontinuierliche Einlesen des Eingangssignals wird bei Audioprozessen in der Regel durch einen Puffer realisiert. Dabei wäre es ideal, wenn dieser immer Segmente der Länge N einlesen und an den Prozess der segmentierten Faltung weitergeben würde. Sollte dies nicht der Fall sein, würde dies für den Fall, dass der Puffer größer als die Segmentlänge N ist, eine erhöhte Latenz bedeuten, weil die Berechnung nicht im gleichem Zeitabstand die Eingangssegmente erhält. Ein zeitlicher Ablauf würde so an die Puffergröße gebunden sein. Ist der Puffer kleiner als N wäre dies kein Problem, es müsste dann lediglich vor der Übergabe an den Prozess ein entsprechender Eingangsvektor programmiert werden, welcher N Abtastwerte lang ist und weitergegeben wird, wenn er komplett gefüllt ist.

Dieser Eingangsvektor wird im Pseudocode mit dem Namen *xk* bezeichnet. Dabei wird davon ausgegangen, dass die Werte eines Segments pro Durchlauf an den Prozess übergeben werden, was durch ein Einlesen der Werte eines Audiostreams realisiert wird. Dieser erhält im folgenden Codebeispiel die Bezeichnung *audiostream* und als Parameter die Abtastrate, Wortbreite und die Segmentlänge N. Die Umsetzung an dieser Stelle wird je nach genutzter Audioanbindung variieren. Eine Übersteuerung im Eingangssignal zu erkennen und zu melden könnte an dieser Stelle einprogrammiert werden, normalerweise findet diese aber direkt am Eingang des Systems statt, deswegen wird an dieser Stelle darauf verzichtet. Um die Variablen und deren Zugehörigkeit zu den Teilprozessen ebenfalls zu zeigen, werden die Deklarationen für die Teilabschnitte jeweils einzeln aufgeführt. In den sich anschließenden Abschnitten der Thesis stehen diese dann als ein Block zu Beginn des Programmiertextes. Bereits in vorigen Abschnitten deklarierten Variablen sind der Übersicht halber nicht erneut aufgeführt.

```
/*Variablen-Deklaration*/

int s;
float xk[N];
float in_freq[Lh/N][2N];

/*Variablen-Initialisierung*/

in_freq = Null fuer (0,0) bis (Lh/N-1, 2N-1);

/*Einlesen des Audiostreams*/
```

```
xk = audiostream(fa, S, N);

/*Transformation und Speichern des Eingangssignals*/

in_freq[s] = FFT von xk mit Laenge 2N;

/*Zeiger fuer die Speicherposition*/

if(s == Lh/N)
    s = 0;
else
    s = s + 1;
end
```

5.3 Multiplikationen mit den Segmenten $H_i[f]$ und Speichern der Ergebnisse

Die Multiplikation der transformierten Eingangssegmente $X_k[f]$ mit den transformierten Segmenten der Impulsantwort $H_i[f]$ erfolgt ebenfalls alle N Abtastwerte. Dabei werden jeweils die Produkte $Y_{k1}[f]$ bis $Y_{k-ti}[f]$ berechnet. Um die Faktoren aus dem Speicher zu lesen, werden ihre Positionen benötigt. Zusätzlich muss angegeben werden, wie viele Multiplikationen im aktuellen Durchlauf berechnet werden müssen. In Abb. 20 ist ein Auszug aus dem Gesamtprozess zu sehen, in welchem der Ablauf für eine Multiplikationsfolge zu sehen ist, wie sie alle N Abtastwerte durchgeführt wird.

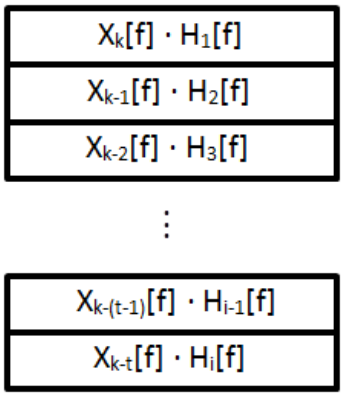

Abbildung 20: Ablauf der Multiplikationen für den k-ten Durchlauf

Auch hier wird mit einer for-Schleife gearbeitet, weil die Anzahl der durchzuführenden Multiplikationen im Voraus bekannt ist. Ebenso ist aus der Abbildung zu entnehmen, dass die Multiplikationen immer nach einem Schema durchgeführt werden, bei dem als erstes Element die Multiplikation des aktuellen Eingangssegments $X_k[f]$ mit dem ersten Segment der Impulsantwort $H_1[f]$ berechnet wird. Es folgt im zweiten Ablauf die Multiplikation von $X_{k-1}[f] \cdot H_2[f]$. Die for-Schleife kann durch

ihren Laufindex daher direkt den Speicherplatz des Segments der Impulsantwort angeben, welches aktuell benötigt wird. Je nachdem, wie viele Segmente sich zu einem Zeitpunkt im Prozess befinden, läuft die Schleife dann durch den Speicher mit den transformierten Segmenten $H_i[f]$ und liefert durch den Laufindex die Speicherpositionen. Für die transformierten Eingangssegmente $X_k[f]$ muss die Schleife von ihrem letzten Wert an herunterzählen, um das korrekte Segment $X_k[f]$ zu laden. Dabei ist der Startpunkt dieses Herunterzählens jedoch nicht jedes Mal gleich wie bei der Impulsantwort. Die Eingangssegmente werden nach bestimmter Zeit überschrieben, deswegen sind die benötigten Segmente $X_k[f]$ an den zuletzt neu beschriebenen Stellen des Arrays *in_freq* zu finden. In dieses Array werden die Segmente mit Hilfe des Zeigers in der Variable s zugewiesen. Wird s nach dem Zuweisen des aktuellen Segments betrachtet, so enthält die Variable zu diesem Zeitpunkt den korrekten Speicherplatz für die erste im Anschluss durchzuführende Multiplikation. Die Variable s kann allerdings nicht als Zähler für die for-Schleife der Multiplikationen genutzt werden, weil ihr Wert für die Zuweisung des nächsten Eingangssegments benötigt wird. Deswegen wird der Wert von s vor dem Beginn der for-Schleife einer Variable t übergeben, welche die Speicherplätze für die Eingangssegmente während der Multiplikationen angibt. Diese Variable muss im Gegensatz zur Variable s innerhalb der Schleife heruntergezählt werden, hinzukommt, dass sie nach Erreichen der ersten Zeile von *in_freq* wieder in die letzte Zeile springen muss. Ein weiterer Mechanismus wird in Form der Variable k eingeführt. Diese zählt solange pro Durchlauf des Prozesses um eins hoch, bis die maximale Überlagerung erreicht ist. Ab diesem Zeitpunkt bleibt k dann konstant. Die Variable k kann damit sowohl die for-Schleife für die Multiplikationen steuern als auch im Overlap-Add Verfahren genutzt werden. Um die Ergebnisse der Multiplikationen direkt zu speichern, wird nach der Multiplikation direkt die Transformation zurück in den Zeitbereich per IFFT durchgeführt. Anschließend wird in einem Array *out_seg* das Ergebnis abgelegt. Dabei muss ein weiterer Zeiger die Zuweisung übernehmen, sodass dieses Array genau wie am Eingang kontinuierlich von vorne bis hinten beschrieben wird. Bei maximaler Überlagerung sind an den Ergebnissen des Overlap-Add Verfahrens $\frac{2 \cdot L_h}{N}$ Segmente beteiligt. Deswegen muss das Array *out_seg* eben jene Anzahl an Zeilen der Länge $2N$ haben, um alle aktuell beteiligten Segmente $y_{ki}[n]$ zu umfassen. Der Zeiger für die Zuweisung der Segmente $y_{ki}[n]$ wird mit der Variable l realisiert, welche in der Multiplikationsschleife hochgezählt wird und anschließend wieder auf den Wert $l = 1$ zurückgesetzt wird. Pro Durchlauf werden bei maximaler Überlagerung genau die Hälfte der Zeilen von *out_seg* neu beschrieben. Die vorigen bleiben dabei unberührt. Somit ist genau jener Zustand vorhanden, wie ihn das Overlap-Add Verfahren verlangt. In Abb. 21 ist zu sehen, wie die Zeigervariablen auf die Segmente verweisen und sich daraus die Multiplikationen zusammensetzen.

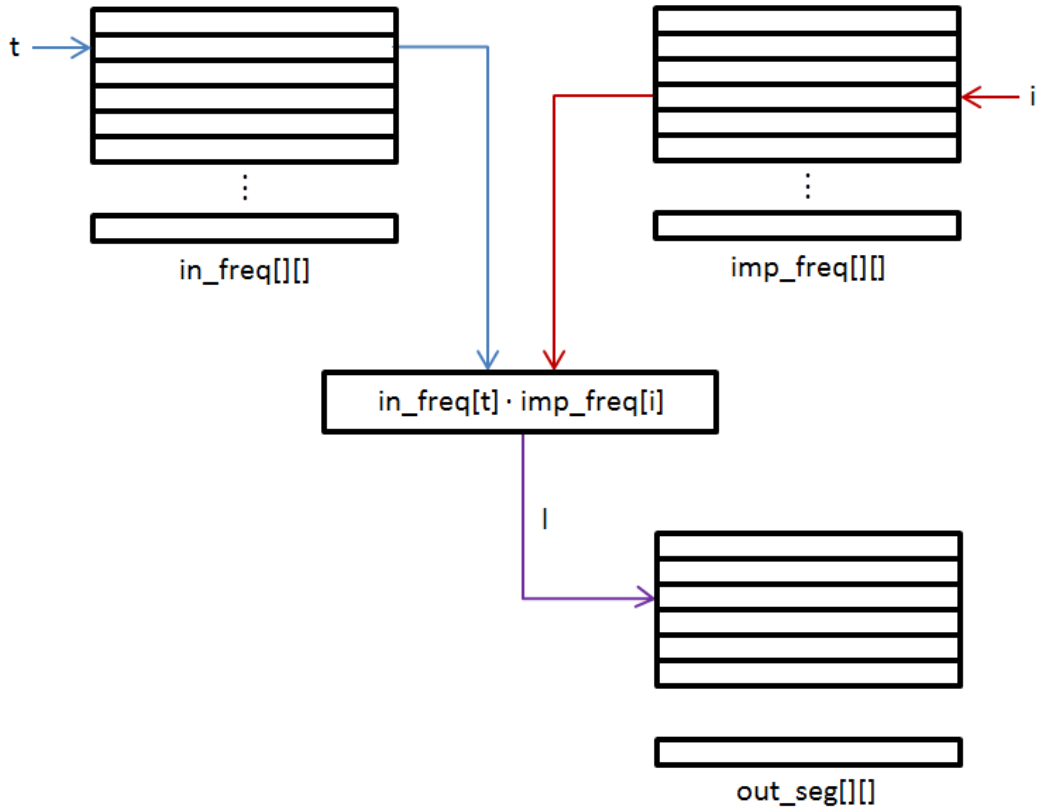

Abbildung 21: Zeigerpositionen für einen Multiplikationsablauf

```
/*Variablen-Deklaration*/

int t;
int l;
int k;
float Yki[2N];
float out_seg[2*(Lh/N)][2N];

/*Variablen-Initialisierung*/

k = 1;
l = 1;
out_seg = Null fuer (0,0) bis (2*Lh/N-1, 2N-1);

/*Multiplikation, IFFT und Speichern der Ergebnisse*/

t = s;
for i = 1 bis k
    Yki = in_freq[t] .* imp_freq[i];
    out_seg[l] = IFFT von Yki mit Laenge 2N;
    l = l + 1;
```

```
    t = t - 1;
    if t == 0
        t = Lh/N;
    end
    if(l == 2*Lh/N+1)
        l = 1;
    end
end

/*Prozessablauf-Variable*/

if k < Lh/N
    k = k + 1;
end
```

5.4 Overlap-Add und Ausgabevektoren

Das Overlap-Add Verfahren wird in zwei Teile aufgeteilt, um die Umsetzung in den Pseudocode durchzuführen. Der erste Bereich ist der Beginn des Verfahrens, welcher bis zum Übergang zur maximalen Überlagerung läuft. Anschließend läuft die maximale Überlagerung bis der Prozess beendet wird. Ein Ende des Prozesses zu realisieren macht für die geplanten Anwendungen keinen Sinn, weil ein System, welches die segmentierte Faltung umsetzt, dann nicht einfach ausgeschaltet werden könnte, sondern bis zum Ende laufen müsste. Dies wäre in der Praxis eher von Nachteil, deswegen läuft die maximale Überlagerung durch.

Zunächst wird nun der Beginn des Verfahrens umgeformt und dargestellt, anschließend wird dann auf die maximale Überlagerung eingegangen. Zu Beginn des Prozesses gibt es nur ein Eingangssegment $x_1[n]$, somit kann auch nur ein Ausgangssegment $y_{11}[n]$ vorliegen. Dieses wird im Abschnitt $0-(N-1)$ direkt ausgegeben. Eine Addition entfällt, somit kann zu diesem Zeitpunkt zusätzlicher Aufwand gespart werden. Der Sonderfall ist genau dann gegeben, wenn die Variable k den Wert $k=1$ hat, im ersten Durchlaufen der Dauerschleife des Prozesses, welche später noch genauer betrachtet wird. Das Segment $y_{11}[n]$ wird direkt in den Ausgabevektor *y_out1* geschrieben, bevor der Prozess dann den zweiten Schleifendurchlauf beginnt. In diesem werden die nächsten N Werte des Ausgangssignals berechnet. Diese bestehen aus den Werten $y_{11}[N]-y_{11}[2N-1]$ und den Werten in $y_{12}[0]-y_{12}[N-1]$ sowie $y_{21}[0]-y_{21}[N-1]$. In diesem Durchlauf hat die Variable k den Wert $k=2$. Dies entspricht genau der Anzahl der im aktuellen Durchlauf berechneten und im Overlap-Add Verfahren benötigten Segmente. Des Weiteren werden $k-1$ Segmente aus dem vorigen Durchlauf benötigt. Dies wird über zwei for-Schleifen realisiert. Eine durchläuft die relevanten Segmente aus dem letzten Durchlauf, die andere jene aus dem aktuellen Durchlauf. Um den Schleifen zu übermitteln, ab wo die Segmente aus dem Array *out_seg* genutzt werden sollen, wird einer Variablen o der Wert $l-1$ übergeben. Die Variable o enthält nun die Zahl der zuletzt beschriebenen Reihe im Array *out_seg*. Den Wert l zu übergeben würde nicht zum korrekten Wert führen, weil jene

Variable schon in der Multiplikationsschleife um eins erhöht worden ist. Innerhalb der for-Schleifen werden zwei Operationen durchgeführt. Neben den Additionen muss die Variable o runtergezählt werden. Dies wird ähnlich wie bei den Segmenten $X_k[f]$ in den Multiplikationen realisiert. Die Variable wird so lange immer nach einer Berechnung um eins verringert, bis diese den Wert $o = 1$ erreicht. Bei diesem wird sie wieder auf den Index der letzten Zeile des Arrays *out_seg* gesetzt.

