UNIVERSITY OF MUSIC AND PERFORMING ARTS GRAZ
INSTITUTE OF ELECTRONIC MUSIC AND ACOUSTICS

Synthesis and Linear Prediction Analysis of Sung Vocal Signals

Evaluation of linear prediction algorithms for singing analysis and visualization of estimated voice qualities and vowels

Audio Engineering Project Thesis

Authors: Paul Armin Bereuter, BSc 01431696
Florian Kraxberger, BSc 01531632

Supervisors: Univ.Prof. Dipl.-Ing. Dr.techn. Alois Sontacchi
Dipl.-Ing. Manuel Brandner, BSc

Date: Graz, May 1, 2021
Version: v.04

Abstract

The focus of this project thesis lies on the task to categorize sung vocal signals with regards to their voice and vowel quality. As an analysis approach, the *source-filter model* is used. The source and its source signal is the airflow's time derivative through the glottis (derivative glottal flow), and the filter represents the human vocal tract. The source or excitation signal holds valuable information on the voice quality, whereas the human vocal tract defines the sung vowel.

Four different *linear prediction* methods are compared, concerning their ability to separate source and filter signals by glottal inverse filtering. Two skewness based low level features are calculated from the estimated source signal. These features are used to graphically indicate the voice quality. The estimated vocal tract is used to calculate the first two formant frequencies (F_1 and F_2), which are subsequently utilized to specify the sung vowel, by visualization within the 2D vowel space.

To evaluate the four *linear prediction* algorithms' performance, a formant error measure is defined and the classification accuracy is assessed with the help of a support vector machine (SVM). *Synthetic signals* with pre-defined parameter sets for different voice qualities (*modal*, *breathy* and *creaky*) and vowels (*/a/*, */e/*, */i/*, */o/* and */u/*) are used for the evaluation of the algorithms with regards to the formant error measure. The evaluation reveals, that the *autocorrelation method with cepstral refinement* and the *covariance method* are favoured. Using the support vector machine, the two methods are further compared with regard to their clustering performance within the 2D voice quality feature space. This analysis shows, that a tradeoff between the linear prediction methods' fundamental frequency dependence, and the test performance exists. The autocorrelation method with cepstral refinement provides a test score of $90.3\,\%$ correctly classified test samples, while maintaining the largest possible fundamental frequency range of $70\,\mathrm{Hz}$ to $320\,\mathrm{Hz}$, and is therefore deemed to be the best performing method.

The voice quality class boundaries resulting from the trained support vector machine are visualized in a 2D voice quality map. The vowels are graphically represented in a 2D vowel map, which visualizes the formant frequency space spanned by F_1 and F_2.

Furthermore, the *autocorrelation method with cepstral refinement* is implemented in a VST plug-in using C++ and the JUCE framework, which provides a graphical indication of the present vowel and voice quality by means of two 2D voice maps. The main idea of the VST plug-in implementation is to provide a possible design of a voice quality and vowel indication tool for professional singers.

Finally, the project's limitations referring to the synthetic modelling of sung vocal signals, the vowel and fundamental frequency dependence of the proposed linear prediction methods, as well as the optimization potential of the VST plug-in implementation are discussed.

Kurzfassung

Der Fokus dieser Projektarbeit liegt darauf, gesungene Stimmsignale im Hinblick auf Stimmqualität und Vokals zu kategorisieren. Als Analyseansatz wird das *Quelle-Filter-Modell* verwendet. Dabei wird die zeitliche Ableitung des Luftstroms durch die Glottis (derivative glottal flow) als Quellsignal angenommen, und der Filter stellt den menschlichen Vokaltrakt dar. Das Quell- oder Anregungssignal enthält Informationen über die Stimmqualität, wohingegen der Vokaltrakt den gesungenen Vokal definiert.

Vier verschiedene lineare Prädiktions-Algorithmen werden hinsichtlich ihrer Performance im Hinblick auf die Trennung von Quellsignal und Filter durch inverse Filterung verglichen. Aus dem geschätzten Quellsignal werden zwei auf dem statistischen Parameter der Schiefe (engl. skewness) basierende Low-Level-Features berechnet. Diese Features werden verwendet, um die Stimmqualität grafisch darzustellen. Der geschätzte Vokaltraktfilter wird verwendet, um die ersten beiden Formantfrequenzen (F_1 und F_2) zu berechnen, die anschließend zur Indikation des gesungenen Vokals durch Visualisierung in einem 2D-Vokalraum verwendet werden.

Um die Leistung der vier *linearen Prädiktionsalgorithmen* zu bewerten, wird ein Formantfehlermaß definiert und die Qualität der Stimmqualitätsindikation mit Hilfe einer Support-Vector-Machine (SVM) beurteilt. Für die Bewertung der Algorithmen hinsichtlich des Formantfehlermaßes werden *synthetische Signale* mit vordefinierten Parametern für verschiedene Stimmqualitäten (*modal*, *behaucht* und *krächzend*) und Vokale (*/a/*, */e/*, */i/*, */o/* und */u/*) verwendet. Die Auswertung zeigt, dass die *Autokorrelationsmethode mit cepstralem Liftering* und die *Kovarianzmethode* am besten abschneiden. Unter Verwendung der SVM werden die beiden Methoden weiter hinsichtlich ihres Clusterings im 2D-Merkmalraum der Stimmqualität verglichen. Diese Analyse zeigt, dass ein Kompromiss zwischen der Grundfrequenzabhängigkeit der linearen Prädiktionsalgorithmen und der Klassifikationsperformance gefunden werden muss. Die Autokorrelationsmethode mit cepstralem Liftering erreicht eine Performance von $90.3\,\%$ korrekt klassifizierter Testsamples, während der größtmögliche Grundfrequenzbereich von $70\,\mathrm{Hz}$ bis $320\,\mathrm{Hz}$ beibehalten wird. Daher wird die Autokorrelationsmethode mit cepstralem Liftering als bestgeeignete Methode angesehen.

Die aus der trainierten SVM resultierenden Stimmqualitäts-Klassengrenzen werden in einer 2D-Stimmqualitätskarte visualisiert. Die Vokale werden grafisch in einer 2D-Vokalkarte dargestellt, die den von F_1 und F_2 aufgespannten Formantfrequenzraum visualisiert.

Darüber hinaus wird die Autokorrelationsmethode mit cepstralem Liftering als VST-Plug-In unter Verwendung von C++ und dem JUCE-Framework implementiert. Dies ermöglicht es den aktuell gesungenen Vokal und die Stimmqualität in zwei farbigen 2D-Diagrammen darzustellen. Die zugrundeliegende Idee der VST-Plugin-Implementierung ist es, ein mögliches Design eines Analysetools für Stimmqualität und Vokal für professionelle Sänger zu präsentieren.

Abschließend werden die Grenzen des Projekts in Bezug auf die synthetische Modellierung von gesungenen Vokalsignalen, die Vokal- und Grundfrequenzabhängigkeit der vorgeschlagenen linearen Prädiktionsalgorithmen sowie das Optimierungspotenzial der VST-Plugin-Implementierung diskutiert.

Statuatory 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	Paul Armin Bereuter
date	Florian Kraxberger

Author Contributions

The following list provides an overview on each of the author's contributions.

- Chapters 1 and 2: Both authors contributed equally.
- Chapter 3: Paul Armin Bereuter is the main author of section 3.1 and section 3.2. Florian Kraxberger is the main author of section 3.3 and section 3.4.
- Chapters 4 and 5: Both authors contributed equally.

Contents

List of Figures

List of Tables

List of Listings

Mathematical Notation

Table 0.1 aims to provide an overview of the mathematical notation used in this project thesis.

Table 0.1 *overview of mathematical symbols and notation*

a, b, c	scalars
$\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$	vectors
$\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$	matrices
$x(t)$	a continuous-time signal
$x[n]$	a discrete-time signal
$\delta(t)$	the Dirac-delta distribution
$(x * h)[n]$	discrete (circular) convolution of $x[n]$ and $h[n]$
$\lceil \cdot \rceil$	ceiling operator, round to the next larger integer
$\lfloor \cdot \rfloor$	floor operator, round to the next smaller integer
$\lfloor \cdot \rceil$	round to the nearest integer (round half towards infinity)
$X(z) = \mathfrak{Z}\{x[n]\}$	z-transform $X(z)$ of the discrete-time signal $x[n]$
$x[n] = \mathfrak{Z}^{-1}\{X(z)\}$	inverse z-transform $x[n]$ of the filter $X(z)$
$\tilde{x}[k] = \mathcal{F}_{n \mapsto k}\{x[n]\}[k]$	discrete N-point Fourier transform $\tilde{x}[k]$ of the discrete-time signal $x[n]$
$x[n] = \mathcal{F}^{-1}_{k \mapsto n}\{\tilde{x}[k]\}[n]$	inverse discrete N-point Fourier transform $x[n]$ of the frequency domain signal $\tilde{x}[k]$
$\mathfrak{Re}\{z\}$, $\mathfrak{Im}\{z\}$	real and imaginary part of a complex variable $z \in \mathbb{C}$
$\lvert z \rvert$	absolute value of a complex variable $z \in \mathbb{C}$
$\mathcal{N}(\mu, \sigma^2)$	a Gaussian random variable with mean μ and standard deviation σ
$\mathbb{E}\{x[n]\}$	the expected value of $x[n]$ over time
$\# (\cdot)$	number of elements
$\hat{a}$	*estimation* of a quantity a
f_0, f_s	fundamental frequency and sampling frequency
$\mathcal{F}_0$	set of possible fundamental frequencies for algorithm performance analysis
F_1, F_2, F_3, F_4	formant frequencies of a vocal tract filter

1 Introduction

Speech and singing voice are the main medium used to transport information and emotions between humans. In this wide research area, this project focuses on the synthesis and analysis of sung vowels, especially on the classification of sung vowels and their voice quality. The initial questions for this project can be formulated as follows: *Which vowel is sung?* and *What is the voice quality of this vowel?* While the concept of a vowel is familiar, the term voice quality might need some clarification. An excellent introduction to the topic of voice qualities is given in [22], it covers the vocal fold's different modes of operation.

One fundamental assumption of this project and related work is the so-called source-filter model. In this context, the volume velocity airflow through the glottis during phonation is defined as the voice source. This signal is called *glottal flow (GF)* and its derivative, the *derivative of the glottal flow (dGF)*. The dGF is of great importance since it transports information about the physical movement of the vocal folds and therefore the voice quality. In a signal processing context the dGF acts as the voice source signal, the vocal tract's input, whereas the vocal tract is modelled as an all-pole filter. The pole's frequencies are called *formant frequencies*.

In this project a combined synthesis and analysis approach was used for this project, such that perfect knowledge of the ground truth is available at the analysis algorithm's end, enabling performance evaluations of the different algorithms. This synthesizer is able to produce the vowels */a/*, */e/*, */i/*, */o/* and */u/* with the voice qualities *modal (m)*, *breathy (b)* or *creaky (c)*. The parameters used for the synthesis of different vowels and voice qualities are described in chapter 2.

The analysis algorithms also assume the source-filter model. They aim to deconvolute the glottis signal and the vocal tract filter. In all four evaluated analysis algorithms the vocal tract filter is estimated using *linear prediction (LP)*, and the vowel is classified based on the first two formants determined by the estimated vowel tract filter. The residual signal of the linear prediction corresponds to the dGF. The dGF is further used for the voice quality classification.

In order to provide an anatomical context of human voice production, Figure 1.1 gives an overview of the anatomical parts involved in voice production. Using the source-filter model, the tracheal airflow, which gets interrupted perodically by the closing vocal folds, corresponds to the glottal flow (GF). Everything from the vocal folds to the mouth opening, in the so-called supraglottal region, is modelled by the vocal tract filter.

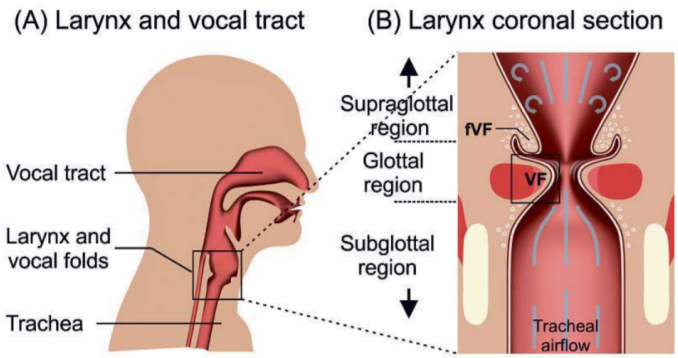


Figure 1.1 *Anatomical structures involved in human voice generation [17, Fig. 1]*

1.1 Structure of this Document

The structure of this project thesis is held in line with the signal flow of the proposed synthesis, analysis and classification algorithm. Therefore, this document consists of 4 major chapters, on which an overview is provided in the following.

Chapter 2 deals with the *synthesis of sung vocal signals* as well as the discussion of the relevant glottal and vocal tract parameters. Based on the source-filter model, an analytic approach to the generation of source signals is presented, which allows to synthesize an excitation signal with a defined voice quality. Additionally, a method of filtering the source signal is used to form the vocal tract filter.

The synthesis chapter is followed by chapter 3, which approaches the different *linear prediction algorithms*. To operate correctly, some pre-processing steps are necessary in order to provide the algorithms with all information they need. The result of the analysis algorithms needs post-processing, which includes the low-level feature calculation and the classification algorithms for voice quality and vowels. To quantify the algorithms' performance, a Monte-Carlo analysis is performed.

In chapter 4, the implementation of the best performing algorithm as a *VST plug-in* using C++ and the JUCE-framework is documented. With the VST plug-in, a tool is presented which enables a visual feedback to singers and enables a detailed feedback regarding the vowel and voice quality, for which a prospective use case is given by teaching student singers.

Finally, a *conclusion* and potential aspects for future research are given in chapter 5.

1.2 Overview of Signal Flow

Figure 1.2 provides an overview of the signal flow starting with the synthesis of the sung vocal signal, followed by the pre-processing and analysis stage, finally the classification is executed. The signal flow shown in Figure 1.2 refers strongly to the `Matlab` implementation, for the implementation with C++/JUCE a few adaptations werde made which are described in chapter 4. In the following section, each of the big signal processing blocks is described briefly.

Synthesis. In the synthesis block, the algorithm's input file is created based on models describing the glottis signal as well as the vocal tract filter. Further information on the process can be found in chapter 2. Alternatively, a recorded audio signal can be loaded. Finally, the input signal gets downsampled and split into blocks. The successive signal processing stages operate on one block at a time.

Pre-Processing. The pre-processing stage uses the residual signal of some linear prediction algorithm to provide an estimate on the fundamental frequency and the time instances where the glottis closes (the *glottal closure instants* or GCIs) and opens (the *glottal opening instants* or GOIs) . Furthermore, it is decided if the synthesized input signal is *voiced* or *unvoiced*. A detailed description of the pre-processing can be found in section 3.1.

Linear Prediction Analysis. Based on the additional knowledge derived from the pre-processing stage, four variants of linear prediction are compared. The differences between the chosen algorithms lie either in the way the input signal is whitened, by suppressing the input signal's fundamental frequency, or in the linear prediction's methodology/algorithmics. While the *covariance* and *autocorrelation* methods do not suppress the fundamental frequency they differ in the linear prediction's

estimation approach. The *windowed covariance method* and the *autocorrelation method with cepstral refinement* differ in the form of fundamental frequency suppression. The *windowed covariance method* performs the suppression in time-domain, whereas the *autocorrelation method with cepstral refinement* applies the suppression in the cepstral domain. A detailed description of the four methods is given in section 3.2.

Post-Processing and Classification. With the help of linear prediction the filter coefficients of the vocal tract filter are estimated. These coefficients are used to assign the signal block to a vowel. Additionally, inverse filtering is used to calculate an estimation of the glottis signal, i.e. the dGF (derivative Glottal Flow). Based on the estimated dGF an estimation of the glottal flow can be calculated and both are used to calculate features that enable the assignment of the signal to a voice quality. The post-processing and classification stages are described in section 3.3.

