HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

ELEKTROTECHNIK

VERTIEFUNGSFACH DIGITALE SIGNALVERARBEITUNG

Schlussbericht der Studienarbeit SS 2001

Pitch-Extractor für Kanal-Vocodersystem

Andy Rohr

11. Juli 2001

Typeset by LaTeX

Inhaltsverzeichnis

Aufgabenstellung **iii**

Abstract **v**

1 Einführung **1**
1.1 Sprachcodierung . . . 1
1.2 Bestehendes System . . . 1
1.3 Voraussetzungen an den Leser . . . 2
1.4 Arbeitsumgebung . . . 2

2 Projektplanung **5**
2.1 Struktur . . . 5
2.2 Projektplan . . . 6
2.3 Zeitaufwand . . . 7

3 Theorie **9**
3.1 Allgemeines zur Grundfrequenzbestimmung (GFB) . . . 9
3.1.1 Definition der Grundfrequenz . . . 9
3.1.2 Verarbeitungsstufen . . . 9
3.1.3 GFB-Algorithmen nach dem Prinzip der Kurzzeitanalyse . . . 10
3.1.4 Bestimmung der Pitch nach einer doppelten Spektraltransformation . . . 13
3.2 Verwendete Methoden . . . 13
3.3 Bestimmung der Anregungsart . . . 14

4 Entwicklung und Resultate **15**
4.1 Simulation in MATLAB . . . 15
4.1.1 Frame-by-Frame-Vergleich der Pitch-Extraction-Methoden . . . 15
4.1.2 Vergleich der Pitch-Extraction-Methoden über ein ganzes Wavefile . . . 15
4.1.3 Framelänge . . . 16
4.1.4 Beispiele von extrahierten Pitchverläufen . . . 17
4.1.5 Schlussfolgerungen . . . 33
4.2 Implementation für den DSP . . . 33
4.2.1 Pitch-Extractor mit Pitch-Generator . . . 33
4.2.2 Einbau des Pitchextractors in das Vocodersystem . . . 38
4.2.3 Datenrate bei der Übertragung . . . 41
4.2.4 Resultate . . . 41
4.2.5 Verbesserungsansätze . . . 41
4.3 Schlusswort . . . 42
4.3.1 Ziel erreicht? . . . 42
4.3.2 Persönliche Bemerkungen . . . 42

A Listings **43**
A.1 MATLAB Programme (Simulation) 43
A.1.1 Frame-by-Frame-Vergleich der Pitch-Extraction-Methoden 43
A.1.2 Vergleich der Pitch-Extraction-Methoden über ein ganzes Wavefile 44
A.2 DSP-Programme (Implementation) 45
A.2.1 Pitch Extractor 45
A.2.2 Vocoder 4 52

Literaturverzeichnis **73**

Abbildungsverzeichnis **75**

Aufgabenstellung

Aufgabe

Im Labor für digitale Signalverarbeitung der HSR wurde zu Demonstrationszwecken ein Vocodersystem entwickelt, worin bei der Synthese ein externes Sprachmelodiesignal als Pitch verwendet wird. Dieses Signal kann (z.B. ab Keyboard) über einen Analogeingang dem Synthesizer-DSP eingespeist werden. Mit einem im Rahmen der Arbeit [2] eingebauten, durch die Frequenzinformation steuerbaren Pitchgenerator kann diese Schwingung aber auch intern erzeugt werden. In dieser Aufgabe soll nun die Analyseseite durch den Einbau eines Pitch-Extraktors ergänzt werden. Dieser hat die Aufgabe, diese Frequenzinformation aus dem Eingangs-Sprachsignal laufend, d.h. in gleichen Zeitschritten wie die Sprachmerkmale zu gewinnen (Grundfrequenzanalyse). Zur Pitch-Analyse sind verschiedene Verfahren bekannt geworden [1]. In einer ersten Phase ist daher eine Evaluation des zu verwendenden Verfahrens unter Berücksichtigung der im bestehenden Vocodersystem verfügbaren Rechenkapazität und des noch vorhandenen Speichers notwendig. Nach ggf. notwendigen Simulationen für den Vergleich der Verfahren ist der gewählte Algorithmus ins Echtzeit-Vocodersystem zu integrieren.

Literatur

(1) Vary P., Heute U. und Hess W., Digitale Sprachsignalverarbeitung. B.G. Teubner Verlag, Stuttgart 1998.

(2) Tresch O. und Sibler G. Pitch-Generator für Kanal-Vocodersystem. Studienarbeit W01-02 am Labor für Digitale Signalverarbeitung der HSR, WS 2000/2001

(3) Handbücher zum SHARC-DSP-System im DS-Labor und Unterlagen zu den im DS-Labor vorhandenen Algorithmen

Bericht

Über die Arbeit ist eine Dokumentation zu verfassen. Alle verwendeten Quellen sind im Literaturverzeichnis dieses Berichts anzugeben (Hinweise im Text). Der Bericht ist in doppelter Ausführung abzugeben; ein Exemplar verbleibt am Labor für digitale Signalverarbeitung der HSR, das Doppel erhalten die Verfasser nach der Korrektur und der Bewertung zurück. Die erstellten und verwendeten Programme sowie der Text des Berichtes sind zur laborinternen Archivierung dem Laborassistenten zu übergeben.

Termine, Bedingungen

Gemäss Vorgaben und Terminplan des Vorstandes der Abteilung für Elektrotechnik. Ausgabe der Aufgabenstellung, Beginn der Arbeit: Montag, 26.03.2001 Arbeitsplatz: DS-Labor 1219 (zuständig:

P. Roffler, Laborassistent) Abgabe des Berichts, Ende der Arbeit: Freitag, 13.07.2001

Assistenz

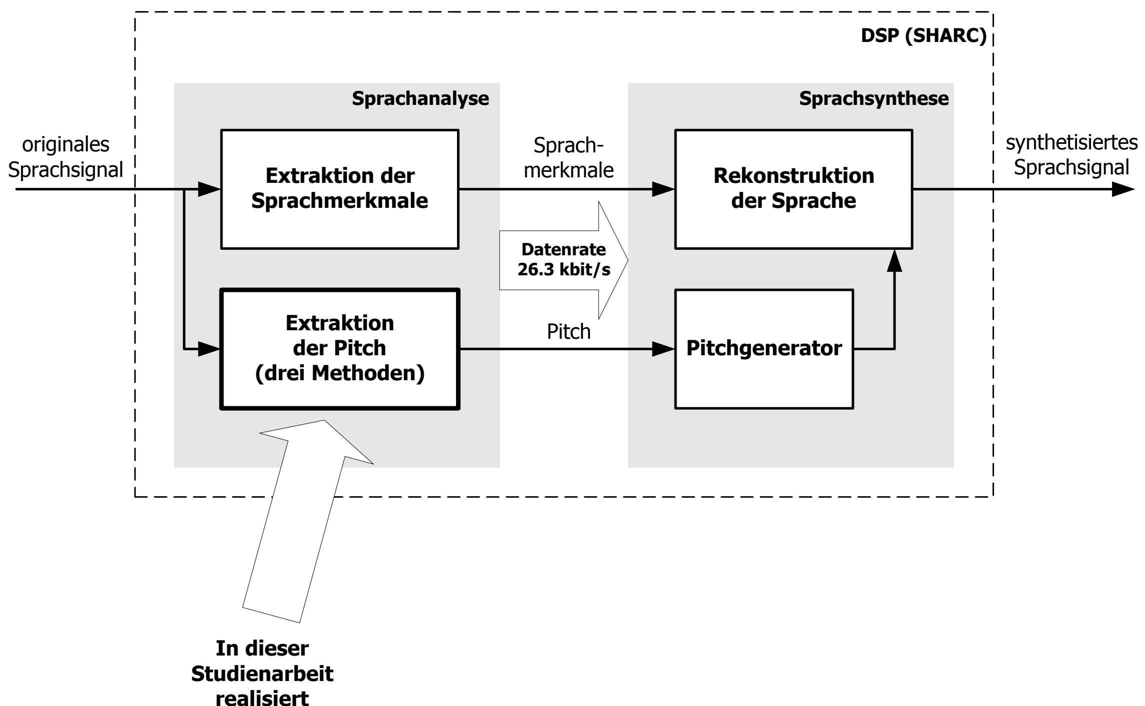
P. Roffler, Laborassistent (Arbeitsplatz: Meo-Labor)
A. Rüegg (in dringenden Fällen; Arbeitsplatz: Ds-Labor).

Rapperswil, 26.03.2001
Prof. Dr. A. Schüeli

Abstract

Im Rahmen dieser Studienarbeit wurde ein bestehendes Kanal-Vocodersystem um die Fähigkeit erweitert, die Pitch, also die Tonhöhe eines Sprachsignals, zu extrahieren. Dieses System läuft auf einem SHARC-DSP der Firma ANALOG DEVICES. Verschiedene in der Literatur bekannte Methoden zur Pitchanalyse wurden studiert. Davon wurden drei zur Umsetzung ausgewählt und miteinander verglichen. In einer Simulationssreihe mit MATLAB wurden die Vor- und Nachteile der drei Methoden eruiert und die Funktion verifiziert. Anschliessen konnte der Algorithmus für den DSP implementiert werden. Zuerst wurde ein Testprogramm erstellt, das nur den entwickelten Pitchextraction-Algorithmus enthält, so dass die Funktion dieser Implementation überprüft werden konnte. Nach erfolgreichen Test und Optimierungen konnte dieser Algorithmus in das bestehende Vocodersystem integriert werden. Anschliessende Test bestätigten die richtige Funktion.

Der auf der Analyseseite angebrachte in dieser Studienarbeit entwickelte *Pitchextractor* steuert auf der Syntheseseite einen *Pitchgenerator* an. Dieser wurde in einer Vorgängerstudienarbeit entwickelt und in das ursprüngliche Vocodersystem eingebaut.



1 Einführung

1.1 Sprachcodierung

Die Mehrzahl der Algorithmen zur Codierung von Audiosignalen stützen sich auf Modelle: Bei der Sprachcodierung vor allem auf Modelle der Spracherzeugung, bei der Musikcodierung vor allem auf Modelle des Gehörs [1]. Bei der Sprachcodierung kann man zwischen der Signalform-Codierung und dem Vocoder-Prinzip unterscheiden. Eine Zwischenstellung nehmen die Hybrid-Coder ein. Bei den Vocoder-Verfahren wird nicht die Signalform, d.h. die Zeitfunktion codiert, sondern ein Parametersatz des zugrundeliegenden Modells der Spracherzeugung. Wichtig sind dabei diejenigen Parameter, welche die Enveloppe des Kurzzeitamplitudenspektrums bestimmen.

Der Kanal-Vocoder

Beim Kanal-Vocoder sind dies z.B. die Amplituden des in Teilbänder («Kanäle») zerlegten Sprachspektrums. Die **Sprachanalyse** besteht also hier primär aus der Ermittlung des frequenzmässig diskretisierten Kurzzeit-Betragsspektrums. Zusätzlich muss für die stimmhaften Intervalle als weiterer Parameter die Grundfrequenz der Sprachmelodie, d.h. die «Pitch» ermittelt und mit codiert werden. In der **Synthese** muss diese «Sprachmelodie» als oberwellenreiche, kurzzeitperiodische Schwingung («Träger») erzeugt, in die Teilbänder zerlegt mit den Amplituden der Kanäle moduliert werden. Anschliessend werden diese rekonstruierten Teilbandsignale wieder zum Gesamtsignal zusammengesetzt. Für stimmlose Intervalle wird anstelle der Pitch-Pulsfolge ein Rauschsignal verwendet. Das dazu nötige Unterscheidungsmerkmal «stimmhaft - stimmlos» kann in der Synthese zusammen mit der Pitch-Detektion ermittelt werden. Der Satz all dieser Parameter muss alle 20...50 ms erneuert werden.

Im Vergleich mit den Signalform-Codern (z.B. gewöhnliche PCM) kann mit diesem Verfahren eine ansehnliche Datenreduktion erreicht werden. Darüberhinaus kann ein Vocodersystem auch im Unterhaltungsbereich zur künstlichen Verfremdung von Sprache und Gesang verwendet werden. Durch die Einspeisung einer vom ursprünglichen Sprachsignal unabhängigen, externen, «künstlichen» Sprachmelodie bei der Synthese entstehen interessante akustische Effekte.

1.2 Bestehendes System

Im Labor für digitale Signalverarbeitung der HSR wurde zu Demonstrationszwecken ein Vocodersystem entwickelt, worin bei der Synthese ein externes Sprachmelodiesignal als Pitch verwendet wird. Dieses Signal kann (z.B. ab Keyboard) über einen Analogeingang dem Synthesizer-DSP eingespeist werden. Mit einem im Rahmen der Arbeit [2] eingebauten, durch die Frequenzinformation steuerbaren Pitchgenerator kann diese Schwingung aber auch intern erzeugt werden.

In dieser Studienarbeit soll nun das bestehende Vocoder-System durch den Einbau eines Pitch-Extraktors vervollständigt werden. Dieser hat die Aufgabe, diese Frequenzinformation aus dem Eingangs-Sprachsignal laufend, d.h. in gleichen Zeitschritten wie die Sprachmerkmale zu gewinnen (Grundfrequenzanalyse) und diese dem Pitchgenerator einspeisen.

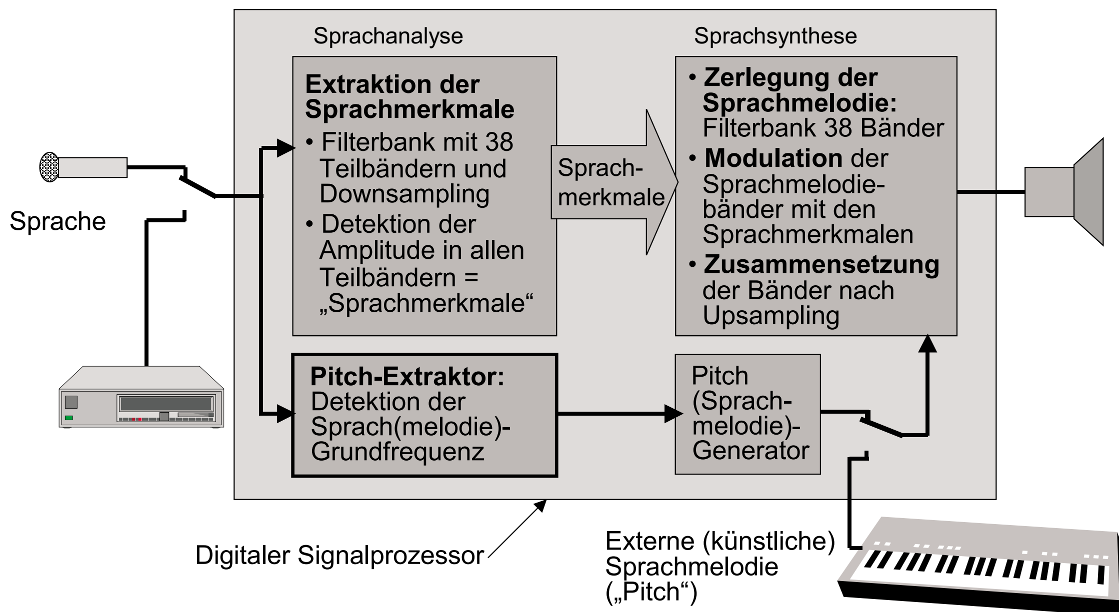


Abbildung 1.1: *Vollständiges Labor-Vocodersystem, ergänzt durch Pitch-Extraktor*

In einer ersten Phase wurden daher Simulationen mit drei verschiedenen bekannten Verfahren durchgeführt, um die Eignung für einen Einsatz im Vocodersystem zu eruieren. In einer zweiten Phase wurden diese in das bestehende Echtzeit-Vocodersystem (DSP) integriert.

1.3 Voraussetzungen an den Leser

Da es sich bei dieser Studienarbeit um eine Fortsetzung einer Vorgängerstudienarbeit [2] handelt, ist die Kenntnis dieser von Nöten. Darin wird auch das allgemeine Vocoderprinzip und grundlegendes zur Sprachcodierung behandelt, weshalb an dieser Stelle nicht im Detail auf diese Themen eingegangen wird.

Diese Vorgängerstudienarbeit ging wiederum von einem bestehenden Vocoder-System aus. In diesem System wurde der Parameter *Grundfrequenz* oder *Pitch* nicht codiert. Es bestand lediglich die Möglichkeit ein Signal als Träger in den Syntheseblock einzuspeisen, der als Grundfrequenz für das synthetisierte Sprachsignal diente. Im Rahmen der Vorgängerstudienarbeit wurde dieses System um einen Block erweitert: Den Pitchgenerator. Dieser ermöglicht die Grundfrequenz intern zu erzeugen.

In Rahmen dieser Studienarbeit wurde das System durch ein weiteres Modul erweitert: Der Pitch-Extractor. Dieser steuert auf der Syntheseseite wiederum den Pitchgenerator an. Somit wurde das System «komplett» und kann nun auch den Parameter *Grundfrequenz* verarbeiten.

1.4 Arbeitsumgebung

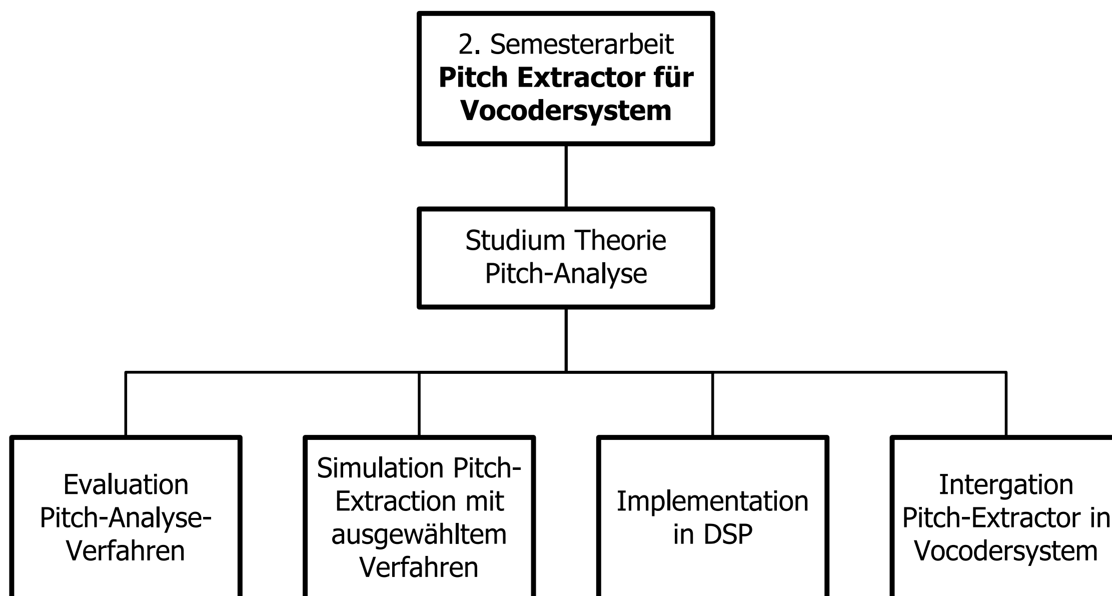
Diese Studienarbeit wurde an einem Arbeitsplatz im Labor für Digitale Signalverarbeitung an der HSR durchgeführt. Dabei konnte ausschliesslich auf vorhandene Arbeitsmittel zurückgegriffen werden.

Zentrales Arbeitsmittel war ein Standard-PC mit einem DSP-Board (PCI) von BITTWARE. Dieses enthält einen DSP von ANALOG DEVICES (ADSP-21062 SHARC) und ein A/D-D/A-Wandler ShuttleBoard. Die Verbindung zur Aussenwelt wurde durch ein BNC-Interface (I/O-Box) hergestellt.

Für die Simulationen wurde MATLAB 6.0 Release 12 (6.0.0.88) von MATHWORKS verwendet. Um die Sourcecodes für den DSP zu übersetzen, kam der Compiler aus der Entwicklungsumgebung Release 3.3 von ANALOG DEVICES zum Einsatz. Dieser wurde über ein an der HSR entwickeltes MATLAB-GUI (*comptool*) bedient. Der Datenaustausch zwischen MATLAB und DSP wurde über die an der HSR entwickelten Library `matlab.dll` (MEX-File) bewerkstelligt.

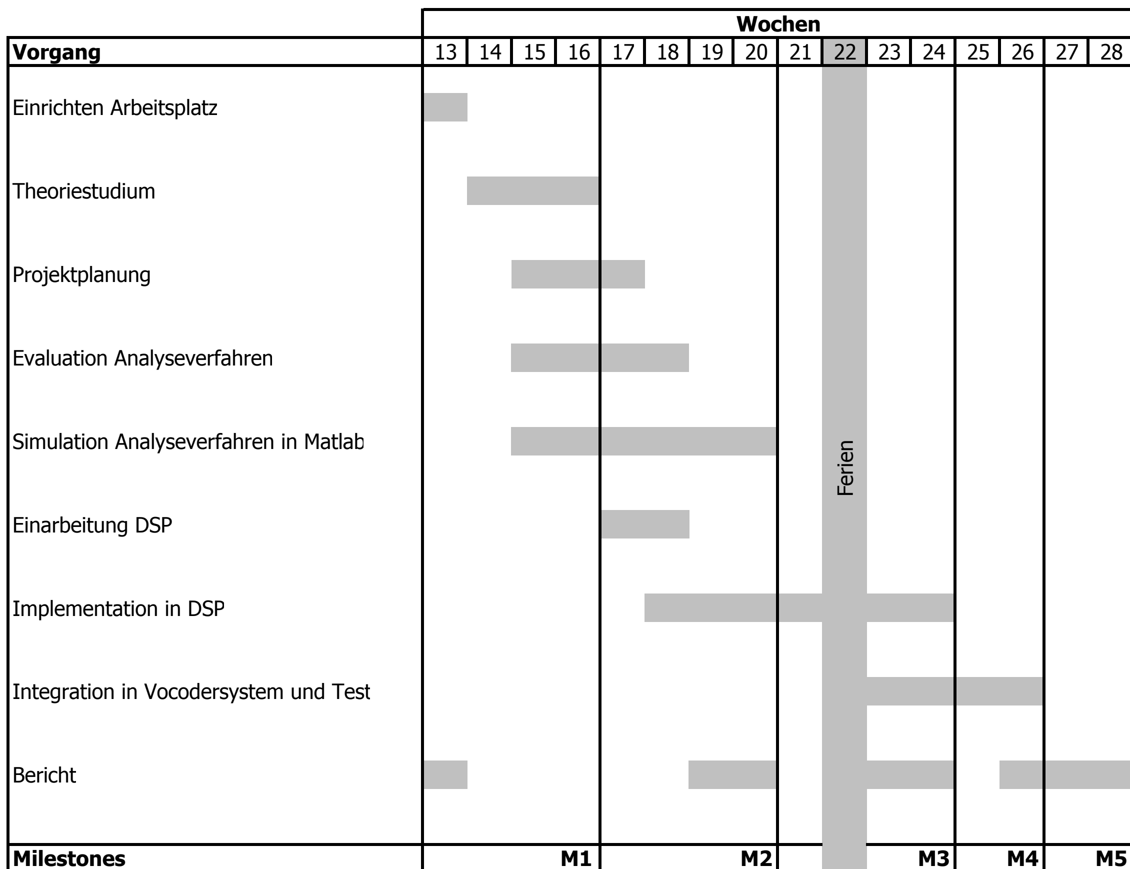
2 Projektplanung

2.1 Struktur



Diese Studienarbeit lässt sich in obengenannte Themenbereiche aufteilen, die separat behandelt wurden. Gleichzeitig stimmt die Anordnung auch mit der Bearbeitungsreihenfolge zusammen.

2.2 Projektplan



Milestones:
M1 Theorie zu Pitch Extraction aufgearbeitet
M2 Simulation des evaluierten Pitch Exctraction Verfahrens beendet
M3 Pitch Extraction in DSP implementiert
M4 Pitch Extractor in Vocodersystem integriert
M5 Schlussbericht

Termine:

Beginn der Arbeit	26.03.2001
Präsentation an Studierende 2. Jahr	11./12.07.2001
Abgabe der Schlussberichts	13.07.2001

2.3 Zeitaufwand

Rubrik	Zeitaufwand [h]
Projektplanung	2.08
Studium Theorie	5.36
Simulation MATLAB	20.50
Einarbeiten DSP	8.45
Implementation DSP	36.06
Dokumentation	61.50
Total	**133.94**

Durchschnittliche Wochenstundenzahl: 8.4 h

3 Theorie

3.1 Allgemeines zur Grundfrequenzbestimmung (GFB)

Die Grundfrequenz nimmt eine Schlüsselstellung unter den Sprachsignalparametern ein. Ein Sprecher überträgt mit ihr wichtige «nonverbale» Information, wie z.B. *das ist eine Frage* oder *das meine ich ironisch*. Das menschliche Ohr ist gegenüber Änderungen der Grundfrequenz um eine Grössenordnung empfindlicher als gegenüber anderen Sprachsignalparametern. Die empfundene Qualität einer Sprachcodierung hängt wesentlich davon ab, wie gut und fehlerfrei die Messung der Grundfrequenz erfolgt.

Aus einer Reihe von Gründen zählt die Grundfrequenzbestimmung jedoch zu den schwierigsten Aufgaben der Sprachsignalverarbeitung. Genaueres kann in [1] Seite 196 gefunden werden. Daher kam es zu einer Entwicklung von hunderten von Verfahren zur Grundfrequenzbestimmung. Keines funktioniert für alle Aufgabestellungen einwandfrei. Die Auswahl eines bestimmten Verfahrens hängt somit von der jeweiligen Anwendung wie auch der Beschaffenheit der zu verarbeitenden Sprachsignale ab.

3.1.1 Definition der Grundfrequenz

Die Grundfrequenz eines Sprachsignals kann auf verschiedene Arten definiert werden. In [1] werden fünf verschiedene genannt. Es existieren daher auch verschiedene Messmethoden, die, auf ein ideales Sprachsignal (periodisch, stationär) angewendet, auch identische Resultate liefern. Jeder Messmethode liegt eine dieser Definitionen zu Grunde. Doch da i.A. ein Sprachsignal nicht stationär ist, beeinflussen die Aspekte der Messmethode das Messergebnis, so dass die verschiedenen Methoden unterschiedliche Ergebnisse liefern.

In dieser Studienarbeit wurde folgende Definition aus [1] verwendet:

> T_0 *ist definiert als die mittlere Dauer mehrerer aufeinanderfolgenden Grundperioden. Auf welche Weise die Mittelung erfolgt und wieviele aufeinanderfolgende Perioden in die Messung involviert sind, bleibt dem einzelnen Algorithmus überlassen.*

3.1.2 Verarbeitungsstufen

Jeder GFB-Algorithmus lässt sich in drei Verarbeitungsstufen einteilen: die Vorverarbeitungsstufe, die Extraktionsstufe und die Nachbearbeitungsstufe. Die Extraktionsstufe führt die eigentli-

Abbildung 3.1: *Die drei Verarbeitungsstufen eines GFB-Algorithmus*

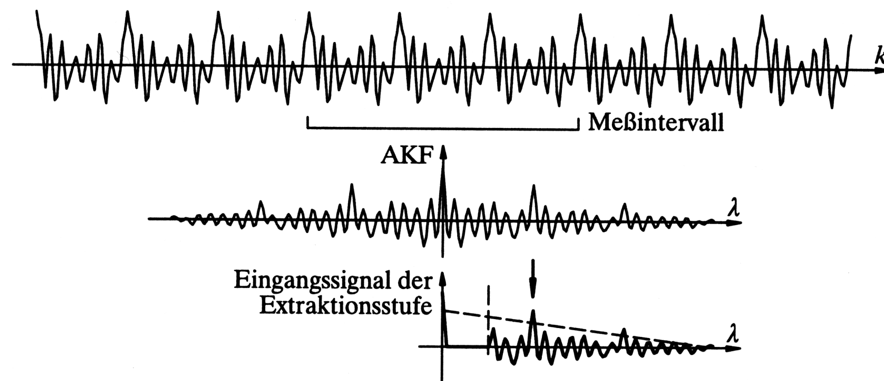


Abbildung 3.2: *Typische Vorverarbeitungsstufe einer GFB-Algorithmus nach dem Prinzip der Kurzzeitanalyse (hier: Autokorrelationsfunktion). (Quelle: [1])*

che Messung durch: sie verwandelt das Eingangssignal in eine Folge von Schätzwerten für die Grundperiodendauer bzw. Grundfrequenz. Die Aufgabe der Vorverarbeitungsstufe besteht in der Datenreduktion, damit die Aufgabe der Extraktionsstufe erleichtert wird. Die Nachbearbeitungsstufe arbeitet mehr anwendungsorientiert: zu ihren Aufgaben gehören Fehlerkorrektur, Glättung des Grundfrequenzverlaufs oder graphische Ausgabe.

3.1.3 GFB-Algorithmen nach dem Prinzip der Kurzzeitanalyse

Die verschiedenen GFB-Algorithmen lassen sich in zwei Hauptgruppen aufteilen: Solche, die nur im Zeitbereich arbeiten und solche, die eine Kurzzeittransformation durchführen und damit blockorientiert arbeiten. Da die in dieser Studienarbeit verwendeten Algorithmen letzteres Prinzip anwenden, wird im Folgenden auf die andere Kategorie nicht weiter eingegangen.

Um eine Kurzzeittransformation durchzuführen, muss eine Anzahl Messpunkte des Zeitsignals zu einem Block oder Frame zusammengefasst werden. Dieses darf nicht zu lang sein, da sonst Änderungen der Grundfrequenz nicht schnell genug erfasst werden können, und es darf nicht zu kurz sein, da mindestens zwei bis drei vollständige Grundperioden in diesem Frame Platz finden müssen, um eine zuverlässige Funktion des Algorithmus zu gewährleisten. Die anschliessende Transformation soll die Information über die Periodizität des Zeitsignals, die über dieses «verstreut» vorhanden ist, in einen markanten Punkt fokussieren. Dieser Punkt kann dann mittels Spitzenwertdetektor ermittelt werden (siehe Abbildung 3.2). Dabei gehen aber Phasenbeziehungen gegenüber dem Zeitsignal verloren, was den Algorithmus gegenüber Phasenverzerrungen wenig empfindlich macht.

GFB mit Hilfe doppelter Spektraltransformation und nichtlinearer Verzerrung im Frequenzbereich

Anstatt einer direkten AKF kann als Kurzzeittransformation auch eine Fouriertransformation ausgeführt werden. Die direkte Bestimmung der Grundfrequenz aus dem ersten Maximum des Leistungsdichtespektrums ist unzuverlässig. Stattdessen wird das Leistungsdichtespektrums nichtlinear verzerrt und wieder in den Zeitbereich zurück transformiert (siehe Abbildung 3.3). Nach dieser Rücktransformation lässt sich T_0 durch ein signifikantes Maximum in der Zeitbereichsfunktion ableiten.