Die Additionsergebnisse landen in den Vektoren *y_out1* und *y_out2*. Diese sind der Einfachheit halber als Array *y_out[2, N]* realisiert, sodass nur ein Index geändert werden muss, um den anderen Vektor anzusprechen, nicht jedoch ein Variablenname. Ein Ergebnis ergibt sich aus der elementweisen Addition des aktuellen Wertes des angesprochenen Vektors mit einem Segment $y_{ki}[n]$ aus dem Array *out_seg*. Würde man dabei nur diese Addition einsetzen, würde der Prozess die Ausgangsvektoren nicht überschreiben, sondern immer weiter darin addieren, was zu unerwünschten Ergebnissen führen würde. Deswegen wird der aktuell zu beschreibende Vektor vor ersten Schleifendurchlauf zunächst mit Nullen beschrieben. Dies erspart eine weitere Verzweigung innerhalb der for-Schleifen, welche Zeit kosten würde, weil eine Bedingung abgeprüft wird, welche nur ein einziges Mal wahr wäre. Eine if-/else- Verzweigung prüft pro Durchlauf des Prozesses, welcher Ausgangsvektor im letzten Durchlauf beschrieben wurde und setzt entsprechend eine Variable out auf den Wert des anderen Vektors.

Der Blick auf die Ausgangsvektoren an dieser Stelle zeigt auch, dass es durchaus möglich wäre, nur einen Ausgangsvektor zu definieren und zu nutzen. Diese Optimierung ist im Programmcode möglich, weil hier davon ausgegangen werden kann, dass das Ergebnis in einem Ausgangsvektor nach dem Schreiben sofort ausgegeben wird. Durch den zyklischen Ablauf des Prozesses im Programmiertext dauert es daher lang genug, bis wieder Daten in den Vektor geschrieben werden. Ein Überschreiben, so wie es in den bisherigen Betrachtungen erläutert worden ist, kommt bei diesem Ansatz nicht vor. Bezüglich der Analysen ändert sich dadurch der Speicheraufwand, welcher sich um $2N$ verringert. Auffällig ist dies aber in den typischen Segmentlängen für die Echtzeitumgebung nicht, sodass die wesentliche Ersparnis bei Nutzung eines einzelnen Ausgangsvektors die if-/else-Verzweigung zu Beginn des Overlap-Add Verfahrens ist, welche abprüft, welcher der Vektoren das Ergebnis erhalten soll. Diese Optimierung wird ab dieser Stelle auch in den folgenden Programmiertexten verwendet, sodass es nur noch einen Vektor *y_out* gibt, welcher mit den Ergebnissen des Overlap-Add Verfahrens beschrieben und anschließend ausgelesen wird.

Die maximale Überlagerung bringt noch einen weiteren Zweig in die Berechnung. Liegt die maximale Überlagerung vor, erreicht die Variable k also den Wert $\frac{L_h}{N}$, so müssen aus dem vorigen Durchlauf genauso viele Segmente im Overlap-Add genutzt werden, wie im aktuellen. Zuvor war dieser Wert um Eins kleiner. Eine if/else-Verzweigung wird mit eben jenem Wert für die Variable k dazu programmiert, die maximale Überlagerung festzustellen und entsprechend auf eine andere Schleife zu schalten, welche die korrekte Anzahl an Segmenten durchläuft. Diese Schleife ist bis auf die obere Grenze des Laufindexes komplett identisch zu jener, welche vom Start bis zur maximalen Überlagerung beim Overlap-Add Verfahren durchlaufen wird.

```
/*Variablen-Deklaration*/
```

```
int o;
float y_out[N];

/*Overlap-Add*/

y_out = Null fuer [0] bis [N];
if k == 1
    y_out = out_seg[1] von [i] bis [N];
else
    o = l - 1;
    if o == 0
        o = 2 * Lh/N;
    end
    for i = 1 bis k
        y_out = y_out + out_seg[o] von [1] bis [N];
        if o == 1
            o = 2 * Lh/N;
        else
            o = o - 1;
        end
    end
    if k == Lh/N
        for i = 1 bis k
            y_out = y_out + out_seg[o] von [N+1] bis [2N];
            if o == 1
                o = 2 * Lh/N;
            else
                o = o - 1;
            end
        end
    else
        for i = 1 bis k -- 1
            y_out = y_out + out_seg[o] von [N+1] bis [2N];
            if o == 1
                o = 2 * Lh/N;
            else
                o = o - 1;
            end
        end
    end
end
```

5.5 Realisierung des dauerhaften Ablaufs

Um den Prozess dauerhaft betreiben zu können, wird eine while-Schleife definiert, welche nach dem Deklarieren und Definieren der Variablen immer wahr ist, sodass die gezeigten Prozessabläufe sich in Dauerschleife so lange wiederholen, bis das Programm durch Benutzereingriff beendet wird. Um Variablen zu sparen wird die Schleife mit dem Wert Eins initialisiert, sodass diese immer wahr ist. Die Schleife wird nach dem Transformieren und Speichern der Impulsantwort gestartet, sodass jener Prozess als System-Initialisierung betrachtet werden kann. Der Start des Dauerprozesses beginnt anschließend in der Schleife mit dem Einlesen des ersten Elements des Eingangssignals.

```
/*---System-Initialisierung---*/

/*Variablen-Deklaration*/
/*Variablen-Initialisierung*/
/*Suche nach moeglichen \Uebersteuerungen und Anpassung*/
/*Transformation und Speichern der Impulsantwort*/

while(1)

/*Einlesen des Audiostreams*/
/*Transformation und Speichern des Eingangssignals*/
/*Multiplikation, IFFT und Speichern der Ergebnisse*/
/*Overlap-Add*/
/*Zeiger fuer die Speicherposition*/
/*Prozessablauf-Variable*/

end\newpage
```

6 Umsetzung als Matlab®-Code

Die Umsetzung der segmentierten Faltung als MATLAB® Code dient als Zwischenstufe zum finalen C-Code und zur Analyse des Programmiertextes. Die Möglichkeiten des Tools umfassen eine genaue Auswertung der Ergebnisse und eine numerische Analyse der Ein- und Ausgabewerte sowie von Zwischenergebnissen. In C sind diese Auswertungen zwar auch möglich, würden aber einen zusätzlichen Programmieraufwand erfordern, welcher in der Umgebung von MATLAB® entfällt. Ebenso bietet MATLAB® eine Analyse der Rechenzeit an, welche als Vergleichswert genutzt werden kann, um die Ergebnisse der Analyse des Arbeitsaufwands auszuwerten.

Die Übersetzung des Pseudocodes in einen MATLAB® Code kann ohne größere Umwege erfolgen. Von Vorteil ist dabei die Umsetzung in Vektoren und Matrizen in der Umgebung. Diese ermöglichen eine einfache Umsetzung der benötigten Arrays und sind zudem einfach zu adressieren. Weitere bereits enthaltene Funktionen sorgen für eine unkomplizierte Umsetzung der komplexeren Strukturen des Prozesses. Hinzu kommt die enthaltene FFT-Bibliothek FFTW (Fastest Fourier Transformation in the West), welche in MATLAB® direkt über entsprechende Funktionen angesprochen werden kann. So entfällt ein umständliches Einbinden einer FFT-Bibliothek.

Zur Umsetzung wird der Pseudocode aus dem vorigen Kapitel noch einmal betrachtet und die Umformung auf den durch MATLAB® interpretierbaren Code beschrieben. Anschließend wird diese untersucht, wobei eine Gegenüberstellung mit einem Testbeispiel und der diskreten Faltung erfolgt. Die abschließende Analyse des MATLAB® Codes erfolgt sowohl audiotechnisch und numerisch.

6.1 Umformung in MATLAB® Code

Ein wesentlicher Vorteil in der Umgebung von MATLAB® ist die bereits eingebaute FFT-Bibliothek FFTW. Diese ist bereits in diversen Projekten umgesetzt und unterliegt der GNU General Public License. Dadurch ist diese kostenlos für nichtkommerzielle Anwendungen. FFTW ist eine C-Bibliothek, wodurch diese auch für den weiteren Verlauf dieser Thesis interessant wird. In MATLAB® wird FFTW durch die Befehle *fft([Daten], [Länge])* und *ifft([Daten], [Länge])* angesprochen. Ein kompliziertes Einprogrammieren einer FFT-Routine entfällt somit. FFTW rechnet je nach vorgegebener FFT-Blocklänge die angegebene FFT mit einem passenden FFT-Algorithmus. Bei der Nutzung von $N = 2^n$ wird daher je nach angegebener Segmentlänge in der segmentierten Faltung neben dem Radix-2 auch mit dem Radix-4 Algorithmus gerechnet. Vergleichsanalysen werden daher auch mit Werten durchgeführt, welche mit dem Radix-2 Algorithmus berechnet werden. Die MATLAB®-Funktion liefert für einen Eingangsvektor die zugehörigen Spektrallinien in einem Ausgangsvektor. Dieser kann über eine entsprechende Variable wieder angesprochen werden.

Die Impulsantwort und ihre Länge sollen im Prozess möglichst flexibel wählbar sein. Dies hat Auswirkungen auf die Berechnung von $\frac{L_h}{N}$. Dieser Wert würde nur für bestimmte N eine ganze Zahl ergeben und als solche verarbeitet werden. An einigen Stellen im Programmtext würden Kontrollmechanismen nicht mehr funktionieren, wenn keine ganze Zahl vorliegt. Ein Aufrunden auf eine ganze Zahl kann in

diesem Fall helfen, diese Probleme zu umgehen. Dies wird in MATLAB® durch die Funktion *ceil()* erreicht. Diese Funktion gibt es ebenfalls in C, was für die spätere Umformung in C von Vorteil ist, weil an dieser Stelle keine Änderungen nötig sind. Der Wert ceil($\frac{L_h}{N}$) enthält immer den nächsten ganzzahligen Wert nach dem Wert $\frac{L_h}{N}$. Diese Funktion muss an allen Stellen eingesetzt werden, an denen Arrays mit der Länge $\frac{L_h}{N}$ deklariert werden, außerdem in den Kontrollstrukturen, welche den Ablauf des Prozesses steuern. Ohne die Rundung würde der Prozess die Arrays nicht korrekt erstellen, weil bei der Deklaration von Arrays ein Integerwert erwartet wird und somit abgerundet werden würde.

Die Struktur von MATLAB® nutzt Vektoren und Matrizen als Ablage für Daten. Arrays im Pseudocode werden daher im System als Matrizen deklariert und entsprechend eingebunden. Das Nutzen der Daten aus Matrizen in MATLAB® funktioniert über die direkte Eingabe der gewünschten Datenbereiche. In Klammern wird dabei zunächst der Zeilenraum, anschließend der Spaltenraum angegeben, aus welchem die Daten gelesen werden sollen. Ein Auslesen durch eine Schleife muss daher nicht programmiert werden. Das folgende Beispiel aus dem MATLAB® Code zeigt den Ablauf der Multiplikationen der Segmente $X_k[f]$ mit den Segmenten $H_i[f]$. Darin kommen alle beschriebenen Abläufe, welche für die Umsetzung in MATLAB® genutzt werden, vor und können anhand des Programmiertextes betrachtet werden.

```
for i = 1:1:k
    Yki = in_freq(t, :).*imp_freq(i, :);
    out_seg(l, :) = ifft(Yki, 2*N);
    l = l + 1;
    t = t - 1;
    if t == 0
        t = ceil(Lh/N);
    end
    if(l == 2*ceil(Lh/N)+1)
        l = 1;
    end
end
```

6.2 Laufzeit-Analyse des MATLAB®-Codes

Die Analyse des Codes wurde auf zwei verschiedenen Plattformen durchgeführt, um auf diese Weise Aussagen zur Laufzeit besser differenzieren zu können. Die Laufzeit lässt sich in MATLAB® über die Funktionen *tic* und *toc* messen. Diese erfassen mit dem Befehl *tic* einen aktuellen Zeitstempel, welcher mit einem weiteren durch den Befehl *toc* verglichen wird. Der Unterschied in Sekunden wird anschließend ausgegeben. Die Funktionen sind für die Analyse so platziert, dass diese vor Beginn der while-Schleife den Zeitstempel auslösen. Dadurch ist die segmentweise Transformation der Impulsantwort nicht in der Zeitanalyse enthalten und wird als Teil der Initialisierung

betrachtet. Beendet wird die Zeitmessung nach dem Komplettdurchlauf, welcher in diesem Fall erreicht ist, wenn alle Segmente von $x\,[n]$ verarbeitet worden sind. Dazu wurde der Programmiertext um eine Kontrollstruktur erweitert, welche prüft, ob der entsprechende Durchlauf erreicht ist, in welchem das letzte Segment von $x\,[n]$ an letzter Stelle des Overlap-Add steht. Die Schleife wird dann an jener Stelle abgebrochen und der Zeitstempel durch die Funktion *toc* wird erfasst.

CPU:	Intel Core™ i7-3770 @ 3.40 GHz
Mainboard:	Gigabyte Z77-D3H
Speicher:	16 GB
Festplatte:	Samsung HD204UI (2 TB, S-ATA)
System:	Windows 7 x64

Tabelle 8: Technische Daten des Tower-PC Testrechners

Die Analyse-Computer sind zum einen ein PC zur Bearbeitung von Audio, zum Spielen von Games der neueren Generation und zur Büroarbeit. Die technischen Daten sind in Tab. 8 gezeigt.

Der Rechner läuft in Standardkonfiguration, weder Prozessor noch weitere Rechnerteile wurden oder werden auf höheren Taktraten als vorgegeben betrieben. Zur Analyse wurden außer MATLAB® alle Programme und Hintergrundanwendungen geschlossen, sodass die CPU-Leistung komplett für den Prozess zur Verfügung steht. Um die Analysewerte zu ermitteln, wurde der Prozess anschließend mit zwei Beispielsignalen bestückt. Einer Impulsantwort $h\,[n]$ und ein Eingangssignal $x\,[n]$, in beiden Fällen endliche Signale mit bekannter Länge. Dies hat den Vorteil, dass eine diskrete Faltung als Referenzwert für die segmentierte Faltung mit denselben Zahlenwerten berechnet wird wie die segmentierte Faltung selbst. Eine Abweichung in den Testsignalen kann somit direkt festgestellt werden. Bezüglich der Laufzeit kann durch die bekannten Größen L_x und L_h die Länge des Ausgangssignals festgestellt werden. Die ermittelte Laufzeit sollte auf jeden Fall kleiner sein als die zeitliche Länge des Ausgangssignals. Wäre dies nicht der Fall, wäre der Prozess nicht in der Lage, das Ausgangssignal $y\,[n]$ schnell genug zu berechnen. Das genutzte Eingangssignal ist ein Auszug aus einem Musikstück. Dieses wurde mit 16 Bit bei 44,1 kHz aufgenommen. Die Länge $L_x = 396.900$ entspricht bei 44,1 kHz genau neun Sekunden. Die genutzte Impulsantwort wurde in einem Raum mit hoher Nachhallzeit aufgenommen. Ihre Länge beträgt $L_h = 107.008$, also ungefähr 2,43 Sekunden. Die Wortbreite und Abtastrate sind bei beiden Signalen identisch.

