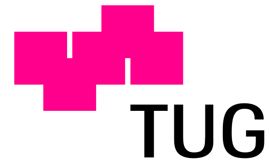
Signal Processing and Speech Communication
Laboratory
Graz University of Technology



Real-Time Enhancement of E-Larynx Speech Signals

Master Thesis

at

Graz University of Technology

submitted by

Thomas Noisternig

Signal Processing and Speech Communication Laboratory
Inffeldgasse 12, A-8010 Graz, Austria

November 26, 2009

© Copyright 2009 by Thomas Noisternig

Advisor: Univ-Prof. DI Dr. Gernot Kubin
Co-Advisor: DI Dr. Martin Hagmüller

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Abstract

People who have lost their larynx (e.g. due to cancer) have no natural possibility to speak any more, because the vocal folds are not longer available to generate the necessary sound source. A solution to this problem is a mechanical device, which substitutes the missing body part - the so-called electrolarynx. Unfortunately, speaking with an electrolarynx suffers from high background noise caused by the sound of the device itself as well as low intelligibility because of the monotonic frequency the electrolarynx device produces. This work aims to reduce these drawbacks by increasing the quality of the electrolarynx speech signal in real-time using a Texas Instruments TMS320C6713B floating-point digital signal processor (DSP). The implemented quality enhancement targets on increasing the speech quality with two approaches: First, the directly-radiated electrolarynx noise (DREL) component is reduced. Therefore the DSP utilises spectrum-based modulation filtering methods for detecting the signal's DREL component and removing it from the signal. The separation of signal and noise is hereby achieved via detecting constant spectral components - the electrolarynx generates it's vibrations at a constant frequency. The second enhancement approach intends to adjust this constant frequency, which otherwise results in an artificial sounding voice with furthermore missing prosodic information. The enhancement algorithm is in such cases able to detect voice variations and apply them to the original, monotonic speech signal. This is done by utilising the speech's formant contour, as detected by the implemented linear predictive coding (LPC) based formant tracker, to control the electrolarynx in order to generate an analogous pitch contour for the sake of making it sound more natural and understandable. To comply with arising requirements towards real-time ability the implementation process was accompanied with various speed optimisation procedures. Primarily these procedures consisted of finding optimal trade-off settings between accuracy and complexity as well as hard-coding constant coefficients, pre-calculating parameters during initialisation, re-arranging data structures and reducing redundancies.

Kurzfassung

Personen, die (beispielsweise durch eine Krebserkrankung) ihren Kehlkopf verloren haben, fehlt die natürliche Möglichkeit zu sprechen, da der Sprechapparat nicht länger in der Lage ist, das Anregungssignal zu erzeugen. Eine Lösung dieses Problems stellt ein mechanisches Gerät dar, das diese Funktion des fehlenden Körperteils ersetzt - der Elektrolarynx. Das Sprechen mit einem Elektrolarynx leidet unter einem hohem Hintergrundgeräusch, hervorgerufen durch die Vibrationen des Geräts selbst sowie einer niedrigen Verständlichkeit, unter anderem aufgrund der monotonen Grundfrequenz, die der Elektrolarynx erzeugt. Die vorliegende Arbeit zielt darauf ab, diese Schwächen zu reduzieren, indem eine Verbesserung der Qualität des Elektrolarynx-Sprachsignals mittels einem Texas Instruments TMS320C6713B floating-point Signalprozessor (DSP) in Echtzeit geschieht. Die durchgeführten Qualitätsverbesserungen zielen darauf ab, die Spachqualität mittels zwei Ansätzen zu erhöhen: Einerseits wird das direkt abgestrahlte Elektrolarynx Störgeräusch (DREL) reduziert. Dazu wird ein Ansatz verwendet, der mit einem Modulationsspektrumsfilter die DREL Komponente vom Signal entfernt. Die Trennung von Signal und Störung erfolgt dabei durch die Detektion von zeit-invarianten Spektralkomponenten - der Elektrolarynx schwingt auf konstanter Frequenz. Der zweite Ansatz zielt auf eine Variation dieser Frequenz ab, da sie ansonsten zu einer künstlich klingenden Stimme wegen fehlender prosodischer Information führt. Der Verbesserungsalgorithmus analysiert Sprachvariationen und erzeugt damit eine künstliche Grundfrequenzkontur. Erreicht wird dies durch die Verwendung der Formanten, die mittels Linear Predictive Coding (LPC) basierten Formanttracker geschätzt werden. Die künstliche Grundfrequenzkontur wird zur Steuerung des Elektrolarynx verwendet, um eine entsprechende Kontur der Tonhöhe zu generieren um dadurch die Sprache natürlicher und besser verständlich klingen zu lassen. Um die Anforderung der Echtzeitfähigkeit zu erfüllen, wurde der Implementationsprozess von verschiedenen Techniken zur Geschwindigkeitsoptimierung begleitet. Primär bestanden ebendiese aus der Suche nach optimalen Einstellungen bezüglich eines Kompromisses zwischen Genauigkeit und Komplexität sowie dem Hard-Coding von konstanten Koeffizienten, vorausberechnen von Parametern während der Initialisierung, restrukturieren von Datenstrukturen und reduzieren von Redundanzen.

Contents

List of Figures vii

List of Tables ix

Abbreviations xi

1 Introduction 1

2 Background 5
- 2.1 Speech and Phonetics Theory . . . 5
  - 2.1.1 Biological Fundamentals . . . 5
  - 2.1.2 Speech Theory . . . 6
  - 2.1.3 Speech Description Parameters . . . 7
  - 2.1.4 Artificial Larynges . . . 9
- 2.2 Speech Signal Processing Techniques . . . 11
  - 2.2.1 Block Processing . . . 11
  - 2.2.2 Linear Predictive Coding . . . 12
  - 2.2.3 Pitch-Synchronous Overlap-Add . . . 13
  - 2.2.4 Pitch-Marking . . . 14
  - 2.2.5 Pitch Tracking . . . 15
  - 2.2.6 Formant Tracking . . . 15
- 2.3 DSK6713 Overview . . . 16
  - 2.3.1 Introduction . . . 16
  - 2.3.2 Digital Signal Processor . . . 16
  - 2.3.3 Audio Codec . . . 18
  - 2.3.4 Memory Management . . . 18
  - 2.3.5 Programming the DSP Starter Kit . . . 19

3 Design 23
- 3.1 Multipath Separation . . . 23
  - 3.1.1 Modulation Spectral Filtering . . . 24
  - 3.1.2 Spectral Subtraction . . . 25
- 3.2 Pitch Contour Generation . . . 26

4 Implementation 29
- 4.1 Experimental Setup . . . 29
- 4.2 Block Processing Framework . . . 30
- 4.3 Hardware Setup . . . 31
  - 4.3.1 Audio Codec . . . 31
  - 4.3.2 Memory Management . . . 32
- 4.4 Modules . . . 34
  - 4.4.1 Fast Fourier Transform . . . 34
  - 4.4.2 Finite Impulse Response Filter . . . 37

4.4.3 Infinite Impulse Response Filter 38
4.4.4 Windowing 39
4.4.5 Linear Predictive Coding 39
4.4.6 Voice Activity Detection 40
4.4.7 Voiced/Unvoiced Detection 40
4.4.8 Pitch Tracking 40
4.4.9 Pitch-Marking 43
4.4.10 Formant Tracking 44
4.4.11 Formant Smoothing 49
4.4.12 Pitch-Synchronous Overlap-Add 50
4.4.13 Multipath Separation 52
4.4.14 Pulse Generation 55
4.5 Real-Time Related Aspects 56
4.5.1 Online/Offline Data Processing Discrepancies 56
4.5.2 Timing Considerations 58
4.5.3 Real-Time Effecting Parameters 60

5 Results **67**
5.1 Measurement and Calculation 67
5.1.1 System Delay 67
5.1.2 Multipath Separation Modules 73
5.1.3 Pitch Contour Generation Modules 77
5.2 Results Comparison with Different Setups 85
5.2.1 Block Size and Fast Fourier Transform Point Size 85
5.2.2 Smoothing Filter Size 85
5.2.3 Filter Type in the Multipath Separation Module 87

6 Conclusion and Outlook **89**

Bibliography **91**

A Appendix **95**
A.1 Matlab Code 95
A.1.1 Filter Design 95
A.1.2 Window Design 99
A.2 External Code 100
A.2.1 Block Processing Framework 100
A.2.2 KISS Fast Fourier Transform 100
A.2.3 Polynomial Roots Calculation 101
A.3 Praat 103
A.4 Apparatus Usage and Operation Modes 104
A.5 Used Devices 105
A.5.1 Description 105

List of Figures

1.1 Simple block diagram of the system's processing chain 3

2.1 Vocal tract organs taking part in the speech generation 6
2.2 Glottal transfer function of a speech signal 8
2.3 Artificial voice boxes 10
2.4 TD-PSOLA working principle 13
2.5 Random signal and it's corresponding pitch-marks 14
2.6 Internal structure of the TMS320C6713B DSP 17

3.1 Signal examples with constant and varying temporal envelope 24
3.2 Multipath separation based on MSF, principle block diagram 25
3.3 Pitch contour generation block diagram, processed signal output 27
3.4 Pitch contour generation block diagram, driving pulses output 28

4.1 Experimental setup of the enhancement framework 29
4.2 Block processing workflow 31
4.3 Hermitian property of the DFT 35
4.4 FIR filter blocks 37
4.5 Biquad structure in DF1 38
4.6 The autocorrelation-based pitch detector's window effect cancellation 41
4.7 Working principle of the zero-crossing pitch detector 42
4.8 Implemented data structures used for pitch-marking 44
4.9 Comparison of index- and expectation-based formant tracking 45
4.10 Multipath separation based on MSF, detailed block diagram 53
4.11 MSF filtering of a certain spectral component 53
4.12 Optimal excitation pulse 55
4.13 Convolution operation to generate the shaker driving signal 56
4.14 Averaging problems at the calculation procedure's beginning 57
4.15 Declination cancelling effect with adaptive mean calculation 58
4.16 The pitch contour generator's processing line 63
4.17 Performance of the implemented sorting algorithms with ten test runs 65

5.1 Processing delay measurement signals, speech input 71
5.2 Processing delay measurement signal, pulse input 72
5.3 Performance of the MSF-based multipath separation 74
5.4 Loudness discrepancy when using the MSF filter 75
5.5 Comparison of the implemented MS methods 76
5.6 VAD performance example 78
5.7 Run-time comparison between LPC and IFC trackers 80
5.8 Formant detection quality comparison between both implemented trackers 81
5.9 Example result of the formant tracker 82
5.10 Example result of the PSOLA module 84
5.11 Smoothing performance comparison using different filter sizes 86

A.1 Frequency response of the AKG HD171 . . . 97
A.2 Final frequency responses of the inverse microphone filters . . . 98
A.3 Hann window coefficients for a window size of 256 . . . 99
A.4 Devices used in the framework . . . 106
A.5 Utility devices used for measurement and amplification . . . 107

List of Tables

2.1 Consonant type and appropriate vocal tract setup 7
2.2 Reference values indicating the DSP's operating speed 18
2.3 Audio codec specifications 18
2.4 Data regions available for the compiler's memory mapping 19

4.1 Matrix in Toeplitz structure 39
4.2 Example results for the pitch-marks list size in `COMP` mode 44
4.3 Optimal filter bandwidths B_x for IFC-based formant tracking 48
4.4 Smoothing algorithm example 50
4.5 BLKSZ examples and frame durations, amount of CPU cycles resulting from it 61
4.6 Example values for the LPC approximator order 62
4.7 Example exit condition setups for the polynomial roots calculator 62

5.1 Latency of various VoIP systems 68
5.2 DSP framework setup for the latency measurement 70
5.3 Processing blocks of LPC and IFC tracker 80
5.4 Formant tracker configuration used for the discussed example 83

A.1 Structure of the coefficients array returned by `tf2sos()` 96
A.2 Structure of the coefficients array needed by `biquad()` 96
A.3 Optimal filter parameters for designing the inverse microphone filter 98
A.4 Functional DIP switch assignment 104
A.5 LED assignment 104
A.6 List of devices used in the framework 108
A.7 List of used utility devices 109

Abbrevations

A/D	**A**nalog-To-**D**igital
ADC	**A**nalog-To-**D**igital **C**onverter
AD/DA	**A**nalog-To-**D**igital / **D**igital-To-**A**nalog
ALU	**A**rithmetical and **L**ogical **U**nit
API	**A**pplication **P**rogrammable **I**nterface
AWGN	**A**dditive **W**hite **G**aussian **N**oise
BP	**B**and-**P**ass
BLKSZ	**Bl**oc**k** **Siz**e
BSL	**B**oard **S**upport **L**ibrary
CCS	**C**ode **C**omposer **S**tudio
CPU	**C**entral **P**rocessing **U**nit
CSL	**C**hip **S**upport **L**ibrary
D/A	**D**igital-To-**A**nalog
DAC	**D**igital-To-**A**nalog **C**onverter
DIP	**D**ual **I**nline **P**ackage
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DREL	**D**irect-**R**adiated **E**lectro**l**arynx Noise
DF1	**D**irect **F**orm **1**
DF2	**D**irect **F**orm **2**
DFT	**D**iscrete **F**ourier **T**ransform
DMA	**D**irect **M**emory **A**ccess
DSP	**D**igital **S**ignal **P**rocessor
DSPLib	**D**igital **S**ignal **P**rocessor **Lib**rary
DSK	**D**igital Signal Processor **S**tarter **K**it
(E)DMA	(**E**nhanced) **D**irect **M**emory **A**ccess
EMIF	**E**xternal **M**emory **I**nter**f**ace
EQ	**Eq**ualizer
F_0	Fundamental Frequency
F_1	**F**ormant **1**
F_2	**F**ormant **2**
F_3	**F**ormant **3**
F_x	**F**ormant **x**
f_s	Sampling Rate
F0gen	Pitch Contour Generation
FIR	**F**inite **I**mpulse **R**esponse
FDATool	**F**ilter **D**esign and **A**nalysis **Tool**
FFT	**F**ast **F**ourier **T**ransform
GUI	**G**raphical **U**ser **I**nterface
HGS	**H**anquinet, **G**renez, **S**choentgen
HP	**H**igh-**P**ass
HPV	**H**uman **P**apilloma **V**irus
IIR	**I**nfinite **I**mpulse **R**esponse
IF	**I**nverse **F**ilter
IFC	**I**nverse **F**ilter **C**ontrol

IFFT	**I**nverse **F**ast **F**ourier **T**ransform
ITU-T	**I**nternational **T**elecommunication **U**nion **-** **T**elecommunication Standardization Sector
I/O	**I**nput / **O**utput
L1	**L**evel **1**
L2	**L**evel **2**
LED	**L**ight **E**mitting **D**iode
LPC	**L**inear **P**redictive **C**oding
MAC	**M**ultiply-**Ac**cumulate
MFLOPS	**M**illion **Fl**oating-**P**oint **O**perations **p**er **S**econd
MIPS	**M**illion **I**nstructions **p**er **S**econd
ML	**M**achine **L**earning
MMACS	**M**illion **M**ultiply-**Ac**cumulate **C**ycles **p**er **S**econd
MMSE	**M**inimum **M**ean **S**quared **E**rror
MS	**M**ultipath **S**eparation
MSF	**M**odulation **S**pectral **F**ilter
OLA	**O**verlap-**A**dd
PC	**P**ersonal **C**omputer
PSOLA	**P**itch-**S**ynchronous **O**verlap-**A**dd
PSL	**P**eripheral **S**upport **L**ibrary
RAM	**R**andom **A**ccess **M**emory
RMS	**R**oot **M**ean **S**quare
R/W	**R**ead / **W**rite
SS	**S**pectral **S**ubtraction
SNR	**S**ignal-To-**N**oise **R**atio
STFT	**S**hort **T**erm **F**ourier **T**ransform
TCP	**T**ransmission **C**ontrol **P**rotocol
TD-PSOLA	**T**ime **D**omain **-** **P**itch-**S**ynchronous **O**verlap-**A**dd
TEP	**T**ransoesophageal **P**uncture
THD	**T**otal **H**armonic **D**istortion
THDN	**T**otal **H**armonic **D**istortion and **N**oise
UDP	**U**ser **D**atagram **P**rotocol
USB	**U**niversal **S**erial **B**us
VAD	**V**oice **A**ctivity **D**etection
VLIW	**V**ery **L**ong **I**nstruction **W**ord
VoIP	**V**oice **O**ver **IP**
V/UV	**V**oiced / **U**nvoiced

Chapter 1

Introduction

Worldwide there are about 600000 laryngectomees, lots of them make use of electrolarynx devices to acoustically communicate with their surrounding ([25]). Being able to do so is a fundamental need of people as well as of high importance for handling every day's work. Therefore electrolarynx users are frequently confronted with having to use this aid - convenience, ease in usage and good acceptance from conversation partners play a big role in supporting them as a good as possible. Manufacturers have been spending money, time and effort in developing smaller, easier and more effective devices to offer the best possible comfort to the user. Nevertheless the ease of operation is mainly focusing on the speaker side when talking about electrolarynx conversations. The mentioned acceptance by conversation partners represents the other side. Obviously, this acceptance comes along with the intelligibility - people prefer taking part in conversations when it is easy to follow the conversation partner. Unfortunately despite the experience in manufacturing electrolarynx devices, the resulting speech is still quite hard to understand. Due to this fact, it is very hard for a handicapped person to talk with others: Either people are daunted by the strange sounding voice (the steady pitch generated by the electrolarynx leads to an artificial sounding voice) or they simply have difficulties understanding electrolarynx spoken words what constrains simple and flawless conversations.

This is where this work focuses on: The approach of increasing the sound quality to make conversations with electrolarynx speakers easier. To do so the sound enhancement intends to reduce two main drawbacks.
At first the sound of the device itself, originating from the necessary vibrations the electrolarynx has to produce, is rather loud. It might even predominate the voice of the speaker, especially when they are not experienced in using the electrolarynx optimally. This sound, the directly-radiated component of the electrolarynx device, can be suppressed using different signal processing approaches that were initially developed and discussed in [20]. The purpose of this work is to implement these techniques in a real-time environment so that electrolarynx users can actually benefit from the enhancement as they speak. At a glance the sound produced by the electrolarynx is split into two parts: The (wanted) speaker's voice and the (unwanted) **D**irect-**R**adiated **E**lectro**l**arynx Noise (DREL) component. To be able to suppress the unwanted component without effecting the wanted component these two signals have to be separated as strict as possible. Therefore a method working in the modulation spectrum domain which is described in chapter 3.1.1 is used. This approach filters temporal trajectories of a signal's spectral components and dampens the ones which have a time-invariant trajectory. Speech signals themselves are modulated with at least $2Hz$ by the

movement of the speaker's mouth and tongue ([6, p.7]). With the DREL component this is not the case. The electrolarynx device vibrates at a constant frequency. So a separation of both signal components is possible by using this property to categorise and afterwards process the signal's components. At a glance, the processing chain consists of:

1. Capturing and pre-processing the input signal
2. Performing a Fourier transform
3. Applying the **M**odulation **S**pectral **F**ilter (MSF)
4. Performing an inverse Fourier transform
5. Outputting the signal

The second drawback the developed speech enhancement framework copes with is the constant pitch of the speaker's voice. As mentioned state-of-the-art electrolarynx devices are not able to adapt their vibration frequency to the words the user is currently speaking. In healthy voices voice's pitch might contain additional information about the meaning of a phrase or sentence that can not be figured out just by analysing the word and sentence structure itself - accents, questions, ironic statements for example make intense usage of this possibility. Electrolarynx users are lacking this possibility. This framework uses the available formant information to equip the pitch with a varying contour. Certainly, this method is not able to reproduce meta-information like irony or speaker boredom but is still able to make the artificial voice sound more natural and transport prosodic information that alters the formant. Accents, for example, are able to do so when pronounced clearly enough. The processing pipeline for performing these tasks looks like follows:

1. The input speech is captured and pre-processed
2. The pre-processed speech is analysed to gather speech activity, voicing and formant information
3. The obtained information is post-processed (smoothing, limiting)
4. The final data stream of meta-information is used to alter the pitch accordingly
5. The result is passed to the output

The implementation used for performing these procedures operates on a **D**igital Signal Processor **S**tarter **K**it (DSK) board equipped with a high-performance **D**igital **S**ignal **P**rocessor (DSP) and a audio codec. The starter kit is a *TMS320C6713 DSK*, it is equipped with a 225MHz floating-point DSP, the audio codec is a 16 bit stereo audio codec. The codec is able to capture the analog audio signal, convert and pass it to the DSP and return the enhanced signal to the analog output. For audio input and monitoring a studio headset is used. At a glance these four components (headset microphone, audio codec, DSK and headset headphone), as pictured in figure 1.1, complete the real-time audio processing chain.

This paper is partitioned into four main chapters: Chapter 2 discusses the theoretical fundamentals the developed framework is based on. In chapter 3 the focus is on the processing algorithm's basic design whereas chapter 4 takes a more detailed view by explaining the

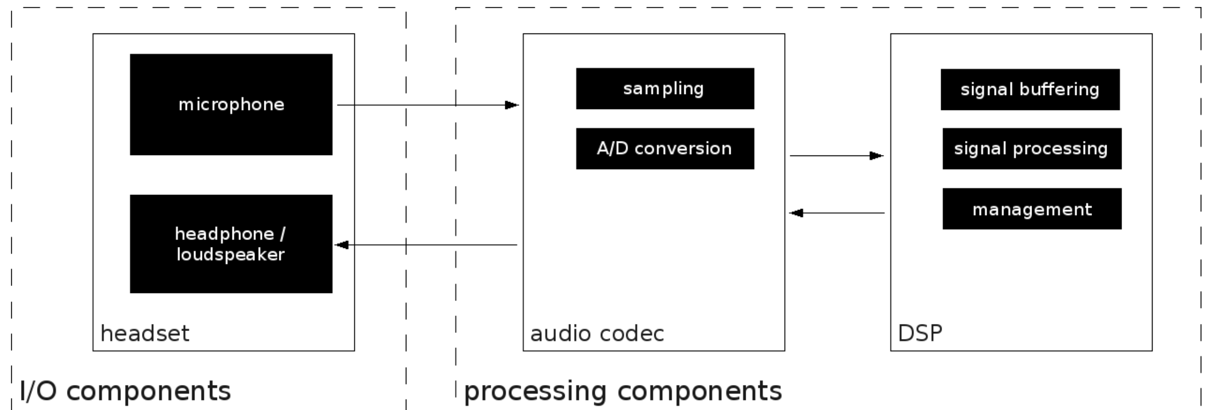


Figure 1.1: *Simple block diagram of the system's processing chain showing the major components and their usage. In principle the microphone captures an analog sound signal and passes it to the audio codec for A/D conversion. The digital signal is enhanced by the DSP and returned to the audio playback device.*

single module's implementation. Finally the achieved enhancement is presented in chapter 5 and analysed in terms of results quality and implementation efficiency. Additionally, several configuration setups are investigated and used to recommend an optimal set of parameters to run this framework as well as comparable other ones.

Chapter 2

Background

2.1 Speech and Phonetics Theory

2.1.1 Biological Fundamentals

Generally speaking, sound is the variation of air pressure captured by the ear. In order to produce sound, the body has to have the ability to produce these variations. At a glance the main body parts taking part in acoustic speech production are the excitation organs like

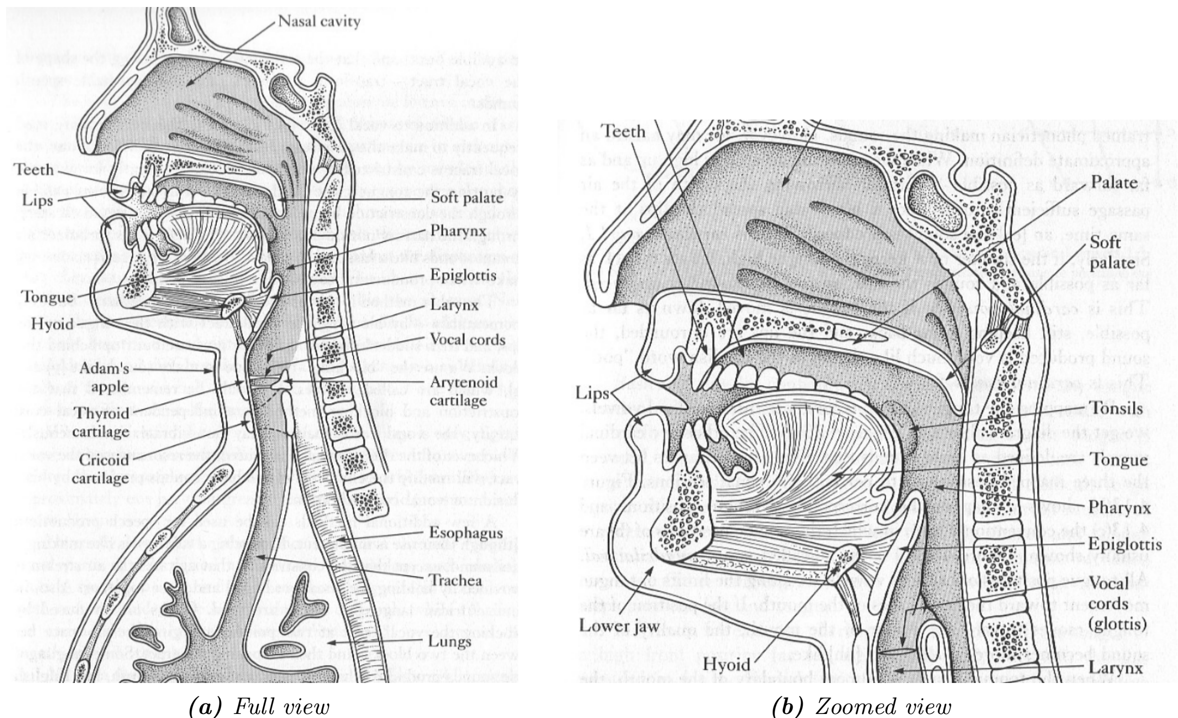
- lungs
- trachea
- glottis
- larynx

and the articulation organs or vocal tract parts

- lower jaw
- tongue
- lips
- velum (soft palate)

as shown in figure 2.1.

The excitation organs form the initial oscillations in the air pressure which is generated by the lungs. In particular this is the job of the vocal folds in the larynx which are capable of varying the inner volume of the air channel. If air is pressed from the lungs through the air channel to the mouth these volume variations result in a pressure change of the air stream that leaves the mouth. With adapting the larynx' behaviour in generating vibration the speech parameters pitch, loudness, voice quality and several other speech determining patterns can be set.
Before the air stream is processed by the articulation organs it has to be generated first. This process of generating the air flow that reaches the vocal tract parts is called *initiation*. The most obvious way is the one explained before: The air pressure is generated by the lungs and reaches the articulation organs through the air channel. This process is called *pulmonic*



(a) Full view

(b) Zoomed view

Figure 2.1: *Human vocal tract organs taking part in the natural speech generation (source: [14]).*

initiation and takes place for example when speaking vowels. Nevertheless there are two other ways to generate phones. Generated by the larynx with raising and lowering it the initiation is called *glottalic*. The `/g/` consonant is for example produced this way. In the process of speaking out stops (Sect. 2.1.2) the sound is generated at the velum by using the tongue, it is referred to as *velaric* initiation.

The excited air stream still does not contain any speaking information. This information is added with the speaker's forming of lower jaw, lips, tongue and velum as the air passes the mouth. So the phonemes are *(a)* formed by the vocal tract with filtering the air stream and *(b)* resonated in the mouth dependent on the setting of the articulation organs. Several unvoiced phones are also generated by specific vocal part patterns. For example the tongue position mainly influences the phone height and fronting, the lip posture determines the spread, nasalisation thus the air flow through the noise channel is utilised for specific speech patterns and of course the duration determines the resulting sound. For detailed information about biological aspects related to speech processing please refer to the proper literature. Good choices for example are [26] and [6].

2.1.2 Speech Theory

A main differentiation between phones is categorising them as one of the following two groups:

1. Consonants: `/f/`, `/s/`, `/h/`, `/t/`, `/w/`, `/l/`, ...

2. Vowels: `/a/`, `/e/`, `/i/`, `/o/`, `/u/`

[27] differs vowels and consonants by the way they are produced: Vowels are free and open sounds whereas consonants are always accompanied by noise. So vowels are always voiced sounds (Sect.2.1.3), they in general have higher amplitudes than consonants for this reason. However consonants might be voiced or unvoiced, dependent on the stricture of the glottis. The exact setting determines the consonant type:

Slightly open: *Fricatives* are spoken out this way

Particularly open: Results in a consonant of *approximant* type

Closed: Produces *stops*

This setting along with *laterality* (airflow besides or centered over the tongue) and *nasality* (airflow is channeled through the nose or orally) is called the *manner of articulation.* A possible sub-type is determined by the *position of articulation* thus if the consonant is generated in the front or back of the mouth. Table 2.1 includes an example list showing common consonants and their vocal tract setting.

Type	Sub-Type	Example	Vocal Tract Setting			
			Stricture	**Nasality**	**Laterality**	**Position**
approximant	semi-vowel	`/w/,/j/`	slightly open	oral	cental	
	liquid	`/l/`	slightly open	oral	lateral	
fricative		`/f/,/s/`	partially open	oral	cental	front
		`/h/`	partially open	oral	cental	back
stop	oral stop	`/t/,/p/`	closed	oral	cental	front
		`/g/,/k/`	closed	oral	cental	back
	nasal stop	`/n/,/m/`	closed	nasal	cental	
thrill		`/r/`	special setting with rolled-back tongue and vibrating sound in the mouth			

Table 2.1: *Vocal tract setup and consonant types and sub-types resulting from it.*

2.1.3 Speech Description Parameters

Pitch

The pitch of a speaker's voice determines the subjective voice "height" and therefore is the characteristic that makes a voice "low" or "high" for the listener. In general the pitch of a male speaker is lower than a woman's with, according to [34], a value of about $120Hz$ compared to $210Hz$. The pitch of a child is even higher than a woman's with being located around $350Hz$.

From a technical point of view the pitch is the perceived speech frequency that corresponds with the fundamental frequency the air oscillates with. Often the pitch and fundamental frequency are confused of being the same. In fact, this might be the case for most cases but nevertheless in some situations it does not apply due to psychoacoustic reasons.

Formants

Formants are a very important characteristic in speech processing. They are those frequencies in the vocal tract impulse response that are resonated and therefore lead to a peak in the vocal tract transfer function (Fig. 2.2). In most cases four to six of these frequencies are found in the speech spectrum which are located in the appropriate kHz area thus **F**ormant **1** (F_1) is - roughly - about $1kHz$, formant two about $2kHz$ and so on. Nevertheless the actual value of a specific formant fluctuates in time and is determined by the spoken phoneme for the first two formants F_1 and **F**ormant **2** (F_2) (e.g. the vocal `/u/` results in the lowest formant frequency and `/i/` in the highest frequency respectively). Higher formants are independent from what is spoken, they rather relate on who is speaking.
From this follows that for creating a speech-dependent pitch contour information from either formant F_1 or F_2 has to be taken into consideration. Formants **F**ormant **3** (F_3) and upwards would cause the contour to be speaker-dependent, for the sake of providing a universally valid application using those formants were avoided when inventing this work.

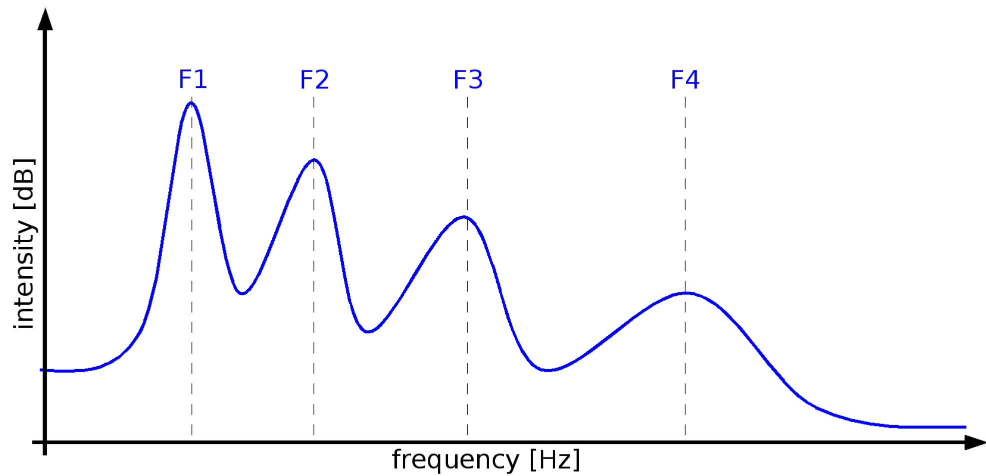


Figure 2.2: *Typical vocal tract transfer function of a speech signal with* 4 *formants.*

Voiced/Unvoiced Phones

One categorisation of phones is their habit of being voiced or unvoiced. Voiced phones (like e.g. the vocal `/a/`) are generated with filtering the oscillations generated by the larynx like described in section 2.1.1. They therefore consist of a clear fundamental frequency with added phone-dependent harmonics.

Unvoiced phones on the other hand can be treated simply as noise from a signal processing point of view. Phones like `/t/` or `/s/` are not produced with filtering the oscillating air stream but by forming sounds on the non-oscillating exhaled air using the vocal tract organs - for example by clicking with the tongue. These facts result in an unvoiced phone having higher-energetic spectral components in the upper frequency spectrum than voiced phones. This property is often used for **V**oiced / **Un**voiced (V/UV) categorisation.

2.1.4 Artificial Larynges

It can have lots of reasons why the body is not capable of producing (intelligible) sound. This starts with birth defects but can also be due to diseases like cancer where the larynx has to be removed surgically or accidents that hurt body parts needed for a clear voiced production.

In the year 2002 there were approximately 140000 people worldwide suffering from laryngeal cancer ([30]), many of them can be helped with artificial larynges. According to [18] there are about four to eight incidences in central Europe per 100000 inhabitants, in Austria about five per 100000 ([19]). Men are nearly 10 times more at risk than as women.

Typical risk factors are (partially taken from [13]):

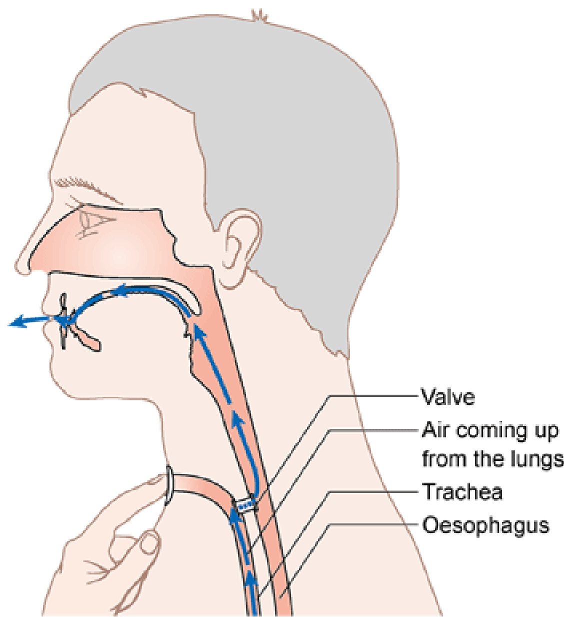
Smoking: With over 95% of all laryngeal cancer patients being smokers, smoking is the highest risk factor that may lead to this type of cancer. This is reasoned with smoking causing genetic alterations and the permanent non-clearance of the respiratory tract.

Alcohol: Alcohol is said to increase the negative effects of smoking so the combination of high alcohol consumption in combination with smoking additionally increases the possibility of coming down with voice box cancer.

Viruses: Certain viruses, such as **H**uman **P**apilloma **V**irus (HPV) or acid reflux also affect the respiratory tract, people suffering from one of these viruses are more likely to suffer from laryngeal cancer for this reason.

Poison exposure: Occupational exposure to chemicals again attacking the respiratory tract, for example coal dust, diesel fumes, nickel, ... also increases the danger of coming down with laryngeal cancer.

To still be able to produce voice without a larynx the missing or malfunctioning vocal organ has to be substituted. In principle these methods are categorised in two groups: Alternative speech methods that bypass the defective organs and artificial substitutes. One common alternative speech method is known as *esophageal speech*. With it one is possible of producing speech without being dependent on the oscillations in the vocal folds. These - necessary - oscillations are produced by pushing previously swallowed air through the esophagus what causes vibrations in the pharynx (muscles of the throat). By forming lips and mouth as usual the vibrations are again filtered like with common speech and produce an utterance. The method referred to as **T**ransoe**e**sophageal **P**uncture (TEP) also belongs this family. With it one surgically gets a connection between wind pipe (stoma) and food pipe (esophagus) which is equipped with a one-way valve, the *voice prosthesis*. When wanting to speak the stoma is covered with the fingers as shown in figure 2.3a. This, when breathing out, redirects the air stream to the gullet and finally to the mouth. The forming of words is again done the common way with filtering the leaving air stream with forming mouth and lips. Generally speaking these methods have the advantage of sounding more natural than artificial substitutes like the electrolarynx, however suffering from a lower pitch than the original voice. The *electrolarynx* device which's sound is supposed to be enhanced with the framework invented here is dealt with separately and more detailed in section 2.1.4.



(a) The TEP device works with surgically connecting stoma and esophagus through a pipe in order to redirect the air stream to the mouth (source: http://www.cancerhelp.org.uk).

(b) Electrolarynx devices are placed on the throat, the vibrations the device generates are forwarded to the air channel and finally the mouth (source: http://www.maxiaids.com).

Figure 2.3: *Examples of common artificial voice boxes that are currently available on the market.*

Electrolarynx

One wide spread aid is the *electrolarynx* or simply *e-larynx* device as shown in figure 2.3b. It is used for replacing the functionality of the larynx in case of a defective or missing natural one. As discussed in section 2.1.1 a function of the larynx is to generate the necessary vibrations in the air channel. When replacing the larynx with an artificial adequate this device must provide a similar functionality. The electrolarynx does so with controlling an electric motor to vibrate at a constant frequency when running freely. Some models include a method of letting the user adapt this frequency with a wheel (continuous alteration) or additional buttons (discrete alteration through button-dependent, pre-defined frequencies). Anyway, a proper coordination gets even more difficult then, some users therefore tend to not make use of this functionality. If the patient now presses the electrolarynx device to his/her throat these vibrations are transmitted to the vocal tract to generate the fluctuations in the passing air stream's pressure. Controlling the device's activity periods can be done with a push button on the electrolarynx. As alternative there are also electrolarynx derivatives on the market that directly alter the air pressure in the air channel. To do so the device uses a small tube which is fit into the corner of the mouth. Through this tube air bursts can directly be injected in the air channel without having to alter it's internal volume.