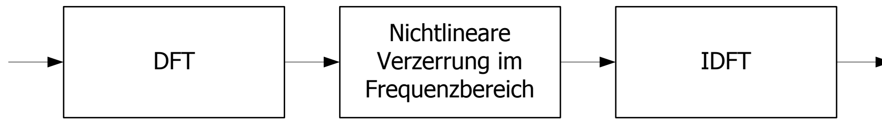


Abbildung 3.3: *GFB-Vorverarbeitungsstufe mit doppelter Spektraltransformation und nichtlinearer Verzerrung im Spektralbereich.*

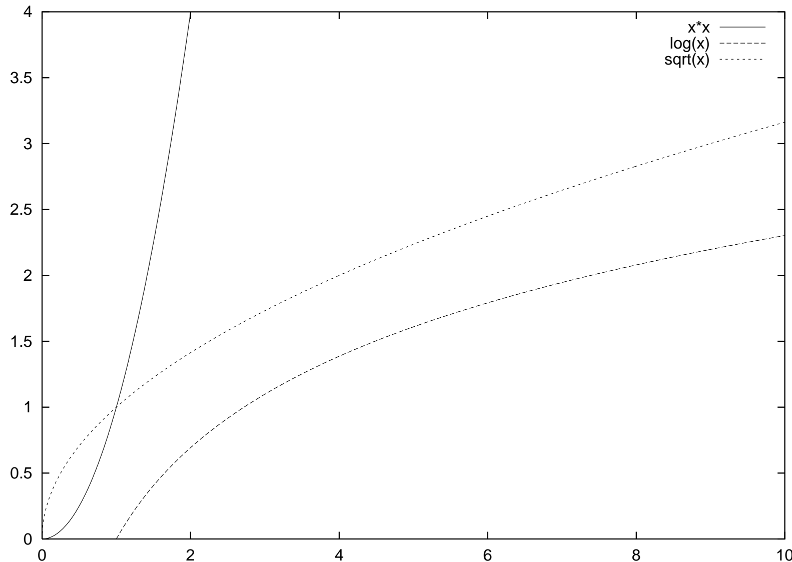


Abbildung 3.4: *Kennlinien der nichtlinearen Verzerrung im Frequenzbereich. Die variable Grösse x steht für $|S(\omega)|$. Die Quadrierung hat eine Expandierung des Spektrums zur Folge. Die Funktionen $\log x$ und $\sqrt{x}$ bewirken eine Kompression (Einebnung) des Spektrums, wobei die Wurzelfunktion eine etwas geringere komprimierende Wirkung hat als die Logarithmusfunktion.*

Einige Funktionen zur nichtlinearen Verzerrung im Frequenzbereich bewirken eine *spektrale Einebnung*. Das hat zur Folge, dass das Maximum, aus dem im Zeitbereich T_0 abgeleitet werden kann, noch prägnanter ausfällt. Andere kleinere Peaks, hervorgerufen durch einen stark ausgeprägten Formanten F1[1], werden noch kleiner. Somit ist eine zuverlässigere Bestimmung der Grundfrequenz zu erreichen.
In der Literatur existieren verschiedene Vorschläge zur nichtlinearen Verzerrung im Frequenzbereich (vgl. [1]):

- $\tilde{S}(\omega) = |S(\omega)|$: Amplitudenspektrum. Von diesem gehen die folgenden Methoden aus.
- $\tilde{S}(\omega) = |S(\omega)|^2$: Leistungsdichtespektrum. Die Rücktransformation führt auf die AKF.
- $\tilde{S}(\omega) = \log(|S(\omega)|)$: Logarithmus aus dem Amplitudenspektrum. Die Rücktransformation führt auf das Cepstrum.

[1] Die Resonanzfrequenzen des Sprachtraktes heissen Formanten. F1 ist der Formant mit der tiefsten Frequenz.

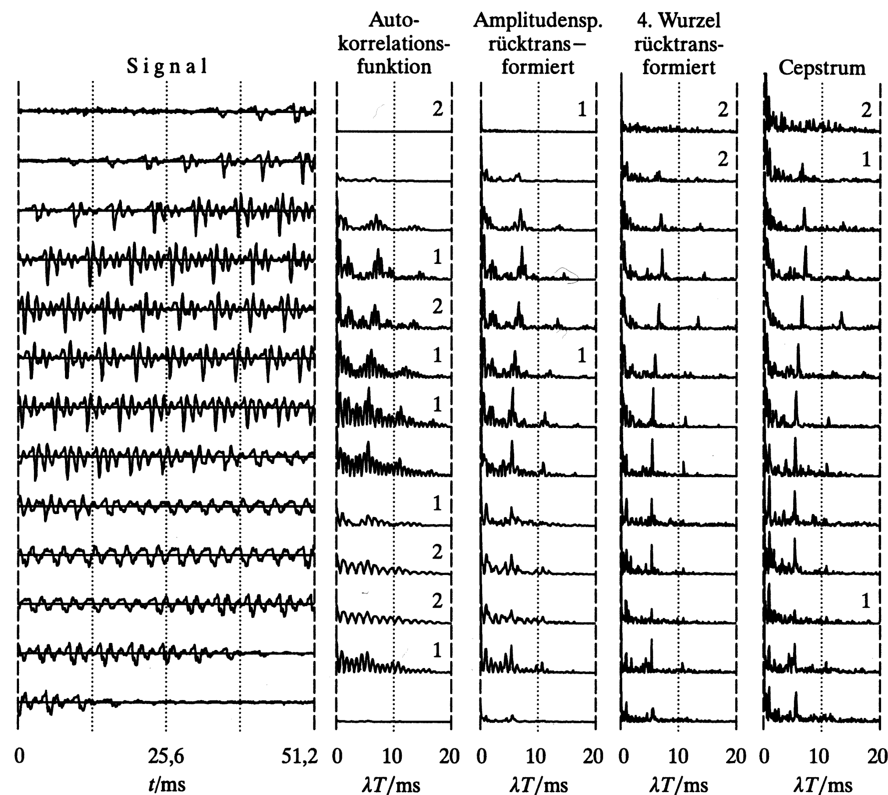


Abbildung 3.5: *GFB mit doppelter Spektraltransformation und nichtlinearer Verzerrung: einige Beispiele zur Wirkungsweise. Signal: Ausschnitt aus «Digitale Verarbeitung ...»; die Zeit läuft in Schritten von 25.6 ms von oben nach unten (50% Überlappung zwischen zwei benachbarten Frames). Gezeigt werden vier Arten der nichtlinearen spektralen Verzerrung. Die inverse Transformation des Leistungsdichtespektrums führt zur AKF, die des logarithmierten Spektrums zum Cepstrum. Die anderen Verzerrungen im Spektralbereich sind: 4. Wurzel aus dem Leistungsdichtespektrum und Amplitudenspektrum (Betragsbildung). Von den inversen Transformierten ist jeweils der Betrag abgebildet. Die Zahlen auf der rechten Seite eines Teiles der inversen Transformierten geben an, wieviele parasitäre Maxima (mehr als 70% der Amplitude des Maximums bei T_0) innerhalb des Messbereiches (1-20 ms) entdeckt wurden. (Quelle: [1])*

- $\tilde{S}(\omega) = \sqrt[4]{|S(\omega)|^2} = \sqrt[2]{|S(\omega)|}$: 4. Wurzel aus dem Leistungsdichtespektrum, oder 2. Wurzel aus dem Amplitudenspektrum. Die Rücktransformation führt auf etwas, das zwischen AKF und Cepstrum liegt. Das Spektrum wird wie beim der Cepstrum-Methode eingeebnet, jedoch etwas weniger stark.

Alle diese Verfahren haben ihre Vor- und Nachteile. Das Cepstrum-Verfahren ist gegenüber dominanten Formanten relativ unempfindlich, weist jedoch eine gewisse Empfindlichkeit gegenüber verrauschten Signalen auf. Die AKF wiederum ist bei der GFB gegen Rauschen sehr unempfindlich, gegenüber dominanten Formanten jedoch recht empfindlich. Betrachtet man die Krümmung der Kennlinie (siehe Abbildung 3.4) der nichtlinearen Verzerrung des Spektrums, so wird der Dynamikbereich bei Verwendung der quadratischen Kennlinie expandiert (also nicht eingeebnet); bei der Verwendung der logarithmischen Kennlinie erfolgt eine Kompression der Amplitude und

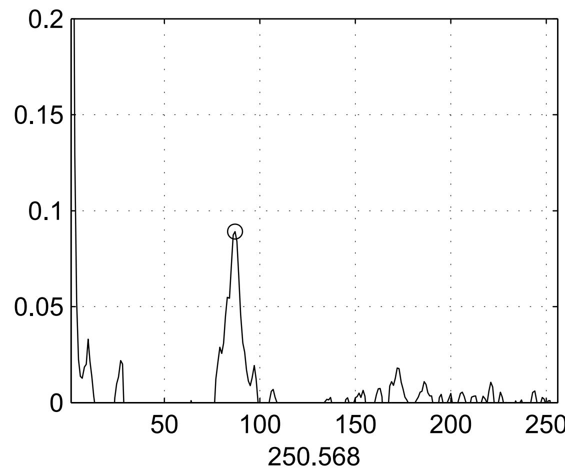


Abbildung 3.6: Bestimmung der Pitch nach der Vorverarbeitungsstufe (doppelte Spektraltransformation nach der Methode 4. Wurzel aus dem PDS). Es wurden nur die nur Werte ab Index 50 zur Bestimmung des signifikanten Maximums herangezogen. Dieses ist durch einen Kreis markiert. Die bestimmte Pitch ist 251 Hz.

damit eine spektrale Einebnung. Offensichtlich ist die spektrale Einebnung wichtig für die GFB. Sie darf aber auch nicht zu stark erfolgen (Gründe dazu siehe [1]). Daher liegt es nahe, eine Verzerrungsfunktion zu finden, die das Spektrum etwas weniger einebnet. Aus Abbildung 3.4 ist ersichtlich, das dies mit der Funktion 4. Wurzel aus dem Leistungsdichtespektrum gegeben ist. Von dieser Kennlinie wird eine verbesserte Robustheit des Algorithmus bei verrauschten Signalen erwartet, wobei die Störanfälligkeit bei dominanten Formanten nicht merklich steigen soll (vgl. [1]). Abbildung 3.5 zeigt ein Beispiel.

3.1.4 Bestimmung der Pitch nach einer doppelten Spektraltransformation

Die Extraktionsstufe hat die Aufgabe aus dem von der Vorverarbeitungsstufe bearbeiteten Frame die Pitch zu bestimmen. Dieses lässt sich nach den genannten Methoden aus einem signifikanten Maximum ableiten. Die Vorverarbeitungsstufe führt die doppelte Spektraltransformation mit anschliessender Beschneidung der entstandenen Zeitfunktion durch (vgl. Abbildung 3.2), da beispielsweise bei der AKF sich das signifikanteste Maximum bei der Verschiebung 0 befindet. Es muss also vorher ein bestimmter Teil abgeschnitten werden. Gleich verhält es sich mit dem anderen Methoden. In Abbildung 3.6 wurden nur Werte ab Index 50 zur Bestimmung des signifikanten Maximums herangezogen.

Die Pitch p berechnet sich nun folgendermassen:

$$p = \frac{f_s}{i} \qquad p : \text{Pitch} \quad f_s : \text{Samplingfrequenz} \quad i : \text{Index der Position des Maximums}$$

3.2 Verwendete Methoden

In dieser Studienarbeit wurden folgende Methoden verwendet: Autokorrelation, 4. Wurzel aus dem Leistungsdichtespektrum und Cepstrum. Diese Verfahren wurden ausgewählt da sie

- einfach zu implementieren sind

- prinzipiell gleich arbeiten (sie unterscheiden sich nur durch die Funktion der nichtlinearen Verzerrung im Frequenzbereich)
- gute Ergebnisse zu liefern versprechen

Im nächsten Kapitel wird genauer auf diese Methoden eingegangen und anhand von Beispielen die Wirkungsweise demonstriert.

3.3 Bestimmung der Anregungsart

Grundsätzlich gilt es auch die Anregungsart zu bestimmen. D.h. herauszufinden, ob eine stimmhafte oder eine stimmlose Anregung vorliegt. Ist keine der beiden Anregungen vorhanden, so spricht man von einem Pausensegment. In der Simulation beschränkt man sich auf eine Unterscheidung zwischen stimmhafter Anregung oder nicht stimmhafter Anregung. Bei der Integration der Pitch-Extraktion in das bestehende Vocodersystem stellt sich dieses Problem gar nicht, da die Bestimmung der Anregungsart dort bereits implementiert ist.

4 Entwicklung und Resultate

Um erst Klarheit über die Wirkungsweise und Robustheit der verschiedenen Methoden zu erlangen wurden der Implementation im DSP vorausgehend einige Simulationen durchgeführt. Es ging dabei auch darum herauszufinden, welche Methode geeignet ist und welche weniger. Die Simulationen wurden in einer ersten Phase mit drei verschiedenen bekannten Verfahren durchgeführt, um die Eignung für einen Einsatz im Vocodersystem zu eruieren. In einer zweiten Phase wurden diese in das bestehende Echtzeit-Vocodersystem (DSP) integriert.

4.1 Simulation in MATLAB

Es wurden zwei verschiedene MATLAB-Programme (M-Files) erstellt. Die Sourcen dieser Programme sind im Anhang zu finden.

4.1.1 Frame-by-Frame-Vergleich der Pitch-Extraction-Methoden

Das erste Programm[1] liest ein Microsoft PCM Wavefile Stück für Stück mit 50% Überlappung ein[2], führt die doppelte Spektraltransformation mit nichtlinearer Verzerrung nach den genannten drei Methoden im Frequenzbereich durch, stellt das Rücktransformierte und das Waveframe dar und extrahiert die Pitch durch Suchen des signifikanten Maximums. Dann wartet das Programm auf einen Tastendruck, der das Einlesen des nächsten Frames auslöst. Damit lassen sich die Methoden Frame für Frame miteinander im Detail vergleichen (vgl. Abbildung 4.1).

4.1.2 Vergleich der Pitch-Extraction-Methoden über ein ganzes Wavefile

Auch das zweite Programm[3] liest ein Microsoft PCM Wavefile nach der gleich Art ein und bestimmt die Pitch nach dem gleichen Verfahren mit den verschiedenen Methoden wie das erste Programm. Dieses stellt jedoch die pro Frame extrahierten Pitches als Pitchverlauf über die ganze Wavelänge dar. Damit lassen sie die verschiedenen Methoden über die gesamte Länge vergleichen. So können die Stärken und Schwächen der verschiedenen Methoden bei verschiedenen Wavefiles studiert werden (vgl. Abbildung 4.2).

Die Entscheidung *stimmhaft/stimmlos* wurde dabei für alle Methoden mittels der AKF getroffen, da stimmlose Laute einem Rauschen sehr ähnlich sind und die AKF davon erwartungsgemäss klein ist: Falls das detektierte Maximum der AKF-Methode einen gewissen Wert unterschreitet, so wird das Frame als *stimmlos* interpretiert, ansonsten als *stimmhaft*. Dies ist zugegebenermassen eine einfach Methode, doch für diese Zwecke reicht sie aus. Es geht nur um eine grobe Unterscheidung und Darstellung des stimmlosen und stimmhaften Bereiche. Sie funktioniert zudem recht zuverlässig. Die besagte Schwelle kann im MATLAB-Programm eingestellt werden mittels der Variable `thresh`. Diese war für alle Versuche gleich eins gesetzt.

[1] File: `ROOT/sim/method_compare_frame.m`
[2] 50% des neuen Frames stammen noch vom alten Frame
[3] File: `ROOT/sim/method_compare_whole.m`

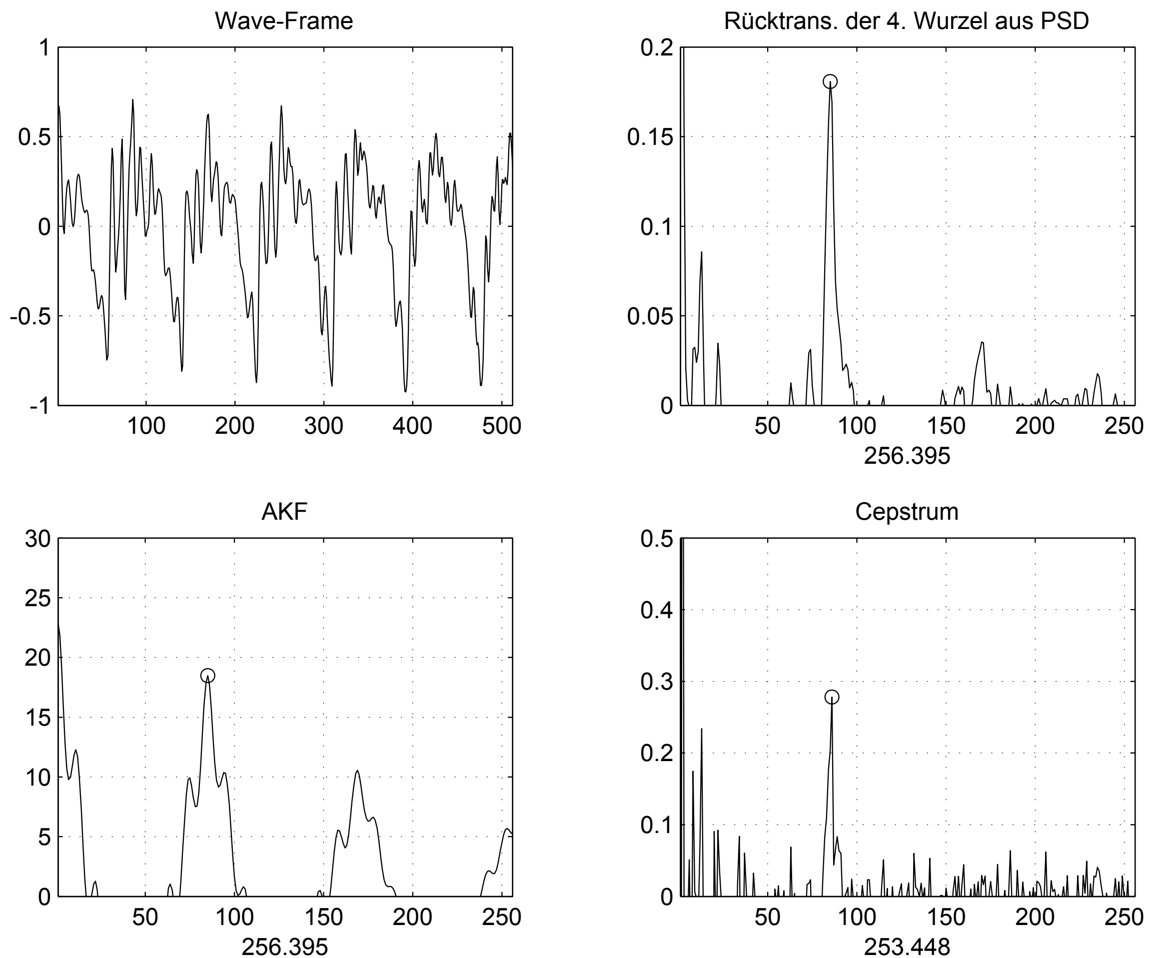


Abbildung 4.1: *Frame-by-Frame-Vergleich der Pitch-Extraction-Methoden: Der kleine Kreis markiert das erkannte Maximum. Die Zahl unter dem Diagramm ist die berechnete Pitchfrequenz.*

4.1.3 Framelänge

Bei beiden Programmen kann die Framelänge festgelegt werden (Konstante im Header des MATLAB-Programms). Die Framelänge beträgt im Hinblick auf die Implementation in den DSP standardmässig 512 Werte. Der Vocoder verwendet Frames der Länge 256 bei einer Samplingfrequenz von 22050 Hz. Daraus ergibt sich eine Framedauer von 11.6 ms. Um einen vernünftigen Bereich der Pitch erfassen zu können ist aber mindestens die doppelte Länge zwingend, also 512 Werte oder 23.2 ms. Dies ergibt einen theoretischen Pitchbereich von $^{22050}/_{256} = 86\,\text{Hz}$[4] bis $^{22050}/_{50} = 441\,\text{Hz}$. Die untere Grenze kann aber praktisch nicht erreicht werden, da die Werte gegen Ende des Frames gegen Null gehen. Daher kann ein signifikantes Maximum in diesem Bereich nur schwer mehr festgestellt werden.

Eine Pitch bis zu einer unteren Grenze von ca. 115 Hz ist erfahrungsgemäss noch relativ zuverlässig extrahierbar. Tiefe Männerstimmen erreichen aber eine Pitch, die deutlich unter dieser

[4]Von den 512 Werten können nur die Hälfte für die Pitchextraktion verwendet werden, da es sich nach der doppelten spektralen Transformation um ein bezüglich der Mitte des Frames spiegelsymetrisches Zeitsignal handelt (analog zu einer Periode im Frequenzbereich).

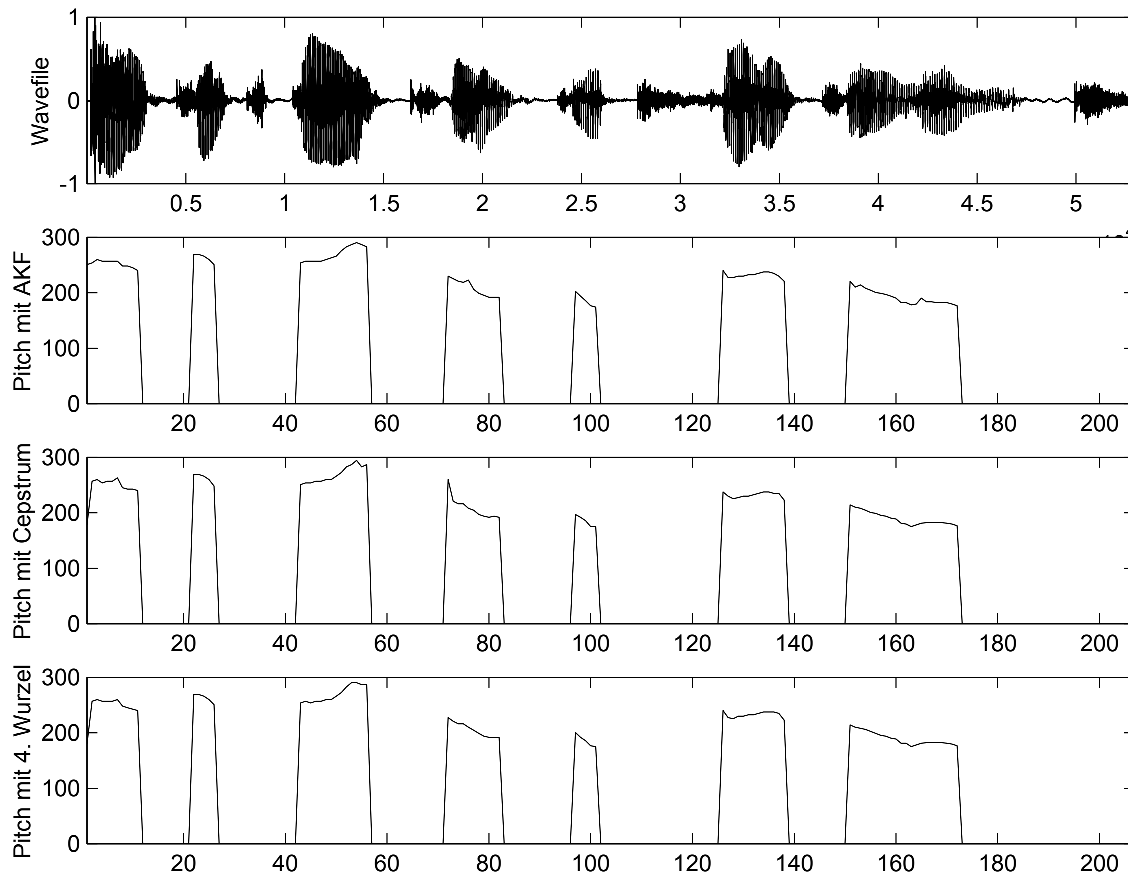


Abbildung 4.2: *Vergleich der Pitch-Extraction-Methoden über ein ganzes Wavefile: Die Pitchverläufe für die verschiedenen Methoden sind übereinander dargestellt.*

Grenze liegen kann[5]. Dieses Problem liesse sich nur lösen, indem das Frame grösser gewählt wird. In der Simulation ist dies auch möglich, doch implementiert man diese Framegrösse auf dem in dieser Studienarbeit verwendeten DSP, so reicht dessen Rechenkapazität nicht mehr aus (vgl. Abschnitt 4.2). Somit müssen einige Einbussen bezüglich des erkennbaren Pitchbereiches in Kauf genommen werden.

4.1.4 Beispiele von extrahierten Pitchverläufen

Einige gesammelte Wavefiles mit Sprecherstimmen dienten als Testfiles um die Stärken und Schwächen der verschiedenen Methoden auszuloten, und um herauszufinden welche am zuverlässigsten arbeitet. Diese wurden vorgängig folgendermassen bearbeitet: Der DC-Anteil wurde entfernt, und die Waves wurden normalisiert, so dass alle in etwa die gleiche Lautstärke besitzen. Dies ist nötig, da ja zur Entscheidung *stimmhaft/stimmlos* eine feste Schwelle benutzt wird.

[5] Die tiefsten Frequenzen liegen bei ca. 80 Hz.

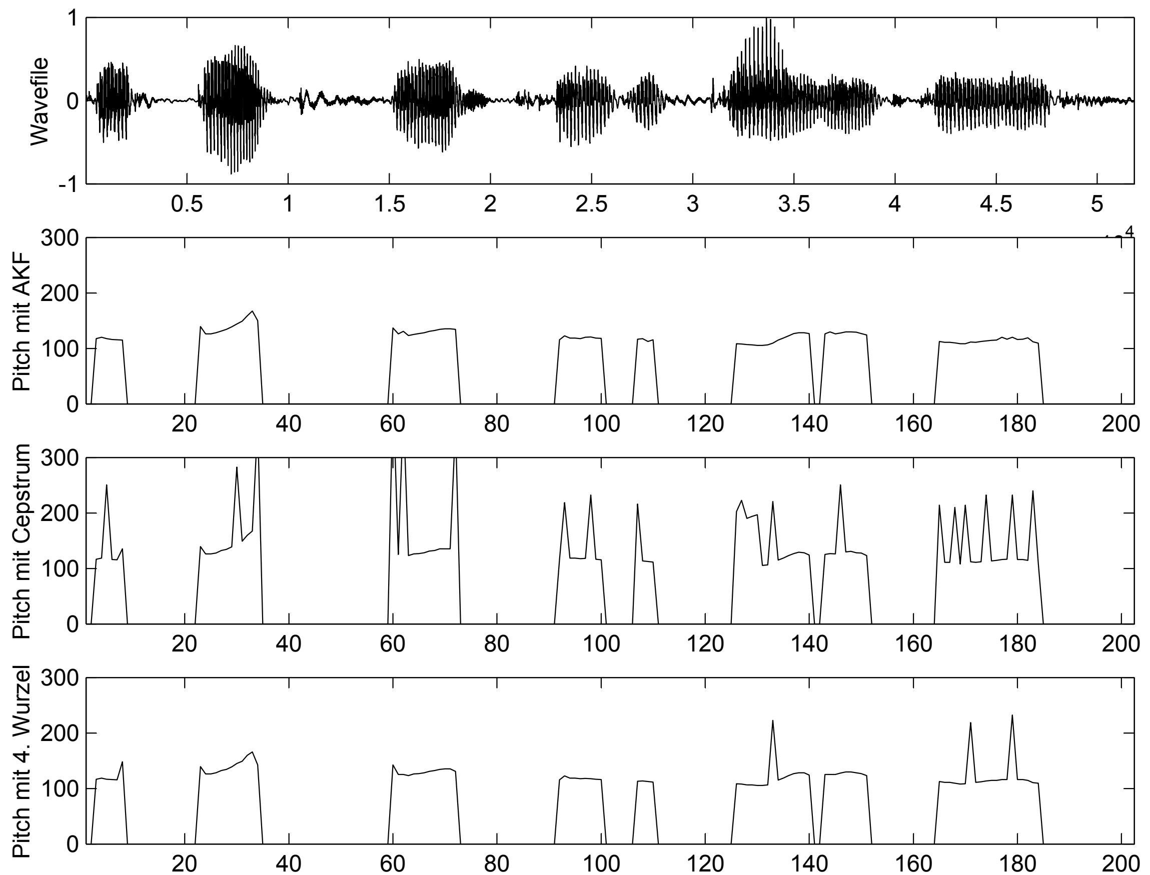


Abbildung 4.3: *Tiefe Männerstimme. File:* `test1.wav`

Diese tiefe Männerstimme hat eine Pitch die nahe an der unteren Grenze der noch erkennbaren Pitch liegt. Die Cepstrum-Methode hat sichtlich Mühe die richtige Pitch zu extrahieren. Einzig die AKF-Methode macht keine Fehler, während die 4. Wurzel-Methode dreimal zu der doppelten Frequenz springt.

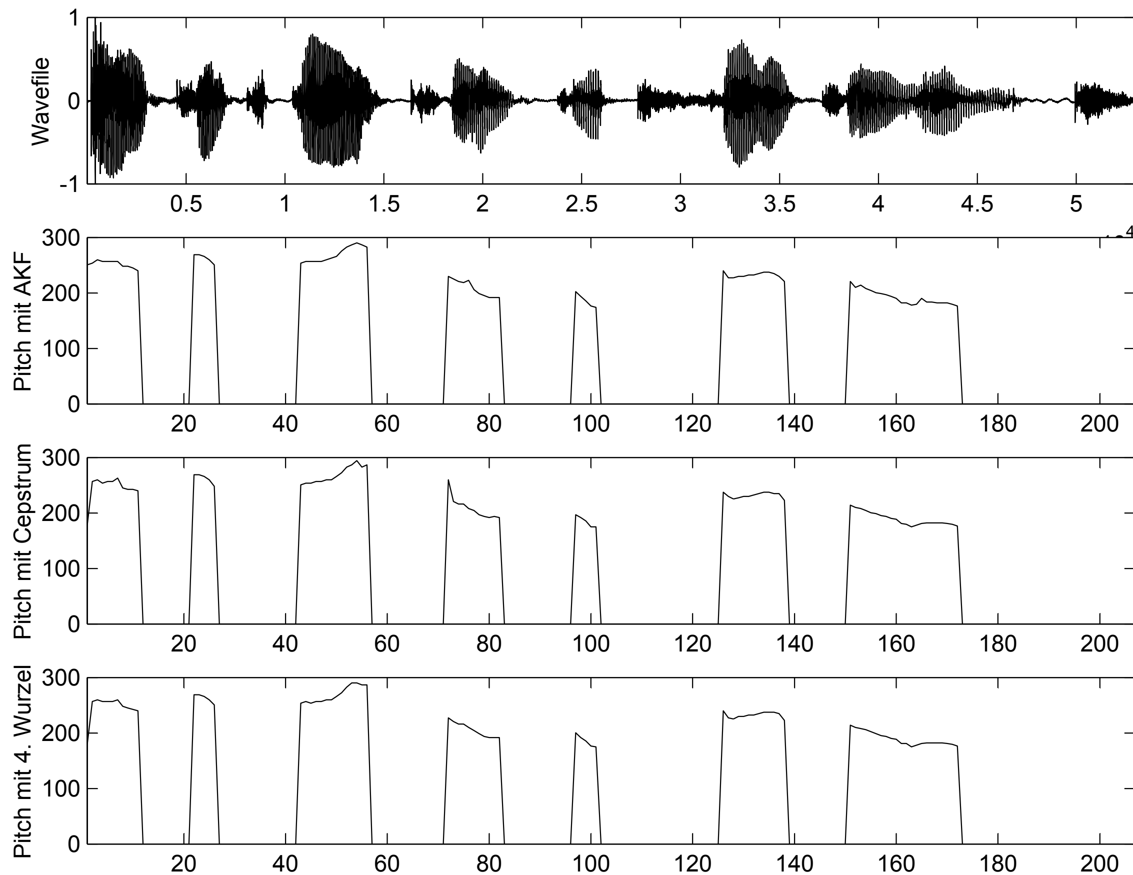


Abbildung 4.4: *Problemlose Frauenstimme. File:* `test2.wav`

Diese Abbildung entspricht der Abbildung 4.2. Es handelt sich dabei um eine Frauenstimme, die naturgemäss eine höhere Pitch besitzt. Erwartungsgemäss kann diese von allen Methoden auch bestens extrahiert werden. Die Pitchverläufe der einzelnen Methoden zeigen praktisch keine Unterschiede.