Aus den bereits bekannten Größen kann nun schon eine Abschätzung für das Ausgangssignal $y\,[n]$ gemacht werden. Dessen Länge L_y kann bereits mit der Formel aus den Grundlagen der diskreten Faltung berechnet werden.

$$\begin{aligned} L_y &= L_x + L_h - 1 = 396.900 - 107.008 = 503.907 \\ \frac{L_y}{f_a} &= \frac{503.907}{44.100\frac{1}{s}} = 11,43\ Sek. \end{aligned} \tag{51}$$

Ein Durchlaufen des Prozesses bis zu jenem Punkt sollte den angegebenen Wert von 11,43 Sekunden nicht überschreiten, wenn eine Verarbeitung in Echtzeit durchgeführt werden soll. Der genutzte Speicher kann durch die bereits bekannte Formel für den Speicheraufwand vorhergesagt werden. Eine Analyse des genutzten Speichers durch MATLAB® ist nur manuell möglich. Dazu werden in dieser Analyse die Arrays anschließend noch einmal betrachtet und mit dem vorberechneten Wert verglichen. Funktionen innerhalb der Umgebung, welche den Speicher analysieren, können mit der verfügbaren Version MATLAB® R2009a nicht so durchgeführt werden, dass ein Wert für die Auslastung des Speichers nur für den angeforderten Prozess ausgegeben wird. Durch die Variablenübersicht kann die manuelle Analyse allerdings erfolgen. In der Prognose sollte der Speicher den Wert in (52) annehmen.

$$M_{ges} = 17 \cdot Lh + 8N = 1.819.136 + 8N \tag{52}$$

Für verschiedene N kann im Voraus nun bereits getestet werden, wie hoch der Einfluss der $8N$ auf das Gesamtergebnis ist. Für einen Wert $N = 4096$ sind $8N = 32768$, der Einfluss auf einen Wert von über 12 Millionen ist daher mit 2,5 % eher gering, wenn auch nicht unauffällig. Die Analyse für die Laufzeit wird neben der Speicherprognose auch an die Analyse des Arbeitsaufwands angelehnt. Sollte beim Speicher der tatsächliche Verlauf der Prognose entsprechen, so müsste in der Kurve der Laufzeit die Analyse des Arbeitsaufwands erkennbar sein. Dies würde bedeuten, dass es für kurze Segmentlängen wesentlich aufwändiger ist, den Prozess durchzuführen, als für die diskrete Faltung. In der Analyse des Arbeitsaufwands ist für Impulsantworten von $L_h = 2\ Sek.$ ein Übergang für $N = 8$ auf $N = 16$ festzustellen, ab welchem die segmentierte Faltung weniger Berechnungen benötigt als die diskrete Faltung.

Betrachtet man den Prozess in MATLAB® auf dem vorgestellten PC-System, erhält man ein anderes Ergebnis. Das liegt daran, dass verschiedene Prozesse, welche bei der Analyse des Arbeitsaufwands mit Standardanweisungen untersucht worden sind, auf der tatsächlichen Umsetzungsplattform individuell verarbeitet werden. Dadurch ist der Arbeitsaufwand als Analyse eine gute Prognose für den Verlauf der Kurve, welche die Laufzeiten enthält. In Tab. 9 sind die Ergebnisse der Zeitanalyse auf dem ersten System zu sehen. In der Tabelle wurde eine weitere Spalte mit der Grundlatenz eingefügt. Diese enthält für die angegebenen Werte für N die Dauer eines Segments, wenn die Abtastfrequenz mit $f_a = 44,1\ kHz$ angenommen wird.

Deutlich zu sehen ist, dass eine Umsetzung eines Echtzeitprozesses mit einer Grundlatenz $<$ 5ms nicht möglich ist. Diese würde eine Segmentlänge von $N = 128$ als Maximum erfordern. Diese benötigt jedoch sechs Mal so viel Laufzeit, wie eigentlich für eine Umsetzung in Echtzeit zur Verfügung stellen würde. Erst ab $N = 1024$ ist eine Verarbeitung möglich, welche schneller als die Dauer des Ausgangssignals erfolgt. Mit dieser Segmentlänge würde die Grundlatenz bereits 23,2 ms betragen, was im Audiobereich schon fast einem Echo gleich kommt. Durch einzelne kleinere Optimierungen konnten die Werte zwar verbessert werden, allerdings konnte die Berechnungszeit für $N = 512$ nicht unter die Vorgabe von $\frac{L_y}{f_a} = 11,43\ Sek.$ gesenkt werden. In Abb. 22 ist die Laufzeitkurve des Prozesses auf der Plattform zu sehen.

N	Grundlatenz [Sek.]	Laufzeit i7 [Sek.]
1	0	$45,12$
32	$7,26 \cdot 10^{-4}$	$302,75$
64	$1,45 \cdot 10^{-3}$	$140,21$
128	$2,90 \cdot 10^{-3}$	$65,17$
256	$5,80 \cdot 10^{-3}$	$29,81$
512	$1,16 \cdot 10^{-2}$	$12,84$
1024	$2,32 \cdot 10^{-2}$	$5,22$
2048	$4,64 \cdot 10^{-2}$	$2,32$
4096	$9,29 \cdot 10^{-2}$	$1,17$
8192	$1,86 \cdot 10^{-1}$	$0,93$

Tabelle 9: Laufzeit auf i7-System

Die Werte für $N = 2$ bis $N = 16$ wurden nicht ermittelt, weil MATLAB® die Berechnungen nicht ohne Absturz durchgeführt hat. Da jene Werte aber ohnehin schon bei der Analyse des Arbeitsaufwands höher ausfallen als die Berechnung der diskreten Faltung, wird auf eine Interpolation der Kurve verzichtet.

Als zweite Plattform zum Testen des MATLAB® Codes dient ein mobiler Rechner, welcher im Jahre 2009 fabriziert wurde. Das Gerät ist ausgelegt für Büroarbeiten und Games mit mittleren Anforderungen in einem zur damaligen Zeit aktuellen Rahmen. Eine Übersicht über die Systemkomponenten ist in Tabelle 10 zu sehen.

CPU:	Intel Pentium Dual-Core T4300 @ 2,1 GHz
Mainboard:	Intel ICH9
Speicher:	4 GB
Festplatte:	Western Digital WD5000BEVT (500 MB, S-ATA)
System:	Windows 7 x64

Tabelle 10: Technische Daten des mobilen PC-Testrechners

Die Tests zur Ermittlung der Laufzeit des Prozesses für ein festes L_x und ein festes L_h wurden unter gleichen Bedingungen wie auf dem ersten System durchgeführt. Dazu wurde der Prozess wieder für verschiedene Segmentlängen durchgeführt. Anschließend wurden die Ergebnisse notiert und mit denen vom ersten System verglichen. Dabei ist auffällig, dass die Zusammenhänge zwischen den einzelnen Segmentlängen N auf dem mobilen System sich zu denen vom Tower-PC mit i7-Prozessor unterscheiden. Wo beim i7-Prozessor bei kleineren Segmentlängen mehr als ein Faktor zwei zwischen den Werten liegt, ist dies beim Pentium Prozessor ein Faktor kleiner zwei. Zu erwarten war bereits im Voraus, dass der Pentium Prozessor insgesamt langsamer ist. Dass hingegen eine Halbierung der Segmentlänge nicht zu einer Verdopplung der Laufzeit führt, war nicht zu erwarten. Der Unterschied entsteht vermutlich durch die verschiedenen Prozessorstrukturen und die Verarbeitung des Prozesses auf der CPU. Im Vergleich der beiden Laufzeitkurven der Systeme (Abb. 22) wird deutlich, dass der Verlauf der

Kurve des Pentium Prozessors nahezu gleich jenem des i7-Prozessors ist. Die Kurve des Pentium Prozessors ist fast gleich einer nach oben verschobenen i7-Kurve.

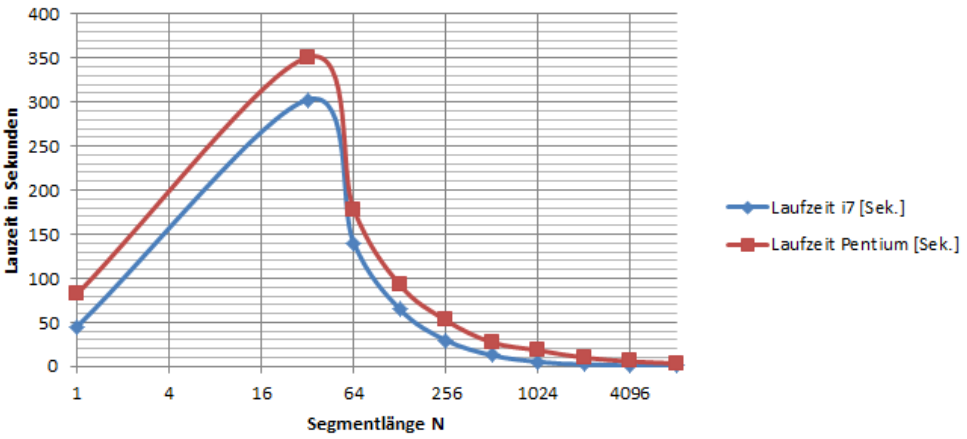

Abbildung 22: Vergleich der Laufzeit auf beiden Testsysteme

Zusammenfassend bleibt festzustellen, dass selbst ein Mitte 2012 aktueller Prozessor (Intel® i7) den Prozess mit diesem Code in MATLAB® nicht bei den entsprechenden Segmentlängen für eine Echtzeitverarbeitung so umsetzen kann, dass die Berechnungszeit unter der Vorgabe der Ausgabedauer bleibt. Die bestmögliche Umsetzung mit dem Code auf der i7-Plattform ist bei der Blocklänge $N = 1024$ vorhanden, dabei entsteht allerdings beim Laden des ersten Segments schon eine Latenz von 23,2 ms.

6.3 Ergebnisanalyse des MATLAB®-Codes

Um das Ergebnis des Prozesses analysieren zu können, wird der Wert des Ergebnisvektors *y_out* ausgewertet und mit dem Ergebnis der diskreten Faltung verglichen. Dieses geschieht mit einem Differenzvektor, welcher elementweise beide Vektoren voneinander subtrahiert. Das Ergebnis kann anschließend als Grafik ausgewertet werden. Um das Ergebnis der segmentierten Faltung über einen Zeitraum auszuwerten, wird der bestehende Code um einige Zeilen erweitert. Es wird ein weiteres Array *out* hinzugefügt, indem die Werte des Ergebnisvektors übernommen werden. Dieses Array wird mit Hilfe eines Zeigers beschrieben, welcher pro Durchlauf um eins weitergestellt wird. Das Array *out* wird so eingerichtet, dass alle relevanten Segmente bis zur Ausgabelänge $L_y = L_x + L_h - 1$ erfasst werden können. Dadurch ist ein Vergleich mit den Werten des Ausgangssignals der diskreten Faltung zu jedem Zeitpunkt möglich. Dieses wird über den Befehl *conv()* in MATLAB® berechnet und als eigener Vektor *y_conv* abgelegt. Dieser Vektor wird anschließend mit dem Vektor *out* berechnet. Um diesen zu erhalten, wird die Matrix mit dem Array *out* um skaliert, sodass nur noch eine Spalte übrig bleibt, in welcher die Werte des Ausgangssignals nun zeilenweise entnommen werden können. Dazu sind zwei Befehle nötig, ein direktes Skalieren würde zu einer falschen Lösung führen. So muss zunächst ein Zeilenvektor erstellt werden, welcher

anschließend zum Spaltenvektor gewandelt wird, um mit dem Spaltenvektor *y_conv* verglichen zu werden.

Aus beiden Vektoren werden nun 100.000 Werte entnommen und als eigene Vektoren abgespeichert. Anschließend wird durch eine elementweise Subtraktion die Differenz zwischen den Einzelwerten in einem neuen Vektor *diff* ablegt. Dabei ist der Minuend der Vektor *out*, sodass die Differenz zum korrekten Faltungsergebnis *y_conv* gezeigt wird. Eine grafische Auswertung (Abb. 23) zeigt den Verlauf des Vektors diff für den Vergleich der ersten 100.000 Werte beider Vektoren.

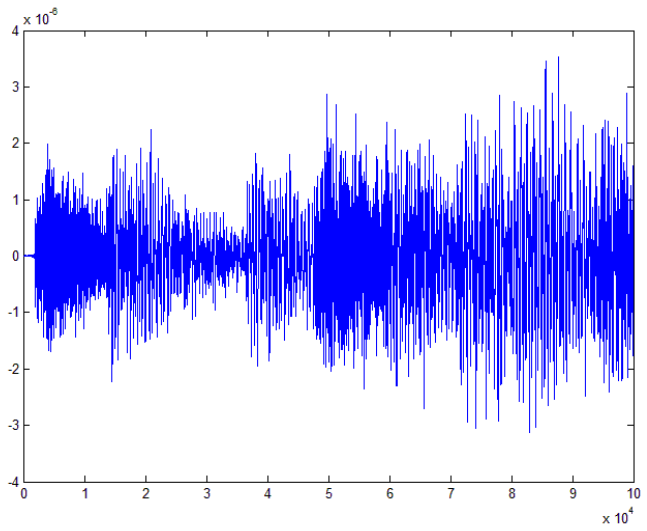

Abbildung 23: Differenz zwischen *y_conv* und *out* über die ersten 100.000 Werte beim MATLAB®-Code

Die Abbildung zeigt, dass es durchaus Unterschiede zwischen den Signalen gibt. Diese liegen im Bereich von 10^{-6}, sind also als gering einzustufen, wenn der Wert eins als Vollaussteuerung angenommen wird. Wird der Differenzvektor durch MATLAB® im PCM Format als wav-Datei ausgegeben und abgespielt, hört man von dem Vektor bei Studiopegel (+ 4 dBu) nichts. Normalisiert man diesen, so ist eine Art Faltungsergebnis zu hören, welche aber nicht dem korrekten Ergebnis entspricht. Die Normalisierung, welche genutzt wurde, um die Prozesse vor Übersteuerung zu sichern, ist dafür nicht verantwortlich. Bei der diskreten Faltung und der segmentierten Faltung ist die Normalisierung mit der Impulsantwort durchgeführt worden. Eine Skalierung am Ausgang könnte bei den Tests zwar genutzt werden, allerdings kann dies bei der segmentierten Faltung im laufenden Prozess nur segmentweise passieren, was zu einem schwankenden Ausgangspegel führen würde. Vergleicht man in der Variablenübersicht in MATLAB® die Minimal- und Maximalwerte der Vektoren *y_conv* und *out* im Bereich von 1 bis 100.000, so werden gleiche Werte angezeigt, welche sich beide in den Aussteuerungsgrenzen befinden. Das zeigt, dass ein Anpassen der Pegel am Ausgang nicht notwendig ist.

Die Unterschiede sind daher auf den Berechnungsablauf zurückzuführen. Vergleicht man die ersten Werte beider Vektoren, so wird deutlich, dass die Ausgabe

der segmentierten Faltung zu Beginn nicht den Wert Null liefert, obwohl dieser laut der disktreten Faltung vorliegen müsste. Der Wert ist aber mit 10^{-18} so gering, dass er, je nach Quantisierung, als Null angenommen werden kann.

Ein grafischer Vergleich von *y_conv*, *out* und *diff* zeigt, dass der Signalverlauf von *diff* den Verlauf des Faltungsergebnisses wiederspiegelt, allerdings phasengedreht (Abb. 24). Dies erklärt auch den Unterschied, welcher beim Abspielen von *diff* in der Audiobearbeitung wahrgenommen werden konnte.