A proper usage of an external electrolarynx can be tricky because the user has to find a location at the throat that optimally forwards the vibrations. This place is referred to as

sweep spot. Of course this spot should also be comfortable for the user to reach at first to place the electrolarynx on and it at second should not start hurting after a longer period of usage. The coordination between spoken words and switching the electrolarynx on and off is also rather difficult at the beginning and needs some training to work fluidly. Same is the case for accenting which can be achieved through varying the device's loudness and frequency as mentioned above if available for the used model.

The overall resulting speech will sound more or less the same independent of the used electrolarynx type. The vibration frequency is constant, leading to a constant pitch of the resulting voice. This sounds somehow artificial because the naturally produced pitch contour slightly varies in time depending on the spoken phoneme. The electrolarynx voice therefore produces a sounding often referred to as "mechanic" or "robotic".

2.2 Speech Signal Processing Techniques

Speech signal processing techniques are tools based on mathematical principles used to analyse, process and alter speech signals. A good overview about this topic can be found in [29]. This book focuses on the *discrete-time* sub-family of signal processing techniques which is also dealt with in this work. In fact every **D**igital **S**ignal **P**rocessor (DSP) application just works with discrete-time data samples due to the digital nature of the signal processor - continuous data as produced by most sensors observing effects in nature is converted by the DSP's **A**nalog-To-**D**igital (A/D) converter. The DSK6713's **A**nalog-To-**D**igital **C**onverter (ADC) is the AIC23 audio codec which is discussed in chapter 2.3.3.

2.2.1 Block Processing

In principle the *block processing* or *framing* paradigm can be described as temporal segmentation of the **I**nput / **O**utput (I/O) data stream in *blocks* or *frames*. This process is helpful in the following processing steps to operate on both definite segments and pieces of data where all included data values are available at the same time. If so the data values can be operated on in one cycle with also having access to their temporal evolution despite of executing the processing step at a single time instance. This is a fundamental need in lots of signal processing applications because signals do not just contain information in the actual data value but also in the developing of the overall signal defined by subsequent samples.

Anyway, this principle comes with the drawback of artificially delaying the processing of samples. Because they all are concentrated and handled at once a single sample can never immediately pass through the processing chain. Actually the first sample in the frame has been captured BLKSZ−1 time slots before the frame is processed, the second sample BLKSZ−2 times, ... This leads to an occuring delay for single samples of between the maximum as given in equation (2.1) for the first sample in the frame and $0.0s$ for the last sample.

$$t_{delay,max} = (BLKSZ - 1) * \frac{1}{f_s} \tag{2.1}$$

To provide a smooth transition between blocks, neighbouring ones are usually overlapped along a definite length, the *overlapping factor* which is given in percent. Overlapping data segments

are summed up value by value, the resulting value then represents the actual input data value for the given time instance. Figure 4.2 shows an example of three sequenced blocks overlapped by a factor of 50%. All data values in one block are applied with a "focusing factor" defining the importance of a certain sample in the current block. In grouping all of these factors used in a single frame together one gets the so-called *window function*. Typical window functions are *Hamming*, *Hann* or *Triangle* windows ([29]). If the window function and overlapping factor are chosen properly all overlapping, summed-up values at any time instance result in a windowing factor of 1.0 - every overlapped data values' sum equals the original input value.

Let's have a closer look at this problem using an example: The overall signal $x[t]$ consists of 20 samples $x[0] \dots x[19]$. Choosing a block size of four with 50% overlapping this results in nine complete block ({ $1 \dots 4]$, $[3 \dots 6]$, $\dots [17 \dots 19$}). The window function is chosen to be a triangular-like one, thus

- the 1st value of every block is applied with factor $w_1 = 0.25$,
- the 2nd sample with $w_2 = 0.75$,
- the 3rd sample with $w_3 = 0.75$ and
- the 4th sample with $w_4 = 0.25$.

So when for example the data value $x[14]$ at time instance 14 has to be reconstructed it can be seen that two blocks exist at this specific time - block b_7 containing samples $\{x[11]_7 \dots x[14]_7\}$ and b_8 containing $\{x[13]_8 \dots x[16]_8\}$. Sample $x[14]$ is now reconstructed with adding both overlapping samples which are applied with their appropriate window function as shown in equation 2.2.

$$\begin{aligned} x[14] &= x[14]_7 + x[14]_8 = w_4 * x[14] + w_2 * x[14] \\ &= 0.25 * x[14] + 0.75 * x[14] = 1.0 * x[14] \\ &= x[14] \end{aligned} \tag{2.2}$$

2.2.2 Linear Predictive Coding

Linear **P**redictive **C**oding (LPC) is a technique that tries to predict future data samples through processing past ones. The amount of past samples taken into consideration for the calculation is called the LPC's degree or order. The predictor itself is mathematically described as shown in equation (2.3), here for an order of three. The fact that the predictor is purely linear can also be seen clearly in this equation - past data samples $(x[n-1], x[n-2], \dots)$ only appear as first order factors.

One widely spread algorithm is based on calculating the autocorrelation of the input signal and using it to determine the coefficients of a chosen model, the wanted polynomial, using **M**inimum **M**ean **S**quared **E**rror (MMSE) approximation. The gathered result - something that all LPC implementations have in common - are the coefficient values a_α. Inserted in equation (2.3) these values can now be used to predict future data $(x[n+1], x[n+2], \dots)$ based in the processing of previous samples.

$$x[n] = a_1 * x[n-1] + a_2 * x[n-2] + a_3 * x[n-3] \tag{2.3}$$

2.2.3 Pitch-Synchronous Overlap-Add

The **P**itch-**S**ynchronous **O**ver**l**ap-**A**dd (PSOLA) technique offers the possibility to alter a signal's pitch without actually changing the duration. The original algorithm, nowadays referred to as time-domain or **T**ime **D**omain **- P**itch-**S**ynchronous **O**ver**l**ap-**A**dd (TD-PSOLA) was first invented by the French Telecom. It is based on working with *fragments* which are small, overlapping parts of a signal. The TD-PSOLA technique now consists of three main steps:

1. Identifying and shaping the fragments based on the pitch-synchronous marks
2. Generating the new pitch-marks based on the target frequency
3. Placing the fragments at the new pitch-marks using **O**ver**l**ap-**A**dd (OLA)

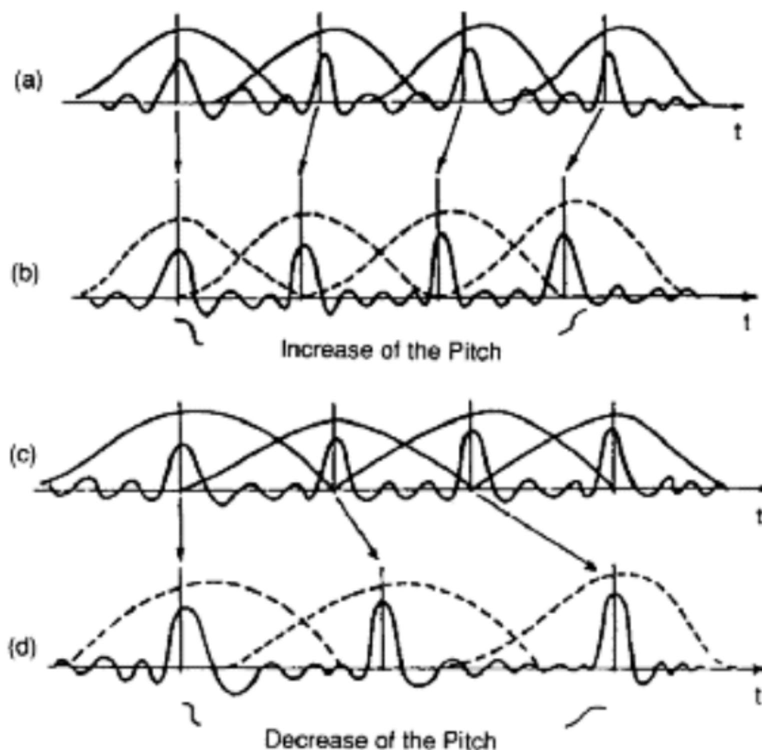


Figure 2.4: *Working principle of the TD-PSOLA algorithm that uses overlapping of windowed fragments in order to* (b) *increase or* (d) *decrease the signal's pitch. For raising it the space between fragments is decreased, otherwise increased (source: [24]).*

Pitch-marks are placed along the input signal spaced by the signal's original pitch (therefore the term pitch synchronous). These pitch-marks identify the center of a fragment, the length l_{frag} is dependent on the degree of overlapping. Usually 50% overlapping is used. The fragment length l_{frag} is therefore $2*(t_{frag_{n-1}} - t_{frag_n})$ with t_{frag_n} indicating the time instance of the fragment number n at the new pitch position. This results in 50% overlapping of neighbouring fragments similar as in block processing (Sect. 2.2.1). To shape the fragments they are applied with a window function, usually a triangle or Hamming window.

The new pitch-marks are calculated by placing marks spaced by the new pitch period $\frac{1}{f_{new}}$ along the signal. The - now shaped - fragments are again placed at the center of these marks. Due to l_{frag} being twice the space between two new pitch-marks they exactly overlap by one half.
With shaping the fragments, now spaced by the original pitch, according to the new pitch it may happen that some information at the fragment border is lost due to the signal being zeroed there. This may happen if the window coefficient reaches 0.0 but is no problem and actually wanted: With overlap-adding the segments, again spaced by the new pitch, the fragments come closer together without artefacts in between - what makes the pitch increase as wanted. On the other hand it might be that the fragments are shaped with a hull curve that is larger than the actual space between the pitch-marks. This makes areas of the signal belong to more than one fragment. But the same way as with shorter windows when again overlap-adding the phonemes at the new pitch-marks there are no artefacts between the newly positioned fragments because of the windowing - this leads to the fragments drift away from each other without resulting artefacts in between the neighbours. The pitch decreases.

A visual representation of the algorithm is shown in figure 2.4.

2.2.4 Pitch-Marking

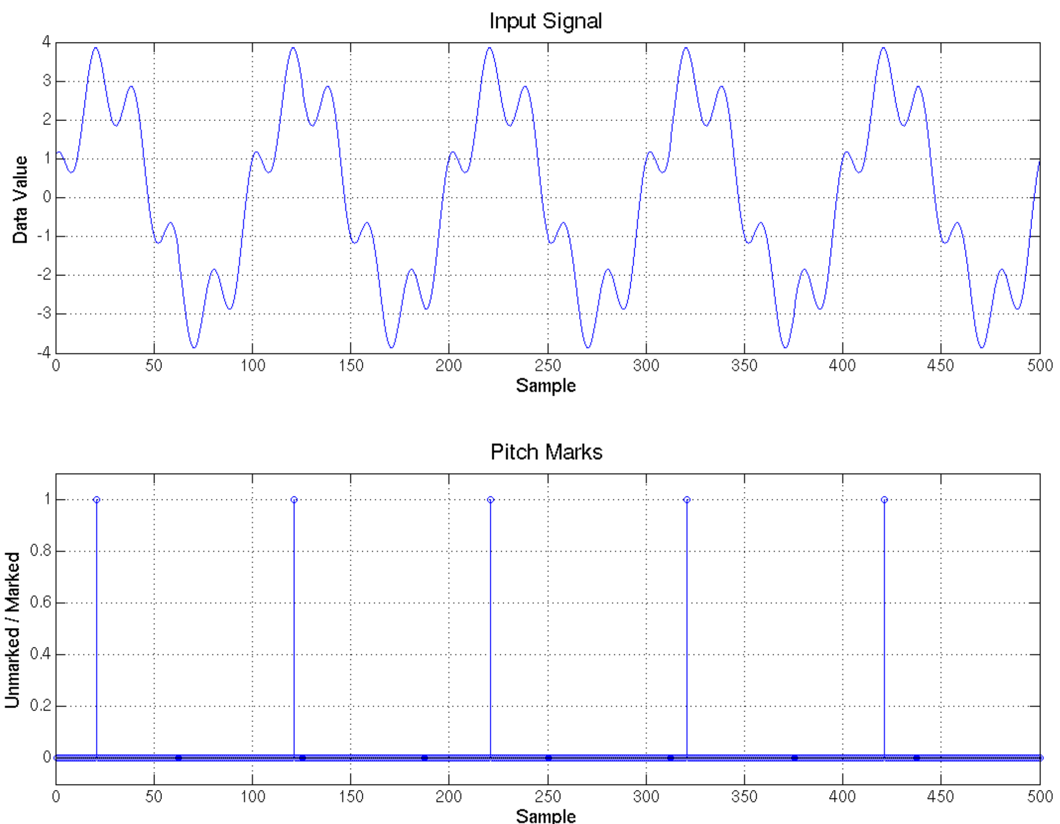


Figure 2.5: *Example of a random signal and it's corresponding pitch-marks that are spaced according to the signal's pitch. The markers themselves are located at the source signal's peak positions.*

Pitch-marking simply refers to the technique of applying timestamps to a signal that are spaced by the signal's pitch. Pitch-marking itself does not process a signal in any way but just adds meta-information to the data stream. This is why pitch-marking is primary used as a pre-processing step to further signal processing techniques such as PSOLA (Sect. 2.2.3). Figure 2.5 shows an example of some input signal and it's corresponding pitch-marks located at the signal peaks and spaced by it's pitch.

2.2.5 Pitch Tracking

A signal's pitch is - maybe besides the loudness of the signal - it's main and most obvious property. Therefore this parameter has been under inspection for over a century meanwhile resulting in a huge amount of possibilities to detect, process and alter it. A lot of those techniques are still valid and commonly used nowadays. An interesting approach when taking about pitch analysis is pitch tracking which is a way to detect the time developing of a signal's pitch. Usually this is done by short-time analysis of the signal's spectrum. A trivial example of a pitch tracking algorithm is *peak detection* in the frequency domain. Anyway this technique is not very exact due to the high peak of the harmonics caused by the framing procedure. This may create harmonics with higher energy than the pitch itself leading to a misinterpretation of harmonics as the signal's actual pitch.
Especially real-time applications, due to their limited processing power, frequently use the *zero-crossing detection* for this reason. One does not have to calculate the signal spectrum and can directly determine the pitch through analysis in the time domain. With zero-crossing detection a signal's pitch is detected by normalising the signal data of every frame (to remove the bias) and count the zero-crossings in the current frame. Knowing the crossing amount $n_{crossings}$, frame length BLKSZ and sampling rate f_s it is easy to calculate the pitch F_0 with this information (Equ. (2.4)).

$$F_0 = \frac{f_s}{\mathtt{BLKSZ}} * n_{crossings} \tag{2.4}$$

For applications that depend on high accuracy pitch values zero-crossing might be to inaccurate - small variations around zero are very common due to noise and might cause double-counts. Therefore nowadays many pitch detectors are based on autocorrelation methods. Purely periodic signals result in the autocorrelation peak at shift zero and (dependent on the used frame size) slightly smaller peaks spaced by the signal period. So finding the highest peak is a well-working approach to detect the signal's pitch even in noisy signals.

2.2.6 Formant Tracking

The principle idea of formant tracking is the same as for pitch tracking - the trajectory of the appropriate signal parameter, in this case the formant, is determined. The basic idea of the algorithms for formant tracking and pitch tracking are the same therefore, only differing by the way the parameter under investigation is calculated. However the short-time analysis and time-domain investigation of the found parameter values remains the same.

To calculate a signal's formant frequencies it can be chosen between several algorithms that are available nowadays. A famous because easy yet accurate one works by finding the local maxima in the transfer function approximated by a LPC analysis ([7]). This approach advances a simple **F**ast **F**ourier **T**ransform (FFT) because of the possibility to choose the

accuracy via LPC order. Besides a LPC analysis is by nature most accurate at the peaks which are the sections of interest for formant detection anyway. These peaks indicate the positions of resonant frequencies in the transfer function, thus the formants of the signal under investigation. The LPC approach also produces smoother results than the FFT approach. Nevertheless implementing a formant detector for real-time applications based on LPC approximation might be too slow for the time critical audio signal processing sector. Especially the polynomial roots calculation is very slow and therefore might be leading to timing issues. An interesting alternative capable of dealing with these drawbacks is the **I**nverse **F**ilter **C**ontrol (IFC) based formant tracking introduced by [47] and [45]. By using inverse filters it is possible to suppress specific signal components from the input to analyse dedicated signal parts separately. The IFC method makes heavy usage of this fact by mutual suppression of specific resonant components to find other ones. For example the inverse filter is used to suppress the frequencies surrounding the first formant to detect the peak at the second. By recursively calculating the found formants and damping them to find neighbouring formants it is possible to very accurately determine the searched formants' frequencies by simple time-domain filtering methods. The formant frequency estimation itself does not have to be time intense because, due to the recursive calculation, the values gets more and more accurate with every iteration even when using simple resonant frequency detectors like zero-crossings and peak detectors.

2.3 DSK6713 Overview

2.3.1 Introduction

The *DSK6713* is a digital signal processing board, manufactured by *Spectrum Digital*, using the *Texas Instruments TMS320C6713B* floating-point signal processor. It is equipped with a $16bit$ audio codec supporting sampling rates from $8kHz$ to $96kHz$ making the kit very suitable for audio processing applications. Further components are internal $512kB$ non-volatile flash memory as well as $16MByte$ **R**andom **A**ccess **M**emory (RAM) memory, four **L**ight **E**mitting **D**iode (LED)'s and **D**ual **I**nline **P**ackage (DIP) switches for simple controlling tasks and an **U**niversal **S**erial **B**us (USB) interface for programming the DSP via connected **P**ersonal **C**omputer (PC). For internal data exchange the **D**igital Signal Processor **S**tarter **K**it (DSK) features an interrupt-based signalling system with **D**irect **M**emory **A**ccess (DMA) support.
The programming environment is a *Microsoft Windows* based software called **C**ode **C**omposer **S**tudio (CCS) and will be focused on in section 2.3.5. The CCS offers the possibility to program and debug the DSP board using the programming language C via USB interface.

2.3.2 Digital Signal Processor

The DSP, responsible for all data processing tasks, is a TMS320C6713B floating-point signal processor. It works at an operating frequency f_{CPU} of 225MHz - this property plays an important role in the development process of this framework because it determines the computing speed, the one property determining the real-time ability of the final application. This can be explained by the clock rate directly affecting the amount of operations that are available in a certain time span, in this case the duration of one frame. Please refer to table 2.2 for exact values.
The internal **C**entral **P**rocessing **U**nit (CPU) architecture is **V**ery **L**ong **I**nstruction **W**ord (VLIW)-based making it possible for the programmer to access all of it's eight

functional units manually and at once via appropriate assembler command calls. This can - at least theoretically - lead to a speed-up of *eight* at maximum if all functional units operate completely parallel. In practice is a 100% load all functional units simultaneously over a long period not possible due to data dependencies (proper partitioning is not possible) and the inability of a single functional unit to perform all accumulated operations[1]. As alternative to manually restructuring assembly code, the compiler can be utilised for optimising the source code to use the processing pipelines as parallel as possible. More on writing source code for optimal CPU load is found in [44]. The internal structure, including the eight functional units *L1*, *M1*, *S1*, *D1* and *L2*, *M2*, *S2* and *D2*, are portrayed in figure 2.6.

Further information about the TMS320C6713B can be found in the appropriate data sheet [40].

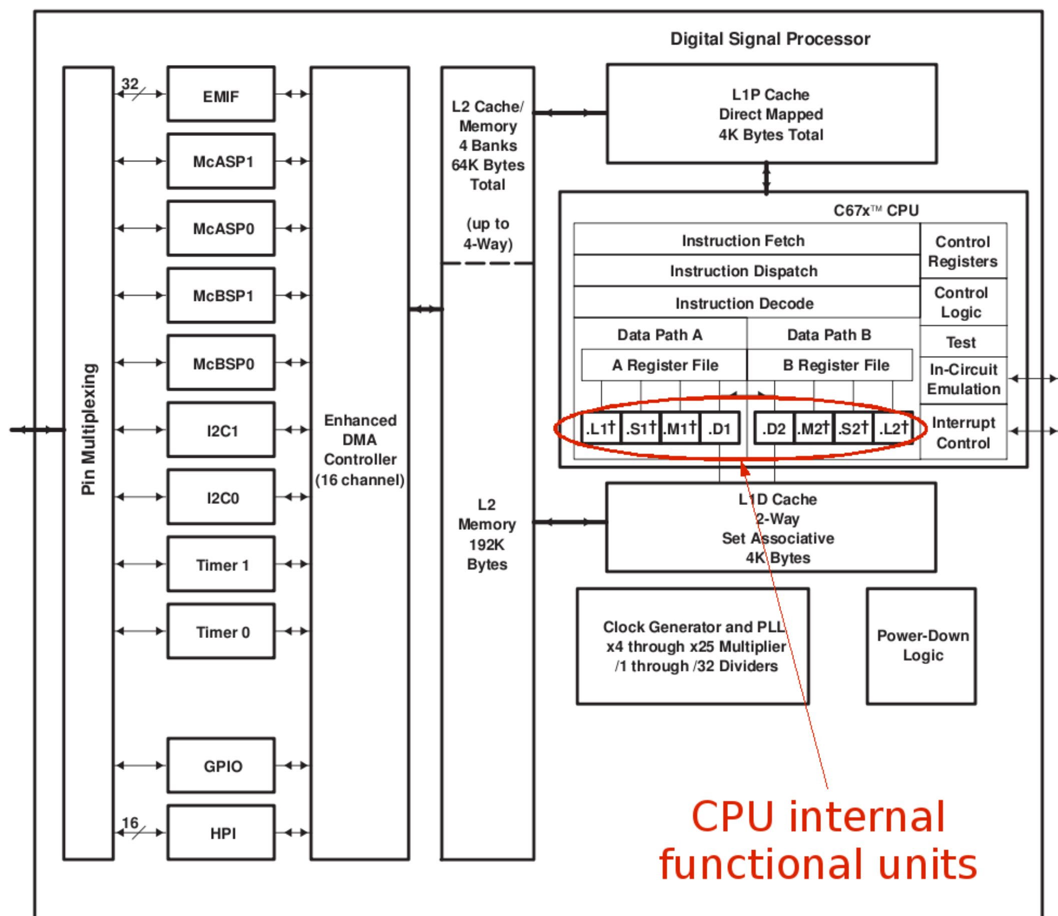


Figure 2.6: *Internal structure of the TMS320C6713B DSP showing the eight functional units of the C6713 CPU. Those units can be utilised for performing calculations simultaneously (source: [40]).*

[1]The functional units are organised in pairs with every pair being of the same type and therefore capable of performing the same set of operations: *Lx* units work as ALU's, *Mx* units are used for MAC operations, *Sx* units perform branching and bit manipulation, *Dx* units execute fixed-point operations ([12, p.76]).

Measurement	Value
Floating-Point Operations	1350 MFLOPS
Instructions	1800 MIPS
Multiply-Accumulate Operations	450 MMACS

Table 2.2: *Reference values indicating the DSP's operating speed.*

2.3.3 Audio Codec

The audio codec used in the DSK6713 is a *TLV320AIC23*. It is used for audio data I/O and therefore features an **A**nalog-To-**D**igital **/** **D**igital-To-**A**nalog (AD/DA) converter which is based on the sigma-delta working principle. The connection interface is realised using 3.5mm audio jacks whereas input devices may be connected to the line or microphone input, line or headphone plugs are used for output. When wiring these interfaces is has to be kept in mind that at the same time only one of the two input and output plugs can be used. Accessing both interfaces leads to the automatic deactivation of one interface.

An overview about the codec's technical specifications is given in table 2.3. The provided values are taken from [37]. This manual also specifies the communication protocol and electrical characteristics that will not be discussed in this paper.

Overall Characteristics	
Type	high performance stereo codec
Internal System Clock	$12MHz$
Nominal I/O Level	$1V_{RMS}$
Cut-Off Frequency (-3dB)	$3.7Hz$ ($44.1kHz$), $4.0Hz$ ($48kHz$)
A/D Converter Characteristics	
Sampling Rate	$8kHz$ to $96kHz$
Input Impedance (Typical)	$35k\Omega$ (line), $14k\Omega$ (microphone)
SNR	$90dB$ maximum
D/A Converter Characteristics	
Amplitude Resolution	8, 10, 12 or 16 bit
Output Impedance (Minimal)	$10k\Omega$ (line), $16/32\Omega$ (headphone)
SNR	$100dB$ maximum

Table 2.3: *Specifications of the DSK's AIC23 audio codec.*

2.3.4 Memory Management

The complete DSK consists of three levels of memory:

1. CPU cache
2. External flash memory
3. External **D**ynamic **R**andom **A**ccess **M**emory (DRAM)

The smallest but fastest CPU cache is included directly in the DSP chip. It includes $8kB$ of L1 cache ($4kB$ program and $4kB$ data) as well as $256kB$ L2 cache. More interesting for the programmer are the external memory components: The $512kB$ flash and $16MB$ DRAM can be directly used for storing program data such as sine tables. In this framework, coefficients of **F**inite **I**mpulse **R**esponse (FIR) and **I**nfinite **I**mpulse **R**esponse (IIR) filters as well as the window and excitation pulse data and FFT twiddle factor tables are stored in this memory.

Texas Instruments provides the possibility of mapping different types of data to different memory modules for it's DSP processors. This feature can be used for speed optimisation: The programmer is usually aware of the used data block's access frequency. The more intense the **R**ead **/** **W**rite (R/W) operations on this data block are the better it would be to place the block in a memory module that is accessible in the fastest possible way. Besides speed optimisation the memory region size is a major role-playing factor when performing the memory mapping. Regions with high memory demands have to be placed in large modules for being able to provide the needed amount of memory - a wrong mapping leads to non-executable programs because either the allocation of variables can not be completed or the CCS is not even able to transfer all parts of the developed program to the DSP.

Table 2.4 contains a list of available memory regions that can be mapped to the appropriate memory modules. The mapping used in this framework is explained in section 4.3.2.

Shortcut	**Memory Region**
`.vec`	interrupt vectors
`.text`	executable program code
`.data`	data from Assembler programs
`.cinit`	tables for initialising variables and constants
`.stack`	stack for local variables
`.const`	initialised constants
`.bss`	global and static variables
`.sysmem`	dynamically allocated data
`.far`	global and static variables declared `far`
`.switch`	tables for switch instructions
`.tables`	other table data
`.cio`	needed as buffer for C I/O functions

Table 2.4: *List of data regions that can freely be mapped to the available DSP memory modules like flash and DRAM.*

2.3.5 Programming the DSP Starter Kit

Development Environment

The preferred tool for developing DSP software is the **C**ode **C**omposer **S**tudio (CCS) development environment, provided by *Texas Instruments*. It is an all-in-one solution offering the possibility to code, build and transfer programs to various types of Texas Instrument's processors. The internal compiler is already enhanced with several levels of code optimisation routines to achieve low memory demand as well as high performance (see [44]). The coding is done using either the low-level Assembler or high-level C programming language. If needed, a

mix of both languages is possible for example with calling Assembler functions within the C code. This approach is a good choice when frequently called or calculational intense functions have to be hand-optimised in Assembler but used within a C environment.

An elaborate overview about the possibilities and usage of the development environment is given in [42], this section focuses on functionality mainly.

Similar to comparable solutions like *Microsoft's Visual Studio* Texas Instruments' Code Composer studio offers various possibilities to debug developed code. The debugging functionality can even be accessed in real-time thus during the code is executed on the DSP. This is a quite convenient when processing I/O data like data samples read by the audio codec. The debugging itself is done with setting break and probe points and checking the current register and variable contents. Break points are already well-known throughout the software development sector but probe points might not be. They slightly differ from break points targeting on meeting up with demands that arise in real-time coding. With probe points the code execution does not stop when passing the marked line - only a snapshot of the current state is given. So every time the execution passes a probe point the development environment updates the variables linked with this probe in the **G**raphical **U**ser **I**nterface (GUI) and continues execution. An additional advantage especially for signal processing applications is that besides simply printing variable contents, it is also possible to plot array contents in form of two-dimensional diagrams. This simplifies the process of following signal changes throughout the processing chain. The method of counting CPU cycles for certain intervals - defined by the break points that are passed in the execution process - is yet another additional tool with special focus on creating DSP applications. CPU cycle counting might be used as a tool to exactly analyse the calculational complexity of program sections. In building real-time applications this can be an approach to find and reduce bottlenecks.

C67xx Function Libraries

When developing applications for hardware devices it is the common case for manufacturers to ship the hardware together with some kind of software **A**pplication **P**rogrammable **I**nterface (API). This eases up work by not having to implement communication protocols and other low-level routines when accessing hardware modules. The DSK6713 includes such an API as well, called **C**hip **S**upport **L**ibrary (CSL). An abstraction of several CSL routines with focus on board-level routines is the **B**oard **S**upport **L**ibrary (BSL). This library simplifies the access to several higher-level DSP board functionality by utilising the CSL. Finally the **D**igital **S**ignal **P**rocessor **Lib**rary (DSPLib) can be integrated in software projects. The DSPLib includes common signal processing functionality using processor-optimised calculation routines.

The mentioned libraries are available in form of pre-compiled `*.lib` files that can be included in CCS projects using the appropriate project configuration setup.

Peripheral Support Language **P**eripheral **S**upport **L**ibrary (PSL) is a term combining both **C**hip **S**upport **L**ibrary (CSL) and **B**oard **S**upport **L**ibrary (BSL). As mentioned before the CSL is a software API written in C for accessing low-level routines to control on-chip peripherals. These peripherals include:

- **E**xternal **M**emory **I**nter**f**ace (EMIF)
- Cache
- Interrupts
- **(E**nhanced **) D**irect **M**emory **A**ccess ((E)DMA)
- Power logic
- Timers
- ...

A complete list of controllable modules as well as included functions and their usage is given in [39].

Equal to the CSL, the BSL API implements functions for abstracting hardware functionality written in C. Contrary to CSL functions BSL functions are used to control board-level peripherals ([38]). The BSL builds up upon the CSL, it can for this reason be considered to be a higher-level of abstraction. BSL functions might be used to

- perform board initialisation and reset,
- access the flash memory,
- control LED states,
- read DIP settings and
- control the audio codec.

Digital Signal Processor Library The **D**igital **S**ignal **P**rocessor **Lib**rary (DSPLib) is a function library including target-optimised general purpose signal processing functions. The target optimisation is done at assembly level, the functions themselves are C callable. Due to the target-DSP optimisation there are several versions of the DSPLib available, one for each supported processor. The library's functions are run-time and memory optimised.

Implemented in the library are:

- Correlation
- Fourier transform
- Filtering and convolution
- Adaptive filtering
- Mathematical routines (vector sum, scalar product, ...)
- Matrix based routines (matrix multiplication, transposing, ...)
- Utilities (memory swapping, format conversions, ...)

Because the TMS320C6713 is a floating-point processor it's DSPLib includes a single-precision and double-precision version for most of these functions. Additionally, for a full list of available functions and their usage please refer to [41].

Chapter 3

Design

With knowing the introduced basics it is now possible to design the enhancement routines. At a glance two different approaches are needed whereas either one intends to handle on of the two drawbacks which were discussed in chapter 1: The suppression of the **D**irect-**R**adiated **E**lectro**l**arynx Noise (DREL) component and the new contour generation for the signal's pitch. DREL suppression is going to be realised using a multipath separation approach. Using this technique the two signal paths speech signal and DREL noise can be parted and the noise component dampened. This might be done with either operating in the time domain (modulation spectral filtering) or in the frequency domain (spectral subtraction). The pitch contour generation module uses several analysis routines to determine necessary signal properties for generating a new contour and utilises them to properly synthesise the output.

3.1 Multipath Separation

Multipath **S**eparation (MS) refers to the technique of separating various signals or signal parts from a data stream consisting of a combination of these signals. This usually happens when a signal is generated by components from different sources that add up to a single data stream before reaching the receiver. One application making heavy use of those algorithms is noise suppression: Every real-world system is affected by the presence of noise so the transport of analog information between sender and receiver is always corrupted by an additional noise component. With digital data the noise component might be present in the signal as well but does not necessarily affect the transported information due to discretisation.
Communication channels showing the behaviour of being **A**dditive **W**hite **G**aussian **N**oise (AWGN) channels are characterised by an additive noise component $n(t)$ which is added to the original signal $s(t)$ and results in the final signal $x(t)$ arriving at the receiver (Equ. (3.1)). MS techniques can now be utilised to identify signal components of either being signal or noise components (thus signal and noise are separated). With finally damping the identified noise components the receiver is able to estimate the data signal $\hat{s}(t)$ out of the received signal $x(t)$.

With applying the MS technique to the current application the data signal $x(t)$ is equal to the speech signal (fine-dotted line in figure 3.2) and the (to be suppressed) noise component $n(t)$ is the DREL component. Both signals originate from a different source - the speech signal from the speaker, the DREL signal from the electrolarynx device - but are spread through the same communication channel, the air between speaker and listener. This makes the two

components overlap and be recognised as single signal at the listener's ear. Unfortunately in doing so the additive DREL component corrupts the speech signal and therefore decreases the intelligibility. With using one of the two multipath separation approaches mentioned below it is possible to again suppress the DREL component at the receiver in order to support the listener in understanding the words spoken by his/her conversation partner, the electrolarynx user.

$$x(t) = s(t) + n(t) \tag{3.1}$$

3.1.1 Modulation Spectral Filtering

Modulation **S**pectral **F**ilter (MSF) is a signal processing technique that analyses and utilises the temporal evolution of a signal's spectral components. MSF performs the processing of specific spectral components by applying a filter to their temporal developing that accordingly alters the components. In order to dampen temporally constant spectral components as needed in this application (the DREL component has this behaviour) this filter must show a high-pass behaviour. Filters working that way result in a suppression of frequencies that are constant in time or at least very low. This can be thought of as a filter that suppresses a constant tone like when blowing a whistle but let's pass an altering signal like a siren without taking influence. See figure 3.1 for a comparison between signals with constant and varying trajectory.

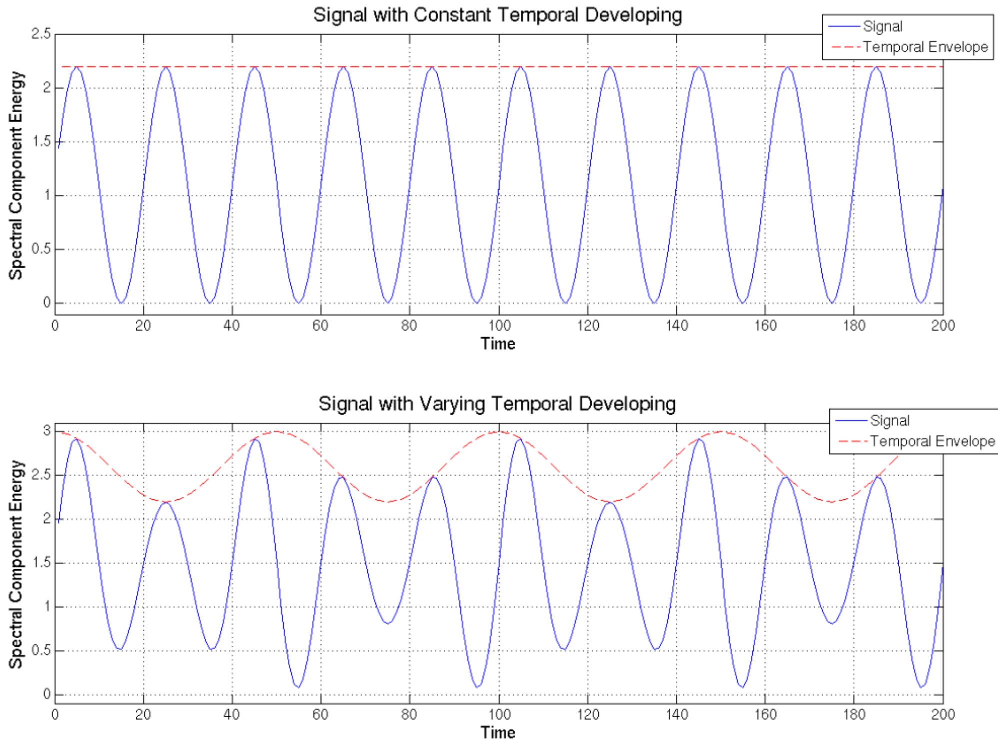


Figure 3.1: *Signal components showing a temporal envelope that is constant in time (upper picture) as well as a varying one (lower picture). The signals themselves are printed in blue, their envelopes in red.*

Due to their behaviour modulation spectral filters may be used to suppress DREL noise. The main property of DREL noise is it's constant-in-time energy so filtering a DREL affected signal with a MSF results in a signal with suppressed DREL component. Nevertheless this approach cannot be applied to signals other than speech because e.g. music consists of various sound sources that might also show signal components with time-invariant character. Those sounds would be heavily affected by the filter - the noise suppression filter would also take influence on the clean signal and not just the noise! With speech input this is not the case: The typical speech signal does not consist of components with a lower frequency than about $2Hz$. So when adjusting the filter in a way that it just mutes components with higher frequencies the clean speech signal will just be minimally altered by the modulation spectral filter with removing (mainly loudness-related) constant parts but keeping the actual speech information. The loudness discrepancy effect will also be focused on in section 5.1.2.

For further information about the process of suppressing DREL noise please refer to [20]. Detailed information about the processing algorithm and implementation can be found in chapter 4, the principle block diagram is shown in figure 3.2.

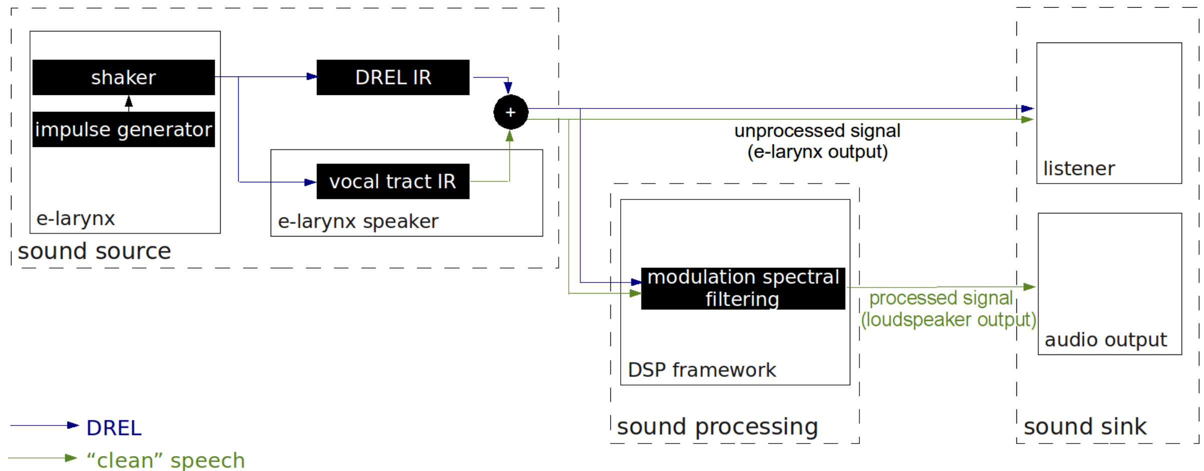


Figure 3.2: *Principle block diagram of the MSF-based MS method. The speech signal is generated in the* sound source block *by filtering the e-larynx speaker's spoken words with the basic oscillations produced by e-larynx. This signal is afterwards enhanced by the* sound processing block *with suppressing it's DREL component and forwarded to the* sink.