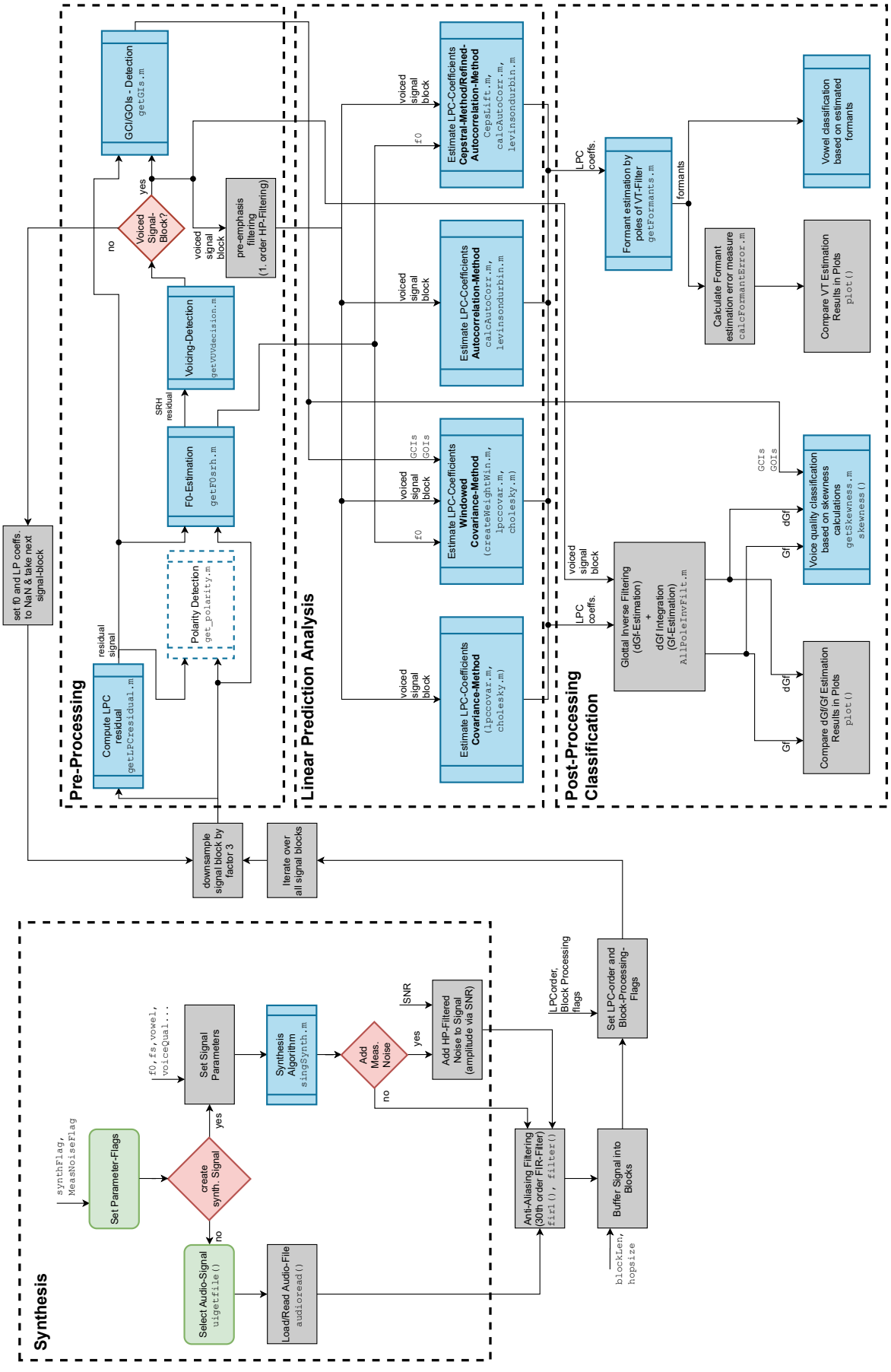


Figure 1.2 *Overview over the synthesis and analysis algorithms with the investigated LP variants. Grey blocks denote small signal operations, green blocks are input parameters, orange blocks are output objects, red blocks mark boolean decisions, and blue blocks denote more complex signal operations used in subroutines.*

2 Synthesis of Sung Vocal Signals

Existing research on synthesis of human voice mainly originates in the synthesis of *spoken* voice [1, 10]. Some researchers extended those synthesis models to provide a better applicability to *singing* voice, e.g. [39]. In essence, the most promising signal theoretic models of speech and sung voice production are based on the source-filter model, where the source is a signal based on the air flow through the vocal folds over time through and the filter is the vocal tract, which is the air filled cavity between the glottis and the mouth opening.

To model the source signal, the Liljencrants-Fant-Model (LF-model) [18] is well-established in corresponding literature. In section 2.1, special emphasis is placed on the synthesis of a sung vocal signal. The vocal tract's All-Pole-Filter-Model is described in section 2.2.

The code framework, forming the proposed synthesis algorithm's basis, was laid out by Alku *et al.* in repository 1 of their OPENGLOT-framework [1]. An overview of the implementation in `Matlab`, which combines the OPENGLOT-framework with our modifications for singing voice signals can be found in section 2.3.

2.1 Excitation Signal Modelling using the LF-Model

First, we discuss the LF-model for one glottal cycle. To synthesize the derivative of the glottal flow (dGF), which is interpreted as the excitation signal in the source-filter model context, the LF-model is executed repeatedly. Thereby, specific parameter variations, necessary for the singing voice signals, are defined to model effects such as vibrato.

2.1.1 The LF-Model for one Glottal Cycle

The LF-model is a four-parameter model of the dGF-signal. In addition to the four parameters, also the fundamental frequency f_0 is needed. Originally published by Liljencrants, Fant and Lin in [18], the LF-model was reused many times as for instance in the literature by Gobl in [22]. In the following, the LF-model and its parameters are presented, for *one glottal cycle*.

Let $E(t)$ be the dGF-signal corresponding to the excitation produced by the vocal folds. According to the LF-model, the dGF-signal $E(t)$ is defined as

$$E(t) = \begin{cases} E_1(t) & \text{for } t \leq t_e \\ E_2(t) & \text{for } t_e < t \leq t_c \end{cases} = \begin{cases} E_0 \mathrm{e}^{\alpha t} \sin(\omega_g t) & \text{for } t \leq t_e \\ \frac{-E_0}{\varepsilon t_p} \left(\mathrm{e}^{-\varepsilon(t-t_e)} - \mathrm{e}^{-\varepsilon(t_c-t_e)} \right) & \text{for } t_e < t \leq t_c \end{cases} \tag{2.1}$$

with the four parameters t_p, t_e, t_a and E_e. One period of the dGF signal is called a *glottal pulse* or *glottal cycle*, and its duration is $t_c = T_0 = \frac{1}{f_0}$, where f_0 is the desired fundamental frequency and T_0

the corresponding fundamental period. Furthermore, we know the following relations:

$$\begin{aligned} t_p &= \frac{1}{2R_g f_0} && \dots \text{time of the zero crossing of dGF} \\ t_e &= \frac{1+R_k}{2R_g f_0} && \dots \text{time instance when the vocal folds close} \\ t_a &= \frac{R_a}{f_0} && \dots \text{time constant of the return phase} \\ \omega_g &= 2\pi R_g f_0 && \dots \text{frequency of dGF in the opening phase,} \end{aligned} \tag{2.2}$$

where R_g, R_k and R_a are parameters of the LF-model formulation by Gobl [22].

For the parameter α in Equation 2.1, we look at $E(t = t_e)$.

$$\begin{aligned} E(t_e) = E_0 \mathrm{e}^{\alpha t_e} \sin(\omega_g t_e) \qquad &\Longleftrightarrow \qquad \mathrm{e}^{\alpha t_e} = \frac{E(t_e)}{E_0 \sin(\omega_g t_e)} \\ &\Longrightarrow \qquad \alpha = \frac{1}{t_e} \ln\left(\frac{E(t_e)}{E_0 \sin(\omega_g t_e)}\right) \end{aligned} \tag{2.3}$$

We call $E(t_e) = E_E$ the amplitude of the dGF at the time instance of the glottal closure (GCI) and the time instance t_0 is called glottal opening instant (GOI). In the OPENGLOT-implementation, the parameter ε is evaluated iteratively, such that at $t = t_e$, enabling a smooth transition between the two parts of the dGF-curve, i.e. $E_1(t = t_e) \approx E_2(t \to t_e)$. Therewith, the LF-model is complete. In Figure 2.1, one glottal cycle of the dGF-signal is shown.

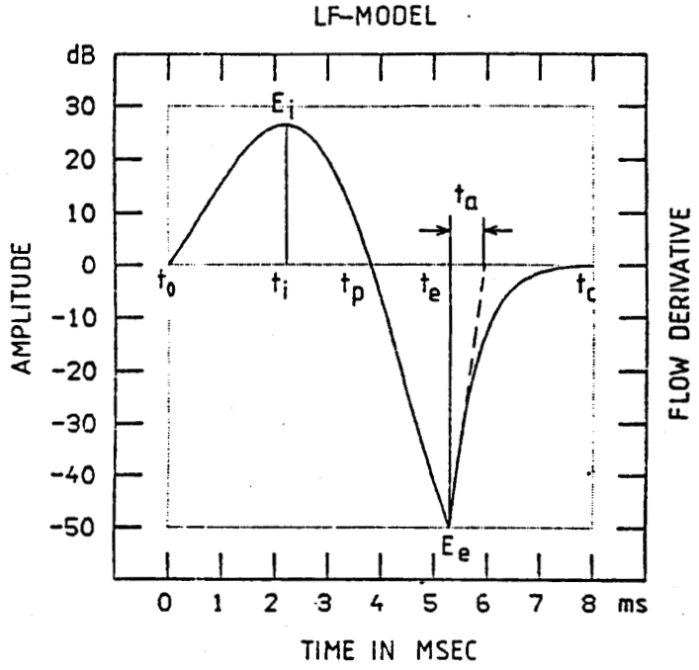


Figure 2.1 *dGF-signal according to the LF-model [18, Fig. 2]*

Due to the nature of digital signal processing and the implementation of the proposed analysis algorithm with software applications such as `Matlab` or C++/JUCE the excitation signal from Equation 2.1

is discretized using a sampling frequency f_s leading to:

$$\begin{aligned} E(t) \xrightarrow{t \mapsto \frac{n}{f_s}} E\left(\frac{n}{f_s}\right) \xrightarrow{\frac{n}{f_s} \mapsto n} E[n] \\ \text{or equivalently} \quad E[n] = \sum_{n=-\infty}^{+\infty} E(t)\delta\left(t - \frac{n}{f_s}\right), \end{aligned} \tag{2.4}$$

where $\delta(\cdot)$ denotes the Dirac-delta distribution used here for the discretization of $E(t)$ at equally spaced sampling intervals, with the sampling period $\frac{1}{f_s}$ and $n \in \mathbb{Z}$, in accordance to [25].

2.1.2 Modelling different Voice Qualities with the LF-Model

The LF-Model defined by Equation 2.11 has four parameters, which shape the dGF-curve, summarized in the parameter vector $\boldsymbol{\theta} = \begin{bmatrix} E_E & R_a & R_g & R_k \end{bmatrix}^T$. The parameters R_a, R_g and R_k determine the time instances mentioned in Equation 2.2. Additionally, the fundamental frequency f_0 and sampling frequency f_s are needed. The parameters $\boldsymbol{\theta}$ define the shape of one glottal cycle and therefore determine the voice quality. Gobl defines the parameters for different voice qualities in [22, Table II.] by describing them as Gaussian random variables $\mathcal{N}(\mu, \sigma^2)$, where μ is the mean value and σ is the standard deviation. Inherently, the parameters have some variability.

Gobl defines E_E in dB and R_a, R_g and R_k in %. The values from Gobl are listed in Table 2.1.

Table 2.1 *LF-model parameters for different voice qualities according to [22, Table II.].*

Parameter	modal (m)	breathy (b)	creaky (c)
$E_{E,\text{dB}}$	$\mathcal{N}(0, 0.1^2)$ dB	$\mathcal{N}(0.7, 0.4^2)$ dB	$\mathcal{N}(1.8, 0.3^2)$ dB
$R_{a,\%}$	$1\,\%$	$\mathcal{N}(2.5, 0.6^2)\,\%$	$\mathcal{N}(0.8, 0.5^2)\,\%$
$R_{g,\%}$	$\mathcal{N}(117, 7.4^2)\,\%$	$\mathcal{N}(117, 7.4^2)\,\%$	$\mathcal{N}(113, 11.0^2)\,\%$
$R_{k,\%}$	$\mathcal{N}(34, 1)\,\%$	$\mathcal{N}(34, 1)\,\%$	$\mathcal{N}(20, 2.4)\,\%$

For the OPENGLOT-framework, linear parameters are needed, thus the following conversion formulas are used:

$$\begin{aligned} E_{E,\text{dB}} \propto \mathcal{N}\left(\mu_{E_{E,\text{dB}}}, \sigma^2_{E_{E,\text{dB}}}\right) \text{dB} \quad &\Longrightarrow \quad E_E \propto 10^{\frac{\mu_{E_{E,\text{dB}}}}{20}} + 10^{\frac{\mathcal{N}\left(0, \sigma^2_{E_{E,\text{dB}}}\right)}{20}} \\ R_{\%} \propto \mathcal{N}\left(\mu_{R_{\%}}, \sigma^2_{R_{\%}}\right) \% \quad &\Longrightarrow \quad R \propto \mathcal{N}\left(\frac{\mu_{R_{\%}}}{100}, \left(\frac{\sigma_{R_{\%}}}{100}\right)^2\right) \end{aligned} \tag{2.5}$$

The parameters of the LF-model need to be adapted for certain voice qualities, because using the Gaussian randomized parameter set, physically impossible parameter combinations also have a certain probability (mainly $t_a < 0$, which is impossible). To cope with this edge case, some parameter adaptions are needed.

Parameter Adaptions for Breathy and Creaky Voice. The parameter R_a must be positive, because this leads to a positive time instance t_a. Therefore, a proposal candidate $R_{a,prop}$ is drawn from

the Gaussian distribution described in Table 2.1. If $R_{a,prop} < -\frac{\mu_{R_{a,\%}}}{100}$, a negative t_a would occur, so a new sample is drawn from the distribution. This operation can be interpreted as a truncation of the Gaussian distribution towards one side. The other parameters are unproblematic and need no further adaptions.

In order to visualize the impact of the chosen voice quality on the dGF signal, Figure 2.2 shows the different dGF waveforms for the three different voice qualities modal, breathy and creaky.

2.1.3 Distinctive Features of Singing Voice

Aspiration Noise for Breathy Voice. In addition to the excitation signal's shape being the determing factor of the voice quality, Lu and Smith suggest the addition of spectrally shaped, amplitude modulated Gaussian noise for a better simulation of breathy voice signals in [39]. The aspiration noise is generated in the following steps: Firstly, Gaussian white noise with an amplitude of $\frac{E_E}{8}$ is amplitude modulated using a Hann window which is centered around the glottal closure instant. The amplitude modulated white noise is filtered with a de-emphasis filter $H(z) = \frac{1}{1-pz^{-1}}$ with $p = 0.9$ followed by a 10th order IIR high-pass filter with a pass-band ripple of $0.2\,\mathrm{dB}$ and a cut-off-frequency of $f_{g,\mathrm{noise}} = 4\,\mathrm{kHz}$. Finally, this aspiration noise is added to the excitation signal created with the LF-model.

Vibrato. Sundberg defined vibrato in [66] as being a regular fluctiation of pitch, timbre and/or loudness. To model the fluctuations, variations in both amplitude and frequency have been considered, with the variation in amplitude being called *shimmer*, whereas the variation in frequency is called *jitter*. Shimmer can be interpreted as an amplitude modulation, and jitter can be interpreted as a frequency modulation. For a shimmer, a sinusoidal signal with the desired shimmer parameters was generated and multiplied onto the dGF-signal. The jitter was created by applying a variation in the fundamental frequency f_0 of the LF-model, i.e. successive glottal cycles show a variation in their fundamental period T_0 according to the frequency modulation parameters. Each modulation has two subparameters describing the extent (ratio) and speed (frequency) of the excitation signal's variation.

The parameters of shimmer and jitter are listed in Table 2.2. Sciri and Sundberg suggest a shimmer ratio in the range of $6\,\%$ to $8\,\%$. In [62, p. 25], Sciri also describes the jitter ratio in a musical context. A ratio of one semitone in both directions, which corresponds to a $\pm\,6\,\%$ pitch deviation around the fundamental frequency f_0, is suggested.

Table 2.2 *parameters of jitter and shimmer*

Parameter	**Value**	**Reference**
jitter ratio	$R_{\mathrm{jit}} = 1 - \frac{1}{2^{1/12}} \mathrel{\hat{=}} \pm$ 1 semitone$\mathrel{\hat{\approx}} \pm\,6\,\%$	[62, p.25]
jitter frequency	$f_{\mathrm{jit}} = 6\,\mathrm{Hz}$	[62]
shimmer ratio	$R_{\mathrm{shim}} = 0.07 \mathrel{\hat{=}} 7\,\%$	[62, 66]
shimmer frequency	$f_{\mathrm{shim}} = 6\,\mathrm{Hz}$	[62]

The application of the vibrato on the synthesized signal is shown in subsection 2.3.3. The synthesized excitation signal $E(t)$ for the three different voice-qualities with the additional features (Aspiration Noise and Vibrato) can be seen in Figure 2.2.