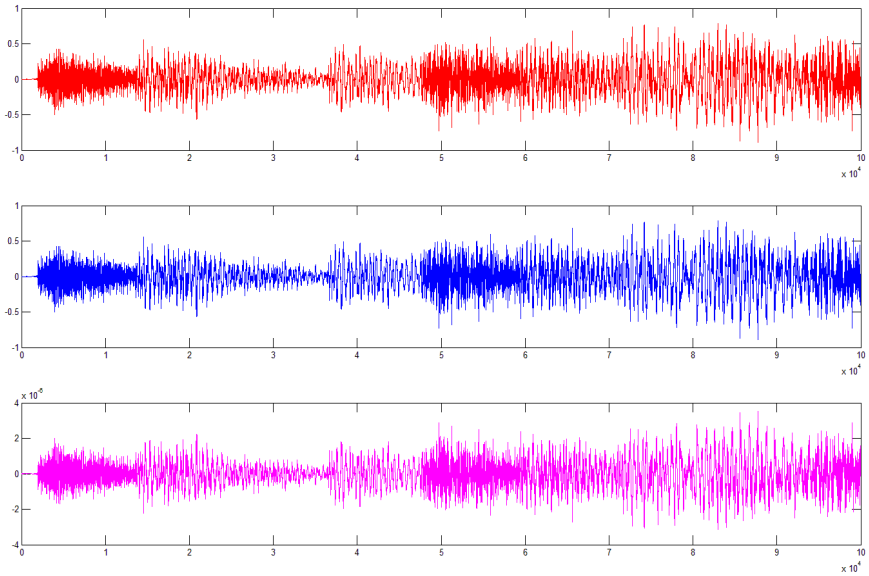

Abbildung 24: Vergleich der Vektoren *y_conv*, *out* und *diff* (von oben nach unten)

Um auch einen Vergleich der berechneten Ergebnisse in einer audiotechnischen Umgebung durchzuführen, wird mit der Software Audition von Adobe ein Test durchgeführt, bei dem beide Ergebnisse über einen Korrelationsgradmesser verglichen werden konnten. Dazu wird eine Stereospur erstellt, welche auf dem linken Kanal den Vektor *y_conv* abspielt, im rechten Kanal wird der Vektor *out* eingefügt. Die Korrelation der Kanäle kann anschließend über ein Werkzeug über der angezeigt werden. Nach einem kurzen Einschaltimpuls bleibt diese konstant bei Eins, was dafür spricht, dass der minimale Unterschied, der durch die Berechnungen entstanden ist, nicht hörbar ist und somit auch kein Einfluss in der Ausgabe zu sehen ist.

Als Ergebnis dieser Analyse wird festgehalten, dass es einen numerischen Unterschied zwischen dem Ausgangssignal der diskreten und jenem der segmentierten Faltung gibt, welcher sich im Bereich 10^{-6} aufhält. Dieser ist jedoch bei einer audiotechnischen Analyse nicht zu ermitteln, sodass der Einfluss auf die Ausgabe als sehr gering bezeichnet werden kann.

7 Umsetzung als C-Code

Bei der Umsetzung in der Programmiersprache C wird anhand des zuvor gezeigten MATLAB®-Codes hergeleitet und umgeformt. Dabei müssen im Wesentlichen alle Funktionen und Methoden, welche in MATLAB® zu einer Vereinfachung des Ablaufs führen, wieder entfernt werden. Ebenso müssen Kontrollstrukturen angepasst werden, weil die Syntax sich von jener in MATLAB® unterscheidet. Die Umformung in C erfolgt daher eher aus der Sicht der Portierbarkeit und Universalität des Programmiertextes, weniger zur Analyse des Programms.

Die Umsetzung als C-Code wird in diesem Abschnitt ähnlich wie im vorigen jene in MATLAB® beschrieben werden. Dabei wird auch auf den seit 2011 in MATLAB® vorhandenen Coder und die Umsetzung in C mit Hilfe dieses Werkzeugs gezeigt werden. Anschließend erfolgt die manuelle Umformung des vorhandenen Codes in die Programmiersprache C. Die Analyse des C-Codes mit Hilfe von MATLAB wird ebenfalls vorgestellt.

7.1 Automatische Umformung mit MATLAB® Coder

In MATLAB® gab es bis 2011 keine Möglichkeit, die selbst geschriebenen Funktionen und Skripte direkt in C umzuformen. Dazu wurde auf zusätzliche Software zurückgegriffen, welche diese Aufgabe übernahm und dem Benutzer eine Ausgabe des Programmiertextes in C ermöglichte. Dazu wurde der Programmtext aus MATLAB® mit Hilfe von C-Strukturen nachgebildet und anschließend ausgegeben. Ein Kompilieren funktionierte dann über einen Compiler nach Wahl und entsprechende Software. Mit der Version 2011a wurde in MATLAB® die Funktion Coder eingeführt, welche es ermöglicht, die Umwandlung in C direkt in der Software durchzuführen. Dabei werden allerdings nur Funktionen unterstützt. Der Coder wird über die Kommandozeile eingestellt, wobei direkt ein Compiler zugewiesen wird. Dadurch ist es möglich, neben C- und C++-Bibliotheken direkt ausführbare Dateien zu erzeugen.

Für den MATLAB®-Code der segmentierten Faltung wurde der Coder in der MATLAB® Version 2013 verwendet. Dabei musste zunächst eine Umformung des als Skript vorliegenden Programmtextes in eine Funktion erfolgen. Dazu muss neben der Variablenübergabe auch eine Angabe über den Inhalt der Variablen erfolgen. MATLAB® benötigt in den internen Skripten und Funktionen keine Angabe über den enthaltenen Datentyp. So kann eine Integer-Variable und eine Double-Variable direkt mit Daten des entsprechenden Typs beschrieben und verarbeitet werden. In C hingegen muss jede Variable zunächst deklariert werden, wobei auch der Datentyp angegeben werden muss. Ein Initialisieren kann aber direkt in der gleichen Zeile erfolgen. Der Coder benötigt zusätzlich noch eine Information über die Array-Größen, welche im Programm entstehen. Diese können bei der segmentierten Faltung erst dann angegeben werden, wenn die Länge der Impulsantwort und die Segmentlänge vorliegen. Die Arrays werden dem Coder daher als Arrays mit unbekannter Länge übergeben und entsprechend umgeformt. Die ersten Versuche, den Programmiertext auf diese Art und Weise umzusetzen, wurden durch Fehler abgebrochen, weil die Variablenzuordnung für komplexe Werte ebenfalls angepasst werden musste. Ist ein

Wert einmal komplex, muss dieser zur Übergabe an ein Array, welches mit reellen Werten initialisiert wurde, umgerechnet werden, sodass auf beiden Seiten der Zuweisung der gleiche Wertebereich eingestellt ist. Nach dem Setzen der Variablen und der Anpassung der Wertebereiche an den fehlerhaften Stellen muss noch der Compiler eingestellt werden. Auf der genutzten Intel® i7 Plattform, welche auch schon zur Analyse des MATLAB®-Codes verwendet wurde, muss wegen des installierten 64-Bit Betriebssystems noch ein Compiler installiert werden, weil für die 64-Bit Version von MATLAB® dieser nicht mitgeliefert wird. Dazu wird der Microsoft Compiler des Software Development Kits 7.1 (SDK) verwendet. Dieser wird von Mathworks für diese Version empfohlen und ist kostenlos im Internet erhältlich. Der Compiler wird in MATLAB® anschließend zugewiesen und eingebunden. Dies funktioniert mit einer Befehlsfolge, welche den Coder-Setup einrichtet. Anschließend wird der Programmtext durch den Coder in C überführt.

Das Ergebnis dieser Umformung durch den Coder ist allerdings wenig zufriedenstellend. Es wird eine Vielzahl von Dateien erstellt, welche Umformungen und Abläufe enthalten. Der Hauptablauf ist durch die Umformung auf diese Dateien verteilt, was den Code in C eher unübersichtlich macht. Auch der Versuch, die Initialisierung als eigene Funktion auszulagern, brachte keine zufriedenstellenden Ergebnisse. Problematisch sind für den Coder vor allem die vielen Variblen und die Zahlenformate, welche verwendet werden. Diese werden aufwändig neu deklariert, obwohl an vielen Stellen einfachere Methoden möglich wären, die den C-Code schlanker halten würden. Hinzu kommt, dass die Umsetzung der komplexen Rechnung wesentlich komplizierter geschieht als in vergleichbaren Übungsbeispielen. Auch die FFT-Routine von FFTW wird nicht korrekt umgesetzt, sodass die Ausgabe des Coders an dieser Stelle nicht ausgeführt werden kann, weil die Funktionszuweisung für die FFT-Berechnung nicht erfolgen kann. Die Umformung in C mit Hilfe des MATLAB® Coders ist daher keine geeignete Umformung des Prozesses in C, welche sich zu einer Analyse eignen würde. Weitere Tests brachten zwar Teile des Codes in eine verwertbare Form, der gesamte Programmtext konnte allerdings nicht durch den Coder umgeformt werden.

7.2 Manuelle Umformung in C

Auf Grund der nicht zufriedenstellenden Lösung durch den automatisierten Coder in MATLAB® wurde der Programmtext Schritt für Schritt in C umgeformt. Dabei wurde allerdings keine feste FFT-Routine eingebaut, ebenso wurden die Testsignale nicht wie in MATLAB® in den Programmtext integriert. Somit ist der Code möglichst universell einsetzbar und kann je nach Anwendung eingebunden werden. Eine Analyse ist ebenfalls möglich, sowohl durch MATLAB® als auch in der Eingabeaufforderung.

Zunächst werden die Standardbibliotheken für C hinzugefügt, als Platzhalter zusätzlich eine Bibliothek *fft*, an dessen Stelle dann eine beliebige FFT-Routine eingefügt werden kann. Dazu wurde für die Analysen die FFTW-Bibliothek genutzt, welche auch in MATLAB® Anwendung findet. Ob diese allerdings auf allen Zielplattformen lauffähig ist, kann bei einer universellen Herangehensweise nicht vorausgesetzt werden, deswegen wird an dieser Stelle die Möglichkeit gelassen, eine gewünschte FFT-Routine

einzubauen. FFTW wird auf Grund der Vergleichbarkeit der Analyseergebnisse mit denen des MATLAB®-Codes gewählt. Die FFT-Routine wird zur Ausgabe der Ergebniswerte in der gewünschten Form als eigene Funktion hinzugefügt, sodass der Ablauf des Programmtextes nicht geändert werden muss und die Funktion *fft(xk, 2*N)* weiterhin die Transformation liefert (Abb. 25).

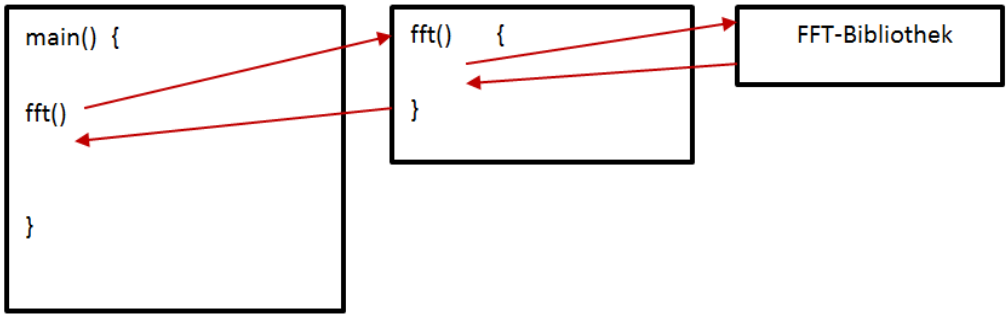

Abbildung 25: Externer Aufruf der FFT-Routine im C-Code

Anschließend werden die Variablen deklariert und, soweit möglich, auch initialisiert. Dabei erhalten alle Laufindexe und die Segmentlänge N eine Deklaration als Integer-Wert. Die Länge der Impulsantwort wird als vorzeichenloser Integer-Wert deklariert. Zum einen gibt es hier keine negativen Werte, zum anderen kann so ein größerer Zahlenraum genutzt werden. Für zwei Sekunden Impulsantwort bei 48 kHz müssen schließlich 96.000 Werte verarbeitet werden. Aus der Segmentlänge N und der Länge der Impulsantwort L_h wird anschließend die Anzahl der Segmente der Impulsantwort berechnet. Dieser Wert wird an vielen Stellen benötigt, sodass er als eigene Variable mit dem Namen *segments* als vorzeichenloser Integerwert definiert wird. Dabei muss mit einem Trick bewirkt werden, dass der nächste ganzzahlige Wert genutzt wird und nicht durch die Division von Integer-Werten Nachkommastellen wegfallen. Dazu werden 0,5 addiert, sodass auf jeden Fall aufgerundet werden muss.

```
unsigned int segments = (int)(Lh/N + 0.5);
```

Nach der Berechnung der Segmentzahl der Impulsantwort werden die Arrays deklariert, welche als Speicher des Prozesses dienen. Dabei kam es bei der Ausführung auf dem mobilen Testsystem zu Problemen mit der Speicherfreigabe. Tests auf weiteren mobilen Rechnern zeigten, dass dieses Problem von der Hardwareumgebung abhängt. Deswegen waren Tests auf dem mobilen Computersystem nicht möglich, sondern wurden lediglich auf dem Tower-PC mit i7-Prozessor durchgeführt. Die Arrays werden mit der gesamten Größe deklariert, allerdings nicht initialisiert. So kann bereits der Speicher mit den leeren Arrays belegt werden und entsteht keine Veränderung mehr in der Speichergröße.

Nach Deklaration und Initialisierung der Variablen folgt der Import und die Transformation der Segmente der Impulsantwort. An dieser Stelle wird eine Datei *h.wav*

eingelesen. Dabei wird die Anzahl der einzulesenden Werte anhand des Wertes L_h festgelegt. In einer Implementierung gäbe es in diesem Fall entweder die Möglichkeit, den Wert für L_h per Benutzereingabe festzustellen oder eine Datei mit einem Pointer zu durchlaufen und anschließend den Wert der letzten Adresse in der Datei zu übernehmen. In diesem Fall ist der Wert L_h gesetzt, sodass in Bezug auf die vorgestellten Möglichkeiten die erste genutzt wurde, allerdings ohne eine Benutzereingabe.

Das Transformieren der Impulsantwort läuft in zwei verschachtelten ab. Die äußere durchläuft die Segmente, die innere die Werte des aktuellen Segments. Diese verschachtelten Schleifen gab es im MATLAB®-Code nicht, weil dort durch Eingabe des gewünschten Bereichs direkt ein Vektor beschrieben wurde. In C muss im Programmiertext jeder Wert einzeln betrachtet vorkommen. Ein Segment $h_i[n]$ wird direkt nach dem Beschreiben mit Werten in den Frequenzbereich transformiert. An dieser Stelle ändert sich im Programmiertext nur die Zuweisung zur entsprechenden Zeile des Arrays *imp_freq*.

```
for (i=0; i < segments; i++)    {
    for (j = 0; j < N; j++)    {
        hi[j] = h[i*N+j];
    }
    imp_freq[i] = fft(hi, 2*N);
}
```

Ist die Transformation abgeschlossen, springt das C-Programm in die Hauptschleife und lädt das erste Segment $x_k[n]$. An dieser Stelle müsste der Audiostream abgefragt werden, welcher von der Peripherie des Prozessors angeliefert wird. Zu Testzwecken wurde an dieser Stelle das Testsignal eingefügt, welches segmentweise eingelesen wird. In der allgemeinen Fassung des Codes steht die Zuweisung zu einer Funktion *audiostream(fs, N)*, welche schon aus dem Pseudocode bekannt ist. Sie steht als Platzhalter für eine Routine, welche N Audiosamples bei der Abtastfrequenz f_a einliest und als Array übergibt. $x_k[n]$ wird nach der Übernahme transformiert und im Array *in_freq* abgespeichert.