3.1.2 Spectral Subtraction

Spectral **S**ubtraction (SS) (see [26]) is a signal processing method, utilisable by the MS approach, that works in the frequency domain. It describes an algorithm that suppresses specific spectral components without touching the remaining components. In terms of noise suppression the frequency pins to be dampened are determined with estimating the noise spectrum. Obviously signal and noise are not allowed to be correlated for being able to exactly part the pins. So SS is mainly used to remove noise components from a signal that are located at definite frequencies. Problems might occur when the clean signal also consists of spectral components in this band because they are suppressed as well. The noise estimation would not work properly in such a situation.

In [9] this method for reducing noise within speech applications was first introduced. The article gives the following description of SS: *In spectral subtraction, an estimate of the average noise magnitude spectrum is subtracted from the short-time speech-plus-noise spectrum to give an improved estimate of the speech signal.* This approach explicitly works with the spectral pin's magnitudes - the phase will be kept untouched. See equation 3.2 for a mathematical description with $S(\omega)$ standing for the approximated speech signal, $N(\omega)$ the noise estimate and $X(\omega)$ the noisy input.

$$S(\omega) = \sqrt{|X(\omega)|^2 - |N(\omega)|^2} * e^{j*arg(X(\omega))} \tag{3.2}$$

Again detailed information about the processing algorithm and implementation can be found in chapter 4.

3.2 Pitch Contour Generation

Pitch Contour Generation (F0gen) is an approach to enhance the quality of electrolarynx speech with avoiding the strange-sounding constant speech's pitch. The idea was first published by [20] and makes use of the relation between formant and pitch in the speech prosody.

The principle workflow consists of analysing the input signal to parse out the formant information, post-process it and finally alter the (constant) pitch according to the resulting formant contour. This alteration can either be done by direct appliance it to the captured speech via **P**itch-**S**ynchronous **O**ver**l**ap-**A**dd (PSOLA) methods (Sect. 2.2.3) or by creating a feedback loop by controlling the electrolarynx vibration frequency by generating pitch contour dependent driving pulses for the electrolarynx device. When directly altering the speech signal it is also necessary to somehow output the generated signal - this is either done via loudspeaker or by replacing the original with the enhanced signal when using an alternative communication channel like a telephone et cetera. In fact this might even be the preferred method because a direct output with loudspeakers would interfere with the original signal and lead to a degrading intelligibility. This problem does not occur when using the framework to control the electrolarynx operating frequency because the feedback affects the speech signal immediately. Figures 3.3 and 3.4 illustrate the difference between the two operating modes. The mentioned post-processing consists of contour smoothing, outliner detection and scaling operations.

Detailed information about the processing algorithm and implementation can be found in chapter 4 as well.

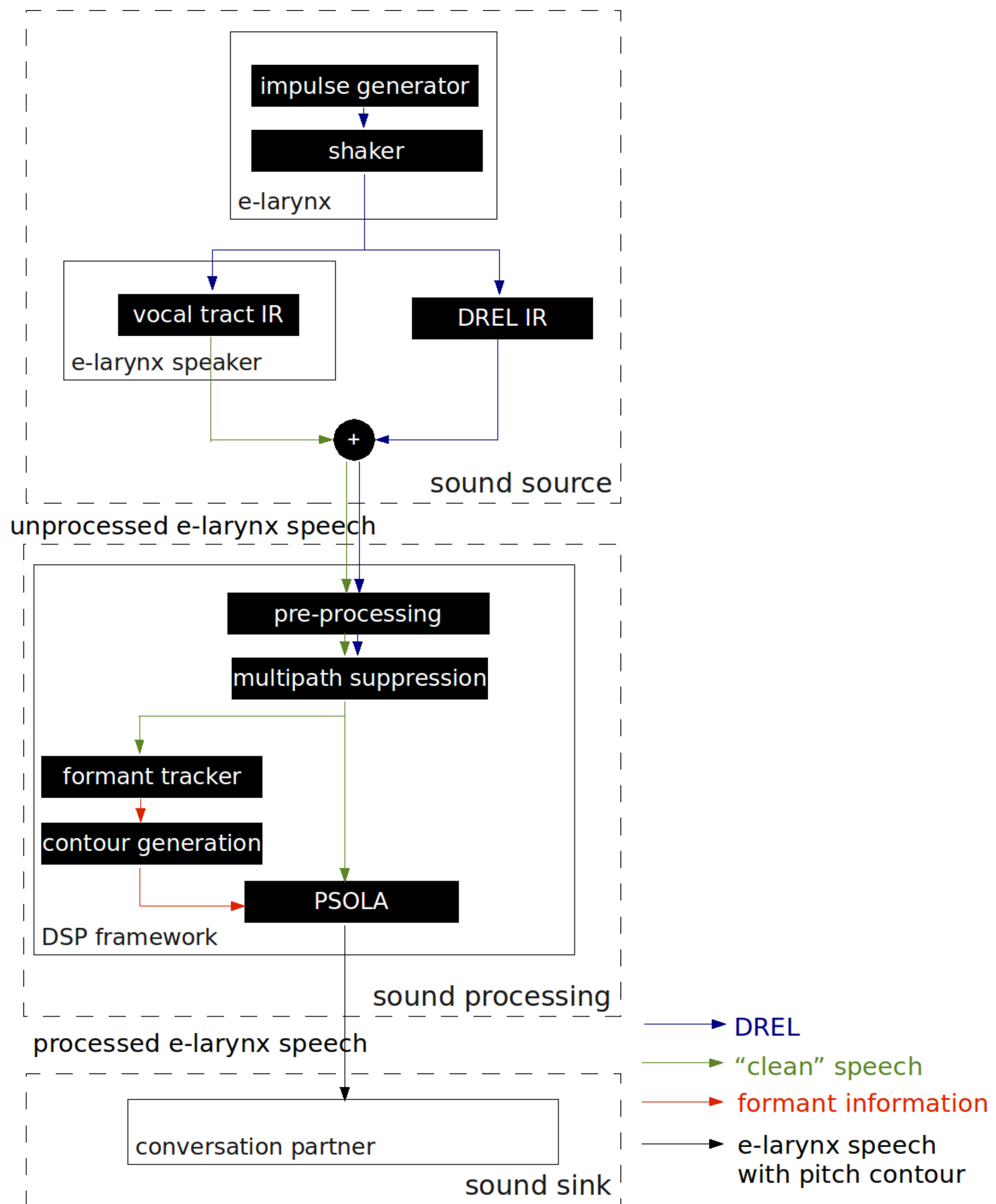


Figure 3.3: *Principle block diagram of the pitch contour generator operating in* "processed signal output" *mode. The speech signal is generated in the* sound source *block, enhanced through DREL suppression (*MS block*) and equipped with a suitable pitch contour (*F0gen blocks*) in the* sound processing unit *and finally passed to the* sound sink*.*

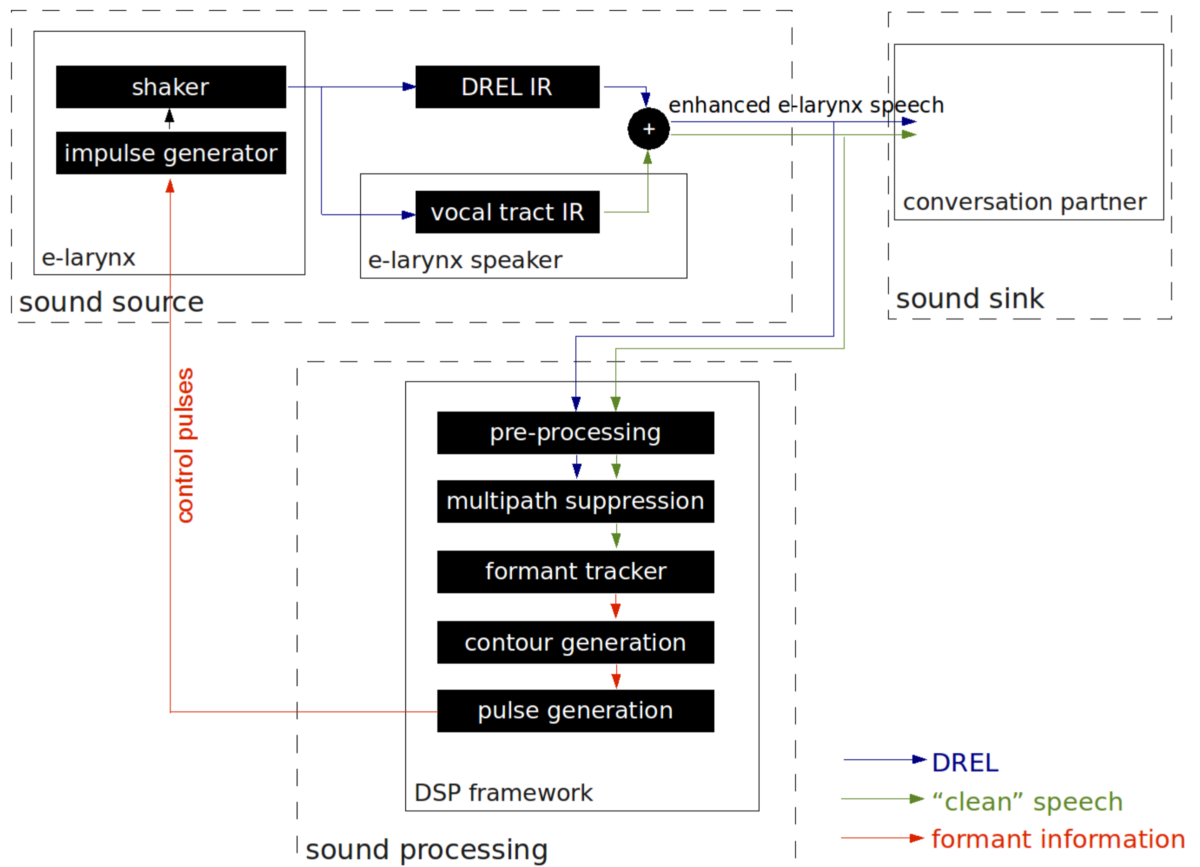


Figure 3.4: *Principle block diagram of the pitch contour generator operating in* ”driving pulses output” *mode. The speech signal is generated in the* sound source block *with the appropriate* F_0 *driving the shaker whereas the* F_0 *calculation and pulse generation are executed in the* sound processing block. *The contour-equipped e-larynx speech signal works as final output as perceived by the conversation partner as well as input for the* sound processing block *in order to perform the enhancement continuously.*

Chapter 4

Implementation

The implementation chapter shall give a deeper and more detailed view about the final realisation of the **D**igital **S**ignal **P**rocessor (DSP) framework. The main focus lies on the implementation of the different processing algorithms, their operating principles, setup and usage. The configuration of the used hardware - the DSK - will also be discussed in terms of AIC23 data exchange and memory management. Additionally relevant factors caused by the effects of real-time processing that influence the development will be presented as well as approaches to handle them properly.

4.1 Experimental Setup

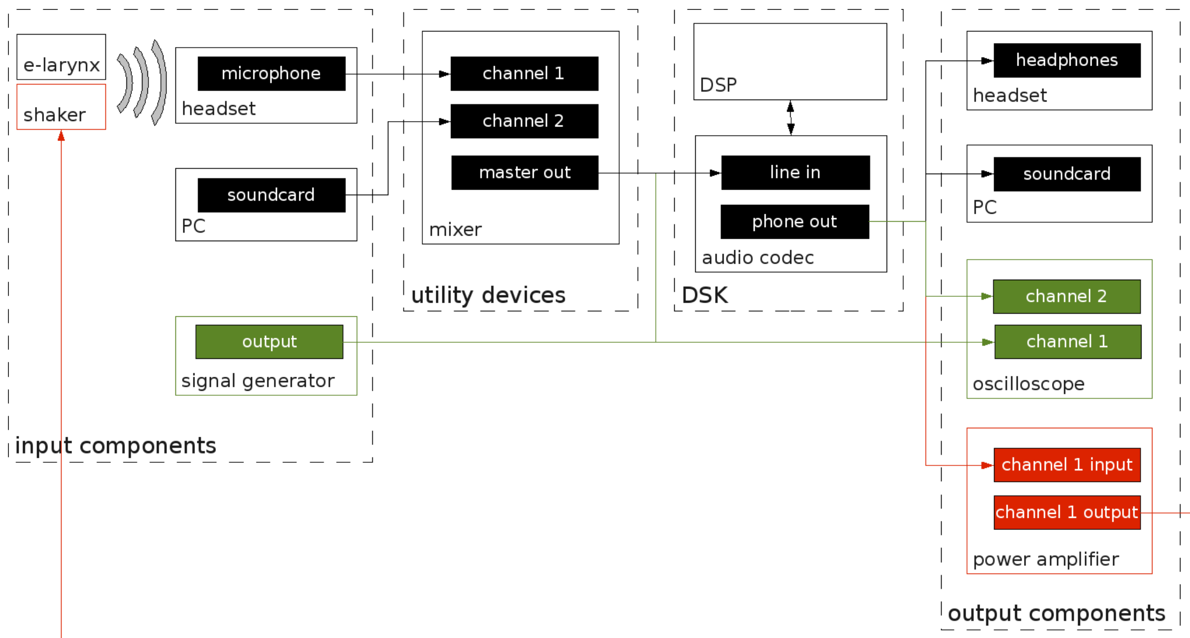


Figure 4.1: *Experimental setup of the enhancement framework: Black blocks indicate the common signal flow between electrolarynx and headphone and red the one used for controlling the shaker in driving pulse output mode. The green path shows the signal flow between signal generator and oscilloscope that was used in several debugging situations.*

The component's wiring as used during the development is shown in figure 4.1. Hereby the typical signal paths for different operation modes are shown in different colours: The black path is the default one between electrolarynx voice input and headphones used for monitoring the result. In case the pitch contour generation is switched on and operating in driving pulses output mode (indicated by the signal path displayed in red) the shaker produces the contour-equipped electrolarynx voice, the generated control pulse signal is returned back from DSP output to the shaker after being amplified in the power amplifier. Finally the green signal path pictures the flow between signal generator which feeds the framework with a dedicated input and the oscilloscope that captures the DSP altered, consequential output. This method was mainly used for debugging procedures where a well-defined input was needed to analyse the application's input-output behaviour.

4.2 Block Processing Framework

Due to the implemented speech enhancement algorithms making intense usage of block filters (**F**inite **I**mpulse **R**esponse (FIR)/**I**nfinite **I**mpulse **R**esponse (IIR), **M**odulation **S**pectral **F**ilter (MSF), ...) and block transforms (**F**ast **F**ourier **T**ransform (FFT)) which based on the principle explained in section 2.2.1, a block processing environment had to be realised. The skeletal structure is provided by [16] as discussed in section A.2.1 and provides all needed functionality: Framing with interrupt-driven **I**nput / **O**utput (I/O), frame overlapping and the possibility to overlap-add frames can be integrated in external source code very easily. The block processing framework features a high modularity and a central configuration file to allow easy setup and handling. Detailed implementation and configuration topics are discussed in [16] and can be referred to if required.

The current implementation performs the framing procedure using a block size of 256 ($32ms$ for f_s=$8kHz$). The reasons for choosing this exact value are discussed in section 4.5.3. At a glance 256 turns out to be the best trade-off between quality and operating speed. Furthermore overlapping was performed, operating with an overlapping factor of two (50% overlapping). With applying a *Hann window* (Sect. A.1.2) to the overlapping frames in combination with overlap-adding them occurring artefacts on the frame transitions are minimised. The overall processing workflow is presented in figure 4.2.

The sampling rate had to be kept to a minimum in order to provide the DSP enough time for performing the enhancement calculations in real-time. The *AIC23 codec*, as described in section 2.3.3, is capable of sampling with $8kHz$ at minimum. According to *Nyquist* this results in a maximal signal frequency of $f_{in} < \frac{f_s}{2} \Rightarrow f_{in} < 4kHz$ which is sufficient for speech processing applications. Speech signals are made up from low-frequency components compared to for example music. The pitch is located around $150Hz$, spectral components higher than $4kHz$ are certainly existent but mainly caused by noise-like sound which does not affect the overall intelligibility of the speech signal that much.

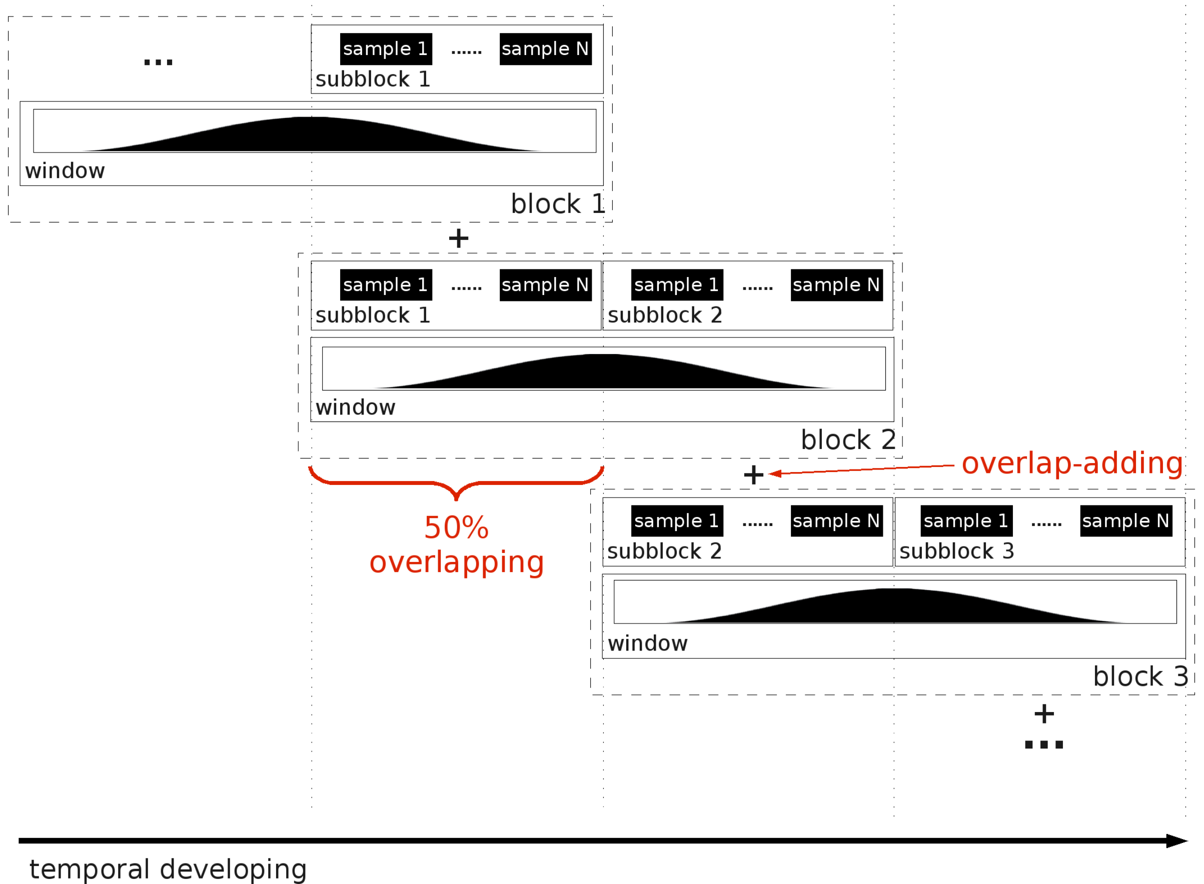


Figure 4.2: *Temporal workflow of the captured audio data as processed by the block processing framework. The input samples are concentrated to sub-blocks that are directly exchanged with the codec. Internally sub-blocks are again reorganised into overlapping blocks. The amount of sub-blocks in one block is determined by the overlapping factor, the transition between blocks is smoothed using OLA of the overlapping block parts after applying an appropriate window.*

4.3 Hardware Setup

4.3.1 Audio Codec

Accessing the codec functionality was done using the **B**oard **S**upport **L**ibrary (BSL). The library provides functionality for setting up as well as **R**ead / **W**rite (R/W) data from/to the AIC23 by implementing the codec's communication protocol and sending the needed commands. Especially this functionality is included in functions `DSK6713_AIC23_openCodec()`, `DSK6713_AIC23_closeCodec()`, `DSK6713_AIC23_config()`, `DSK6713_AIC23_setFreq()` for the setup and configuration procedure and `DSK6713_AIC23_read()` as well as `DSK6713_AIC23_write()` for data exchange purposes. The sampling data is transported in pairs with one pair including the I/O samples of left and right channel. In practice this leads to every data value being a 32*bit* sample consisting of the upper 16*bits* for the left channel and the lower 16*bits* for the right one. The initial configuration procedure includes the possibility to set up behaviour like the volume of the channels as well as the sampling frequency.

The actual data exchange may be done using either polling or an interrupt-driven approach. Due to the high demand towards operating speed the interrupt mode was chosen to be implemented - polling wastes way too much effort in periodically checking for new data. To implement the data exchange routines using interrupts, the data handling routine has to be linked to the desired interrupt. The interrupt vector table is the correct way to do so - the wanted interrupt simply includes a branch command to the data exchange routine. The alteration of the interrupt vector table is done via adapting the vector file `vectors.asm`. This file is included in every **C**ode **C**omposer **S**tudio (CCS) project and tells the development environment how to set up the interrupt table when programming the DSP. Finally the interrupt itself has to be activated and linked to the audio codec during the initialisation. The BSL functions `IRQ_enable()`, `IRQ_nmiEnable()` and `IRQ_globalEnable()` perform these operations.

```
INT8:
        b _edma_isr
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
```

The above section shows the used implementation in the interrupt vector table: *Interrupt 8* branches to the `edma_isr()` routine that includes the data exchange routines with the audio codec. With linking interrupt 8 to the AIC23 using `IRQ_enable()` the audio codec will raise the appropriate interrupt every time new data arrives.

4.3.2 Memory Management

The memory setup is configured similar to the interrupt vector table as discussed in section 4.3.1: The CCS expects a special configuration file it utilises for setting up the compiler. In case of memory management this is the linker command file. It's name can be chosen freely, the identification is done via it's `*.cmd` extension. This file contains information about size and assignment of memory sections. Important sections will be discussed in the following lines:

```
-stack  0x900
-heap   0x2F0000
```

This section informs the compiler about the size of heap and stack. The heap contains the program's dynamically allocated memory whereas the stack is used for saving return addresses and parameters when calling subroutines. The stack can be kept rather small, it usually takes high values using recursive calls but those should be avoided due to large management overhead in DSP applications anyway. The heap on the other hand has to be of a larger size because this framework makes heavy use of dynamically allocated arrays to save input and delayed frame data, results of spectral analysis, et cetera.

```
vecs:   o = 00000000h   l = 200h
IRAM:   o = 00000200h   l = 0003fe00h
CE0:    o = 80000000h   l = 01000000h
```

The above lines set up the memory regions used in the linker command file whereas the first parameter specifies the name of the region, the *o* parameter the **o**rigin (the start address) and *l* the region **l**ength. In this example `IRAM` defines the flash memory starting right after the interrupt vectors `vect` with a length of $256kB$. The start address of the **D**ynamic **R**andom **A**ccess **M**emory (DRAM) is `0x80000000`, `CE0` therefore defines a memory segment in the DRAM with length $16MB$, the complete DRAM memory.

```
.vec      >    vecs
.text     >    IRAM
.data     >    IRAM
.cinit    >    IRAM
.stack    >    IRAM
.const    >    CE0
.bss      >    CE0
.sysmem   >    CE0
.far      >    IRAM
.switch   >    IRAM
.tables   >    IRAM
.cio      >    IRAM
```

Finally the specific data segments are mapped the the memory regions defined above. This example is the final mapping used by the sound enhancement framework. It defines that all data except `.const`, `.bss` and `.sysmem` are stored in the flash memory, the mentioned sections are stored in the DRAM. For a better understanding table 2.4 explains the existing data segments and their shortcuts (a full list of sections can be found in [43]). Using this information it can be seen that the DRAM holds constants, global variables and dynamically allocated data. This is necessary due to the high memory requirement of these regions - constants are stored as floating-point values so every single value requires four bytes of memory. With the high amount of data values the overall memory requirement is too high for the flash memory. In the global area high amounts of logging data are stored (mainly floats as well), dynamically allocated data is used for most arrays and matrices in the enhancement procedure. Nevertheless the flash memory should be preferred over the DRAM, access to this module is faster.

4.4 Modules

The speech enhancement methods were implemented using a module-based structure. Every component forms it's own module that interacts with others through sharing either the same data or meta-information. In most cases this shared data is the input frame which exchanged between modules via `main()` method which is called on every occurring interrupt. Meta data like formant or spectral information is held by the module that calculated it, the transfer between modules is again done by the `main()` method with reading the calculated data from one and passing it to the other module.

The following section describes the working principle of the various modules, their interaction with each other sticks to the principles explained in chapter 3, the `main()` method's supply with frame data is based on the block processing principles as shown in section 4.2.

4.4.1 Fast Fourier Transform

There are two possibilities to perform a FFT dependent on the block/frame length which equals the FFT point size: For lengths being a multiple of two the **D**igital **S**ignal **P**rocessor **Lib**rary (DSPLib) function `DSPF_sp_cfftr2_dit()` can be used. This pre-compiled and processor optimised routine optimally loads the processor's functional pipelines what makes it the fastest and most effective method of performing a Fourier transform. The drawback when choosing this method is a limitation to input data lengths being a power of two because `DSPF_sp_cfftr2_dit()` is based on the radix-2 FFT implementation.

The `DSPF_sp_cfftr2_dit()` is called with parameters x, w and n whereas x is utilised as input and output as well. Input data is the time domain signal, output data is the resulting spectrum. The data values for both input and output have to be complex in form of the input array consisting of the real value parts stored at even array indices and the imaginary parts in the subsequent odd ones like shown in equation (4.1). The data signal captured by the audio codec is purely real, it therefore has to be expanded with imaginary parts that are set to zero. The w parameter is the twiddle factor, the complex coefficients used by the FFT and finally n describes the input data length (which is equal to the FFT point size). See equation (4.2) for the exact prototype.

$$\begin{aligned} complex &= [\ \texttt{Re(x1)},\ \texttt{Im(x1)},\ \texttt{Re(x2)},\ \texttt{Im(x2)},\ \texttt{...},\ \texttt{Re(xn)},\ \texttt{Im(xn)}\ ] \\ real &= [\ \texttt{Re(x1)},\ \texttt{0},\ \texttt{Re(x2)},\ \texttt{0},\ \texttt{...},\ \texttt{Re(xn)},\ \texttt{0}\ ] \end{aligned} \tag{4.1}$$

```
void DSPF_sp_cfftr2_dit(float* x, float* w, int n)            (4.2)
```

In the process of finding the best performing value for the block size also other, rather common block sizes like *160, 192, 320* et ectera, were utilised in the development process. To perform transforms with these block sizes an external FFT implementation had to be used. Please refer to section A.2.2 for details.

The mentioned implementation already includes the possibility to limit the spectral transform to purely real input signals what is a great benefit for the framework. The FFT's input data is provided by the audio codec, therefore the Fourier transform is always applied to purely real input data. In contrast to complex signals real ones always result in a spectrum where all spectral components satisfy the symmetry property $X_k = X^*_{N-k}$ of the **D**iscrete **F**ourier **T**ransform (DFT) as explained in [29, p.568]. This property, called *Hermitian symmetry*, explains that every spectral pin above $\frac{N}{2}$ is a complex conjugate of the mirrored pin in the lower spectrum half. This can be seen in figure 4.3: The mirrored magnitude component pairs are equal, their phases antisymmetric. This property can be used for speeding up the MSF: Just one half of the spectrum has to be processed, the other half can be mirrored.
The DSPLib does not provide a FFT function for calculating purely real input. The utilisation has to be done manually with just using one half of the gathered spectrum for further calculations and mirroring the missing half before performing the **I**nverse **F**ast **F**ourier **T**ransform (IFFT).

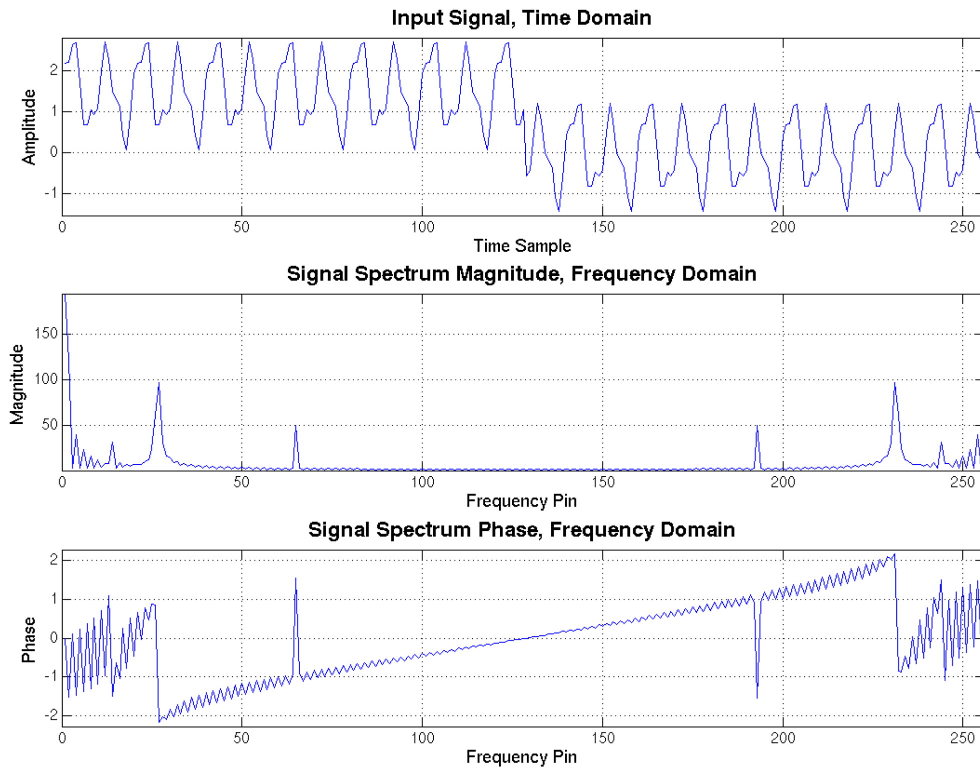


Figure 4.3: *Purely real input signal (upper picture) and it's amplitude (center picture) and phase (lower picture) response. The Hermitian symmetry of DFT transformed, purely real input signals is indicated by the mirrored frequency components' magnitudes their mirrored, antisymmetric phase.*

To accomplish even further speed enhancement the forward Fourier transform was also tested using *even/odd decomposition*. This technique, as covered in [35], can be applied to real-valued input signals and is capable of transforming a signal of length BLKSZ using a

$\frac{BLKSZ}{2}$ point transform instead of BLKSZ. The algorithm utilises the Fourier transform's Hermitian symmetry property by mapping the real-valued input to real (even elements) and "fake" imaginary (odd elements) components as shown in equation (4.3).

$$\begin{array}{lcl} \textit{real input} & = & [\ \texttt{Re(x1)},\ \texttt{Re(x2)},\ \texttt{Re(x3)},\ \texttt{Re(x4)},\ \ldots,\ \texttt{Re(xn)}\ ] \\ \textit{"complex" input} & = & [\ \texttt{Re(x1)},\ \texttt{Im(x2)},\ \texttt{Re(x3)},\ \texttt{Im(x4)},\ \ldots,\ \texttt{Im(xn)}\ ] \end{array} \tag{4.3}$$

The accordingly arranged input array is then converted using the complex FFT and post-processed to recombine the transforms of even and odd components into a single result for the actual input signal. The listing below explains the rough processing steps using the algorithm from [35].

```
01.   //1.  pre-processing
02.   re-arrange input array x[] to x_{e/o}[]
03.   //2.  Fourier transform
04.   X_{e/o} = F{x_{e/o}[n]} with resolution BLKSZ/2
05.   //3.  post-processing
06.   parse X_{even} = F{x_e[]} from X_{e/o}
07.   parse X_{odd} = F{x_o[]} from X_{e/o}
08.   combine X_{even} and X_{odd} to X = F{x[]}
```

Nevertheless the tested implementation performed even worse than the method using nulled imaginary components in terms of calculational complexity. Without using even/odd decomposition the forward FFT could be executed in $436k$ CPU cycles compared to $494k$ when using it. This numbers result in a loss of complexity equal to 11%. Several studies, including [35], name a gain of about 40% in average - the high discrepancy occuring here is caused by the inefficient post-processing that additionally has to be performed when using even/odd decomposition: The calculational effort of manual post-processing (no assembly-level optimisation) is much higher than the last butterfly stage in a BLKSZ point FFT (optimised if using the DSPLib version) that would be spared otherwise.

The IFFT is performed more or less similar to the forward Fourier transform by using the DSPLib quoted in equation (4.4). Parameters are equal to the forward FFT except of the input being the frequency-domain spectrum instead of time-domain signal. Internally the IFFT implements a decimation-in-frequency algorithm (hinted by the function name's ending) instead of the decimation-in-time the forward FFT uses. Nevertheless this does not affect the way of calling the function or preparing the input data. For more information about FFT algorithms please refer to [29, p.635].

$$\texttt{void DSPF_sp_icfftr2_dif(float* X, float* w, int n)} \tag{4.4}$$

4.4.2 Finite Impulse Response Filter

FIR filters are digital filters with a finite response length - it is zero outside the operating length of the filter. This is explained by the missing feedback compared to IIR filters. The response of a FIR filter is calculated with equation (4.6) where $x[t]$ stands for the input sample at time slot t, $y[t]$ for the appropriate output sample and c_i is the ith coefficient, the amount of coefficients c_N determines the filter size N. This equation, formally a convolution of signal $x[t]$ and FIR coefficients $c[i]$, can also be expressed in summation notation as shown in equation (4.7). The convolution functionality is provided by the DSPLib function `DSPF_sp_fir_gen()` and should again be preferred over self-coded implementations due to it's optimisation for the DSP's **C**entral **P**rocessing **U**nit (CPU).
`DSPF_sp_fir_gen()` (see equation (4.5) and figure 4.4) is called with parameters x, h, y, n_h and n_y. Parameter x is the input signal, y the output and h the coefficient array pointer. n_h and n_y specify the parameter lengths with n_h indicating the coefficient array length and n_y the output data length. The input data length has to be of length $n_h + n_y - 1$ caused by the internal convolution.

```
void DSPF_sp_fir_gen(float* x, float* h, float* y, int nh, int ny)     (4.5)
```

$$y[t] = a_0 * x[t] + a_1 * x[t-1] + \cdots + a_N * x[t-N] \tag{4.6}$$

$$y[t] = \sum_{i=0}^{N} a_i * x[t-i] \tag{4.7}$$

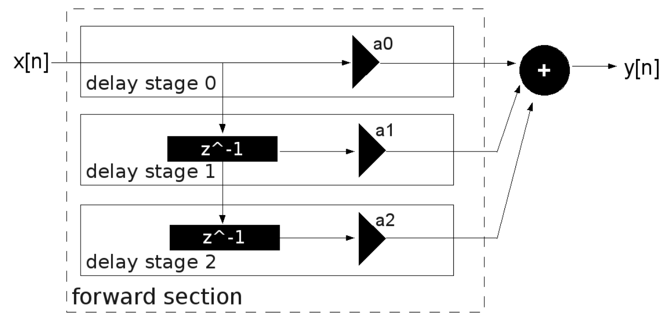


Figure 4.4: *Processing blocks of a - in this case - 2rd order FIR filter with z^{-1} being the 1st order time delay elements and a_x the coefficients of the gain blocks.*

The primary purpose of FIR filters in this framework is to perform pre-emphasis calculations and inverse filtering the microphone's frequency response. The pre-emphasis is a $1 - z^{-1}$ filter and therefore a FIR filter with $N = 2$ and $c_0 = 1$, $c_1 = -1$. It is used in the **I**nverse **F**ilter **C**ontrol (IFC)-based formant detection approach (Sect. 4.4.10) as part of the signal pre-processing. The inverse filtering of the microphone's frequency response is a countermeasure against the distinctive near-field effect of the used microphone. Figure A.1 shows the frequency response of the used headset microphone. The *10dB* increase of low frequency signal components between $0Hz$ and about $300Hz$ at small distances to the microphone ($30mm$ in

the figure) can be clearly seen. To linearise the frequency response a inverse filter was implemented as part of the overall pre-processing routine. It targets on cancelling this gain using a FIR filter designed in Matlab. The filter design is discussed in section A.1.1.
Additionally an implementation of the MSF was done using FIR filters instead of IIR's. It's purpose was to compare the filter results dependent on the used filter type. Section 5.2.3 analyses the gathered results.