It is visible in Figure 2.2, that the amplitude for *creaky* vocal signals is smaller than the for *breathy* or *modal* voice. Furthermore, the creaky vocal signal has a larger time interval where it is zero, which is

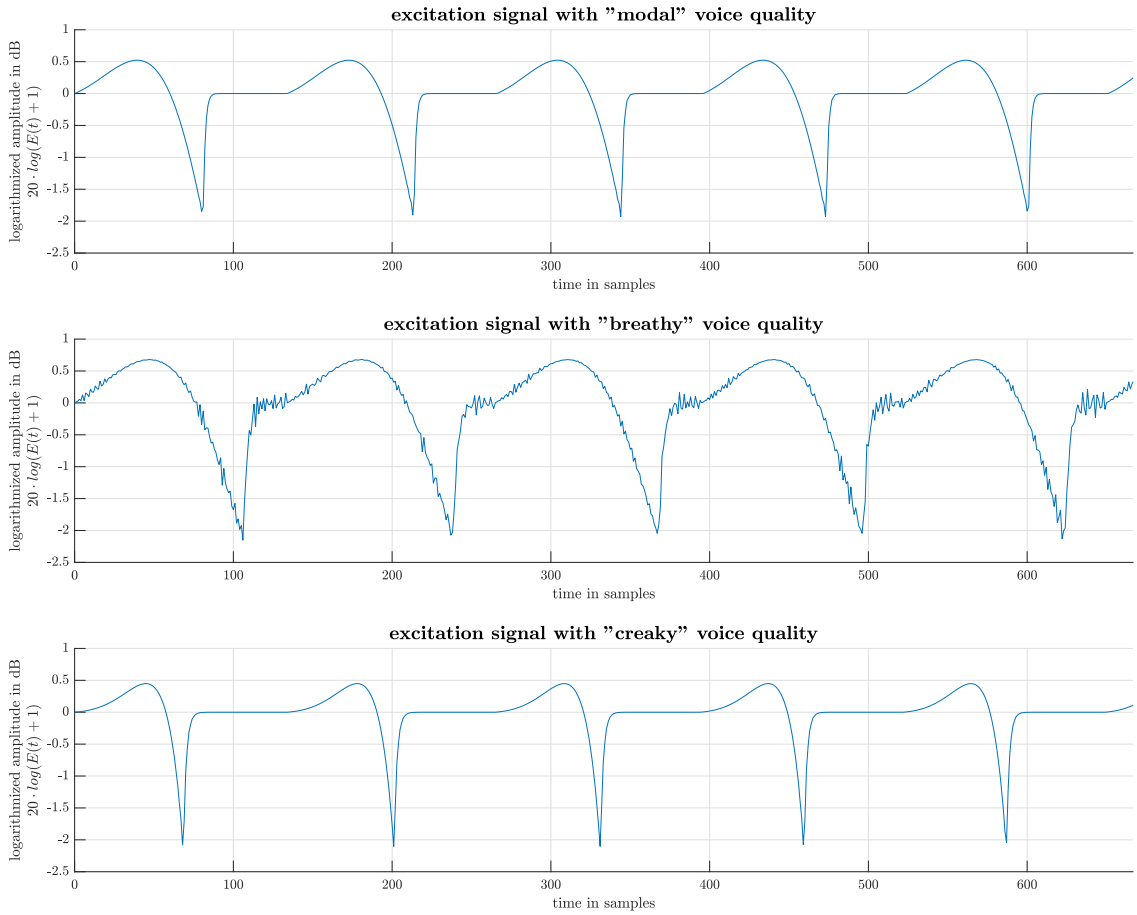


Figure 2.2 *dGF-signal for the different voice qualities of the LF-model [18, Fig. 2]*

the reason why the creaky vocal signal can be described as *sparse*. The effects of the aspiration noise are visible for the *breathy* case and the effects of vibrato (in the form of an amplitude modulation) can be seen in the non-constant negative amplitudes occuring at the GCIs for all voice qualities. The difference in the amplitude values with respect to Figure 2.1 can be lead back to the fact that different amplitude gains were used. Note that, while Figure 2.1 was taken from [18], Figure 2.2 was created with the synthesizer implemented in `Matlab` and explained in section 2.3.

2.2 All-Pole-Filter Modelling of a Vocal Tract

In the context of this work, the vocal tract's responsibility is solely considered to be the articulation and voicing of different vowels, which are produced using the variability of the vocal tract's geometry. Interpreted in a signal processing context, these variations result in different filter transfer functions for each spoken or sung sound. Gold and Rabiner suggested in [23], that the vocal tract can be modelled with an all-pole filter. In contrast to that, Ziegerhofer suggested in her Master's Thesis [74] that the all-pole-model is limited, when it comes to modelling female vocal tracts due to tracheal coupling and interactions between the voice source and the vocal tract. As the literature on gender-specific topics of this matter is rare, we focused on existing data, which is unfortunately mainly present for males. Therefore, four formants with the frequencies F_1, F_2, F_3 and F_4 (in Hz) have been

modelled with the all-pole filter model from [23]. Therein, the transfer function $H_i(z)$ for the i-th formant is given by

$$H_i(z) = \frac{1 + r_i^2 - 2r_i \cos\left(\frac{b_i}{f_s}\right)}{1 - \left(2r_i \cos\left(\frac{b_i}{f_s}\right)\right) z^{-1} + r_i^2 z^{-2}}, \tag{2.6}$$

where $r_i = \mathrm{e}^{-\frac{\pi g_i}{f_s}}$ with f_s being the sampling frequency, $b_i = 2\pi F_i$ is the angular frequency and g_i is the bandwidth of the i-th formant.[1] By combining the four transfer functions $H_1(z)$ to $H_4(z)$, the transfer function $H_{\mathrm{VT}}(z)$ of the vocal tract can be written as

$$H_{\mathrm{VT}}(z) = \prod_{i=1}^{4} H_i(z). \tag{2.7}$$

Transforming this relation back into time-domain results in the impulse response $h_{\mathrm{VT}}[n]$, such that

$$h_{\mathrm{VT}}[n] = \mathfrak{Z}^{-1}\{H_{\mathrm{VT}}(z)\}, \tag{2.8}$$

where $\mathfrak{Z}^{-1}\{\cdot\}$ denotes the inverse z-transform.

The source filter $S(z)$ and the transfer function $(1 - z^{-1})$ approximating the mouth-to-transducer radiation, which were also introduced in [23, p. 83], have been neglected here in accordance to [1].

Formant Frequencies and Bandwidths. The literature on the exact frequencies of the formants for *singing* voice is rare. Fleischer *et al.* [19] investigated 3D models of human vocal tracts singing the vowels */a/*, */i/* and */u/*, which were acquired with MRI data for sung vowels. They simulated the acoustic field inside the vocal tract with a finite element method and evaluated the formant frequencies and bandwidths. The missing data for the vowels */e/* and */o/* was taken from [23], but we set $F_4 = 3000\,\mathrm{Hz}$. For the formant frequencies of the vowel */a/*, [23] was used with the bandwidths from [19]. In Table 2.3, the parameters of the vocal tract filter (formant frequencies F_1 to F_4 and bandwidths b_1 to b_4) which were used in the synthesizer are listed with their respective references.

Table 2.3 *formant frequencies and formant bandwidths of the vocal tract filter according to [19, 23]*

Vowel	**Frequency** (in Hz)				**Bandwidth** (in Hz)				**Reference**	
	F_1	F_2	F_3	F_4	g_1	g_2	g_3	g_4	Frequency	Bandwidth
/a/	730	1090	2440	3000	68	33	69	74	[23, Tab. I]	[19, Tab. 1]
/e/	530	1840	2480	3000	60	100	120	175	[23, Tab. I]	[23, Tab. II]
/i/	360	1700	2313	2827	49	38	59	74	[19, Tab. 1]	[19, Tab. 1]
/o/	570	840	2410	3000	60	100	120	175	[23, Tab. I]	[23, Tab. II]
/u/	409	1356	2533	2819	38	42	67	79	[19, Tab. 1]	[19, Tab. 1]

The final step to obtain the synthesized sung vocal signal $y[n]$ is a convolution of the discretized excitation signal $E[n]$ from Equation 2.1 with the impulse response of the vocal tract filter $h_{\mathrm{VT}}[n]$. This corresponds to filtering the source (dGF) signal $E[n]$ with the vocal tract filter $H_{\mathrm{VT}}(z)$, i.e.

$$s[n] = (E * h_{\mathrm{VT}})[n]. \tag{2.9}$$

[1] Gold and Rabiner used the *half-bandwidth*, which in our formulation is compensated by the factor π instead of 2π in the exponent.

The convolution of the discretized excitation signal with the vocal tract filter impulse response is not explicitly calculated in the implementation, as there are a lot of Matlab-commands enabling easier computation (for example `filter()`). More details on the Matlab-implementation can be found in the following section 2.3.

2.3 Implementation in Matlab

For the purpose of this project, parts of the code provided in repository 1 of OPENGLOT [1], used for spoken vowel synthesis, were extended to better suit the nature of *sung* vowels, as described in section 2.1 and section 2.2. Figure 2.3 shows the flow chart and provides an overview on the synthesis algorithm. In this section, the implementation of the synthesis algorithm's individual blocks is discussed.

2.3.1 Input Parameters

The entry point to the synthesis algorithm is the Matlab file `singsynth.m`. In Figure 2.3 the input parameters are marked in green. To clarify the connection between Figure 2.3 and the Matlab code, the input parameters are listed with the variable names used in Matlab.

- `f0:` Fundamental frequency f_0. If jitter is applied, this variable represents the *central value* of fundamental frequency, around which the actual fundamental frequency is fluctuating.
- `jitter`: Array containing the jitter parameters

$$\texttt{jitter} = \begin{bmatrix} \texttt{jitterFlag} & R_{\text{jit}} & f_{\text{jit}} \end{bmatrix}, \tag{2.10}$$

  where `jitterFlag` is a boolean value denoting if jitter should be applied, R_{jit} is the jitter ratio and f_{jit} the jitter frequency.
- `voiceQual`: The parameter `voiceQual` is a string that denotes the desired voice quality, which can be either modal, breathy or creaky (in short `m`, `b`, or `c`, respectively), i.e. $\texttt{voiceQual} \in \{\texttt{m}, \texttt{b}, \texttt{c}\}$.
- `fgNoise`: Cut-off frequency $f_{g,\text{noise}}$ for the high-pass filter that is used for shaping the aspiration noise as described in subsection 2.1.3. $f_{g,\text{noise}}$ can also be used as the parameter fixing the noise-level of the "breathy" voice. A lower cut-off frequency adds more aspiration noise and lets the voice become more "breathy". A natural breathyness level can be achieved with $f_{g,\text{noise}} = 4000\,\text{Hz}$.
- `vowel:` String that denotes the desired vowel, which either /a/, /e/, /i/, /o/ or /u/, i.e. $\texttt{vowel} \in \{\texttt{a}, \texttt{e}, \texttt{i}, \texttt{o}, \texttt{u}\}$.
- `shimmer:` Array containing the parameters of the shimmer such that

$$\texttt{shimmer} = \begin{bmatrix} \texttt{shimmerFlag} & R_{\text{shim}} & f_{\text{shim}} \end{bmatrix}, \tag{2.11}$$

  where `shimmerFlag` is a boolean value denoting if shimmer should be applied, R_{shim} is the shimmer ratio and f_{shim} the shimmer frequency.

These input parameters are complemented by the sampling frequency f_s and the desired duration `sigLen` in seconds, which are implicitly needed, The `renderFlag`, determines if a `wav`-file of the voice sound should be rendered. Furthermore, the parameter `LFParams` can be used in conjuction with $\texttt{voiceQual} = \text{'}\texttt{custom}\text{'}$, to skip the randomization of the glottal parameters defined in Table 2.1 and use fixed LF parameters.

2.3.2 Return Parameters

The `Matlab` function `singsynth()`, whose input parameters are described in subsection 2.3.1, returns the following parameters, which are marked orange in Figure 2.3. To clarify the connection between Figure 2.3 and the `Matlab` code, the output parameters are listed with the variable names used in `Matlab`.

- `trueParamGlot:` The true glottal parameters $\boldsymbol{\theta} = \begin{bmatrix} E_E & R_a & R_g & R_k \end{bmatrix}^T$, that were actually used for the synthesis of the dGF signal.
- `sig:` Synthesized singing voice signal (dGF-signal filtered by vocal tract filter with vibrato, if desired).
- `dGF:` Synthesized dGF signal
- `GF:` Synthesized glottal flow signal
- `trueParamVT:` filter coefficients of the vocal tract filter $H_{\mathrm{VT}}(z)$
- `f0return:` frequency vector containing the fluctuating fundamental frequency f_0. For each element in `f0return`, the LF-model method `lf.m` was called once, meaning that successive glottal cycles are following the frequency modulation defined by the jitter parameters listed in Table 2.2.

2.3.3 Signal Flow of the Synthesis Algorithm

The signal flow shown in Figure 2.3 can be divided into three parts: First, the dGF-signal is synthesized, then the vocal tract filter coefficients are calculated, and finally, the dGF-signal is filtered with the vocal tract filter. In the following section, these parts are discussed.

Synthesis of the dGF-Signal. If jitter is desired (`jitterFlag` $= 1$), a frequency vector $\boldsymbol{f}_{0,mod}$ (`f0mod` in `Matlab`) containing the fluctuating fundamental frequency f_0 is created according to R_{jit} and f_{jit} as follows,

$$\begin{aligned} \boldsymbol{f}_{0,mod} &= \begin{bmatrix} f_{0,mod}[0] & \cdots & f_{0,mod}[N_{\text{jit}} - 1] \end{bmatrix}^T \\ f_{0,mod}[n] &= f_0 + R_{\text{jit}} \cdot f_0 \cdot \sin\left(\frac{2\pi f_{\text{jit}}}{f_{\text{s,jit}}} n\right) \\ f_{\text{s,jit}} &= \frac{N_{\text{jit}}}{t_{\text{len}}} = \frac{\lceil f_0 \cdot t_{\text{len}} \rceil}{t_{\text{len}}}, \end{aligned} \tag{2.12}$$

where $N_{\text{jit}} = \lceil f_0 \cdot t_{\text{len}} \rceil$ is the frequency vector length in samples (calculated with the ceiling operation $\lceil \cdot \rceil$), t_{len} is the signal length in seconds, denoted in `Matlab` with `tlen`, and f_0 is the mean value of fluctuating fundamental frequency $f_{0,mod}$.

For each element of the fundamental frequency vector $\boldsymbol{f}_{0,mod}$, `lf.m` is called to create one cycle of the dGF signal. Thereby, the randomized glottal parameters according to Table 2.1 are used, which are determined by the voice quality (parameter `voiceQual`). The true glottal parameters that were actually used (after the randomization), are returned in the array `trueParamGlot`. For breathy voice, aspiration noise according to subsection 2.1.3 is added to the dGF-signal. The resulting signal is the synthetic dGF signal, which is returned via the parameter `dGF`. The dGF-signal $E[n]$ can be numerically integrated with the `Matlab`-function `cumsum()` [41], which results in the glottal flow (GF).

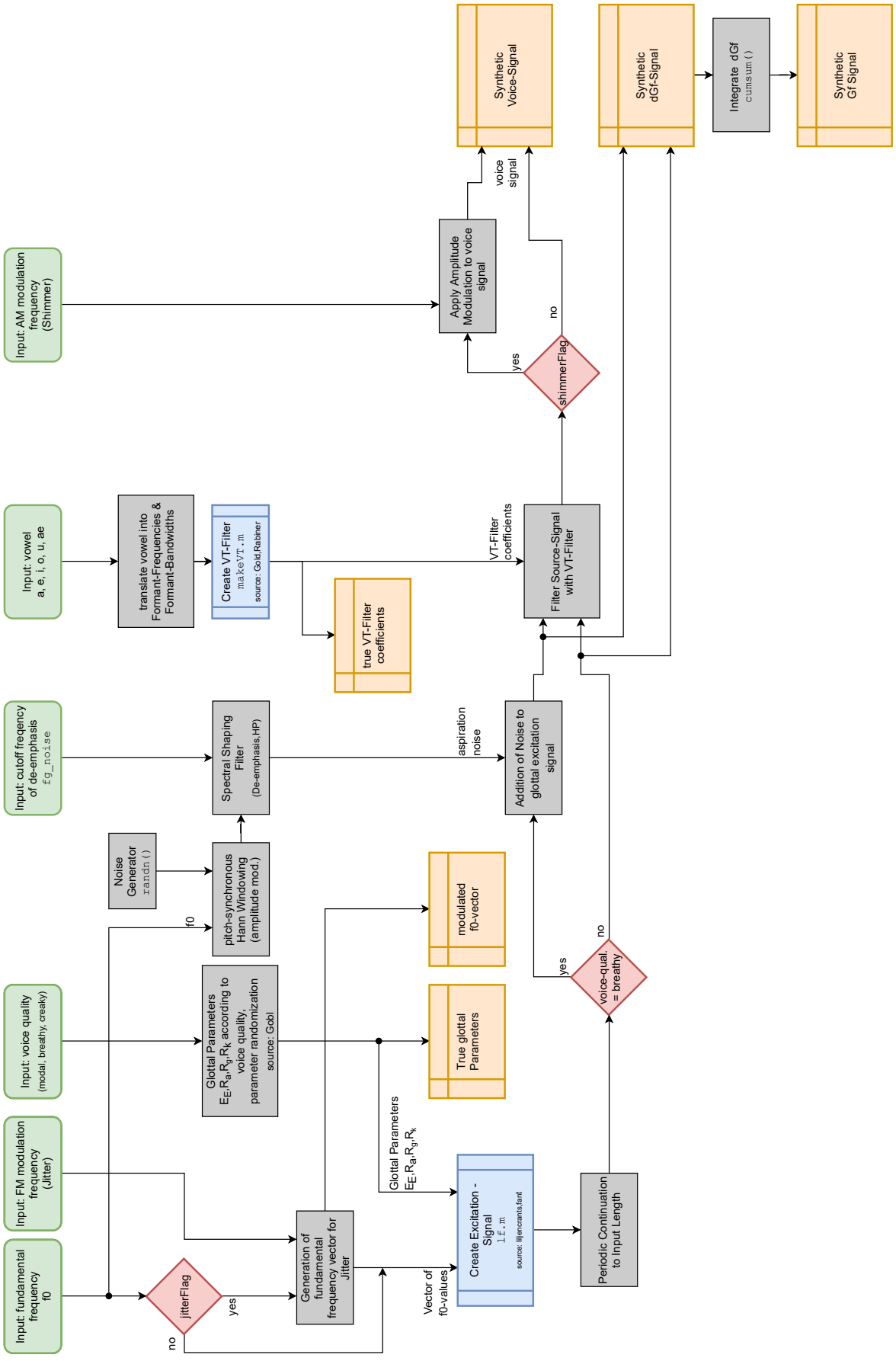


Figure 2.3 *Overview of the synthesis algorithm. Grey blocks denote small modified synthesis operations, green blocks are input parameters, orange blocks are output objects, red blocks mark boolean decisions, and blue blocks denote subroutines from OPENGLOT.*

After the frequency modulated excitation signal is created. Amplitude modulation is applied according to R_{shim} and f_{shim}, if desired. Mathematically this can be formulated such that

$$E_{\text{AM}}[n] = \left(1 - R_{\text{shim}} + R_{\text{shim}} \cdot \cos\left(\frac{2\pi f_{\text{shim}}}{f_s} n\right)\right) \cdot E[n].$$

$$E[n] \ldots \text{excitation signal before amplitude modulation (Equation 2.9)} \tag{2.13}$$

$$E_{\text{AM}}[n] \ldots \text{amplitude modulated excitation signal}$$

Note that the amplitude modulation for the sung vocal signals created in the course of this project, was applied post vocal tract filtering. Meaning that in Equation 2.13, $E[n]$ should actually be replaced with $s[n]$. However as mentioned in [74], where shimmer and jitter are defined as measures calculated from so called Electroglottography (EGG) signals, which correspond to the excitation signal at the vocal folds, hence the formulation of Equation 2.13 including the excitation signal. Also the modelled vocal tract filter is a linear time invariant filter, leading to negligible differences between signals where the amplitude modulation was applied pre or post vocal tract filtering. An exemplary comparision of a sung vocal signal where the amplitude modulation was applied onto the excitation signal ($s_{\text{AM, pre}}[n]$) with another sung vocal signal where the amplitude modulation was applied post vocal tract filtering ($s_{\text{AM, post}}[n]$) is shown in Figure 2.4. The differences between the two signal variants lie in the range of the third decimal place. For the evaluations executed in section 3.4 signals where the amplitude modulation was applied post filtering ($s_{\text{AM, post}}[n]$) were used.