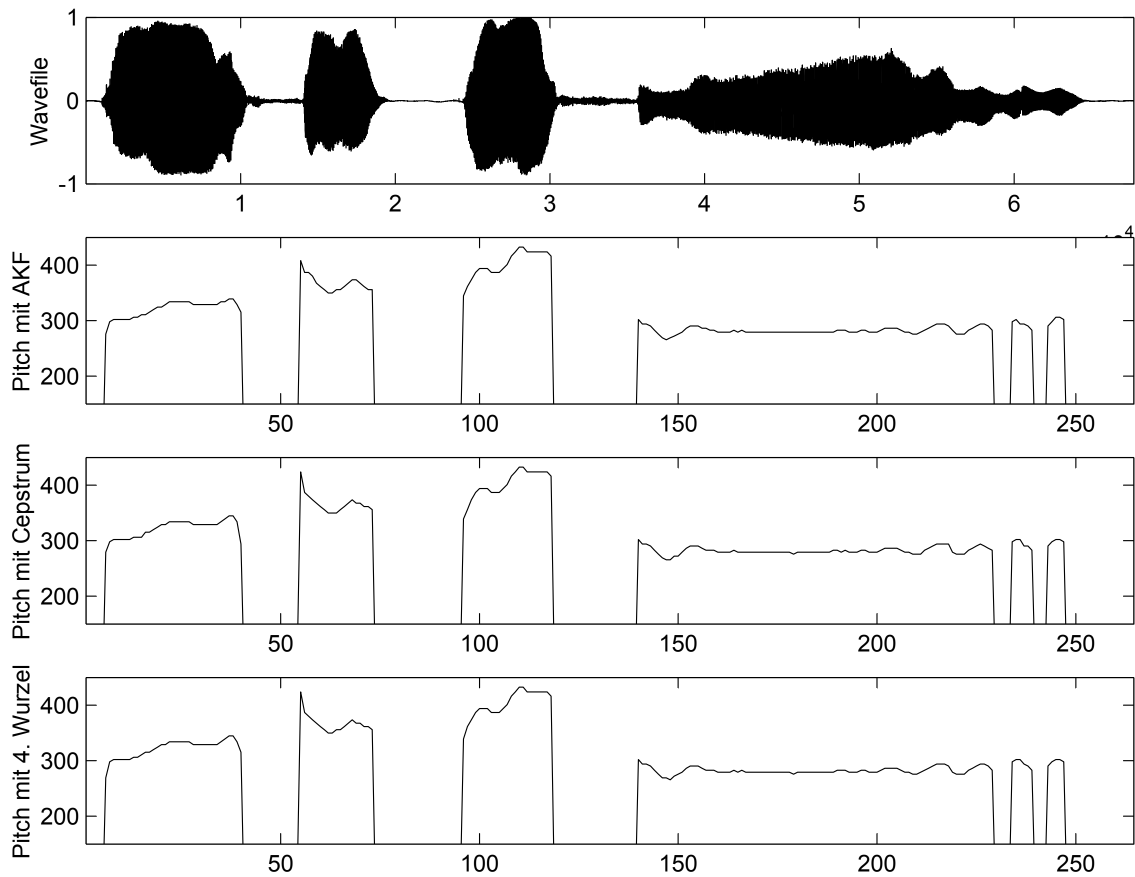


Abbildung 4.5: *Hoher Gesang. File:* `test3.wav`

Auch bei diesem Beispiel handelt es sich um eine Frauenstimme, diesmal aber um Gesang. Wieder sind alle drei Methoden in der Lage die Pitch sauber zu extrahieren. Sehr schön zu sehen ist auch das leichte Tremolo der Stimme am Ende der Sequenz.

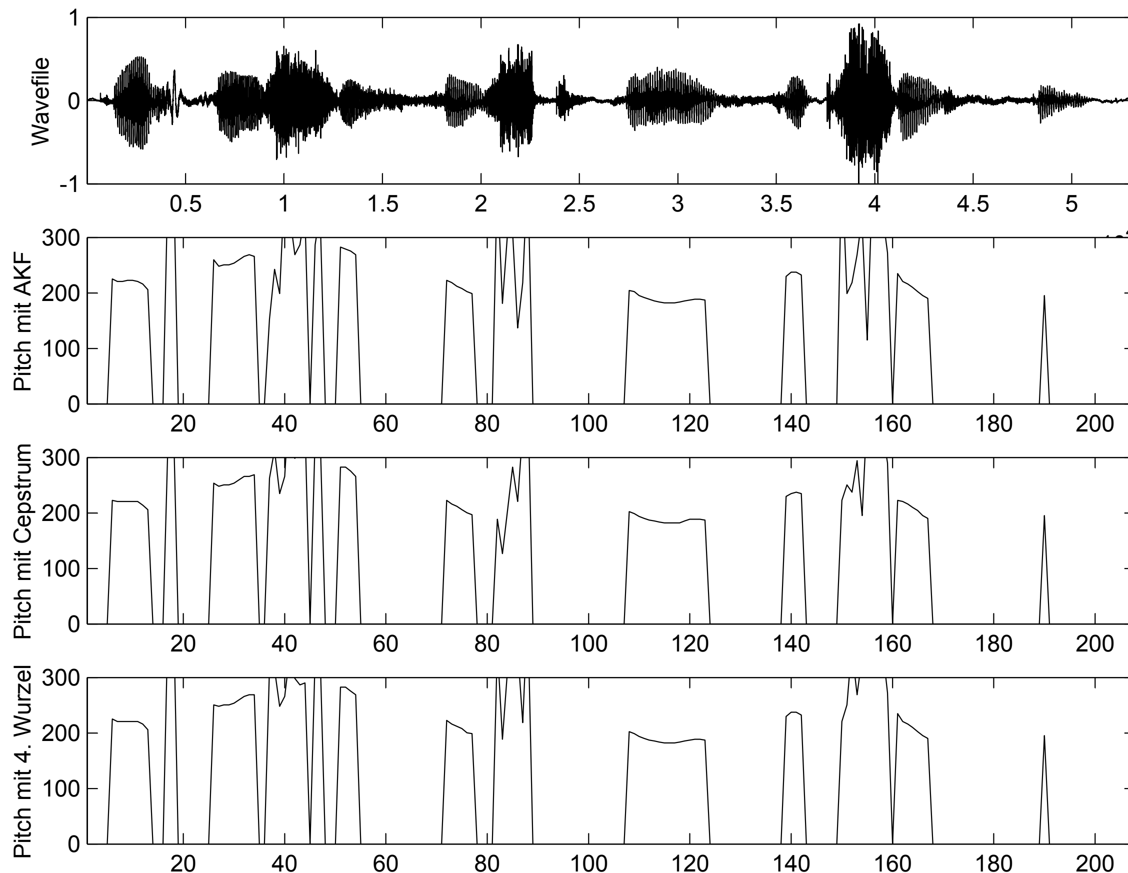


Abbildung 4.6: *Frauenstimme mit lauten Zischlauten. File:* `test4.wav`

Diese Sequenz bereitet allen drei Methoden scheinbar Mühe, da Teilweise sehr schnelle Wechsel der Pitch von sehr tief nach sehr hoch angezeigt werden, doch dabei handelt es sich um ein anderes Problem: Die erwähnte Entscheidung stimmhaft/stimmlos basiert auf der AKF und dem davon detektierten Maximalwert. In diesem Beispiel haben die Zischlaute, die als stimmlos detektiert werden sollten einen derart hohen Pegel, so dass der detektierte Maximalwert der AKF grösser als die Schwelle `tresh` wird. Dies lässt sich mit dem Programm `method_compare_frame.m` überprüfen. Daher sind diese Teile nicht als falsch extrahierte Pitch sondern als versagen der Entscheidung stimmhaft/stimmlos zu werten. In anderen Teilen wir die Pitch von allen drei Methoden durchaus zuverlässig extrahiert.

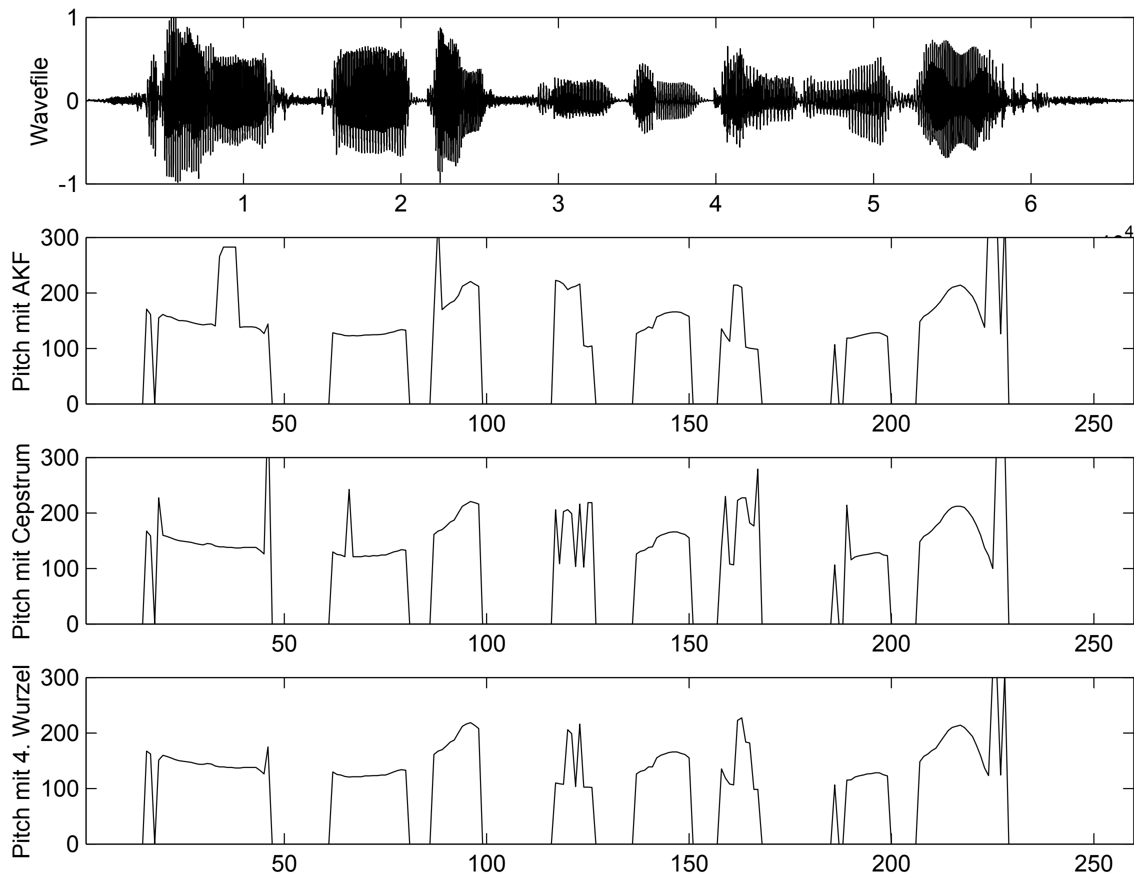


Abbildung 4.7: *Männerstimme: AKF Probleme mit F1 und tiefen Pitchfrequenzen. File:* `test5.wav`

Dieses Beispiel bringt gleich zwei Probleme zu Tage: Die AKF hat Mühe dem Formanten F1 (besonders deutlich im Bereich der ersten 50 Frames zu erkennen), denn es handelt sich um eine Männerstimme. Daneben haben alle Methoden Mühe mit den zum Teil recht tieffrequenten Pitches vor allem im Bereich von Frame 100 bis Frame 130. Am besten schneidet hier die 4. Wurzel-Methode ab.

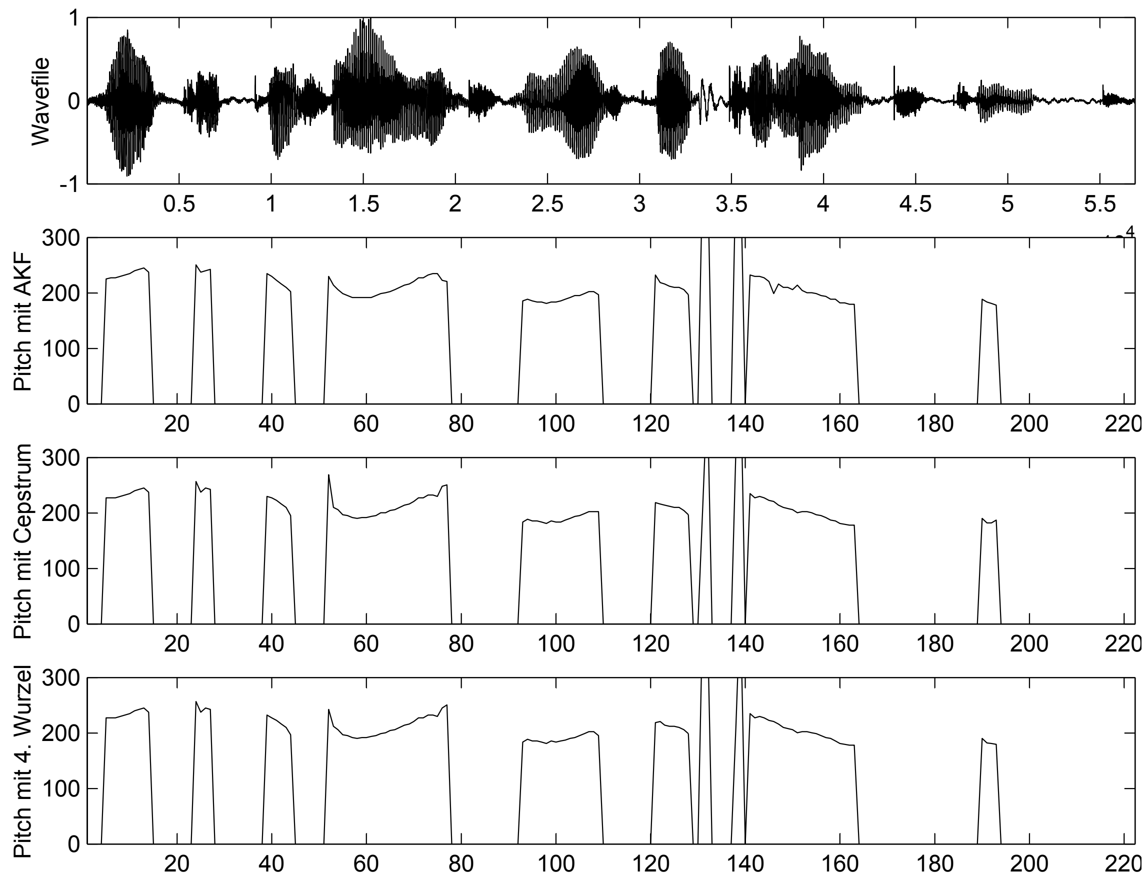


Abbildung 4.8: *Entscheidungsproblem stimmhaft/stimmlos bei einem starken Konsonanten «p». File:* `test6.wav`

Alle Methoden extrahieren die Pitch zuverlässig, denn es handelt sich um eine Frauenstimme. Das einzige Problem tritt wiederum bei der Entscheidung stimmhaft/stimmlos auf. Der starke Konsonant «p» lässt ein Maximum der AKF grösser als `tresh` entstehen.

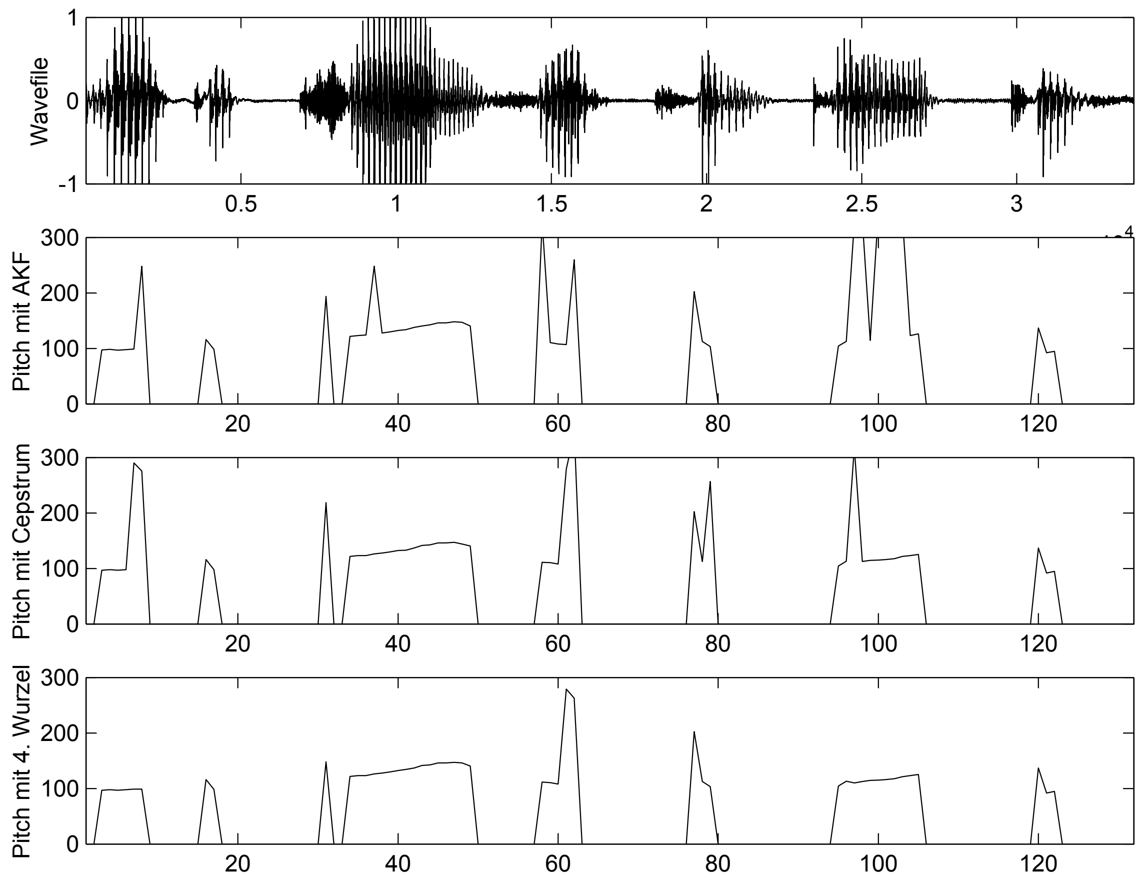


Abbildung 4.9: *Tiefe verzerrte Männerstimme. File:* `test7.wav`

Diese Stimme ist leicht verzerrt (übersteuert) und relativ tief. Hier offenbart sich eine Schwäche der AKF-Methode gegenüber den anderen Methoden mit spektraler Einebnung. Von diesen beiden ist die 4. Wurzel-Methode klar die beste. Die Fehler, die diese Methode macht, sind eher auf die zu tiefe Pitch zurückzuführen als auf die Verzerrung.

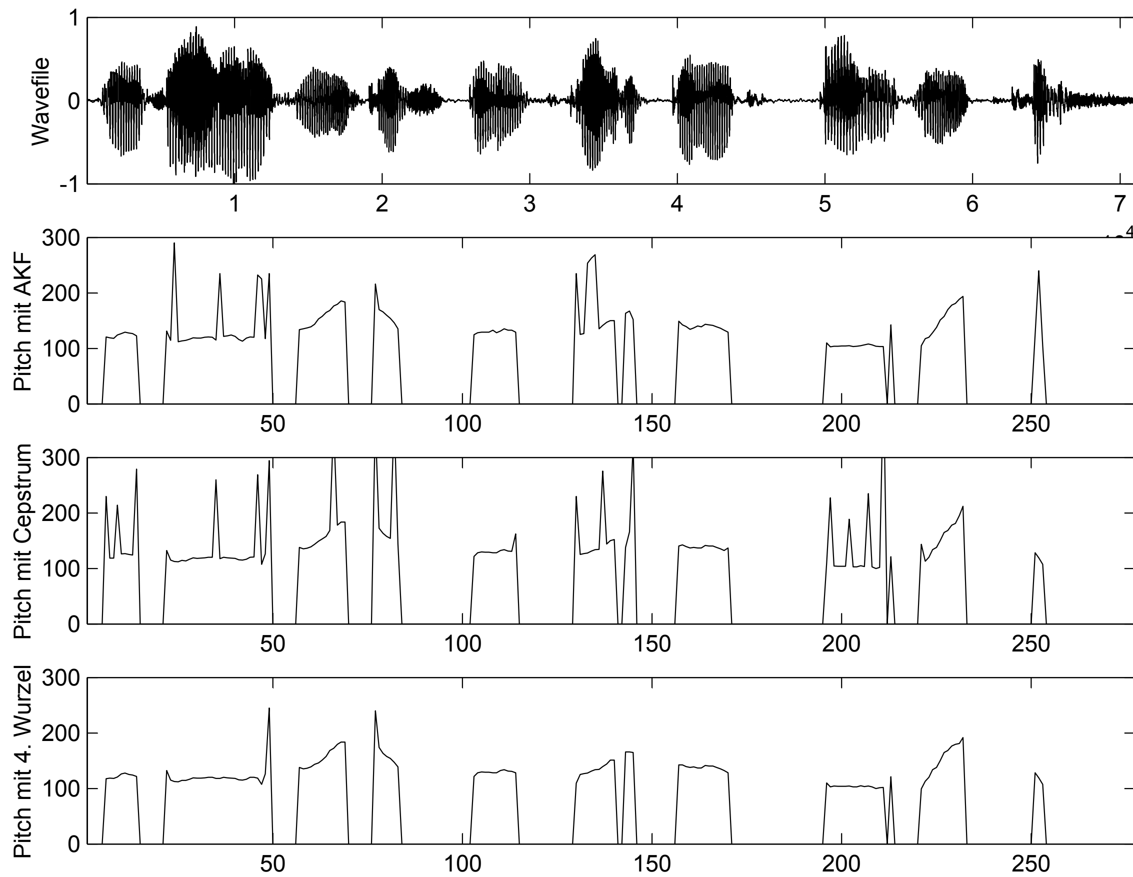


Abbildung 4.10: *Mittlere Männerstimme. File:* `test8.wav`

Hier haben die AKF-Methode und die Cepstrum-Methode grosse Mühe die richtige Pitch zu extrahieren. Die 4. Wurzel-Methode liefert klar das beste Ergebnis. Die Probleme sind auf einen ausgeprägten Formanten und auf die recht tiefe Pitchfrequenz zurückzuführen.

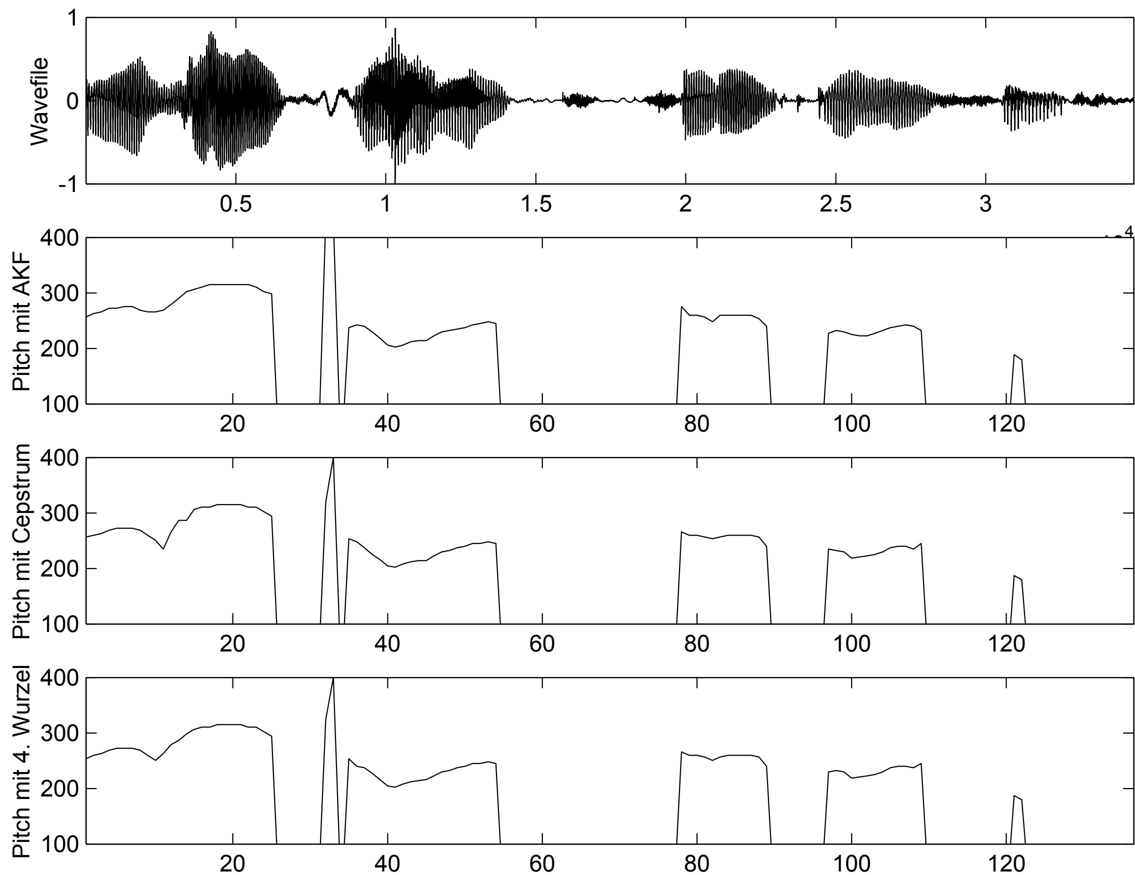


Abbildung 4.11: *Hohe Frauenstimme. File:* `test9.wav`

Die hohe Frauenstimme macht keinem der Methoden Probleme. Einzig die Entscheidung stimmhaft/stimmlos entscheidet sich bei Konsonanten «h» falsch, so dass die Algorithmen an dieser Stelle eine nicht zu bewertende Pitch extrahieren.

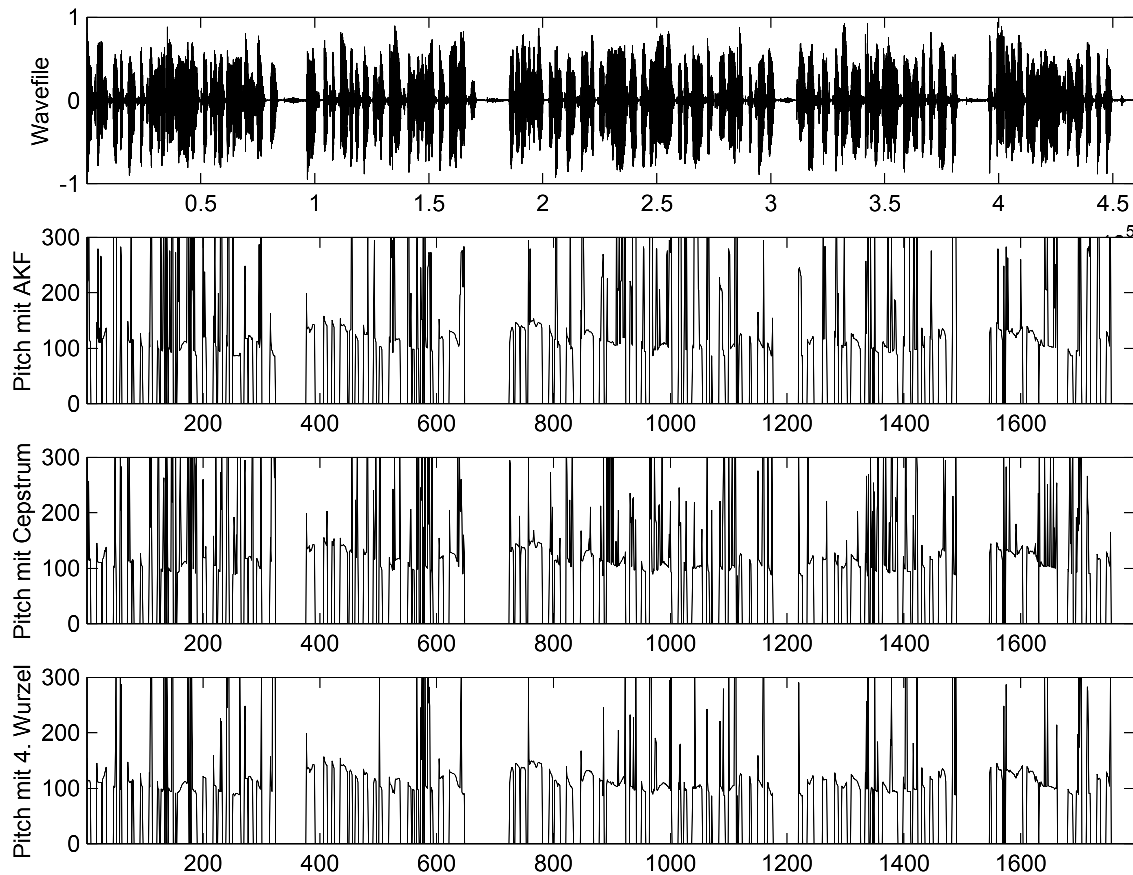


Abbildung 4.12: *Männlicher Nachrichtensprecher. File:* `test0.wav`

Diese Sequenz ist um einiges länger als die vorangegangenen. Es handelt sich dabei um einen Nachrichtensprecher, der mit sehr tiefer Stimme spricht. Erwartungsgemäss haben alle Methoden Mühe die Pitch richtig zu extrahieren. Dazu kommen sehr ausgeprägte Formanten, die zusammen mit der tiefen Stimme die Pitchbestimmung ausserordentlich schwierig machen. Es kommt daher zu vielen Fehlern. Die 4. Wurzel-Methode schneidet noch am besten ab. Die folgende Abbildung 4.13 zeigt einen Ausschnitt dieser Sequenz (erste 390 Frames).