4.4.3 Infinite Impulse Response Filter

The implementation of IIR filters is based on the *biquad* structure described in [29, p.356]. Biquads are 2nd order IIR filters, with chaining them arbitrary filter orders can be reached. A typical biquad block, as shown in figure 4.5, is described by it's five parameters a_1, a_2, b_0, b_1 and b_2 with the a components being the feedback gain or denominator (these components are located in the filter impulse responses denominator) and the b components being the forward gain or numerator respectively. The IIR behaviour is caused by the internal feedback which is the main difference compared to FIR filters. The feedback behaviour is determined by the a parameters as seen in figure 4.5 as well as in equation (4.8).

$$y[t] = b_0 * x[t] + b_1 * x[t-1] + b_2 * x[t-2] - a_1 * y[t-1] - a_2 * y[t-2] \tag{4.8}$$

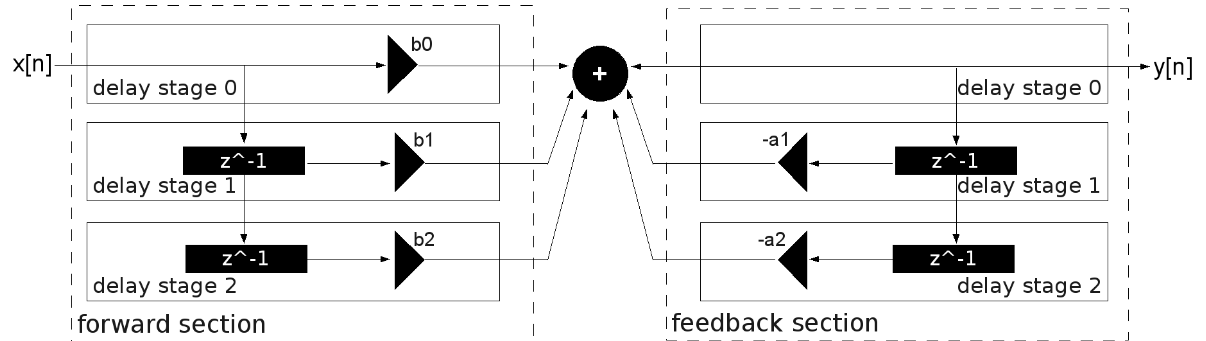


Figure 4.5: *Structure of a biquad IIR filter in DF1 with forward (b_x) and feedback (a_x) coefficients as well as 1st order time delay elements z^{-1}.*

Knowing these coefficients IIR filtering is performed using the `biquad()` function provided by the DSPLib framework. As shown in equation (4.9) It is provided with parameters x and n being the input data sample and biquad order, c are the filter coefficients b and a in a specially aligned form as shown in table A.1. Worth mentioning is that parameter b_0 is assumed to be 1.0. Eventually the other b parameters have to be scaled to fulfill this requirement. Return value of the function is the current output sample $y[t]$. Finally the z parameter contains the current internal delay state of the filter structure. To apply the filter to a complete frame, the method has to be called in a loop.

```
float biquad(int n, float* c, float* z, float* x)        (4.9)
```

The parameter calculation is discussed in section A.1.1.

4.4.4 Windowing

The windowing functionality was a straight-forward implementation: Dependent on the length of the input frame the window coefficients are calculated as shown in section A.1.2. Every frame sample is then multiplied with the appropriate coefficient. Worth mentioning here is that the coefficients for frequently used frames and therefore window sizes are pre-calculated to save calculation time.

4.4.5 Linear Predictive Coding

The **L**inear **P**redictive **C**oding (LPC) algorithm used for the implementation is based on three fundamental steps:

1. Build the autocorrelation matrix of the system: To get the entries for the matrix an autocorrelation is performed for every delayed signal sample in order to determine the coefficients $a_{ss}[i]$ of the matrix. Index i stands for the delay of the autocorrelation value. The resulting matrix possesses a Toeplitz structure which looks like shown in table (4.1) for a size of N with N being the LPC order. A Toeplitz matrix therefore has equal diagonal elements, in this case it is symmetric as well.

2. Find the system solution: To get the wanted coefficients the system, described by a Toeplitz matrix, has to be solved. This is done using the *Levinson-Durbin algorithm*, an iterative approach to solve a system being described by a Toeplitz-form matrix. The necessary steps are shown in equation (4.10) with A_{ss} being the Toeplitz matrix elements. The algorithm is a slightly altered version of [23].

3. Store the system coefficients: The wanted coefficients can be obtained after performing the Levinson-Durbin algorithm, they are contained in the α matrix as row vector at position $N-1$, thus $\alpha[*][N-1]$.

$$\begin{bmatrix} a_{ss}[0] & a_{ss}[1] & a_{ss}[2] & \dots & a_{ss}[N-1] \\ a_{ss}[1] & a_{ss}[0] & a_{ss}[1] & \dots & a_{ss}[N-2] \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ a_{ss}[N-1] & a_{ss}[N-2] & \dots & a_{ss}[1] & a_{ss}[0] \end{bmatrix}$$

Table 4.1: *Matrix in Toeplitz structure: The elements in the diagonals are equal and mirrored along the main diagonal.*

$$\begin{array}{llll} \mathbf{0}. & \alpha_{x,y} & = 0 & \dots \quad 0 < x < N-1 \ , \ 0 < y < N-1 \\ & k_x & = 0 & \dots \quad 0 < x < N-1 \\ & e_x & = 0 & \dots \quad 1 < x < N-1 \\ & e_0 & = A_{ss}[0] & \\ & & & \\ \multicolumn{4}{l}{\texttt{for i = 0 to N-1}} \\ \mathbf{1}. & k_i & = \frac{1}{e_i} * (A_{ss}[i+1] - \sum_{i=0}^{i-1} \alpha_{j,i-1} * A_{ss}[|i-j|]) & \\ \mathbf{2}. & \alpha_{j,i} & = \alpha_{j,i-1} - k_i * \alpha_{i-1-j,i-1} & \\ & \alpha_{i,i} & = k_i & \\ \mathbf{3}. & e_{i+1} & = e_i * (1-k_i^2) & \end{array} \tag{4.10}$$

4.4.6 Voice Activity Detection

The decision whether a frame includes speech data or just background noise is made using an energy threshold. If the overall frame data's energy E_{abs} or E_{db} if given in dB exceeds a certain threshold it is considered to be `ACTIVE`. This approach has the advantage of performing very well despite of being computational cheap. The frame energy is calculated using equation (4.11): It simply consists of N =BLKSZ additions for calculating the sum and N square operations for getting the data sample's power $x[i]^2$. The maximal possible energy E_{max}, occurring if all data samples would have the maximal value possible x_{max}, just has to be pre-calculated once in the initialisation procedure. The value a data sample can have at maximum is determined by the audio codec's amplitude resolution. The BSL function `DSK6713_AIC23_read()` converts samples to pairs of $16bit$ values so x_{max} results in $\pm 2^{16} = \pm 32767$.

$$\begin{aligned} E_{max} &= \sum_{i=0}^{N-1} x_{max}^2 = \sum_{i=0}^{N-1} 32767^2 = N * 32767^2 \\ E_{abs} &= \sum_{i=0}^{N-1} x[i]^2 \\ E_{db} &= 10 * log_{10}(\tfrac{E_{abs}}{E_{max}}) \end{aligned} \tag{4.11}$$

4.4.7 Voiced/Unvoiced Detection

The principle the **V**oiced / **Unv**oiced (V/UV) detector is based on is the different spectral behaviour of voiced and unvoiced sounds in the high-frequency region. In voiced speech segments the energy majority is located within the fundamental frequency and first few harmonics because of the high periodicity of voiced phones. On the other hand unvoiced phones can be considered to be noise from a signal processing point of view. There is no dominant periodic part to claim the energy majority for it's spectral component. The energy fragmentation is more consistent in this case, the relation between energy of high- and low-frequency spectral components is more constant therefore.
This property can be utilised using a threshold decision for the spectrum's energy centroid. Due to the higher energy of spectral pins in the upper part of the spectrum the centroid is shifted towards higher frequencies in unvoiced frames. This is why frequency thresholding of the calculated centroid can be used for the V/UV decision: Voiced frames lead to centroids below the frequency threshold, unvoiced frames vice versa. See equation (4.12) for an algebraic description.

$$f_{centroid} = \frac{1}{N} * \sum_{i=1}^{N} i * X[i]^2 \Rightarrow \begin{cases} V & f_{centroid} \leq f_{th} \\ UV & f_{centroid} > f_{th} \end{cases} \tag{4.12}$$

This algorithm was also investigated by [28].

4.4.8 Pitch Tracking

To perform pitch tracking two fundamental different approaches were implemented. The goal was to determine both algorithms' performance in terms of speed and accuracy for selecting the overall better performing one.

Autocorrelation Method

The pitch extraction procedure is an implementation of the proposal by [8]. This algorithm is also used in Praat ([31]) and therefore already tested in it's robustness and accuracy. In it's basics this approach already exists for over 40 years. It uses a detection of local peaks in the signal's autocorrelation to determine the periodicity. The distance between two autocorrelation peaks is identified to calculate the period of the analysed signal.
The improvement [8] proposes targets on including the framing effects of short-time processed signals in the detection procedure. This is done via dividing the autocorrelation result $r_a(\tau)$ of the frame under investigation by the autocorrelation of the used window $r_w(\tau)$. This procedure reverses the multiplication of the input signal $x(t)$ with the window $w(t)$ in the time domain to make $r_a(\tau)$ window-independent in the autocorrelation domain. An example is given in figure 4.6 showing the different results of the autocorrelation procedure with ($r_x(\tau)$) and without ($a(t)$) cancelling the window effects. It can be seen that the enhanced result $r_x(\tau)$ fits the true autocorrelation result of $x(t)$ way better than $r_a(\tau)$. The algorithm also introduces a method for smoothing the values of sequential results, this calculation is referred to as *global pathfinder*. It works with applying additional cost to octave and V/UV jumps and finding an optimal path throughout the pitch contour.

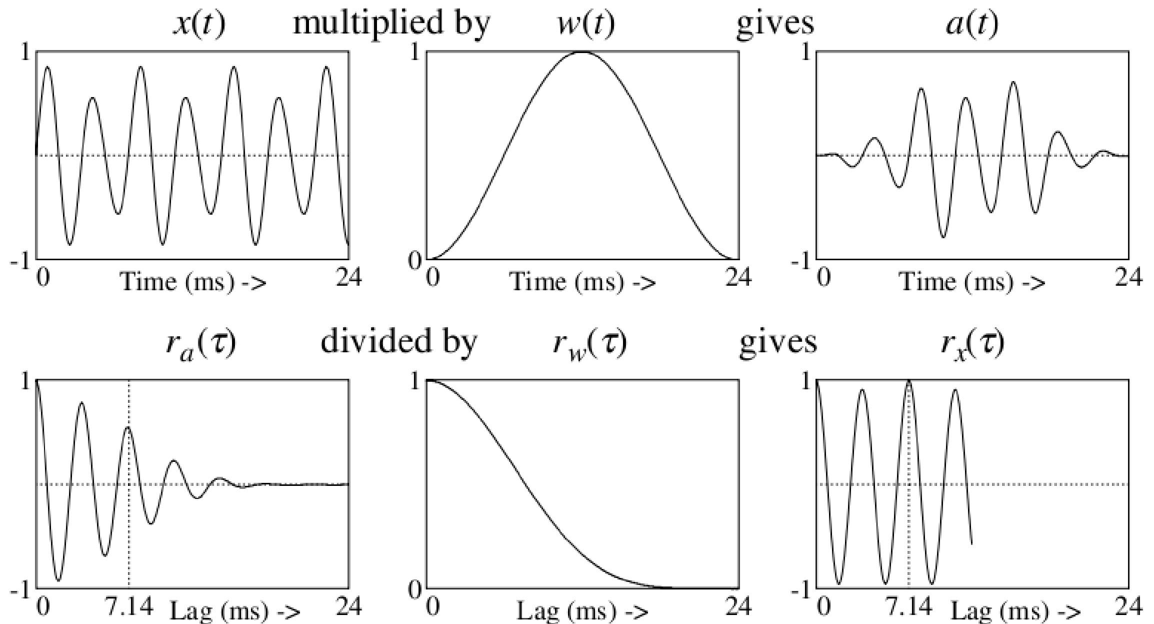


Figure 4.6: *The autocorrelation-based pitch detector's window effect cancellation with the upper pictures being the input signal $x(t)$, window function $w(t)$ and windowed input $a(t)$. The lower pictures represent the appropriate autocorrelations with $r_a(\tau)$ being the original and $r_x(\tau)$ the window-effect cancelled signals, $r_w(\tau)$ is the autocorrelation of the window function (source: [8]).*

The actual implementation is based on the calculation steps proposed by [8] including some adaptions: Instead of calculating the autocorrelation using a FFT and squaring the spectral components the autocorrelation procedure is performed directly. This approach advances the original one in terms of calculational speed because of being able to use the speed-optimised

function provided by the DSPLib instead of choosing the detour via the Fourier transform. Furthermore, the recommended pre-processing steps are performed already by several pre-processing routines of other analysis modules before even arriving at the pitch detector.

Zero-Crossing Method

As alternative to the autocorrelation method the - faster but less accurate - zero-crossing method was integrated in the enhancement framework. The choice of this approach is reasoned by the actual usage of this application: The signal processing is performed on electrolarynx speech signals. Those signals produce an as good as constant pitch anyway so detection accuracy is secondary. This lead to designing the pitch detector with focus on operating speed for saving calculation time rather than accuracy. The formant tracker benefits from the additional time and it's results exhibit a significantly higher effect on the enhancement result.

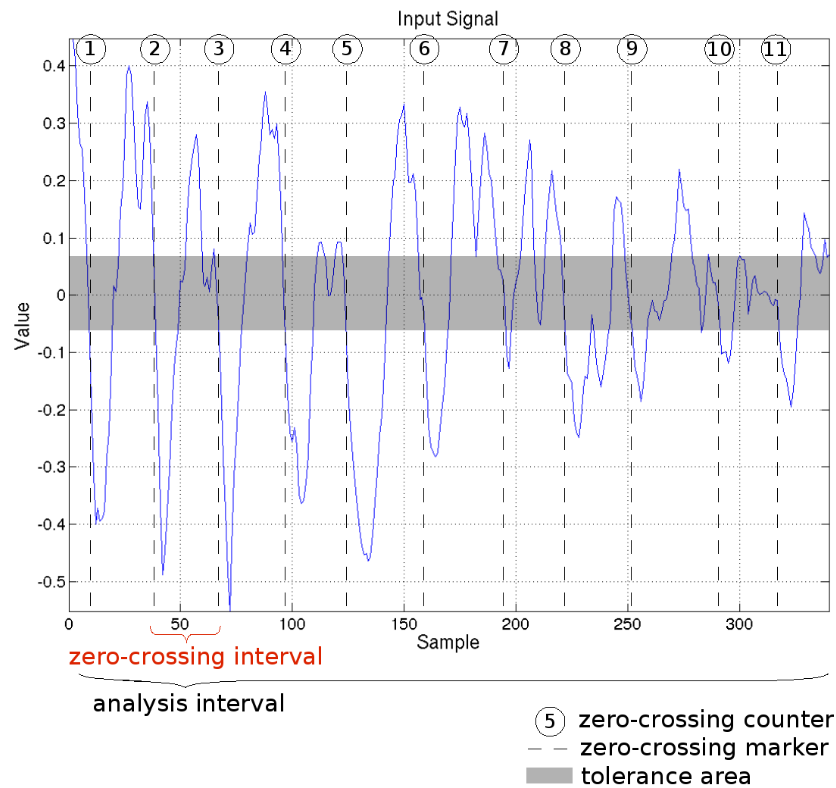


Figure 4.7: *Working principle of the zero-crossing pitch detector: The pitch is calculated by counting zero-crossings in a certain analysis interval and converting their average spacing to a frequency value. The tolerance interval minimises miscalculations caused by noise - internal zero-crossing fluctuations in this area are not counted.*

The operating principle of the zero-crossing pitch detector, as illustrated in figure 4.7, works with counting the amount of sign changes in a given time interval. When applying this method to a mean-free signal the average distance between zero-crossings corresponds to the signal's pitch. In order to reduce the effects of noise on the gathered result the input is pre-processed using a low-pass filter. Additionally a tolerance threshold (of 0.05 for normalised input) is considered when counting the sign changes. These techniques avoid the effect of counting

small fluctuations around the zero value caused by noise. Using equation (2.4) and assuming a sampling rate $f_s = 8kHz$ the signal's pitch is calculated to $F_0 = \frac{8000}{340} * 10 = 235.3Hz$. The actually implemented algorithm, using $f_s = 8kHz$ and BLKSZ= 256 leads to an approximate amount of $\lfloor \frac{100}{8000} * 256 \rfloor = 3$ zero-crossings per frame considering an expected pitch F_0 of approximately $100Hz$. Considering this data the used configuration seems to be a good choice in terms of detection accuracy.

4.4.9 Pitch-Marking

The pitch-marking procedure is implemented using an additional array that holds the positions of the marks in the current frame. The pitch-mark positions itself are calculated using the sampling rate and resulting pitch value to calculate the amount of samples n_{per} that one signal period lasts. After every n_{per} a marker is placed for as long as the calculated pitch value is valid (which is the current frame). The period in samples is calculated with equations (4.13) and (4.14).
The data structure holding the pitch-mark positions is a - one-dimensional - array. In the process of experimenting with gaining run-time reserve two different designs for this array were compared:

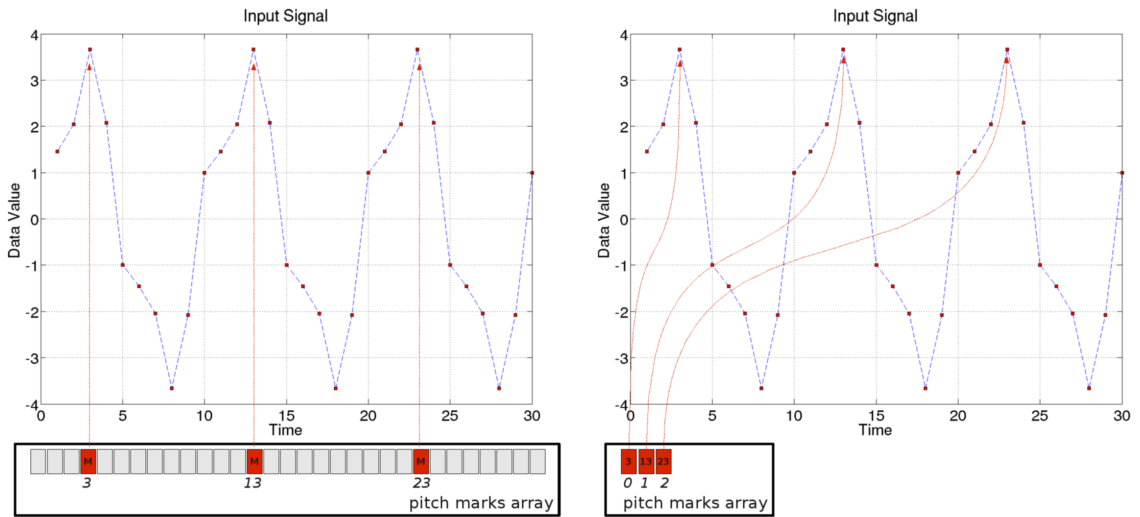
`FULL`: This implementation uses an array with a length equal to **`Block Size`** `(BLKSZ)` which holds a `MARK` value at every pitch-mark position, other ones are set to `NOMARK`. This design comes with the disadvantage of high memory usage and the need to traverse the array in order to find neighbouring marks. On the other hand it is easier to access the pitch-mark position in the array when just knowing the position in the frame because they are equal (Fig. 4.8a).

`COMP`: Different to `FULL` mode `COMP` uses an array with just consisting of as much elements as there are pitch-marks in the current frame. Every array entry holds the pitch-mark position as it's value. This method saves memory and neighbouring marks are found easily with increasing/decreasing the array index by one. Nevertheless it lacks in accessing pitch-mark entries when just knowing their position in the frame - the array has to be traversed to find the appropriate array entry (Fig. 4.8b).

$$n_{per} = \lfloor \frac{\texttt{BLKSZ}}{\texttt{PITCHMARKSSZ}} \rfloor = \lfloor \frac{f_s}{f_0} \rfloor \tag{4.13}$$

The actual length of the pitch-marks array in `COMP` mode is dependent on the used sampling rate and block size. Using equation (4.14) leads to the example results listed in table 4.2. When taking a closer look at these values it can be observed that in some setups the list size even falls below one. This is the case for high sampling rates: With increasing Sampling Rate (f_s) the temporal duration represented by one sample decreases, one pitch period has to include more samples. If BLKSZ remains fixed at a certain value the amount of samples belonging to one pitch period might exceed BLKSZ and therefore the current block. In such situations a pitch-mark is just include in every few blocks. For example when using `BLKSZ`$= 128$ and $f_s = 16kHz$ equation (4.14) results in an exact value 0.8 for the pitch-marks list size `PITCHMARKSSZ` - an additional pitch-mark occurs in every fourth frame.

$$\texttt{PITCHMARKSSZ} = \lfloor \frac{T_{blk}}{T_0} \rfloor = \lfloor \frac{\frac{\texttt{BLKSZ}}{f_s}}{\frac{1}{f_0}} \rfloor = \lfloor \texttt{BLKSZ} * \frac{f_0}{f_s} \rfloor \tag{4.14}$$



(a) Pitch-marking mode `FULL` featuring an array of length BLKSZ that holds `MARK` and `NOMARK` entries at the appropriate positions.

(b) Pitch-marking mode `COMP` featuring an array of adaptive length that only holds `MARK` entries, their positions in the corresponding frame are given by the entries values.

Figure 4.8: *Implemented data structures used internally in the pitch-marking procedure.*

BLKSZ	f_s [Hz]	PITCHMARKSSZ
128		1
256	8k	3
512		4
128		0
256	16k	1
512		3

Table 4.2: *Example results for the pitch-marks list size in `COMP` mode with varying BLKSZ and f_s. The space between pitch-marks is determined by the signal's pitch, $100Hz$ in these examples.*

4.4.10 Formant Tracking

The formant tracker is the most important module in the pitch contour generation process. The final contour is a direct derivative of the formant data provided by this module, therefore a detection as accurate as possible is crucial for the overall quality. Unfortunately the formant tracker is also the most time-intense module in the framework caused by the polynomial root calculator and LPC analysis that take part in the formant detection process. Despite of this issue with wisely setting the tracker's configuration parameters (especially the LPC order because it directly affects the LPC complexity and amount of input data for the polynomial root detector as well, see section 4.5.3) the tracker produces satisfying results within an acceptable duration.
Nevertheless it was also experimented with an alternative formant detector approach based on inverse filtering. This method, introduced by [45], was specially designed to meet up with the expectations towards complexity and speed that arise when performing tracking in DSP

applications. However the tested implementation exceeded the available temporal window the formant detector was allowed to last to finish it's calculations in nearly every calculation cycle. The paper's authors also had to face this problem but solved it by optimising the code using different approaches. The most effective one - manually rearranging the code on assembly level - could not be taken into consideration for this implementation. The effort was not in proportion to the expense: Assembly-level optimisation is a complex and long-lasting procedure and the LPC-based tracker performed well anyway.

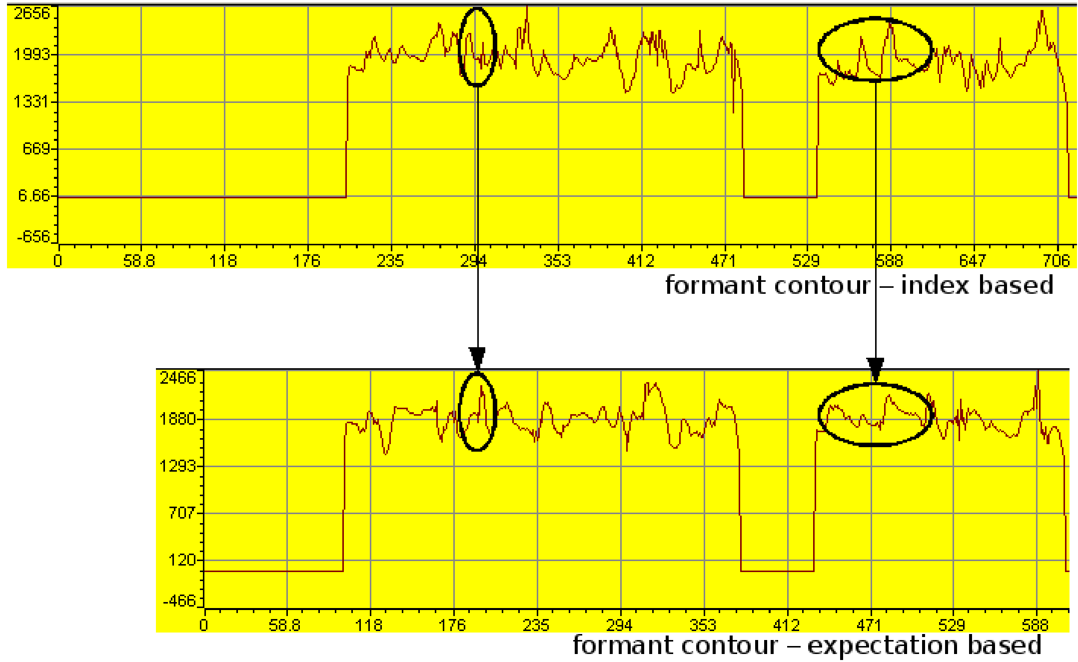


Figure 4.9: *Comparison of* index- *and* expectation-based *formant tracking: Index-based formant selection (based on the entries position in the sorted candidates array) results in accurate contours with possible outliners, expectation-based selection (based on the absolute difference between default and detected formant value) prevents outliners but also smoothens the contour unnecessarily.*

A question arising for both algorithms was the choice of the final formant value from the candidates list. This list contains those $n_{candidates}$ frequencies with the highest gain in the transfer function. In theory those are the $n_{candidates}$ lowest formants because the gain decreases with increasing formant number (Fig. 2.2). So setting the candidates list size $n_{candidates}$ to the highest wanted formant number plus one is a good choice, for example $n_{candidates} = 3 + 1 = 4$ to detect formants up to **F**ormant **3** (F_3). With very accurate formant trackers it may even be sufficient to set the list size to the exact wanted formant number but in most applications it is better to add some tolerance. When sorting the candidates array by frequency the entries positions correspond to the formant number, index $i = 0$ contains **F**ormant **1** (F_1), $i = 1$ **F**ormant **2** (F_2), et cetera. Picking the formant based on the index number turns out to be an obvious yet effective approach. Problems might occur if the formant tracker fails to detect one formant, this shifts all formants following the missing one position left - the index approach fails in this case. The next approach is based on the expected frequency value. Roughly formants are located near the appropriate kHz value, thus $F_1 \approx 1kHz$, $F_2 \approx 2kHz$, ... Traversing the

candidates array and finding the formant with the minimum distance to the expected frequency yields the candidate with the highest probability of being the expected one. Unfortunately this approach also turns out to not be optimal. The formant frequencies are highly fluctuating. For example the average F_2 frequency of vowel a is according to [49] located at about $1400Hz$. The above algorithm would treat this formant of being F_1 if F_1 itself is farther away from $1kHz$ than $1400Hz - 1000Hz = 400Hz$. The most accurate approach is implementing a global pathfinder algorithm. However for online processing this can be problematic due to increasing the overall system's input-output delay. Refer to section 5.1.3 for detailed information.
After testing both, the index-based and expectation-based approach, it was decided to pick the index-based method. It might lead to rare outliners but their occurrence can be decreased by increasing the formant tracker's accuracy. The expectation-based method produces very smooth contours what works against the pitch contour generator's principle of creating a distinct and audible pitch contour - see figure 4.9 for a contour example of both approaches.

Linear Predictive Coding Based

As mentioned in sections 2.1.3 and 2.2.6 the formant frequencies are the local maxima of the vocal tract transfer function and can therefore be determined with performing a **L**inear **P**redictive **C**oding (LPC) analysis of the input signal. The LPC approximates a signal's transfer function with a polynomial of given order thus accuracy. This operation helps in finding the the local maxima of the transfer function (it's poles) through the approximation polynomial: The polynomial's roots are equal to the transfer function's poles. So performing a polynomial root analysis of the LPC polynomial leads to determining the high-gain sections of the transfer function - the closer a polynomial root is to the unit circle the higher is the gain linked to it. So the combination of LPC analysis and subsequent polynomial root detection can be used to determine these local maxima. At a glance the necessary processing steps are:

1. Calculate the LPC polynomial that approximates the transfer function of the input frame
2. Calculate the polynomial roots of the LPC polynomial to determine it's maxima
3. Calculate the formant frequencies using the LPC polynomial's maxima positions and used sampling rate

The functionality of LPC analysis and polynomial root detector are already treated in the appropriate sections (4.4.5 and A.2.3). Formant calculation simply consists of combining and interpreting these results. The processing chain consists of at first providing LPC module with the frame data. Secondly the calculated LPC coefficients that form the polynomial are directly inputted to the polynomial roots detector. The output the polynomial roots detector produces are the polynomial's complex root values that correspond with the formant frequencies. Those are in a final step calculated using the formula given in equation (4.15) with f_s being the sampling rate, φ_x the angle of the complex polynomial root number x and F_x being the formant belonging to this polynomial root.

$$F_x = \frac{f_s}{2 * \pi} * \varphi_x \tag{4.15}$$

The LPC order determines the accuracy of the formant tracker. An LPC analysis of order p results in an approximation polynomial of the same order. Polynomials of order p also consist of p polynomial roots (with some roots being complex conjugate pairs) what in sum

leads to p candidates for a formant. A high number of formant candidates finally minimises the chance of missing the formant to be determined. Section 4.5.3 deals with this topic in detail.

The choice whether a candidate is a formant or not is answered with cross-checking the formant data with the facts listed below.

- The formant frequency has to be higher than the fundamental frequency
- The pole's absolute value must exceed a certain threshold (the higher the pole's absolute value the higher the transfer function's gain)
- For complex results (which are all complex conjugate pairs) just the value in quadrant one or two has to be taken into consideration

Inverse Filter Control Based

The IFC based approach, as introduced in section 2.1.3, uses a completely different approach to perform formant tracking. In principle it utilises the mutual suppression of spectral components surrounding one formant frequency to perform a FFT-peak based formant calculation on the other formant. This procedure is recursively executed until convergence using neighbouring pairs of formants. Signal parts outside the interesting frequency interval are filtered in a pre-processing stage, parts being located around another formant (none of the two ones currently under investigation) in the interval are temporary suppressed with using the mentioned **I**nverse **F**ilter (IF).

Implementing this algorithm can be concentrated to three main processing blocks:

1. **Overall pre-processing:** This step performs the filtering of unwanted spectral components (below the lowest allowed formant frequency and above the highest one) and a pre-emphasis for amplifying higher frequencies.

2. **Sub-block level pre-processing:** Formants are determined by pair-wise mutual convergence approaching. To do so pairs of neighbouring formants have to be chosen and other ones suppressed before starting the current calculation procedure.

3. **Formant approaching:** The two formants currently under investigation are approached with mutually applying an inverse filter to one and calculating the other formant.

Calculation steps **2.** and **3.** are performed in a loop to reach convergence. The mutual calculation is optimally repeated three times for each formant pair, the calculation process for all formant pairs again has to be repeated three times as well. This leads to two nested loops with each repeating three times - one cycle of the outer loop calls the inner loop with all possible formant pairs combinations, one cycle of the inner loop performs one approaching for each of the two formants. For a better understanding the processing chain is explained using simplified pseudo-code:

```
01.   //1.  overall pre-processing
02.   perform pre-emphasis
03.   perform bandpass-filtering
04.   //2.  sub-block level pre-processing
05.   FOR i = 1 TO 3
06.       pick formant pair
07.       perform inverse filter on other formants
08.       //3.  formant approaching
09.       FOR i = 1 TO 3
10.         perform inverse filter around formant X
11.         calculate formant Y
12.         perform inverse filter around formant Y
13.         calculate formant X
14.       ENDFOR
15.   ENDFOR
```

The formant calculation itself (lines *11* and *13*) is done using spectral peak detection. Therefore a FFT has to be applied to the prepared signal $((2_{mutual} * 3)_{innerloop} * 3)_{outerloop} = 18$ times. This huge effort is impossible to be performed in time. [45] proposes to use a rough spectrum based approximation just for the first pass of the loop, combined with a more accurate weighted zero-detection for later loop cycles. In doing so the strongest spectral component of the current signal can be determined. This component is the wanted formant frequency - all other formants are suppressed by inverse filters, the fundamental frequency was already filtered in the pre-processing.

The used inverse filters are band-stop filters with a rather high bandwidth in the stop-band. This leads to their impulse response being more flat and suppressing spectra along a larger frequency interval around the center frequency. A mathematical description of the impulse response is given in equation (4.16) with $H(z)$ being the impulse response in the z domain, parameters are α, β and γ. The center frequency $f_c = \frac{w_c}{2*\pi}$ is converging to the appropriate formant frequency F_x, B_x is the bandwidth linked with the formant under investigation. [45] proposes the bandwidth values given in table 4.3 for a good performance.

$$\begin{array}{ll} H(z) & = \gamma * (1 + \beta * z^{-1} + \alpha * z^{-2}) \\ \alpha & = e^{2*\pi*B_x} \\ \beta & = 2 * e^{\pi*B_x} * cos(w_c) \\ \gamma & = \frac{1}{1+\alpha+\beta} \end{array} \tag{4.16}$$

Formant Number	Proposed Bandwidth [Hz]
1	50
2	70
3	90
4	130

Table 4.3: *Optimal IF filter bandwidths B_x as to be used in the IFC-based formant tracker. The bandwidth B_x is dependent on formant F_x that is suppressed with this filter.*

Implementation Details The configuration settings and changes to the original suggestions are targeting purely on minimising the calculation speed issues. So the highest formant to be calculated is set to three, this reduces the number of possible formant pairs from four (`[1,2] ; [2,3] ; [3,4] ; [4,5]`) to two (`[1,2] ; [2,3]`). Nevertheless the algorithm is implemented to let the developer choose the highest formant freely between two and four. Furthermore the weighting algorithm used by the zero-crossing detector is skipped completely. This might lead to a higher formant tracker accuracy but the detector is accurate enough without this routine anyway. An implementation would just cause additional calculational overhead without bringing an additional increase in performance.

4.4.11 Formant Smoothing

The formant smoothing process is needed to create a consistent formant contour. Without smoothing the formant values are highly dynamic and do not yield an evolution which can be processed furthermore. The smoothing procedure itself consists of three stages with every one holding a certain algorithm that processes the data results from the previous stage:

1. **Outliner cancellation:** Analyses neighbouring values and replaces highly differing components by a linear interpolation
2. **Median filter:** Replaces the component to be smoothed with the median of those values surrounding it
3. **Linear smoothing filter:** Replaces the component to be smoothed with the linear average of those values surrounding it

Outliner Cancellation

This type of smoothing algorithm traverses the formant data array with always comparing the current value with it's successor. If both data values differ by more than a certain threshold, a linear interpolation is done between the current element's left and right neighbour, the center value is set to the interpolation result. For the current implementation a threshold of 500 was chosen, thus formant frequency jumps above $500Hz$ are cancelled.
Multiple bordering outlines are concentrated - the interpolation procedure is done for a longer interval in such cases, see table 4.4 for an example.

Median Filter

Median filters are common in signal processing applications. This realisation does not differ from those examples in it's basics: The data values list is traversed and every value in the list is replaced by the median of the samples surrounding it. The amount of samples to be considered is determined by the *filter size* which is focused on in detail in sections 4.5.3 and 5.2.2. Table 4.4 holds an example for this filter type as well.

One particularity about this filter are the implemented operating modes. To deal with bordering effects (at the frame edges the smoothing values sink due to filling the missing array entries with zeros) three operating modes were tested:

Null mode: Missing data samples are replaced by zeros (default implementation)

Skip mode: Missing data samples lead to an adaptive shrinking of the filter size

Expand mode: Missing data samples are replaced by the last available value at the array edge

Linear Smoothing Filter

The linear smoothing filter works the same way the median filter does. Only difference is the way the new value is being calculated: Instead of the median a linear average calculation is performed. Again refer to table 4.4 for a simple example using the parameters $th_{outliner} = 300$, $len_{medfilt} = len_{linsmthfilt} = 3$.

Algorithm	Data Values [Hz]						
Original	1500	1670	1800	1300	900	950	950
Outliner Cancellation	1500	1670	1800	1516.7	1233.3	950	950
Median (Skip Mode)	1670	1670	1670	1516.7	1233.3	950	950
Linear Smoothing (Expand Mode)	1670	1670	1618.9	1473.3	1233.3	1044.4	950

Table 4.4: *Smoothing procedure showing the different algorithm's working principles by applying them to an example data array. The red numbers indicate changes to the previous processing stage.*

4.4.12 Pitch-Synchronous Overlap-Add

The **P**itch-**S**ynchronous **O**ver**l**ap-**A**dd (PSOLA) algorithm must be provided with three input parameters:

- The input frame
- The pitch-marks array holding the original pitch-marks
- The pitch-marks array holding the new pitch-marks

Being called with these parameters there is enough information available to generate the new signal according to the pitch contour provided by the pitch-mark arrays. In principle the PSOLA algorithm traverse the new pitch-marks array and places a windowed version of the appropriate input frame region on this marker using an **O**ver**l**ap-**A**dd (OLA) procedure. The exact workflow is best explained using the pseudo-code listed below:

```
01. //0.  pre-processing
02. clear output frame
03. //1.  determine frame type
04. IF frame == VOICED
05.   //1.1 traverse new pitch-marks array
06.   FOR i = 1 TO length(newPitchMarksArray)
07.     get new pitch-mark $pm_{n,i}$ at array position i
08.     get nearest original pitch-mark $pm_{o,i}$
09.     //1.2 create sub-window
10.     get new pitch-mark $pm_{n,j}$ closest to $pm_{n,i}$
11.     calculate window with size $n_{wnd} = 2 * (pm_{n,j} - pm_{n,i}) + 1$
12.     //1.3 get appropriate frame input, apply window, place on new pitch-mark
13.     get subframe (start:$pm_{o,i} - \frac{n_{wnd}}{2}$, length:$n_{wnd}$)
14.     apply window to sub-frame
15.     perform OLA with sub-frame and output frame (start:$pm_{n,i} - \frac{n_{wnd}}{2}$, length:$n_{wnd}$)
16.   ENDFOR
17. ELSE
18.   //2.1 unvoiced frames cannot be pitch-altered (they are unpitched)
19.   copy complete input to output frame
20. ENDIF
```

The algorithm performance primarily depends on the search algorithms of the pitch-mark arrays. Searching certain pitch-marks has to be done three times for every frame: To find the next pitch-mark (line **07**) in the contour generated pitch-marks array, the nearest pitch-mark to the found one in the original pitch-marks array(line **08**) and the nearest neighbour to pitch-mark $pm_{n,i}$ (line **10**) for being able to set the window size properly. Implementing these steps is done as part of the pitch-marks module as described in section 4.4.9 in two different days, namely the two operating modes `FULL` and `COMP`. By simply considering the search algorithm implementation both operating modes seem to perform approximately equal. The operations to be done in `COMP` mode consist of the two steps *(a)* looking for the current pitch-marks entry (PSOLA does not know the array index, just the frame position) and *(b)* the the trivial step of increasing/decreasing the array index by one and returning it's content. Using `FULL` mode the frame and array positions match, searching is not necessary. However the array has to be traversed to find the next/neighbouring pitch-mark, increasing/decreasing the counter by one would return a `NOMARK` entry. So the amount of operations to be performed is roughly the same. Despite of this fact `COMP` performs better overall because the array to be traversed is much shorter, it just contains `MARK` entries (Fig. 4.8).