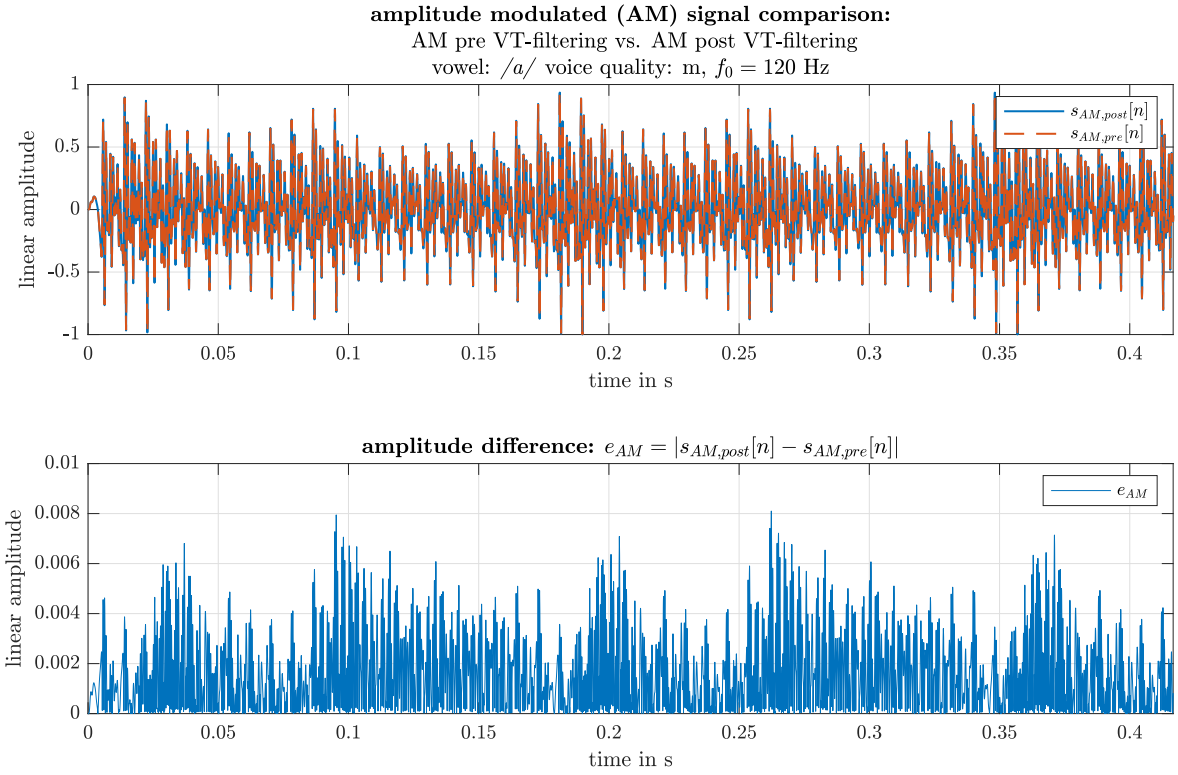


Figure 2.4 *Comparison of signals with amplitude modulation pre VT-filtering vs. post VT-filtering*

Synthesis of the Vocal Tract Filter. The vocal tract filter is synthesized according to the desired vowel with the procedure described in section 2.2. Thereby, the formant frequencies and bandwidths

defined in table Table 2.3 are used. The vocal tract filter is stored by means of its coefficients, which are also returned via the parameter `trueParamVT`.

Filtering the dGF-Signal with the Vocal Tract Filter. Using the Matlab-function `filter()` [45], the synthesized dGF-signal is filtered with the vocal tract filter. This corresponds to the convolution mentioned in Equation 2.9. If shimmer is desired the convolution changes to:

$$s[n] = (E_{\text{AM}} * h_{\text{VT}})[n] \tag{2.14}$$

This concludes the synthetic voice signal generation. The synthetic vocal signal $s[n]$, is returned via the parameter `sig`.

Measurement Noise. Under certain circumstances, it might be desired to add a small amount of shaped white noise to the synthesized vocal signal. On the synthesis side, this can be interpreted as making the synthesized signals more "realistic"[2], and on the analysis side this noise is used to regularize the analysis algorithm calculations. Essentially, zero-mean, unit-variance Gaussian white noise $w[n]$ is filtered with the transfer function $H_{\text{noiseSh}}(z)$, which is defined as

$$\begin{aligned} H_{\text{noiseSh}}(z) &= \frac{z}{z - \mathrm{e}^{\frac{-2\pi f_{\text{c,noiseSh}}}{fs}}} \\ h_{\text{noiseSh}}[n] &= \mathfrak{Z}^{-1}\{H_{\text{noiseSh}}(z)\} \end{aligned} \tag{2.15}$$

where $f_{\text{c,noiseSh}}$ is the cutoff frequency of the noise shaping filter and $h_{\text{noiseSh}}[n]$ is the impulse response of this filter. Thus, the result of the noise shaping can be calculated such that

$$w_{\text{noiseSh}}[n] = (w * h_{\text{noiseSh}})\,[n]. \tag{2.16}$$

Then, a signal to noise-ratio $\text{SNR}_{\text{noiseSh}}$ in dB is introduced, which is used to calculate the amplitude of the additive shaped noise $w_{\text{noiseSh}}[n]$. The noise is added to the synthesized vocal signal (here $s[n]$ is used to denote a synthesized vocal signal with or without shimmer), such that

$$s_{\text{noiseSh}}[n] = s[n] + \max\left(s[n]\right) \cdot 10^{\frac{-\text{SNR}_{\text{noiseSh}}}{20}} \cdot w_{\text{noiseSh}}[n]. \tag{2.17}$$

In the present implementation, the following values for $f_{\text{c,noiseSh}}$ and $\text{SNR}_{\text{noiseSh}}$ were used

$$f_{\text{c,noiseSh}} = 10\,\text{Hz} \qquad \text{and} \qquad \text{SNR}_{\text{noiseSh}} = 96\,\text{dB}. \tag{2.18}$$

In the Matlab-implementation, the addition of the measurement noise, as defined in Equation 2.17, can be found in the file `V12b_LPA_JUCE_Matlab_Reference/Main_LP_Analysis_Algorithm_Prototype.m`. It is activated by setting `MeasNoiseFlag` $= 1$.

2.3.4 Matlab Code Files

The synthesis algorithm shown in Figure 2.3 was implemented in seperate files. The structure of the Matlab files listed below was introduced by Alku *et al.* in repository 1 of [1]. The purpose of each file is listed as follows:

[2] In this context, "realistic" means *similar* to a *real recording* of a *real singer*, which always ehibit some background noise.

- `singsynth.m`: This file acts as an entry point to the synthesis algorithm and it is called from the `Matlab`. In itself, the file calls `synthFrame_V04.m` and exports the sound by calling `renderFile.m`, if desired.
  - The input parameters are listed in subsection 2.3.1.
- `synthFrame_V04.m`:
  - maps voice quality to glottal parameters according to Table 2.1 and Table 2.2
  - creates dGF signal by calling `lf.m` repeatedly
  - maps vowel to formant frequencies and bandwidths according to Table 2.3
  - creates vocal tract filter by calling `makeVT.m`
  - filters dGF signal with vocal tract filter
- `lf.m`: creates one glottal cycle according to the LF-model with the parameters defined in `synthFrame_V04.m`
- `makeVT.m`: calculates the coefficients of the vocal tract filter $H_{\mathrm{VT}}(z)$
- `renderFile.m`: exports the audio files in `wav`-format, if desired

3 Analysis of Sung Vocal Signals

In chapter 2 the main idea of the synthesis algorithm is the *source-filter model*, where the convolution of an excitation signal with a vocal-tract filter delivers a synthesized sung vocal signal. The main goal of the algorithm's analysis part is to calculate back to the *source signal*, or equivalently, the *excitation signal*. This means that the convolution of the vocal tract filter's impulse response with the excitation signal shown in Equation 2.9 needs to be reverted. This is obtained by inversely filtering the sung vocal signal with an estimated all-pole vocal tract filter, which leads to the excitation signal, also called derivative glottal flow (dGF) as mentioned in section 2.1. This method, called Glottal Inverse Filtering (GIF) is described in [1]. A template for the analysis' signal flow was proposed by Drugman *et al.* in [13]. It consists of a pre-processing stage, after which the glottal signals are estimated by Glottal Inverse Filtering (GIF).

In the following sections the adapted pre-processing stage (section 3.1) as well as different estimation methods used to estimate the vocal tract filter (section 3.2) are introduced. In the post-processing stage, a vowel estimation based on the estimated formant frequencies of the vocal tract is performed, and a method to evaluate the voice quality is introduced (section 3.3). The results of the different vocal tract filter estimation methods are evaluated using a Monte Carlo simulation (see section 3.4). In the synthesis and analysis algorithm's overview shown in Figure 1.2 the blocks *Pre-Processing* and *Linear Prediction Analysis* are the relevant blocks dicussed in this section.

As a basis for the forthcoming sections, some considerations need to be made. Firstly, it is defined, that we operate with a *block-based signal processing* approach, which uses *downsampling* of the original signal. Both are defined a priori in the following paragraphs.

Block-Based Signal Processing. The analysis-stage of this algorithm, as discussed in this section, operates as a block-based signal processing chain. This means that the executed subroutines (marked as blue blocks in Figure 1.2) work on the signal in a blockwise manner. The signal is blocked with the following blocking parameters:

- block-length: $t_{\text{block}} = 80\,\text{ms}$
- hopsize: $t_{\text{hop}} = 24\,\text{ms}$ (corresponds to an overlap of $70\,\%$)

In the discrete time-domain t_{block} becomes $N_{\text{block}} = \lfloor f_s \cdot t_{\text{block}} \rceil$, the same holds for the hopsize t_{hop}. In `Matlab` the blocking is executed using the `buffer()` command [40].

Downsampling. For speech analysis algorithms the second step of the analysis stage is commonly the decimation of the sampling frequency. As shown in Figure 1.2, before the signal is organized according to the buffer structure ($80\,\text{ms}$ blocks with $70\,\%$ overlap), the signal is filtered with a 30th order Anti-Aliasing FIR-Filter created with the `Matlab` command `fir1()` [47]. Each signal block is then downsampled by a factor of three, meaning that every third sample of a signal block is used, whereas the others are discarded. Effectively, a signal sampled with $48\,000\,\text{kHz}$ is downsampled to $f_s = 16\,000\,\text{kHz}$ and $44\,100\,\text{kHz}$ signals are downsampled to $f_s = 14\,700\,\text{kHz}$.

It is important to note that, from now on, the sampling frequency's symbol f_s refers to the *decimated sampling frequency*, and also the sung vocal signal's notation $s[n]$ refers to a downsampled $80\,\text{ms}$ signal-block rather than the whole signal, as the following steps are all executed in a blockwise manner and were created with the intention of a real-time application implementation.

3.1 Pre-Processing

The pre-processing step's aim is to deliver information on the sung vocal signal, necessary for further analysis as mentioned in section 3.2. The pre-processing can be summarized by four processing steps:

1. computation of the residual signal
2. estimation of fundamental frequency
3. estimation of glottal instants
4. pre-emphasis filtering

In [13] the *polarity estimation* is introduced as an additional pre-processing step. Due to the fact, that the analysis stage was created to solely analyze the synthesized signals described in chapter 2, where the polarity of the signals can be fully controlled, a polarity estimation is not necessary for the proposed analysis algorithm. In Figure 1.2 the polarity check is visualized as a optional subroutine but it is not implemented in the analysis stage and therefore isn't further explained. Nevertheless, a robust way of polarity estimation was proposed by Drugman in [12].

The remaining steps concerning the estimation of the fundamental frequency proposed in [14], and the detection of the glottal instants mentioned in [15] and [16] both use a residual signal in order to evaluate the relevant information. The calculation of the residual signal is described in subsection 3.1.1, and its further processing to retrieve the fundamental frequency and the glottal instants are introduced in subsection 3.1.2 and 3.1.3.

3.1.1 Computation of the LP residual

The residual signal $e[n]$ is obtained by inverse filtering as described in [14]. So at this point of the algorithm a rough estimation of a vocal tract filter using linear prediction is already executed. With the help of the Levinson-Durbin recursion mentioned in subsection 3.2.1 a rough estimate of the vocal tract filter is obtained and the sung vocal signal is inversly filtered. A LP order p_{rough} of

$$p_{\text{rough}} = \left\lfloor \frac{f_{\text{s, dec}}}{1000} \right\rceil + 2 \tag{3.1}$$

is used ($\lfloor \cdot \rceil$ denotes rounding to the nearest integer).

To ensure the distinction between the residual signal and the dGf-signal, whose calculation with the linear prediction analysis is mentioned in section 3.2, the residual signal is denoted with $e[n]$ whereas the dGF-signal is denoted with $E[n]$ and its estimate with $\hat{E}[n]$.

Due to the all-pole-filter assumption concerning the vocal tract filter, its inverse filter corresponds to a finite impulse response (FIR) filter. In `Matlab` the inverse filtering is executed with the `filter()`-command [45], in which the roughly estimated vocal tract filter poles are used as zeros. The first $2 \cdot p_{\text{rough}} + 1$ samples are set to zero in order to delete FIR-filtering artifacts which, which would effect the final normalization of the residual signal in an unwanted manner. A comparison of a sung vocal signal, and the residual signal estimated from it is shown in Figure 3.1.

The impulse-like character of the residual signal $e[n]$ is visible in the second subplot of Figure 3.1. If the residual signal is compared to the exemplary dGF signals of figure Figure 2.2, the form of the dGF signal $E[n]$ can already be suspected even though it looks very noisy. The aim of the pre-processing is to obtain information from the vocal signal which helps to modify the LP algorithms or their input data, such that the final dGF estimates the synthesized dGF better.

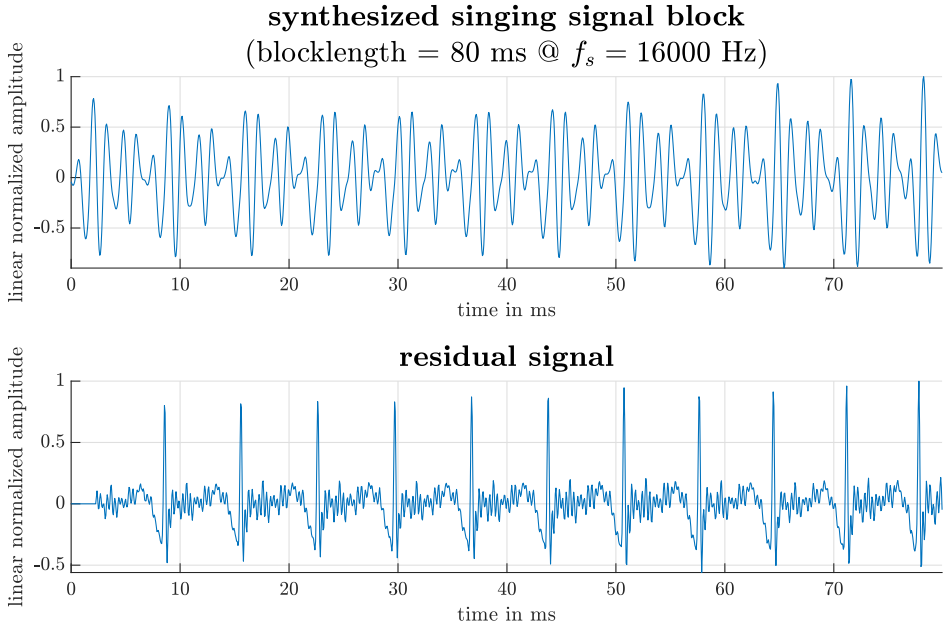


Figure 3.1 *Synthesized sung vocal signal and residual signal obtained by inverse filtering with a roughly estimated vocal tract*

3.1.2 Estimation of Fundamental Frequency f_0 and Voiced/Unvoiced Detection

The first usage of the residual signal $e[n]$ is found in the estimation of the fundamental frequency f_0, which also includes the voiced/unvoiced detection of the current signal block. The f_0 estimation and voicing detection are based on [14]. The fundamental idea is based on a adapted summation of the so-called residual harmonics. Three steps are executed before the fundamental frequency is estimated and the voiced/unvoiced decision is made.