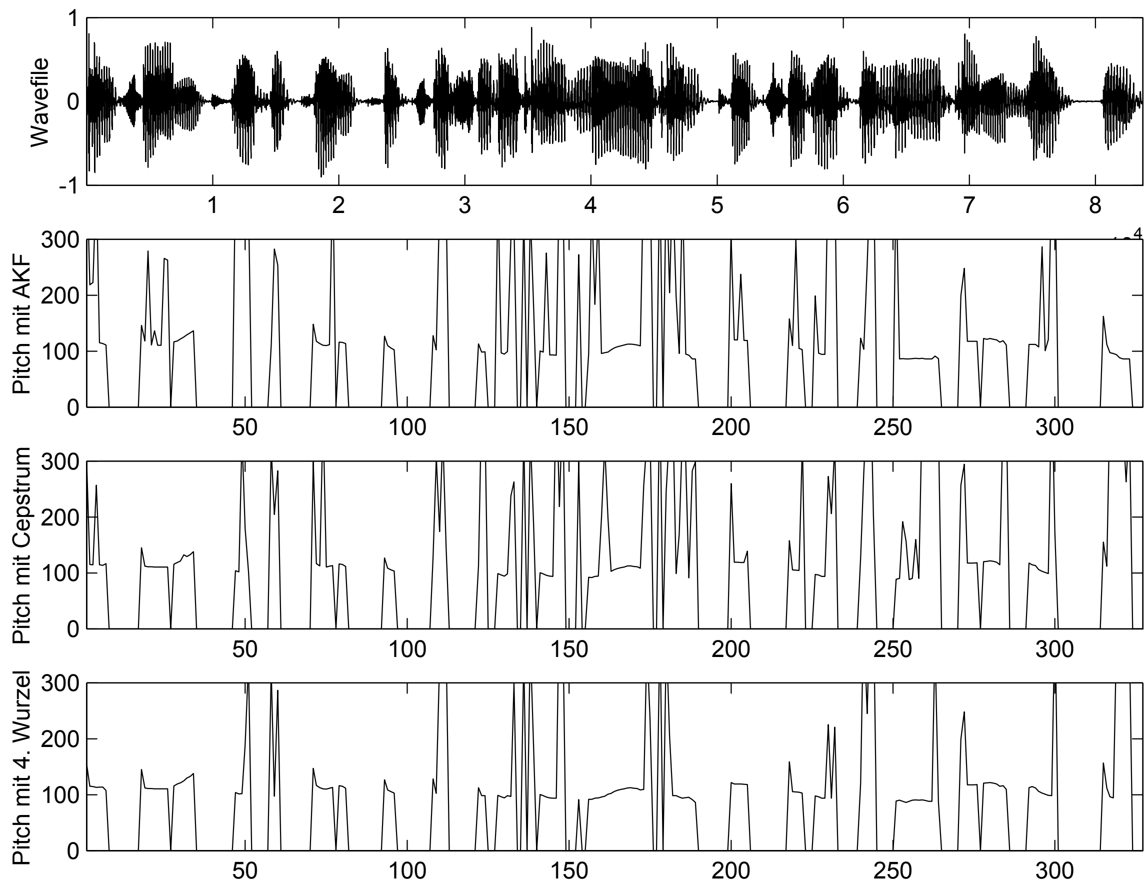


Abbildung 4.13: *Ausschnitt männlicher Nachrichtensprecher. File:* `test0_short.wav`

Diese Sequenz stellt die härtesten Anforderungen an die Pitchextraktion. Der Sprecher spricht schnell, tief und mit ausgeprägten Formanten. Wie bereits erwähnt, arbeitet die 4. Wurzel-Methode noch am zuverlässigsten. Ob sich das auch im Hörtest betätigt, wird sich bei der Versuchen mit dem DSP-Vocoder zeigen.

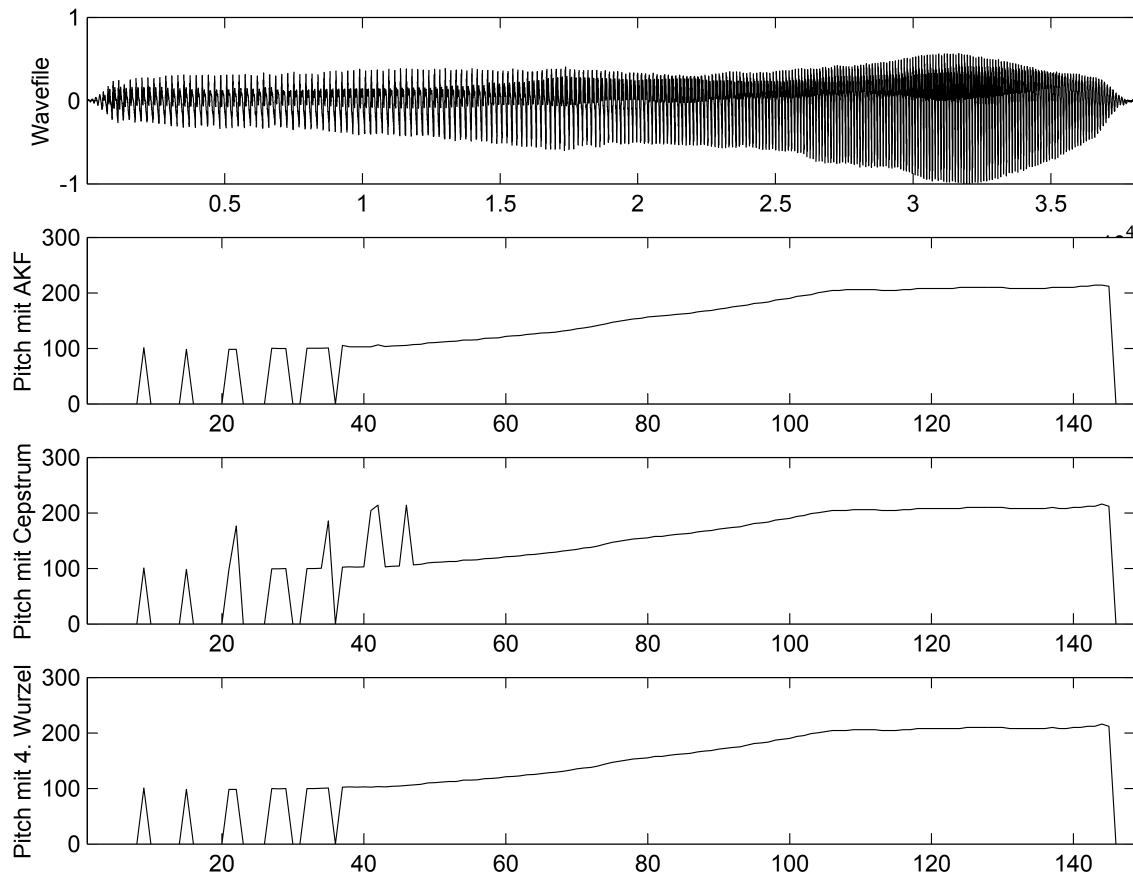


Abbildung 4.14: *Vokal «e» mit steigender Pitch. File:* `e.wav`

Die letzten vier Beispiele verwenden die Vokale «e», «i», «o» und «u», die mit steigender Pitch aufgezeichnet wurden. Man kann sehr schön beobachten, dass vor allem die Cepstrum-Methode bei tiefen Pitchfrequenzen grosse Schwierigkeiten hat die richtige Pitch zu finden, währenddessen die höheren Pitches von allen Methoden zuverlässig und genau extrahiert werden.

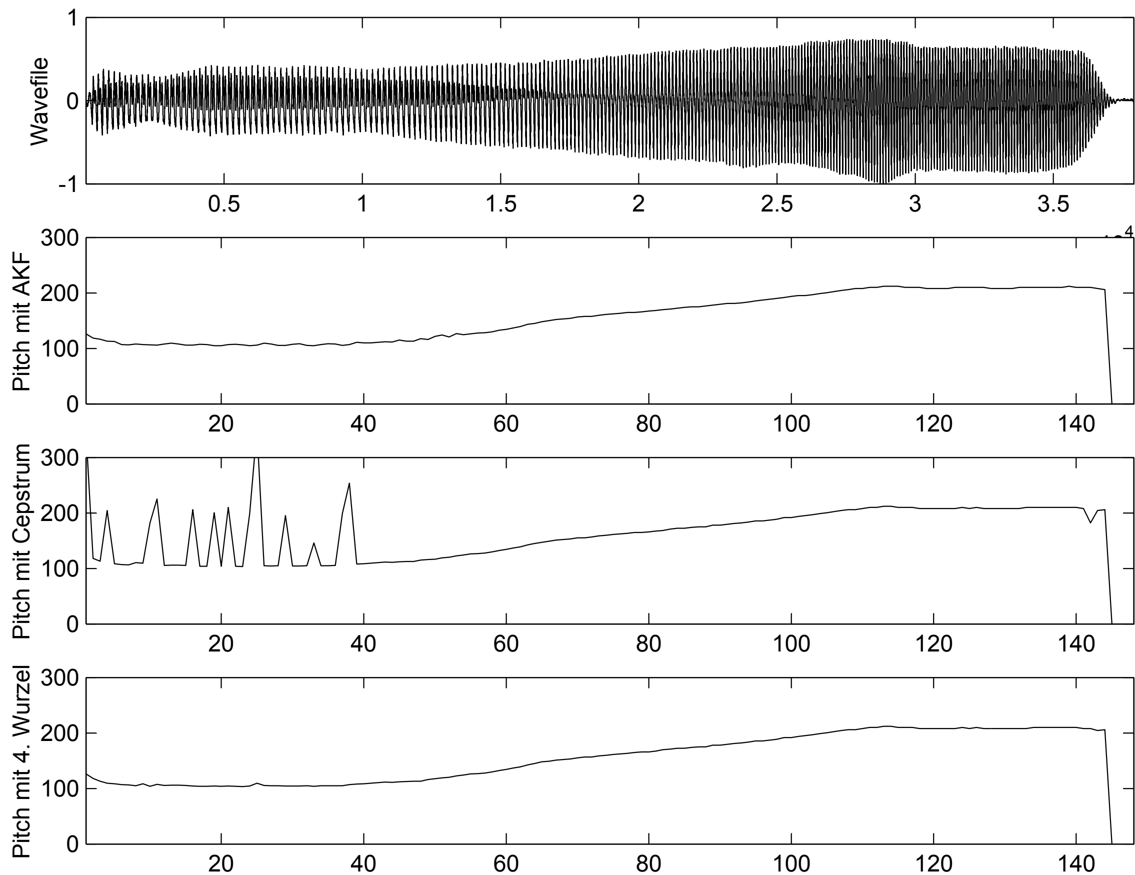


Abbildung 4.15: *Vokal «i» mit steigender Pitch. File:* `i.wav`

Die Cepstrum-Methode hat sichtlich Mühe mit tiefen Pitchfrequenzen, währenddessen die anderen beiden Methoden die Pitch zuverlässig extrahieren.

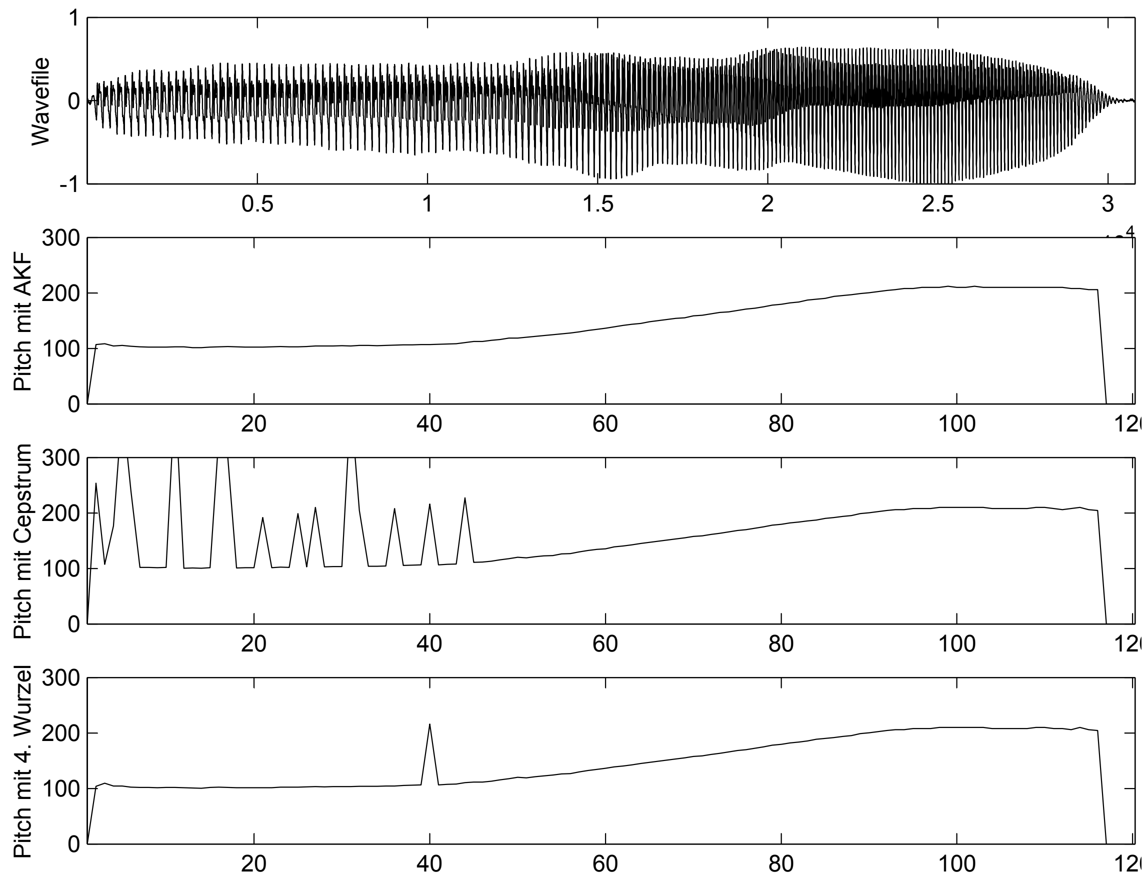


Abbildung 4.16: *Vokal «o» mit steigender Pitch. File:* `o.wav`

Auch hier hat die Cepstrum-Methode sichtlich Mühe mit tiefen Pitchfrequenzen, währenddessen die anderen beiden Methoden die Pitch zuverlässig extrahieren.

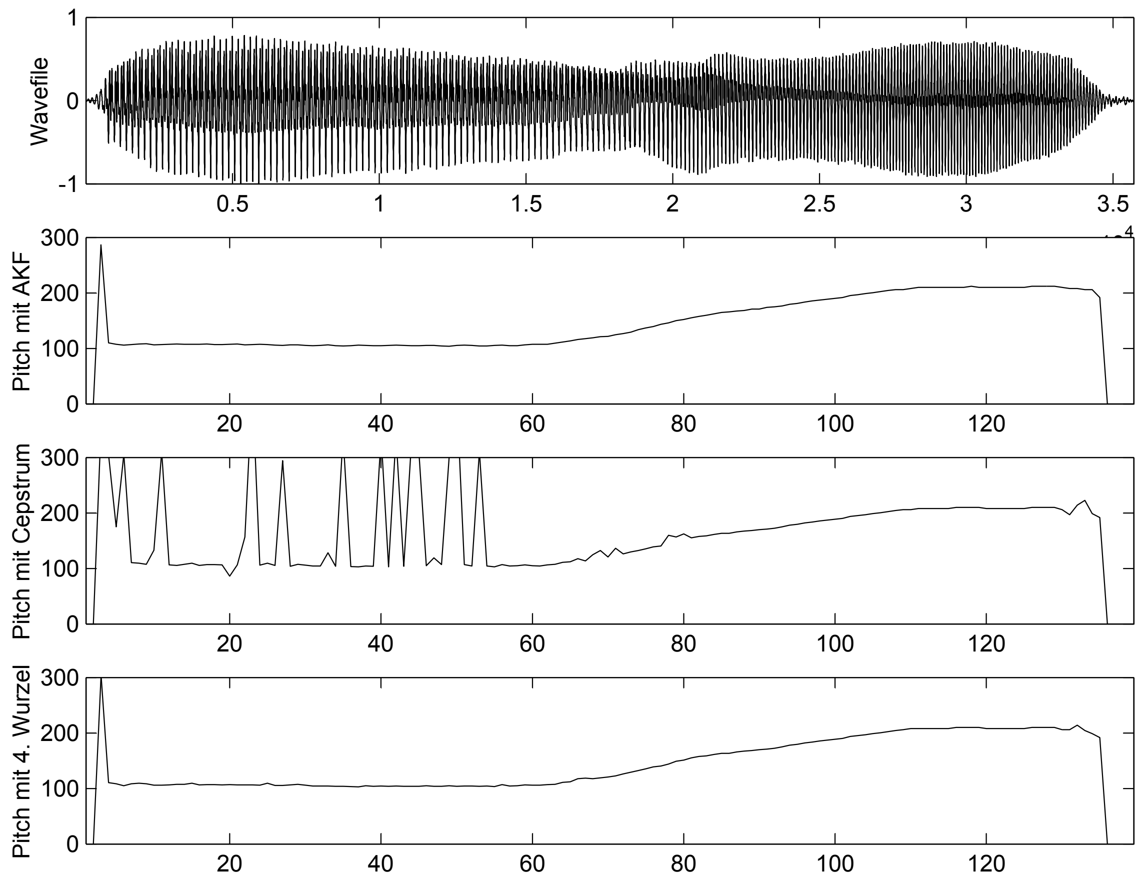


Abbildung 4.17: *Vokal «u» mit steigender Pitch. File:* `u.wav`

Und auch hier hat die Cepstrum-Methode hat sichtlich Mühe mit tiefen Pitchfrequenzen, währenddessen die anderen beiden Methoden die Pitch zuverlässig extrahieren.

4.1.5 Schlussfolgerungen

Aus den Simulationen lassen sich folgende Schlussfolgerungen ziehen:

- Hohe Pitchfrequenzen, wie sie in Frauenstimmen typischerweise zu finden sind, werden von allen Methoden recht zuverlässig extrahiert.
- Erwartungsgemäss werden tiefe Pitchfrequenzen unter ca. 115 Hz nur selten richtig extrahiert, am ehesten mit der 4. Wurzel-Methode.
- Die AKF-Methode reagiert wesentlich empfindlicher auf ausgeprägte Formanten als die anderen beiden Methoden. Diese Feststellung deckt sich mit der Literatur [1].
- Der implementierte Entscheidungsalgorithmus stimmhaft/stimmlos ist nicht sehr robust. Dies ist aber nicht tragisch, da diese Aufgabe für den DSP-Vocoder bereits gelöst wurde.
- Die robusteste Methode ist die 4. Wurzel-Methode. Sie ist relativ unempfindlich gegenüber ausgeprägten Formanten und vermag auch noch relativ tiefe Pitchfrequenzen zu erkennen. Der «Mittelweg» zwischen AKF und Cepstrum scheint also eine gute Methode zu sein, was auch die Literatur [1] bestätigt.

4.2 Implementation für den DSP

Die Implementation für den DSP lässt sich in zwei Phasen unterteilen: Erstellung eines Pitchextractors mit Pitchgenerator und die anschliessende Integration in das bestehende Vocodersystem.

4.2.1 Pitch-Extractor mit Pitch-Generator

In der ersten Phase wurde der Pitchextraction-Algorithmus mit den drei Methoden für den DSP implementiert. Dabei wurde ein Testprogramm[6] erstellt, das aus einem eingespeisten Sprachsignal die Pitch extrahiert und mittels dem in der Vorgängerstudienarbeit [2] erstellen Pitchgenerator wieder einen Ton mit der extrahierten Pitchfrequenz erzeugt. Dabei können die vom DSP berechneten Daten (das Rücktransformierte des Spektrums und der Pitchverlauf) mittels MATLAB heraufgeladen und visualisiert werden.

C-Programm für DSP

Der DSP wurde in der Sprache C programmiert. Dafür standen der C-Compiler der Firma ANALOG DEVICES zur Verfügung, der bequem mit dem an der HSR erstellten MATLAB-GUI *comptool* bedient werden kann. Dieses compiliert die Sourcen nicht nur, sondern erledigt auch gleich die Aufgaben Boardreset, Download und Start des Programms. Es mussten also keine Skripte zur Erledigung dieser Aufgaben geschrieben werden.

Das Testprogramm liest vom A/D-Wandler Interrupt-gesteuert per chained-DMA-Transfer Frame um Frame in den Speicher ein, um diese dann innerhalb einer Interrupt Service Routine (ISR) zu bearbeiten. Dabei wird die Pitch gemäss der gewählten Methode extrahiert und dem Pitchgenerator eingespeist. Dieser generiert ein Signal mit der gewünschten Pitch das anschliessend wieder an den D/A-Wandler ausgegeben wird. Die Ein- und Ausgangssignale liegen auf Kanal 1 auf der I/O-Box an. Für genauere Informationen sind der Source-Code und die darin enthaltenen Kommentare zu konsultieren (im Anhang).

Eine Schwelle gegeben durch die Variable `tresh`, dient als simple Entscheidungsschwelle zur Unterscheidung ob ein Signal anliegt oder nicht. Die Extraktionsstufe setzt die Pitch im letzten

[6] File: `ROOT/imp/pextract/pextract.c`

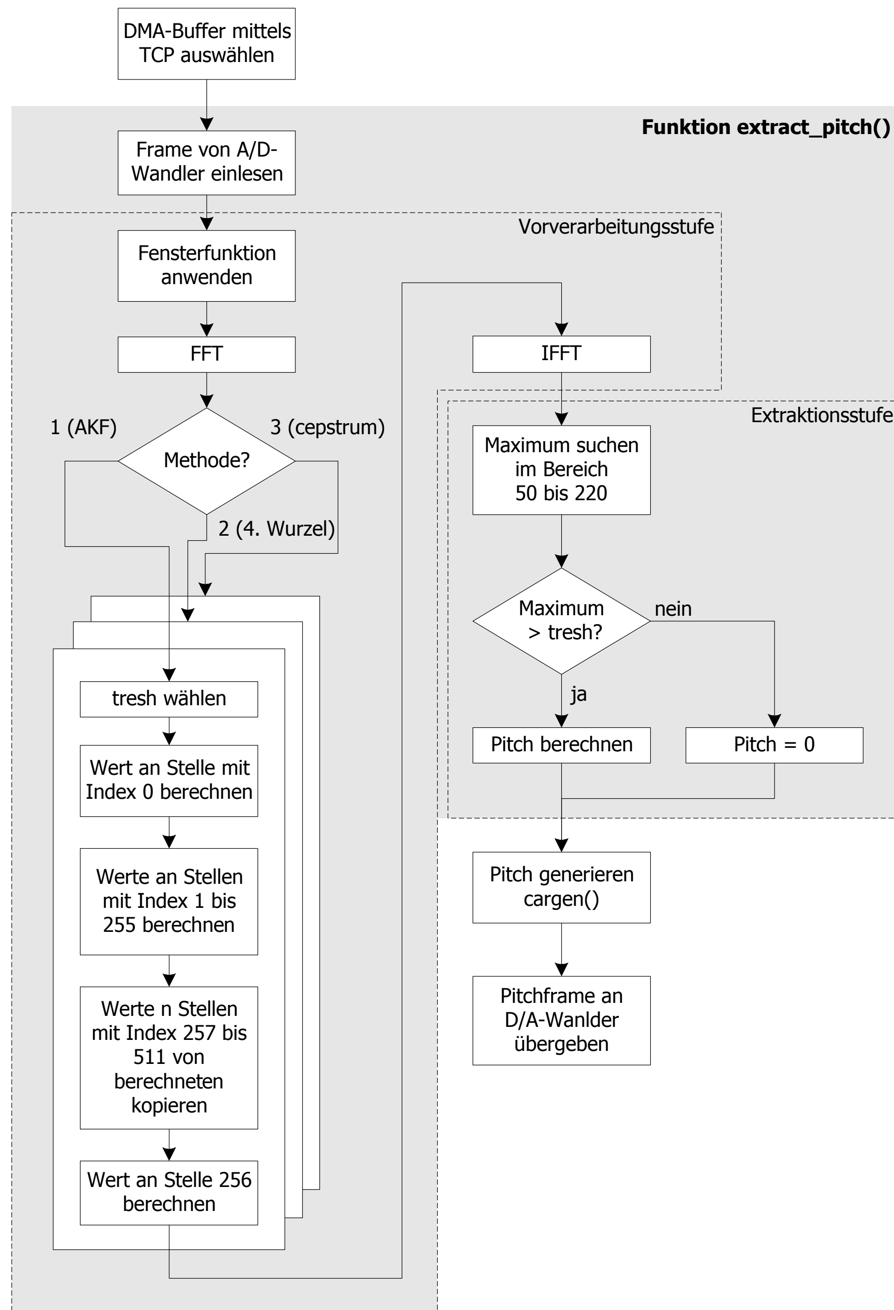


Abbildung 4.18: *Interrupt Service Routine (ISR). Die grau hinterlegte Fläche markiert den in der Funktion* `extract_pitch()` *implementierte Code. Die dreidimensional übereinander gezeichneten Rechtecke stellen die analogen Algorithmen der anderen Methoden dar. Der einzige Unterschied besteht in der Funktion der nichtlinearen Verzerrung (nicht explizit dargestellt).*

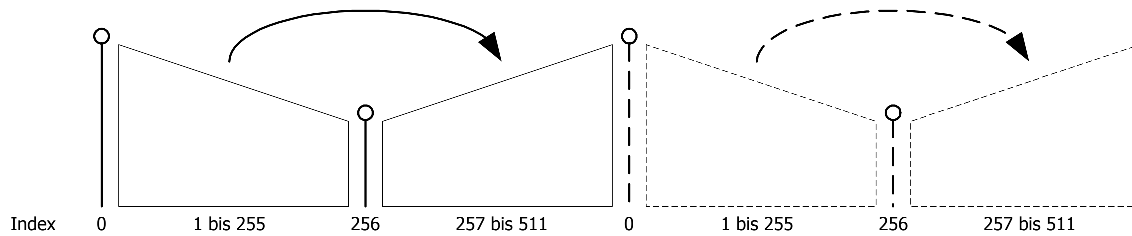


Abbildung 4.19: *Nichtlineare Verzerrung im Frequenzbereich: Die Werte an den Stellen mit Index 1 bis 255 müssen nur einmal berechnet werden und können an die Stellen mit Index 257 bis 511 gespiegelt werden. Der gestrichelt gezeichnete Teil stellt die nächste Periode des periodischen Spektrums dar.*

Fall gleich Null. Damit wird ein unkontrolliertes wobbeln des Ausgangssignals verhindert, falls kein Eingangssignal anliegt, was besonders bei Experimenten mit einem Mikrophon angenehm ist. Da jede Methode in einem anderen Wertebereich arbeitet[7], muss auch diese Schwelle für jede Methode entsprechend gewählt werden. Abbildung 4.18 zeigt das Flussdiagramm der ISR des Programms `pextract.c`.

Optimierungen

Zwei «Optimierungstricks» gilt es noch zu erklären: Das eingelesene mit einer Fensterfunktion[8] gewichtete und in den Frequenzbereich transformierte Frame soll ja einer nichtlinearen Verzerrung unterzogen werden. Da es sich bei dem Amplitudenspektrum um eine Periode eines periodischen Spektrums handelt, muss die Funktion der nichtlinearen Verzerrung nur auf die erste Hälfte des Spektrums angewendet werden, die zweite Hälfte kann von der ersten kopiert, bzw. gespiegelt werden. Die Rücktransformation liefert wieder eine rein reelle Zeitfunktion.

Da das Frame eine Länge mit gerader Anzahl Werte besitzt, muss der mittlere Wert bei Index 256 nicht gespiegelt dafür aber separat (nicht in einer Schleife) bearbeitet werden. Dies trifft auch auf den Wert bei Index 0 zu, da dieser am Ende der Periode nicht mehr vorkommt (dieser Wert wäre wieder der Wert bei Index 0 der *nächsten* Periode). Die Werte mit Index 1 bis 255 werden in einer Schleife berechnet und zugleich an die Stellen mit Index 257 bis 511 gespiegelt (vgl. Abbildung 4.19).

Der zweite «Trick» ist folgender: Die nichtlineare Verzerrung ist bei der Cepstrum-Methode wie folgt definiert:

$$\tilde{S}(\omega) = \log\left(|S(\omega)|\right) = \log\left(\sqrt{\mathrm{Re}^2 + \mathrm{Im}^2}\right) = \frac{1}{2}\log\left(\mathrm{Re}^2 + \mathrm{Im}^2\right) \Rightarrow \log\left(\mathrm{Re}^2 + \mathrm{Im}^2\right)$$

Der Faktor $1/2$ kann auch weggelassen werden, da nur die Position des Spitzenwertes interessiert, nicht der Wert selbst. Die Wurzelfunktion kann also Dank der Logarithmusfunktion weggelassen werden, was einen entscheidenden Vorteil bringt. Würde man diese nicht weglassen, so reicht die Rechenleistung des verwendeten DSPs nicht mehr, um die Cepstrum-Methode zu verarbeiten (vgl. Abschnitt «Auslastung des DSP»).

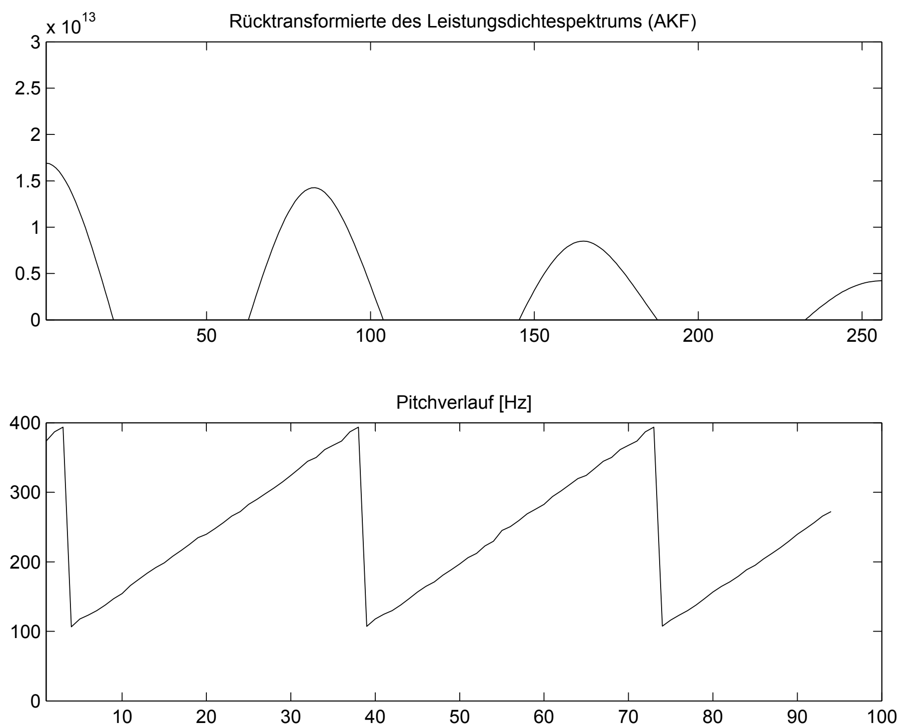


Abbildung 4.20: *Von einem Sweep von 100 bis 400 Hz wird mit der AKF-Methode die Pitch bestimmt und dessen Verlauf aufgezeichnet. File: `getpitch_akf.m`*

Datenvisualisierung mit MATLAB

Neben den Tonausgabe der Pitch kann auch mit MATLAB auf die berechneten Daten zugegriffen werden. MATLAB erhält dafür mittels dem MEX-File `matlab.dll` Zugriff auf das DSP-Board. Mit Hilfe einfacher Befehle können so aus MATLAB Daten vom DSP hoch- oder auch heruntergeladen werden. Durch herunterladen der Variable *method* kann von MATLAB aus gesteuert werden welche Pitchextraction-Methode der DSP anwenden soll.

Zur Visualisierung der berechneten Daten wurden drei MATLAB-Programme[9] (M-Files) erstellt, die für jede der drei Methoden die Daten *Rücktransformiertes des nichtlinear verzerrten Spektrums* und *die berechnete Pitch* vom DSP hochladen und darstellen. Ersteres wird laufend dargestellt, d.h. es ist immer das momentane Rücktransformierte zu sehen, während letzteres als Pitchverlauf dargestellt wird. Die Kommunikation zwischen DSP und MATLAB verläuft nicht synchronisiert. D.h. es kann vorkommen, dass der Buffer im DSP mit neuen Werten überschrieben wird während MATLAB diesem hochlädt. Das macht sich bemerkbar als kurzzeitiger «Darstellungsfehler». Im Allgemeinen funktioniert es aber recht gut.

Abbildungen 4.20, 4.21 und 4.22 zeigen für jede Methode ein Beispiel. Eingangssignal war ein Sweep von 100 bis 400 Hz eines Dreiecksignals mit der Amplitude 2 V_{pp}. Wieder schön zu sehen ist wie die Cepstrum-Methode Mühe hat tiefe Frequenzen zu erkennen. Der Peak ist bei tiefen Pitchfrequenzen derart klein, dass er sich nicht genügend von den anderen «Störpeaks» abhebt.

[7] die AKF-Methode expandiert den Wertebereich während die Cepstrum-Methode und die 4. Wurzel-Methode den Wertebereich verschieden stark komprimieren

[8] Es wurde stets ein Hanning-Fenster verwendet.