The actual pitch frequency also affects PSOLA's run-time. With increasing frequency more pitch-marks fit into one frame and have to be dealt with. Anyway the developer has no influence on the resulting Fundamental Frequency (F_0) except of restricting it via setting an upper limit.

Implementation Details A specialty that differs online from offline calculation is the overlap-adding of sub-frames. Pitch-marks being located near the frame borders lead to sub-frames that exceed the current frame. When not considering the overhanging part in the adjacent frames' OLA operations some sort of disturbance comes into existence - the resulting signal at the frame's left and right border periodically decreases due to the skipped parts. To avoid this problem the algorithm has to additionally add the overhanging parts from neigh-

bours to the current frame. This is done via checking if one of the current frame's sub-frames overhangs. If so there is also sub-frame data of the counterpart reaching into this one. The overhanging part has to be calculated and additionally overlap-added with the sub-frame of the current frame. A simple carry-over from previous to current frame does not solve the problem even though it would save calculation time. The overhanging can also occur between current and next frame so the subsequent, future frame affects the current. An additional delay would have to be integrated in the processing chain. However there is already a delay caused by the smoothing operation - both delays would add up and lead to a disturbing input-to-output discrepancy. Utilising the smoothing-caused delay is also not possible because PSOLA needs the smoothed pitch value as input data.

4.4.13 Multipath Separation

Modulation Spectral Filtering

The **M**ultipath **S**eparation (MS) method using modulation spectral filtering consists of the seven fundamental steps listed below which are also visually represented in figure 4.10:

1. Pre-processing
2. Determine spectrum via FFT
3. Compress magnitudes
4. Apply **H**igh-**P**ass (HP) filter to spectral pins using delay states from previous calculations to include the temporal developing
5. Decompress magnitudes
6. Retransform filtered signal into time domain via IFFT
7. Post-processing

Figure 4.11 shows a filtering example as mentioned in **step 4**: The upper picture contains the overall spectrum of the current frame, the two pictures below the spectral component at *260Hz* whereas the lower left one shows the pin's unfiltered temporal developing and the lower right the developing after applied MSF.

Pre- and post-processing is done with applying a $80Hz$ high-pass filter to the time-domain signal in order to suppress humming and other non-speech related noise. The Fourier transform and it's inverse are executed using the DSPLib functions discussed in section 4.4.1. For filtering the spectral pins it is important to save the internal delay state of every spectral pin to access it in the subsequent filtering operation of the same pin (see sections 4.4.2 and 4.4.3 for details about internal filter delays). Two approaches turn out to be possible for a realisation: Either *(a)* instancing a filter object holding the appropriate delay states for every spectral pin or *(b)* using one filter instance and exchanging the delay states as needed by using an external array for saving them. Obviously method *(a)* leads to unnecessary overhead, method *(b)* should be preferred therefore. The applied dynamic range compressor additionally enhances the subsequent MSF's operating performance because of reducing the frequency pin's dynamic before applying the filter.

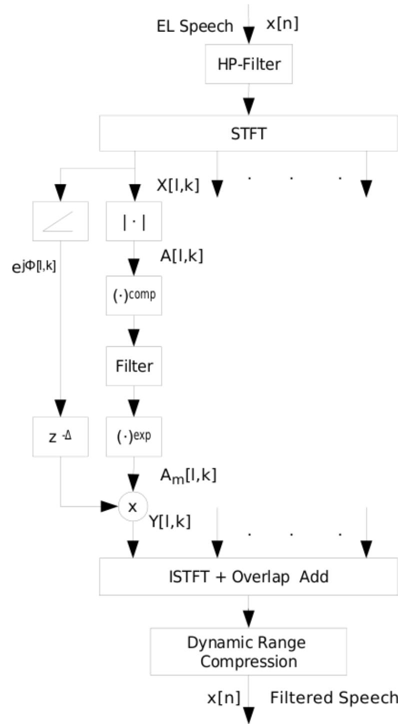


Figure 4.10: *Detailed block diagram of the MSF-based MS algorithm. The input data is transformed to the frequency domain in the STFT block and processed separately for every frequency pin. These pins are demodulated to magnitude and phase components, the magnitude compressed, filtered, decompressed and again remodulated with the untouched phase. The resulting spectrum is transformed back into the time domain and post-processed using a dynamic range compression filter (source: [20]).*

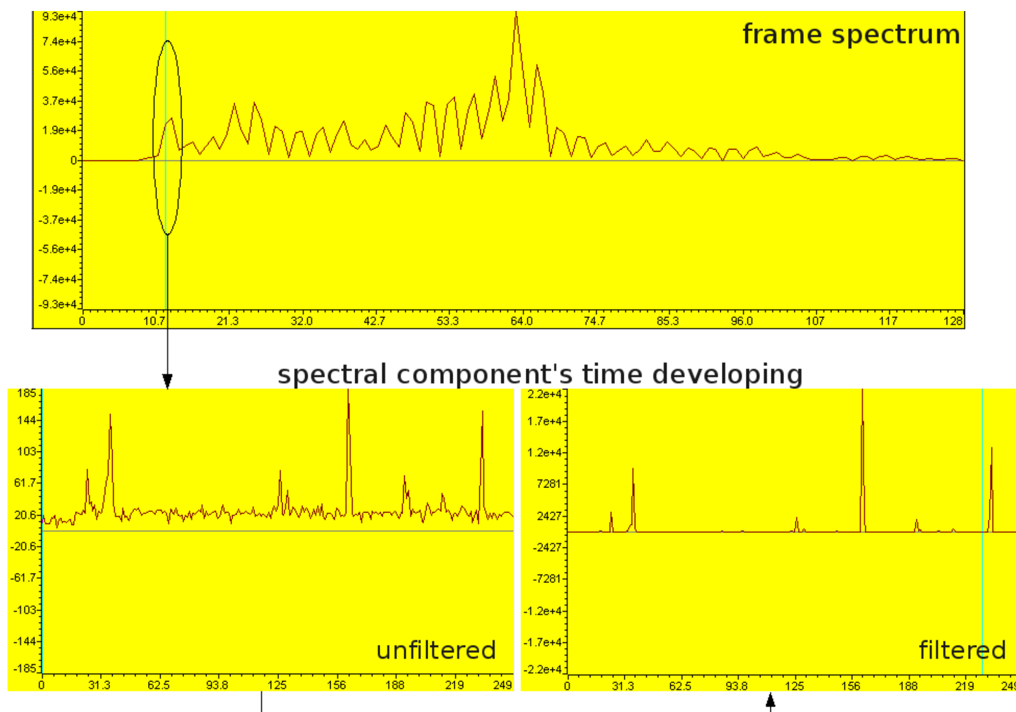


Figure 4.11: *MSF filtering of a certain spectral component in the DSP: The upper picture shows the overall signal spectrum and chosen spectral component, the lower pictures represent unfiltered (left) and filtered (right) temporal developing of the appropriate pin.*

The MSF-based method was tested using FIR as well as IIR filters in order to compare operation efficiency and resulting quality. The test results and concluding final implementation are focused on in section 5.2.3.

Spectral Subtraction

The principle processing chain when using **S**pectral **S**ubtraction (SS) is about the same as the one used by the MSF method. Both use pre- and post-filtering high-passes as well as the FFT and IFFT because either one operates using frequency domain information. Nevertheless the spectral subtraction works with estimating the noise components in the signal's spectrum based on statistics, more detailed the percentile of the frequency pin's magnitudes over time, using a certain amount $n_{samples}$ of recently passed values. $n_{samples}$ is implemented as user-definable parameter which indirectly controls the filter's accuracy and run-time respectively. Increasing $n_{samples}$ enhances the accuracy because the percentile value is based on larger statistics, on the other hand will the filter's run-time be reduced when decreasing the value. This is because of the sinking amount of values the calculation has to be based on. After all, the buffer size is a trade-off between both requirements - refer to sections 4.5.3 and 5.1.2 for a detailed analysis regarding this topic.

The actual filtering algorithm works with reducing the energy of the frequency pin currently under investigation by a percentile-dependent value. The exact formula is shown in equation (4.17). It shows that the current spectral component's energy $E(f,t)$ is reduced by twice the 10% percentile (for components in the higher frequency spectrum) or 20% percentile (for lower frequency components, they are of more importance for speech applications) of it's previous samples. This operation results in a higher-gain damping for spectral components with high percentile values what exactly meets up with the requirement of suppressing **D**irect-**R**adiated **E**lectro**l**arynx Noise (DREL) noise: The energy of the pin affected by DREL noise is constant over time, it's percentile converges to this constant value and leads to a complete elimination of the frequency pin after applying the subtraction. Other spectral components show more alteration in their energy developing, the centroid value keeps a lower value, the subtraction is performed with a smaller value and is less effective.

$$\begin{aligned} E(f,t) &= E(f,t) - 2 * percentile_{20}(E(f,t-i))\big|_{i=0}^{n_{samples}} \quad \ldots f < f_{focus} \\ E(f,t) &= E(f,t) - 2 * percentile_{10}(E(f,t-i))\big|_{i=0}^{n_{samples}} \quad \ldots f \geq f_{focus} \end{aligned} \tag{4.17}$$

The threshold frequency f_{focus} determines the spectral components for performing the 20% percentile operation. Components with frequencies lower than f_{focus} are altered this way, components above the threshold are processed using the 10% percentile. Through using different percentiles it is possible to control the suppression intensity. A percentile with lower percentage value leads to an overall smaller value of the percentile itself so applied on a frequency pin the energy reduction will be less intense. In the current implementation the 18th spectral pin was chosen as threshold. This leads to a threshold frequency $f_{focus} = \frac{f_s}{\mathtt{BLKSZ}} * n_{focuspin} = \frac{8000}{256} * 18 = 562.5Hz$ valid for the default implementation using f_s= $8kHz$ and BLKSZ= 256.

4.4.14 Pulse Generation

The pulse generation procedure is used to generate the driving signal for the shaker. The shaker's operation can be controlled by supplying it with an analog signal. This signal provides information about the cam intensity as well as the operating frequency: The cam is controlled via the pulse amplitude and shape, the vibration frequency with the pulse interval.

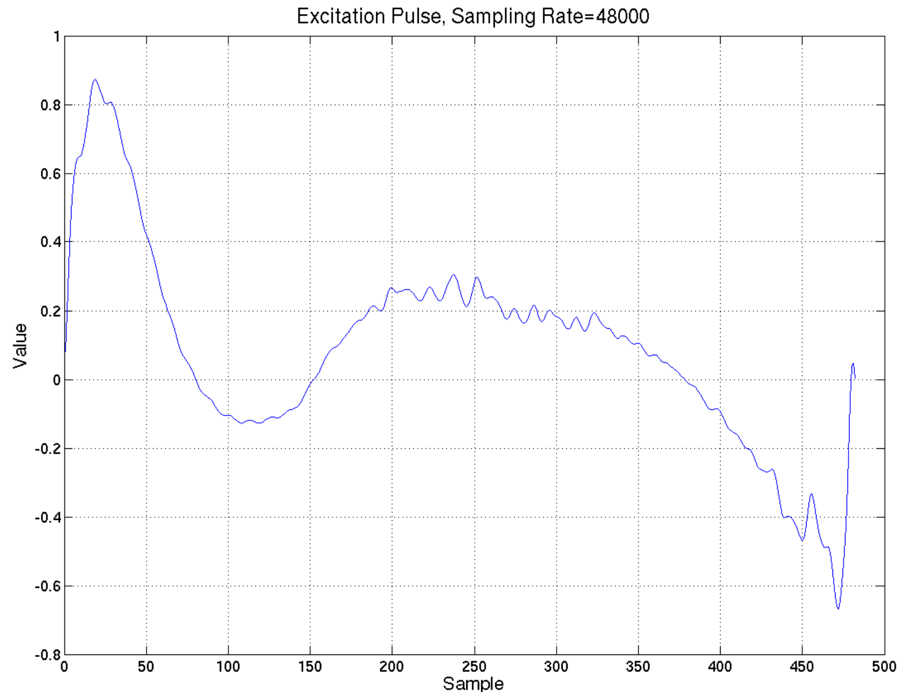


Figure 4.12: *Optimal excitation pulse as used by the F0gen in* "driving pulses output" *mode, the pulse itself is a HGS-model pulse as proposed by [22].*

Pulse interval generation is a straight-forward operation. A simple convolution of pulse and pitch-marks generates the correctly spaced pulses (Fig. 4.13). The pitch-marks are represented as array that holds pulses spaced by the calculated output frequency. So convolving pulse and pitch-marks generates a signal with the pulses at the pitch-mark positions. The pulse amplitude controls the shaker's cam range thus the motion level of the vibrating metal bolt. The amplitude-cam relation is proportional thus a higher amplitude results in a higher cam range. The optimal amplitude value causes a cam that leads to well audible sound with at the same time still being convenient for the electrolarynx user even after long periods of usage. Finally, an appropriate pulse shape has to be chosen. This parameter heavy influences the subjective impression of the resulting electrolarynx voice. At a glance, sharp pulses like rectangles and saw-teeth can be considered to produce a sharp tone whereas smooth pulses like *Hann pulses* produce a very soft tone. The optimal choice - the pulse producing the highest intelligible signal - can be found in between these shapes. Using the research in [22] combined with own listening tests the best choice turned out to be a **H***anquinet,* **G***renez,* **S***choentgen (HGS)-model* pulse[1] having a shape as shown in figure 4.12.

[1]This is a phonatory excitation model for speech signals particularly suitable for the synthesis of disordered speech.

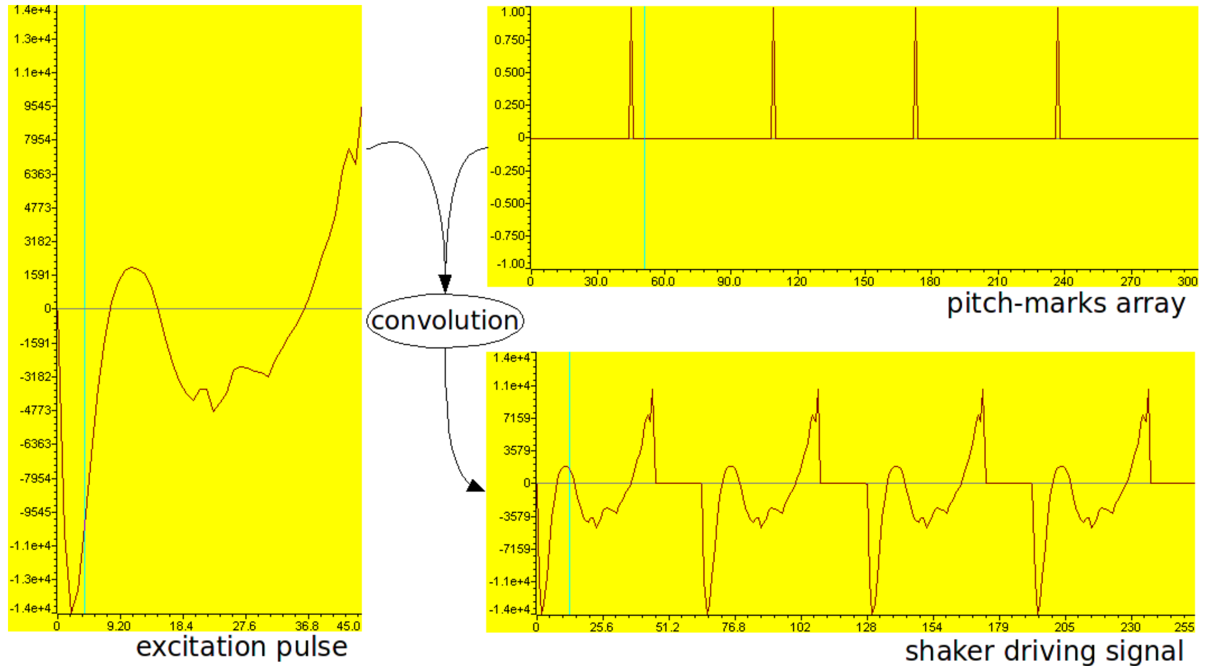


Figure 4.13: *Convolution operation to generate the shaker driving signal: The excitation pulse is convolved with the pitch-marks to create an analog signal with optimally spaced driving pulses.*

4.5 Real-Time Related Aspects

4.5.1 Online/Offline Data Processing Discrepancies

The pitch contour generation module makes heavy usage of the signal's statistical behaviour - mean cancellation, variance adaptation et cetera play an important role in this process. In fact a hundred percent correct calculation of these values is impossible in online data processing reasoned by not knowing future values. Every calculated mean value can at best (if there is enough memory available for storing all past values) represent the mean until this time instance. This might not be a big problem after enough data is collected anyway but right after starting the calculation process statistical values are more or less useless. Figure 4.14 shows an example of a processed nine second long input speech signal. It can be seen that the transition from formant contour (middle window) to new pitch contour (below window) is completely wrong at the beginning but starts getting accurate after a short time period. This effect is the result of failing statistical calculations: The processing pipeline consists of calculating the formant signal's mean value and deriving contour changes in relation to this value. These changes are applied to the final pitch. Due to neither the mean nor the variance of the formant contour being correct during this time period, caused by missing input data, the pitch contour is not showing the same developing as the formant contour for the first few frames.

The time duration after startup wherein these effects occur lies at about the first 60 to 90 samples, being mainly dependent on the used sampling rate. For example with a sampling rate of $8kHz$ this would be the first $7.5ms$ to $11.25ms$. The fact that with increasing f_s the same time range shows an equal statistical behaviour for captured data samples (minimal,

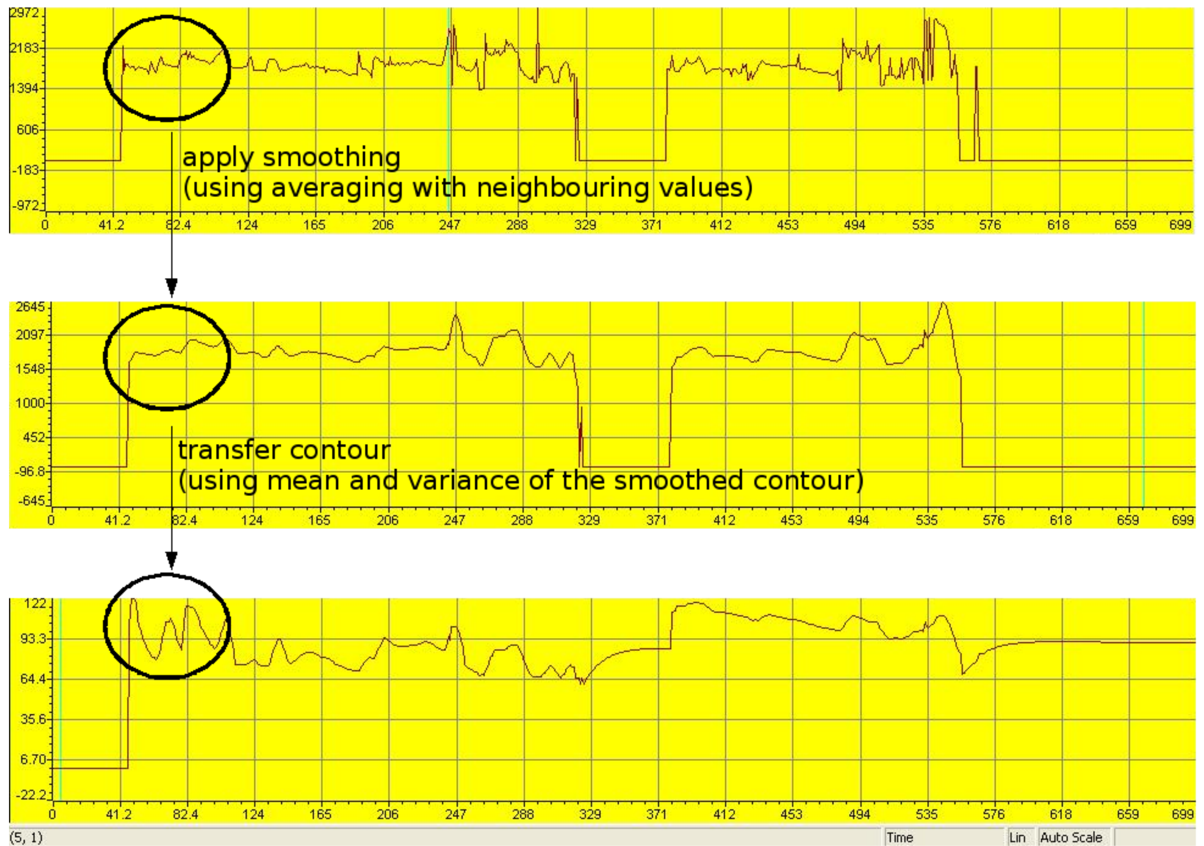


Figure 4.14: *Averaging problems at the calculation procedure's beginning: The upper picture shows the original formant contour as captured, after smoothing (center picture) it is converted to an adequate pitch contour (lower picture). Due to failing statistical calculations caused by the lack of input data right after starting the framework the contour is not correctly converted to the appropriate pitch contour.*

maximal and average values do not change when sampling faster) and should therefore also cause the same problems is not true because with higher sampling rates the data arrives faster in the processing pipeline - the effect of single values on the overall result decreases and initial fluctuations smoothen out faster. In fact, for an increasing sampling rate the time range decreases to $1.25ms$ to $1.88ms$ when sampling with $48kHz$.

Another highly interesting effect that occurred was the cancelling of the declination effect when using adaptive mean calculation. With switched-on declination the pitch slowly decays with time during an `ACTIVE` period. However when using adaptive mean calculation the sinking pitch value is recognised by the mean calculator and therefore affecting the resulting mean value. Due to the removal of the mean from the currently analysed pitch value in the contour parsing process the continuously decreasing mean causes continuously increasing results (can be explained by the subtraction: $x_{new} = x_{in} - x_{mean} \Rightarrow x_{mean} \downarrow \Rightarrow x_{new} \uparrow$). Increasing new pitch and sinking contour then simply cancel each other. This effect especially occurs at the beginning of the calculation process when just few values can be taken into consideration for the mean calculation and a single new input values still has a high effect on

the result. Figure 4.15 shows this effect: The final pitch contour shows nearly no declination during the two ACTIVE periods despite switched-on declination.

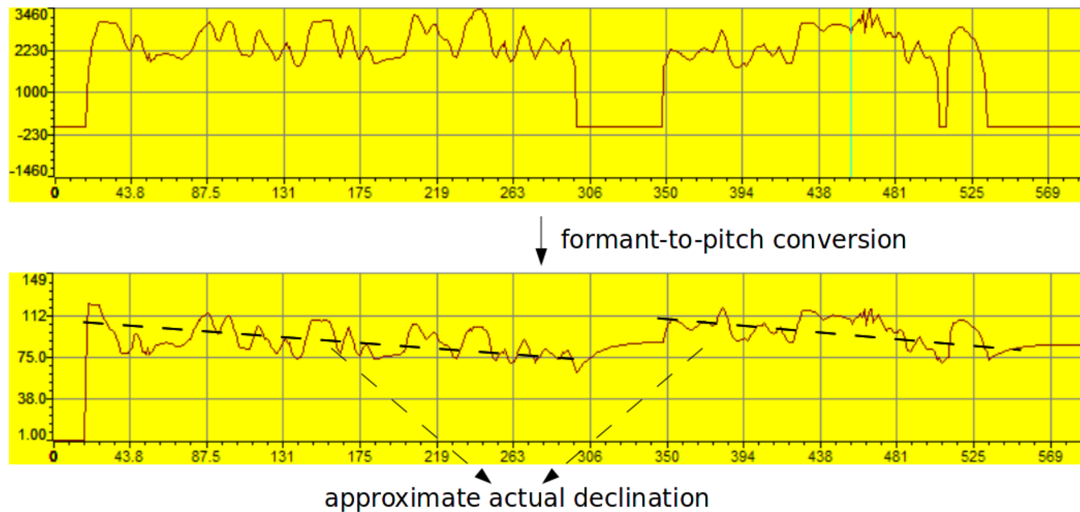


Figure 4.15: Declination cancelling effect with adaptive mean calculation: When using an adaptive calculation of the pitch's mean the applied declination is cancelled by the framework due to the mean adapting to the declination effect.

In some cases the calculation algorithms themselves also slightly differ between online and offline calculation. Offline calculation can be performed on the overall data stream at once, bordering effects due to block-wise calculation do not occur. This is even the case if the offline calculator works block-wise because carry-overs between frames can directly be taken into consideration when traversing the frames. The implemented PSOLA algorithm for example must be able to handle these carry-overs, both exact problem and solution are explained in section 4.4.12.

4.5.2 Timing Considerations

The biggest problem in the whole development process was handling the framework's real-time ability. To guarantee this property the overall calculation process for analysing the current input and outputting the result must be finished before new data arrives. In fact proving this ability in a mathematical manner is an immense effort, in most cases an approximation is sufficient and preferred therefore. Nevertheless the processing time does not fluctuate very much when the system operates using the same configuration. Crucial parameters, thus framework settings having a high influence on the overall run-time, are focused on in section 4.5.3. When keeping those parameters constant during analysing the overall calculation time the result is not going to alter much - checking the real-time ability with several examples should be sufficient to guarantee a certain run-time within a small tolerance.

Algorithm complexity and the computational cost resulting from it are crucial for most DSP applications. DSP's are optimised for mobility and energy efficiency rather than operating speed because of their operating area, mostly mobile and embedded systems. For the sake of offering an appropriate tool to cope with these problems the used development

environment (*Texas Instrument's Code Composer Studio*, see section 2.3.5) is able to let the developer investigate the algorithm complexity very detailed. This is done via counting CPU clock cycles and transferring the gathered data into timing values using the CPU clock speed. More on this topic, including exact calculation examples, can be found in the following sections.

The initial straight-forward implementation of the framework's algorithms was not able to finish it's calculations in time as demanded. Besides clock cycle analysis this could even be heard in the resulting audio: When already starting a new calculation process before finishing the previous it might happen that data samples are overwritten internally, mixed up between frames or skipped. The **D**igital Signal Processor **S**tarter **K**it (DSK) output therefore

Sounds noisy: Due to overwritten or mixed up data samples and therefore invalid calculation results

Sounds choppy: If the (periodically called) interrupt routine does not have a yet completely filled output buffer

Shows random artefacts: Caused by miscalculations and erroneous data

Shows periodic artefacts: The calculation period stays more or less constant what leads to the framework constantly skipping the same procedures; this periodicity causes some sort of musical noise

To overcome these shortcomings the first approach was to reduce redundant calculations. A clean programming style usually demands a high amount of modularity for the sake of being reusable, concisely and easily maintainable. This modularity had to be cut back slightly to efficiently make use of results from previous calculations and get rid of avoidable data copying procedures. In a second stage some internal data structures were rearranged. This might have led to a growing memory requirement but made search algorithms more efficient or even unnecessary and several processing loops shorter. For example the array containing the pitch-marks was initially realised as additional array of the same size as the frame's input sample array. It had either a `MARKED` or `NOT MARKED` flag at the appropriate sample position - calculating the PSOLA sub-window size and similar operations therefore had to traverse the array to find neighbouring `MARKED` entries. After rearrangement the array now only contains a single entry for every `MARKED` element containing the appropriate index - searching neighbour markers is unnecessary because they are on the adjacent positions in the array as well.
Besides high-level source code optimisations Assembler code can be rearranged to achieve the highest-possible degree of parallelisation. This is done via loading the eight functional units in the CPU (Sect. 2.3.2) as equal as possible. Certainly, rewriting complex algorithms in Assembler is a huge effort, optimising this code even more so. Therefore, in this work, a trade-off between speed and implementation effort was chosen by using the already pre-compiled and assembly-level optimised DSPLib functions (Sect. 2.3.5) as much as possible without implementing own low-level code. Furthermore, the CCS compiler provides the possibility to optimise high-level source code on it's own. This can be done by accordingly setting the compiler's options. Still it has to be considered that this type of optimisation is just available in *release* but not in *debug mode* - the compiler skips and combines commands from the high-level code so C and Assembler source simply differ too much for being able to guarantee proper debugging anymore. For more information about code optimisation please refer to [44].

4.5.3 Real-Time Effecting Parameters

The capability of processing data in real-time is determined by the amount of operations the CPU must execute during one data frame. This value can be altered by adapting the detail of operation for the executed tasks. The following section focuses on parameters having a high influence on this amount.

Block Size

The BLKSZ determines the temporal resolution of the framework hence choosing a smaller value causes the system to update captured data in shorter intervals. An application based on the block processing principle operates blockwise so the actual update interval is determined by needed time to completely fill an operating block with samples. With samples arriving every $\frac{1}{f_s}$ seconds the duration an input block/frame needs to be filled is calculated with $\texttt{BLKSZ}*\frac{1}{f_s}$, if overlapping is used this time is reduced by the overlapping factor (or update rate) and finally results in an overall needed interval as calculated in equation (4.18) again using the system's default setup.

$$t_{update} = BLKSZ * \frac{1}{f_s} * n_{overlap} = 256 * \frac{1}{8000} * 0.5 = 16ms \tag{4.18}$$

The optimal temporal resolution is dependent on the application's purpose: The higher the dynamic of the signal to be analysed the shorter is the optimal block size. With speech signals the optimal update interval t_{update} lies at roughly $10ms \ldots 20ms$ leading to a block size of $\approx 160 \ldots 320$, with higher f_s the block size increases accordingly.

From an implementation point of view the system's ability to cope with the accruing amount of input data determines the block size: Shorter frame/block sizes are in general less effectively processed than larger ones due to the sinking possibility of parallelising the handling of input data. Parallelisation is possible because most modern processors consist of more than one data processing unit that can partition - independent - parts of data for simultaneous calculation. The processor used in this project consists of eight floating-point units capable of being utilised for parallelisation (further information is available in chapter 2.3.2 and the DSK manual [40]). On the other hand the growing amount of input data for larger block sizes has to be considered. Especially the FFT point size which is directly derived from the block size has an high effect on the calculation time. Additionally significantly increasing the block size can lead to memory management problems in the DSP: Some algorithms in the processing pipeline are dependent on the last few input data frames. So they have to be stored in a ring buffer memory whereas one entry of the ring buffer contains the data samples of one captured frame. With increasing BLKSZ the amount of needed memory obviously grows proportionally. Finding a well-suited trade-off value is therefore crucial for the overall framework's performance quality. It is discussed in chapter 5.2.1 in detail.

When knowing the block size it is possible to directly calculate the allowed maximal duration of one frame using equation 4.19.

$$T_{frame} = T_{sample} * \texttt{BLKSZ} = \frac{1}{f_s} * \texttt{BLKSZ} \tag{4.19}$$

Using T_{frame} and the CPU clock rate of the DSP kit the amount of CPU cycles that fit into one frame can be determined - this value is the upper limit the program is allowed to last to produce real-time results of the enhanced audio data. The formula for calculating the amount of allowed cycles n_{cycles} is given in equation (4.20). The resulting value can now be used for cross-checking it with the implementation's actual one using CCS. The *update* parameter used in this equation is the update rate of the framework. This rate indicates the interval in which the program sends finished calculation results via interrupt to the audio codec. This implementation used an update rate *update* = 2 (originates from the 50% overlapping factor which is described in section 4.2) what leads to the framework always passing data to the audio codec when half a frame has been calculated completely - so two updates occur in one frame.

$$n_{cycles} = \frac{\frac{T_{frame}}{update}}{t_{CPU}} = \frac{\frac{\frac{1}{f_s} * \texttt{BLKSZ}}{update}}{\frac{1}{f_{CPU}}} \tag{4.20}$$

Examples for block sizes as well as allowed frame durations and CPU operations, considering the default sampling rate $f_s = 8kHz$ and a CPU clock rate $f_{CPU} = 225MHz$ (the TMS320C6713 standard clock rate), can be found in table 4.5. When increasing sampling rate the frame duration and CPU cycle parameters decrease proportionally. So increasing the input data accuracy negatively affects the framework's run-time.

Block Size [pt]	Frame Duration [s]	CPU Cycles
64	8m	0.90M
128	16m	1.80M
160	20m	2.25M
192	24m	2.70M
256	32m	3.60M
320	40m	4.50M
512	64m	7.20M

Table 4.5: *Maximal allowed frame duration and amount of CPU cycles dependent on the used BLKSZ in order to guarantee real-time processing.*

Linear Predictive Coding Order

Altering the LPC order (p) influences the accuracy of the formant calculation. This behaviour can be reasoned with the consideration of more past samples $x[n-\alpha]$ to predict the current one $x[n]$. Formants are calculated through the roots of the polynomial that is described by the LPC's coefficients: $a_3 * x^3 + a_2 * x^2 + a_1 * x + a_0$ is the polynomial with a_α being the LPC coefficients. In general it is the case that with increasing the LPC order the approximation of the predictor with the actual input signal gets more accurate. Anyway if chosen way too high this can negatively influencing the approximation, the approximator starts getting affected by signal details rather than the overall developing. This also leads to a depency between sampling rate and optimal LPC order - a well-known rule of thumb claims the optimal order of being roughly $p \approx \frac{f_s}{1000} + 2$. Besides the aspect of approximation quality the implementation complexity also has to be taken care of. More coefficients have to be calculated when increasing the LPC's order (as explained in chapter 2.2.2) and this is again effecting the calculational complexity and temporal effort. So it can be seen that the actually

implemented order has to be chosen carefully - choosing a too low order might even lead to undetected formants. This happens when the approximation produces polynomial roots that are located below the threshold of the formant tracker.

The majority of papers focusing on LPC-based formant tracking propose an order $p = 12$, the ones with focus on detail rather than computational effectivity even 15 or 20. This implementation in fact performed best using an order of 10. This value also exactly matches with the previously mentioned rule of thumb: $p \approx \frac{f_s}{1000} + 2 = \frac{8000}{1000} + 2 = 10$. Tests with $p = 9$ also resulted in relatively good values, the same was true for $p = 11$. Nevertheless choosing an order of below 9 the enhancement application started skipping formants and led to incorrect formant developings, when using orders of above 11 the possibility of not finishing the calculation in time already becomes too high. These values are also summarised in table 4.6.

	Proposed Value	**Used Value**
LPC Order	$12 \dots 20$	$9 \dots 11$

Table 4.6: *Example values for the LPC approximator order as used in common applications as well as in this framework.*

Exit Conditions for the Polynomial Roots Calculation

As with the LPC order, discussed in chapter 4.5.3, the polynomial root calculator's accuracy directly determines the quality of the formant tracker. Polynomial root calculation is very problematic in DSP processing environments because of it's intense computational effort. This effort is caused by the iterative approximation technique used for numerical solutions, the root values are repeatedly calculated until convergence. In order to reach a certain level of accuracy eventually huge amounts of iterations have to be executed. Practical algorithms therefore include several exit conditions to keep the run-time within boundaries. These exit conditions are:

- The maximal accuracy of the result (fluctuating just occurs within a small data range any more)
- The maximal number of iterations (the total amount of repeated calculation steps)

	Range	**Used Value**
Value Accuracy	$10^{-5} \dots 10^{-15}$	10^{-6}
Amount of Iterations	$100 \dots 500$	90

Table 4.7: *Example exit condition setups for the iteratively working polynomial roots calculator.*

A bad choice of these parameters leads to the violation of the system's real-time ability. If too high the processing pipeline wastes all of it's available calculation time determining the polynomial roots. Too low values cause outliners in the formant contour because the polynomial roots for identifying the formant have not been converging yet and are inaccurate. Example configurations used in other practical applications as well as the values that turned out to perform best in this framework are provided in table 4.7.

Sorting Array Sizes

Two calculation routines in the enhancement framework are dependent on sorted input data arrays. Sorting is a calculational expensive procedure due to the usually high amount of loop cycles especially with large data arrays. In order to keep the sorting process short these arrays should be kept as short as possible with not negatively affecting the routine's quality at the same time due to too little input data. The following paragraphs therefore investigate these calculation routines, the formant smoothing and spectral subtraction's percentile calculation, trying to determine optimal trade-off values for their input array lengths as well as the best fitting implementation for the sorting algorithm itself.

Formant Smoothing Filter Size The same as with several other parameters in this framework the smoothing filter size's upper limit is not determined by the resulting speech intelligibility but the calculational effort. Mainly this is the median filter's responsibility. A median calculation always depends on a sorted array of values (with length *filter size*) - a sorting procedure has to be performed for every resulting median value. In sum this results in an amount of sorting processes dependent on the overall length of the data values array and the smoothing filter size.