1. The residual signal $e[n]$ is transformed to frequency domain and normalized to the overall spectral energy

$$\begin{aligned} \tilde{e}[k] &= \mathcal{F}_{n\mapsto k}\{e[n]\}[k] \\ e[k] &= \frac{\tilde{e}[k]}{\sqrt{\sum_{k=0}^{N_{\text{FFT}}-1} \tilde{e}^2[k]}} \\ \Delta f &= \frac{f_s}{N_{\text{FFT}}} \\ f_k &= \Delta f \cdot k, \qquad \text{with} \qquad k = 0, 1, \ldots, N_{\text{FFT}} - 1, \end{aligned} \tag{3.2}$$

   where f_k if the frequency in Hz, k is the discrete frequency index, f_s is the sampling frequency in Hz, N_{FFT} is the length of the frequency transform, Δf is the discrete frequency resolution and $\mathcal{F}\{\cdot\}$ is a N-point discrete Fourier transform (N-point DFT), as implemented in `Matlab`'s `fft()`-command [43]. Alternatively, we can also address the discrete frequency bins k using their respective frequency f_k, thus it is possible to write $e(f_k)$ where $f_k = \Delta f \cdot k$, as defined in the equation above.

2. The residual harmonics are summed to obtain the *summation of residual harmonics* (SRH), inside a frequency interval given with $f_k \in [f_{0,\text{min}}, f_{0,\text{max}}]$. The energy of the harmonic series

is summed up and the energy at frequencies between the harmonics is subtracted and therefore deemphasized as described in [14], such that

$$SRH(f_k) = e(f_k) + \sum_{m=2}^{N_{\text{harm}}} e(m \cdot f_k) - e((m - 1/2) \cdot f_k), \tag{3.3}$$

where N_{harm} is the number of harmonics that are to be summed. In the implementation from [14], a interval of $f_k \in [f_{0,\text{min}}, f_{0,\text{max}}] = [50\,\text{Hz}, 640\,\text{Hz}]$, a frequency resolution of $\Delta f = 1\,\text{Hz}$ and $N_{\text{harm}} = 5$ harmonics are used.

3. The last step is that the maximum value of $SRH(f_k)$ is determined. The frequency for which the maximum value occurs determines the fundamental frequency estimate $\hat{f}_0$ and the value at the maximum SRH_{max} is used for simple thresholding in order to achieve the voiced/unvoiced decision.

$$\begin{aligned} SRH_{\text{max}} &= \max\{SRH(f_k)\} \\ \hat{f}_0 &= \underset{f_k}{\text{argmax}}\{SRH(f_k)\} \end{aligned} \tag{3.4}$$

A signal block is deemed to be a voiced signal block if

$$SRH_{\text{max}} > 0.07. \tag{3.5}$$

The `Matlab` implementation of [14] found in the COVAREP database established by Degottex *et al.* in [11] was created for offline analysis of whole signals, and a blocking of the signal is included within the SRH algorithm. Due to the fact, that the implementation of this algorithm is already based on blockwise processing, the given implementation found in [11] was adapted and the signal blocking was removed. Also, in the implementation by Drugman, a more adaptive thresholding approach is given, where the statistics of $SRH(f_k)$ are taken into account and the voiced/unvoiced threshold is adapted according to the standard deviation of $SRH(f_k)$. In the algorithm proposed in this project the simpler method with a fixed threshold was chosen. Nevertheless, a smoothing of SRH_{max} over the course of the signal blocks is applied with the help of a first order integrator leading to the following difference equation

$$\overline{SRH}_{\text{max}}[n] = (1 - \alpha) \cdot \overline{SRH}_{\text{max}}[n - 1] + \alpha \cdot SRH_{\text{max}}[n], \tag{3.6}$$

where $\alpha = 0.5$ is a smoothing factor, and n is the time index on block-basis.

With the help of the synthesizer described in chapter 2, sung vocal signals with different vowels, fundamental frequencies and voice qualities were created to analyze the stability of the proposed voiced/unvoiced decision algorithm in a frequency range of $f_0 \in [50\,\text{Hz}, 520\,\text{Hz}]$. The $2\,\text{s}$ long signals were cut in half and $0.5\,\text{s}$ of a zero-mean Gaussian white noise were added in the middle. An exemplary signal is shown in Figure 3.2.

A smoothing factor of $\alpha = 0.5$ was chosen, and the smoothed SRH-values were calculated according to Equation 3.6. In Figure 3.5, the results of the analysis are shown. It can be seen that over the whole frequency range and all analyzed vowels, the voiced regions in the time range of $0\,\text{s} - 1\,\text{s}$ and $1.5\,\text{s} - 2.5\,\text{s}$ lie above the chosen threshold of $\overline{SRH}_{\text{max}}[n] = 0.07$. And for the unvoiced region in the middle, $\overline{SRH}_{\text{max}}[n]$ decreases below the threshold, denoting an unvoiced signal segment.

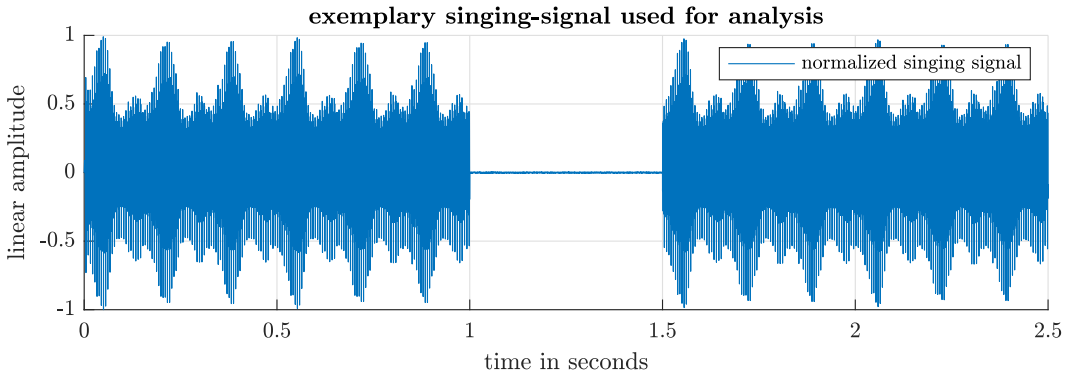


Figure 3.2 *Exemplary signal used for analysis of f_0-estimation and VUV-detection*

The course of one fundamental frequency estimation over time shown in Figure 3.3 corresponds to one synthesized signal for one vowel and one fundamental frequency f_0. The signals are buffered into 80 ms-blocks with a 70 % overlap. The block length was chosen to ensure that a block contains at least 5 periods of the lowest period in our range of interest. Considering a lowest frequency of 70 Hz, 5.6 periods can be contained in one 80 ms long signal block. If the block length is halfed to 40 ms it is not possible to estimate the lowest considered frequency of 70 Hz with the proposed method, as visible in Figure 3.4. The estimated fundamental frequency $\hat{f}_0$ is then calculated for each block with the proposed f_0-estimation, based on the summation of residual harmonics shown in Equation 3.4. The estimated fundamental frequency values for each block of the signals created for the different vowels and fundamental frequency were all plotted into the same figure. As for the stability of the f_0-estimation, it can be seen that the estimated fundamental frequency lies within the range defined by the jitter ratio defined in Table 2.2 for all vowels and all fundamental frequencies. The mentioned range is defined as $\pm$ 1 semitone from $f_{0,\text{true}}$.

During the unvoiced part, no fundamental frequency is estimated (denoted in the implementation with not-a-number (NaN) values). The analysis of the f_0-estimation and the voiced/unvoiced detection were perfomed in `Matlab` and the corresponding code `Main_SRH_Analysis.m` can be found in the folder `V11_Pre_Processing_Analysis/Pitch_Tracker_Comparison/`.

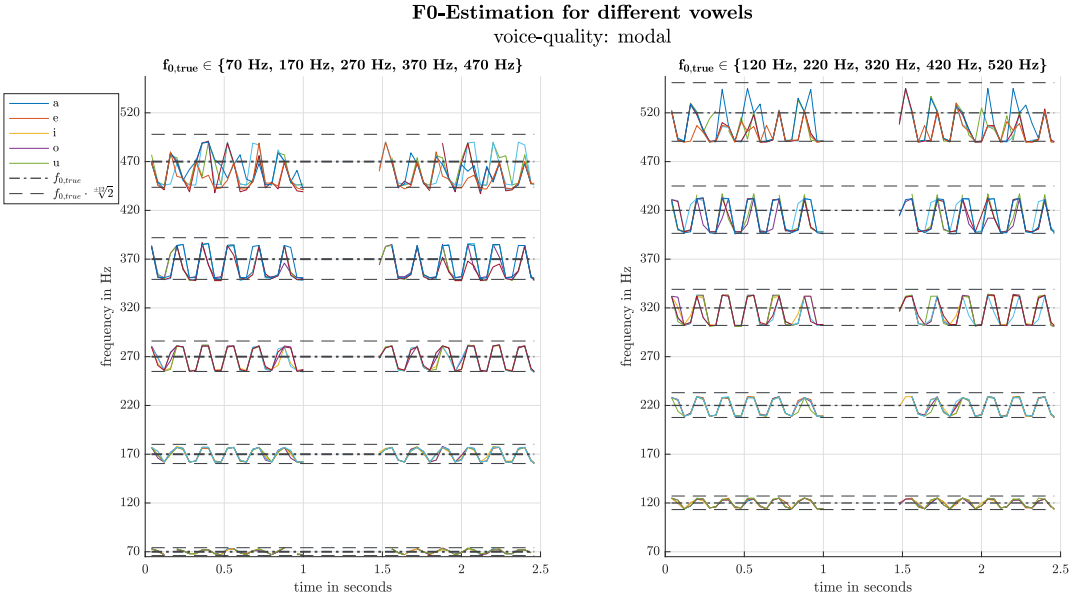


Figure 3.3 *f_0-estimation analysis over frequency for modal voice with a block-length of* 80 ms

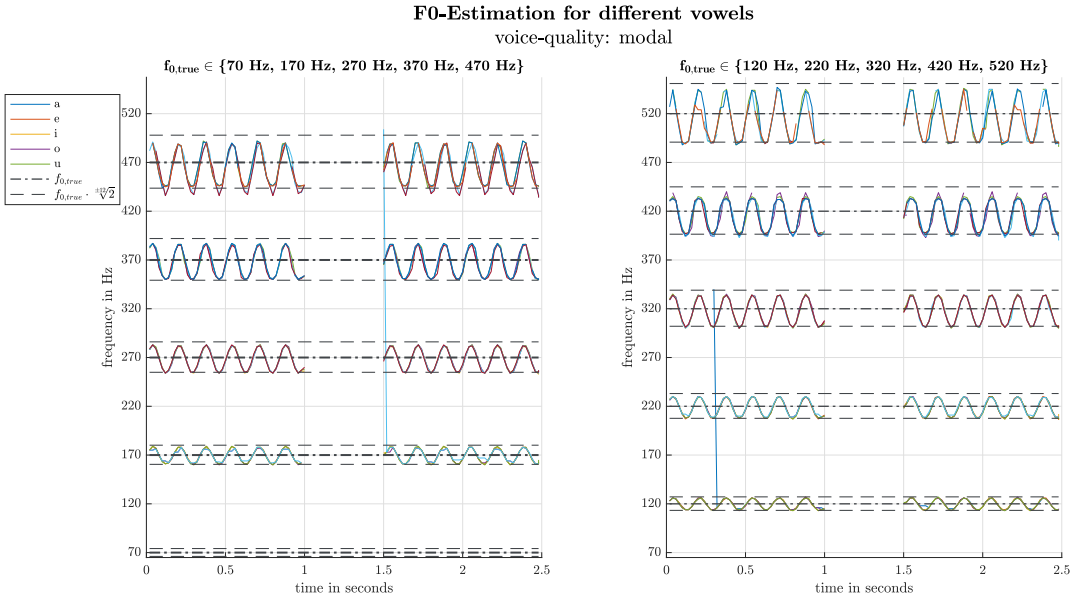


Figure 3.4 *f_0-estimation analysis over frequency for modal voice with a block-length of* 40 ms

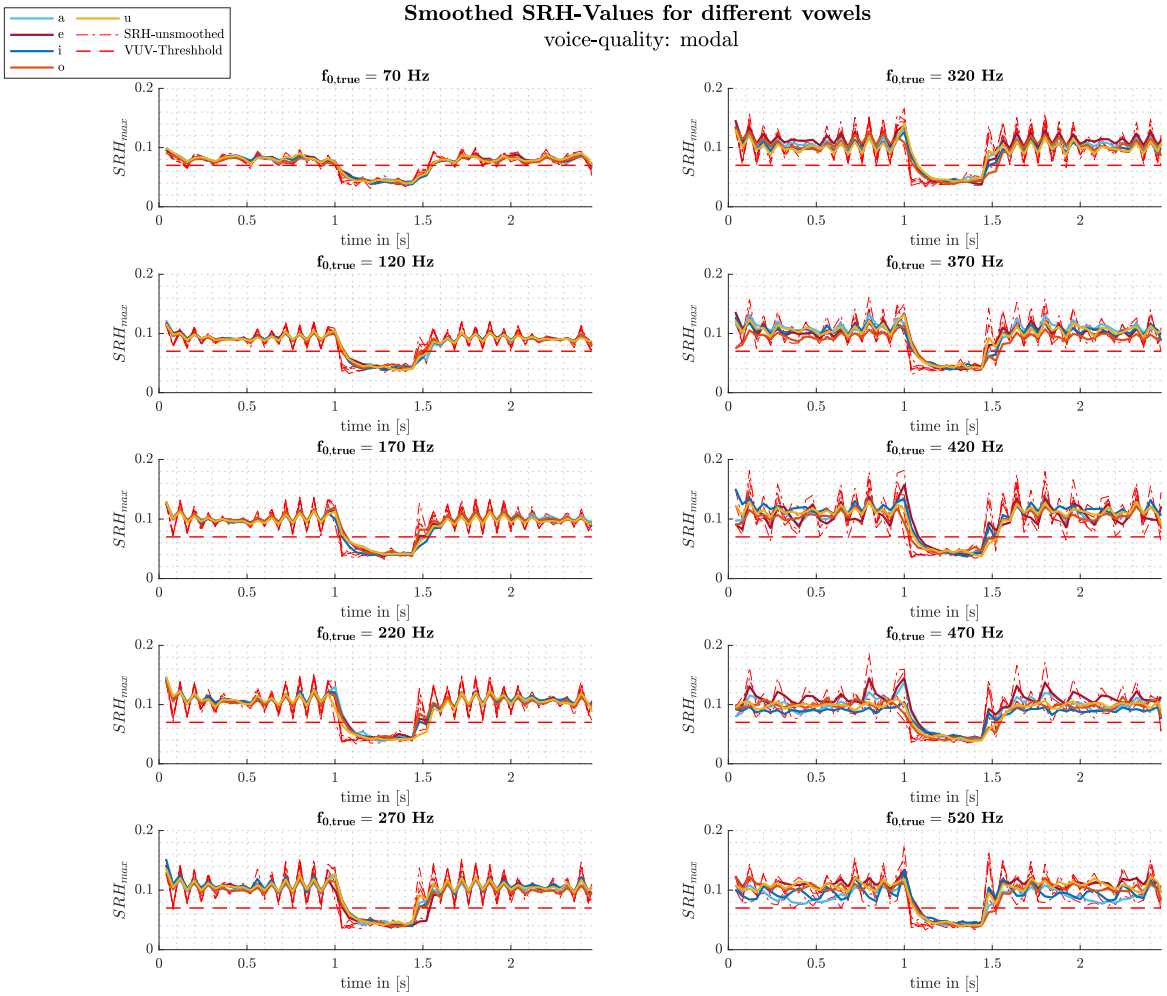


Figure 3.5 *$\overline{SRH}_{max}[n]$ for different vowels, fundamental frequencies and modal voice quality*

As visible in Figure 1.2, $\hat{f}_0$ is needed for the *autocorrelation method with cepstral refinement*, and for the *windowed covariance method*, described in subsections 3.2.2 and 3.2.4, respectively.

3.1.3 Detection of Glottal Opening and Closure Instants

The next pre-processing step, which a voiced sung vocal signal block passes, is the glottal instant (GI) detection. Its purpose is to detect the glottal opening instants (GOI), and the glottal closure instants (GCI) of all glottal cycles included in the signal block. The presented GI detection was put together from two publications. The GCI detection was taken from the SEDREAMS approach discussed in [16], for which an exemplary implementation is published in [11]. The GOI detection has been implemented as proposed in [15]. For both methods the residual signal as well as the so-called mean-based signal are of significant importance.