[9] `ROOT/imp/pextract/getpitch_akf.m`, `ROOT/imp/pextract/getpitch_ceps.m` und `ROOT/imp/pextract/getpitch_sqrt.m`

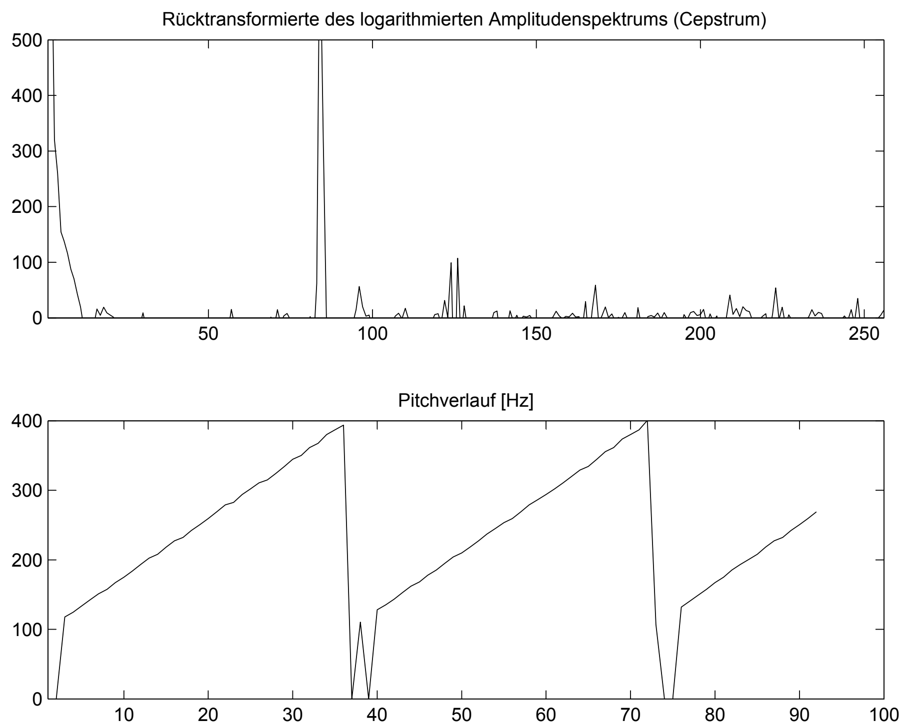


Abbildung 4.21: *Von einem Sweep von 100 bis 400 Hz wird mit der Cepstrum-Methode die Pitch bestimmt und dessen Verlauf aufgezeichnet. File:* `getpitch_ceps.m`

Die anderen beiden Methoden arbeiten besser.

Bedienung

Um das System in Betrieb zu nehmen sind folgende Schritte nötig:

1. Programm `pextract.c` mittels *comptool* kompilieren, herunterladen und DSP starten.
2. Eingangssignal an Input des Kanals 1 der I/O-Box anschliessen. Dabei ist darauf zu achten, dass die Eingangsspannung etwa 2 V_{pp} beträgt.
3. Das Ausgangssignal liegt an Output des Kanals 1 an. Dieses kann auf einen aktiven Lautsprecher gegeben werden.
4. Entsprechend der gewünschten Methode eines der M-Files `getpitch_akf.m`, `getpitch_ceps.m` oder `getpitch_sqrt.m` in MATLAB starten.
5. Um das MATLAB-Programm wieder zu beenden muss im MATLAB-Commandwindow CTRL-C gedrückt werden, da das Programm eine Endlosschleife enthält.

Läuft das System, kann in Echtzeit die Pitchextraction graphisch und akustisch mitverfolgt werden. Schliesst man am Eingang ein Mikrophon mit Vorverstärker an, kann auf effektive Weise die Funktion der drei Methoden verifizieren, die erwartungsgemäss arbeiten. Alternativ kann auch der Ausgang der Soundkarte des PCs angeschlossen werden, um von diesem Wavefiles abzuspielen. Da die Anzeige im MATLAB vergleichsweise träge ist (das Aktualisierungsintervall ist verhältnismässig lang), kann eine Aufzeichnung des Pitchverlaufs im Stile der Simulation nicht

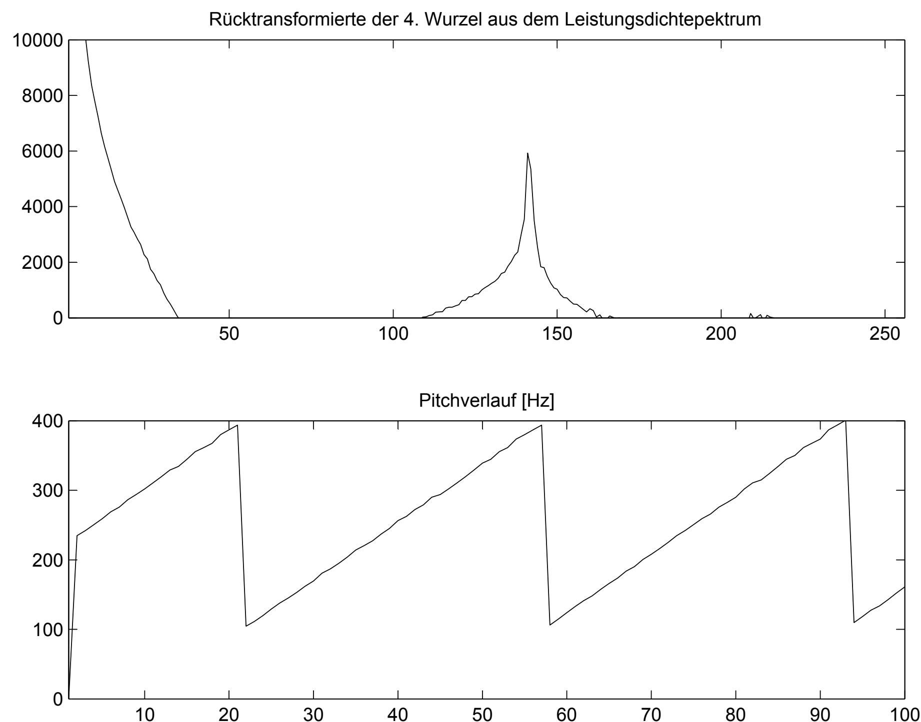


Abbildung 4.22: *Von einem Sweep von 100 bis 400 Hz wird mit der 4. Wurzel-Methode die Pitch bestimmt und dessen Verlauf aufgezeichnet. File:* `getpitch_sqrt.m`

erwartet werden. Es empfiehlt sich daher das MATLAB-Programm nicht ablaufen zu lassen und nur eine akustische Kontrolle vorzunehmen. Aufgrund des sehr simplen stimmhaft/stimmlos-Entscheidungsalgorithmus werden die stimmlosen Laute nicht zuverlässig erkannt, weshalb die generierte Pitch in diesen Bereichen zufällig umherspringt. Während den stimmhaften Lauten aber kann eine zuverlässige Pitchextraktion beobachtet werden.

Resultate

Der entwickelte Pitchextraktionsalgorithmus konnte erfolgreich für den DSP implementiert werden. Mittels der MATLAB-Visualisierung und der akustischen Kontrolle konnte die richtige Funktion und die aus der Simulation erwarteten Probleme – relativ hohe tiefste erkennbare Pitchfrequenz aufgrund der kurzen Framelänge und die Schwäche bei der Entscheidung stimmhaft/-stimmlos durch einen allzu simplen Algorithmus – verifiziert werden. Das letztere Problem stellt, wie bereits erwähnt, kein wirkliches Problem dar, da im bestehenden Vocodersystem bereits ein besserer solcher Algorithmus eingebaut ist. Der nächste Schritt – der Einbau des Pitchextractors in das Vocodersystem – konnte also in Angriff genommen werden.

4.2.2 Einbau des Pitchextractors in das Vocodersystem

Beim Einbau in das bestehende Vocodersystem wurde von dem in der Vorgängerstudienarbeit [2] erstellten System mit bereits eingebautem Pitchgenerator ausgegangen[10]. Die Implementation mit Pitchextractor ist unter `ROOT/imp/vocoder4/vocoder4.c` abgelegt. Das Vocodersystem arbeitet

[10]File: `ROOT/imp/vocoder3/vocoder3.c`

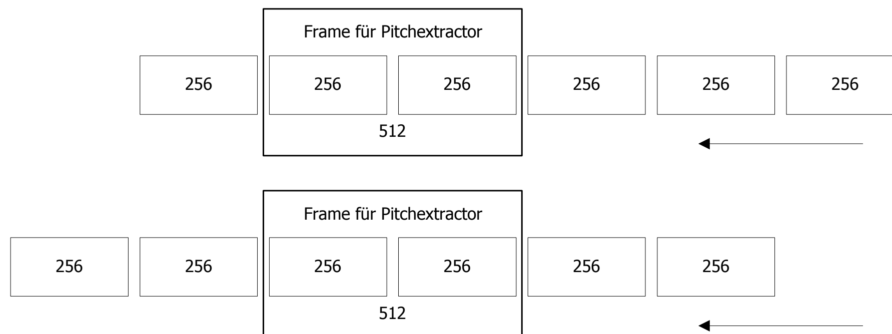


Abbildung 4.23: *Zwei «Vocoderframes» der Länge 256 werden in überlappender Weise zu einem Frame der Länge 512 für dem Pitchextractor zusammengefasst. Das «alte Vocoderframe» im «Pitchextractorframe» muss erst an den Anfang kopiert werden bevor das «neue Vocoderframe» an das Ende des «Pitchextractorframe» kopiert werden kann (siehe Zeilen 1313 bis 1319 in `vocoder4.c`).*

auch frameorientiert, d.h. das Eingangssignal wird in Frames eingelesen, jedoch mit einer Framelänge von 256 Werten, also der Hälfte der Werte als bis anhin in der Simulation und im Testprogramm verwendet wurden. Da aber eine Framelänge von mindestens 512 Werten zwingend für einen ausreichenden erkennbaren Pitchbereich von Nöten ist, musste Abhilfe geschaffen werden.

Überlappende Frames

Die Lösung besteht in der Zusammenfassung von zwei Frames der Länge 256 zu einem Frame der Länge 512. Diese wurden aber nicht nur paarweise zusammengefasst, sondern in überlappender Weise, wie es schon in der Simulation geschehen ist. Abbildung 4.23 erklärt diesen Vorgang. Diese Überlappung hat zwei Vorteile:

1. Es wird für jedes Frame eine Pitchfrequenz bestimmt, obwohl die Framelänge von 256 Werten dazu nicht ausreichen würde. Dies verleiht dem System die Fähigkeit, schnell (alle 11.6 ms) auf Änderungen der Pitchfrequenz zu reagieren.
2. Mit der Überlappung wird sozusagen als Nebeneffekt gleich eine leichte Glättung des Pitchverlaufs vorgenommen.

Integration

Die Integration in das bestehende System umfasste folgende Schritte:

1. Erstellen der C-Funktion `calcpitch()` (hier findet die Pitchextraction statt) und deren Funktionsaufruf in der ISR (Zeile 248).
2. Erstellen der C-Funktion zur Variableninitialisierung `initcalcpitch()` und dessen Aufruf in `main()` (Zeile 1541).
3. Aktualisieren der Variable `fs`, die die aktuelle Pitchfrequenz enthält (der Pitchgenerator liest diese Variable aus).

Die Funktion `calcpitch()` konnte aus dem Testprogramm `pextract.c` übernommen werden. Einzig das Einlesen des Frames in `pextract.c` musste in `vocoder4.c` durch die erwähnte Zusammenfassung von zwei «Vocoderframes» ersetzt werden.

Bedienung

Wie das Testprogramm kann `vocoder4.c` mittels *comptool* kompiliert, auf den DSP heruntergeladen und gestartet werden. Bereits dann ist der Vocoder mit Pitchextractor voll in Betrieb. Das Eingangssignal muss in Input in Kanal 2 der I/O-Box eingespeist werden, das Ausgangssignal muss in Output des Kanals 1 abgegriffen werden. Mit MATLAB kann die Pitchextraction-Methode mittels folgender Befehle gewählt werden:

- `sharc('dl_int','method',1)` um die AKF-Methode auszuwählen
- `sharc('dl_int','method',2)` um die 4. Wurzel-Methode auszuwählen
- `sharc('dl_int','method',3)` um die Cepstrum-Methode auszuwählen

So kann im laufenden Betrieb zwischen den verschiedenen Methoden umgeschaltet werden.

Hörtests mit Wavefiles

Um die Funktion und die Fähigkeit der Implementation zu testen, wurden einige Versuche mit Wavefiles angestellt. Wie zu erwarten war, bereiten diejenigen mit tieffrequenten Sprecherstimmen Probleme bei der Pitchextraction. Die Pitch springt dann gerne zu einer höheren Frequenz. Wenn dies nur sehr kurzzeitig der Fall ist, stört das auch nicht übermässig. Die Grenzen dieses Algorithmus werden aber klar bemerkbar. Die Beobachtungen die bereits in der Simulation gemacht wurden, bestätigten sich auch bei diesen Hörtests: Die Cepstrum-Methode hat besonders Mühe mit tiefen Pitchfrequenzen. Die höherfrequenten Frauenstimmen werden jedoch gut übertragen. Es ist erstaunlich, wie sehr natürlicher die synthetisierte Stimme klingt, wenn die Pitch mit übertragen wird.

Insgesamt lässt sich sagen, dass sich die 4. Wurzel-Methode als die robusteste herausstellte. Sie arbeitet auch noch mit relativ tiefen Pitchfrequenzen recht zuverlässig. Ähnlich verhält es sich mit der AKF-Methode. Am schlechtesten schnitt die Cepstrum-Methode ab. Sie hat die grössten Probleme mit tiefen Pitchfrequenzen, während sie ansonsten eigentlich zuverlässig arbeitet.

Auslastung des DSP

Ein wichtiges Thema ist die Auslastung des DSP. Diese darf auf keinen Fall über 100% steigen, da dann die ISR nicht mehr komplett abgearbeitet werden kann. Um die Auslastung zu messen wird beim Start der ISR ein Timer gestartet der nach dem Beenden der ISR wieder gestoppt wird. Die Anzahl der Taktzyklen, die während dieser Zeit vergangen sind, werden in der Variablen `cycles` gespeichert. Mit dem MATLAB-Befehl

```
sharc('ul_int','cycles')/(40e6/22050*256)*(-100)
```

kann diese Variable heraufgeladen und zugleich in Prozent Auslastung umgerechnet werden. Folgende Messresultate wurden für die verschiedene Methoden ermittelt:

Methode	Auslastung
AKF-Methode	96.1%
Cepstrum-Methode	98.2%
4. Wurzel-Methode	99.6%

Die Cepstrum-Methode bewegt sich also hart an der Grenze der Rechenleistung des DSP. Im Betrieb konnten jedoch keine Auffälligkeiten festgestellt werden. Die AKF-Methode benötigt klar am wenigsten Rechenleistung, da bei dieser Methode rechenaufwändige Funktionen wie `sqrtf()`

und `logf()` wegfallen. Aus dem Vergleich mit der Auslastung von 81.9%, die ohne den Pitchextractor gemessen wird, kann einen ungefähre Anzahl für die Pitchextraction (Cepstrum-Methode) benötigter Taktzyklen von 82183 berechnet werden, was 17.7% der verfügbaren Rechenleistung entspricht.

Jetzt wird klar dass eine grössere Framelänge für die Pitchextraction nicht in Frage kommt. Die nächst grössere Stufe wäre eine Framelänge von 1024 Werten[11]. Diese FFT- und IFFT-Funktionen zusammen würden 69144 Taktzyklen mehr benötigen[12], was weiteren 14.9% der verfügbaren Zyklen entspricht. Diese Anforderung kann der verwendete DSP sicherlich nicht erfüllen.

4.2.3 Datenrate bei der Übertragung

Interessant ist die Datenrate, die schlussendlich durch die Sprachcodierung entsteht. Geht man davon aus, dass alle Sprachmerkmale (38 Amplitudenwerte in den 38 Teilbändern plus die Pitch) mit 8 Bit codiert werden[13], so entsteht folgende Datenrate:

$$\frac{39 \cdot 8\,\text{Bit}}{11.6\,\text{ms}} = 26897\,{}^{Bit}\!/_{\text{s}} = 26.3\,{}^{kBit}\!/_{\text{s}}$$

Dies entspricht gegenüber einem PCM-Sprachsignal mit einer Datenrate von 64000 Bit/s (8 Bit @ 8 kHz) einem Reduktionsfaktor von 2.4. In Anbetracht dessen, dass es sich bei diesem Vocodersystem um die einfachste und ursprünglichste Implementation handelt und der Sprachqualität, die dieses System liefert, ist dies ein gutes Resultat.

4.2.4 Resultate

Der entwickelte Pitchextraction-Algorithmus konnte mit allen drei Methoden erfolgreich in das bestehende Vocodersystem eingebaut werden. Die Rechenleistung des DSP reicht gerade noch aus, um die rechenaufwändigste Methode – die Cepstrum-Methode – zu verarbeiten. Probleme mit tieffrequenten Sprecherstimmen traten erwartungsgemäss auch hier auf. Insgesamt lässt sich sagen, dass sich die 4. Wurzel-Methode als die robusteste herausstellte. Sie arbeitet auch noch mit relativ tiefen Pitchfrequenzen recht zuverlässig. Sprachsignale mit hoher Pitch, typischerweise Frauenstimmen, klangen natürlich und werden praktisch ohne Fehler in der Pitch übertragen. Die Verständlichkeit und Natürlichkeit des synthetisierten Signals konnte also durch den Einbezug der Pitch stark verbessert werden.

4.2.5 Verbesserungsansätze

Das grösste Problem des implementierten Algorithmus – die Beschränkung auf höhere Pitchfrequenzen – lässt sich mit folgenden Ansätzen begegnen:

1. Erhöhung der Fensterlänge für den Pitchextractor auf 1024 Werte. Dies bedingt aber eine Verwendung der entsprechenden (I)FFT-Funktionen, was ev. nur mit einem leistungsfähigeren DSP möglich ist.

2. Eine Optimierung des Algorithmus auf Assemblerebene. Da die (I)FFT-Funktionen, die kaum noch zu optimieren sind, etwa die doppelte Rechenzeit gegenüber dem restlichen

[11] Die FFT und IFFT arbeitet am effizientesten bei Längen von 2^k. Daher sind in der C Runtime Library zu diesem DSP auch nur solche (I)FFT-Funktionen vorhanden.

[12] Gemäss ANALOG DEVICES C Runtime Library Manual zu ADSP-21000 Family

[13] Im aufgebauten System werden die Sprachmerkmale in Variablen des Typs `float` gespeichert, der eine Grösse von 32 Bit hat. Das System arbeitet also mit der vierfachen Datenrate. Es wird davon ausgegangen, dass ohne merkliche Verluste auch mit 8 Bit gearbeitet werden könnte.

Teil des Algorithmus in Anspruch nehmen, scheint dieser Ansatz aber nicht sehr lukrativ. Die Optimierung müsste so weit gehen (und daher auch den Teil *Pitchgenerator* umfassen), dass mit der zur Verfügung stehenden Rechenleistung nach der Handoptimierung (I)FFT-Funktionen der Länge 1024 ausgeführt werden könnten.

3. Die Verwendung eines anderen hier nicht behandelten Algorithmus, der mit der zur Verfügung stehenden Rechenleistung ein besseres Resultat liefert.

Weitere Verbesserungen bezüglich der Natürlichkeit könnte ev. durch einen aufwändigeren Glättungsalgorithmus (Glättung des Pitchverlaufs) erzielt werden. Experimente mit einer anderen Fensterfunktion könnte ev. auch eine Optimierung der Fehlerrate bringen.

4.3 Schlusswort

4.3.1 Ziel erreicht?

Es konnte gezeigt werden, dass mit einem relativ simplen Algorithmus die Pitch in Echtzeit aus einem Sprachsignal extrahiert werden kann. Eingesetzt im bestehenden Vocodersystem erhöht es dessen Qualität der Codierung ungemein. Die synthetisierte Sprache gewinnt erheblich an Verständlichkeit und Natürlichkeit. Was bleibt ist die Erkennungsschwäche für tiefe Pitchfrequenzen. Diese liess sich nicht zufriedenstellend beseitigen, da dazu die zur Verfügung stehende Rechenleistung nicht ausreicht. Für höhere Pitchfrequenzen jedoch funktioniert der Algorithmus mit allen drei Methoden gut.

4.3.2 Persönliche Bemerkungen

Ich habe diese Studienarbeit als interessant und abwechslungsreich empfunden. Ich konnte mich einerseits etwas mehr in MATLAB einarbeiten und die SHARC-DSP-Programmierung näher kennenlernen und andererseits mich mit dem grundlegenden Prinzipen der Sprachcodierung befassen. An dieser Stelle möchte ich mich für die Betreuung und Unterstützung bei Prof. Alex Schüeli und dem Laborassistenten A. Rüegg bedanken.

Andy Rohr

Rapperswil, den 11. Juli 2001

A Listings

A.1 MATLAB Programme (Simulation)

A.1.1 Frame-by-Frame-Vergleich der Pitch-Extraction-Methoden

File: ROOT/sim/method_compare_frame.m

```
1  % Vergleich der drei verschiedenen Methoden:
2  % AKF, 4. Wurzel aus Leistungspektrum und Logarithmus vom Amplitudenspektrum (Cepstrum)
3  % Beschreibung: Mit jedem Tastendruck wird ein Wavefile Stück für Stück mit 50% Überlappung eingelesen und mit den
       obigen drei
4  % Methoden die Pitch bestimmt.
5
6  % Konstanten:
7  filename = 'o';                           % Name des zu analysierenden Wavefiles
8  samplefreq = 22050;                          % Die Samplefrequenz dieses Wavefiles
9  %framelenght = 35;                            % Länge eines Frames in Milisekunden
10 %framesize = samplefreq * framelenght * 0.001; % Länge eines Frames in Samples
11 framesize = 2 * 256;                         % Länge eines Frames in Samples
12 startindex = 50;                             % Startindex ab dem ein die Daten für die Pitchextraktion berücksichtigt
       werden
13
14
15 [inwav,fs,nbits] = wavread(filename);       % ganzes Wavefile einlesen
16 totalsize = length(inwav);                   % und die Anzahl samples bestimmen
17
18 for startsample = 0:framesize/2:totalsize*2, % das ganze Wavefile wird Stückweise mit 50% Überlappung in frame umkopiert
19     if (startsample+framesize < totalsize)   % Prüfen auf das Ende des Wavefiles
20         frame = inwav(startsample+1:startsample+framesize);
21     end
22
23     spec = abs(fft(frame.*hanning(framesize))); % Amplitudenspektrum des hanninggewichteten Wave-Frames
24
25     akf  = real(ifft(spec.*spec));           % AKF über den Frequenzbereich
26     ceps = real(ifft(log(spec)));            % Cepstrum
27     sqrt4 = real(ifft(sqrt(spec)));          % 4. Wurzel aus dem Leistungsdichtespektrum
28
29     clf;                                     % Figur löschen
30
31     subplot(2,2,1);                          % Wave-Frame zeichen
32     plot(frame,'k');
33     title('Wave-Frame');
34     axis([1 framesize -1 1]);
35     grid on;
36
37     subplot(2,2,3);                          % AKF zeichen
38     plot(akf,'k');
39     hold on;
40     title('AKF');
41     [maximum, pos] = max(akf(startindex:framesize/2)); % Maximum bestimmen
42     freq = samplefreq/(pos+startindex);             % Grundfrequenz berechnen
43     xlabel(freq);                            % Berechnete Grundfrequenz unter das Diagramm schreiben
44     plot(pos+startindex-1,maximum,'ko');     % Maximum mit Punkt markieren
45     axis([1 framesize/2 0 30]);
46     grid on;
47
48     subplot(2,2,4);                          % Cepstrum zeichen
49     plot(ceps,'k');
50     hold on;
51     title('Cepstrum');
52     [maximum, pos] = max(ceps(startindex:framesize/2)); % Maximum bestimmen
53     freq = samplefreq/(pos+startindex);              % Grundfrequenz berechnen
```

```
54     xlabel(freq);                         % Berechnete Grundfrequenz unter das Diagramm schreiben
55     plot(pos+startindex-1,maximum,'ko');  % Maximum mit Punkt markieren
56     axis([1 framesize/2 0 0.5]);
57     grid on;
58
59     subplot(2,2,2);                       % 4. Wurzel aus dem Leistungsdichtespektrum zeichen
60     plot(sqrt4,'k');
61     hold on;
62     title('Rücktrans. der 4. Wurzel aus PSD');
63     [maximum, pos] = max(sqrt4(startindex:framesize/2)); % Maximum bestimmen
64     freq = samplefreq/(pos+startindex);           % Grundfrequenz berechnen
65     xlabel(freq);                         % Berechnete Grundfrequenz unter das Diagramm schreiben
66     plot(pos+startindex-1,maximum,'ko');  % Maximum mit Punkt markieren
67     axis([1 framesize/2 0 0.2]);
68     grid on;
69
70     wavplay(frame,fs); wavplay(frame,fs); wavplay(frame,fs); % Wave-Frame dreimal ausgeben
71
72     pause;                                % Warten bis Taste gedrückt
73
74 end
```

A.1.2 Vergleich der Pitch-Extraction-Methoden über ein ganzes Wavefile

File: ROOT/sim/method_compare_whole.m

```
1  % Vergleich der drei verschiedenen Methoden:
2  % AKF, 4. Wurzel aus Leistungspektrum und Logarithmus vom Amplitudenspektrum (Cepstrum)
3  % Beschreibung: Es wird ein Wavefile Stück für Stück mit 50% Überlappung eingelesen und mit den obigen drei
4  % Methoden die Pitch bestimmt. Diese wird in einem Diagramm für jede Methode zusammen mit dem Wavefile dargestellt.
5
6  % Konstanten:
7  filename = 'test2';                   % Name des zu analysierenden Wavefiles
8  samplefreq = 22050;                   % Die Samplefrequenz dieses Wavefiles
9  %framelenght = 35;                    % Länge eines Frames in Milisekunden
10 %framesize = samplefreq * framelenght * 0.001; % Länge eines Frames in Samples
11 framesize = 2 * 256;                  % Länge eines Frames in Samples
12 startindex = 50;                      % Startindex ab dem ein die Daten für die Pitchextraktion berücksichtigt
       werden
13 tresh = 1;                            % Schwelle zur Entscheidung stimmhaft/stimmlos (AKF dient als Quelle)
14 loPitch = 0;                          % Tiefster Pitch-Wert auf der Pitch-Achse (Graph)
15 hiPitch = 300;                        % Höchtser Pitch-Wert auf der Pitch-Achse (Graph)
16
17 % Vektordeklaration
18 freq_akf  = [];
19 freq_ceps = [];
20 freq_sqrt4 = [];
21
22 [inwav,fs,nbits] = wavread(filename);       % ganzes Wavefile einlesen
23 totalsize = length(inwav);                  % und die Anzahl samples bestimmen
24
25 for startsample = 0:framesize/2:totalsize*2, % das ganze Wavefile wird Stückweise mit 50% Überlappung in frame umkopiert
26     if (startsample+framesize < totalsize)  % Prüfen auf das Ende des Wavefiles
27         frame = inwav(startsample+1:startsample+framesize);
28     end
29
30     spec  = abs(fft(frame.*hanning(framesize))); % Amplitudenspektrum des hanninggewichteten Wave-Frames
31
32     akf   = real(ifft(spec.*spec));         % AKF über den Frequenzbereich
33     ceps  = real(ifft(log(spec)));          % Cepstrum
34     sqrt4 = real(ifft(sqrt(spec)));         % 4. Wurzel aus dem Leistungsdichtespektrum
35
36     % Maxima bestimmen
37     [max_akf, pos_akf]   = max(akf(startindex:framesize/2));
38     [max_ceps, pos_ceps] = max(ceps(startindex:framesize/2));
39     [max_sqrt4, pos_sqrt4] = max(sqrt4(startindex:framesize/2));
40
41     % Falls stimmhafte Anregung Frequenz berechnen und dem Vektor hinzufügen, sonst 0 hinzufügen
42     if max_akf < tresh
43         freq_akf  = [freq_akf 0];
44         freq_ceps = [freq_ceps 0];
45         freq_sqrt4 = [freq_sqrt4 0];
46     else
```

```
47        freq_akf  = [freq_akf  samplefreq/(pos_akf+startindex)];
48        freq_ceps = [freq_ceps samplefreq/(pos_ceps+startindex)];
49        freq_sqrt4 = [freq_sqrt4 samplefreq/(pos_sqrt4+startindex)];
50    end
51 end
52
53
54 % Zeichne Wavefile
55 subplot(4,1,1);
56 plot(inwav,'k');
57 ylabel('Wavefile');
58 axis([1 totalsize -1 1]);
59
60 % Zeichne Pitchverlauf bestimmt mit der AKF-Methode
61 subplot(4,1,2);
62 plot(freq_akf,'k');
63 ylabel('Pitch mit AKF');
64 axis([1 2*totalsize/framesize loPitch hiPitch]);
65
66 % Zeichne Pitchverlauf bestimmt mit der Cepstrum-Methode
67 subplot(4,1,3);
68 plot(freq_ceps,'k');
69 ylabel('Pitch mit Cepstrum');
70 axis([1 2*totalsize/framesize loPitch hiPitch]);
71
72 % Zeichne Pitchverlauf bestimmt mit der 4.Wurzel aus PDS-Methode
73 subplot(4,1,4);
74 plot(freq_sqrt4,'k');
75 ylabel('Pitch mit 4. Wurzel');
76 axis([1 2*totalsize/framesize loPitch hiPitch]);
```

A.2 DSP-Programme (Implementation)