Die Multiplikationen der Segmente $X_k[f]$ mit den Segmenten $H_i[f]$ erfordern erneut zwei verschachtelte Schleifen. Eine Funktion zur elementweisen Multiplikation existiert in C nicht, jede Multiplikation muss daher anhand der Schleife durchgeführt werden. Die äußere Schleife durchläuft dabei die Multiplikationssegmente, die innere führt die Multiplikationen innerhalb der Segmente durch. Bei den Zeigern für die Speicherpositionen ändert sich nichts, lediglich die Kontrollstrukturen wie if-/else-Zweige werden mit den geschweiften Klammern versehen und die end-Befehle dadurch ersetzt.

```
t = s;

for (i = 1; i<k+1; k++)    {
    for (j = 1; j < 2*N+1; j++)      {
```

```
            Yki[j] = in_freq[t][j]*imp_freq[i][j];
        }
        out_seg[l] = ifft(Yki, 2*N);
        l = l + 1;
        t = t - 1;
        if (t == 0) {
            t = ceil(Lh/N);
        }
        if(l == 2*ceil(Lh/N)+1)  {
            l = 1;
        }
}
```

Beim Overlap-Add Verfahren werden ebenfalls verschachtelte Schleifen notwendig, um den Ausgabevektor out zu beschreiben. In C ist dieser ebenfalls ein Array, welches N Werte enthält. Auch die Additionen müssen einzeln für jedes Element durchgeführt werden, ebenso muss das Array zu Beginn des Overlap-Add Verfahrens beim Beschreiben mit Nullen komplett durchlaufen werden.

7.3 Laufzeit-Analyse des C-Codes

Um den Code zu analysieren, werden die vom MATLAB®-Code bereits bekannten Signale eingesetzt. Dabei wird mit den C-Funktionen zum Einlesen von Dateien gearbeitet, welche eine Datei sampleweise einlesen können. Ein Zeiger durchläuft dabei die Datei und kann je nach Programmanfrage Samples oder Samplebereiche ausgeben. Die transformierten Segmente der Impulsantwort könnten auch in einer Datei gespeichert werden, welche je nach Bedarf wieder ausgelesen wird. Darauf könnte bei einer Implementierung zurückgekommen werden, um die Transformation einer bereits bekannten Impulsantwort zu umgehen. Die schnellere Variante ist allerdings die Nutzung der Arrays, weil die Daten direkt aus dem von C eingerichteten Speicher landen und nicht umgeformt werden müssen.

Die Laufzeitanalyse in C kann mit Hilfe der Funktion *clock()* erfolgen. Dazu muss zusätzlich die Bibliothek *time.h* geladen werden. Im C Programm wird dann über die Funktion *clock()* ein Zeitstempel zu Beginn der Hauptschleife erstellt, welcher in einer Variable gespeichert wird. Im Beispiel ist dies die Variable *pstart*. Diese hat den Typ *clock_t*, welcher in der eingebundenen Bibliothek definiert ist. Der gleiche Aufruf nur mit anderer Variable (*pende*) erfolgt noch einmal, wenn die Hauptschleife die gewählte Zahl an Durchläufen erledigt hat. Die Anzahl der Durchläufe wird so eingestellt, dass, genau wie im MATLAB®-Code auch, das Eingangssignal komplett verarbeitet ist. Die Zeitmessung ist dann anschließend vergleichbar mit der Umsetzung in MATLAB®. Durch eine Subtraktion wird die Differenz der Taktzyklen zwischen den beiden Zeitstempeln ermittelt. Diese liegt allerdings in Taktzyklen vor, eine Angabe, welche mit den bereits ermittelten Laufzeiten nicht vergleichbar wäre. Deswegen wird mit dem Befehl *CLOCKS_PER_SEC* die Anzahl der Taktzyklen pro Sekunde

N	Grundlatenz [Sek.]	Laufzeit i7 [Sek.]
1	0	$37,52$
32	$7,26 \cdot 10^{-4}$	$251,72$
64	$1,45 \cdot 10^{-3}$	$116,58$
128	$2,90 \cdot 10^{-3}$	$54,19$
256	$5,80 \cdot 10^{-3}$	$24,80$
512	$1,16 \cdot 10^{-2}$	$10,70$
1024	$2,32 \cdot 10^{-2}$	$4,39$
2048	$4,64 \cdot 10^{-2}$	$2,02$
4096	$9,29 \cdot 10^{-2}$	$0,97$
8192	$1,86 \cdot 10^{-1}$	$0,77$

Tabelle 11: Laufzeit des C-Codes auf i7-System

ermittelt und durch diesen Wert dividiert. Das Ergebnis enthält dann die Laufzeit in Sekunden als Fließkommawert.

```
#include <time.h>

int main(void)
{
    clock_t pstart, pende;
    /* INIT */
    pstart = clock();
    while (run > 1)    {
    }
    pende = clock();
    time = (float)(pende-pstart) / CLOCKS_PER_SEC
    printf("Laufzeit: %f Sekunden \n", time);
}
```

Die Laufzeitanalyse wurde nur auf dem i7-System durchgeführt. Zu Testzwecken wurden verschiedene mobile Rechner eingesetzt, allerdings ist der Code auf dem beim MATLAB®-Code eingesetzten mobilen Rechner nicht lauffähig und kann somit nicht getestet werden. Die Testdurchläufe wurden im gleichen Rahmen wie bei zuvor beim MATLAB®-Code durchgeführt. Es wurden Segmentlängen von $N = 32$ bis $N = 8196$ durchgerechnet, die Ergebnisse mehrerer Durchläufe führen zu den angegebenen Werten in Tab. .

Dabei ist zu erkennen, dass der C-Code bei kürzeren Segmentlängen schneller läuft als der zuvor gemessene MATLAB®-Code. Eine Echtzeitfähigkeit bei geringen Segmentlängen leistet auch dieser Code nicht. Die beste zu erreichende Grundlatenz für den Prozess für ein L_h von 2,4 Sekunden liegt bei 11,6 ms, was einer Segmentlänge von $N = 512$ entspricht. Eine bereits von VST-Plugins erreichte Segmentlänge von

$N = 64$ erreicht der Prozess in dieser einfachen Umsetzung nicht. Auffällig ist jedoch, dass C innerhalb der Eingabeaufforderung und in Konsolen allgemein schneller rechnet als MATLAB®. Tests auf anderen Systemen zeigen gleiche Abweichungen, wenn der Prozess nur in der Konsole berechnet wird. So sind in den Tests Verbesserungen bei der Rechenzeit von bis zu einem Faktor 0,75 ermittelt worden. Um noch bessere Werte zu erreichen, müsste der Code noch mit weiteren komplexen C-Routinen erweitert werden, welche aber nicht Teil dieser Thesis sind, sondern in den meisten Fällen auf die Zielsysteme angepasst werden müssen.

7.4 Ergebnisanalyse des C-Codes

Die numerische Auswertung des C-Codes wird mit Hilfe von MATLAB® durchgeführt. Dazu werden die Werte im Ausgabearray des C-Codes in eine Datei geschrieben, welche in MATLAB® importiert werden kann und dort mit dem Faltungsergebnis $y\_conv\,[n] = x\,[n] * h\,[n]$ verglichen wird. Diese Auswertung fand gesondert zu der vorigen Laufzeitmessung statt, weil ein Beschreiben einer weiteren Datei in der späteren Umsetzung nicht vorgesehen ist und somit die Laufzeit erhöht hätte. Das Faltungsergebnis $y\_conv\,[n]$ wird in MATLAB® über den *conv()*-Befehl ermittelt und wie zuvor auf 100.000 Werte begrenzt in einem eigenen Vektor gespeichert. Die Ergebnisse der Berechnung aus dem C-Code werden importiert und liegen anschließend auch als Vektor in MATLAB® vor. Auch jener wird auf 100.000 Werte begrenzt. Anschließend wird der Differenzvektor *diff* berechnet und in einer Grafik ausgegeben (Abb. 26).

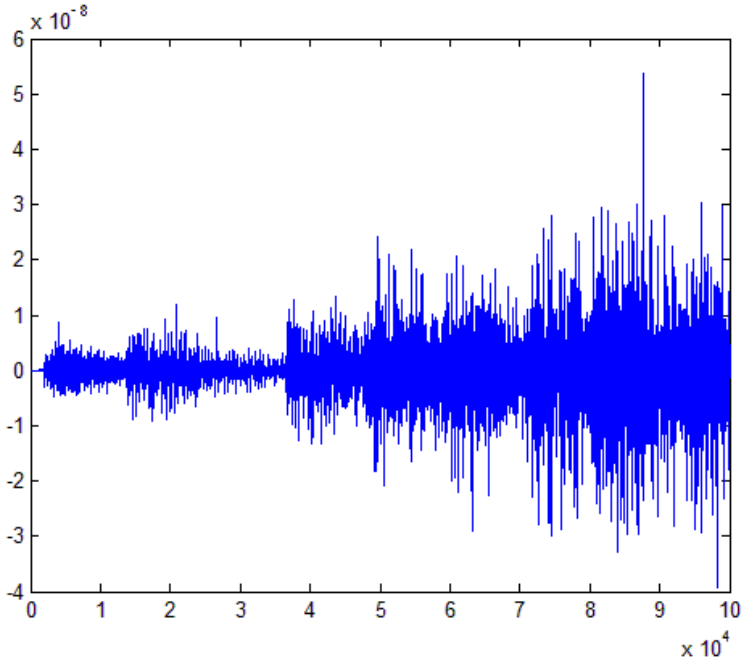

Abbildung 26: Differenz zwischen *y_conv* und *out* über die ersten 100.000 Werte beim C-Code

Auch hier sind Unterschiede im Vergleich zu den bereits ermittelten Werten fest-

zustellen. So rechnet der C-Code noch wesentlich genauer als der MATLAB®-Code. Bei jenem waren Abweichungen von 10^{-6} festzustellen. Diese sind zwar nicht hörbar, aber dennoch auffällig gewesen. Beim C-Code sind Abweichungen im Bereich von 10^{-8} festzustellen, was eine deutliche Verbesserung bedeutet.

Auch mit dem Ergebnis des C-Codes wurde eine audiotechnische Analyse durchgeführt, um festzustellen, wie die Korrelation zwischen dem Faltungsergebnis der diskreten Faltung und dem Ergebnis der segmentierten Faltung ist. Dazu wurde wie auch zuvor beim MATLAB®-Code in Adobe Audition eine Stereospur angelegt, in welcher die Signale in den Kanälen platziert wurden. Der Korrelationsmesser wurde auf einen Bereich von 120 dB eingestellt, welche analysiert wurden. Das Ergebnis ist gleich jenem des MATLAB®-Codes. Das Messgerät zeigt nach einem kurzen Einpendeln direkt den Wert Eins, welcher für Übereinstimmung steht. Aus audiotechnischer Sicht ist das Ergebnis des C-Codes der segmentierten Faltung also gleich dem Faltungsergebnis $y\_conv\,[n]$.

8 Ansätze zur Implementierung auf Zielplattformen

Eine Umsetzung der vorhandenen Programmiertexte auf Prozessoren erfordert die Untersuchung der Anschlussmöglichkeiten für die benötigten Funktionen. So muss die Impulsantwort im Speicher abgelegt sein, dieser muss anschließend an den Prozess angebunden werden. Ebenso muss das Eingangssignal an den Prozess übergeben werden. Diese Anschlüsse sind direkt von der verwendeten Hardware abhängig, welche einen Rahmen für die Implementierung vorgibt. In diesem Abschnitt soll auf mögliche Entscheidungsschritte bei einer Implementierung auf mobilen Plattformen eingegangen werden. Dabei stehen zunächst das gewählte Zahlenformat und die Berechnung der FFT im Vordergrund. Anschließend werden Prozessortechnologien verglichen und in Bezug auf eine Implementierung der segmentierten Faltung untersucht, ebenso wird ein Blick auf die Thematik der Echtzeit-Verarbeitung in Bezug auf Anwendungen der Audiotechnik geworfen. Schließlich folgen Anpassungen und mögliche Optimierungen des im Rahmen dieser Thesis vorgestellten Programmiertextes.

8.1 Fixed vs. Floating Point

Die Frage nach dem Zahlenformat stellt sich bei der Prozessorwahl immer an zwei wesentlichen Stellen: Kosten und Programmieraufwand. Die Genauigkeit im Bereich Audio ist bei beiden Technologien gegeben, sodass es zwar numerisch einen Unterschied macht, welche von beiden verwendet wird, allerdings in Bezug auf die Audioausgabe kaum einen Einfluss gibt. Grundsätzlich besitzen Floating Point Prozessoren ein komplexeres Rechenwerk, welches in der Regel 32 oder 64 Leitungen für entsprechende Wortbreiten besitzt. Dadurch wird der Prozessor teurer als der meist mit 16 Bit Wortbreite betriebene Fixed Point Prozessor. Neben der Frage der numerischen Präzision benötigt der Fixed Point Prozessor für einfachere Rechenoperationen oft komplexe Programmiertexte. Das Floating Point Format hingegen kommt mit wesentlich einfacheren Programmierstrukturen aus und ist dadurch einfacher in der Entwicklung von Programmtexten. Die nun anschließende Gegenüberstellung der Zahlenformate soll eine Entscheidungshilfe sein. Die Informationen über die Formate sind aus [Smi, 2011] und [Fra, 2004] entnommen.

8.1.1 Fixed Point

In der Fixed Point Darstellung ist jeder Zahl ein Binärwert zugeordnet. Bei 16 Bit Wortbreite ergeben sich entsprechend $2^{16} = 65.535$ Werte möglich. Dabei muss allerdings nicht zwangsläufig der Wertebereich von Null bis 65.535 genutzt werden. Es gibt verschiedene Möglichkeiten, die Zahlen auf eine Werteskala zu übertragen und diese zu nutzen. So kann zum Beispiel auch der Bereich von Null bis Eins mit Hilfe der verfügbaren 216 Werte dargestellt werden. Auch negative Zahlen sind möglich, ein Wertebereich könnte dann von -1 bis 1 laufen und mit den Werten beschrieben werden.

Die Verarbeitung der binären Werte bietet ebenfalls verschiedene Möglichkeiten an, die Zahlen zu interpretieren. Neben dem bekanntesten Schema unsigned integer

(0 – 65535 in 16 Bit) gibt es noch drei weitere, wie die Zahlen auf die Binärwerte gesetzt werden:

- $0 - 65.535$, $0 = 0000$ (unsigned integer)
- $-32.767 - 32.768$, $0 = 0111$ (offset binary)
- $-32.767 - 32.767$, $0 = 0000 = 1000$ (sign and magnitude)
- $-32.768 - 32.767$, $0 = 0000$ (two's complement)

Die Möglichkeiten, welche auf Prozessoren in der Regel genutzt werden, sind unsigned integer und two's complement. Die Nutzung des Zweierkomplements hat mit der Implementierung auf Hardware zu tun. Dabei ist die Umsetzung eines Flags für negative Zahlen als most significant bit (MSB) einfach umzusetzen und zu interpretieren. Fixed Point Prozessoren sind im Gegensatz zu Floating Point Prozessoren günstig in der Anschaffung, das liegt an der ebenfalls einfacheren internen Struktur. In Bezug auf digitale Signalprozessoren (DSPs) sind Fixed Point Programmtexte wesentlich komplexer als jene im Floating Point Bereich. Wo für eine Multiplikation auf einem DSP bei Fixed Point zum Teil mehr als 20 Zeilen Code in Assembler nötig ist, kann diese auf einem Floating Point mit nur einer Zeile Code durchgeführt werden. Somit kann Fixed Point Technologie als günstig, aber aufwändig in der Programmierung beschrieben werden.