Computation of the mean-based signal. The mean-based signal can be understood as the output of a 0 Hz oscillator, which means it can be interpreted as a sinusoidal-like signal oscillating at the fundamental frequency of the given speech singal. According to Drugman and Dutoit, the mean-based signal $y_{\text{mean}}[n]$ is computed with [15]

$$\begin{aligned} y_{\text{mean}}[n] &= \frac{1}{2N+1} \sum_{m=-N}^{N} w[m]s[n+m], \\ N &= \left\lfloor \frac{k \cdot f_s}{2 \cdot \hat{f}_0} \right\rceil, \qquad k \in [1.5,\ 2] \end{aligned} \tag{3.7}$$

where $\hat{f}_0$ is the fundamental frequency estimate for the current signal block, $w[n]$ is the window function with a length of $2N + 1$ samples, f_s denotes the current sampling frequency, and $s[n]$ denotes the signal block of the synthesized sung vocal signal. Furthermore, the factor k lies within the range of 1.5 to 2 according to [15], and was chosen to be $k = 1.7$ following the implementation in [13].

As visible in Equation 3.7, the mean-based signal is calculated from a correlation between a window function and the signal block. In accordance to [16], for $w[n]$ a *Blackman*-window is used. The window length is $2N + 1$ and N is calculated from the current signal block's fundamental frequency estimate. If the window-length is chosen too short, multiple unwanted extrema occur, that cause false alarms. A window length chosen too long could lead to oversmoothing, effecting in false alarms concerning the glottal instants [16, p. 4].

In `Matlab` the computation of Equation 3.7 is approximated using the `filter()`-command [45], whereas the array containing the window values is used as numerator coefficients and the window length is used as the sole denominator coefficient of `filter()`.

Figure 3.6 shows a comparison between a sung vocal signal block and its corresponding mean-based signal.

GCI-Detection. The detection of GCIs, as proposed in [16], is executed in two steps. The first step is to derive intervals of presence, where a GCI is expected, based on the mean-based signal. The time intervals discussed in [16] deviate from the intervals calculated in the implemented version in [11]. Due to the good performance of the implemented version found in the repository of [11], its GCI detection procedure was replicated. Therein, a peak finding algorithm is used, which detects all extrema in the mean-based signal. Additionally, it has to be ensured that the first occuring extremum

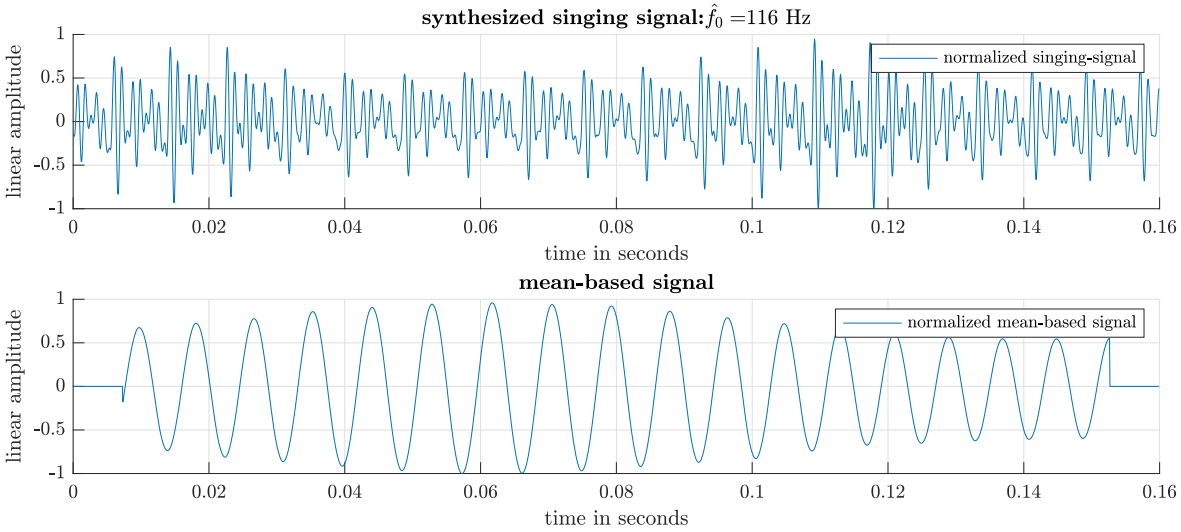


Figure 3.6 *Comparison of a sung vocal signal block and its mean-based signal*

is a minimum and the last a maximum. In `Matlab`, such a peak finder is given with the `findpeaks()`-command [46]. Also, there must be as many minima, as there are maxima. After all, minima and maxima of the mean-based signal are detected within the intervals, in which a GCI is anticipated. The intervals are centered around the minima of the mean-based signal, and their length is given with 0.35 times the fundamental period. For further explanation please note that the array $\boldsymbol{y}_{\text{mean, min}}$ holds the positions of all mean-based minimas in one signal block.

How the intervals are centered around the minimum is part of the second step: the refinement with the help of the residual signal. Each maximum of the residual signal exceeding a threshold of 0.4 is used to calculate a relative position of the GCI inside a glottal cycle.

The set of the residual signal peak-positions is denoted as the vector $\boldsymbol{e}_{\text{peak}}$, whereas a single peak-position is denoted as e_{peak}. The same notation holds for the mean-based extrema. Thus, $\boldsymbol{e}_{\text{peak}}$ can be formulated as

$$\boldsymbol{e}_{\text{peak}} = \{n \in [1, N_{\text{block}}] \ : \ e[n] > 0.4\} \, . \tag{3.8}$$

The `Matlab` equivalent of Equation 3.8 could be achieved with the `find()` or `findpeaks()` command [46]. For each residual peak-position e_{peak}, the position of the mean-based signal's closest minimum is evaluated by determining the minimal distance between each residual peak-position e_{peak} and all mean-based minima positions, denoted as $\boldsymbol{y}_{\text{mean, min}}$, such that

$$y_{\text{mean, min}} = \min \left\{ \left| \boldsymbol{y}_{\text{mean, min}} - e_{\text{peak}} \right| \right\} . \tag{3.9}$$

For each peak in the residual signal, the distance between the closest minimum $y_{\text{mean, min}}$ and its following maximum $y_{\text{mean, max}}$ is calculated. The relative position of a GCI within a glottal cycle λ_{GCI} is then calculated by normalizing the distance between a residual peak-position e_{peak} and the closest mean-based signal minimum $y_{\text{mean, min}}$, with the distance between the minima and maxima closest to the peak. This is done for each detected peak in the residual signal and the median value of the resulting vector is calculated, such that

$$\lambda_{\text{GCI}} = \text{median} \left\{ \frac{(\boldsymbol{e}_{\text{peak}} - \boldsymbol{y}_{\text{mean, min}})}{(\boldsymbol{y}_{\text{mean, max}} - \boldsymbol{y}_{\text{mean, min}})} \right\} . \tag{3.10}$$

Note that equation Equation 3.10 only works if the the residual peak-positions in $\boldsymbol{e}_{\text{peak}}$ and the mean-based extrema-positions in $\boldsymbol{y}_{\text{mean, min}}$ and $\boldsymbol{y}_{\text{mean, max}}$ are sorted in such a away that only the closest

mean-based minima is subtracted from each single residual peak-position, as indicated by Equation 3.9. Also each array indicated by bold notation in Equation 3.10 holds position values and therefore λ_{GCI} results in a median relative position. The median relative position of a GCI inside a glottal cycle λ_{GCI} is then used as a ratio to refine the intervals in which a GCI is expected. In Figure 2.1, the continuous-time instant at which a GCI occurs, is denoted as t_e. Considering a discrete-time calculation basis, the discrete-time instant is written as $n_e = t_e \cdot f_s$, where $n_e \in [n_{e,\text{start}}, n_{e,\text{end}}]$. Using the GCI-ratio λ_{GCI}, which stands in no relation to any time-basis due to its relative nature[1], the relative start and end instances α_{start} and α_{end} of the anticipated GCI time-intervals can be calculated, such that

$$\alpha_{\text{start}} = \lambda_{\text{GCI}} - 0.35 \qquad \text{and} \qquad \alpha_{\text{end}} = \lambda_{\text{GCI}} + 0.35. \tag{3.11}$$

With the information of Equation 3.11, the GCI-intervals can be derived for all minima of the mean-based signal. Firstly, half of the current signal-block's local pitch period in samples $\frac{\boldsymbol{N}_0}{2}$ is calculated by computing the distance between two consecutive extrema, i.e.

$$\frac{\boldsymbol{N}_0}{2} = \boldsymbol{y}_{\text{mean, max}} - \boldsymbol{y}_{\text{mean, min}}. \tag{3.12}$$

Again, bold notation indicates that the calculations in the implementation are executed multiple times for all remaining extrema, leading to multiple local pitch period estimates in one block, denoted as $\boldsymbol{N}_0$. With the relative start and end instances, α_{start} and α_{end} respectively, and the local half pitch-period, the starting and end instances for the GCI-intervals in the signal block can be calculated using

$$\begin{aligned} \boldsymbol{n}_{e,\text{start}} &= \boldsymbol{y}_{\text{mean, min}} - \left\lfloor \alpha_{\text{start}} \cdot \frac{\boldsymbol{N}_0}{2} \right\rceil \\ \boldsymbol{n}_{e,\text{end}} &= \boldsymbol{y}_{\text{mean, min}} - \left\lfloor \alpha_{\text{end}} \cdot \frac{\boldsymbol{N}_0}{2} \right\rceil . \end{aligned} \tag{3.13}$$

Within the all intervals defined by $\boldsymbol{n}_{e,\text{start}}$ and $\boldsymbol{n}_{e,\text{end}}$, the position of the residual signal's maximum amplitude are evaluated, delivering the GCI estimates $\hat{\boldsymbol{n}}_e$ in one block, such that

$$\begin{aligned} \hat{n}_e^i &= \underset{n}{\text{argmax}} \left(e[n] \right) \Big|_{n_{e,\text{start}}^i}^{n_{e,\text{end}}^i} \\ \forall\, i &= 1, 2, \ldots, N_{\text{GCI}} = \#\left(\text{GCI-intervals}\right) \\ \hat{\boldsymbol{n}}_e &= \left\{ \hat{n}_e^1, \hat{n}_e^2, \ldots, \hat{n}_e^{N_{\text{GCI}}} \right\} \end{aligned} \tag{3.14}$$

The $\#(\cdot)$ operator returns the number of elements in an array. It can be viewed as the mathematical notation of the matlab command `length()` [50]. The GCI-Detection process is summed up in Figure 3.8. The intervals the GCIs are anticipated in, centered around the mean-based minima, are shown in the second subplot. The location of the intervals and their relation to the residual signal is shown in the third subplot. The fourth subplot of Figure 3.8 shows an exemplary estimation of the GCIs, as the estimated GCIs coincide with the true GCIs located at the negative maximum amplitude of the dGf.

GOI-Detection. For the GOIs, the position estimation is also executed in presence intervals where a GOI is anticipated. In [15], the presence intervals are determined to start at a maximum and end

[1] In fact, it is dimensionless.

at the next negative zero-crossing of the mean-based signal. The comparison with the ground truth showed, that for the evaluation of the synthesized sung vocal signal the interval has to be chosen between the positive zero-crossing and the next maximum, rather than the proposed interval of [15]. The start instances at the positive zero-crossing are denoted with $\boldsymbol{n}_{0,\text{start}}$, and the end of the intervals with $\boldsymbol{n}_{0,\text{end}}$. Additionally, a guarding margin of $0.25\,\text{ms}$ is proposed to ensure that the GOI is placed within the anticipation interval. In order to achieve a better adaption towards sung vocal signals the guarding margin is modified: Considering a frequency modulation of $\pm$ 1 semitone, as a consequence to the vibrato defined in Table 2.2, the GOIs in the signal also move within the vicinity of $\pm$ 1 semitone. This leads to a relation between the guarding margin's length N_{guard} and the current fundamental frequency estimate $\hat{f}_0$, i.e.

$$N_{\text{guard}} = \left(\frac{1}{\hat{f}_0 \cdot 2^{-\frac{1}{12}}} - \frac{1}{\hat{f}_0 \cdot 2^{\frac{1}{12}}} \right) \cdot f_s. \tag{3.15}$$

To compute the intervals, $\left\lfloor \frac{N_{\text{guard}}}{2} \right\rceil$ is added to $n_{0,\text{end}}$, and the other half is subtracted from $n_{0,\text{start}}$. The estimated discrete-time GOI position $\hat{n}_0$ is calculated by evaluating the residual signal amplitude maximum's position in the calculated interval range, similiar to the GCI estimation described earlier.

$$\begin{aligned} \hat{n}_0^i &= \underset{n}{\text{argmax}} \left(e[n] \right) \Big|_{n_{0,\text{start}}^i - \left\lfloor \frac{N_{\text{guard}}}{2} \right\rceil}^{n_{0,\text{end}}^i + \left\lfloor \frac{N_{\text{guard}}}{2} \right\rceil} \\ &\forall\, i = 1, 2, \ldots, N_{\text{GOI}} = \#\,(\text{GOI-intervals}) \\ \hat{\boldsymbol{n}}_0 &= \left\{ \hat{n}_0^1, \hat{n}_0^2, \ldots, \hat{n}_0^{N_{\text{GOI}}} \right\} \end{aligned} \tag{3.16}$$

In the second subplot of Figure 3.8 the modified intervals between the positive zero-crossing and the next maximum are visible. The interval's placement around the GOI is visible when comparing the GOI-interval and the true dGF. It becomes visible that the interval is centered around the GOI.

Performance Evaluation of GI Detection. When looking at the fourth subplot of Figure 3.8 it becomes visible, that the GCI detection is much more accurate than the GOI detection. In order to analyze the performance of the GI-detection, sung vocal signals with different vowels, fundamental frequencies and voice qualities were synthesized and the GI-detection was evaluated. To quantify the performance for different parameter sets, two percentage measures were evaluated. The number of the true glottal instants (known from the groundtruth dGf) located within the found glottal instant intervals was set into relation with the overall number of the true glottal instants. This results in the percentage measure $P_{\text{GI,1}}$ which is calculatable for both glottal instants (GOIs and GCIs).

$$P_{\text{GI,1}} = \frac{\#\,(\text{true GIs in GI-intervals})}{\#(\text{true GIs})} \tag{3.17}$$

If for instance an interval extends over a whole signal-block the percentage measure $P_{\text{GI,1}}$ would still result in a high percentage. This shows that $P_{\text{GI,1}}$ alone is a non-informative measure. In order to make statements on the GI-Detection's performance, a second percentage measure $P_{\text{GI,2}}$, comparing the number of intervals with the number of the true glottal instants is introduced, such that

$$P_{\text{GI,2}} = \frac{\#\,(\text{GI-intervals})}{\#\,(\text{true GIs})}. \tag{3.18}$$

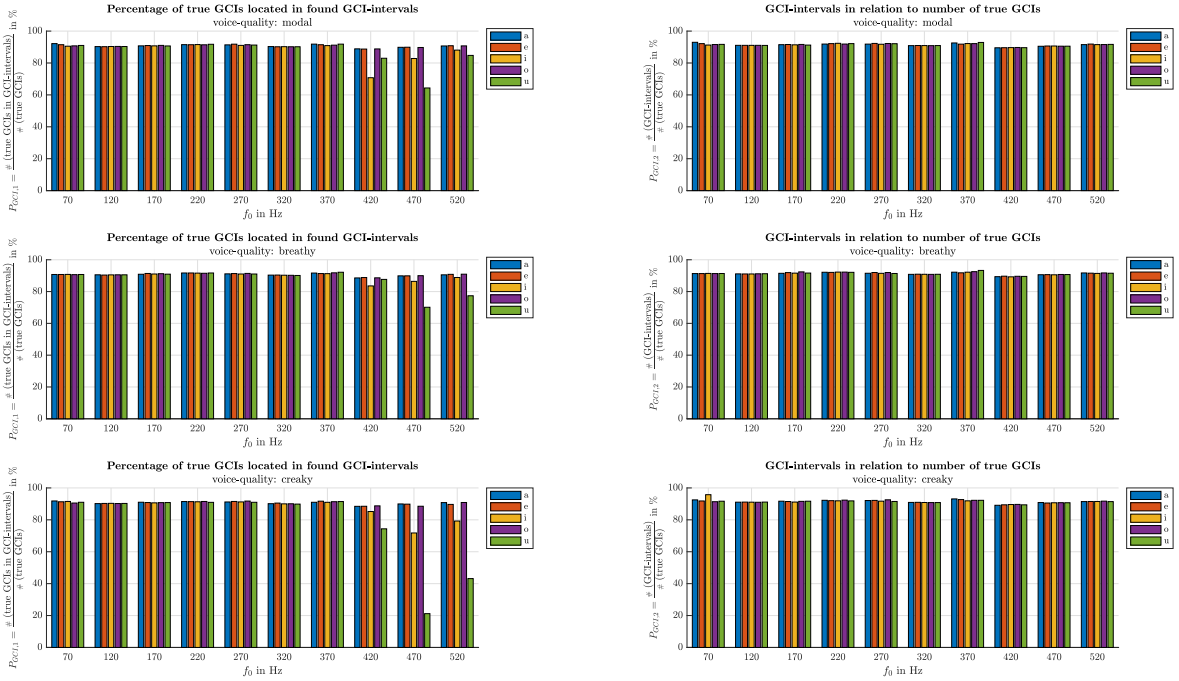
The performance measures are calculated for all GCIs and GOIs, which is indicated by their index notation. The previously mentioned signals used to evaluate $P_{\mathrm{GI},1}$ and $P_{\mathrm{GI},2}$ were created in such a way, that the signal length depends on the fundamental frequency f_0, ensuring that each signal realization contains the same amount of glottal instants. In Figure 3.7, the percentage measures concerning the GCI-and GOI-measures are visualized. The subplots of the left column show the evaluation of $P_{\mathrm{GI},1}$ and the subplots in the right column compare $P_{\mathrm{GI},2}$ for different fundamental frequencies f_0, different vowels (*/a/*, */e/*, */i/*, */o/* and */u/*) and voice qualities (modal, breathy and creaky).