A.2.1 Pitch Extractor

File: `ROOT/imp/p_ext/pextract.c`

```
1
2  /***************************************************************************\
3  *                                                                           *
4  *  Pitch Extractor and Generator (Blacktip PCI, Toolkit 5.0)                *
5  *  --------------------------------------------------------
6  *  The Pitch extractor uses the Pitchgenerator developped in SA WS 2000/2001 *
7  *  Developpers: G. Sibler, O. Tresch
8  *                                                                           *
9  *    input CH1 : modulation signal (speech)                                 *
10 *    output CH1 : pitch signal                                              *
11 *                                                                           *
12 *                                                                           *
13 *    Ver - Rev History                                                      *
14 *    --------- ------------------------------------------------------ *
15 *    1.0 - 0   05.07.01 - initial coding (Andy Rohr)                        *
16 *                                                                           *
17 \***************************************************************************/
18
19 /* ADSP-21060 System Register bit definitions */
20 #include <def21060.h>       // register definitions
21 #include <21060.h>          // architecture
22 #include <signal.h>         // interrupt
23 #include <sport.h>          // serial port
24 #include <math.h>           // fmod
25 #include <trans.h>          // for complex values
26 #include <complex.h>        // for complex values
27 #include <stats.h>          // autocorrelation
28 #include "bitsibb.h"        // bitsi board
29 #include "bitsi.h"          // bitsi board
30
31 #define CP_PCI 0x20000      // Program-Controlled Interrupts bit
32 #define CP_MAF 0x1ffff      // Valid memory address field bits
33
34 #define SetIOP(addr, val) (* (int *) addr) = (val)
```

```
35  #define GetIOP(addr)    (* (int *) addr)
36
37  #define MAX_U_AD 2.75       // maximale Eingangsspannung des A/D-Wandlers
38  #define MAX_U_DA 3.0        // maximale Ausgangsspannung des A/D-Wandlers
39
40  #define TCB_BUFS 2          /* number of TCB buffers needed */
41
42  #define FC 22050.0          // Sampling-Frequenz in Hz
43  #define TC 1/FC             // Peridode der Sampling-Frequenz in s
44
45  #define FP 40000000.0       // Prozessor-Takt-Frequenz in Hz
46  #define TP 1/FP             // Peridode der Prozessor-Takt-Frequenz in s
47
48  #define PI 3.14159265358979
49
50  #define FRAME_SIZE 512      // frame size
51  #define P 50                // Buffergroesse Extwave
52
53  /*-----------------------------------------------------------------------------------*/
54
55  /* Initialisierung der Hardware, Spg.-Anpassung an Wandler */
56  int *dms3;                  // pointer for serial port (sport) transfer
57
58  /* ADC-Eingang, DAC-Ausgang: Normierung auf 1 */
59  float norm = 1 / 32767.0;   // normalization of sampled input values
60  float corr = 32767.0;       // denormalization for output values
61
62  /* Variabeln fuer Interrupt-Service-Routine (ISR) */
63  unsigned buf_sel = 1;       // Selektor fuer DMA-Buffer
64
65  /* Variabeln fuer Interrupt-Service-Routine (cargen) (ISR) */
66  int cargenCycles = 0;       // cargen-Zyklen (Zeitmessung)
67  float cargenTime = 0.0;     // cargen-Dauer (Zeitmessung)
68
69
70  /* Variablen fuer Funktion cargen(), Default des Ausgangssignals */
71  int wavetable = 4;          // Nummer des Wavetable, Selektor
72  float fs = 260.0;           // Signal-Frequenz in Hz
73  float amplitude = 1.0/3;    // Amplitude: 0..1, 0=>0Volt(min), 1=>3Volt(max)
74  int tastgrad = 20;          // Tastgrad in % (nur bei Rechteck)
75  float schritthoehe = 0.0;   // Schritthoehe (nur bei Saegezahn, Dreieck)
76  float x1 = 0.0;             // Schrittlaenge1 (nur bei Sinus)
77  float x2 = 0.0;             // Schrittlaenge2 (nur bei Sinus)
78  int countGen = 0;           // Zaehler fuer Takte pro Periode, cargen()
79  float extwave[P];           // Buffer fuer Wave-File-Klang
80
81  /* Buffer, in den cargen samples schreibt */
82  float carframe[FRAME_SIZE]; // Frame fuer CarrierSignal (Pitch)
83
84  /* Variables for the Pitchextractor */
85  float frame[FRAME_SIZE];
86  float acorr[FRAME_SIZE-1];
87  float spec_re[FRAME_SIZE];
88  float spec_im[FRAME_SIZE];
89  float resultframe[FRAME_SIZE];
90  float dummyframe[FRAME_SIZE];
91  float spectrum[FRAME_SIZE];
92  float zeros[FRAME_SIZE];
93  float window[FRAME_SIZE];
94  int method = 1;
95  int max_i;
96  float max;
97  int tresh;
98
99  /* DMA chaining Transfer Control Blocks (TCB) */
100 typedef struct {
101     unsigned** cp;            // Zeiger auf naechsten TCB
102     unsigned  c;              // Count register, Laenge des Buffers
103     int       im;             // Index modifier register, Schrittweite
104     unsigned * ii;            // Index register, Startadresse buffer
105 } __tcb;
106
107 int rx0_buf[TCB_BUFS][FRAME_SIZE];     /* receive buffer */
108 int tx0_buf[TCB_BUFS][FRAME_SIZE];     /* transmit buffer */
109
110 /* TCB's */
```

```
111  __tcb rx0_tcb[TCB_BUFS] = { {0, FRAME_SIZE, 1, 0}, /* receive tcb */
112      {0, FRAME_SIZE, 1, 0}};  /* receive tcb */
113
114  __tcb tx0_tcb[TCB_BUFS] = { {0, FRAME_SIZE, 1, 0}, /* transmit tcb */
115      {0, FRAME_SIZE, 1, 0}};  /* transmit tcb */
116
117  /* SPort DMA-Buffer, TCB's */
118
119  void cargen(void)
120
121  {
122      int maxCountGen = FC/fs; // Anzahl Takte pro Periode des Wavetables
123      int index;               // Schleifen-Zaehler
124      int CountGrenze;         // Zaehlgrenze fuer Teil-Periodenabschnitte
125      float schritt;           // Einzelschrittes bei Saegezahn, Dreieck und Sinus
126      float amplitude2;        // Amplitude fuer 2. Teil der Periode (nur Dreieck)
127
128
129      switch (wavetable) {
130
131      case 1: // Rechteck generieren
132
133          CountGrenze = tastgrad*0.01*maxCountGen;// Tastgrad einstellen
134
135          for (index=0; index < FRAME_SIZE; index++) {  // carframe fuellen
136              if (countGen >= maxCountGen) {    // Reset wenn Perioden-Ende
137                  countGen = 0;
138              }
139
140              if (countGen < CountGrenze) {     // 1. Teil der Periode
141                  carframe[index] = amplitude;
142              } else {                            // 2. Teil der Periode
143                  carframe[index] = -amplitude;
144              }
145
146              countGen++;                     // Zahler inkrementiern
147          }
148
149          break;
150
151
152      case 2: // Saegezahn generieren
153
154          // Kommentar: mit 1.999999 ist 2 gemeint. Wegen eines Optimierungs-Fehlers
155          //            des Comilers (-O2) funktioniert 2 nicht !!
156
157          schritt = 1.999999*amplitude*TC*fs; // Hoehe eines Schrittes
158          //     = 2*amplitude/maxCountGen;  // Aequivalent obere Zeile
159
160          for (index=0; index < FRAME_SIZE; index++) {  // carframe fuellen
161              if (countGen >= maxCountGen) {    // Reset wenn Perioden-Ende
162                  countGen = 0;
163                  schritthoehe = 0.0;
164              }
165
166              carframe[index] = (amplitude-schritthoehe);
167              schritthoehe += schritt;        // Schritthoehe hochzaehlen
168
169              countGen++;                     // Perioden-Zaehler inkrementiern
170          }
171
172          break;
173
174
175      case 3: // Dreieck generieren
176
177          schritt = 4*amplitude*TC*fs;        // Hoehe eines Schrittes
178          //     = 4*amplitude/maxCountGen;  // Aequivalent obere Zeile
179
180          CountGrenze = maxCountGen*0.5;      // Grenze bei halber Periode
181          amplitude2 = 3*amplitude;           // Ampliude fuer 2. Teil der Periode
182
183          for (index=0; index < FRAME_SIZE; index++) {  // carframe fuellen
184              if (countGen >= maxCountGen) {    // Reset wenn Perioden-Ende
185                  countGen = 0;
186                  schritthoehe = 0.0;
```

```
187             }
188 
189             if (countGen < CountGrenze) {     // 1. Teil der Periode
190                 carframe[index] = (-amplitude+schritthoehe);
191                 schritthoehe += schritt;
192             } else {                            // 2. Teil der Periode
193                 carframe[index] = (amplitude2-schritthoehe);
194                 schritthoehe += schritt;    // Schritthoehe hochzaehlen
195             }
196 
197             countGen++;                     // Perioden-Zaehler inkrementiern
198         }
199 
200         break;
201 
202 
203     case 4: // Sinus generieren
204 
205         schritt = 2*3.141592*TC*fs;         // Laenge eines Schrittes
206         //     = 2*3.141592/maxCountGen;    // Aequivalent obere Zeile
207 
208         CountGrenze = maxCountGen*0.5;      // Grenze bei halber Periode
209 
210         for (index=0; index < FRAME_SIZE; index++) {  // carframe fuellen
211             if (countGen >= maxCountGen) {    // Reset wenn Perioden-Ende
212                 countGen = 0;
213                 x1 = 0;
214                 x2 = 0;
215             }
216 
217             if (countGen < CountGrenze) {     // 1. Teil der Periode
218                 carframe[index] = amplitude*(x1-((x1*x1*x1)*0.1666)
219                                    +((x1*x1*x1*x1*x1)*0.008333)
220                                    -((x1*x1*x1*x1*x1*x1*x1)*0.0001984)
221                                    +((x1*x1*x1*x1*x1*x1*x1*x1*x1)*0.000002756)
222                                    -((x1*x1*x1*x1*x1*x1*x1*x1*x1*x1*x1)*0.00000002505));
223                 x1 += schritt;              // Abszissenpunkt hochzaehlen
224             } else {                            // 2. Teil der Periode
225                 carframe[index] = -amplitude*(x2-((x2*x2*x2)*0.1666)
226                                     +((x2*x2*x2*x2*x2)*0.008333)
227                                     -((x2*x2*x2*x2*x2*x2*x2)*0.0001984)
228                                     +((x2*x2*x2*x2*x2*x2*x2*x2*x2)*0.000002756)
229                                     -((x2*x2*x2*x2*x2*x2*x2*x2*x2*x2*x2)*0.00000002505));
230                 x2 += schritt;              // Abszissenpunkt hochzaehlen
231             }
232 
233             countGen++;                     // Perioden-Zaehler inkrementiern
234         }
235 
236         break;
237 
238 
239     case 5: // externer Wave-File-Klang
240 
241         for (index=0; index < FRAME_SIZE; index++) {
242             if (countGen >= P) {          // Reset wenn Perioden-Ende
243                 countGen = 0;
244             }
245 
246             carframe[index] = extwave[countGen];// Werte umkopieren
247 
248             countGen++;                     // WaveTableLaenge-Zaehler inkrementiern
249         }
250 
251         break;
252 
253     default: // carframe mit 0 fuellen, wenn ungueltiges Wavetable gewaehlt
254 
255         for (index=0; index < FRAME_SIZE; index++) {
256             carframe[index] = 0.0;
257         }
258     }
259 }
260 
261 // This function reads in a frame by DMA-Transfer and extracts the pitch. The pitchextraction-
262 // method is selected by the variable "method" which ca be set for MATLAB
```

```
263 // MATLAB programms can also load up the pitch (variable "fs") and the resultframe
264 // (see project documentation)
265 void extract_pitch() {
266
267     int i;
268
269     // fill workingframe and apply window
270     for (i=0; i < FRAME_SIZE; i++) {
271         frame[i] = (float)(rx0_buf[buf_sel][i]) * norm * window[i];
272     }
273
274     // transform the the windowed frame to frequency domain
275     rfft512(frame, spec_re, spec_im);
276
277     // apply the AKF-method
278     if (method == 1) {
279         tresh = 2000000000;
280         // need only to apply nonlinear function to first half of spectrum
281         // the second half is copied from the first half (mirror)
282         spectrum[0] = spec_re[0]*spec_re[0] + spec_im[0]*spec_im[0];
283         for (i=0; i < FRAME_SIZE/2; i++) {
284             spectrum[i] = spec_re[i]*spec_re[i] + spec_im[i]*spec_im[i];
285             spectrum[FRAME_SIZE-i] = spectrum[i];
286         }
287         spectrum[FRAME_SIZE/2] = spec_re[FRAME_SIZE/2]*spec_re[FRAME_SIZE/2] + spec_im[FRAME_SIZE/2]*spec_im[FRAME_SIZE/2];
288     }
289
290     // apply the "quadroot of PSD"-method
291     if (method == 2) {
292         tresh = 1000;
293         // need only to apply nonlinear function to first half of spectrum
294         // the second half is copied from the first half (mirror)
295         spectrum[0] = sqrtf(sqrtf(spec_re[0]*spec_re[0] + spec_im[0]*spec_im[0]));
296         for (i=1; i < FRAME_SIZE/2; i++) {
297             spectrum[i] = sqrtf(sqrtf(spec_re[i]*spec_re[i] + spec_im[i]*spec_im[i]));
298             spectrum[FRAME_SIZE-i] = spectrum[i];
299         }
300         spectrum[FRAME_SIZE/2] = sqrtf(sqrtf(spec_re[FRAME_SIZE/2]*spec_re[FRAME_SIZE/2] + spec_im[FRAME_SIZE/2]*spec_im[
               FRAME_SIZE/2]));
301     }
302
303     // apply the cepstrum-method
304     if (method == 3) {
305         tresh = 150;
306         // need only to apply nonlinear function to first half of spectrum
307         // the second half is copied from the first half (mirror)
308         spectrum[0] = logf(spec_re[0]*spec_re[0] + spec_im[0]*spec_im[0]);
309         for (i=0; i < FRAME_SIZE/2; i++) {
310             spectrum[i] = logf(spec_re[i]*spec_re[i] + spec_im[i]*spec_im[i]);
311             spectrum[FRAME_SIZE-i] = spectrum[i];
312         }
313         spectrum[FRAME_SIZE/2] = logf(spec_re[FRAME_SIZE/2]*spec_re[FRAME_SIZE/2] + spec_im[FRAME_SIZE/2]*spec_im[
               FRAME_SIZE/2]);
314     }
315
316     // transform back to time domain
317     ifft512(spectrum, zeros, resultframe, dummyframe);
318
319     // initialize vars
320     max = 0;
321     max_i = 1;
322
323     // search the peak in the resultframe which determines the pitch
324     for (i=50; i < FRAME_SIZE/2-35; i++) {
325         if (resultframe[i] > max) {
326             max = resultframe[i];
327             max_i = i;
328         }
329     }
330
331     // finally calculating the pitch if the peak is above "tresh"
332     if (max > tresh) fs = FC/max_i;
333     else           fs = 0;
334 }
335
336 /*-----------------------------------------------------------------------------------*/
```

```
337 // Interrupt-Service-Routine fuer Wert-Ausgabe D/A-Wandler, IRQ von DMA
338 void spr0_asserted( int sig_num )
339 {
340     int index;
341
342     if (buf_sel == 1) { // TCB-select, change DMA-Buffer
343         buf_sel = 0;
344     } else {
345         buf_sel = 1;
346     }
347
348     extract_pitch();    // read a frame in from A/D converter and extract the pitch
349     cargen();           // neue Werte in carframe schreiben
350
351     // write the carrierframe to the output buffer
352     for (index=0; index < FRAME_SIZE; index++) {      // DMA-Buffer fuellen
353         tx0_buf[buf_sel][index] = (((unsigned) (corr * carframe[index]) << 16) & 0xffff0000);
354     }
355 }
356
357 /*-------------------------------------------------------------------------------------*/
358
359 void setup_sport0( void ){
360     /* Configure SHARC serial port */
361
362     /* TRANSMIT CONTROL REGISTER */
363     /* An alternate (and more efficient) way of doing this would be to */
364     /* write the 32-bit register all at once with a statement like this: */
365     /*      SetIOP(STCTL0, 0x001c00f2);                        */
366     /* But the following is more descriptive...                */
367
368     sport0_iop.txc.mdf = 0;   /* multichannel frame delay (MFD)  */
369     sport0_iop.txc.schen = 1; /* Tx DMA chaining enable          */
370     sport0_iop.txc.sden = 1;  /* Tx DMA enable                   */
371     sport0_iop.txc.lafs = 0;  /* Late TFS (alternate)            */
372     sport0_iop.txc.ltfs = 1;  /* Active low TFS                  */
373     sport0_iop.txc.ditfs = 0; /* Data independent TFS            */
374     sport0_iop.txc.itfs = 0;  /* Internally generated TFS        */
375     sport0_iop.txc.tfsr = 1;  /* TFS Required                    */
376
377     sport0_iop.txc.ckre = 0;  /* Data and FS on clock rising edge */
378     sport0_iop.txc.gclk = 0;  /* Enable clock only during transmission*/
379     sport0_iop.txc.iclk = 0;  /* Internally generated Tx clock   */
380     sport0_iop.txc.pack = 0;  /* Unpack 32b words into two 16b tx's */
381
382     sport0_iop.txc.slen = 31; /* Data word length minus one      */
383     sport0_iop.txc.sendn = 0; /* Data word endian 1 = LSB first  */
384     sport0_iop.txc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
385     /* Data type specifier             */
386     sport0_iop.txc.spen = 1;  /* Enable (clear for MC operation) */
387
388     /* RECEIVE CONTROL REGISTER */
389     sport0_iop.rxc.nch = 31;  /* multichannel number of channels - 1 */
390     sport0_iop.rxc.mce = 0;   /* multichannel enable             */
391     sport0_iop.rxc.spl = 0;   /* Loop back configure (test)      */
392     sport0_iop.rxc.d2dma = 0; /* Enable 2-dimensional DMA array  */
393     sport0_iop.rxc.schen = 1; /* Rx DMA chaining enable          */
394     sport0_iop.rxc.sden = 1;  /* Rx DMA enable                   */
395     sport0_iop.rxc.lafs = 0;  /* Late RFS (alternate)            */
396     sport0_iop.rxc.ltfs = 1;  /* Active low RFS                  */
397     sport0_iop.rxc.irfs = 0;  /* Internally generated RFS        */
398     sport0_iop.rxc.rfsr = 1;  /* RFS Required                    */
399     sport0_iop.rxc.ckre = 0;  /* Data and FS on clock rising edge */
400     sport0_iop.rxc.gclk = 0;  /* Enable clock only during transmission*/
401     sport0_iop.rxc.iclk = 0;  /* Internally generated Rx clock   */
402     sport0_iop.rxc.pack = 0;  /* Pack two 16b rx's into 32b word */
403
404     sport0_iop.rxc.slen = 31; /* Data word length minus one      */
405     sport0_iop.rxc.sendn = 0; /* Data word endian 1 = LSB first  */
406     sport0_iop.rxc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
407     /* Data type specifier             */
408     sport0_iop.rxc.spen = 1;  /* Enable (clear for MC operation) */
409
410     /* Enable sport0 xmit irqs */
411     interruptf(SIG_SPR0I, spr0_asserted);    // recieve
412     //interruptf(SIG_SPT0I, spt0_asserted);  // transmit
```

```
413
414     /* Set up Transmit Transfer Control Block (TCB) for chained DMA */
415     tx0_tcb[0].ii = tx0_buf[0];   // DMA source buffer address
416     tx0_tcb[1].ii = tx0_buf[1];   // DMA source buffer address
417
418     tx0_tcb[0].cp = &tx0_tcb[1].ii; // define ptr to next TCB (point to self)
419     tx0_tcb[1].cp = &tx0_tcb[0].ii; // define ptr to next TCB (point to self)
420
421     /* Set up Recieve Transfer Control Block (TCB) for chained DMA */
422     rx0_tcb[0].ii = rx0_buf[0];   // DMA source buffer address
423     rx0_tcb[1].ii = rx0_buf[1];   // DMA source buffer address
424
425     rx0_tcb[0].cp = &rx0_tcb[1].ii; // define ptr to next TCB (point to self)
426     rx0_tcb[1].cp = &rx0_tcb[0].ii; // define ptr to next TCB (point to self)
427
428     /* define ptr to current TCB (kick off DMA); SPORT0 transmit uses DMA ch 0 */
429     /* rx0_tcb[0] ist erster TCB, der aktiviert wird                  */
430     SetIOP(CP0, (((int)&rx0_tcb[0].ii) & CP_MAF) | CP_PCI);
431
432     /* define ptr to current TCB (kick off DMA); SPORT0 transmit uses DMA ch 2 */
433     /* tx0_tcb[0] ist erster TCB, der aktiviert wird                  */
434     SetIOP(CP2, (((int)&tx0_tcb[0].ii) & CP_MAF) | CP_PCI);
435 }
436
437 /*-----------------------------------------------------------------------------------*/
438
439 void setup_bitsibb( void ){
440
441     BITSIBB_TIM0  = BB_TIM(FC); // Sampling-Frequenz setzen
442     // BITSIBB_DIODAT = 0x5555;
443     // BITSIBB_DIODIR = 0xFF00; /* D15:8] OUT, D[7:0] IN */
444     BITSIBB_CTL   = 1;          /* Enable just one D/A */
445     BITSIBB_CLKSEL = 0x0844;    /* Ch 1-2 Tim 0, Ch 3-4 Tim 1 */
446 }
447
448 /*-----------------------------------------------------------------------------------*/
449 // init the buffers
450 void init_Buffers( void ){
451     int i, k;
452
453     for (i=0; i < TCB_BUFS; i++) {
454         for (k=0; k < FRAME_SIZE; k++) {
455             tx0_buf[i][k] = 0;
456             rx0_buf[i][k] = 0;
457         }
458     }
459
460     for (i=0; i < FRAME_SIZE; i++) {
461         //window[i] = 0.54 - 0.46 * cos(2*PI*i/FRAME_SIZE); // Hamming-Window
462         window[i] = sin(PI*i/FRAME_SIZE) * sin(PI*i/FRAME_SIZE); // Hanning-Window
463     }
464
465     for (i=0; i < FRAME_SIZE; i++) {
466         carframe[i] = 0.0;
467         zeros[i]   = 0.0;
468     }
469
470     for (i=0; i < P; i++) {
471         extwave[i] = 0.0;
472     }
473
474 }
475
476 /*-----------------------------------------------------------------------------------*/
477
478 void main ( void ){
479     // GET BASE LOCATION OF DMS3 (BITSI)
480     init_ms_bases();  // Initialize memory select bases
481
482     // set IOP registers
483     SetIOP(WAIT,0x21A4E421);
484
485     // Initialisierung Buffers
486     init_Buffers();
487
488     // initialize hardware
```

```
489     setup_bitsibb();
490     setup_sport0();
491
492     // Endlosschleife
493     while (1);
494     {
495         idle();        // ProgCounter vom Prozessor bleibt hier stehen, bis IRQ kommt
496     }
497 }
```