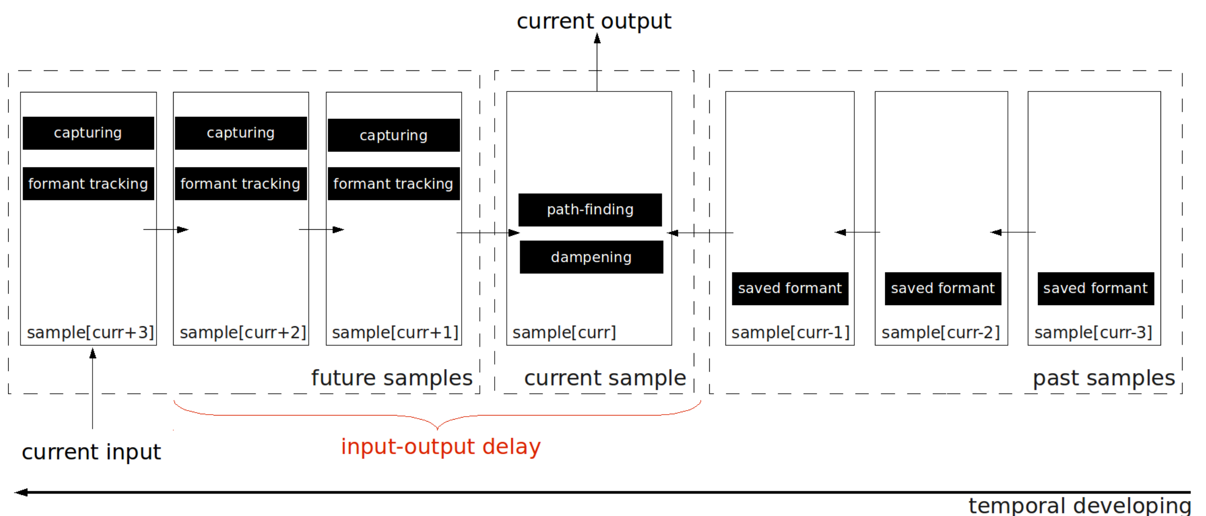


Figure 4.16: *The pitch contour generators processing line using a delay of 3 frames. A block delay of 3 means a difference of 3 frames between current input and output, "future" frames are frames that are already captured and processed but not yet outputted, past frames are just stored to provide their formant data to the smoothing algorithm.*

The mentioned data values array is the one holding the list of formants from previously captured frames. It's size is determined by either system latency (in delayed frames) or smoothing filter size - whatever demands a longer memory of past frames determines the it's length. With common applications this will always be the smoothing filter size: The option to delay frames (the actual number is determined by the counting the difference between currently captured input and outputted frame as shown in figure 4.16) only had to be implemented to be able to

perform smoothing after all[2] and should be kept as small as possible. So the smoothing filter size is mainly responsible for causing this delay and will always be the upper bound for the discussed array therefore. It's size lies in the area of several frames and additionally adds up with the anyway present delay due to block processing as treated in section 2.2.1. Without smoothing there is even the possibility to not delay frames at all[3] - in this case the single input samples are only delayed by the always existing block processing delay. Figure 4.16 shows the explained processing chain for a block delay size of $n_{delay} = 3$ frames what leads to a maximal smoothing filter size of $2 * n_{delay} + 1 = 2 * 3 + 1 = 7$.
Considering this facts leads to equation (4.21) for calculating the resulting amount of sorting operations. It can be seen that the smoothing complexity is only determined by the smoothing filter size, $O_{sorting}()$ indicates the algorithmic complexity of the sorting algorithm itself.

$$\begin{aligned} n_{sortings} &= len_{datavalueslist} * O_{sorting}(n_{filtersz}) \\ &\hat{=} len_{filtersz} * O_{sorting}(n_{filtersz}) \\ &\propto n_{filtersz} \end{aligned} \tag{4.21}$$

Spectral Subtraction's Percentile Buffer Size As with median calculations in the smoothing filter the percentile calculation used by the spectral subtraction module depends on sorted data arrays. In fact every sorting procedure on the $n_{samples}$ long data array holding the past energy values has to be performed once for every spectral component. This number, as calculated in equation (4.22), is again defined by the used BLKSZ value when considering that the amount of spectral components is only defined by the frequency resolution (equal to BLKSZ) and the fact that the spectrum is Hermitian symmetric (Sect. 4.4.1).
This results in 129 sorting procedures of $n_{samples}$ long data arrays when using the default block size of 256.

$$n_{sortings} = \lfloor \frac{\texttt{BLKSZ}}{2} \rfloor + 1 = \lfloor \frac{256}{2} \rfloor + 1 \tag{4.22}$$

With additionally setting $n_{samples}$ to a value of 10 (the percentile is calculated from the last 10 spectral pin's magnitude values) the spectral subtraction algorithm achieved the best possible DREL suppression - the percentile is calculated using the largest possible statistics - with at the same time still being capable to execute the complete enhancement framework's processing chain in real-time.

Sorting Algorithm For picking an appropriate sorting algorithm the results gathered by [48] were used. This paper investigates the performance of sorting algorithms with small input data sizes (below 80). As mentioned above the implemented routines in this framework also only handles small amounts of input data to be sorted. This leads to the optimal sorting algorithm for this application being one that *(a)* performs well on small input arrays and *(b)* shows as little overhead as possible. [48] proposes *insertion sort* for input sizes of four to six, therefore this algorithm has been implemented. Nevertheless the first version of the sorting algorithm was realised using *bubble sort*. Due to it's ease of implementation bubble sort was a good candidate for initially testing the functionality.

[2]Without delaying complete frames the formant data of ones subsequent to the currently processed would not be available, smoothing could just be performed on past elements and the current. This would make the smoothing filter - that prepares the frame that is about to be passed to the output - unable to adapt to immediately following fluctuations even when being extreme!

[3]This is not completely true because the formant tracker's global path finder would also cause a delay but there is none implemented anyway.

Having both algorithms available it was possible to run a performance comparison on our own. As to be seen in figure 4.17, using the formant smoothing block for the comparison, the preference of insertion sort over bubble sort could be reproduced and matched with the results of [48]: The average amount of CPU cycles for median smoothing that utilises the sorting algorithm was less with insertion sort, namely an average saving of 6159 CPU cycles or 10.15% for the ten performed tests.

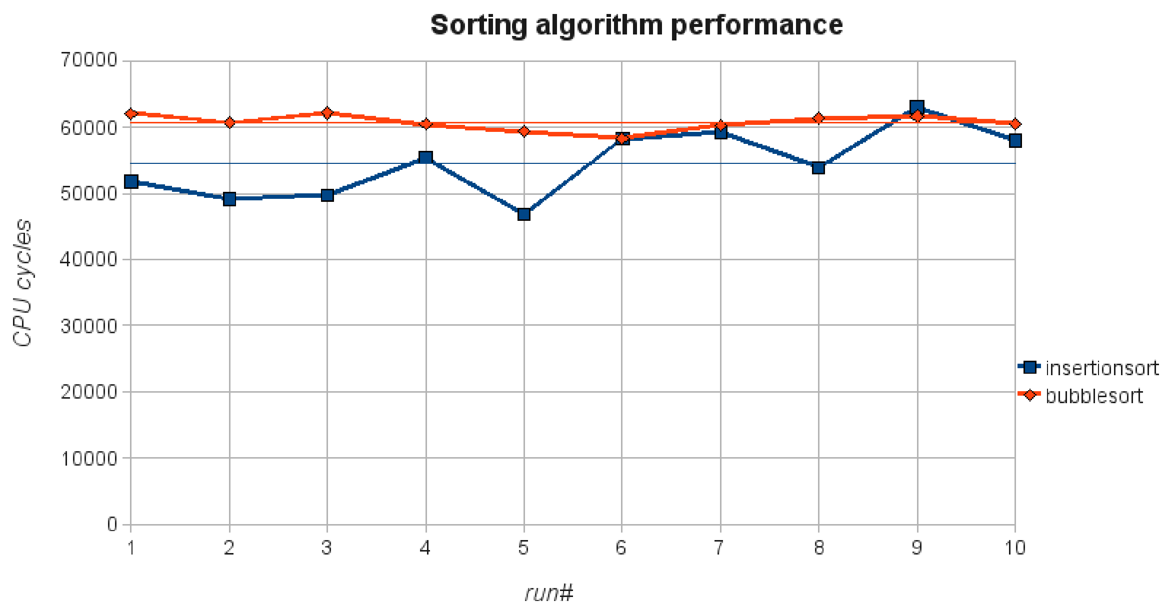


Figure 4.17: *Performance measurement of the implemented sorting algorithms* insertion *and* bubble sort*. The metric used for comparison are CPU cycles needed by the appropriate algorithm to completely execute the* `medianfilter()` *function with a smoothing filter size of* 7*.*

Speedup Through Retaining Array Order To additionally speed up the sorting process it was experimented with retaining the array order between subsequent sorting procedures of the same data set. Both algorithms that are dependent on sorted input, formant smoothing and spectral subtraction, work iteratively - the array to be sorted only changes by two values with every call: *(1)* the oldest element is removed and *(2)* a new one is added to the data set. The idea was therefore to keep the array order to only have to insert the new value into the already sorted data set. Unfortunately this is not possible for either algorithm: The median filter in the formant smoothing block which needes sorted input is surrounded by the outliner cancellation and linear smoothing filters. These filters change the array's values between subsequent calls of `medianfilter()` causing it to be unsorted again.

For the spectral subtraction block a comparable problem occurs: Here the entries order in the data set is an integral part of the subtraction algorithm. Spectral subtraction makes use of the single spectral pin's trajectories. The necessary time information is again included in the data value's position in the buffer array - the lower the entry's index the older it is. If this order is scrambled due to sorting the temporal information becomes invalid. A

workaround could be an additional array containing this temporal information for every entry. Nevertheless if doing so the oldest entry which has to be replaced by the new value always has to be searched for because it's position is unknown after sorting. Additionally the complete buffer array's time data has to be adapted because with replacing the oldest value by the newest all other entries automatically grow older as well. Without overhead the usage of pre-sorted array brings an advantage of about 20% in average, the spectral subtraction block is calculated in $2.05 * 10^6$ CPU cycles compared to $2.5 * 10^6$ cycles without pre-sorting. So with additional overhead this approach is nearly as time consuming as retaining the order based on time and newly sorting by value whenever needed in the first place.

Chapter 5

Results

5.1 Measurement and Calculation

5.1.1 System Delay

To determine the maximal delay of the framework two side conditions have to be taken into consideration:

1. Practical limitation: The user acceptance regarding the unfamiliar delay effect

2. Technical limitation: The amount of available system memory

When implementing the delay line with respect to these aspects the practical limitation soon turned out to be the crucial one. In fact the final application's implementation is showing just a small system delay resulting from *(a)* the processing delay which is mandatory to properly perform smoothing and the **P**itch-**S**ynchronous **O**ver**l**ap-**A**dd (PSOLA) routine and *(b)* the block processing caused delay. The block processing delay is mainly dependent on the overlapping factor (see [16]) whereas the processing delay can be set manually and may range from two frames upwards. It's upper limit is from a technical point of view just determined by the available amount of memory, practically by speech intelligibility and smoothness of the conversion flow. These factors are defined by ([46]):

1. Difficulty in listening to one-way speech
2. Difficulty in talking
3. Difficulty in conversing during turn-taking

Practical Limitation

There have already been several researches regarding the upper latency threshold for guaranteeing a flawless and perceptual conversation quality. Nevertheless the resulting limit varies a lot between the studies, lying roughly between $100ms$ and $600ms$ ([17]). However, an official recommendation by the **I**nternational **T**elecommunication **U**nion **- T**elecommunication Standardization Sector (ITU-T) defines the one-way or mouth-to-ear delay to be optimal for highly interactive applications below $100ms$, acceptable below $150ms$ and still tolerable between $150ms$ and $400ms$ ([10]).

Soft-Phone System	Average Delay [ms]
Ekiga	213
MSN Messenger	88
Skype	142
Yahoo Messenger	138
∅	**145**

Table 5.1: *Mouth-to-ear delay of several state-of-the-art soft-phones using VoIP (source: [1]).*

Dependent on the used sampling rate, block size (Equ. (4.19)) and overlapping factor, the system's processing delay results in for example $32ms$ using the default settings f_s= $8kHz$, BLKSZ= 256, 50% overlapping and a lag of two frames. Therefore, the latency caused by the enhancement routine is located in an area that seems to be sufficient for interactive applications. Considering an upper limit of around $150ms$ leads to a maximal number of *seven* delayed frames for the processing delay again with the same configuration which adds up with the always present block processing caused delay of *one* frame at maximum. However when talking about the maximal delay to be allowed another fact also has to be considered: When the framework is configured to produce an enhanced signal as output (Sec. 3.2) this signal will probably just be used as input for the actual communication system - so the speech enhancement framework serves as some kind of pre-processing for the distracted electrolarynx speech signal before actually being transmitted. The used communication system could be for example a telephone, mobile or **V**oice **O**ver **IP** (VoIP) device. Especially with current VoIP systems, due to the packet switching paradigm of it's communication channel, the Internet, there is an additional delay that adds up with the delay of the enhancement framework. State-of-the-art applications show a one-way delay of around $145ms$ as shown in table 5.1. The mentioned values of course vary dependent on used transport protocol (TCP, UDP), communication protocol (*Speex*, *G.711*, . . .) as well as the geographic location of sender and receiver. To still stay below the tolerable delay threshold as suggested by the ITU-T, this results in an allowed delay of $400ms - 145ms = 255ms$ for our system.

Technical Limitation

The practically caused delay is distinctly below the technically caused one, which is generated by memory limitations. For every additional delayed frame, two more frames have to be stored because the currently processed frame must be center-aligned when applying the formant contour smoothing (Fig. 4.16). Otherwise the smoothing result would not be very accurate because of ignoring neighbouring values from one side. Increasing the processing delay by one, therefore, automatically increases the number of "future" frames: The addition of one more frame on the right side (relative to the currently processed frame as shown in the figure) must be balanced with an additional frame on the left in order to keep the balance both sides.
Storing one frame's data needs at maximum 1556 bytes of additional memory as calculated in (5.1) considering the structure of the marker list entries as listed below. Therefore increasing the delay size by one leads to $2 * 1556 = 3112$ bytes of additional memory.

```
typedef struct
{
   float f0, fx;
   float declin;
   float timestamp;

   unsigned char vuv, vad;

   unsigned short* pitchmarks;
   unsigned short pitchmarkssz;

   float* sigbuf;
}
```

The `PITCHMARKSSZ` referred to in (5.1) depends on the used pitch-marking mode (Sect. 4.4.9) and can either be equal to BLKSZ in `FULL` mode or about $\lfloor \texttt{BLKSZ} * \frac{F_0}{f_s} \rfloor$ as calculated in equation (4.14) in `COMP` mode. In case of calculating the maximal memory usage the mode with higher memory usage has to be picked for further calculations - as hinted by it's name this is the `FULL` mode because it marks every sample and not just the ones having a `MARK` entry. The final memory demand of one marker list entry is therefore calculated with equation (5.1)'s last three lines. When switching off all possible debugging routines (they occupy much memory due to logging a high number of data values) the largest possible processing delay size is equal to 467 frames featuring a memory demand of $(2 * 467) * 1556 = 1.45MB$. In fact this value is high enough for not having to consider the technically caused system delay limitation at all - an output delay of 467 in the enhancement routine means having an overall time lag of $(467 + 1) * T_{frame} = 14.98s$ (including the block processing delay of one frame and again using the default configuration, see equation (4.19)). This duration is far beyond being a usable value.

$$
\begin{aligned}
mem_{total} = \; & mem_{f0} + mem_{fx} + mem_{declin} + mem_{vuv} + mem_{vad} + mem_{timestamp} + \ldots \\
& \quad mem_{pitchmarks} + mem_{pitchmarkssz} + mem_{sigbuf} = \\
= \; & 4 * mem_{float} + 2 * mem_{char} + 1 * mem_{short} + \ldots \\
& \quad \texttt{PITCHMARKSSZ} * mem_{short} + \texttt{BLKSZ} * mem_{float} \\
= \; & 4 * mem_{float} + 2 * mem_{char} + 1 * mem_{short} + \texttt{BLKSZ} * mem_{short} + \texttt{BLKSZ} * mem_{float} = \\
& 4 * 4 + 2 * 1 + 1 * 2 + \texttt{BLKSZ} * 2 + \texttt{BLKSZ} * 4 = \\
& 1556 \; bytes_{\texttt{BLKSZ}=256 \text{ (default)}}
\end{aligned} \tag{5.1}
$$

Delay Measurement

To verify the calculated delay the real-time generated output signal produced by the framework was measured and analysed. Therefore a specially designed input signal was created. This signal consisted of two sync pulses preceding the actual speech signal, the electrolarynx spoken sentence *"Die Oma trinkt einen Kaffee."*. The generated signal was stored as audio file for later comparisons with the captured result (both is found in figure 5.1) and passed to the **D**igital **S**ignal **P**rocessor (DSP) enhancement framework during the measurement. The measurement procedure itself consisted of the following steps:

```
01. //1.  pre-processing
02. //1.1 correctly wire components
03. connect source computer (contains input audio file) to framework input
04. connect framework output to sink computer (captures resulting output)
05. //1.2 start environment
06. start DSP in bypass mode
07. line-up environment
08. start capturing
09. //2.  measurement
10. play input until sync pulses are passed
11. switch on F0gen during silent phase between sync pulses and speech signal
12. wait until finished ...
13. //3.  post-processing
14. stop capturing and save result
15. post-process input and captured file by synchronising them using the sync pulses
16. measure latency between input and resulting speech signal
```

Some notes regarding the measurement:

- Capturing was performed using the *Wavesurfer*[1] software
- The audio file was played back using *Wavosaur*[2] software
- The sync pulses passed the DSP in bypass mode and are not delayed by the enhancement processing pipeline therefore - this enables the synchronisation in the first place

Parameter	Value
delay	3 frames
BLKSZ	256
f_s	$8kHz$
overlapping	50% (≙ update factor 2)

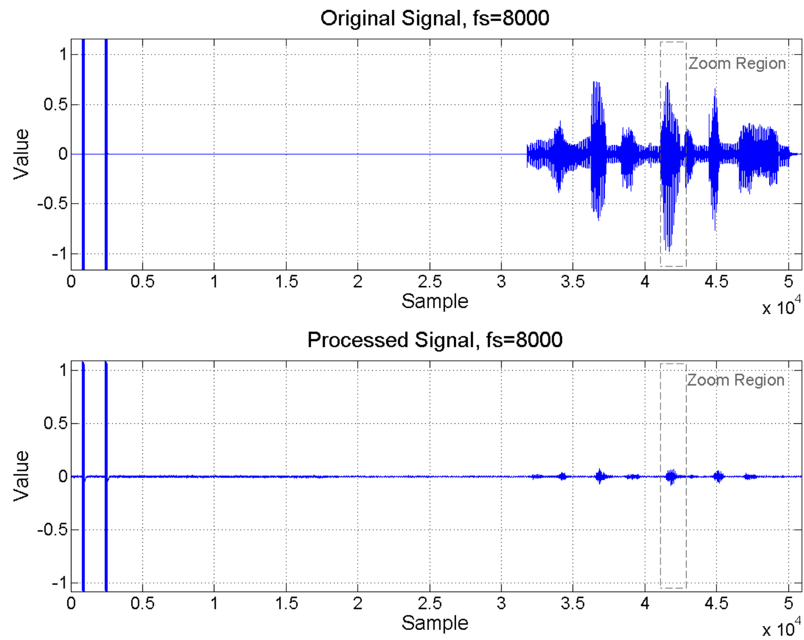
Table 5.2: *DSP framework setup as used in the processing latency measurement.*

The post-processing measurement was done twice: Once with Wavosaur and once with Matlab. To measure the actual latency the difference between sync pulse and speech signal start as well as sync pulse and speech signal peak value (occurs during the word *"... einen ..."* in the sentence) was quantified for both input and output signal. The analysis using Wavosaur resulted in a measured processing delay of $5.190s - 5.238s = -48ms$, the delay when determined using Matlab turned out to be 386 samples or $\frac{n_{delay}}{f_s} = \frac{386}{8000} = 0.04825 = 48.25ms$ (using equation (4.19)). Both results corresponded with the calculated value in equation (5.2) considering the framework's setup as shown in table 5.2.

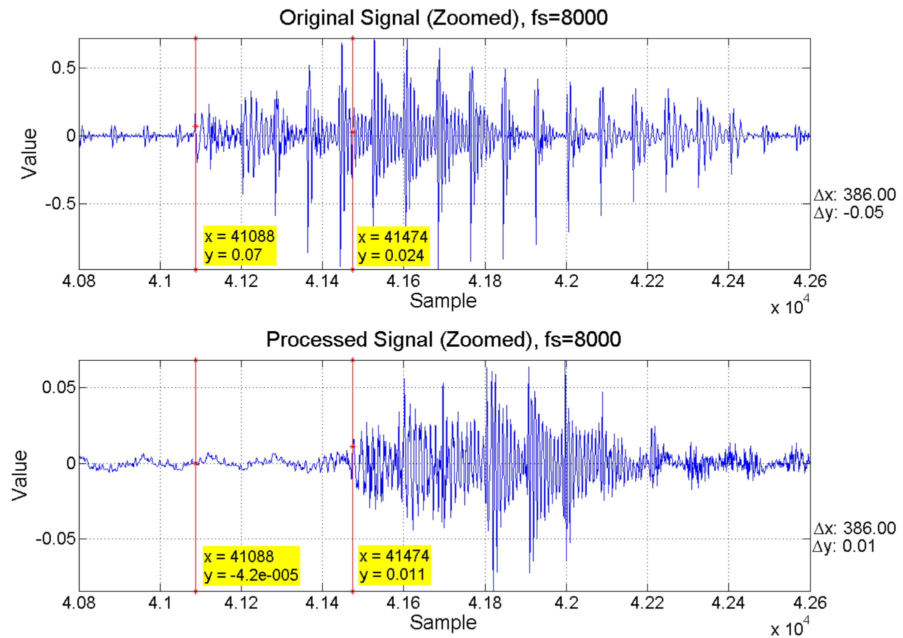
$$t_{delay} = n_{frames} * \frac{1}{\text{update}} * \frac{\texttt{BLKSZ}}{\frac{1}{f_s}} = 3 * \frac{1}{2} * \frac{256}{\frac{1}{8000}} = 0.048s = 48ms \tag{5.2}$$

[1] http://www.speech.kth.se/wavesurfer/

[2] http://www.wavosaur.com/



(a) Full view of the processing delay measurement signals, the long period between sync pulses and speech signal is necessary to have enough time for switching on the F0gen module during the measurement. The marked regions indicate the location of the zoomed-in in the overall signal as shown in the below figure.



(b) Zoomed view of the processing delay measurement signals, already equipped with the delay measurement markers. Their difference indicates the lag between speech signal start of the original and processed signal and therefore the processing delay.

Figure 5.1: *Input and output signals used in the processing delay measurement process. The two sync peaks at the beginning of both signals as well as the MSF caused loudness decrease can be clearly seen here.*

For the sake of accuracy this measurement was also performed using a pulse instead the above used valid speech signal. The sharp-edged pulse simplifies identifying the beginning of the delayed signal and therefore leads to a more exact location of the reference sample for determining the latency. The hereby utilised signal is shown in figure 5.2, the executed procedure was the same as above. With a resulting processing latency of $47.12ms$ or 377 samples the obtained delay also nearly matched with the previously determined what indicates the correct measurement in both cases. A completely exact match is nearly not possible to achieve because the pulse gets distorted during transmission (cable capacities, **A**nalog-To-**D**igital / **D**igital-To-**A**nalog (AD/DA) conversion, et cetera) and therefore always looks different in the captured result - what makes it hard to find the reference sample.

Worth mentioning for this measurement process is that the detected processing delay does not represent the actual mouth-to-ear latency of the complete framework. The sync pulses used for aligning input and output are also delayed by the block processing framework even when passing it in bypass mode. So the determined value of about $48ms$ only indicates the delay caused by the interrupt-service routine where the enhancement procedures are executed - as mentioned before there is an additional latency of BLKSZ samples caused by framing with 50% overlapping (refer to [16] for more about this topic). So the overall mouth-to-ear delay turns out to be $48ms + 32ms = 80ms$.

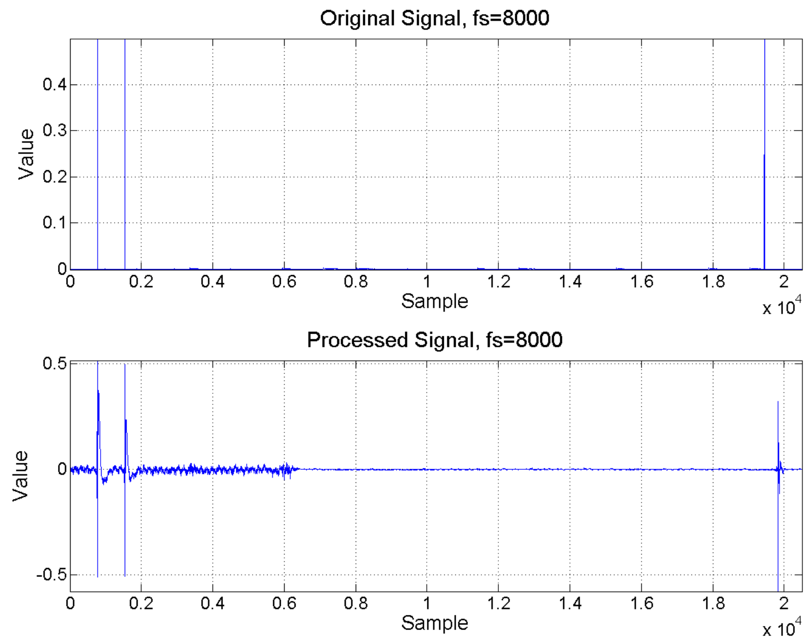


Figure 5.2: *Alternative signals used in the processing delay measurement process. The two peaks at the beginning are again used for synchronising input and output, the delayed and investigated signal is a pulse here as well. This simplifies determining the reference sample at the beginning of the* 3rd *pulse and increases the accuracy.*

5.1.2 Multipath Separation Modules

Modulation Spectral Filtering

In the process of analysing the **M**ultipath **S**eparation (MS) results via informal tests the achieved intelligibility of the **M**odulation **S**pectral **F**ilter (MSF) algorithm turned out to be higher than the one obtained by the alternative **S**pectral **S**ubtraction (SS). Figure 5.3 as well as another example in figure 5.5 show the original and MSF filtered signal's spectrum for electrolarynx speech with **D**irect-**R**adiated **E**lectro**l**arynx Noise (DREL) component. One can clearly see the suppressed constant component in the processed spectrogram. In the unprocessed one the DREL component is identified as horizontal line at about $110Hz$ and it's harmonics in figure 5.5a and approximately $230Hz$ in figure 5.3a respectively. Figure 5.5b shows the MSF filtered speech signal of the sentence *"Ich will ihn nicht umfahren sondern umfahren!"*, figure 5.3b *"Eins, zwei, drei, vier, fünf, sechs, sieben, acht, neun, zehn."*.

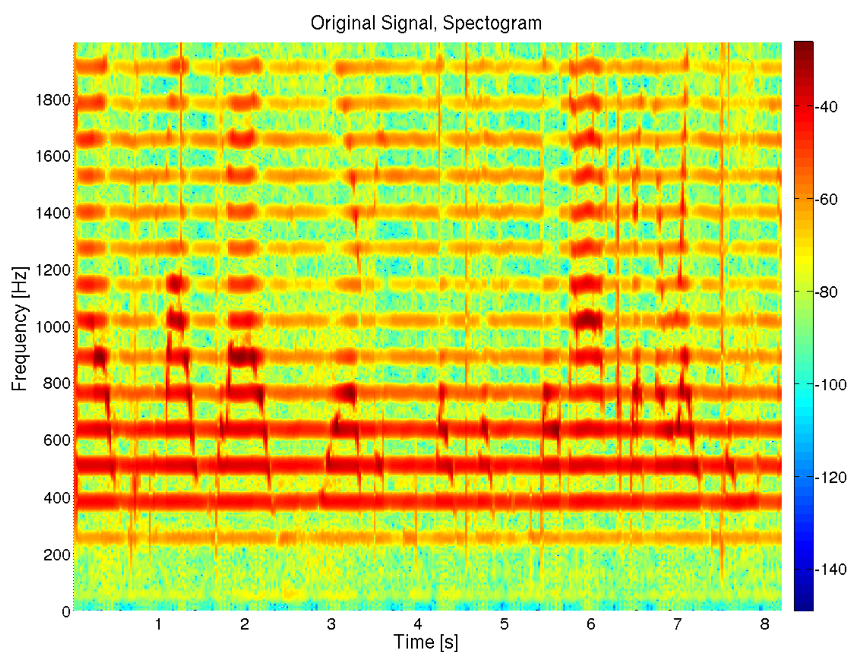
Nevertheless there are two drawbacks when directly comparing MSF generated results with the ones achieved via SS:

- Lower degree of DREL suppression

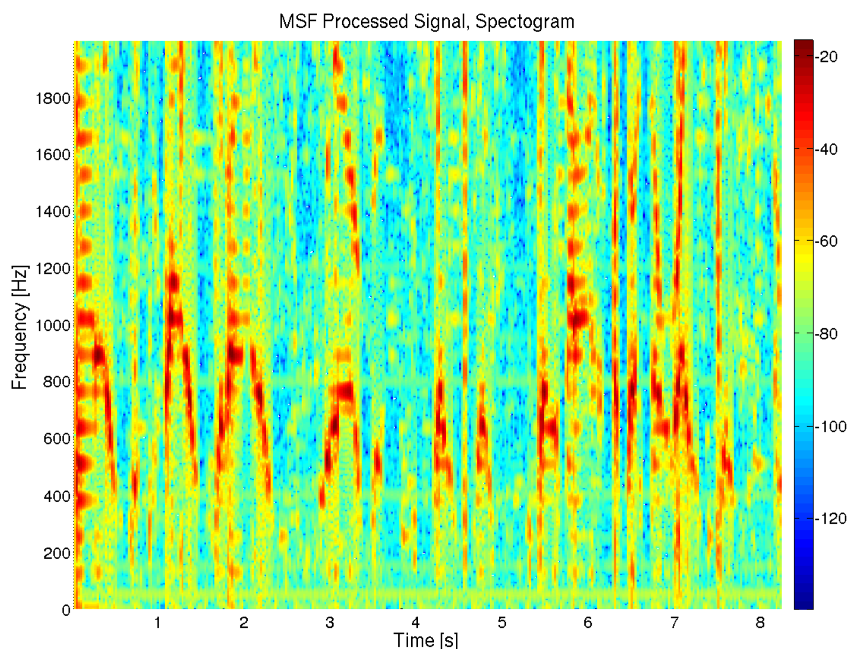
  The first and more significant disadvantage of modulation spectral filtering over the spectral subtraction method is the lower degree of DREL suppression. Figure 5.5 clearly show this problem - the constant component as shown in the signal's spectra is still (slightly) visible in the MSF output. This is not the case in the SS's output spectrum. An explanation of this phenomenon is the MSF filter's higher sensibility towards the constant component's dynamics. Due to the energy of the DREL's spectral component being not completely constant but slightly altering the SS algorithm is more suitable to adapt to these fluctuations. It is based on a statistical approach, the percentile, to perform filtering. Operations based on statistics are in general less vulnerable to small changes of the observed parameter.

- Decreased output signal loudness

  The second drawback is the decreasing loudness of the resulting signal. The loss of loudness can be reasoned with the working principle of modulation spectral filters. They operate on every frequency pin independent of whether the spectral component is part of the speech or noise signal. This leads to suppressing constant components of the speech signal spectral as well - it decreases in it's amplitude. A comparison between the resulting signal amplitudes is given in figure 5.4. The left plot shows the signal with adapted gain, thus the loudness it should have, the right plot contains the actual signal with the dampened amplitude.



(a) Spectrogram of the original signal which is heavily affected by the DREL component, the horizontal line at about $230Hz$ *and it's harmonics.*



(b) Spectrogram of the MSF enhanced signal showing just a weak remaining DREL component.

Figure 5.3: *Performance of the MSF-based multipath separation when being applied to the* $8.2s$ *long electrolarynx speech signal* "Eins, zwei, drei, vier, fünf, sechs, sieben, acht, neun, zehn.".

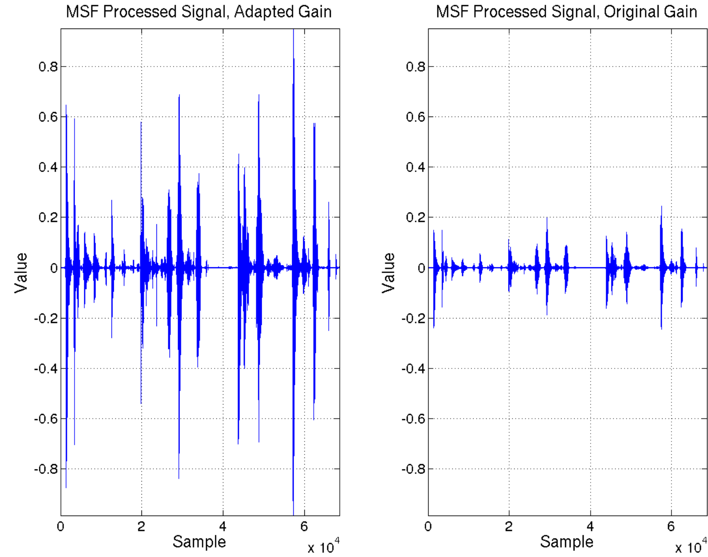


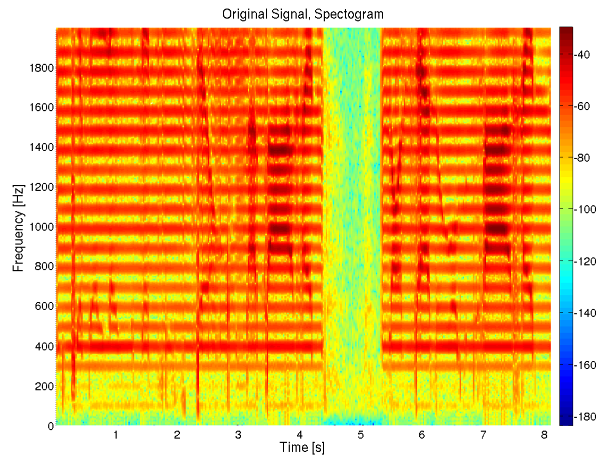
Figure 5.4: *Loudness discrepancy when using the MSF filter: Because the MSF filter does not differ between signal and noise frequency components the (desired) signal's constant components are suppressed as well. This leads to an overall decreasing loudness of the MSF enhanced signal.*

Spectral Subtraction

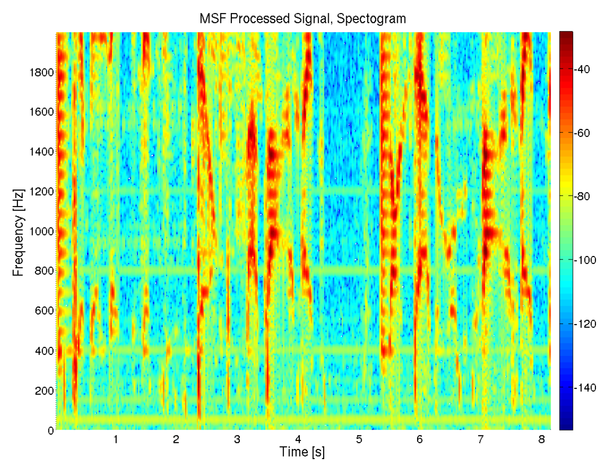
When directly comparing the SS and MSF algorithm's result the SS approach did not produce an as intelligible output signal as it's alternative. This was mainly caused by the existence of clearly perceivable artefacts in form of musical noise. The MSF-based implementation did not show this behaviour. This is why, in terms of intelligibility, the modulation spectral filter enhancement method should be preferred over spectral subtraction even though the MSF method produces an overall quieter signal. This can be avoided using a simple volume control. From the calculational efficiency point of view the spectral subtraction method's run-time turned out to be much higher than the MSF alternative caused by the heavy usage of sorting in order to calculate the percentile. Therefore a trade-off between calculation speed and considered past data samples had to be done. The SS algorithm as described in section 4.4.13 uses a percentile-derived cancellation factor for every spectral pin. With growing amount of memorised data samples for the percentile calculation the input data array length for the sorting procedures increases (Equ. (5.3)). To keep the framework's real-time ability this number had to be cut short even though a reduction leads to a decreasing accuracy of the resulting percentile. The decreasing accuracy again reduces the algorithm's ability to optimally cancel the DREL component.

$$\begin{aligned} O_{insertionsort} &= O_{bubblesort} = O({n_{samples}}^2) \\ n_{sortings} &= n_{percentiles} = n_{freqpins} = \texttt{BLKSZ} \\ O_{sortings} &= n_{sortings} * O_{xxxsort} = \texttt{BLKSZ} * O({n_{samples}}^2) \end{aligned} \tag{5.3}$$

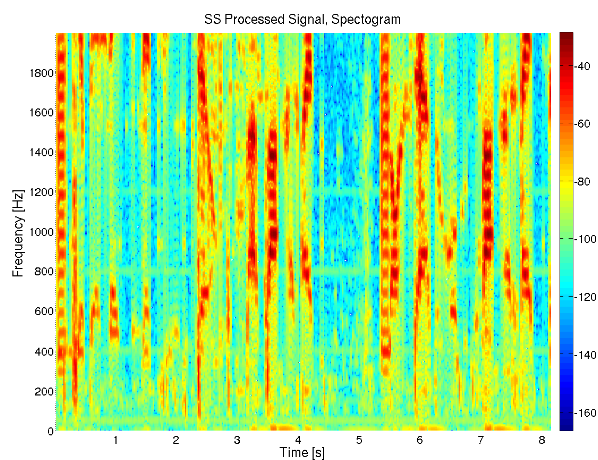
The maximal possible amount of memorised past data samples ($n_{samples}$ in equation (5.3)) turned out to be *eleven*. This is the maximal number of data that can be sorted within the available time period in one frame besides all other mandatory calculations like pre-processing filters, Fourier transform, pitch contour generation, et cetera. Obviously this threshold can be increased when performing the multipath separation only because of the spared time otherwise needed for the pitch contour generation.



(a) *Spectrogram of the original signal which is heavily affected by the DREL component, the horizontal line at about* $110Hz$ *and it's harmonics.*



(b) *Spectrogram of the MSF enhanced signal showing just a weak remaining DREL component.*



(c) *Spectrogram of the SS enhanced signal showing nearly no remaining DREL component.*

Figure 5.5: *Comparison of the implemented MS methods by applying them to the* $8.0s$ *long electrolarynx speech signal* "Ich will ihn nicht umfahren sondern umfahren!".

Direct Comparison

At a glance the performed MS, thus the reduction of DREL noise, was set up to use the MSF approach in it's default configuration, mainly because of the important requirements towards intelligibility and run-time performance. In fact the SS approach needed so much CPU cycles for execution that it could not be used in the pitch contour generation's pre-processing stage - the combined temporal demand of Pitch Contour Generation (F0gen) and SS exceeded the limits for real-time calculation. A stand-alone execution (configured as shown in chapter A.4) is anyway possible within the system's update interval and therefore in real-time. A direct comparison between the results of both implemented MS methods is given in figure 5.5. Especially the better DREL suppression can be observed when comparing figures 5.5b and 5.5c, the constant component at $110Hz$ is nearly not present in the spectral subtraction enhanced result any more contrary to the (weak) remaining component when using the modulation spectral filter.