For both glottal instants the high percentage of $P_{\mathrm{GI},2}$ indicates, that the number of found intervals coincides with the number of true GIs. The reason why $P_{\mathrm{GI},2}$ does not reach $100\,\%$ can be found in the zero-padded beginning and end of the mean-based signal. Due to the zero-padding, which is necessary to conceal filtering artifacts, the first and last glottal instants of each signal block are not detected correctly, leading to a systematic error of arround $10\,\%$. Concerning $P_{\mathrm{GI},1}$, Figure 3.7 shows, that for certain vowels and higher fundamental frequencies the presence intervals determined by the mean-based signal analysis are misplaced for all three voice-qualities. Especially the GOI-intervals are wrongly placed, when the signals are synthesized with $f_0 = 370\,\mathrm{Hz}$ or higher.

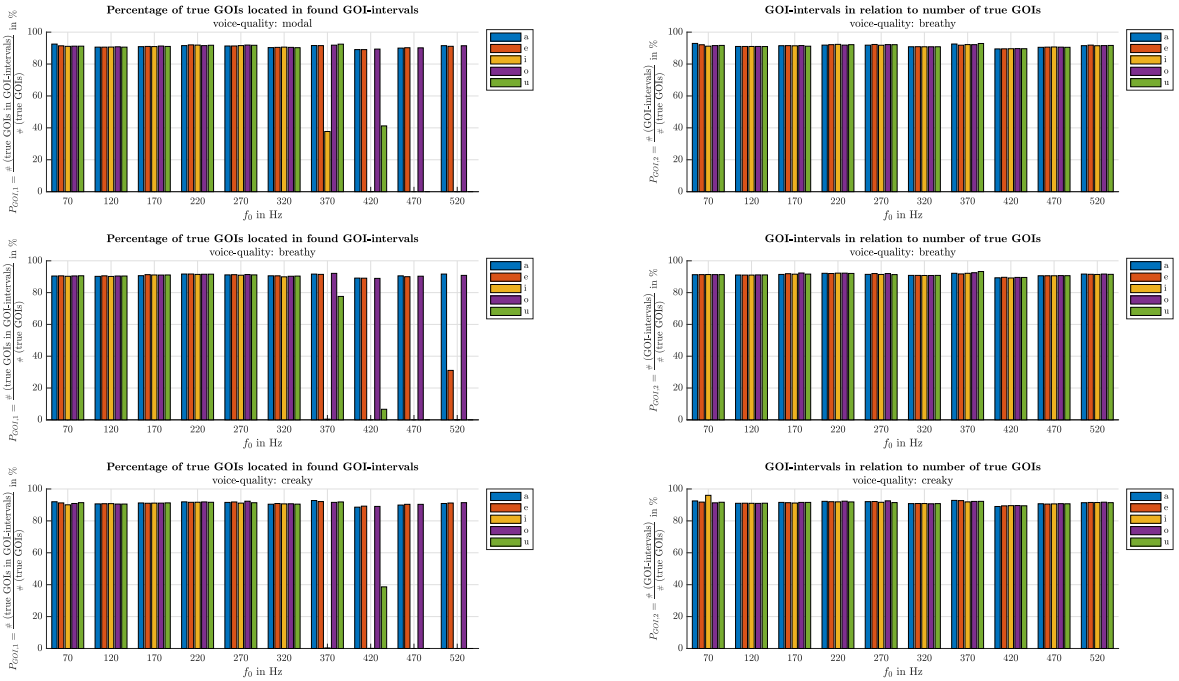
In addition, for signals with vowels whose first formant is placed at relatively low frequencies (that are */i/* and */u/*), the detection reaches its limits, when the fundamental frequency or its first harmonics reach the vicinity of the first formant. This leads to a faulty rough estimation of the vocal tract filter, which is necessary for the computation of the residual signal mentioned in subsection 3.1.1. Consequently, this might entail a incorrect refinement using the residual signal when estimating the GCI-intervals.

The placement of the GOI intervals within the signal block only depends on the positions of the positive zero crossings and the maxima of the mean-based signal. As there is no refinement using the residual signal, it can also be assumed that the GI-interval estimation based on the mean-based signal reaches its limits for higher fundamental frequencies. In Figure 3.9 a worst case example is shown. The synthesized signal block contains the sung vowel */i/* and its fundamental frequency is given with $f_0 = 470\,\mathrm{Hz}$. It is visible, that the GCI-intervals are still placed in close vicinity to the true GCIs, whereas the GOI-intervals are displaced, leading to wrong estimated glottal opening instants, which reflects the results obtained in subsection 3.3.4.

The GCI- and GOI-detection is needed for the window computation used in the *windowed covariance method* described in subsection 3.2.4. Also the GCIs are a vital part of the *voice quality classification* based on the skewness features described in section 3.3.



(a) GCI detection measures



(b) GOI detection measures

Figure 3.7 *Glottal instants performance measures in dependence on f_0, vowels and voice-quality*

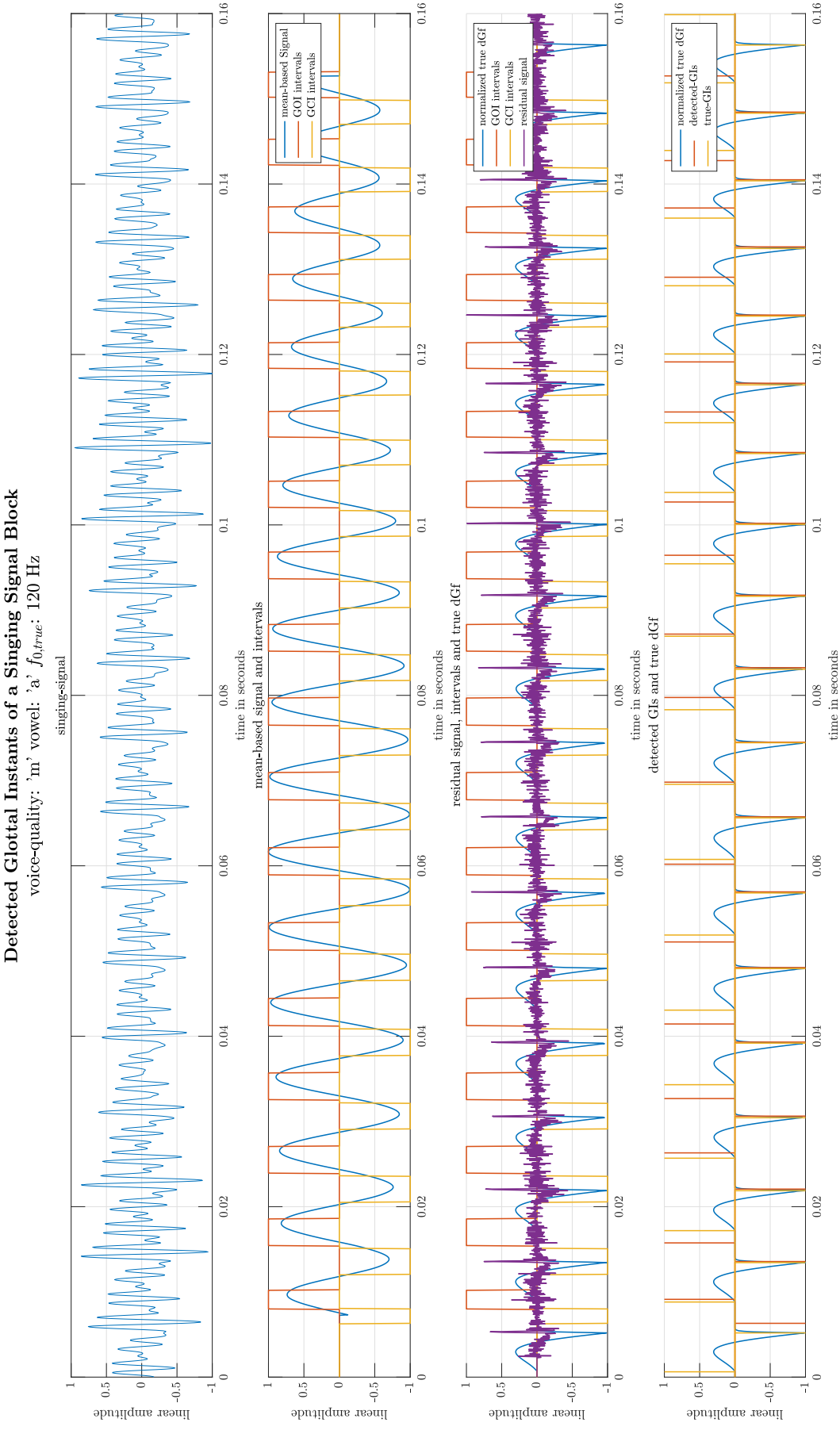


Figure 3.8 *Detected GIs, mean-based signal and anticipated GI-intervals for /a/ with $f_0 = 120$ Hz and modal voice quality*

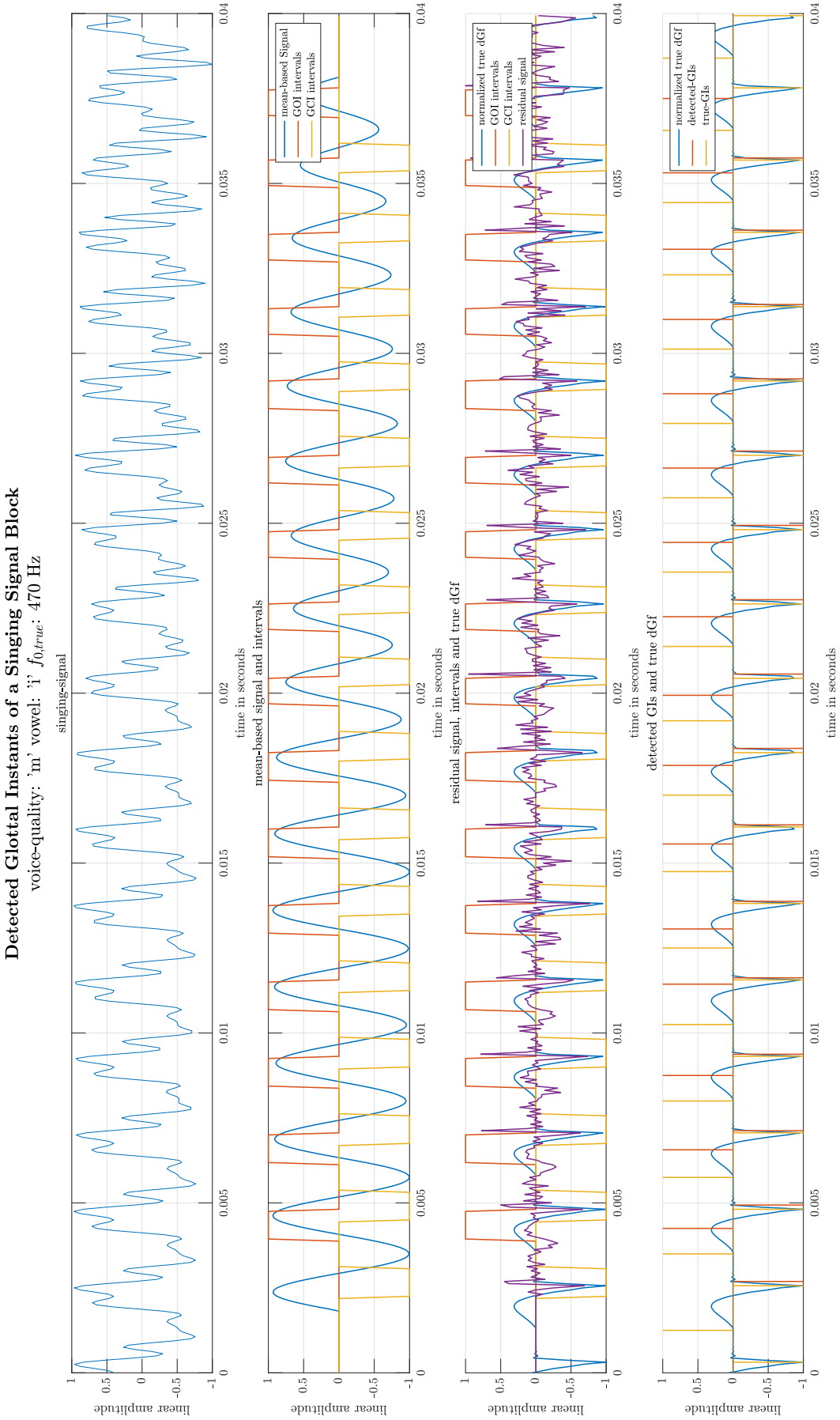


Figure 3.9 *Detected GIs, mean-based signal and anticipated GI-intervals for /i/ with $f_0 = 470\,\mathrm{Hz}$ and modal voice quality. The GOI detection does not work with a satisfying amount of accuracy due to the bad constellation of f_0 and the formant frequencies of /i/.*

3.1.4 Pre-Emphasis Filtering

The last pre-processing step before the vocal tract filter estimation with linear prediction is executed, is the pre-emphasis filtering. The goal of the pre-emphasis filtering is to "whiten" the sung vocal signal, meaning that we aim for a flatter spectrum of the input signal. The whitening helps to regularize the linear prediction process, as it was designed for white noise signals. As proposed in [67, p.200], a very simple first order IIR high-pass-filter is chosen for the pre-emphasis filter. The transfer-function of the pre-emphasis filter is given with

$$\begin{aligned} H_{\text{pre}}(z) &= 1 - \alpha_{\text{pre}} \cdot z^{-1} \\ \alpha_{\text{pre}} &= e^{-\frac{2\pi \cdot f_{\text{pre}}}{f_s}}. \end{aligned} \tag{3.19}$$

For the implementation $f_{\text{pre}} = 10$ was chosen, with the downsampled sampling frequency $f_s = 16\,000\,\text{kHz}$ this leads to $\alpha_{\text{pre}} = 0.9961$.

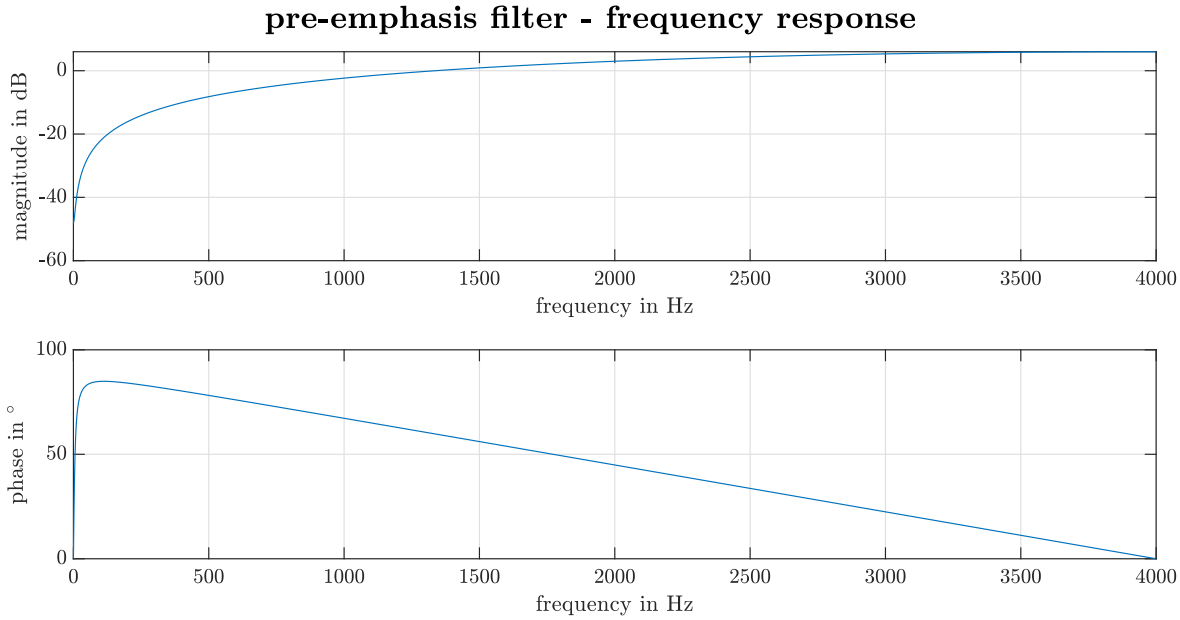


Figure 3.10 *Frequency response of the pre-emphasis filter*

The pre-emphasis filtering is the last processing step a signal block runs through before it is processed by the linear prediction methods described in the following section 3.2.

3.2 Vocal Tract Filter Esimation using Linear Prediction

With the information on the fundamental frequency $\hat{f}_0$ and the position of the glottal instants ($\hat{n}_e$ and $\hat{n}_0$), a more detailed linear prediction analysis is performed on the pre-emphasis filtered sung vocal signal. Before the different methods are thoroughly explained, a general theoretical introduction into *linear prediction* is given.