A.2.2 Vocoder 4

File: ROOT/imp/vocoder4/vocoder4.c

```
1
2  /****************************************************************************\
3  *                                                                            *
4  *  Vocoder (Blacktip PCI, Toolkit 5.0)                                       *
5  *  -----------------------------------                                       *
6  *                                                                            *
7  *   input CH1 : carrier (only if carint = 0)                                 *
8  *         CH2 : modulation signal (speech)                                   *
9  *   output CH1, CH2 : vocoder                                                *
10 *                                                                            *
11 *                                                                            *
12 *   Ver - Rev  History                                                       *
13 *   ---------  ----------------------------------------------------------   *
14 *   1.0 - 0    11.09.00 - initial coding (AR)                                *
15 *   1.1 - 01   05.02.00 - frame for generating the carrier intern            *
16 *   1.2        05.07.01 - pitch extraction and generation added (Andy Rohr) *
17 *                                                                            *
18 \****************************************************************************/
19
20
21 /*--------------------------------------------------------------------------*/
22 /* ADSP-21060 System Register bit definitions */
23 #include <def21060.h>       // register definitions
24 #include <21060.h>          // architecture
25 asm("#include <def21060.h>");
26 #include <signal.h>         // interrupt
27 #include <sport.h>          // serial port
28 #include <math.h>           // fmod
29 #include <stdlib.h>         // rand
30 #include <stats.h>          // autocorrelation
31 #include <trans.h>          // for fouriertransformations
32 asm("#include <asm_sprt.h>"); // ccall
33 #include "bitsibb.h"        // bitsi board
34 #include "bitsi.h"
35
36 #define CP_PCI 0x20000      /* Program-Controlled Interrupts bit */
37 #define CP_MAF 0x1ffff      /* Valid memory address field bits */
38
39 #define SetIOP(addr, val) (* (int *) addr) = (val)
40 #define GetIOP(addr)    (* (int *) addr)
41
42 #define PI      3.14159265358979
43
44 /*--------------------------------------------------------------------------*/
45 /* CH1: Carrier, CH2: Speech */
46
47 /* Settings for the vocoder */
48 #define M        256     /* frame size (M=2^x)              */
49 #define P 50             // Buffergroesse Extwave
50 asm("#define M   256");
51 #define BANDNR   38      /* Number of filterbands */
52 asm("#define BANDNR 38");
53 #define N        39      /* Num. of filtercoeff. for bandbassfilt. (odd) */
54 asm("#define N   39");
55 #define NHB      51      /* Num. of filtercoeff. for halfbandfilter */
56                          /* (NHB + 1)%4 = 0                          */
57 asm("#define NHB 51");
58 #define FC       22050.0 /* sampling frequency */
59 #define TC 1/FC          /* 1/(sampling frequency) */
```

```
60  #define FGMIN     180     /* cutoff freq. from lowpass (lowest filterbnd) */
61
62  /*------------------------------------------------------------------------*/
63
64  /* DMA chaining Transfer Control Blocks */
65  typedef struct {
66     unsigned pm ** cp;   /* Chain Pointer to next TCB */
67     unsigned  c;         /* Count register        */
68     int       im;        /* Index modifier register */
69     unsigned pm * ii;    /* Index register        */
70  } __tcb;
71
72  /* TCB's */
73  __tcb rx0_tcb[2] = { {0, M, 1, 0},  /* receive tcb */
74                      {0, M, 1, 0} };  /* receive tcb */
75
76  __tcb tx0_tcb[2] = { {0, M, 1, 0},  /* transmit tcb */
77                      {0, M, 1, 0} };  /* transmit tcb */
78
79  /* Necessary datas for the filters */
80  typedef struct {
81     float * inp;      // pointer to inputvalues of the filter
82     float * out;      // pointer to outputvalues of the filter
83     int siz;          // Filter length
84     int frmsize;      // frame size
85     float pm * pcoeff; // pointer to buffer with coefficients
86     float * pval;     // pointer to delayed values
87     float * pcurpos;  // pointer to current position in buffer for del. val.
88  } FILTERDAT;
89
90  /*------------------------------------------------------------------------*/
91
92  float weightout = 20.0;   /* weight outputvalue                */
93  float norm = 1 / 32767.0; /* normalization of sampled input values */
94  float corr = 32767.0;     /* denormalization for output values */
95  float unvoithres = 0.3;   /* if modulation signal has more than     */
96                            /* 100*unvoithres % zerocrossing -> sig. unvoic.*/
97
98  /* input and output buffer */
99  int pm inpbuf[2][M];      /* input buffer (data from ADC) */
100 int pm outbuf[2][M];      /* output buffer (data to ADC) */
101 int curbuf = 1;           /* current input and output buffer number */
102 float pm modframe[M];     /* input frame for modulationsignal    */
103 float pm carframe[M];     /* input frame for carriersignal       */
104 float outframe[M];        /* output frame before float2int and denorm */
105
106 /* calculated values for vocoder */
107 int hbnum;                /* Number of halfbandfilter */
108 int smpfact[BANDNR];      /* 2^smpfact = downsamplingfactor for each band */
109 int frmlengbp[BANDNR];    /* framelength for each bandpass           */
110 float invfrmlengbp[BANDNR]; /* invfrmlengbp[BANDNR] = 1 / frmlengbp  */
111 int * frmlenghb;          /* pointer to framelength for each halfband */
112 int * bpperhb;            /* pointer to number of bandpasses per halband */
113
114 /* different buffers */
115 float pm bpcoeff[BANDNR][N]; /* array for filtercoeff. for bandpassfilter */
116 float bpfiltermod[BANDNR][N];/* array for delayed values in bandpassfliter */
117                           /* for the modulation signal              */
118 float bpfiltercar[BANDNR][N];/* array for delayed values in bandpassfliter */
119                           /* for the carrier signal                 */
120 float pm hbcoeff[NHB];    /* array for filtercoeff. for halfbandfilter */
121 float * hbfiltdecmod;     /* pnt to array for del. val. for halfbandfilt. */
122                           /* for decimation for modulation signal   */
123 float * hbfiltdeccar;     /* pnt to array for del. val. for halfbandfilt. */
124                           /* for decimation for modulation signal   */
125 float * hbfiltint;        /* pnt to array for del. val. for halfbandfilt. */
126                           /* for interpolation                      */
127 FILTERDAT bpdatmod[BANDNR]; /* Datas for the bandpassfilters for mod.sig. */
128 FILTERDAT bpdatcar[BANDNR]; /* Datas for the bandpassfilters for car.sig. */
129 FILTERDAT * hbdatdecmod;  /* pnt to datas for halfbandfilters for   */
130                           /* decimation the modulation signal       */
131 FILTERDAT * hbdatdeccar;  /* pnt to datas for halfbandfilters for   */
132                           /* decimation the carrier signal          */
133 FILTERDAT * hbdatint;     /* pnt to datas for halfbandfilt. for interpol. */
134
135 float ** dwnspfrm;        /* pnt to pnts to array for downsampled frame */
```

```
136 float ** delaybuf;        /* pnt to pnts to array for delay the values, so*/
137                           /* that all outputs from halfbands are in phase */
138 float ** dlyinpadr;       /* pnt to pnts to the input of the delay arrays */
139 float upsptempout[M];     /* temp. buffer for upsampled values befor add. */
140 int * delays;             /* pnt to array with the different delays */
141 float * bpout[BANDNR];    /* array of pnts to output frames of bandpasses */
142 float weights[BANDNR];    /* weight for each band, calculated from  */
143                           /* modulation signal                      */
144 float tempbuf[M];         /* Buffer for temporary values            */
145 
146 /* variabeles for generating random numbers */
147 #define RANDBUFSIZ 17
148 asm("#define RANDBUFSIZ 17");
149 #define ROT1      5
150 asm("#define ROT1     5");
151 #define ROT2      3
152 asm("#define ROT2     3");
153 #define DIFJK     10
154 asm("#define DIFJK   10");
155 #define DIFJKNEG -10
156 asm("#define DIFJKNEG -10");
157 int randbuf[RANDBUFSIZ];   /* history buffer    */
158 float scalernd;            /* norm int to float */
159 int * curposrandbuf = randbuf;
160 
161 /* different variables */
162 int cycles;               /* measured clock cycles */
163 int filterdatsize;        /* sizeof(FILTERDAT)  */
164 int nounvoice = 0;        /* 0 : voiced/unvoiced analyses */
165                           /* 1 : no voiced/unvoiced analyses */
166 int carint = 1;           /* 1 : generate carrier in the programm */
167                           /* 0 : take carrier from Input CH1 */
168 
169 /* Variablen fuer Funktion calcpitch() */
170 float longmodframe[M*2];
171 float w_longmodframe[M*2];
172 float spec_re[M*2];
173 float spec_im[M*2];
174 float resultframe[M*2];
175 float dummyframe[M*2];
176 float spectrum[M*2];
177 float zeros[M*2];
178 float window[M*2];
179 
180 volatile int method = 2;
181 float max_i;
182 float max;
183 
184 /* Variablen fuer Funktion cargen(), Default des Ausgangssignals */
185 int wavetable = 1;           // Nummer des Wavetable, Selektor
186 float fs = 260.0;            // Signal-Frequenz in Hz
187 float amplitude = 1.0/3;     // Amplitude: 0..1, 0=>0Volt(min), 1=>3Volt(max)
188 int tastgrad = 20;           // Tastgrad in % (nur bei Rechteck)
189 float schritthoehe = 0.0;    // Schritthoehe (nur bei Saegezahn, Dreieck)
190 float x1 = 0.0;              // Schrittlaenge1 (nur bei Sinus)
191 float x2 = 0.0;              // Schrittlaenge2 (nur bei Sinus)
192 int countGen = 0;            // Zaehler fuer Takte pro Periode, cargen()
193 float extwave[P];            // Buffer fuer Wave-File-Klang
194 
195 /*----------------------------------------------------------------------*/
196 
197 /* sample receive irq */
198 asm("
199 .endseg;
200 .segment /pm seg_pmco;
201 
202 .global _spr0_asserted;
203 _spr0_asserted:
204 
205   r4=0xffffffff;
206   TCOUNT=r4;                 // initialize timer
207   BIT SET MODE2 TIMEN;       // start timer
208 
209   // change buffer
210   r2=dm(_curbuf);
211   r2=btgl r2 by 0;
```

```
212     dm(_curbuf)=r2;
213     // Calculate startaddress from current input buffer
214     r4=M;
215     r2=r2*r4 (SSI);
216     i9=r2;
217     modify(i9,_inpbuf);
218     // prepare input value
219     i10=_modframe;
220     i11=_carframe;
221     f8=dm(_norm);
222     r3=pm(i9,m14);               // Get value
223     r2=dm(_carint);              // generate carrier intern?
224     r2=pass r2;
225     if gt jump(pc,intcar);      // if (carint > 0) jump
226       r2=fext r3 by 0:16 (se); // convert 16 right bits in a 32 bit word
227       r4=fext r3 by 16:16 (se); // convert 16 left bits in a 32 bit word
228       lcntr=M, do prepinp1-1 until lce;
229         f4=float r4;           // convert int -> float
230         f2=float r2;
231         f2=f2*f8;              // normalize modulation value
232         f4=f4*f8,r3=pm(i9,m14); // normalize car.val., get next value
233         r2=fext r3 by 0:16 (se),pm(i10,m14)=f2; // conv. 16 right in 32 bit,..
234                                              // ..save modulation signal
235         r4=fext r3 by 16:16 (se),pm(i11,m14)=f4;// conv. 16 left in 32 bit,..
236                                              // ..save carrier signal
237       prepinp1:
238       jump(pc,vocoder);
239     intcar:                    // take intern carrier
240       r2=fext r3 by 0:16 (se); // convert 16 right bits in a 32 bit word
241       lcntr=M, do prepinp2-1 until lce;
242         f2=float r2;           // convert int -> float
243         f2=f2*f8,r3=pm(i9,m14); // normalize modulation val., get next value
244         r2=fext r3 by 0:16 (se),pm(i10,m14)=f2; // conv. 16 right in 32 bit,..
245                                              // ..save modulation signal
246       prepinp2:
247       /* calculate the pitch */
248       ccall(_calcpitch);       // Call C-Function calcpitch()
249       /* Generate carrier */
250       ccall(_cargen);          // Call C-Function cargen()
251     vocoder:
252
253   /* Register r2: Address of frame to downsample */
254   /* Register r4: Address of array of ptrs to downsampled frames */
255   /* Register r8: Address of array of FILTERDAT of decimation filter */
256
257     r2=_modframe;
258     r4=dm(_dwnspfrm);
259     r8=dm(_hbdatdecmod);
260     call(pc,_downsampling);
261
262   /* Register r2: Address of array of FILTERDAT of bandpass filter */
263
264     r2=_bpdatmod;
265     call(pc,_splittobands);
266
267   /* Calculate weights from modulation bands */
268     call(pc,_calcweights);
269
270   /* decide if modulation signal is voiced or unvoiced. If the signal is */
271   /* unvoiced, the carrierframe is filled with noise.           */
272
273     r2=dm(_nounvoice);
274     r2=pass r2;
275     if eq call(pc,_voiunvoi);
276
277   /* Register r2: Address of frame to downsample */
278   /* Register r4: Address of array of ptrs to downsampled frames */
279   /* Register r8: Address of array of FILTERDAT of decimation filter */
280
281     r2=_carframe;
282     r4=dm(_dwnspfrm);
283     r8=dm(_hbdatdeccar);
284     call(pc,_downsampling);
285
286   /* Register r2: Address of array of FILTERDAT of bandpass filter */
287     r2=_bpdatcar;
```

```
288	   call(pc,_splittobands);
289	
290	/* weight the different bands from the carrier signal */
291	   call(pc,_weightbands);
292	
293	/* Delay */
294	   call(pc,_delay);
295	
296	/* Add bandpass outputs per halfband */
297	   call(pc,_addhb);
298	
299	
300	/* Register r2: Address of array of ptrs to frames for upsamling*/
301	/* Register r4: Address of array of FILTERDAT of interpolation filter */
302	
303	   r2=dm(_delaybuf);
304	   r4=dm(_hbdatint);
305	   call(pc,_upsampling);
306	
307	   // Calculate startaddress from current output buffer
308	   r4=M;
309	   r2=dm(_curbuf);
310	   r2=r2*r4 (SSI);
311	   i9=r2;
312	   modify(i9,_outbuf);
313	   //prepare outputvalue
314	   i2=_outframe;
315	   f8=dm(_corr);
316	   f2=dm(i2,m6);              // get next output value
317	   f2=f2*f8;                  // denormalize output value
318	   lcntr=M, do prepout-1 until lce;
319	      r2=trunc f2;            // convert float -> int
320	      r4=fdep r2 by 0:16;     // convert 32 bit word to 16 right bit
321	      r4=r4 or fdep r2 by 16:16,f2=dm(i2,m6); // conv. 32 to 16 left bit,..
322	                                             // ..get next ouput value
323	      f2=f2*f8,pm(i9,m14)=r4; // denormalize output value, write to outp.
324	   prepout:
325	
326	   BIT CLR MODE2 TIMEN;       // stop timer
327	   rti (DB);
328	   r4=TCOUNT;                 // read timer
329	   dm(_cycles)=r4;
330	
331	");
332	
333	/*-------------------------------------------------------------------------*/
334	
335	void setup_sport0( void )
336	{
337	   /* Configure SHARC serial port */
338	
339	   /* TRANSMIT CONTROL REGISTER */
340	   /* An alternate (and more efficient) way of doing this would be to */
341	   /* write the 32-bit register all at once with a statement like this: */
342	   /*       SetIOP(STCTL0, 0x001c00f2);                          */
343	   /* But the following is more descriptive...                   */
344	
345	   sport0_iop.txc.mdf = 0;   /* multichannel frame delay (MFD)  */
346	   sport0_iop.txc.schen = 1; /* Tx DMA chaining enable          */
347	   sport0_iop.txc.sden = 1;  /* Tx DMA enable                   */
348	   sport0_iop.txc.lafs = 0;  /* Late TFS (alternate)            */
349	   sport0_iop.txc.ltfs = 1;  /* Active low TFS                  */
350	   sport0_iop.txc.ditfs = 0; /* Data independent TFS            */
351	   sport0_iop.txc.itfs = 0;  /* Internally generated TFS        */
352	   sport0_iop.txc.tfsr = 1;  /* TFS Required                    */
353	
354	   sport0_iop.txc.ckre = 0;  /* Data and FS on clock rising edge */
355	   sport0_iop.txc.gclk = 0;  /* Enable clock only during transmission*/
356	   sport0_iop.txc.iclk = 0;  /* Internally generated Tx clock   */
357	   sport0_iop.txc.pack = 0;  /* Unpack 32b words into two 16b tx's */
358	
359	   sport0_iop.txc.slen = 31; /* Data word length minus one      */
360	   sport0_iop.txc.sendn = 0; /* Data word endian 1 = LSB first  */
361	   sport0_iop.txc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
362	                             /* Data type specifier             */
363	   sport0_iop.txc.spen = 1;  /* Enable (clear for MC operation) */
```

```
364
365     /* RECEIVE CONTROL REGISTER */
366     sport0_iop.rxc.nch = 31;  /* multichannel number of channels - 1 */
367     sport0_iop.rxc.mce = 0;   /* multichannel enable              */
368     sport0_iop.rxc.spl = 0;   /* Loop back configure (test)       */
369     sport0_iop.rxc.d2dma = 0; /* Enable 2-dimensional DMA array   */
370     sport0_iop.rxc.schen = 1; /* Rx DMA chaining enable           */
371     sport0_iop.rxc.sden = 1;  /* Rx DMA enable                    */
372     sport0_iop.rxc.lafs = 0;  /* Late RFS (alternate)             */
373     sport0_iop.rxc.ltfs = 1;  /* Active low RFS                   */
374     sport0_iop.rxc.irfs = 0;  /* Internally generated RFS         */
375     sport0_iop.rxc.rfsr = 1;  /* RFS Required                     */
376     sport0_iop.rxc.ckre = 0;  /* Data and FS on clock rising edge */
377     sport0_iop.rxc.gclk = 0;  /* Enable clock only during transmission*/
378     sport0_iop.rxc.iclk = 0;  /* Internally generated Rx clock    */
379     sport0_iop.rxc.pack = 0;  /* Pack two 16b rx's into 32b word */
380
381     sport0_iop.rxc.slen = 31; /* Data word length minus one       */
382     sport0_iop.rxc.sendn = 0; /* Data word endian 1 = LSB first   */
383     sport0_iop.rxc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
384                               /* Data type specifier              */
385     sport0_iop.rxc.spen = 1;  /* Enable (clear for MC operation) */
386
387     /* Set up Transmit Transfer Control Block for chained DMA */
388     tx0_tcb[0].ii = outbuf[0]; /* DMA source buffer address           */
389     tx0_tcb[1].ii = outbuf[1]; /* DMA source buffer address           */
390
391     tx0_tcb[0].cp = &tx0_tcb[1].ii;/* define ptr to next TCB (point to self) */
392     tx0_tcb[1].cp = &tx0_tcb[0].ii;/* define ptr to next TCB (point to self) */
393
394     /* Set up Receive Transfer Control Block for chained DMA */
395     rx0_tcb[0].ii = inpbuf[0]; /* DMA destination buffer address      */
396     rx0_tcb[1].ii = inpbuf[1]; /* DMA destination buffer address      */
397
398     rx0_tcb[0].cp = &rx0_tcb[1].ii;/* define ptr to next TCB (point to self) */
399     rx0_tcb[1].cp = &rx0_tcb[0].ii;/* define ptr to next TCB (point to self) */
400
401     SetIOP(CP0, (((int)&rx0_tcb[0].ii) & CP_MAF) | CP_PCI);
402                               /* define ptr to current TCB (kick off DMA) */
403                               /* (SPORT0 receive uses DMA ch 0)      */
404
405     SetIOP(CP2, (((int)&tx0_tcb[0].ii) & CP_MAF) | CP_PCI);
406                               /* define ptr to current TCB (kick off DMA) */
407                               /* (SPORT0 transmit uses DMA ch 2)     */
408
409     /* The name of the interruptfunction is declared in the assembler file
410        voco_hdr.asm. This file must be compiled to the file 060_hdr.obj,
411        which contains the runtime header.                         */
412   }
413   /*------------------------------------------------------------------------*/
414
415   void setup_bitsibb( void )
416   {
417
418     BITSIBB_TIM0  = BB_TIM(FC);
419     // BITSIBB_DIODAT = 0x5555;
420     // BITSIBB_DIODIR = 0xFF00; /* D15:8] OUT, D[7:0] IN */
421     BITSIBB_CTL   = 1;        /* Enable just one D/A */
422     BITSIBB_CLKSEL = 0x0844;  /* Ch 1-2 Tim 0, Ch 3-4 Tim 1 */
423   }
424
425   /*------------------------------------------------------------------------*/
426
427   void filterdesign( )
428   // Prepare filter
429   {
430     int i, j, k;
431     float temp;
432     float infKc;                // infKc = 1/Kc = fg/fc
433     int nhb05;                  // nhb05 = (NHB-1)/2
434     int n05;                    // n05 = (N-1)/2
435     float infnhb;               // infnhb = 1/(NHB-1)
436     float infn;                 // infn = 1/(N-1)
437     float infFC = 1/FC;
438     float df;                   // bandwith/fc*2 (logarithmic)
439     float fg[BANDNR][2];        // cutoff frequency bandpass [low; high]
```

```
440    float bndwidth, fm;          // bandwith, middle freq. from bandpass
441
442    nhb05 = (NHB - 1) / 2;
443    n05 = (N - 1) / 2;
444    infnhb = 1 / (float) (NHB-1);
445    infn = 1 / (float) (N-1);
446
447    // Halfbandfilter
448    //----------------
449
450    // Calculate filtercoefficients
451    for (i = 0; i < NHB; i++)
452    {
453       hbcoeff[i] = 2 * 0.25 * sin(2*PI*0.25*(i-nhb05))/(2*PI*0.25*(i-nhb05))
454                  * (0.54 - 0.46 * cos(2*PI*i*infnhb)); // Hammingwindow
455    }
456    hbcoeff[nhb05] = 2 * 0.25;
457
458    hbnum = (int) floorf(-logf(FGMIN/FC*2)/logf(2)); // Number of halfbandfilt.
459
460    // allocate Memory for FIR-Datas for the halfbandfilter
461    hbdatdecmod = (FILTERDAT *) calloc(sizeof(FILTERDAT)*hbnum, 1);
462    hbdatdeccar = (FILTERDAT *) calloc(sizeof(FILTERDAT)*hbnum, 1);
463    hbdatint = (FILTERDAT *) calloc(sizeof(FILTERDAT)*hbnum, 1);
464
465    // allocate Memory for delayed values from halfbandfilter
466    hbfiltdecmod = (float *) calloc(hbnum*NHB, 1); // filter for decimation
467    hbfiltdeccar = (float *) calloc(hbnum*NHB, 1); // filter for decimation
468    hbfiltint = (float *) calloc(hbnum*NHB, 1);  // filter for interpol.
469
470    // Calculate framelength for each halfband
471    frmlenghb = (int *) calloc(hbnum+1, 1);   // allocate Memory
472    for (i = 0; i < hbnum+1; i++)
473       frmlenghb[i] = M * pow(2, -i);
474
475    // Initialize Datas for halfbandfilters
476    for (i = 0; i < hbnum; i++)
477    {
478       hbdatdecmod[i].siz = NHB;
479       hbdatdeccar[i].siz = NHB;
480       hbdatint[i].siz = NHB;
481       hbdatdecmod[i].frmsize = frmlenghb[i+1]; // after downsampling
482       hbdatdeccar[i].frmsize = frmlenghb[i+1];
483       hbdatint[i].frmsize = frmlenghb[i+1]; // before upsampling
484       hbdatdecmod[i].pcoeff = hbcoeff;
485       hbdatdeccar[i].pcoeff = hbcoeff;
486       hbdatint[i].pcoeff = hbcoeff;
487       hbdatdecmod[i].pval = hbfiltdecmod + i * NHB;
488       hbdatdeccar[i].pval = hbfiltdeccar + i * NHB;
489       hbdatint[i].pval = hbfiltint + i * NHB;
490       hbdatdecmod[i].pcurpos = hbfiltdecmod + i * NHB;
491       hbdatdeccar[i].pcurpos = hbfiltdeccar + i * NHB;
492       hbdatint[i].pcurpos = hbfiltint + i * NHB;
493    }
494
495    // allocate Memory for value for delays
496    delays = (int *) calloc(hbnum+1, 1);
497    // calculate delay for the halfbands, so that the signals after upsampling
498    // and before addition have the same delays
499    for (j = 0; j < hbnum+1; j++)
500    {
501       temp = 0.0;
502       for (i = 1; i <= hbnum-j; i++)
503          temp += powf(2.0, i);
504       delays[j] = 1+nhb05*temp + n05*(powf(2.0, hbnum-j)-1);
505     }
506
507    // Bandpassfilter
508    //---------------
509
510    // Calculate cutoff frequencies
511    df = -log10(FGMIN/FC*2)/(BANDNR-1);     // bandwith/fs*2 (logarithmic)
512    for (i = 0; i < BANDNR; i++)            // cutoff frequencies bandpass
513    {                                       // fg/fc
514       fg[i][0] = 0.5 * powf(10.0, -df*(i+1)); // low cutoff frequency
515       fg[i][1] = 0.5 * powf(10.0, -df*i);  // high cutoff frequency
```

```
516     }
517
518     // allocate Memory for number of bandpasses per halfband
519     bpperhb = (int *) calloc(hbnum+1, 1);
520     // Calculate smpfact for each filterband (2^smpfact = downsamplefactor)
521     j = 0;
522     for (i = 0; i <= hbnum; i++)
523     {
524        k = 0;
525        while ((fg[j][1] > 0.5 * powf(2.0, -(1+i))) && (j < BANDNR))
526        {
527           smpfact[j] = i;
528           j++;                        // next frequenzband
529           k++;
530        }
531        bpperhb[i] = k;                // num. of bandpasses per halfband
532     }
533
534     // Calculate framelength for each bandpass
535     for (i = 0; i < BANDNR; i++)
536     {
537        frmlengbp[i] = M * pow(2, -smpfact[i]);
538        invfrmlengbp[i] = 1.0 / frmlengbp[i];
539     }
540
541     // Calculate coefficients for highpassfilter (hightest band)
542     infKc = 0.5 - fg[0][0];                 // infKc = 1/Kc = fgTP/fc
543     for (i = 0; i < N; i++)
544     {
545        bpcoeff[0][i] = 2 * infKc * sin(2*PI*infKc*(i-n05))/(2*PI*infKc*(i-n05))
546                      * powf(-1.0, i-n05)              // TP -> HP
547                      * (0.54 - 0.46 * cos(2*PI*i*infn)); // Hammingwindow
548     }
549     bpcoeff[0][n05] = 2 * infKc;
550
551     // Calculate coefficients for bandpassfilter
552     for (j = 1; j < BANDNR-1; j++)
553     {
554        bndwidth = (fg[j][1] - fg[j][0]) * pow(2, smpfact[j]); // Bandwidht
555        fm = (fg[j][0] + fg[j][1]) * pow(2, smpfact[j]); // middle freq.
556        infKc = bndwidth * 0.5;                          // 1/Kc = fgTP/fc
557        for (i = 0; i < N; i++)
558        {
559           bpcoeff[j][i] = 2 * infKc * sin(2*PI*infKc*(i-n05))
560                             / (2*PI*infKc*(i-n05))
561                           * 2 * cos(2*PI*fm*0.5*(i-n05))  // TP -> BP
562                           * (0.54 - 0.46 * cos(2*PI*i*infn)); // Hammingwindow
563        }
564        bpcoeff[j][n05] = 2 * infKc * 2;
565     }
566
567     // Calculate coefficients for lowpassfilter (lowest band)
568     infKc = fg[BANDNR-1][1] * pow(2, smpfact[j]);    // 1/Kc = fg/fc
569     for (i = 0; i < N; i++)
570     {
571        bpcoeff[BANDNR-1][i] = 2 * infKc * sin(2*PI*infKc*(i-n05))
572                                  / (2*PI*infKc*(i-n05))
573                           * (0.54 - 0.46 * cos(2*PI*i*infn)); // Hammingwindow
574     }
575     bpcoeff[BANDNR-1][n05] = 2 * infKc;
576
577     // Initialize Datas for bandpassfilters
578     for (i = 0; i < BANDNR; i++)
579     {
580        bpdatmod[i].siz = N;
581        bpdatcar[i].siz = N;
582        bpdatmod[i].frmsize = frmlengbp[i];
583        bpdatcar[i].frmsize = frmlengbp[i];
584        bpdatmod[i].pcoeff = bpcoeff[i];
585        bpdatcar[i].pcoeff = bpcoeff[i];
586        bpdatmod[i].pval = bpfiltermod[i];
587        bpdatcar[i].pval = bpfiltercar[i];
588        bpdatmod[i].pcurpos = bpfiltermod[i];
589        bpdatcar[i].pcurpos = bpfiltercar[i];
590     }
591  }
```

```
592
593 /*-------------------------------------------------------------------------*/
594
595 void allocatbuffers( )
596 // Prepare different buffer
597 {
598    int i;
599
600    // alloc mem for ptrs to frames after downsampling
601    dwnspfrm = (float**) malloc(hbnum+1);
602    // alloc mem for frames of modulation signal after downsampling
603    for (i = 0; i < hbnum+1; i++)
604       dwnspfrm[i] = (float *) malloc(frmlenghb[i]);
605
606    // alloc mem for ptrs to delay buffers
607    delaybuf = (float**) malloc(hbnum+1);
608    // alloc mem for delay buffers
609    for (i = 0; i < hbnum+1; i++)
610       delaybuf[i] = (float *) malloc(frmlenghb[i]+delays[i]);
611
612    // alloc mem for ptrs to the input of delay buffers
613    dlyinpadr = (float**) malloc(hbnum+1);
614
615    // alloc mem for frames after bandpassfiltering
616    for (i = 0; i < BANDNR; i++)
617       bpout[i] = (float *) malloc(frmlengbp[i]);
618
619 }
620 /*-------------------------------------------------------------------------*/
621
622 void setinoutbuffilterdat( )
623 // Set pointer to input- and outputbuffer in FILTERDAT
624 {
625    int i, temp;
626
627    filterdatsize = sizeof(FILTERDAT);
628
629    // Set pointer to input- and outputbuffer in FILTERDAT of decimationsfilter
630    // for modulation signal
631    for (i=0; i < hbnum; i++)
632    {
633       hbdatdecmod[i].inp = dwnspfrm[i];
634       hbdatdecmod[i].out = dwnspfrm[i+1];
635    }
636
637    // Set pointer to input- and outputbuffer in FILTERDAT of decimationsfilter
638    // for carrier signal
639    for (i=0; i < hbnum; i++)
640    {
641       hbdatdeccar[i].inp = dwnspfrm[i];
642       hbdatdeccar[i].out = dwnspfrm[i+1];
643    }
644
645    // Set pointer to input- and outputbuffer in FILTERDAT of interpolation-
646    // filter
647    for (i=0; i < hbnum; i++)
648    {
649       hbdatint[i].inp = outframe;
650       hbdatint[i].out = upsptempout;
651    }
652
653    // Set pointer to input- and outputbuffer in FILTERDAT of bandpassfilter
654    for (i=0; i < BANDNR; i++)
655    {
656       temp = smpfact[i];
657       bpdatmod[i].inp = dwnspfrm[temp];
658       bpdatcar[i].inp = dwnspfrm[temp];
659       bpdatmod[i].out = bpout[i];
660       bpdatcar[i].out = bpout[i];
661    }
662 }
663 /*-------------------------------------------------------------------------*/
664
665 void setdlyinp( )
666 // Set pointer to input for delays
667 {
```

```
668      int i;
669
670      for (i = 0; i<hbnum+1 ; i++)
671         dlyinpadr[i] = delaybuf[i] + delays[i];
672   }
673   /*-------------------------------------------------------------------------*/
674
675   void initrandbuf( )
676   // initialize randbuf
677   {
678      int i;
679      int seed = 12345678;
680
681      for (i=0; i<RANDBUFSIZ; i++)
682      {
683         randbuf[i] = seed;
684         asm("%0=rot %1 by 5;" : "=d" (seed) : "d" (seed));
685         seed += 97;
686      }
687
688      scalernd = 1.0 / (0xffffffff-1) / weightout * 10.0;
689   }
690   /*-------------------------------------------------------------------------*/
691
692   void initcalcpitch()
693   // initialize things for calculating the pitch
694   {
695       int i, k;
696
697       // fill window and zeros with lenght 2*M
698       for (i=0; i < 2*M; i++) {
699           k = i * 1.0;
700           //window[i] = 0.54 - 0.46 * cos(2*PI*k/(2*M-1)); // Hamming Window
701           window[i] = sin(PI*k/(2*M-1)) * sin(PI*k/(2*M-1)) ; // Hanning Window
702           zeros[i] = 0.0;
703       }
704   }
705   /*-------------------------------------------------------------------------*/
706
707   /* fill the carrierframe with noise                              */
708   asm("
709   .endseg;
710   .segment /pm seg_pmco;
711
712   .global _makerandinp;
713   _makerandinp:
714
715   // used registers: i1,i9
716   //              m1,m2
717   //              r2,r4,r8
718
719      // Calculate random values:
720      // X(n)=(rot X(n-j) by ROT1) + (rot X(n-k) by ROT2)
721      // for details look at http://www.agner.org/random
722      i9=_carframe;           // address of carrier frame
723      b1=_randbuf;            // set buffer as circular buffer
724      l1=RANDBUFSIZ;
725      i1=dm(_curposrandbuf);  // adr. of current position in randbuf
726      m1=DIFJK;
727      m2=DIFJKNEG;
728      f8=dm(_scalernd);
729
730      lcntr=M, do fillrand-1 until lce;
731         r2=dm(i1,m2);        // get X(n-j) and jump to X(n-k)
732         r2=rot r2 by ROT1, r4=dm(i1,m1); // rotate X(n-j) by ROT1, ...
733                                  // get X(n-k) and jump to X(n-j)
734         r4=rot r4 by ROT2;   // rotate X(n-k) by ROT2
735         r2=r2+r4;            // (rot X(n-j) by ROT1)+(rot X(n-k) by ROT2)
736         f2=float r2,dm(i1,m6)=r2; // save value in X(n-j), int -> float
737         f2=f2*f8;            // scale to -1 ... 1
738         pm(i9,m14)=f2;        // save value in carrier frame
739      fillrand:
740      dm(_curposrandbuf)=i1;  // save current position in randbuf
741      l1=0;                   // reset circular buffer
742      rts;
743   ");
```

```
744  /*-------------------------------------------------------------------------*/
745
746  /* decide if modulation signal is voiced or unvoiced. If the signal is */
747  /* unvoiced, the carrierframe is filled with noise.             */
748  asm("
749  .endseg;
750  .segment /pm seg_pmco;
751
752  .global _voiunvoi;
753  _voiunvoi:
754
755  // used registers: i1,i2,i9
756  //
757  //                r2,r4
758
759    // voiced or unvoiced ?
760    i9=_modframe;
761    i2=_tempbuf;
762    f4=1.0;
763
764    lcntr=M, do findsign-1 until lce;  // determine sign
765      f2=pm(i9,m14);                    // get modulation value
766      f2=f4 copysign f2;               // if (f2>0) f2=1 else -1
767      dm(i2,m6)=f2;                    // save sign
768    findsign:
769
770    i1=_tempbuf;
771    i2=_tempbuf+1;
772    f8=0.0;                            // sum = 0;
773    // find signchanges if (signchange) res=-1 else res=1
774    // and then calculate sum(res)
775    lcntr=M-1, do signchanges-1 until lce;
776      f2=dm(i1,m6);                    // get n. value
777      f4=dm(i2,m6);                    // get n+1. value
778      f2=f2*f4;                        // (n. val) * (n+1 val)
779      f8=f8+f2;                        // sum = sum + (n. val) * (n+1 val)
780    signchanges:
781
782    // calculate number of signchanges
783    r2=M-1;
784    f2=float r2;
785    f2=f2-f8;                         // M - 1 - sum
786    f4=0.5;
787    f2=f2*f4;                         // changnum = (M - 1 - sum) / 2
788    f4=dm(_invfrmlengbp);             // f2=1 / M
789    f2=f2*f4;                         // f2=changnum / M
790
791    f4=dm(_unvoithres);
792    comp(f2,f4);                      // changnum / M > unvoithres ?
793    if gt call(pc,_makerandinp);      // fill the carrierframe with noise
794    rts;
795  ");
796  /*-------------------------------------------------------------------------*/
797
798  /* weight the different bands from the carrier signal */
799  asm("
800  .endseg;
801  .segment /pm seg_pmco;
802
803  .global _weightbands;
804  _weightbands:
805
806  // used registers: i0,i1,i2,i4
807  //
808  //                r2,r4
809
810    i0=_weights;                       // adr. of array with weights
811    i1=_bpout;                        // adr. of array with pnt to bp-outp.
812    i2=_frmlengbp;                     // adr. of array with frame length
813
814    lcntr=BANDNR, do weightband-1 until lce;
815      r2=dm(i1,m6);                    // adr. of current bp-output
816      i4=r2;
817      f2=dm(i0,m6);                    // get current weight
818  //f2=0.1; // all weight const.
819      r4=dm(i2,m6);                    // get current framelength
```

```
820        lcntr=r4, do weightval-1 until lce;
821           f4=dm(i4,m5);                // get bandpass value
822           f4=f4*f2;                    // weight value
823           dm(i4,m6)=f4;                // save value
824        weightval:
825        nop;
826     weightband:
827
828     rts;
829  ");
830  /*-------------------------------------------------------------------------*/
831
832  /* calculate weight for each band from the bands of the modulation signal */
833  asm("
834  .endseg;
835  .segment /pm seg_pmco;
836
837  .global _calcweights;
838  _calcweights:
839
840  // used registers: i0,i1,i2,i3,i4
841  //
842  //               r0,r1,r2,r3,r4,r12,r8
843
844     i1=_bpout;                        // adr. of array with pnt to bp-outp.
845     i0=_weights;                      // adr. of array with weights
846     i2=_frmlengbp;                    // adr. of array with frame length
847     i3=_invfrmlengbp;                 // adr. of array with 1/frame length
848     f8=3.0;                           // for calculate square root
849     f1=0.5;                           // for calculate square root
850
851     lcntr=BANDNR, do calcweight-1 until lce;
852        r2=dm(i1,m6);                  // adr. of current bp-output
853        i4=r2;
854        r4=dm(i2,m6);                  // get current framelength
855        f12=0.0;                       // weight = 0
856        f2=dm(i4,m6);                  // get value
857        lcntr=r4, do sumupweight-1 until lce;
858           f2=f2*f2;                   // value^2
859           f12=f12+f2,f2=dm(i4,m6);    // weight=weight+value^2, get val.
860        sumupweight:
861        f2=dm(i3,m6);                  // get 1/(frame length)
862        f3=f12*f2;                     // weight = weight/(frame length)
863
864        f0=f3;
865        // Calculate square root (according to ADSP-2106x User's manual
866        // page B-40)
867        // input f0, output f3
868        f4=rsqrts f0;                  // Fetch 4-bit seed
869        f12=f4*f4;                     // f12=x0^2
870        f12=f12*f0;                    // f12=c*x0^2
871        f4=f1*f4,f12=f8-f12;           // f4=.5*x0, f12=3-c*x0^2
872        f4=f4*f12;                     // f4=x1=.5*x0(3-c*x0^2)
873        f12=f4*f4;                     // f12=x1^2
874        f12=f12*f0;                    // f12=c*x1^2
875        f4=f1*f4,f12=f8-f12;           // f4=.5*x1, f12=3-c*x1^2
876        f4=f4*f12;                     //f4=x2=.5*x1(3-c*x1^2)
877        f3=f3*f4;                      // sqrt(inp)=inp/sqrt(inp)
878
879        dm(i0,m6)=f3;                  // save weight
880     calcweight:
881
882     rts;
883  ");
884  /*-------------------------------------------------------------------------*/
885
886  /* add the bands which lay in the same halfband */
887  asm("
888  .endseg;
889  .segment /pm seg_pmco;
890
891  .global _addhb;
892  _addhb:
893
894  // used registers: i0,i1,i2,i3,i4,i5
895  //
```

```
896 //               r1,r2,r3,r4,r8
897
898    i1=_bpout;                        // adr. of array with pnt to bp-outp.
899    i2=dm(_dlyinpadr);                // adr. of array with pnt to dlybufinp.
900    i0=dm(_frmlenghb);                // adr. of array with frame length
901    i4=dm(_bpperhb);                  // adr. of array with num. of bp per hb
902    r2=dm(_hbnum);
903    r2=r2+1;
904    lcntr=r2, do adddifhb-1 until lce;
905       r8=dm(i0,m6);                  // current framelength
906       r2=dm(i1,m6);                  // adr. of current bandpass output
907       i5=r2;
908       r1=dm(i2,m6);                  // adr. of current delay buffer input
909       i3=r1;
910       lcntr=r8,do copybpout2dly-1 until lce;
911          f2=dm(i5,m6);               // get bandpass outputval
912          dm(i3,m6)=f2;               // write to delaybuffer input
913       copybpout2dly:
914
915       r3=dm(i4,m6);                  // currrent number of bp per hb
916       r3=r3-1;
917       lcntr=r3, do addcurhb-1 until lce;
918          r2=dm(i1,m6);               // adr. of current bandpass output
919          i5=r2;
920          i3=r1;                      // adr. of current delay buffer input
921          f4=dm(i5,m6);               // get bp value
922          lcntr=r8,do addup-1 until lce;
923             f2=dm(i3,m5);            // get old value
924             f2=f2+f4,f4=dm(i5,m6);   // add, get bp value
925             dm(i3,m6)=f2;            // save sum
926          addup:
927          nop;
928       addcurhb:
929       nop;
930    adddifhb:
931    rts;
932 ");
933 /*-------------------------------------------------------------------------*/
934
935 /* shift the old values in the delay buffer */
936 asm("
937 .endseg;
938 .segment /pm seg_pmco;
939
940 .global _delay;
941 _delay:
942
943 // used registers: i0,i1,i2,i3
944 //               m2,m3
945 //               r2,r4
946
947 // delay um 1 falsch??? Schlaufenlängen korrekt??
948 // wird der richtige Bereich und die richtige Anzahl Werte geschoben???
949 //
950    i1=dm(_delays);                    // adr. of delay lengths
951    i2=dm(_delaybuf);                  // adr. of array with pnt to delaybuf.
952    i3=dm(_frmlenghb);                 // adr. of array with framelength
953    r2=dm(_hbnum);
954    r2=r2+1;
955    lcntr=r2, do shiftdly-1 until lce;
956       r2=dm(i3,m6);                   // get current frame length
957       m2=r2;                          // m2 = frame length
958       r4=dm(i2,m6);                   // adr. of current delaybuffer
959       i0=r4;
960       modify(i0,m2);                  // adr. of first data to copy
961       r4=-r2;
962       m3=r4;                          // m3 =- frame length
963       r4=r2+1;
964       m2=r4;                          // m2 = frame length + 1
965       r2=dm(i1,m6);                   // delay length
966       lcntr=r2, do copydly-1 until lce;
967          f4=dm(i0,m3);                // get value and jump to dest. pos.
968          dm(i0,m2)=f4;                // wrt val. and jump to nxt source pos
969       copydly:
970       nop;                            // otherwise problem
971    shiftdly:
```

```
972     rts;
973  ");
974  /*--------------------------------------------------------------------------*/
975
976  /* split the signal after downsapmling in bands */
977  /* Register r2: Address of array of FILTERDAT of bandpass filter */
978  /* Remark: address of input- and outputbuffers in the FILTERDAT of filters */
979  /* are set in function setinoutbuffilterdat() */
980  asm("
981  .endseg;
982  .segment /pm seg_pmco;
983
984  .global _splittobands;
985  _splittobands:
986
987     lcntr=BANDNR, do split-1 until lce;
988        call(pc,_FIR_Filter) (DB);
989        dm(i7,m7)=r2;                // put addr. of cur. filterdat on stack
990                                     // -> (save r2)
991        nop;
992        modify(i7,m6);
993        r2=dm(i7,m5);                // get addr. from stack
994        r4=dm(_filterdatsize);       // sizeof(FILTERDAT)
995        r2=r2+r4;                    // address of next filterdat
996     split:
997
998     rts;
999  ");
1000 /*--------------------------------------------------------------------------*/
1001
1002 /* downsampling */
1003 /* Register r2: Address of frame to downsample */
1004 /* Register r4: Address of array of ptrs to downsampled frames */
1005 /* Register r8: Address of array of FILTERDAT of decimation filter */
1006 /* Remark: address of input- and outputbuffers in the FILTERDAT of filters */
1007 /* are set in function setinoutbuffilterdat() */
1008 asm("
1009 .endseg;
1010 .segment /pm seg_pmco;
1011
1012 .global _downsampling;
1013 _downsampling:
1014
1015    i9=r2;                          // adr. of frame to downsample
1016    i2=r4;                          // adr. of pointers to buffers
1017    r2=dm(i2,m6);                   // get address of first buffer
1018    i3=r2;                          // address of first buffer
1019    lcntr=M, do copyinpbuf-1 until lce; // copy inp. signal to first buffer
1020       f2=pm(i9,m14);
1021       dm(i3,m6)=f2;
1022    copyinpbuf:
1023    r4=dm(_hbnum);                  // number of dec. filter
1024    r2=r8;                          // address of 1. filterdat
1025    lcntr=r4, do downsample-1 until lce;
1026       call(pc,_downsampby2) (DB);
1027       dm(i7,m7)=r2;                // put addr. of cur. filterdat on stack
1028                                    // -> (save r2)
1029       nop;
1030       modify(i7,m6);
1031       r2=dm(i7,m5);                // get addr. from stack
1032       r4=dm(_filterdatsize);       // sizeof(FILTERDAT)
1033       r2=r2+r4;                    // address of next filterdat
1034    downsample:
1035
1036    rts;
1037 ");
1038
1039 /*--------------------------------------------------------------------------*/
1040
1041 /* upsampling */
1042 /* Register r2: Address of array of ptrs to frames for upsamling*/
1043 /* Register r4: Address of array of FILTERDAT of interpolation filter */
1044 /* Result of upsampling saved in array outframe.                */
1045 /* Remark: address of input- and outputbuffers in the FILTERDAT of filters */
1046 /* are set in function setinoutbuffilterdat() */
1047 asm("
```

```
1048 .endseg;
1049 .segment /pm seg_pmco;
1050
1051 .global _upsampling;
1052 _upsampling:
1053
1054 // used registers: i0,i1,i2,i3,i4
1055 //              m2
1056 //              r2,r3,r4,r8
1057
1058    i1=r2;                         // address of array of ptrs to inp.frames
1059    i2=r4;                         // adddress of array of FILTERDAT
1060    r2=dm(_hbnum);                 // number of inpterpolation filter
1061    m2=r2;                         // m2=hbnum
1062    modify(i1,m2);                 // addr. of ptr to array of last inpfrm
1063    r3=dm(_filterdatsize);         // sizeof(FILTERDAT)
1064    r2=r2-1;                       // hbnum-1
1065    r2=r2*r3 (SSI);                // sizeof(FILTERDAT)*(hbnum-1)
1066    r2=r4+r2;                      // addr. of last filterdat
1067    i3=_upsptempout;
1068    f4=0.0;
1069    lcntr=M, do clrout-1 until lce; // clear array upsptempout
1070       dm(i3,m6)=f4;
1071    clrout:
1072
1073    r4=dm(_frmlenghb);            // get addr. of array of framelength
1074    i4=r4;                        // addr. of array of framelength
1075    modify(i4,m2);                // addr. of framelength of last filter
1076    r4=dm(_hbnum);
1077    lcntr=r4, do upsample-1 until lce;
1078       // add current inputframe to current outputframe
1079       r4=dm(i4,m7);              // get current framelength
1080       r8=dm(i1,m7);              // get addr. of current inputframe
1081       i0=r8;                     // addresse of current inputframe
1082       i2=_upsptempout;           // addr. of result of previous upsampling
1083       i3=_outframe;              // addr. of outframe
1084       lcntr=r4, do addinpfrm-1 until lce;
1085          f4=dm(i2,m6);           // get upsptempout
1086          f8=dm(i0,m6);           // get inputvalue
1087          f4=f4+f8;               // inputvalue + upsptempout
1088          dm(i3,m6)=f4;           // outputval = inputval + upsptempout
1089       addinpfrm:
1090       // upsampling
1091       dm(i7,m7)=r2;             // save addr. of current filterdat on stack
1092       r4=i1;
1093       dm(i7,m7)=r4;             // save addr. of ptr to array of inpfrm
1094       r4=i4;
1095       dm(i7,m7)=r4;             // save current addr. of framelength
1096       call(pc,_upsampby2);
1097       modify(i7,m6);            // get values from stack
1098       r4=dm(i7,m6);
1099       i4=r4;                    // get current addr. of framelength
1100       r4=dm(i7,m6);
1101       i1=r4;                    // get addr. of ptr to array of inpfrm
1102       r2=dm(i7,m5);             // get addr. of cur. filterdat from stack
1103       r4=dm(_filterdatsize);
1104       r2=r2-r4;                 // address of previous filterdat
1105    upsample:
1106    r4=dm(i4,m7);                // get current framelength
1107    r8=dm(i1,m7);                // get addr. of current inputframe
1108    i0=r8;                       // addresse of current inputframe
1109    i2=_upsptempout;
1110    i3=_outframe;                // addr. of outframe
1111    lcntr=r4, do addinpfrm_-1 until lce;
1112       f4=dm(i2,m6);             // get upsptempout
1113       f8=dm(i0,m6);             // get inputvalue
1114       f4=f4+f8;                 // inputvalue + upsptempout
1115       dm(i3,m6)=f4;             // outputval = inputval + upsptempout
1116    addinpfrm_:
1117
1118    rts;
1119 ");
1120
1121 /*-----------------------------------------------------------------------*/
1122
1123 /* decimation and downsampling by 2*/
```

```
1124 /* Register r2: Addresse of FILTERDAT for the FIR-decimation-Filter */
1125 /* framesize in FILTERDAT = framesize after downsampling    */
1126 /* length(input) = 2*length(output) */
1127 asm("
1128 .endseg;
1129 .segment /pm seg_pmco;
1130
1131 .global _downsampby2;
1132 _downsampby2:
1133
1134 // used registers: i0,i1,i2,i3,i12
1135 //               l1,b1,m1,m2,m3,m10,m11,m12
1136 //               r2,r3,r4,r8,r11,r12
1137
1138    i0=r2;                         // Address FILTERDAT for the FIR-Filter
1139    r3=dm(2,i0);                   // Get Filterlength
1140    l1=r3;                         // Length of circular buffer (del.val.)
1141    b1=dm(5,i0);                   // Baseaddress of circular buffer
1142    i1=dm(6,i0);                   // Current startpos in circ. buffer
1143    r3=r3-1;                       // Decrement Filterlength
1144    r3=ashift r3 by -1;            // r3=(N-1)/2 (every 2. coeff = 0)
1145    r2=-r3;                        // r2=-(N-1)/2
1146    m10=r2;                        // m10=-(N-1)/2
1147    r2=-2;
1148    r2=r2-r3,m2=r3;                // r2=-((N-1)/2+2), m2=(N-1)/2
1149    m3=r2;                         // m3=-((N-1)/2+2)
1150    r3=r3-1,m11=m3;                // Decrement loop leng., m11=-((N-1)/2+2)
1151    i2=dm(0,i0);                   // Get Address from inputvalues
1152    i3=dm(1,i0);                   // Get Address from outputvalues
1153    r4=dm(3,i0);                   // Get framesize (after downsampl.)
1154    m1=2;
1155    m12=2;
1156
1157    i12=dm(4,i0);                  // Get startaddress from coefficients
1158    f12=0.0;   // ensure that f12 is different NaN because of f12=f12-f12
1159    f2=dm(i2,m6);                  // Get 1. inputvalue
1160    lcntr=r4, do decframe-1 until lce;
1161       f12=f12-f12,dm(i1,m7)=f2;   // output=0, insert inp.val. in circ. buf
1162       f2=dm(i2,m6);               // Get inputvalue
1163       dm(i1,m5)=f2;               // insert inp. val. in circ. buf
1164       // Calculate Filter
1165       f2=dm(i1,m1),f4=pm(i12,m12); // get 1. val, get 1. coeff
1166       f8=f3*f4,f2=dm(i1,m1),f4=pm(i12,m12); // ..., get 2. val, get 2. coeff
1167       lcntr=r3, do decfilt-1 until lce; // do (N-1)/2-1 times
1168          f8=f2*f4,f12=f8+f12,f2=dm(i1,m1),f4=pm(i12,m12);
1169       decfilt:
1170       modify(i1,m3);               // jump to middle value
1171       modify(i12,m11);             // jump to middle coefficient
1172       f12=f8+f12,f2=dm(i1,m2),f4=pm(i12,m10); // .., get middle val. and ...
1173                                     // ...jmp to oldest, get middle ...
1174                                     // ...coeff and jmp to start of coeff.
1175       f8=f2*f4;
1176       f12=f8+f12,f2=dm(i2,m6);     // .., get inputvalue
1177       dm(i3,m6)=f12;               // save outputvalue
1178    decframe:
1179    l1=0;                           // reset circular buffer
1180    rts (DB);
1181    r8=i1;                          // save current startpos of the..
1182    dm(6,i0)=r8;                    // ..circular buffer
1183
1184 ");
1185
1186 /*------------------------------------------------------------------------*/
1187
1188 /* upsampling by 2 and interpolation*/
1189 /* Register r2: Addresse of FILTERDAT for the FIR-interpolation-Filter */
1190 /* framesize in FILTERDAT = framesize before upsampling       */
1191 /* length(output) = 2*length(output) */
1192 asm("
1193 .endseg;
1194 .segment /pm seg_pmco;
1195
1196 .global _upsampby2;
1197 _upsampby2:
1198
1199 // used registers: i0,i1,i2,i3,i12
```

```
1200 //               l1,b1,m1,m2,m4,m10,m11,m12
1201 //               r1,r2,r3,r4,r5,r8,r12
1202
1203    i0=r2;                          // Address FILTERDAT for the FIR-Filter
1204    r3=dm(2,i0);                    // Get Filterlength
1205    l1=r3;                          // Length of circular buffer (del.val.)
1206    b1=dm(5,i0);                    // Baseaddress of circular buffer
1207    i1=dm(6,i0);                    // Current startpos in circ. buffer
1208    r3=r3-1;                        // Decrement Filterlength
1209    r3=ashift r3 by -1;             // r3=(N-1)/2 (every 2. coeff = 0)
1210    r2=-r3;                         // r2=-(N-1)/2
1211    m10=r2;                         // m10=-(N-1)/2
1212    r2=-2;
1213    r2=r2-r3,m2=r3;                 // r2=-((N-1)/2+2), m2=(N-1)/2
1214    r3=r3-1,m11=r2;                 // Decrement loop leng., m11=-((N-1)/2+2)
1215    i2=dm(0,i0);                    // Get Address from inputvalues
1216    i3=dm(1,i0);                    // Get Address from outputvalues
1217    r4=dm(3,i0);                    // Get framesize (after downsampl.)
1218    m1=2;
1219    m12=2;
1220    m4=-2;
1221    f1=0.0;
1222    f5=2.0;
1223
1224    i12=dm(4,i0);                   // Get startaddress from coefficients
1225    f12=0.0;    // ensure that f12 is different NaN because of f12=f12-f12
1226    f2=dm(i2,m6);                   // Get 1. inputvalue
1227    f2=f2*f5;                       // 2*inpval (because of upsampling)
1228    lcntr=r4, do intframe-1 until lce;
1229       f12=f12-f12,dm(i1,m5)=f2;    // output=0, insert inp.val. in circ. buf
1230       // Calculate Filter
1231       f2=dm(i1,m1),f4=pm(i12,m12); // get 1. val, get 1. coeff
1232       f8=f2*f4,f2=dm(i1,m1),f4=pm(i12,m12); // ..., get 2. val, get 2. coeff
1233       lcntr=r3, do intfilt-1 until lce; // do (N-1)/2-1 times
1234          f8=f2*f4,f12=f8+f12,f2=dm(i1,m1),f4=pm(i12,m12);
1235       intfilt:
1236       f12=f8+f12,modify(i1,m4);     // ..., jump to the oldest value
1237       dm(i1,m2)=f1;         // insert 0 in circ. buf and jump to middle val.
1238       dm(i3,m6)=f12;                // save outputvalue
1239       modify(i12,m11);              // jump to middle coefficient
1240       f2=dm(i1,m2),f4=pm(i12,m10);  // output=0, get middle val. ...
1241                                     // ...and jmp to oldest, get middle ...
1242                                     // ...coeff. & jump to start of coeff.
1243       f8=f2*f4,f2=dm(i2,m6);        // (middle val.) * (middle coeff.),..
1244                                     // ..get next inputvalue
1245       f2=f2*f5,dm(i3,m6)=f8;        // 2*inpval, save outputvalue
1246    intframe:
1247    l1=0;                            // reset circular buffer
1248    rts (DB);
1249    r8=i1;                           // save current startpos of the..
1250    dm(6,i0)=r8;                     // ..circular buffer
1251
1252 ");
1253 /*-------------------------------------------------------------------------*/
1254
1255 /* FIR-Filter */
1256 /* Register r2: Addresse of FILTERDAT for the FIR-Filter */
1257 asm("
1258 .endseg;
1259 .segment /pm seg_pmco;
1260
1261 .global _FIR_Filter;
1262 _FIR_Filter:
1263
1264 // used registers: i0,i1,i2,i3,i12
1265 //               l1,b1,m2
1266 //               r2,r3,r4,r8,r12
1267
1268    i0=r2;                          // Address FILTERDAT for the FIR-Filter
1269    r3=dm(2,i0);                    // Get Filterlength
1270    l1=r3;                          // Length of circular buffer (del.val.)
1271    b1=dm(5,i0);                    // Baseaddress of circular buffer
1272    i1=dm(6,i0);                    // Current startpos in circ. buffer
1273    r3=r3-1;                        // Decrement Filterlength
1274    i2=dm(0,i0);                    // Get Address from inputvalues
1275    i3=dm(1,i0);                    // Get Address from outputvalues
```

```
1276    r2=dm(3,i0);                    // Get framesize
1277    m2=-2;
1278
1279    f12=0.0;    // ensure that f12 is different NaN because of f12=f12-f12
1280    lcntr=r2, do firframe-1 until lce;
1281       i12=dm(4,i0);                 // Get startaddress from coefficients
1282       f2=dm(i2,m6);                // Get inputvalue
1283       f12=f12-f12,dm(i1,m5)=f2;    // output=0, insert sample in circ. buf
1284       // Calculate Filter
1285       f2=dm(i1,m6),f4=pm(i12,m14); // get 1. val, get 1. coeff
1286       f8=f2*f4,f2=dm(i1,m6),f4=pm(i12,m14); // ..., get 2. val, get 2. coeff
1287       lcntr=r3, do firfilt-1 until lce; // do (siz-1) times
1288          f8=f2*f4,f12=f8+f12,f2=dm(i1,m6),f4=pm(i12,m14);
1289       firfilt:
1290       f12=f8+f12,modify(i1,m2);     // ..., jump to the oldest value
1291       dm(i3,m6)=f12;                // save outputvalue
1292    firframe:
1293    l1=0;                            // reset circular buffer
1294    rts (DB);
1295    r8=i1;                           // save current startpos of the..
1296    dm(6,i0)=r8;                     // ..circular buffer
1297
1298 ");
1299
1300 ///////////////////////////////////////////////////////////////////////////////////////////////////
1301 // Function calcpitch()
1302 // Author: Andy Rohr, E98a, zweite Studienarbeit SS2001, HSR Hochschule Rapperswil
1303 // Date: 16.06.2001
1304 // Description: This function extracts the pitch from the voice by using three different methods.
1305 // "method" ist a global variable (which can be set from MATLAB interface) to switch between this methods.
1306 // the calculated pitch is written to "fs", a variable used by the pitchgenerator.
1307 void calcpitch( void )
1308 {
1309     int i;
1310
1311     // compose doublesize frame (we need a frame twice the size of the workingframe in the vocoder)
1312     // This doublesize frame is overlapped 50%.
1313     for (i=0; i < M; i++) {
1314         // copy the old frame of lenght M at the beginning of the doublesize frame
1315         longmodframe[i] = longmodframe[i+M];
1316
1317         // copy the actual frame at the end of the doublesize frame
1318         longmodframe[i+M] = modframe[i];
1319     }
1320
1321     // apply window to doublesize frame
1322     for (i=0; i < 2*M; i++) {
1323         w_longmodframe[i] = longmodframe[i] * window[i];
1324     }
1325
1326     // transform the the windowed doublesize frame to frequency domain
1327     rfft512(w_longmodframe, spec_re, spec_im);
1328
1329     // apply the AKF-method
1330     if (method == 1) {
1331         // need only to apply nonlinear function to first half of spectrum
1332         // the second half is copied from the first half (mirror)
1333         spectrum[0] = spec_re[0]*spec_re[0] + spec_im[0]*spec_im[0];
1334         for (i=0; i < M; i++) {
1335             spectrum[i] = spec_re[i]*spec_re[i] + spec_im[i]*spec_im[i];
1336             spectrum[2*M-i] = spectrum[i];
1337         }
1338         spectrum[M] = spec_re[M]*spec_re[M] + spec_im[M]*spec_im[M];
1339     }
1340
1341     // apply the "quadroot of PSD"-method
1342     if (method == 2) {
1343         // need only to apply nonlinear function to first half of spectrum
1344         // the second half is copied from the first half (mirror)
1345         spectrum[0] = sqrtf(sqrtf(spec_re[0]*spec_re[0] + spec_im[0]*spec_im[0]));
1346         for (i=1; i < M; i++) {
1347             spectrum[i] = sqrtf(sqrtf(spec_re[i]*spec_re[i] + spec_im[i]*spec_im[i]));
1348             spectrum[2*M-i] = spectrum[i];
1349         }
1350         spectrum[M] = sqrtf(sqrtf(spec_re[M]*spec_re[M] + spec_im[M]*spec_im[M]));
1351     }
```

```
1352
1353     // apply the cepstrum-method
1354     if (method == 3) {
1355         // need only to apply nonlinear function to first half of spectrum
1356         // the second half is copied from the first half (mirror)
1357         spectrum[0] = logf(spec_re[0]*spec_re[0] + spec_im[0]*spec_im[0]);
1358         for (i=0; i < M; i++) {
1359             spectrum[i] = logf(spec_re[i]*spec_re[i] + spec_im[i]*spec_im[i]);
1360             spectrum[2*M-i] = spectrum[i];
1361         }
1362         spectrum[M] = logf(spec_re[M]*spec_re[M] + spec_im[M]*spec_im[M]);
1363     }
1364
1365     // transform back to time domain
1366     ifft512(spectrum, zeros, resultframe, dummyframe);
1367
1368     // initialize vars
1369     max = 0;
1370     max_i = 1;
1371
1372     // search the peak in the resultframe which determines the pitch
1373     for (i=50; i<M-35; i++) {
1374         if (resultframe[i] > max) {
1375             max = resultframe[i];
1376             max_i = i;
1377         }
1378     }
1379
1380     // finally calculating the pitch
1381     fs = FC/max_i;
1382     //fs = 250;
1383
1384 }
1385
1386 /*--------------------------------------------------------------------------*/
1387 // CarrierGeneration (CarGen), generiert die Pitch und speichert sie im Buffer carframe.
1388 // Diese Funktion wird zusätzlich in den bestehenden Vocoder eingebaut, um eine interne
1389 // Pitch zu erzeugen. Autor: Oliver Tresch, E98b, Hochschule Rapperswil
1390
1391 void cargen( void )
1392 {
1393     int maxCountGen = FC/fs; // Anzahl Takte pro Periode des Wavetables
1394     int index;               // Schleifen-Zaehler
1395     int CountGrenze;         // Zaehlgrenze fuer Teil-Periodenabschnitte
1396     float schritt;           // Einzelschrittes bei Saegezahn, Dreieck und Sinus
1397     float amplitude2;        // Amplitude fuer 2. Teil der Periode (nur Dreieck)
1398
1399
1400     switch (wavetable) {
1401
1402     case 1: // Rechteck generieren
1403
1404         CountGrenze = tastgrad*0.01*maxCountGen;// Tastgrad einstellen
1405
1406         for (index=0; index < M; index++) {   // carframe fuellen
1407             if (countGen >= maxCountGen) {    // Reset wenn Perioden-Ende
1408                 countGen = 0;
1409             }
1410
1411             if (countGen < CountGrenze) {     // 1. Teil der Periode
1412                 carframe[index] = amplitude;
1413             } else {                            // 2. Teil der Periode
1414                 carframe[index] = -amplitude;
1415             }
1416
1417             countGen++;                      // Zahler inkrementiern
1418         }
1419
1420         break;
1421
1422
1423     case 2: // Saegezahn generieren
1424
1425         // Kommentar: mit 1.999999 ist 2 gemeint. Wegen eines Optimierungs-Fehlers
1426         //            des Comilers (-O2) funktioniert 2 nicht !!
1427
```

```
1428        schritt = 1.999999*amplitude*TC*fs; // Hoehe eines Schrittes
1429        //      = 2*amplitude/maxCountGen;  // Aequivalent obere Zeile
1430
1431        for (index=0; index < M; index++) {   // carframe fuellen
1432            if (countGen >= maxCountGen) {    // Reset wenn Perioden-Ende
1433                countGen = 0;
1434                schritthoehe = 0.0;
1435            }
1436
1437            carframe[index] = (amplitude-schritthoehe);
1438            schritthoehe += schritt;        // Schritthoehe hochzaehlen
1439
1440            countGen++;                     // Perioden-Zaehler inkrementiern
1441        }
1442
1443        break;
1444
1445
1446    case 3: // Dreieck generieren
1447
1448        schritt = 4*amplitude*TC*fs;        // Hoehe eines Schrittes
1449        //      = 4*amplitude/maxCountGen;  // Aequivalent obere Zeile
1450
1451        CountGrenze = maxCountGen*0.5;      // Grenze bei halber Periode
1452        amplitude2 = 3*amplitude;           // Ampliude fuer 2. Teil der Periode
1453
1454        for (index=0; index < M; index++) {   // carframe fuellen
1455            if (countGen >= maxCountGen) {    // Reset wenn Perioden-Ende
1456                countGen = 0;
1457                schritthoehe = 0.0;
1458            }
1459
1460            if (countGen < CountGrenze) {     // 1. Teil der Periode
1461                carframe[index] = (-amplitude+schritthoehe);
1462                schritthoehe += schritt;
1463            } else {                          // 2. Teil der Periode
1464                carframe[index] = (amplitude2-schritthoehe);
1465                schritthoehe += schritt;     // Schritthoehe hochzaehlen
1466            }
1467
1468            countGen++;                     // Perioden-Zaehler inkrementiern
1469        }
1470
1471        break;
1472
1473
1474    case 4: // Sinus generieren
1475
1476        schritt = 2*3.141592*TC*fs;         // Laenge eines Schrittes
1477        //      = 2*3.141592/maxCountGen;   // Aequivalent obere Zeile
1478
1479        CountGrenze = maxCountGen*0.5;      // Grenze bei halber Periode
1480
1481        for (index=0; index < M; index++) {   // carframe fuellen
1482            if (countGen >= maxCountGen) {    // Reset wenn Perioden-Ende
1483                countGen = 0;
1484                x1 = 0;
1485                x2 = 0;
1486            }
1487
1488            if (countGen < CountGrenze) {     // 1. Teil der Periode
1489                carframe[index] = amplitude*(x1-((x1*x1*x1)*0.1666)
1490                                 +((x1*x1*x1*x1*x1)*0.008333)
1491                                 -((x1*x1*x1*x1*x1*x1*x1)*0.0001984)
1492                                 +((x1*x1*x1*x1*x1*x1*x1*x1*x1)*0.000002756)
1493                                 -((x1*x1*x1*x1*x1*x1*x1*x1*x1*x1*x1)*0.00000002505));
1494                x1 += schritt;               // Abszissenpunkt hochzaehlen
1495            } else {                          // 2. Teil der Periode
1496                carframe[index] = -amplitude*(x2-((x2*x2*x2)*0.1666)
1497                                  +((x2*x2*x2*x2*x2)*0.008333)
1498                                  -((x2*x2*x2*x2*x2*x2*x2)*0.0001984)
1499                                  +((x2*x2*x2*x2*x2*x2*x2*x2*x2)*0.000002756)
1500                                  -((x2*x2*x2*x2*x2*x2*x2*x2*x2*x2*x2)*0.00000002505));
1501                x2 += schritt;               // Abszissenpunkt hochzaehlen
1502            }
1503
```

```
1504            countGen++;                     // Perioden-Zaehler inkrementiern
1505        }
1506
1507        break;
1508
1509
1510    case 5: // externer Wave-File-Klang
1511
1512        for (index=0; index < M; index++) {
1513            if (countGen >= P) {          // Reset wenn Perioden-Ende
1514                countGen = 0;
1515            }
1516
1517            carframe[index] = extwave[countGen];// Werte umkopieren
1518
1519            countGen++;                     // WaveTableLaenge-Zaehler inkrementiern
1520        }
1521
1522        break;
1523
1524
1525    default: // carframe mit 0 fuellen, wenn ungueltiges Wavetable gewaehlt
1526
1527        for (index=0; index < M; index++) {
1528            carframe[index] = 0.0;
1529        }
1530
1531    }
1532
1533 }
1534
1535 /*------------------------------------------------------------------------*/
1536 void main ( void )
1537 {
1538    filterdesign();
1539    allocatbuffers();
1540    setinoutbufffilterdat();
1541    setdlyinp();
1542    initrandbuf();
1543    initcalcpitch();
1544
1545    corr = corr * weightout;
1546
1547    /* GET BASE LOCATION OF DMS3 (BITSI) */
1548    init_ms_bases(); /* Initialize memory select bases */
1549
1550    /* initialize hardware */
1551    setup_bitsibb();
1552    setup_sport0();
1553
1554    asm("set_const_reg:
1555      Init_Circ:
1556
1557      Set_interrupts:
1558         bit set imask SPR0I;      /* Unmask interrupt SPT0I */
1559       bit set mode1 IRPTEN;     /* Enable interrutps   */
1560       ");
1561
1562    /* do forever */
1563    while(1)
1564    {
1565       asm("idle;");
1566    }
1567 }
```