8.1.2 Floating Point

Die Zahlendarstellung im Floating Point ist komplexer als im Fixed Point und benötigt auch mehr Bits zur Darstellung der Zahlen. Die typische Wortbreite für Floating Point Prozessoren beträgt 32 Bit. Mittlerweile existieren auch 64 Bit Floating Point Prozessoren, welche typischerweise als GPP (general purpose processor) in Heimcomputern verbaut sind. Eine Zahl im Floating Point Format besteht aus einem Umkehrbit, welches vergleichbar mit dem Negativflag beim Fixed Point ist. Das MSB eines wird als Exponent zum Wert -1 interpretiert. Ist das Bit gesetzt, liegt ein negativer Wert vor. Die nächsten Bits enthalten einen Wert für einen Exponent, welcher anschließend in eine Formel eingesetzt wird. Anschließend folgt die Mantisse, welche eine binäre Bruchzerlegung enthält, welche mit einer führenden Eins versehen ist. Diese ist bei einer 32 Bit Darstellung 23 Bit lang. Die Berechnung der korrekten Zahl erfolgt mit Hilfe der Formel in (Gl. 53).

$$a_{\text{float}} = (-1)^S \cdot M \cdot 2^{(E-127)} \tag{53}$$

Der Wertebereich eines 32 Bit Floating Point Rechenwerks kann mit Hilfe der Formel ermittelt werden. Dazu müssen zunächst die Grenzen der Mantisse ermittelt werden. Sind alle Bits der Mantisse auf den Wert Null gesetzt, enthält die Mantisse den Wert Eins. Werden alle Bits der Mantisse auf eins gesetzt, so hat die Mantisse bei 32 Bit Floating Point Darstellung den Wert $2 - 2^{23}$. Der Exponent kann Werte von Null bis 255 einnehmen. Die Grenzen einer 32 Bit Zahlendarstellung im Floating Point können nun bestimmt werden (Gl. 54).

$$
\begin{aligned}
a_{\min} &= (-1)^S \cdot 1 \cdot 2^{(0-127)} = 5,9 \cdot 10^{-39} \\
a_{\max} &= (-1)^S \cdot \left(2 - 2^{23}\right) \cdot 2^{(255-127)} = 6,8 \cdot 10^{38}
\end{aligned} \tag{54}
$$

Um weitere wichtige Werte darstellen zu können, wurden entsprechende Sonderregeln entwickelt. Der Wert $a = 0$ liegt vor, wenn sowohl die Bits der Mantisse als auch jene des Exponenten Null sind. Ein Wert von a = unendlich wird durch das Setzen von allen Bits der Mantisse auf den Wert Null und allen Bits des Exponenten auf Eins erreicht.

Die Strukturen eines Prozessors mit Floating Point Architektur sind komplexer als jene bei Fixed Point Prozessoren. Die Herstellungskosten sind dadurch höher, daher ist die Entscheidung für Floating Point auch immer eine finanzielle. Die Algorithmen für die Floating Point Prozessoren sind allerdings einfacher zu programmieren, wie bereits im Fixed Point Abschnitt erwähnt. Floating Point kann daher als teurer mit niedrigerem Programmieraufwand bezeichnet werden.

8.2 FFT-Routinen

Als FFT-Routine (teils auch FFT-Subroutine) wird eine eingebundene Funktionsdatei bezeichnet, welche die Berechnungen der FFT durchführt. Diese wird in einen C-Code per #include eingebunden. Die Funktionen der Routine sind dann direkt über den C-Code nutzbar, sodass dieser übersichtlich bleibt und der Programmierer sich den Aufwand für die FFT sparen kann.

Hinzu kommt, dass mittlerweile eine Vielzahl von guten und auch präzisen FFT-Routinen geschrieben worden sind, welche in ihrer komplexen Struktur wesentliche Einsparungen in der Berechnungszeit im Gegensatz zu einer eigenen Implementierung mitbringen. Ein Großteil dieser sind im Internet frei verfügbar, sodass diese für Projekte genutzt werden können, solange sich diese im Forschungsstatus befinden oder die Routine zu Testzwecken eingesetzt wird. Die Nutzung einer bereits vorhandenen Routine bietet sich daher in einem Projekt wie diesem an. Dabei muss für eine möglichst universelle Verarbeitung gewährleistet sein, dass die Routine auf möglichst vielen Zielsystemen flüssig läuft. Dabei ist insbesondere die Art des verwendeten Prozessors entscheidend. Viele FFT-Routinen, welche im Internet angeboten werden, sind für GPPs entwickelt worden und funktionieren auf DSPs nur in eingeschränkten Bereichen. Dies ist mit der Struktur der DSPs zu begründen, deren Kerne anders aufgebaut sind und in vielen Fällen schon technische Möglichkeiten zu einer einfacheren Berechnung der FFT besitzen. Die Vorstellung von zwei FFT-Routinen (FFTW und kissFFT) für C-Programmiertexte ist daher auf die Programmnutzung auf einem GPP fokussiert. Anschließend wird eine Routine für einen DSP vorgestellt, welche vom Hersteller vorgegeben ist und direkt implementiert werden kann.

8.2.1 FFTW

Die bekannteste und laut verschiedenen Statistiken am meisten genutzte Routine ist FFTW (Fastest Fourier Transformation in the West). Diese ist auch Teil des

MATLAB® Programmpakets und wurde für die Ermittlung der Laufzeiten der Codes aus den vorigen Abschnitten dieser Thesis genutzt. FFTW wurde 1997 in der ersten Version veröffentlicht. Programmiert und veröffentlicht wird FFTW von Matteo Frigo and Steven G. Johnson am Massachusetts Institute of Technology. Die Routine unterstützt sowohl eindimensionale als auch mehrdimensionale FFTs und lässt eine Vielzahl von Segmentlängen zu. Optimal ist auch an dieser Stelle die Nutzung von Zweier- oder Viererpotenzen zur Nutzung der vorgestellten Radix-2 bzw. Radix-4 Algorithmen.

FFTW ist in C programmiert, wird allerdings auf Linux entwickelt, was eine Einbindung in Windows-Systeme insbesondere bei multidimensionalen FFTs kompliziert machen kann. Der Vorteil von FFTW gegenüber anderen Routinen ist die hohe Rechengeschwindigkeit. Benchmarks auf verschiedenen Plattformen zeigen eine deutliche Überlegenheit, besonders bei kürzeren Segmentlängen, was für die Umsetzung der segmentierten Faltung von Vorteil ist. Ein Vergleich im Rahmen dieser Thesis ergab für $N = 32$ einen Faktor Fünf bei der Rechenzeit zwischen FFTW und der anschließend vorgestellten Routine kissFFT. Ein Blick in den Programmiertext zeigt, dass FFTW intern die Eingangswerte analysiert und entsprechend die schnellstmögliche FFT-Routine durchführt. Dabei kann es auch zu einer Segmentierung kommen, um Zeit zu sparen. Für eine Nutzung der FFT auf GPPs ist FFTW daher die FFT-Routine der Wahl, wenn das Zielsystem diese verarbeiten kann. Eine Recherche ergab, dass dies für fast alle heutigen GPPs der Fall ist.

8.2.2 kissFFT

Diese Routine ist eine beliebte Alternativlösung zu FFTW, welche insbesondere bei kleineren Projekten eingesetzt wird. Die Routine wird seit 2004 von Mark Borgerding auf der Open-Source Plattform Sourceforge entwickelt. Der entscheidende Unterschied zu FFTW ist die schlanke Umsetzung der FFT im C-Code. Die Einbindung ist simpler und kissFFT benötigt nur ein Zehntel der Programmzeilen von FFTW. Der Funktionsumfang von kissFFT ermöglicht sowohl fixed- als auch floating point Verarbeitung, ebenso ist die Berechnung von multi-dimensionalen FFTs möglich.

Das Prinzip von kissFFT legt weniger Wert auf die Geschwindigkeit als FFTW, was sich in den Benchmarks und auch eigenen Untersuchungen deutlich zeigt. Die Effizienz im Arbeitsaufwand und eine möglichst simple Einbindung stehen stattdessen im Hintergrund. So ist das Einbinden von kissFFT auf einem Windows-System wesentlich einfacher als es bei FFTW der Fall war. Auch die Dateigrößen der Programme, welche mit kissFFT erstellt werden, ist um ein Vielfaches kleiner als jene, die FFTW enthalten.

8.2.3 TI HWAFFT

In DSPs werden, wie bereits beschrieben, in der Regel vom Hersteller vorgefertigte FFT-Abläufe genutzt. Mittlerweile wird die FFT sogar hardwareseitig implementiert, um Rechenzeit zu sparen und Softwareabläufe einfacher zu gestalten. Beispielhaft wurde im Rahmen dieser Thesis die FFT-Implementierung der Chipsätze CMS320C55X von Texas Instruments betrachtet ([TI]). Diese verbindet einen Hardwarebaustein im Kern des Prozessors mit einer Softwareroutine. Dabei wird der Radix-2 Algorithmus als

Co-Prozessor umgesetzt. Dazu wurde ein Schmetterlingsgraph für Radix-2 als Hardware umgesetzt, welcher in einem Taktzyklus einzeln oder doppelt durchlaufen werden kann. Die Verdopplung der Durchläufe sorgt für schnellere Berechnung von FFTs mit längeren Blocklängen, kann allerdings bei FFTs mit ungerader Stufenzahl im Radix-2 Algorithmus nicht dauerhaft genutzt werden. Deswegen existiert auch der einfache Durchlauf, welcher in diesem Fall (z.B. für $N = 8$ mit 3 Stufen im Algorithmus) genutzt wird. Durch eine Pipeline, in welcher die Multiplikation mit den Gewichtungsfaktoren und die Addition oder Subtraktion getrennt bearbeitet werden, kann eine Verarbeitung noch weiter beschleunigt werden. Zusätzlich werden wesentliche Funktionen für C-Routinen im ROM-Speicher abgelegt und können über entsprechende Anweisungen im Programmiertext aufgerufen werden. Um weitere Berechnungszeit zu sparen, sind die Gewichtungsfaktoren W_N^0 bis W_N^{511} ebenfalls im Speicher abgelegt und brauchen nicht berechnet zu werden. Für die Berechnung der IFFT werden entsprechend des vorgestellten Schemas zur Berechnung der IFFT mit Hilfe der FFT die konjugiert komplexen Gewichtungsfaktoren verwendet. Ebenso sind Skalierungsfunktionen für eine Berechnung von Fixed Point Werten eingebaut, welche direkt über die Software gesteuert werden kann. Der Aufruf wird durch diese Implementierung auch softwareseitig einfacher. Mit einer Zeile lässt sich, ähnlich wie beim Aufruf in MATLAB®, die FFT-Berechnung programmieren.

8.3 Prozessortechnologien

Die Auswahl des Prozessors zur Implementierung eines Prozesses ist in erster Linie eine Entscheidung für eine Prozessortechnologie. Diese wird besonders bei kommerziellen Projekten in Bezug auf die finanziellen Möglichkeiten festgelegt. Um Prozessoren zu gruppieren, gibt es vier wesentliche Technologien von Prozessoren:

- GPP (general purpose processor), Anwendung: Heimcomputer-CPU
- DSP (digital signal processor), Anwendung: Audiosysteme
- GPU (grafic processing unit), Anwendung: Grafikkarten
- FPGA (field programmable gate array), Anwendung: komplexe Digitalsysteme

Im Folgenden werden die Prozessortechnologien in Bezug auf eine mögliche Umsetzung der segmentierten Faltung verglichen. Dabei stehen sowohl Arbeits- und Speichernutzung, deren mögliche Optimierung auf den Prozessoren sowie die Verarbeitung der FFT im Mittelpunkt.

8.3.1 GPU

Eine Realisierung der segmentierten Faltung auf GPUs wird nicht in Betracht gezogen. Zwar sind die 2D- und teils 3D-Verarbeitung von Grafikkarten gut geeignet, um FFTs entsprechend zu berechnen. Da bei der Audiosignalverarbeitung für die segmentierte Faltung aber nur eindimensionale FFTs nötig werden, würde eine Berechnung auf einer GPU nur dann Sinn machen, wenn die Segmentlänge gesplittet wird und somit

eine 2D-Verarbeitung möglich ist. Ist allerdings nach der Wahl des Prozessors gefragt, sollte die GPU nicht als primäre Lösung genannt werden, weil wesentliche Funktionen der Grafikverarbeitung als Vorteil der GPU im Allgemeinen im Prozess nicht verwendet werden können. Eine GPU ist daher nur als Ausweichlösung zu betrachten und ist eher für lange FFTs mit entsprechender 2D-Verarbeitung von Vorteil.

8.3.2 GPP

Die einfachen Umsetzungen der segmentierten Faltung im Rahmen dieser Thesis haben gezeigt, dass die GPPs an vielen Stellen durch ihre Individualität den Prozess eher bremsen. Allein die gewählte FFT-Routine macht einen Großteil der Rechenzeit aus und ist somit ein entscheidender Teil des Programmiertextes. Die interne Struktur des GPP ist zudem nicht auf Rechenprozesse wie jene der digitalen Audiosignalverarbeitung ausgelegt, sondern auf die typischen Anwendungen der Heimcomputer. Hauptaugenmerk bei GPPs ist die Rechengeschwindigkeit, welche möglichst hoch sein soll, um auch komplexe Anwendungen wie Games umsetzen zu können. Auf Grund von aktuellen technischen Grenzen wird mittlerweile mit Mehrkernprozessoren gearbeitet, um die Leistung von Computersystemen weiter zu erhöhen.

Wird die segmentierte Faltung auf einem GPP berechnet, werden komplexe Programmiertexte nötig, um Abläufe für den Prozessor zu optimieren und somit geringere Laufzeiten zu erreichen. In Umsetzungen der segmentierten Faltung, zum Beispiel in einem VST-Plug-In, wird anhand der Struktur des Programmiertextes die Komplexität sichtbar. Mehrere Tausend Zeilen Code benötigen die Plug-Ins, um eine Latenz von 5 ms zu erreichen, welche in einer Recherche Anfang Juli 2013 als der beste bisher erreichte Wert ermittelt wurde. Die im Rahmen dieser Thesis vorgestellte einfache Umsetzung in den Programmiertext würde für eine Umsetzung auf GPP mit entsprechender Latenz wesentlich weiter ausgedehnt werden müssen. Funktionsbibliotheken zur Verarbeitung von komplexen Zahlen und Optimierungen für die Schleifenmechanismen sind zwei wesentliche Punkte, welche in Betracht gezogen werden sollten, wenn eine Umsetzung auf einem GPP geplant ist. Ebenso ist der Compiler eine nicht zu unterschätzende Instanz, welche ebenfalls bei der Programmierung für GPP besonders getestet werden sollte. Oft finden Optimierungen statt, welche für die Umsetzung sinnvoll sind. Es kann aber an dieser Stelle auch zu ungewünschten Umsetzungen kommen, welche erst durch Debuggen mit verschiedenen Compilern sichtbar werden. Mittlerweile sind aktuelle Compiler in der Lage, an vielen Stellen die Optimierungen automatisch korrekt durchzuführen. Dennoch empfiehlt es sich, die Ergebnisse verschiedener Compiler gegen einander zu halten und zu testen.

Ein GPP ist daher zwar geeignet, aber nicht zu empfehlen, weil die Strukturen des Prozessors für einen Mehraufwand in der Programmierung sorgen, welcher dazu dient, den Prozess an den Prozessor anzupassen.