5.1.3 Pitch Contour Generation Modules

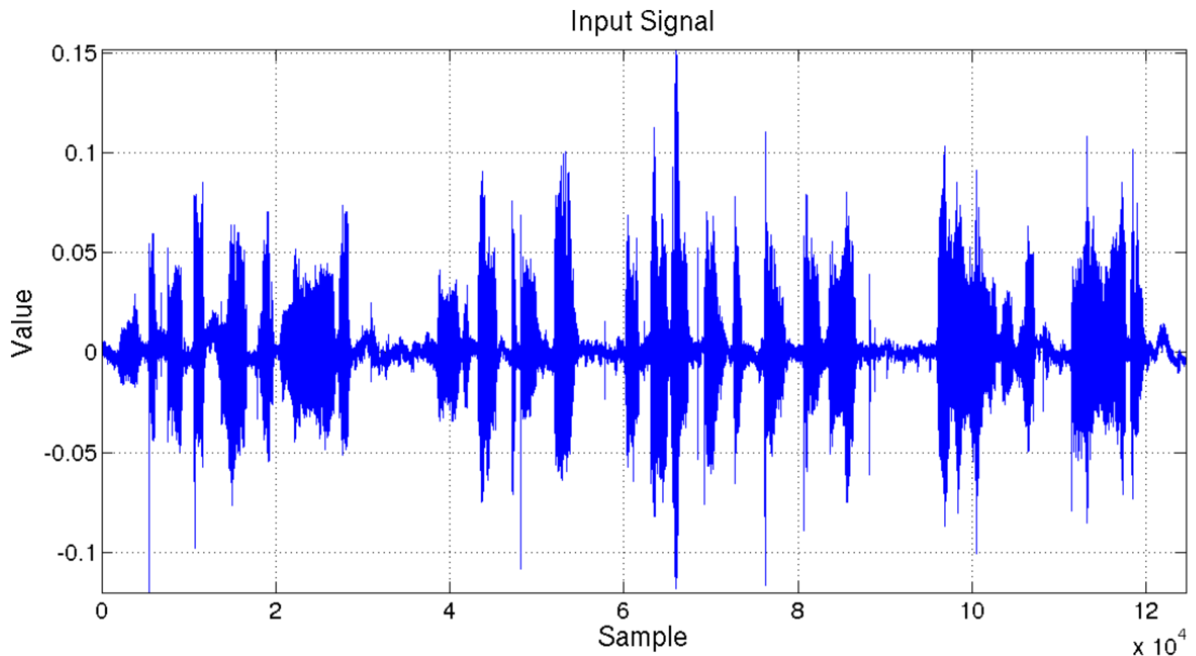
Voice Activity Detection

The implemented **V**oice **A**ctivity **D**etection (VAD)'s working principle is based on a rather simple but robust calculation method. One crucial parameter hereby is the energy threshold. This parameter, as described in section 4.4.6, determines the minimum amount of signal energy a frame has to hold to be marked as `ACTIVE`. Obviously, the actual threshold value is dependent on the environment, especially the following factors that are directly involved in the processing chain. For a better understanding of the refered to components effect on the processing chain and their interaction section 4.1 describes the device's wiring as used by the application during the development process.

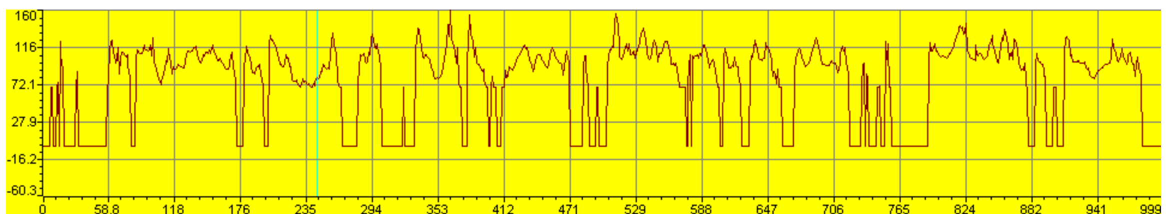
- The mixer's gain setup
- The used microphone's internal amplification
- Specifications of the used **I**nput **/** **O**utput (I/O) interfaces
- Volume control positions when using additional amplifiers

In fact the threshold is already set to the best performing value in the current framework version by default. Anyway this value is just valid if the audio input level of the **A**nalog-To-**D**igital **C**onverter (ADC) is set correctly, thus close to full-scale but without occuring clipping. This is a condition to be aspired to anyway because it reduces the occuring **S**ignal-To-**N**oise **R**atio (SNR) to a minimum. If this condition is met, the optimal threshold is located at a relative level $E_{db} \approx -55dB$ (Equ. (4.11)).

An example of the VAD's performance, using the sentence *"Knapp 40 Jahre nach der Trennung der berühmtesten Band der Welt erscheinen jetzt die Alben."* spoken with an electrolarynx, is given in figure 5.6. `SILENT` regions are hereby indicated by an invalid formant value entry (`-1`). Those blocks are, as desired, exactly located at time intervals where the speaker makes a pause.



(a) Waveform of the 15.5*s long input signal,* `SILENT` *regions are identified as sections of the signal with small amplitude values.*



(b) Formant contour determined by the DSP framework's formant tracker, the regions marked as `SILENT` *are indicated by invalid formant values (*`-1`*). During* `ACTIVE` *regions the tracker produces frequency values matching the found formant value.*

Figure 5.6: *Example for analysing the VAD's performance, indicated by the correlation of* `ACTIVE` *and* `SILENT` *regions between input waveform and DSP mapping, using the electrolarynx speech signal* "Knapp 40 Jahre nach der Trennung der berühmtesten Band der Welt erscheinen jetzt die Alben." *as input.*

When speaking about electrolarynx input signals the processing even gets easier compared to natural voice: The voice `ACTIVE` and `SILENT` regions can be clearly separated by aVAD - the electrolarynx produced, rather loud DREL noise is permanently present when the user speaks and causes high energy in frames containing voice. During periods of silence there is no need for the user to switch on the device - DREL noise is absent in those frames leading to a definite energy discrepancy between `SILENT` and `ACTIVE` frames. This observation might be the reason the VAD detector is performing so well despite of being implemented using a simple analysis algorithm. Nevertheless this behaviour inures to the benefit of the multipath separation and formant tracking algorithms because due to the spared complexity the VAD saves calculation time that can be drawn on by these modules.

Voiced/Unvoiced Detection

When identifying the best-performing spectral centroid value for **V**oiced / **U**n**v**oiced (V/UV) thresholding the used sampling rate must also be taken into consideration in the decision process. The reason is explained easily: The centroid value represents the center of gravity over all existing frequency pins. It is therefore altered by the overall frequency interval - which again is set by the used sampling rate due to Shannon's theorem. So when reducing Sampling Rate (f_s) the overall frequency interval is also reduced with cutting spectral components between $\frac{f_{s,new}}{2}$ and $\frac{f_{s,old}}{2}$. This leads to the centroid being shifted towards smaller frequencies although the analysed signal has never been touched in any way.

This problematic was not covered by the theoretic concept discussed in section 4.4.7. The decision process for picking an appropriate centroid value to be used for implementing this module consisted of two steps:

1. At first implementations using the same V/UV algorithm were studied. The majority of these studies proposed a threshold of about $3kHz$ using sampling frequencies of $12kHz$ to $16kHz$.

2. As second step the found configurations were applied to this framework with adapting the centroid / sampling frequency setting to match the used sampling frequency of $8kHz$. Therefore the centroid threshold of $3kHz$ had to be shifted to a slightly lower frequency in order to adapt to the reduced frequency interval. Tests were performed at first with focusing on the $1.5kHz \ldots 2.5kHz$ area - nevertheless this threshold did not produce good results: Too much frames were marked `UNVOICED`. So the decision value was increased and cross-checked with the resulting data in several test trials. All of these trials were performed in an interval of $2kHz \ldots 3kHz$. In doing so a final V/UV decision threshold of $f_{centroid,new} = 2750Hz$ turned out to perform best. Most likely this value is caused by the sinking energy for higher frequency pins. They do not weight as much as lower frequency components when calculating the centroid, it's value therefore does not decrease as fast as the sampling frequency is reduced.

Formant Tracker

It can be concluded that the formant tracker operated within an acceptable tolerance, meaning that eventually occuring outliners were short and rare enough to not be perceivable in the final pitch contour as such - a comparison between the formant contour calculated by Praat and the DSP framework is printed in figure 5.9. This example analyses a sentence about nine seconds long, spoken with an electrolarynx *"Ich will ihn nicht umfahren sondern umfahren!"* with having accented syllables underlined. Both evolutions of the chosen F_2 formant match, the average frequency is nearly equal. However, there are several areas in the DSP contour with obvious mismatches. This can not be explained by the DSP tracker's working principle, either of them (Praat's implementation and the **L**inear **P**redictive **C**oding (LPC)-based tracker) are implemented using the same algorithm. Therefore the issues have to be traced back to certain implementation details. These discrepancies originate from cutting back calculation detail in the DSP implementation to save calculation time. As discussed in section 4.5, several trade-offs had to be made in order to keep the real-time ability of the system. Reducing the LPC order, interrupting the polynomial root calculation and not implementing a global pathfinder algorithm effects the formant tracking quality.

Run-Time As mentioned, the **I**nverse **F**ilter **C**ontrol (IFC)-based formant tracker was not able to produce it's results in time and therefore violated the real-time criterion. This is why the final version of the application makes use of the LPC-based algorithm. The problematic operation blocks concerning calculational effort are identified when directly comparing the needed CPU cycles for all executed blocks (listed in table 5.3). Figure 5.7 graphically presents the gathered results: Until the second block the IFC tracker is slightly faster because it does only calculate pre-emphasis and band-pass filtering (*block 1*) as well as an IFC filtering with fixed frequency (*block 2*). The LPC tracker on the other hand needs to execute LPC analysis (*block 1*) and polynomial root detection (*block 2*). However after the second block the LPC tracker is almost finished, in *block 3* there is just the missing formant value calculation and insertion into the sorted candidates list to be done. The IFC tracker just starts it's intense operations after *block 2*: *Block 3* performs the mutual formant calculation using IFC filtering, *block 4* is the formant frequency value extraction from the filtered signal. This block is based on a **F**ast **F**ourier **T**ransform (FFT) calculation with following peak detection. These operations are called 16 times: *Block 4* consists of three encapsulated loops, more detailed two main loop cycles, two large loop cycles for both formants F_1 and F_2 and two small loop cycles for reaching convergence. So the overall 16 calls originate from $2_{main} * (2_{largeloop} * 2_{F1,F2}) * 2_{smallloop}$ loop cycles.

This problem did not just occur in the DSP implementation but could also be reconstructed using a Matlab simulation of both algorithms which was analysed using the Matlab-internal profiler. Profiling resulted in an overall calculation time of $0.432s$ (LPC type) compared to $1.391s$ (IFC type) with the LPC tracker even calculating five formants instead of three.

The proposed alternative approach of using a zero-crossing detector instead of a spectral-peak based also did not solve the problem. A straight-forward implementation does not bring very good results, the approach of weighting the zero-crossing results again adds much complexity.

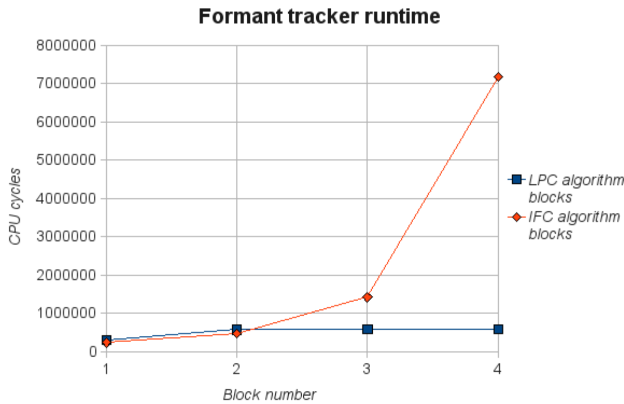
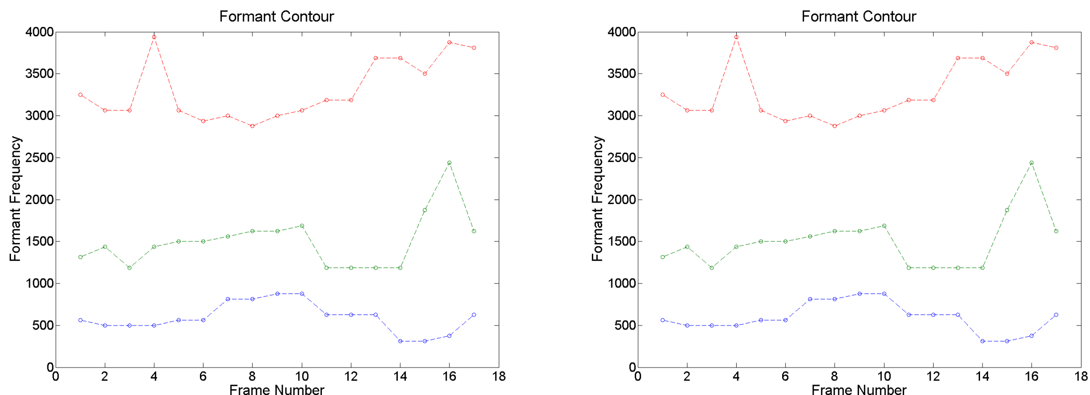


Figure 5.7: *Run-time comparison between LPC and IFC based formant trackers by counting the amount of consumed CPU cycles per block.*

Block	Module
	LPC
1	LPC analysis
2	polynomial root calculation
3	formant value calculation
4	-
	IFC
1	pre-emphasis and BP filter
2	IFC pre-filtering
3	mutual formant calculation
4	formant frequency extraction

Table 5.3: *Sequentially processed blocks of the LPC and IFC formant tracker.*

Detection Quality Directly comparing the tracking results of LPC and IFC tracker is not possible with the DSP implementation because the IFC tracker misses input data frames due to it's run-time problems. Therefore a comparison was performed using a Matlab simulation of both approaches which resulted in the formant contours as shown in figure 5.8. It can be seen that the resulting formant contours more or less match, the IFC tracker just performs slightly worse due to it's run-time trade-off configuration (the formant frequency calculation is done via FFT peak detection only instead of mixing weighted zero-crossing and FFT peaking).



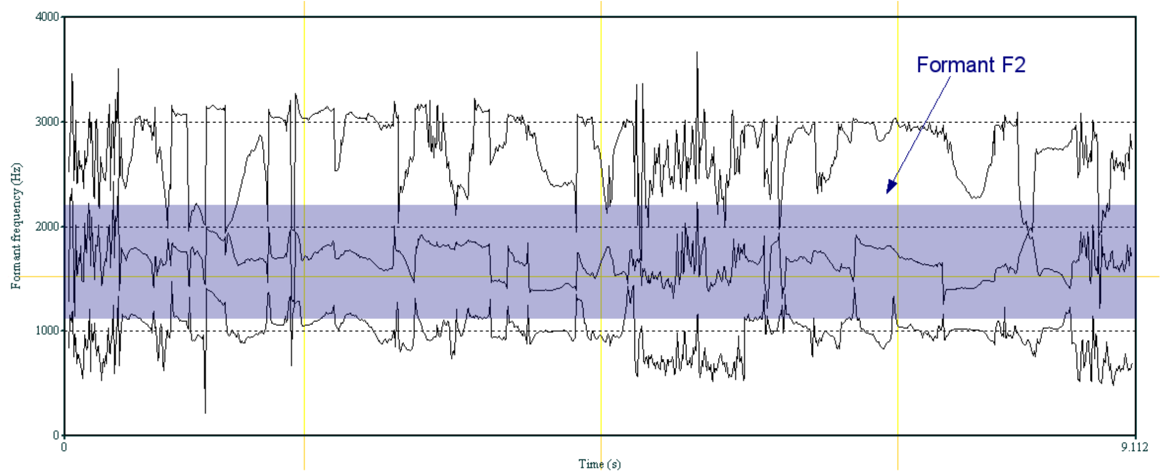
(a) Formant contour of the first 3 *formants as parsed by the IFC-based tracker using a Matlab implementation of the algorithm. It was configured to use a LPC order of* 3 *and a detection threshold of* 0.3*.*

(b) Formant contour of the first 5 *formants as parsed by the LPC-based tracker using a Matlab implementation of the algorithm. It was configured to use a* 2 *iterations for both loops (inner and outer).*

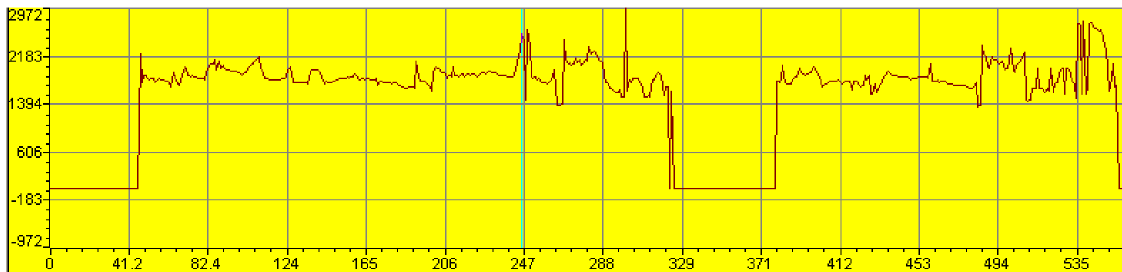
Figure 5.8: *Formant detection quality comparison between both implemented trackers by analysing the word* "Hello"*, spoken by a female person. The detector hereby used a block size of* 128 *and a sampling rate of* $8kHz$*.*

Implementing a global pathfinder algorithm, as mentioned above, would not just add much computational effort to the framework but also lead to a higher input-output delay: The pathfinder's quality depends on the amount of samples surrounding the currently processed to have enough data for an accurate path-finding calculation. In order to have enough samples in both directions, thus past and "future" samples it would be necessary to delay the output process to get the needed formant data from future samples (the term "future" just expresses the future output of already captured frames relatively to the currently outputted) as well.
Concerning the reduced detail of tracker parameters the LPC order is the one having the most influence on the result. Changing this value determines the amount of candidates available for the formant detection. It might even be the case that a too low degree of the LPC polynomial leads to skipped formants. Tests showed that an order of below *nine* tends to do so. A reduction in detail when determining the polynomial roots mainly leads to single outliners. They originate from interrupting the not-yet-finished convergence calculation when approximating the root's value, refer to section 4.5.3 for further information.

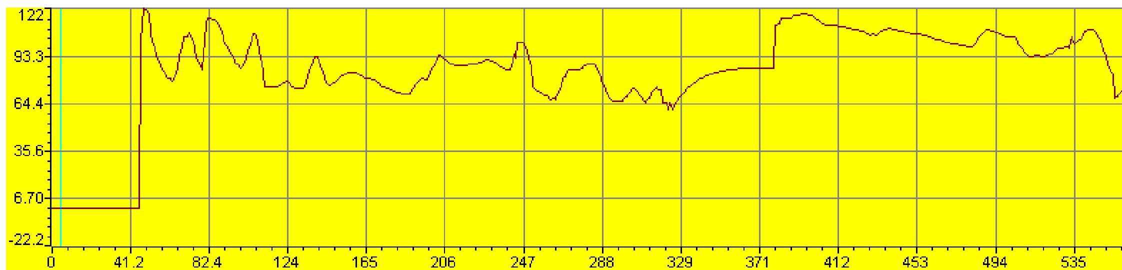
An example of the overall tracking result using the LPC-based formant detector is given in figure 5.9 showing the tracked second formant contour of the sentence *"Ich will ihn nicht umfahren sondern umfahren!"*, spoken with using an electrolarynx. The system's configuration looked like mentioned in table 5.4.



(a) *F_2 formant contour determined by Praat. The chosen formant is highlighted, the voice inactive periods are noticeable by highly fluctuating formant values.*



(b) *F_2 formant contour determined by the DSP framework using the LPC-based tracker with default configuration. Voice inactive periods are noticeable by an invalid formant value (`-1`).*



(c) *F_0 contour determined by the DSP framework. The default pitch is set to $100Hz$, declination is switched on. Again voice inactive periods are noticeable by an invalid marker (`-1`), in this case applied to the pitch's frequency value.*

Figure 5.9: *Formant tracking result using the LPC-based tracker to analyse the $9.1s$ long electro-larynx speech sample* "Ich will ihn nicht umfahren sondern umfahren!"*. Voice inactive periods are located at time instances $0.0s \ldots 0.5s$, $5.0s \ldots 5.5s$ and $9.0s \ldots 9.112s$.*

In the process of analysing the tracking result it could be observed that the contour determined by Praat (Fig. 5.9a, the highlighted second curve is the tracked second formant) matches the DSP-generated result (Fig. 5.9b) leading to the final pitch contour in figure 5.9c. The three regions at about $0.0s \ldots 0.5s$, $5.0s \ldots 5.5s$ and $9.0s \ldots 9.112s$ are `SILENT` regions thus the speaker made a pause during these time intervals. Praat still tries to detect a formant value in

the mentioned regions, this is why the tracking result highly fluctuates there. Nevertheless the DSP's formant tracker correctly detects inactive VAD regions and sets the formant value to an invalid value (`-1`). Despite of the good overall result there is an outliner in the DSP-based tracking result at frame 302 displayed by a single peak of $2972Hz$.

Tracker Setup		**Other Setup**	
Tracked Formant	F_2	**Declination**	Active
LPC Order	10	**Smoothing**	Active
Formant Decision	index-based	**Smoothing Filter Size**	5
Formant Detection Threshold	0.25		

Table 5.4: *Used formant tracker configuration that produced the example result discussed in this section.*

Pitch-Synchronous Overlap-Add

The PSOLA algorithm is responsible for transforming the data stream containing the pitch contour frequency values (meta-information about the signal) into the final pitch contour of the signal itself. When trying to discuss this module's quality there are always questions regarding algorithm correctness and run-time to be answered. Nevertheless there is no further characteristic that could be taken into consideration: Every parameter that effects the final speech signal's quality (pitch contour values, pitch-markers, VAD and V/UV regions) is generated by external modules, PSOLA's quality is therefore directly determined by the modules it gets the input data from without affecting quality itself. However to check the correct implementation of the PSOLA algorithm a cross-checking between the array containing the calculated pitch contour values and the actual contour of the output signal can be performed.

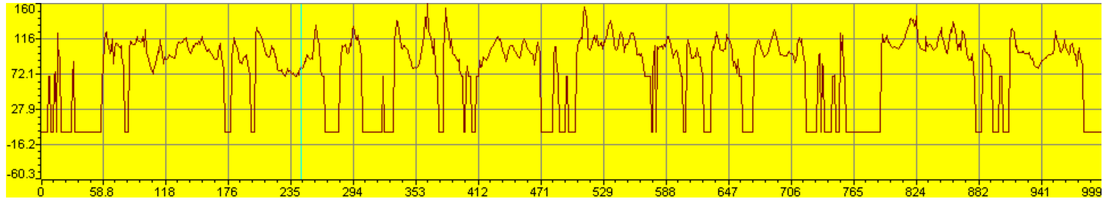
If for example the resulting pitch contour as calculated by the formant tracker, smoothing algorithm and formant-to-pitch contour conversion looks like

```
[115.0 113.5 102.3 95.4 99.8]
```

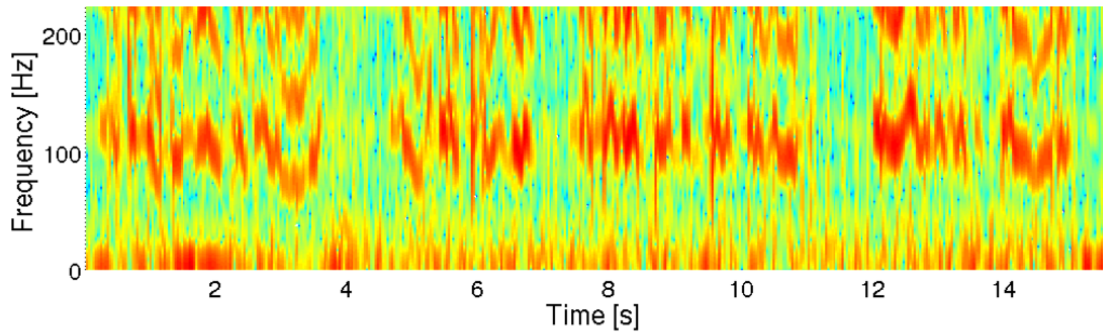
the outputted speech signal must show a pitch value of

- 115.0 for the signal in the 5_{th}-last frame,
- 113.5 for the signal in the 4_{th}-last frame,
- ...

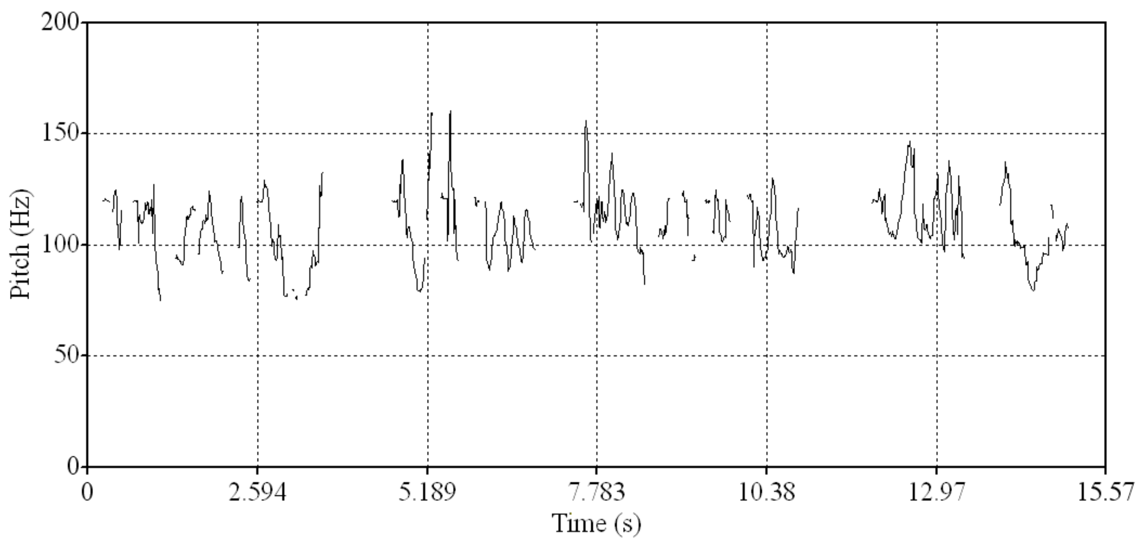
In order to perform the mentioned cross-checking the sentence *"Knapp 40 Jahre nach der Trennung der berühmtesten Band der Welt erscheinen jetzt die Alben."*, spoken with an electrolarynx, was analysed. The gathered pitch contour array as calculated by the DSP is shown in figure 5.10a, the captured signal with accordingly altered pitch contour is displayed in figures 5.10b and 5.10c whereas the first one shows the signal's spectrum and the second the pitch contour as parsed by Praat. Comparing those three pictures the match between to-be-applied pitch values and resulting pitch-altered signal can be clearly seen.



(a) *Pitch contour determined by the DSP framework through tracking F_2 and converting the gathered developing to an adequate pitch contour.*



(b) *Resulting spectrum of the captured output signal. The audio data was analysed using Matlab, the captured signal's pitch contour matches the one calculated by the DSP.*



(c) *Pitch contour determined by Praat. The captured audio data was analysed using Praat to determine the signal's pitch, the contour again matches the one calculated by the DSP.*

Figure 5.10: *Example result of the PSOLA module: With cross-checking the array holding the pitch contour values as calculated by the framework (upper picture) and the actual pitch contour of the output signal (lower picture) the functionality of the PSOLA algorithm can be determined - in a correct implementation the formant-derived pitch contour frequency values and the appropriate output signal's pitch match.*

5.2 Results Comparison with Different Setups

5.2.1 Block Size and Fast Fourier Transform Point Size

Varying the **Bl**ock **Siz**e (BLKSZ) directly affects the time resolution of the overall system - the smaller the block size the finer the resolution in the time domain and vice versa. The same is valid for the resolution in the frequency domain, just the opposite way due to the $f = \frac{1}{T}$ relation. It is therefore necessary to find an appropriate trade-off for the used block size. Usually when considering the dynamic characteristics of the human voice a value around 160 to 320 points seems to be a good choice because the spectral resolution is of less importance than the temporal (see also section 4.5.3). This quite low value originates from the high temporal dynamics of voice due to articulatory movements. Common telephony systems just use a frequency range of $300Hz$ up to $3400Hz$ to transmit vocal information. In DSP applications there is also the computational cost to be kept in mind. Data processing has to be finished in real-time to provide smooth and more importantly correct output. Further information about this topic is found in chapter 4.5.3.

Several test runs resulted in an optimal BLKSZ of 256 for this implementation. Decreasing this value still produces good results down to 192 points. Below this value it can not be guaranteed to calculate the data in time thus until the next data frame reaches (Equ. (4.19)). For higher block sizes the framework still produced good results unit passing 384 samples per frame. With even longer frames the quality again sinks due to a combination of memory management issues as well as a too high amount of input data to be processed at once - the proportion between BLKSZ and amount of accruing data is not completely linear (because of nested loops and similar). The optimal BLKSZ value of 256 is a combination of quality aspects based on the above mentioned facts. So the block size has to be chosen to fit the dynamic behaviour of the input signal, the human voice, with still being able to be calculated in real-time. If not treated correctly the framework can not provide the processing results in time and the resulting output sounds either noisy - the output contains invalid data values because the correct data values have not been written to the output memory punctually - or choppy. This effect occurs if the processor is stuck in a calculation procedure and therefore fails to raise the output interrupt.

5.2.2 Smoothing Filter Size

Smoothing filters are used in the framework to avoid extreme pitch fluctuations in the pitch contour generation module by smoothing the formant developing which is again used as source for the pitch contour generation. So varying the smoothing filter size just affects this module, the multipath separation module is independent from this parameter.

The chosen value is mainly determined by two factors:

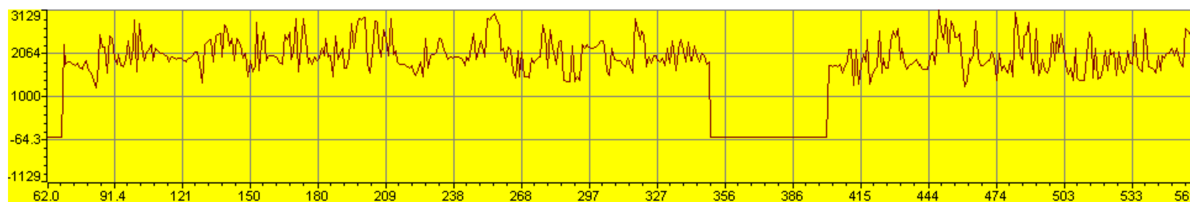
1. Needed calculation time

   Again the computational cost has to be considered when choosing a suitable value for this parameter: The higher the filter size the more data has to be handled by the processor. Appropriate smoothing filter sizes range from 3 (the minimum size) up to about 11. 3 is the minimal value because the filter size obviously has to be larger than one and odd, the upper limit of 11 is determined by the framework's real-time ability.

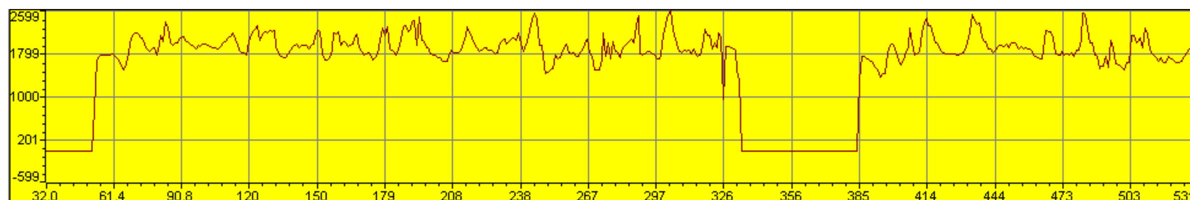
It might be possible to additionally increase the smoothing filter size further without violating the run-time boundaries when cutting back on other calculations. However this is not recommended because on the one hand a too high smoothing filter size again reduces the speech quality and on the other hand LPC approximation, formant tracking and similar are of more importance to the quality.

2. Block size

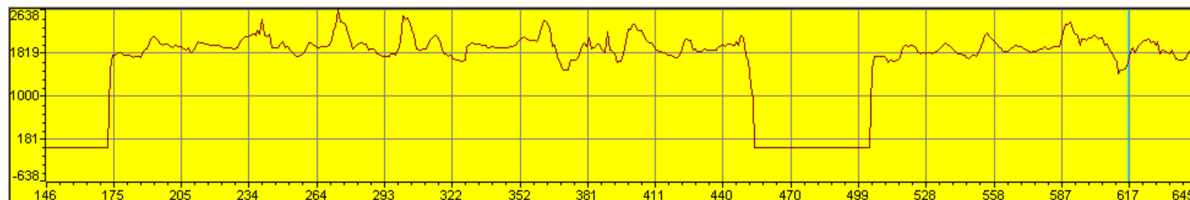
   As mentioned in chapter 4.5.3 the block size directly influences the temporal resolution of the signal. Therefore the smoothing has an even higher effect on the signal when dealing with large block sizes because the samples are already spaced by a larger interval before being passed to the smoothing filter.



(a) Formant contour without smoothing, hence filter size 0.



(b) Formant contour with smoothing, using a filter size of 5.



(c) Formant contour with smoothing, using a filter size of 9.

Figure 5.11: *Smoothing performance comparison with different filter sizes on an example formant 2 developing as tracked by the DSP's formant detector. The used algorithms were outliner cancellation, median filtering and linear smoothing filtering, performed consecutively in this order. As input signal the sentence* "Ich will ihn nicht umfahren sondern umfahren!" *spoken by an elecrolarynx user was chosen.*

Considering these facts led to the current implementation using a filter size of *five*. This value smooths intervals of length $32ms * 5 = 160ms$ if using the default configuration with with BLKSZ=256 and f_s=$8kHz$, see table 4.5. A value of $160ms$ is a good trade-off between cancelling extreme fluctuations in the pitch contour and still having a high dynamic that is perceived by the listener. Smoothing over longer intervals should just be performed when using high sampling rates or small block sizes because they lead to short frame durations. This guarantees the retaining speech dynamics.

In figure 5.11 a comparison of the formant contour is shown when using various smoothing filter sizes. Figure 5.11a uses no smoothing at all whereas in figure 5.11b a filter with size *five* and in figure 5.11c one with size *nine* for both the median and linear smoothing filter is applied.

5.2.3 Filter Type in the Multipath Separation Module

Again the primary factor determining the choice of the filter used by the multipath separation module is the time factor. Due to **I**nfinite **I**mpulse **R**esponse (IIR) filters working in a recursive way they are able to achieve the same filtering effect as **F**inite **I**mpulse **R**esponse (FIR) filters with a significantly less amount of factors. Most filtering in this framework was done using filter orders of either *one* or *three*, resulting in a rather small amount of IIR coefficients and therefore multiplication and addition operations necessary. This is why IIR filters perform better in terms of calculation effectiveness than FIR filters.

The MS module made use of *first* order Butterworth filters operating at a cut-off frequency $f_c = 1Hz$ (to only suppress constant - thus $0Hz$ - components). This leads to a coefficient size of *eight* for the IIR filter (results from equation (5.4), see also figure 4.5). The adequate FIR filter was calculated using the Matlab `impz()` function (Ch. A.1.1) with an adaptive size parameter to calculate it's coefficients. The FIR's coefficients for this filter type show the behaviour of converging towards 0.0 with increasing index. The size parameter was therefore chosen in a way that the last calculated filter coefficient c_N - the one with the highest index N - had to be the first with value 0.0. Doing so a loss of detail when applying the filter can be avoided because no non-zero coefficients are skipped. At the same time the filter size N is kept to a minimum. Obviously c_N depends on the amount of decimals displayed by Matlab when calculating the coefficient's values. In "high resolution" format this value is set to 15. This format is activated using the Matlab command `format long`. Dependent on the sampling rate this leads to a coefficient size of ≈ 400 in average. Increasing the sampling rate leads to even bigger sizes due to the need of providing one coefficient per sampling value with an increasing amount of sampling values.

$$2 * n_{biquad} \leftarrow n_{biquad} = 2 * n_{numerator} + 2 * n_{denominator} \tag{5.4}$$

Comparing these values - *eight* for IIR filters and ≈ 400 for FIR filters - the significant advantage when using IIR filters to save calculation time can easily be imagined. Several trials resulted in a maximal FIR filter size of less than 100 to still be able to provide the real-time ability of the framework - a value much too small for still producing acceptable filtering results.

Chapter 6

Conclusion and Outlook

In sum the overall speech enhancement framework produced a distinctively less noise corrupted, natural sounding voice. The **M**ultipath **S**eparation (MS) produces the same results as the non-real-time implementation showing an effective suppression of the electrolarynx signal's **D**irect-**R**adiated **E**lectro**l**arynx Noise (DREL) component. Whether the resulting speech causes a higher acceptance among electrolarynx users as well as their conversation partners or not is a matter of personal taste: the DREL suppression is achieved at the price of higher artefacts in the signal - either one appreciates the multipath separated result or is disturbed by these artefacts. The Pitch Contour Generation (F0gen) on the other hand does not have this kind of trade-off. The only question here is if the formant tracker performs well enough to lead to a natural sounding pitch contour. This is again determined by the accuracy of the tracker which is just limited by it's calculational effort. As shown in chapter 5, this is the case with the tracker based on the **L**inear **P**redictive **C**oding (LPC) implementation.

The biggest problem during the whole implementation process was to meet up with the run-time restrictions for the sake of producing the results in real-time with still keeping a high level of accuracy. There certainly were several trade-offs to be complied with, realising faster performing algorithms could definitely increase the speech enhancement quality further. An approach in this direction would be the assembly-level optimisation of frequently called calculation routines. This has not been not done at the current level of implementation simply because this approach is very time intense and with the framework performing rather well anyway there was no immediate need to do so. Anyway an approach optimally utilising the **D**igital **S**ignal **P**rocessor (DSP) could lead to a speed-up of (the more or less unlikely maximal value) of *eight* times at best (Sect. 2.3.2) - the hereby gathered time reserve could be used to noticeably increase the enhancement framework's detail and accuracy. Besides the obvious raise of parameters immediately enhancing the calculation quality (LPC order, used **F**inite **I**mpulse **R**esponse (FIR) filter lengths, ...) the additional buffer in time also offers the possibility to furthermore experiment with the complete framework setup. The tight time flow at the moment nearly makes it impossible to tune parameters like block size and sampling rate, they have to be kept to a minimum or at least in an area close to it. An additional buffer could be used to increase the system's temporal resolution with the result of achieving a more dynamic/adaptive formant and consequential pitch contour.

Besides enhancing the framework from the implementation side there is also the possibility to approach from the algorithm side. The **M**odulation **S**pectral **F**ilter (MSF) implementation currently lacks in keeping the original signal's loudness as discussed in section 5.1.2. Adapting

the algorithm to get rid of this drawback would make the - currently mandatory - usage of a volume control needless. This would increase the achievable **S**ignal-To-**N**oise **R**atio (SNR) because with gaining the overall output the noise components are also amplified, having a better fitted filter would keep the SNR low with suppressing noise and keeping the signal untouched. A completely different approach could also be to implement some sort of pattern matching between electrolarynx and healthy voice to reconstruct a pitch contour. This would replace the currently active formant tracker and might lead to promising results as well: Machine learning algorithms can take much more speech features into consideration to evaluate a pitch contour than simply the formant. In order to not violate the existing real-time constraints an algorithm would have to be chosen where the calculational expensive training can be done offline. The actual mapping, the **M**achine **L**earning (ML)'s testing procedure, shows a complexity that is distinctively lower than training, with having access to an already pre-configured neural net it might again be executed on the DSP application in real-time.