Literaturverzeichnis

[1] P. Vary, U. Heute, W. Hess *Digitale Sprachsignalverarbeitung*, Teubner, 1998 Stuttgart, ISBN 3-519-06165-1

[2] Tresch O. und Sibler G. *Pitch-Generator für Kanal-Vocodersystem*, Studienarbeit W01-02 am Labor für Digitale Signalverarbeitung der HSR, WS 2000/2001

[3] Schüeli Prof. Dr. A., Autographie zur Vorlesung Digitale Signalverarbeitung an der HSR Abteilung E

[4] Handbücher zu ANALOG DEVICES ADSP-2106x (*User's Manual, C Tools Manual, C Runtime Library Manual*)

Abbildungsverzeichnis

1.1 Vollständiges Labor-Vocodersystem, ergänzt durch Pitch-Extraktor 2

3.1 Die drei Verarbeitungsstufen eines GFB-Algorithmus 9
3.2 Typische Vorverarbeitungsstufe einer GFB-Algorithmus nach dem Prinzip der Kurzzeitanalyse 10
3.3 GFB-Vorverarbeitungsstufe mit doppelter Spektraltransformation und nichtlinearer Verzerrung im Spektralbereich. 11
3.4 Kennlinien der nichtlinearen Verzerrung im Frequenzbereich. 11
3.5 GFB mit doppelter Spektraltransformation und nichtlinearer Verzerrung: einige Beispiele zur Wirkungsweise. 12
3.6 Bestimmung der Pitch nach der Vorverarbeitungsstufe (doppelte Spektraltransformation nach der Methode 4. Wurzel aus dem PDS). 13

4.1 Frame-by-Frame-Vergleich der Pitch-Extraction-Methoden. 16
4.2 Vergleich der Pitch-Extraction-Methoden über ein ganzes Wavefile. 17
4.3 Tiefe Männerstimme. 18
4.4 Problemlose Frauenstimme. 19
4.5 Hoher Gesang. 20
4.6 Frauenstimme mit lauten Zischlauten. 21
4.7 Männerstimme: AKF Probleme mit F1 und tiefen Pitchfrequenzen. 22
4.8 Entscheidungsproblem stimmhaft/stimmlos bei einem starken Konsonanten «p». . . 23
4.9 Tiefe verzerrte Männerstimme. 24
4.10 Mittlere Männerstimme. 25
4.11 Hohe Frauenstimme. 26
4.12 Männlicher Nachrichtensprecher. 27
4.13 Ausschnitt männlicher Nachrichtensprecher. 28
4.14 Vokal «e» mit steigender Pitch. 29
4.15 Vokal «i» mit steigender Pitch. 30
4.16 Vokal «o» mit steigender Pitch. 31
4.17 Vokal «u» mit steigender Pitch. 32
4.18 Interrupt Service Routine (ISR) 34
4.19 Nichtlineare Verzerrung im Frequenzbereich. 35
4.20 Pitchverlauf von Sweep mit AKF-Methode. 36
4.21 Pitchverlauf von Sweep mit Cepstrum-Methode. 37
4.22 Pitchverlauf von Sweep mit 4. Wurzel-Methode. 38
4.23 Kombination von zwei «Vocoderframes» zu einem «Pitchextractorframe». 39