8.3.3 DSP

Digitale Signalprozessoren sind für viele Audioanwendungen verwendet worden und stellen in der Audiotechnik die am meisten genutzten Prozessoren dar. DSPs werden mit externem Speicher versorgt, welcher mit Hilfe von seriellen Schnittstellen

abgerufen wird. Auf Grund der Auslegung ist eine Speichernutzung in dieser Prozessortechnologie einfach umzusetzen. Der Datenfluss in DSPs ist ebenfalls größtenteils seriell, sodass eine schnelle Umsetzung einfacher Rechenoperationen möglich wird. Wo zu Beginn der DSP-Entwicklung nur eine Operation pro Taktzyklus umgesetzt wurde, laufen in heutigen DSPs mehrere Operationen in einem Zyklus. Die Recheneinheiten, mittlerweile zwei oder mehr, werden für verschiedene Prozesse gleichermaßen genutzt und erhöhen so die Ablaufgeschwindigkeit. Generell sind Algorithmen für DSPs am Hauptrechenwerk ausgelegt und werden an jener Stelle für maximale Geschwindigkeit entwickelt. Programmiert werden die Prozessoren genau wie GPU und CPU in C oder C++, allerdings meist in herstellerspezifischen Befehlssätzen, welche oft eine Vielzahl von Funktionen vorgeben. An dieser Stelle wird auch die Umsetzung der FFT relevant, welche für DSPs effizient wird, weil Hersteller zum Teil Co-Prozessoren in den Kern des Prozessors einfügen, welche die FFT-Berechnung übernehmen. Somit entsteht ein Spezialrechenwerk, welches auf die FFT optimiert ist und über einfache Befehlsstrukturen genutzt werden kann. Komplexe Programmiertexte wie beim GPP entstehen bei der FFT auf einem DSP daher in der Regel nicht. Die bereits angesprochenen Fließkommazahlen (Floating Point) sind bei DSPs einfach umzusetzen, ebenso gibt es viele Bibliotheken und vorgefertigte Codes, welche individuell nutzbar sind. Die Entwicklungszeit für DSPs kann auf Grund dieser Möglichkeiten als eher gering im Vergleich zu den nachfolgend vorgestellten FPGAs beschrieben werden. Durch die eher geringen Kosten für DSPs sind diese auch in Bezug auf Vermarktung von Endgeräten als günstig einzustufen. Ein DSP ist allerdings in Bezug auf Flexibilität eher eingeschränkt. Komplexe Strukturen stoßen auf Grund der Auslegung teils an Grenzen. DSPs sind daher für die segmentierte Faltung als Einzelprozess sehr gut geeignet, weil die Standardfunktionen wie die FFT durch optimierte Prozessstrukturen schneller umgesetzt werden können. Ebenso gehört eine Speicherung von Zwischenergebnissen mit DSPs zu den Hauptabläufen, was sich ebenfalls positiv auf eine Implementierung auswirkt.

8.3.4 FPGA

Ein FPGA besteht aus einer Anordnung von frei programmierbaren logischen Einheiten, dem so genannten sea of gates. Die Einheiten enthalten je nach Modell Rechenbausteine wie zum Beispiel Addierer oder Multiplizierer. Auch Speicher wird in den logischen Einheiten umgesetzt. Durch die Möglichkeit, die logischen Einheiten durch Programme in diversen Kombinationen zu verknüpfen, können parallele Abläufe einfach realisiert werden. So kann eine FFT wesentlich schneller berechnet werden, weil Multiplikationen und Additionen als eine Reihe von logischen Einheiten umgesetzt werden können, welche durchlaufen werden muss. Jenes kann parallel mehrfach umgesetzt werden, sodass eine schnellere Gesamtumsetzung erreicht werden kann. Eine Vielzahl von typischen Funktionen der zuvor vorgestellten DSPs lässt sich auch für FPGAs umsetzen, oft sogar noch wesentlich schneller. Das FPGA enthält im Vergleich zum DSP mit 1-2 Rechenwerken einen Wert von 50 bis zu 100 Rechenwerken, welche parallel geschaltet werden können. So werden je nach Modell bis zu zehn Mal so viele Multiplikationen von einem FPGA berechnet wie in der gleichen Zeit von einem DSP. Die Implementierung auf FPGAs ist allerdings wesentlich komplexer als jene auf DSPs.

Programmiert wird in VHDL, was zwar ähnlich zu C ist, aber nicht C entspricht. Zusätzliche Bibliotheken, welche benötigt werden, liegen für typische Anwendungsfälle zwar vor, speziellere sind jedoch meistens nicht verfügbar. Das liegt an der Individualität der Umsetzungen auf FPGAs. Durch die Möglichkeit, den zu implementierenden Prozess möglichst auf die individuellen Anforderungen anzupassen, sind Routinen oft nicht universell einsetzbar. Ebenso ist deren Entwicklung wesentlich komplexer als bei den anderen Technologien. Hinzu kommt, dass FPGAs von Haus aus nur wenig Datenspeicher mitbringen, welcher oft nicht ausreicht. Zwar kann dieser durch die logischen Einheiten erweitert werden, jedoch ist auch jener Speicher begrenzt, weil oft eher auf Rechenleistung als auf Speicher gesetzt wird. Externe Speicher sind für FPGAs zwar möglich, allerdings nicht so schnell ansprechbar wie bei DSPs. Sieht man sich die Komplexität des Prozesses der segmentierten Faltung in Bezug auf FPGAs an, so wird deutlich, dass die Umsetzung besonders an Stellen mit Bedingungsabfragen wie einer if-/else-Struktur kompliziert wird. Diese und ähnliche Abfragen sind auf FPGAs nur durch komplexe Programmierung möglich. Deswegen sind FPGAs nur dann für eine Umsetzung von Vorteil, wenn die Teilstrukturen des Prozesses auf dem FPGA abgebildet werden können und die entsprechende Entwicklungszeit vorhanden ist. Außerdem ist in Bezug auf eine kommerzielle Nutzung des mobilen Endgeräts zu bedenken, dass FPGAs teurer als DSPs sind.

8.3.5 Ergebnisse: DSP vs. FPGA

Eine Umsetzung der segmentierten Faltung auf mobilen Plattformen wird am besten auf einem DSP oder einem FPGA funktionieren. Dabei steht bei einer Entscheidung für eine der beiden Technologien sowohl Kosten, Komplexität der Programmierstruktur und Rechengeschwindigkeit im Mittelpunkt. DSPs überzeugen durch geringere Kosten und können mit Hilfe von fertigen optimierten Codes der Hersteller einfach programmiert werden. Dabei werden Programmiertexte auf die vorhandene Infrastruktur ausgelegt und angepasst. Die Rechengeschwindigkeit ist hoch, eine parallele Verarbeitung aber nicht möglich. An dieser Stelle sind FPGAs von Vorteil, welche die Geschwindigkeit bei den Berechnungen durch ein Anpassen der Abläufe auf dem Prozessor an die Prozessstruktur erhöhen. Die Parallelität, welche erreicht werden kann, ist ein weiterer Vorteil der FPGAs. Durch die frei programmierbare Struktur wird die Entwicklung von Codes schwieriger und aufwändiger als bei DSPs, die ebenso höheren Kosten sind ein weiterer Nachteil der FPGAs.

Eine Empfehlung für die Umsetzung der segmentierten Faltung für eine anschließende kommerzielle Nutzung sind daher DSPs. Um die schnellste Variante des Prozesses zu erreichen, sind FPGAs durch die Strukturen überlegen.

8.4 Echtzeit Audio

Im Rahmen der Bearbeitung des Themas der segmentierten Faltung kam an vielen Stellen auch in Gesprächen die Frage auf, was denn Echtzeit überhaupt bedeutet und wann man von Echtzeit im Bereich der Audioverarbeitung spricht. Dieser Absatz befasst sich daher mit dem Begriff der Echtzeit im Bereich der computerbasierten Prozesse.

Oft wird Echtzeit mit der Uhrzeit in Verbindung gebracht, etwa bei der in BIOS von Mainboards verwendeten RTC (real-time clock). Dieser Baustein sorgt für das Halten der Uhrzeit im BIOS, wenn der Computer ausgeschaltet ist. Betrieben wird dieses Echtzeitsystem mit einer 3V Knopfzelle, welche auf dem Mainboard zu finden ist. Die BIOS-RTC ist eines der einfachsten Echtzeitsysteme. In diesem Fall muss lediglich eine Uhr gespeichert und aktualisiert werden. Durch die Umsetzung als Unix-Zeitstempel wird ein Zahlenwert gespeichert und jede Sekunde um eins erhöht. Die Verknüpfung der Uhrzeit bzw. einer Sekunde mit dem Begriff Echtzeit ist allerdings im Allgemeinen nicht korrekt. Echtzeit kann aber anhand einer Sekunde erklärt werden, um eine allgemeine Form der Definition von Echtzeit zu erhalten. Dabei hilft ein genauer Blick auf den Ablauf der RTC des BIOS. Diese muss innerhalb einer Sekunde um eins hochgezählt werden, außerdem muss der aktuelle Wert gespeichert werden. Beide Schritte müssen genau in der Zeitspanne einer Sekunde geschafft sein, weil sonst der nächste Schritt nicht rechtzeitig ausgeführt werden kann. Die Sekunde ist somit eine zeitliche Begrenzung, in welcher die Operationen ausgeführt werden müssen. Diese zeitliche Begrenzung ist der Kern einer allgemeinen Definition der Echtzeit. Operationen werden in einem festgelegten zeitlichen Rahmen ausgeführt. Die Antwort des Systems erfolgt ebenfalls in diesem Rahmen, sodass bei Beginn der nächsten Zeitspanne neu begonnen werden kann. Eine weitestgehend allgemeine Definition der Echtzeit in der Informatik lautet: „Echtzeit bedeutet dass das Ergebnis einer Berechnung innerhalb eines gewissen Zeitraumes garantiert vorliegt d.h. bevor eine bestimmte Zeitschranke erreicht ist.“ [Ko, S. 91]

In Bezug auf die Umsetzung der segmentierten Faltung in Echtzeit müssen zur Berechnung in Echtzeit also Grenzen gesetzt werden, in welchen der Prozess durchgehend abläuft. Diese lassen sich aus den Ergebnissen der Analyse ermitteln. So beginnt ein Zeitabschnitt immer mit dem Einlesen eines neuen Segments $x_k[n]$ und endet mit der Ausgabe des Ergebnisvektors *y_out*. Laut der angegebenen Definition könnte die Zeitschranke auch die bei $N = 1024$ vorliegenden ca. 30 ms betragen, somit wäre der umgesetzte Prozess echtzeitfähig. Ein Blick in die Eckdaten der Audiotechnik zeigt aber, dass eine Verzögerung von 30 ms und mehr bereits als Echo wahrgenommen wird. Die Zeitschranke, mit der die Echtzeitfähigkeit des Systems gemessen wird, ist bei Prozessen der Audiotechnik auch von dieser abhängig. Deswegen muss bei einem solchen Prozess zuvor feststehen, welche Latenz im Bereich der Audiotechnik noch für eine entsprechende Wahrnehmung akzeptiert wird. Im Fall des in dieser Thesis vorgestellten Prozesses wurde von einer Latenz von 5 ms ausgegangen, bei welcher der Prozess audiotechnisch so umgesetzt ist, dass die Latenz in der Ausgabe fast nicht wahrnehmbar ist. Diese gewünschte Latenz ist auch die Vorgabe für die Umsetzung in Echtzeit. Diese wäre erreicht, wenn das System mit $N = 64$ schnell genug die Ergebnisse liefern würde.

Soll ein audiotechnischer Prozess also in Echtzeit umgesetzt werden, muss zunächst die Zeitschranke festgelegt werden, welche der gewünschten Latenz entspricht. Der Prozess kann anschließend anhand der vorgestellten Definition auf Echtzeitfähigkeit überprüft werden.

8.5 Optimierungsmöglichkeiten für vorgestellte Programmiertexte

Die in dieser Thesis vorgestellten Programmiertexte zeigen eine einfache und lauffähige Struktur des Prozesses der segmentierten Faltung. Die Analysen der Programme zeigt aber, dass diese nur eine Segmentlänge von minimal $N = 1024$ in Echtzeit verarbeiten können. Um die gewünschten $N = 64$ zu erreichen, müssen die Programmiertexte überarbeitet und optimiert werden. Dies ist nicht Teil der Thesis, dieser Abschnitt soll aber einige mögliche Optimierungspotentiale und Problemstellen zeigen, welche beim Debuggen des Prozesses aufgefallen sind.

Eine zeitliche Auswertung in MATLAB® macht es möglich, jene Stellen im Programmtext zu finden, welche die längste Zeit benötigen und somit das größte Optimierungspotential beinhalten. Die bei der Analyse ermittelten größten Zeitverbraucher sind in Tab. 12 gezeigt.

(Teil-)Prozess	**Laufzeit [Sek.]**	**Anteil [%]**
Gesamtlaufzeit $N = 1024$, $L_y = 10\ Sek.$	$4,694$	100
IFFT der Segmente $Y_{ki}[f]$ in *out_seg*	$2,471$	$52,6$
Berechnen des Ausgabevektors *y_out*	$1,491$	$31,8$
Berechnen von $Y_{ki}[f] = X_k[f] \cdot H_i[f]$	$0,516$	11
Restliche Operationen	$0,216$	$4,6$

Tabelle 12: Auswertung der Laufzeiten des Programmcodes

Die Optimierungsmöglichkeiten sollten der Tabelle nach zunächst für die IFFT und die Ablage der Segmente in *out_seg* ermittelt werden. Das Ablegen eines Vektors im Speicher geschieht noch an anderen Stellen im Programmiertext. Deren Analyse zeigt, dass diese Operation im Vergleich zur IFFT so schnell ausgeführt wird, dass die IFFT als die zeitaufwändigste Operation festgestellt werden kann. In MATLAB® ist bereits die FFTW-Routine eingebaut, welche in den Vergleichen grundsätzlich am schnellsten war. Es kann daher davon ausgegangen werden, dass andere Routinen schlechtere Ergebnisse erzielen würden. Ein Test mit verschiedenen Einstellungen in FFTW zeigte, dass es nur Verbesserungen im Mikrosekundenbereich gibt, wenn die Methode zur Berechnung der IFFT verändert wird. Auch andere Möglichkeiten, die Routine zu konfigurieren brachten keine Verbesserung. Es kann daher nur durch eine Optimierung in der Berechnung Zeit gespart werden. Dazu bietet sich bei den FFT- und IFFT Operationen die Berechnung mit einer Verschachtelung zweier Blöcke als Real- und Imaginär- Teil einer doppelt so langen IFFT an. Dadurch müsste der IFFT-Operator fast nur halb so oft durchlaufen werden, zusätzliche Zeit würde aber durch die Verschachtelung entstehen, ebenso muss diese anschließend wieder rückgängig gemacht werden, wozu ebenfalls Zeit benötigt wird. Ein Experiment zum Vergleich beider Möglichkeiten wurde in einem separaten Programm umgesetzt. Dabei wurden jeweils 10.000 IFFTs durchlaufen, einmal jeweils mit einem Vektor, anschließend mit der Verschachtelung. Die Vektoren enthielten jeweils Zufallsdaten mit Werten von 0.01 bis 1. Dabei erwies sich die Verschachtelung mit einer Zeitersparnis von 35 %

als eine Variante, welche zur Optimierung auf jeden Fall in Betracht gezogen werden sollte. Da die Zahl der Durchläufe dieser Operation bei kleineren Segmentlängen N ansteigt, ist eine zeitliche Ersparnis an dieser aufwändigsten Stelle von Vorteil.

Der zweitgrößte Einfluss auf die Berechnung der segmentierten Faltung ist die Berechnung des Ausgangsvektors *y_out*. Diese läuft so ab, dass innerhalb der Schleife der aktuelle Wert von *y_out* mit einem Ausgabesegment $y_{ki}[n]$ addiert wird. Eine Optimierung dieser Zeilen ist nicht möglich, weil diese nur Additionen enthalten, welche schon die kleinstmögliche Berechnungsdauer auf dem Prozessor darstellen. Sowohl Additionen als auch Subtraktionen werden in der Regel in einem Taktzyklus berechnet, somit ist keine Ersparnis möglich. Eine Analyse mit $N = 64$ zeigt, dass diese Zeilen mit kleinerer Segmentlänge einen entscheidenden Anteil an der Berechnungsdauer haben, welcher nicht optimiert werden kann.