Determining the achieved intelligibility enhancement via subjective tests is a task that was already planned to be done in the course of this work. Unfortunately time constraints prevented me from doing so. Getting people's subjective impression towards the resulting speech signal can not be replaced by purely objective, technical measurement because every person reacts unique to the perceived sound. And after all achieving the best possible intelligibility, regardless of measurement method or listener, allows to finally come closer to the real purpose of this work: Supporting electrolarynx dependent speakers the best possible way with providing tools that help them to have smooth and flawless conversations...

Bibliography

[1] Agastya Chitra, Mechanic Dan, Kothari Neha. Factors Affecting Call Quality in Popular VoIP Clients. Technical report, Columbia University, 2008. 68

[2] Agilent Technologies, Colorado Springs. *Oszilloskope Agilent 54621A/22A/24A und Mixed-Signal-Oszilloscope Agilent 54621D/22D Benutzerhandbuch*, 2000. 109

[3] AKG Acoustics, Vienna. *HSC 171/271, HSD 171/271/Single User Instructions*. 108

[4] APart. *APart PA4060 Amplifier*. 109

[5] Behringer, Willich-Münchheide II. *Eurorack MX602A Bedienungsanleitung*, 1.2 edition, 2001. 108

[6] Benesty Jacob, Sondhi Mohan M., Huang Yiteng. *Handbook of Speech Processing*. Springer, Heidelberg, 2008. 2, 6

[7] Bőehm Tamás, Németh Géza. Algorithm for Formant Tracking, Modification and Synthesis. In *Híradástechnika Journal*, volume 62, pages 15 – 20, Budapest, 2007. Scientific Association for Infocommunications. 15

[8] Boersma Paul. Accurate Short-Term Analysis of the Fundamental Frequency and the Harmonics-to-Noise Ratio of Sampled Sound. In *IFA Proceedings 17*, pages 97 – 110, Amsterdam, 1993. University of Amsterdam. 41

[9] Boll Steven. Suppression of Acoustic Noise in Speech Using Spectral Subtraction. In *IEEE Transactions on Speech and Audio Processing*, volume 27, pages 113 – 120, Salt Lake City, 1979. University of Utah. 26

[10] Boutremans Catherine. *Delay Aspects in Internet Telephony*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, 2009. 67

[11] Brüel & Kjær, Nærum. *Schwingerreger Typ 4810 Technische Dokumentation*, 1990. 108

[12] Chassaing Rulph. *Digital Signal Processing and Applications with the C6713 and C6416 DSK*. John Wiley & Sons, New Jersey, 2005. 17

[13] Laryngeal Cancer, October 2009. http://www.commonwealthent.com/Laryngeal_Voice_Box_Cancer.htm. 9

[14] Denes Peter, Pinson Elliot. *The Speech Chain: The Physics and Biology of Spoken Language*. Worth Publishers, 2. edition, 1993. 6

[15] Ellenberger Kenneth W. Algorithm 30: numerical solution of the polynomial equation. *Commun. ACM*, 3(12):643, 1960. 102

[16] Feldbauer Christian. Real-Time Block Processing Environment. Technical report, Graz University of Technology, Graz, 2005. 30, 67, 72, 100

[17] Geelhoed Erik, Parker Aaron, Williams Damien J., Groen Martin. Speech Processing: Theory of LPC Analysis and Synthesis. Technical report, Hewlett-Packard, 2009. 67

[18] Friedrich Gerhard. Kehlkopfkrebs (Larynxkarzinom), October 2009. http://www.netdoktor.at/krankheiten/fakta/kehlkopfkrebs.htm. 9

[19] Kehlkopfkrebs, October 2009. http://gin.uibk.ac.at/thema/krebs/kehlkopfkrebs.html. 9

[20] Hagmüller Martin. *Speech Enhancement for Disordered and Substitution Voices*. PhD thesis, Graz University of Technology, Graz, 2009. 1, 25, 26, 53

[21] Hewlett-Packard, Loveland. *Agilent 33120A Function Generator / Arbitrary Waveform Generator User's Guide*, 6. edition, 2000. 109

[22] Jochum Christian, Reiner Peter. Comparison of Excitation Signals for an Electronic Larynx. Master's thesis, Graz University of Technology, Graz, 2008. 55

[23] Jones Douglas L.,Swaroop Appadwedula, Berry Matthew, Haun Mark, Janovetz Jake, Kramer Michael, Moussa Dima, Sachs Daniel, Wade Brian. Effects of Latency on Telepresence. Technical report, Connexions Project, 2004. 39

[24] Lemmetty Sami. Review of Speech Synthesis Technology. Technical report, Helsinki University of Technology, Helsinki, 1999. 13

[25] Liu Hanjun, Ng Manwa L. Electrolarynx in Voice Rehabilitation. In *Aurus Nasus Larynx*, volume 34, pages 327 – 332, Evanston, Hong Kong, 2007. Elsevier. 1

[26] Loizou Philipos C. *Speech Enhancement: Theory and Practice*. CRC Press, Dallas, 1. edition, 2007. 6, 25

[27] Mills Wesley. *Voice Production in Singing and Speaking*. BiblioBazaar, 4. edition, 2008. 7

[28] Nishiguchi Masayuki, Matsumoto Jun, Shinobu Ono. Voiced/Unvoiced Decision Based on Frequency Band Ratio. Technical report, Sony Corporation, Tokyo, 1997. 40

[29] Oppenheim Alan V., Schafer Ronald W., Buck John R. *Discrete-time Signal Processing*. Prentice Hall, New Jersey, 2. edition, 1999. 11, 12, 35, 36, 38, 99

[30] Parkin Max D., Bray Freddie, Ferlay J., Pisani Paola. Global cancer statistics, 2002. In *A Cancer Journal for Clinicians*, volume 55, pages 73 – 108, Atlanta, 2005. American Cancer Society. 9

[31] Boersma Paul. Sound: To Pitch (ac), October 2009. http://www.fon.hum.uva.nl/praat/manual/Sound__To_Pitch__ac____.html. 41

[32] Realtec Semiconductors, Hsinchu. *ALC260/ALC260D Series 2 Channel High Definition Audio Codec Datasheet*, 1.4 edition, 2005. 109

[33] Servona, Troisdorf. *Servox Digital Instruction Manual*. 108

[34] Skandera Paul, Burleigh Peter. *A Manual of English Phonetics and Phonology*. Narr, Tübingen, 1. edition, 2005. 7

[35] Sorensen Henrik V., Jones Douglas L., Heideman Michael T., Burrus Sidney C. Real-Valued Fast Fourier Transform Algorithms. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, volume 35, pages 849 – 863, Houston, 1987. Rice University. 35, 36

[36] Spectrum Digital, Stafford. *TMS320C6713 DSK Technical Reference*, b. edition, 2004. 108

[37] Texas Instruments, Dallas. *TLV320AIC23 Stereo Audio Codec Data Manual*, 2001. 18

[38] Texas Instruments, Houston. *TMS320C6000 DSK Board Support Library API User's Guide*, 2001. 21

[39] Texas Instruments, Houston. *TMS320C6000 Chip Support Library API User's Guide*, 2004. 21

[40] Texas Instruments, Houston. *TMS320C6713B Floating Point Digital Signal Processor*, b. edition, 2006. 17, 60

[41] Texas Instruments, Dallas. *TMS320C67x DSP Library Programmer's Reference Guide*, b. edition, 2006. 21, 100

[42] Texas Instruments, Houston. *Code Composer Studio Development Tools Getting Started Guide*, 3.3 edition, 2008. 20

[43] Texas Instruments, Houston. *TMS320C6000 Assembly Language Tools User's Guide*, 6.1 edition, 2008. 33

[44] Texas Instruments, Houston. *TMS320C6000 Optimizing Compiler User's Guide*, 6.1 edition, 2008. 17, 19, 59

[45] Ueda Yuichi, Hamakawa Tomoya, Sakata Tadashi, Hario Syota, Watanabe Akira. A Real-Time Formant Tracker Based on the Inverse Filter Control Method. In *Acoustic Society of Japan*, volume 28, pages 271 – 274, Kumamoto, 2007. Kumamoto University. 16, 44, 48

[46] Wah Benjamin W., Sat Batu. The design of voip systems with high perceptual conversational quality. In *Journal of Multimedia*, volume 4, pages 49 – 62, Illinois, 2009. University of Illinois. 67

[47] Watanabe Akira. Formant Estimation Method using Inverse-Filter Control. In *IEEE Transactions on Speech and Audio Processing*, volume 9, pages 317 – 326, Kumamoto, 2001. Kumamoto University. 16

[48] Werneck Renato, Ribeiro Celso. Sorting Methods For Small Arrays. Technical report, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2000. 64, 65

[49] Formant, September 2009. http://en.wikipedia.org/wiki/Formant. 46

Appendix A

Appendix

A.1 Matlab Code

The main usage of the Matlab software package during this project was to generate the coefficients needed in filter and window design. Due to Matlab's very good signal processing support it was also used to test various routines and algorithms before implementing in the DSP framework as well as to verify results produced by the real-time environment. Nevertheless these test implementations had no direct influence on the actual speech enhancement application and will not be discussed in this thesis therefore.

A.1.1 Filter Design

As discussed in chapter 4.4 the signal processing framework uses several FIR and **I**nfinite **I**mpulse **R**esponse (IIR) filters during it's calculations. Every of those filters was a *Butterworth* filter either of low- or high-pass type with an order between *one* (the MSF envelope **H**igh-**P**ass (HP) filter) and *three* (e.g. the pre- and post-processing HP filters in the multipath separation and F0gen routines). Filters of higher order were firstly not necessary and would secondly lead to higher calculation times because of the increasing amount of coefficients with every degree of order.

Infinite Impulse Response Filter Design

The IIR filter coefficients were initially calculated with the `butter()` function as shown in equation (A.1). This function calculates the numerator a and denominator b values for a **D**irect **F**orm **2** (DF2) filter in array form dependent on the parameters order p, normalised center frequency $f_{c,n}$ and filter type T which can either be `high` for a high-pass or `low` for a low-pass.

$$\texttt{[b,a] = butter(p,}f_{c,n}\texttt{,T)} \tag{A.1}$$

The normalised frequency $f_{c,n}$ is calculated with dividing the desired frequency f_c by the sampling rate f_s as shown in equation (A.2).

$$f_{c,n} = \frac{f_c}{\frac{f_s}{2}} \tag{A.2}$$

The calculated numerator and denominator values can not be used in the source code directly because IIR filter calculations are performed on biquad modules with the run-time optimised Assembler function `biquad()` (Equ. (4.9)) and therefore have to be converted to this filter form. Matlab provides the `tf2sos()` function for these purposes. It is called with the numerator a and denominator b as parameters and returns the biquad coefficients c and gain g for every biquad stage (Equ. (A.3)).

```
[c,g] = tf2sos(b,a)
```
(A.3)

The remaining processing stage before being able to use these values in the source code is a conversion of the parameter format. The coefficients returned by `tf2sos()` are in a mathematically correct representation (having the correct sign and the first denumerator a_0 is 1.0, see table A.1) but are needed to be in a form like shown in table A.2 to save calculation speed with already negating the denominator values (they are the factor in the subtraction term) and skipping the redundant a_0 value. The single stage gain values are also converted to an overall system gain by simply multiplying them - this saves calculation time because the filter calculation just needs a single multiplication for applying the gain instead of N.

```
[ b01   b11   b21   1   a11    a21
  b02   b12   b22   1   a12    a22
  ...
  b0N   b1N   b2N   1   a1N   a2N ]
```

Table A.1: *Structure of the IIR coefficients array as returned by Matlab's* `tf2sos()` *function.*

```
[ -a11   -a21   b11    b21
  -a12   -a22   b12    b22
  ...
  -a1N   -a2N   b1N   b2N ]
```

Table A.2: *Structure of the IIR coefficients array as needed by the DSPLib function* `biquad()`*.*

Finite Impulse Response Filter Design

FIR filter coefficients are calculated in a similar way as the IIR parameters with calculating numerator a using equations (A.1) and (A.2). The denominator b can be set to 1.0 because FIR filters do not use recursions per definition. The FIR filter factors for every delay line $x[t-i]$ as shown in equation 4.7 are determined using the numerator b and filter length N as input for the Matlab `impz()` function which returns these filter factors c in array form when calling the function as shown in equation (A.4).

```
[c] = impz(b,a,N)
```
(A.4)

Inverse Microphone Filter Besides realising common high- and low-passes an **I**nverse **F**ilter (IF) had to be designed to counterfeit the distinctive near-field behaviour of the used microphone as shown in figure A.1. The specification was to dampen low frequencies between $0Hz$ and $350Hz$ with $10dB$, between $350Hz$ and $1kHz$ the suppression should slowly converge towards $0dB$.

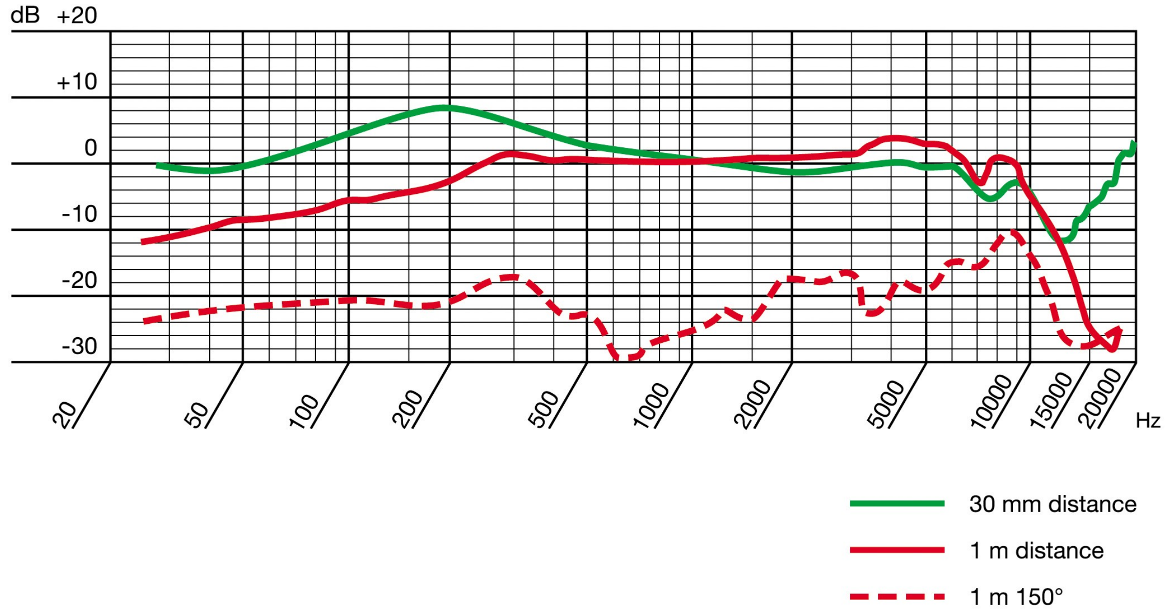


Figure A.1: *Frequency response of the used AKG HD171 microphone for different speaker distances. The green curve indicates the near-field behaviour, the red curve the far-field behaviour and the dashed red one the far-field behaviour using a non-direct speaker angle (source: AKG).*

In order to design the inverse filter Matlab's **F**ilter **D**esign and **A**nalysis **Tool** (FDATool) was used. Due to the specific behaviour within given bandwidths the *multi-band* filter type was chosen. With adapting the wanted parameters and cross-checking them with the resulting frequency response the *least-squares* approach for coefficient calculation performed best.

Nevertheless the results were not optimal due to filter order limitations caused by runtime constraints: An increasing order might lead to an accurate frequency response but causes more coefficients and therefore a higher calculation time - especially when designing for higher sampling rates ($32kHz$ and above). A trade-off between wanted attenuation in the low frequency region and unwanted in higher frequency regions, mainly between $1kHz$ and $2kHz$, had to be performed.

The discussed approach finally resulted in the parameter values presented in table A.3 leading to the FIR's frequency responses as shown in figure A.2. The mentioned *weight* parameter is a ratio indicating the importance of accuracy for the bands among each other.

Parameter	Desired Tendency	Used Value
frequency bands	accurate	0Hz ... 300Hz 500Hz ... $\frac{f_s}{2}$Hz
filter order	minimal	6 (f_s= $8kHz$) 8 (f_s= $16kHz$) 10 (f_s= $32kHz$) 12 (f_s= $44.1kHz$) 14 (f_s= $48kHz$)
weight	optimal	8:1 (f_s= $8kHz$) 4:1 (f_s= $16kHz$) 6:1 (f_s= $32kHz$) 8:1 (f_s= $44.1kHz$) 7:1 (f_s= $48kHz$)

Table A.3: *Optimal parameters of the multi-band FIR filter as used in Matlab's FDATool in order to design the inverse microphone filter for different sampling rates.*

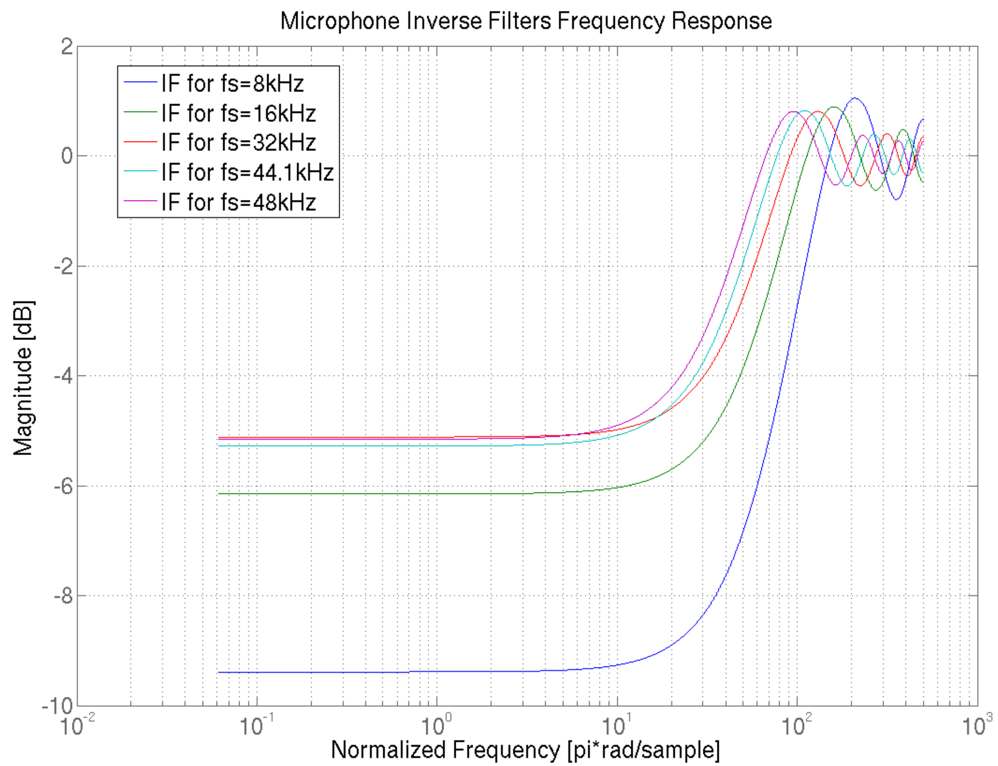


Figure A.2: *Final frequency responses of the designed inverse microphone filters dependent on the target sampling rate. The optimal filter shall dampen the frequency band* $0Hz \ldots 300Hz$ *by* $10dB$ *with leaving other bands untouched.*

A.1.2 Window Design

Windowing is a necessity when performing short-time operations on continuous input data for the sake of avoiding artefacts on frame borders in the overlap-adding process. This is done via weighting the current frame data and overlap-adding it with parts of the previous. The choice of a weighting function plays an important role in this process - it determines the importance of the specific overlapping data values in the frame that holds them. When equipping those samples with strictly linear decreasing weighting factors in frame border direction, the most simple window form, this is called a triangular window.

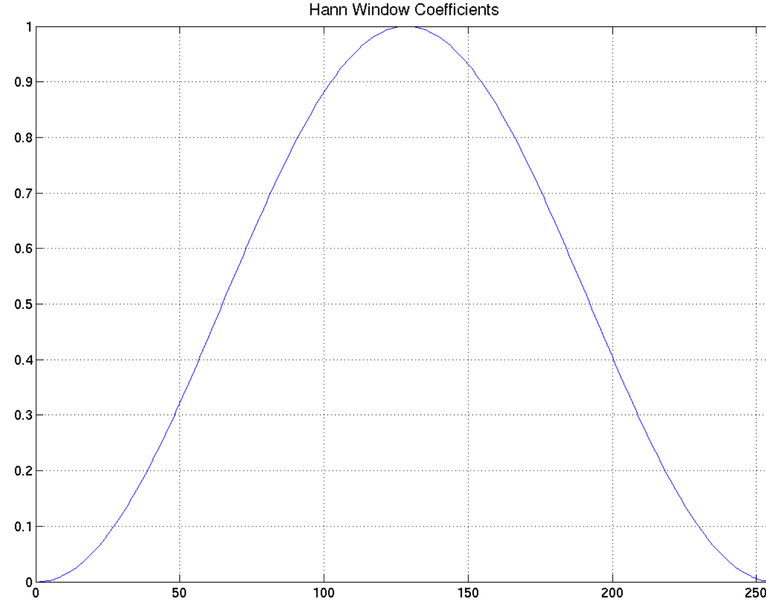


Figure A.3: *Hann window coefficients for a window size of* 256.

This framework primarily used the so-called *Hann windows* which are created using equation (A.5) to calculate the single coefficients. Variables in this equation are the coefficient index i indicating the coefficient's position in the block and the window length N which is determined by the **Block Size** (BLKSZ). The Hann coefficients for BLKSZ= 256 are shown in figure A.3, further information about this type of window as well as alternatives can be found in [29].

$$h[i] = 0.5 * (1.0 - cos(i * \frac{2 * \pi}{N - 1})) \tag{A.5}$$

Matlab provides several built-in functions to perform window coefficient calculation. The mentioned Hann window coefficients are calculated using the `hann()` function, called like demonstrated in equation (A.6). Function parameter is the window size N. The returned value h is an array of size N consisting of the calculated coefficients at their appropriate frame positions.

```
[h] = hann(N)                                    (A.6)
```

A.2 External Code

A.2.1 Block Processing Framework

Data exchange between audio **I**nput **/ O**utput (I/O) device and signal processing unit is based on block processing or framing as explained in section 4.2 in detail. The skeletal structure providing this functionality was originally implemented by a colleague at the laboratory for being used in his classes ([16]).

A.2.2 KISS Fast Fourier Transform

As discussed in section 4.4.1 and more detailed in [41] the **F**ast **F**ourier **T**ransform (FFT) functions provided by the **D**igital **S**ignal **P**rocessor **Lib**rary (DSPLib) are limited to a definite group of input parameters for the sake of being able to optimise their calculation time in the best possible way. Especially the restriction to always have to have input data lengths being a power of two (primarily 64, 128, 256, 512 and 1024), necessary for the radix-2 character of the processing algorithm, was problematic. During testing the output quality several trials were done with different frequency resolutions in order to find an optimal value. Unfortunately the input length restriction caused a very small pool of allowed testing resolutions - FFT calculations below $64pt$ resulted in a way too bad quality, resolutions above $256pt$ were problematic due to increasing calculation times for a single frame which made real-time processing impossible.

Therefore the *KISS FFT* framework was used for Fourier transforms with non-power-of-two frequency resolutions like $160pt$ and $192pt$ as frequently used in comparable applications. It is important to mention here that a complete migration to the KISS FFT framework is not advisable. Therefore the power-of-two Fourier transforms are still calculated with the DSPLib provided function because of their calculation speed optimisation for the used signal processor.

Usage

The Kiss FFT framework was designed to be simple and without unnecessary overhead - as hinted by the framework's name: **KISS** stands for **K**eep **I**t **S**imple, **S**tupid. Basically it can be differed between using the routines handling purely real input data (processed in routines labelled with `fftr` at the beginning) and complex input (routines whose name start with `fft`). For further information please refer to http://sourceforge.net/projects/kissfft/.

Copyright

The KISS FFT source code can be used freely when including the copyright notice written by the author as mentioned in the code listing below.

```
/*
Copyright (c) 2003-2004, Mark Borgerding

All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list
of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this
list of conditions and the following disclaimer in the documentation and/or other
materials provided with the distribution.
* Neither the author nor the names of any contributors may be used to endorse
or promote products derived from this software without specific prior written
permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ''AS IS'' AND ANY
EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
*/
```


A.2.3 Polynomial Roots Calculation

To calculate the roots of polynomials, needed by the formant tracking algorithm described in chapter 4.4.10, external parts of code were used. Because the computational effort of root-finding algorithms is very high it was experimented with several implementations. Finally the implementation by *C. Bond* which is discussed in chapter A.2.3 turned out to be the better choice. The *Speex* implementation (Sect. A.2.3) stuck in an infinite loop that froze the complete framework sometimes.

Both implementations use the *Bairstow algorithm* to determine the polynomial roots. The Bairstow algorithm is an approach, based on numerical analysis, to determine the roots of a polynomial by using the Newton method for splitting a polynomial of arbitrary degree in polynomials of second order. These polynomials can then simply be solved using the quadratic formula as given in equation (A.7).

$$a * x^2 + b * x + c = 0 \rightarrow x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a} \tag{A.7}$$

Speex Implementation

Speex[1] is an audio compression format optimised for speech signals. It is being developed as open-source project and completely license-free. Therefore the code for the polynomial root-finding algorithm can be used without any problems. The realisation itself is based on the *ACM algorithm #30* implementation of the Bairstow algorithm that was introduced by [15] in 1960.

Copyright


```
/* Copyright (C) 1981-1999 Ken Turkowski.  <turk@computer.org>
*
* All rights reserved.
*
* Warranty Information
* Even though I have reviewed this software, I make no warranty
* or representation, either express or implied, with respect to this
* software, its quality, accuracy, merchantability, or fitness for a
* particular purpose.  As a result, this software is provided "as is,"
* and you, its user, are assuming the entire risk as to its quality
* and accuracy.
*
* This code may be used and freely distributed as long as it includes
* this copyright notice and the above warranty information.


Code slightly modified by Jean-Marc Valin (2002)

Speex License:

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

*/
```


[1] http://speex.org/

C. Bond's Implementation

The implementation by C. Bond also uses the Bairstow algorithm to determine polynomial's second order coefficients. Those coefficients can simply be converted to the polynomial roots with using the quadratic formula (Equ. (A.7)). The code itself is "offered freely and without restriction"[2].

Copyright

```
 /* bairstow.c -- Bairstow's complex root finder
*
* (C) 1991, C. Bond
*
* Finds all real and complex roots of polynomials
* with real coefficients.
*
* Features:
* o Global, self-starting method,
* o Does not require initial estimates,
* o Finds all roots and quadratic factors,
*
* Caveats:
* All polynomial root finders have problems
* maintaining accuracy in the presence of
* repeated roots.
* This is no exception!
*/
```

A.3 Praat

Praat[3] is a free acoustic signal processing software with special focus on speech signals. Because of it's big functional range and high accuracy it is wide spread among speech processing developers. Praat offers possibilities including:

Sound I/O: Recording or importing of sound files, export / synthesis / data manipulation

Speech analysis: Pitch detection, formant detection, spectral analysis, jitter analysis

Speech synthesis: Source-filter based synthesis (reproduced by the glottal impulse response), articulatory synthesis

Speech manipulation: Pitch alteration, filtering

Speech labelling and segmentation: Phonetic alphabet based segmentation and labelling, appliance of feature based grids (pitch / formant / amplitude / intensity / duration)

Statistics: Discriminant analysis, multidimensional scaling, principal component analysis

ML: Neural network analysis, optimal theory analysis

In the process of developing this framework mainly the speech analysis functionality was used, primarily as a reference for the self-implemented pitch and formant trackers.

[2] http://www.crbond.com/
[3] http://www.fon.hum.uva.nl/praat/

A.4 Apparatus Usage and Operation Modes

In principle, the complete framework needs no user interaction for a proper usage. The speech enhancement is performed automatically, all configuration parameters are optimised and already set to guarantee the best possible enhancement result. Nevertheless the four **D**ual **I**nline **P**ackage (DIP) switches on the DSP board provide a simple user interface for altering the internal processing line. These changes include activating and deactivating components as well as changing to alternative calculation algorithms for some internal routines. Table A.4 shows the exact switch assignment.

DIP Switch	Mode	Functionality	
1	-	**on**	speech enhancement active
		off	signal bypass
2	-	**on**	MS active
		off	MS inactive
3	-	**on**	F0gen active
		off	F0gen inactive
4	MS on, F0gen off	**on**	MSF
		off	SS
	MS off, F0gen on	**on**	enhanced signal output
		off	pulse generator output
	MS on, F0gen on	**on**	enhanced signal output
		off	pulse generator output

Table A.4: *Functional assignment of the DSK's four available DIP switches in order to interactively control the system behaviour.*

Worth mentioning is also the default configuration of the MS module: In case of mode *MS on, F0gen on* there is no possibility to choose the filtering mode (DIP 4 is used to configure the pitch contour generation module in this case). The MS module always uses the MSF in this mode. This is reasoned with the better performance of this filter (Sect. 5.1.2).

LED	Mode	Indication
1	-	speech enhancement activity
2	-	multipath separation activity
3	-	pitch contour generation activity
4	enhanced signal output	enhanced signal output is active
	pulse generator output	voice activity (current VAD state)
		V/UV property (current V/UV state)

Table A.5: *Indicated functionality of the DSK's four available LED's dependent on the system's operating mode.*

The four on-board **L**ight **E**mitting **D**iode (LED)'s are primarily used to indicate the operating mode chosen by the DIP switch setup. So LED 1 is active if DIP 1 is active et cetera. The only exception occurs with activated pulse generator output: In this case the LED's 3 and 4 indicate the voice activity and voiced/unvoiced behaviour. This simple user feedback makes

it more convenient for the user to adjust the input level to guarantee a proper **V**oice **A**ctivity **D**etection (VAD) functionality (refer to section 4.4.6 for more information about this topic). In table A.5 a detailed list of LED indication modes is given.

A.5 Used Devices

A.5.1 Description

A list of used devices and their most important technical specifications can be found in tables A.6 and A.7. These are at first the *Servox digital* electrolarynx and *Brüel & Kjær* shaker devices[4] [5] that are used as testing devices for the production of electrolarynx voice input. The Servox digital is a commercial electrolarynx device featuring high mobility and comfort. The Brüel & Kær device however is not intended to be used by the common user. It is far more expensive and heavy than the Servox digital and targets on being used in scientific applications. Therefore it is very accurate and more importantly offers the possibility to control the vibration frequency - a necessary feature in the pitch contour generation process. The Servox digital does not provide this functionality. It's vibration frequency is fixed at a user-defined value, dependent on the pressed button one the two pre-defined frequencies can be chosen. The main component in the processing chain, the *Spectrum Digital* DSP board *TMS320C6713*[6], used for signal processing, is discussed in section 2.3 in detail and will not be discussed here furthermore. For mixing and routing various signals a *Eurorack MX602A*[7] was employed. Finally voice capturing and monitoring was carried out using a *AKG HSD171D* headset[8]. All of the above devices are directly taking part in the signal flow and are essential for the proper functionality of the framework therefore. Pictures of these devices can be found on page 106.

For amplification, measurement and verification purposes a waveform generator, oscilloscope and power amplifier were used. The waveform generator, a *Agilent 33120A*[9] was primarily needed to generate well-defined input signals for debugging purposes. The *Agilent 54622D* oscilloscope[10] was needed in such debugging situations as well - when being aware of the exact input and output accurate conclusions could be made about the internal functionality of the developed DSP program. At last the power amplifier, a *APart PA4060*[11], was necessary to amplify the generated pitch contour pulses for controlling the shaker. The rather low-energy output generated by the **D**igital Signal Processor **S**tarter **K**it (DSK) was not able to power the shaker's electric motor on it's own. Additionally a **P**ersonal **C**omputer (PC) was utilised to calculate spectrograms by capturing the enhanced audio data. In order to do so the contour-equipped electrolarynx speech signals were recorded via headset microphone and captured by the computer's sound card, namely a *Realtec ALC260*[12]. Pictures of these devices can be found on page 107.

---

[4]Manual: http://www.servona.com/picture/pdf/GBA_SERVOXdigital_0705.pdf
[5]Manual: http://www.bksv.com/doc/bp0232.pdf
[6]Manual: http://c6000.spectrumdigital.com/dsk6713/V2/docs/dsk6713_TechRef.pdf
[7]Manual: www.behringerdownload.de/MX602A/MX602A_ENG_Rev_F.pdf
[8]Manual: http://www.akg.com/mediendatenbank2/psfile/datei/3/hsc_hsd_17430d5f588a7cf.pdf
[9]Manual: http://cp.literature.agilent.com/litweb/pdf/5968-0125EN.pdf
[10]Manual: http://cp.literature.agilent.com/litweb/pdf/54622-97036.pdf
[11]Manual: http://www.apart-audio.com/uploaded_files/productFiles/PA4060/PA4060-TS.pdf
[12]Manual: ftp://Webser:Ds8MtJ3@152.104.238.19/pc/audio/ALC260(D)_DataSheet_1.4.pdf

(a) Electrolarynx, © http://www.luminaud.com

(b) Shaker, © http://eolsurplus.com

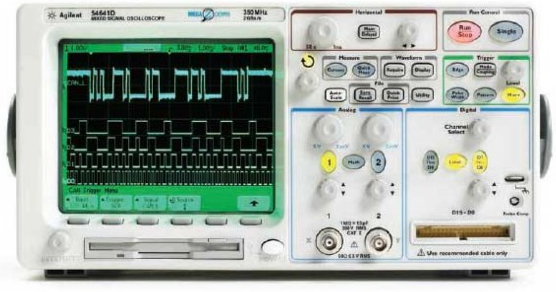
(c) DSP board, © http://www.roinos.com

(d) Mixer board, © http://www.behringer.com

(e) Headset, © http://uk.shopping.com

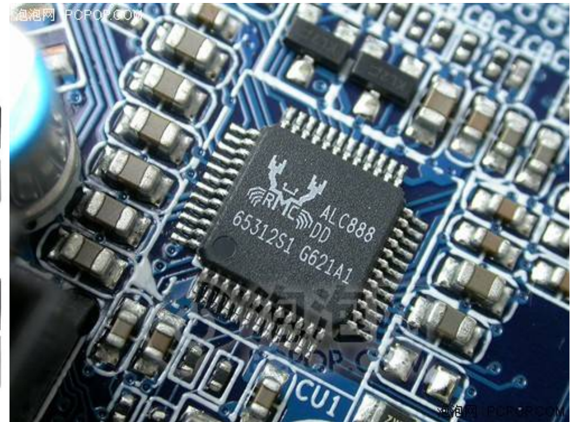
Figure A.4: *Devices that are an integral part of the framework because of being directly involved in the framework's signal processing chain.*

(a) Waveform generator, © http://www.testequity.com



(b) Oscilloscope, © http://www.daytona-sensors.com

(c) Power amplifier, © http://www.fullwave.be



(d) Sound card, © http://tech.163.com

Figure A.5: *Devices being used for utility purposes like measurement tasks and signal amplification that are not directly involved in the processing chain.*

Device Type	Product Name	Characteristics
Electrolarynx	Servox digital (source: [33])	Pitch: $\approx$ **100Hz (Button A), 250Hz (Button B)**[13] Weight: **110g device + 60g battery**
Shaker	Brüel & Kjær Type 4810 (source: [11])	Frequency range: **0Hz ... 18kHz** Energising current: $\mathbf{1.8A_{eff}}$ Weight: **1.1kg**
DSP board	Spectrum Digital TMS320C6713 DSK (source: [36])	DSP operating frequency: **225MHz** Codec type: **stereo** Memory: RAM **16MB SDRAM**, non-volatile flash **512kB** User I/O: **4 LED's, 4 DIP's** Data I/O: **Mic/line input, headphone/line output (3.5mm audio jacks)**
Mixer board	Eurorack MX602A (source: [5])	Channels: **6** Input: **2 mono (6.5mm audio jack or XLR), 2 stereo (6.5mm audio jack)** EQ: $\frac{\mathbf{3}}{\mathbf{channel}}$ **(80Hz, 2.5kHz, 12kHz)** Frequency response: **10Hz ... 60kHz (mono channel) / 55kHz (stereo channel)** Frequency range, mic input: **22Hz ... 20kHz** Output level: **+22dBu (main mix) / +22dBu (control room)** Noise absolute: **-90dBu** SNR: **113.6dB (internal) / 112dB (main mix output)** THDN: **0.007%**
Headset	AKG HSD171 (source: [3])	Microphone type: **dynamic** Headphones type: **closed-back, supraaural, dynamic** Microphone frequency range: **60Hz ... 17kHz** Headphones frequency range: **18Hz ... 26kHz** Microphone polar pattern: **hypercardioid** Headphones THD: $<$ **0.4%**

Table A.6: *List of all used devices being a part of the system's processing chain and their technical specifications.*

Device Type	Product Name	Characteristics
Waveform generator	Aglient 33120A (source: [21])	Waveforms available: **10** Sample rate: $\mathbf{40M\frac{samples}{s}}$ Resolution: Amplitude **12bit**, frequency **10μHz** Bandwidth: **100μHz . . . 15MHz (sine, square) / 100kHz (triangle, ramp)** Output: Amplitude $\mathbf{50mV_{pp}} \ldots \mathbf{10V_{pp}}$, impedance **50Ω**
Oscilloscope	Aglient 54622D (source: [2])	Type: **analog** Channels: **2** Sampling rate: $\mathbf{200M\frac{samples}{s}}$ Resolution: Amplitude **8bit**, time **40ps** Bandwidth: **0Hz . . . 100MHz**
Power amplifier	APart PA4060 (source: [4])	Channels: **4** Input: **Cinch, unbalanced** Input resistance: **10kΩ** Output: **50W / 60W (RMS mode), maximum 70W / 90W (RMS mode)** Output resistance: **8Ω/4Ω** (RMS mode) Bandwidth: **15Hz . . . 30kHz** Distortion: $< \mathbf{0.03}\%$
Sound card	ALC260 (source: [32])	Type: **on-board chip** Channels: **2** Data I/O: **Mic/line input, headphone/line output** Amplitude resolution: **16/20/24bit DAC, 16/20bit ADC** Sampling rate: **44.1/48/96/192kHz** Bandwidth: **0Hz . . . 19.2kHz** SNR: **95dB**

Table A.7: *List of all devices used for utility purposes and their technical specifications.*