Auch die Kontrollstrukturen wie if-/ else-Verzweigungen werden bei einer Analyse für kleinere Segmentlängen wesentlich häufiger durchlaufen und sind somit auch zeitlich einflussreicher. Ihre Zahl zu vermindern kann daher auch zu einer Ersparnis führen, welche für jene kleinen Segmentlängen die Berechnungszeit senken könnte. Ein Vergleich zeigt diesen Zusammenhang. Eine if-/else-Verzweigung bei $N = 64$ wird bis zu 11,7 Millionen Mal durchlaufen. Je nach Komplexität der zu prüfenden Bedingung sind für diese Prüfung 2,27 bis zu 5,3 Sekunden nötig. Könnten diese eingespart werden, wären dies bei den kleineren Segmentlängen eine nicht unwesentliche Verbesserung.

Ein Blick auf die Multiplikationen der Segmente $X_k[f]$ mit den Segmenten $H_i[f]$ zeigt ebenfalls einen wesentlichen Einfluss auf die Berechnung. Der Prozentsatz an Zeit für diese Operationen ist für kleinere Segmentlängen ebenfalls bei über 10 % (Bsp. $N = 64 \rightarrow 14,4\%$). Komplexe Multiplikationen, welche hier durchgeführt werden, könnten optimiert werden, wenn man diese für den Prozessor anpasst. MATLAB® verwendet für die Berechnung einen Standardalgorithmus. VST-Plug-Ins, welche die segmentierte Faltung mit $N = 64$ umsetzen können, nutzen optimierte Multiplikationsverfahren. Diese zu integrieren, könnte laut Angaben der verschiedenen Algorithmen eine Ersparnis von bis zu 15 % bringen. Diese könnte in diesem Fall auch erreicht werden, weil die Multiplikationen immer mit gleichem Zahlenformat durchgeführt werden, lediglich die Werte sind verschieden.

Dennoch bleibt die Frage, warum ein VST-Plug-In mit mehreren 1000 Zeilen Programmtext schnellere Umsetzungen schafft als ein MATLAB®- oder C-Code mit ca. 150 Zeilen Programmtext (in beiden Fällen ohne FFT-Routine betrachtet). Diese soll nun abschließend betrachtet werden, um weitere Optimierungsmöglichkeiten vorzustellen.

Grundsätzlich spielt an vielen Stellen die Genauigkeit eine entscheidende Rolle. Wie viele Nachkommastellen verarbeitet werden müssen, damit kein wahrnehmbarer Unterschied in der Ergebnisausgabe besteht ist wohl eine der wesentlichen Entscheidungen bei der Optimierung für eine Prozessorplattform. Ein Runden wäre bei einer entsprechenden Dynamik durchaus denkbar, sodass dadurch die Menge des Speichers

gespart werden könnte. In Bezug auf die Berechnungsdauer würden gerundete Werte zu einer Steigerung der Rechengeschwindigkeit in allen Prozessschritten führen. Es muss aber sichergestellt werden, dass die Ergebnisse der FFT-Operationen immer noch korrekt sind. An dieser Stelle können die größten Abweichungen entstehen, wenn zuvor gerundet wird.

Neben der Rundung von Fließkommazahlen steht auch die Verarbeitung komplexer Zahlen in der Liste der möglichen Optimierungen. Für C gibt es eine Vielzahl von Bibliotheken, welche komplexe Zahlen in optimierter Form verarbeiten. Ebenso können reelle Werte schneller Verarbeitet werden, wenn man auch zwei reellen Zahlen eine komplexe Zahl erstellt und mit dieser die Berechnungen durchführt. Die Geschwindigkeit der Verarbeitung komplexer Zahlen ist natürlich auch wesentlich vom Zielsystem bzw. dem verwendeten Prozessor abhängig.

9 Fazit und Aussichten

Die Analysen der Thesis haben gezeigt, dass die schnelle, segmentierte Faltung durch eine Vielzahl von Faktoren in ihrer Umsetzung beeinflusst wird. Geringe Latenz erfordert einen hohen Rechenaufwand, welcher oft über Prozessorgrenzen hinausgeht, wenn eine einfache Umsetzung gewählt wird. Der Speicheraufwand kann hingegen durch Optimierung in einem Rahmen gehalten werden, sodass eine Prognose für diverse Segmentlängen möglich wird. Der Arbeitsaufwand hingegen kann immer nur für ein festes Zeitfenster angegeben werden. Der Zusammenhang der Segmentlänge mit den drei Größen Latenz, Arbeits- und Speicheraufwand macht eine Auswertung vom Programmtexten der segmentierten Faltung einfacher, weil die Eckdaten der Berechnung bereits mit Hilfe von Segmentlänge und der Länge der Impulsantwort ermittelt werden können.

Die Wahl der einfachen Umsetzung führt auf aktuellen GPPs noch zu keinen brauchbaren Ergebnissen. Eine Auswertung der Analyse für den im Apple iPhone 5 verbauten A6X- Prozessor zeigte, dass auch dieser noch nicht in der Lage wäre, den Arbeitsaufwand der segmentierten Faltung in der vorgestellten Programmstruktur zu bewältigen. Weitere Zahlenvergleiche mit Werten des Samsung Galaxy S4 und des Google Nexus 4 zeigten ebenfalls, dass diese Umsetzung mit geringerer Latenz nicht funktionieren wird. Als Ergebnis ist daher für eine einfache Umsetzung der segmentierten Faltung ein Wert von $N = 1024$ als minimale Segmentlänge festzuhalten, bei welcher die Berechnung in Echtzeit möglich ist. Selbst durch Optimierung konnte eine Segmentlänge von $N = 512$ nicht erreicht werden.

Um eine Umsetzung der segmentierten Faltung auf einer mobilen Plattform dennoch mit geringer Latenz möglich zu machen, sind weitere Optimierungen erforderlich, die über den Rahmen dieser Thesis hinaus gehen. Programmierstrukturen lassen eine Vielzahl von Einsparungen zu, welche allerdings eher in den Aufgabenbereich erfahrener Programmierer gehören. Eine Umsetzung des Prozesses würde neben der Optimierung auch die Wahl des Zielsystems beinhalten. Hier bleibt festzuhalten, dass DSPs als günstige und einfach zu programmierende Prozessoren die in den Analysen favorisierte Lösung sind.

A Anhang

A.1 Matlab-Code zur Berechnung der segmentierten Faltung

```
% globale Variablen festlegen
s = 1;
o = 0;
ind = 1;
l = 1;
m = 1;
run = 2;
k = 1;
N = 64;
fs = 44100;
Lh = 107008;
Lx = 396900;
in = zeros(ceil(Lh / N), N);
in_freq = complex(zeros(ceil(Lh / N), 2*N));
imp_freq = complex(zeros(ceil(Lh / N), 2*N));
out_seg = complex(zeros(2*ceil(Lh / N), 2*N));
out = complex(zeros(ceil((Lx + Lh - 1)/N + 2), N));

% Impulsantwort laden

h = imp_sig(1,Lh);

% ---INIT: Impulsantwort in Segmenten transformieren --- %

for i=1:ceil(Lh/N)-1
    hi = h((i-1)*N+1:(i-1)*N+N);
    HI = fft(hi, 2*N);
    imp_freq(i,:) = HI;
end

% ---SYSTEMSTART--- %

x = input_sig(1,Lx);

while run > 1
    if m > Lx/N
        xk = zeros(1,N);
    else
        xk = x((m-1)*N+1:(m-1)*N+N);
    end

% --- 1: Transformieren von Eingangssignalsegmenten --- %
```

```
in_freq(s, : ) = fft(xk, 2*N);

% --- 2: Multiplikationen mit den Segmenten Hi[f] und IFFT --- %

t = s;

for i = 1:1:k
    Yki = in_freq(t, :).*imp_freq(i, :);
    out_seg(l, :) = ifft(Yki, 2*N);
    l = l + 1;
    t = t - 1;
    if t == 0
        t = ceil(Lh/N);
    end
    if(l == 2*ceil(Lh/N)+1)
        l = 1;
    end
end


% --- 3: Overlap-Add --- %
if k == 1
    out(m , : ) = out_seg(1,1:N);
else
    o = l - 1;
    if o == 0
        o = 2*ceil(Lh/N);
    end
    for i = 1:1:k
        out(m , : ) = out(m , : ) + out_seg(o,1:N);
        if o == 1
            o = 2*ceil(Lh/N);
        else
            o = o - 1;
        end
    end
    if k == ceil(Lh/N)
        for i = 1:1:k
            out(m , : ) = out(m , : ) + out_seg(o,N+1:2*N);
            if o == 1
                o = 2*ceil(Lh/N);
            else
                o = o - 1;
            end
```

```
        end
    else
        for i = 1:1:k-1
            out(m , : ) = out(m , : ) + out_seg(o,N+1:2*N);
            if o == 1
                o = 2*ceil(Lh/N);
            else
                o = o - 1;
            end
        end
    end
end

if k < ceil(Lh/N)
    k = k + 1;
end

m = m + 1;

if s == ceil(Lh/N)
    s = 1;
else
    s = s + 1;
end

end
```

A.2 C-Code zur Berechnung der segmentierten Faltung

```
#include <stdio.h>
#include <stdlib.h>
#include <fft.h>

int main(int argc, char *argv[]){
    int N;
    unsigned int Lh;
    int Lx;
    int fs;
    int i;
    int j;
    int t;
    int k = 1;
    int l = 1;
```

```
    int m = 1;
    int o = 0;
    int s = 1;
    int run = 2;
    unsigned int segments = (int)(Lh/N + 0.5);

    double in_freq[segments][2*N];
    double imp_freq[segments][2*N];
    double out_seg[segments][2*N];
    double out[N];
    double xk[N];
    double h[Lh];
    double hi[N];
    double Yki[2*N];

 // --- INIT: Impulsantwort in Segmenten transformieren ---

    h = dataimport(fs, Lh, 'dateiname');
    for (i=0; i < ceil(Lh/N); i++) {
        for (j = 0; j < N; j++) {
            hi[j] = h[i*N+j];
        }
        imp_freq[i] = fft(hi, 2*N);
    }

 while (run > 1) {

     xk = audiostream(fs, N);

// --- 1: Transformieren von Eingangssignalsegmenten ---

     in_freq[s] = fft(xk, 2*N);

// --- 2: Multiplikationen mit den Segmenten Hi[f] und IFFT ---

     t = s;

     for (i = 1; i<k+1; k++) {
         for (j = 1; j < 2*N+1; j++) {
              Yki[j] = in_freq[t][j]*imp_freq[i][j];
         }
         out_seg[l] = ifft(Yki, 2*N);
         l = l + 1;
         t = t - 1;
         if (t == 0) {
```

```
            t = ceil(Lh/N);
        }
        if(l == 2*ceil(Lh/N)+1) {
            l = 1;
        }
    }

// --- 3: Overlap-Add ---

    for(i = 0; i < N+1; i++) {
        out[i] = 0;
    }

    if (k == 1) {
        for(i = 0; i < N; i++) {
            out[i] = out_seg[0][i];
        }
    }
    else {
        o = l - 1;
        if (o == 0) {
            o = 2*ceil(Lh/N);
        }
        for (i = 1; i < k+1; i++) {
            for(j = 0; j < N; j++) {
                out[j] = out[j] + out_seg[o][j];
            }
            if (o == 1) {
                o = 2*ceil(Lh/N);
            }
            else {
                o = o - 1;
            }
        }
        if (k == ceil(Lh/N)) {
            for (i = 1; i < k+1; k++) {
                for(j = 0; j < N; j++) {
                    out[j] = out[j] + out_seg[o][N+j];
                }
                if (o == 1) {
                    o = 2*ceil(Lh/N);
                }
                else {
                    o = o - 1;
                }
```

```
            }
        }
        else {
            for (i = 1; i < k; k++) {
                for(j = 0; j < N; j++) {
                    out[j] = out[j] + out_seg[o][N+j];
                }
                if (o == 1) {
                    o = 2*ceil(Lh/N);
                }
                else {
                    o = o - 1;
                }
            }
        }
      }

    if (k < ceil(Lh/N)) {
        k = k + 1;
    }

    if (s == ceil(Lh/N)) {
        s = 1;
    }
    else {
        s = s + 1;
    }

 }
 system("PAUSE");
 return 0;
}
```

Literatur

[Mey, 2009] Meyer, Martin: Signalverarbeitung, 5. Auflage, 2009, Vieweg + Teuber Verlag

[Wer, 2006] Werner, Martin: Digitale Signalverarbeitung mit MATLAB, 3. Auflage, 2006, Vieweg Verlag

[Neu, 2012] Neubauer, André: DFT – Diskrete Fourier-Transformation, 2012, Springer Verlag

[Chu, 2000] Chu, Eleanor & George, Alan: Inside the FFT Black Box, 2000, CRC Press LLC

[Gar, 1995] Gardner, William G.: Efficient convolution without input-output delay. J. Audio Eng. Soc. 43 (3), 127-136, 1995

[Jon, 2006] Jones, Douglas L. (2006), „Decimation-in-time (DIT) Radix-2 FFT“, URL: http://cnx.org/content/m12016/latest/ [Stand: 22.10.2012]

[Ol, 1997] Olschner, Ralf (1997), „Referat Digitale Signalprozessoren“, URL: http://www.weblearn.hs-bremen.de/risse/RST/SS97/DSP/DSP_3.HTM [Stand: 30.10.2012]

[Szi, 2003] Szillat, Holger (2003), „FFT – Die schnelle Fourier Transformation“, URL: http://instant-chaos.com/downloads/fourier-slides.pdf [Stand: 21.11.2012]

[Ny, 2007] Nyack, Cuthbert (2007), „FFT, Brief Introduction“, URL: http://dspcan.homestead.com/files/FFT/fftintr.htm [Stand: 21.11.2012]

[Zöl, 2005] Zölzer, Udo: Digitale Audiosignalverarbeitung, 3. Auflage, 2005,Vieweg + Teubner Verlag

[Smi, 2007] Smith, Steven W. (2007), „FFT convolution and the overlap-add method“, URL: http://eetimes.com/design/signal-processing-dsp/4017492/ [Stand: 02.12.2012]

[TI] McKeown, Mark (2013), „FFT Implementation on the TMS320VC5505, TMS320C5505, and TMS320C5515 DSPs“, URL: http://www.ti.com [Stand: 12.07.2013]

[Bat, 2011] Battenberg, Eric & Avižienis, Rimas (2011), „Implementing Real-Time Partitioned Convolution Algorithmus On Conventional Operating Systems“, URL: http://www.eecs.berkeley.edu/~ericb/school/partconvDAFx2011.pdf [Stand: 02.12.2012]

[Smi, 2011] Smith, Steven W. (2011), „The Scientist and Engineer's Guide to Digital Signal Processing“, URL: http://www.dspguide.com [Stand: 02.12.2012]

[Fra, 2004] Frantz, Gene (2004), „Comparing Fixed- and Floating-Point DSPs“, URL: http://www.ti.com/lit/wp/spry061/spry061.pdf [Stand: 08.01.2013]

[Ko] Koch, Robert: Systemarchitektur zur Ein- und Ausbruchserkennung in verschlüsselten Umgebungen, 1. Auflage, 2011, Books on Demand Verlag