It was already mentioned in chapter 2, that the underlying model for speech signals is the *source-filter model*, where the filter in this case represents the vocal tract filter. It is modeled as an all-pole filter with p poles showing the following transfer function:

$$H(z) = \frac{G}{1 - \sum_{k=1}^{p} a_k z^{-k}} = \frac{S(z)}{V(z)} \tag{3.20}$$

The integer p is also called the *linear prediction order*, and defines the amount of filter coefficients calculated in order to estimate the vocal tract filter in all methods discussed in the following subsections. The choosing of the linear prediction order still remains a subject open for discussion, as there are several approaches e.g. [69] propose a heuristic method using a "reflection coefficient cutoff", whereas in [31] a LP order dependent on the fundamental frequency is proposed. Nevertheless, for the analysis executed during this project the same order as used by Degottex *et al.* in the implementations of [11] was used which corresponds to the rough LP order mentioned in Equation 3.1.

Based on Equation 3.20 and the assumption of an input signal $v[n]$, a linear difference equation with constant coefficients can be formulated by inverse z-transform [56, p. 934], such that

$$\begin{aligned} \mathfrak{Z}^{-1}\left\{ S(z) - S(z)\left(\sum_{k=1}^{p} a_k z^{-k}\right) + GV(z) \right\} &= s[n] - \sum_{k=1}^{p} a_k s[n-k] + Gv[n]. \\ s[n] &= \sum_{k=1}^{p} a_k s[n-k] + Gv[n] \end{aligned} \tag{3.21}$$

Equation 3.21 allows the following interpretation: a signal prediction $\hat{s}[n]$, calculated via a linear combination of coefficients a_k and past signal samples $s[n-k]$, is subtracted from the current signal sample $s[n]$, resulting in the prediction error $e_{\text{LP}}[n]$. Thus, the relation can be rewritten such that

$$\begin{aligned} s[n] &= \underbrace{\sum_{k=1}^{p} a_k s[n-k]}_{\hat{s}[n]} + \underbrace{Gv[n]}_{e_{\text{LP}}[n]}. \\ e_{\text{LP}}[n] &= s[n] - \hat{s}[n] = s[n] - \sum_{k=1}^{p} a_k s[n-k] \end{aligned} \tag{3.22}$$

With the assumption, that the zeroth coefficient is one ($a_0 = 1$) the sum forming $\hat{s}[n]$ in Equation 3.22 can be extended for $k = 0$, leading to the following equation for the error $e_{\text{LP}}[n]$,

$$e_{\text{LP}}[n] = s[n] - \sum_{k=1}^{p} a_k s[n-k] = \sum_{k=0}^{p} a_k s[n-k], \text{ with } a_0 = 1. \tag{3.23}$$

For the error $e_{\text{LP}}[n]$, a *minimum mean square error* (MMSE) problem can be set up. Let $\boldsymbol{a}$ denote a vector containing the filter coefficients $a_k \; \forall \; k = 1, \ldots, p$, the cost function $J_{\text{MSE}}(\boldsymbol{a})$ for such a

problem is given with

$$J_{\text{MSE}}(\boldsymbol{a}) = \mathbb{E}\Big\{e_{\text{LP}}^2[n]\Big\} = \mathbb{E}\left\{\left(\sum_{k=0}^{p} a_k s[n-k]\right)^2\right\}$$
$$J_{\text{MSE}}(\boldsymbol{a}) = \mathbb{E}\left\{\left(\sum_{i=0}^{p} a_i s[n-i]\right)\left(\sum_{j=0}^{p} a_j s[n-j]\right)\right\} = \mathbb{E}\left\{\sum_{i=0}^{p}\sum_{j=0}^{p} a_i s[n-i]s[n-j]a_j\right\} \tag{3.24}$$

As shown in [38, p.10], the expectation operator $\mathbb{E}\{\cdot\}$ can also be viewed as an averaging over time inside the interval defined by n_0 and n_1. Therefore, if the expectation operator is rewritten as a sum, the cost function $J_{\text{MSE}}(\boldsymbol{a})$ is obtained, such that

$$J_{\text{MSE}}(\boldsymbol{a}) = \frac{1}{|n_1 - n_0|}\left[\sum_{n=n_0}^{n_1}\sum_{i=0}^{p}\sum_{j=0}^{p} a_i s[n-i]s[n-j]a_j\right]. \tag{3.25}$$

Due to the independence of a_i and a_j with respect to n, the sum operations can be interchanged. Thus

$$J_{\text{MSE}}(\boldsymbol{a}) = \frac{1}{|n_1 - n_0|}\left[\sum_{i=0}^{p}\sum_{j=0}^{p} a_i a_j \underbrace{\left(\sum_{n=n_0}^{n_1} s[n-i]s[n-j]\right)}_{\phi_{ij}}\right]. \tag{3.26}$$

The term ϕ_{ij} describes a *covariance function*. It will later be important, concerning the distinction between the *autocorrelation method* and *covariance method* for linear prediction. Using the notation ϕ_{ij}, the cost function can further be simpliefied to [38, eq.2.9]

$$J_{\text{MSE}}(\boldsymbol{a}) = \frac{1}{|n_1 - n_0|}\left[\sum_{i=0}^{p}\sum_{j=0}^{p} a_i a_j \phi_{ij}\right] = \frac{1}{|n_1 - n_0|}\left[\sum_{i=0}^{p}\sum_{j=0}^{p} a_i \phi_{ij} a_j\right]. \tag{3.27}$$

In order to calculate to calculate the coefficients which minimize the given cost function $J_{\text{MSE}}(\boldsymbol{a})$, the corresponding derivative with respect to the coefficients a_j is calculated and set to zero [38, eq.2.11], such that

$$\frac{\partial J_{\text{MSE}}(\boldsymbol{a})}{\partial a_j} = \frac{2}{|n_1 - n_0|}\left[\sum_{i=0}^{p} a_i \phi_{ij}\right] \stackrel{!}{=} 0.$$
$$\sum_{i=0}^{p} a_i \phi_{ij} \stackrel{!}{=} 0 \qquad \text{with} \qquad a_0 = 1 \tag{3.28}$$
$$\sum_{i=1}^{p} a_i \phi_{ij} + \phi_{0j} = 0$$

Thus, the following system of equations for the coefficients a_i is reached.

$$\sum_{i=1}^{p} a_i \phi_{ij} = -\phi_{0j} \qquad \text{where} \qquad j = 1, 2, \ldots, p \tag{3.29}$$

If Equation 3.29 is solved for the coefficients a_i, the cost function is minimized, and the obtained filter coefficients can be used as the estimate for the all-pole filter's coefficients mentioned in Equation 3.20. To solve Equation 3.29, ϕ_{ij} has to be calculated. It is given with:

$$\phi_{ij} = \sum_{n=n_0}^{n_1} s[n-i]s[n-j] \tag{3.30}$$

The averaging interval's range, defined through n_0 and n_1, defines which optimization method is applied. There are two basic methods concerning the size of the interval, leading to the distinction between *autocorrelation* or *covariance* methods for linear prediction. [38, p.11]

Estimation Error and Filter Gain. If one takes a closer look at the relations established in Equation 3.22, the estimation error $e_{\text{LP}}[n]$ is given with

$$e_{\text{LP}}[n] = Gv[n]. \tag{3.31}$$

The mean square error $J_{\text{MSE}}(\boldsymbol{a})$ is then given with

$$J_{\text{MSE}}(\boldsymbol{a}) = \mathbb{E}\Big\{e_{\text{LP}}^2[n]\Big\} = \mathbb{E}\Big\{G^2v^2[n]\Big\} = G^2\mathbb{E}\Big\{v^2[n]\Big\}. \tag{3.32}$$

Ideally, $v[n]$ is deemed to be zero-mean, Gaussian white noise with unit variance, as discussed in [71, p.230]. This would result in $\mathbb{E}\{v^2[n]\} = \sigma_v^2 = 1$, and thus the filter gain could be exactly estimated by taking the square root of the remaining prediction error, such that

$$G = \frac{\sqrt{\mathbb{E}\{e_{\text{LP}}^2\}}}{\sqrt{\mathbb{E}\{v^2[n]\}}} = \frac{\sqrt{\mathbb{E}\{e_{\text{LP}}^2\}}}{\sigma_v^2} = \sqrt{\mathbb{E}\{e_{\text{LP}}^2\}}. \tag{3.33}$$

Nevertheless, the underlying input-signal modeled by the LF-model defined in section 2.1 is *not* a white noise process and therefore the estimated filter gain $\hat{G}$ is always erroneous. In fact, the filter gain error is defined by the term $\sqrt{\mathbb{E}\{v^2[n]\}}$, such that

$$\hat{G} = G\sqrt{\mathbb{E}\{v^2[n]\}} = \sqrt{\mathbb{E}\{e_{\text{LP}}^2\}}. \tag{3.34}$$

Thus, for the given excitation signals, synthesized by the proposed vocal signal synthesis algorithm of chapter 2, the gain estimation using the remaining prediction error, *can not* be an exact estimation of G.

This behavior also explains the reason for the implemented whitening process using the pre-emphasis filter mentioned in subsection 3.1.4. The filter estimation using linear prediction works best, if Gaussian white noise input signals are used. Unfortunately, Gaussian noise processes are not realistic excitation signals for voiced speech/vocal signals. As this is the only available method to estimate the gain with linear prediction, the estimated gain $\hat{G}$ calculated from the remaining prediction error's square root is used in all implemented methods. In the following subsections, the four linear prediction methods used in the analysis are discussed.

3.2.1 Autocorrelation Method

The autocorrelation method is given, if the averaging interval mentioned in Equation 3.29 ranges from $n_0 = -\infty$ to $n_1 = \infty$. Naturally, in practical applications the infinite time interval is limited with

$n \in [0, N-1]$, where N is the number of available samples. Therefore, we use the following relation to calculate ϕ_{ij}

$$\phi_{ij} = \sum_{n=-\infty}^{\infty} s[n-i]s[n-j] \approx \sum_{n=0}^{N-1} s[n-i]s[n-j] \tag{3.35}$$

The limits of the sum can be manipulated in such a way, that the covariance function ϕ_{ij} becomes the autocorrelation function $r_{ss}[|i-j|]$, as it can be found in [38, eq. 2.12]. Thus,

$$\phi_{ij} = \sum_{n=0}^{N-1} s[n-i]s[n-j] = \sum_{n=0}^{N-1-|i-j|} s[n]s[n+|i-j|] = r_{ss}[|i-j|]. \tag{3.36}$$

In order to calculate an exact autocorrelation function, infinitely many signal samples would be necessary. The limitations due to finite intervals lead to a remaining prediction error, additional to the error inflicted by the non-white input signal as described in Equation 3.32 [56, p. 942-944].

Therefore, the calculation of the coefficients minimizing the cost function are to be interpreted as an *estimation* and the coefficient's notation changes from a_i to $\hat{a}_i$, where the hat symbol denotes the estimation. For the minimization equation shown in Equation 3.29, this results in

$$\sum_{i=1}^{p} \hat{a}_i r_{ss}[|i-j|] = -r_{ss}[|j|], \qquad j = 1, 2, \ldots, p. \tag{3.37}$$

Using vector-matrix notation, the sum of Equation 3.37 can be written as an inner product, such that

$$\begin{bmatrix} r_{ss}[|1-j|] & r_{ss}[|2-j|] & \ldots & r_{ss}[|p-j|] \end{bmatrix} \begin{pmatrix} \hat{a}_1 \\ \hat{a}_2 \\ \vdots \\ \hat{a}_p \end{pmatrix} = -r_{ss}[|j|], \qquad j = 1, 2, \ldots, p \tag{3.38}$$

This vector operation has to be executed for all $j = 1, 2, \ldots, p$, thus the equation system can be formulated as a matrix vector multiplication:

$$\underbrace{\begin{bmatrix} r_{ss}[0] & r_{ss}[1] & r_{ss}[2] & \ldots & r_{ss}[p-1] \\ r_{ss}[1] & r_{ss}[0] & r_{ss}[1] & \ldots & r_{ss}[p-2] \\ r_{ss}[2] & r_{ss}[1] & r_{ss}[0] & \ldots & r_{ss}[p-3] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{ss}[p-1] & r_{ss}[p-2] & r_{ss}[p-3] & \ldots & r_{ss}[0] \end{bmatrix}}_{\boldsymbol{R}_{ss}} \underbrace{\begin{pmatrix} \hat{a}_1 \\ \hat{a}_2 \\ \hat{a}_3 \\ \vdots \\ \hat{a}_p \end{pmatrix}}_{\hat{\boldsymbol{a}}_{\text{opt}}} = - \underbrace{\begin{pmatrix} r_{ss}[1] \\ r_{ss}[2] \\ r_{ss}[3] \\ \vdots \\ r_{ss}[p] \end{pmatrix}}_{\boldsymbol{r}_{ss+1}} \tag{3.39}$$

Here it is obvious, that $\boldsymbol{R}_{ss}$ is an autocorrelation matrix. Due to the fact, that only real-valued input signals are considered, the autocorrelation matrix is symmetric and it shows Toeplitz structure. These equations are known as the *Yule-Walker* equations. The vector $\boldsymbol{r}_{ss+1}$ can be interpreted as an autocorrelation vector with a lag of one. Assuming that the symmetric matrix $\boldsymbol{R}_{ss}$ is invertible, the equations can be solved for the coefficients with the following relation. [56, p.938]

$$\hat{\boldsymbol{a}}_{\text{opt}} = \boldsymbol{R}_{ss}^{-1} \boldsymbol{r}_{ss+1} \tag{3.40}$$

An efficient algorithm enabling the solution of this equation system was proposed by Trench in [68], called the *Levinson-Durbin* algorithm. The iterative implementation of this algorithm is well and

extensively documented and a renowned way to solve the Yule-Walker equations [38, p. 12]. Due to the considerable documentation [68], [38, p. 12] or [70, p. 182] and already existing implementations, e.g. in form of the `Matlab` command `levinson()` [51], no further discussion of the Levinson-Durbin algorithm is carried out.

Implementation in `Matlab`. In the implementation proposed in this project, the autocorrelation function is calculated via the frequency domain. Firstly, a voiced signal block is filtered with the pre-empahsis filter mentioned in subsection 3.1.4. The pre-emphasis filtered input signal is denoted with $s[n]$. Afterwards, the signal is windowed with a Hann-window $w[n]$ resulting in $\tilde{s}[n] = s[n] \cdot w[n]$, and finally the autocorrelation function $r_{ss}[m]$ is calculated by squaring the magnitude in the frequency domain, as shown in the following Equation 3.41.

$$
\begin{aligned}
N_{\text{block}} &= \lfloor t_{\text{block}} \cdot f_s \rceil \qquad \text{with} \qquad t_{\text{block}} = 80\,\text{ms} \\
N_{\text{DFT}} &= 2^{\lfloor \log_2(N_{\text{block}})+1 \rceil} \quad \ldots \text{ number of DFT samples} \\
W &= \frac{\sum_{n=0}^{N_{\text{block}}} w[n]}{N_{\text{block}}} \\
r_{ss}[m] &= \mathcal{F}^{-1}_{k \mapsto m} \left\{ \left| \frac{\mathcal{F}_{n \mapsto k}\{\tilde{s}[n]\}[k]}{W} \right|^2 \right\}[m] \qquad \text{with} \qquad m \in [0, N_{\text{DFT}} - 1],
\end{aligned}
\tag{3.41}
$$

where W denotes a correction factor necessary due to the windowing. Due to the fact that $N_{\text{DFT}} > N_{\text{block}}$, the lag index of the autocorrelation function $r_{ss}[m]$ is denoted with m, rather than the original time index n. The autocorrelation $r_{ss}[m]$ of the current signal block is then used to execute the Levinson-Durbin recursion which solves Equation 3.40 to obtain the estimated optimal all-pole vocal tract filter coefficients $\hat{\boldsymbol{a}}_{\text{opt}}$ for this signal block.

Theoretically, the autocorrelation would only be exact, if there were infinitely many signal samples available. Nevertheless, due to the occurrence of finite length signals in practical implementations, there remains a prediction error for the autocorrelation method [56, p. 942-944]. The remaining prediction error is used to estimate the filter gain with equation 3.34.

The calculation of the autocorrelation is executed in the `Matlab` script `calcAutoCorr.m`. The transformation into the frequency domain is carried out with the `fft()` command [43].

To summarize the autocorrelation method, its signal flow leading to the estimated poles of the VT filter is visualized in Figure 3.11. Figure 3.12 shows an exemplary signal block and the corresponding autocorrelation function. In `Matlab` it has to be ensured that the autocorrelation values corresponding to lag zero are placed in the center of the array as shown in figure Figure 3.12.

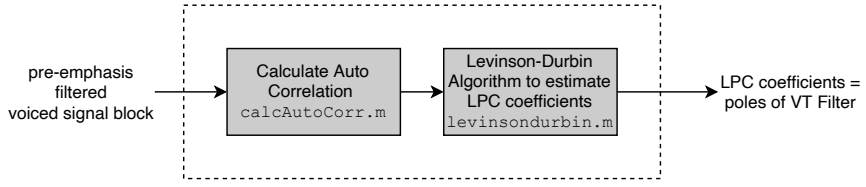


Figure 3.11 *Processing steps of the autocorrelation method*

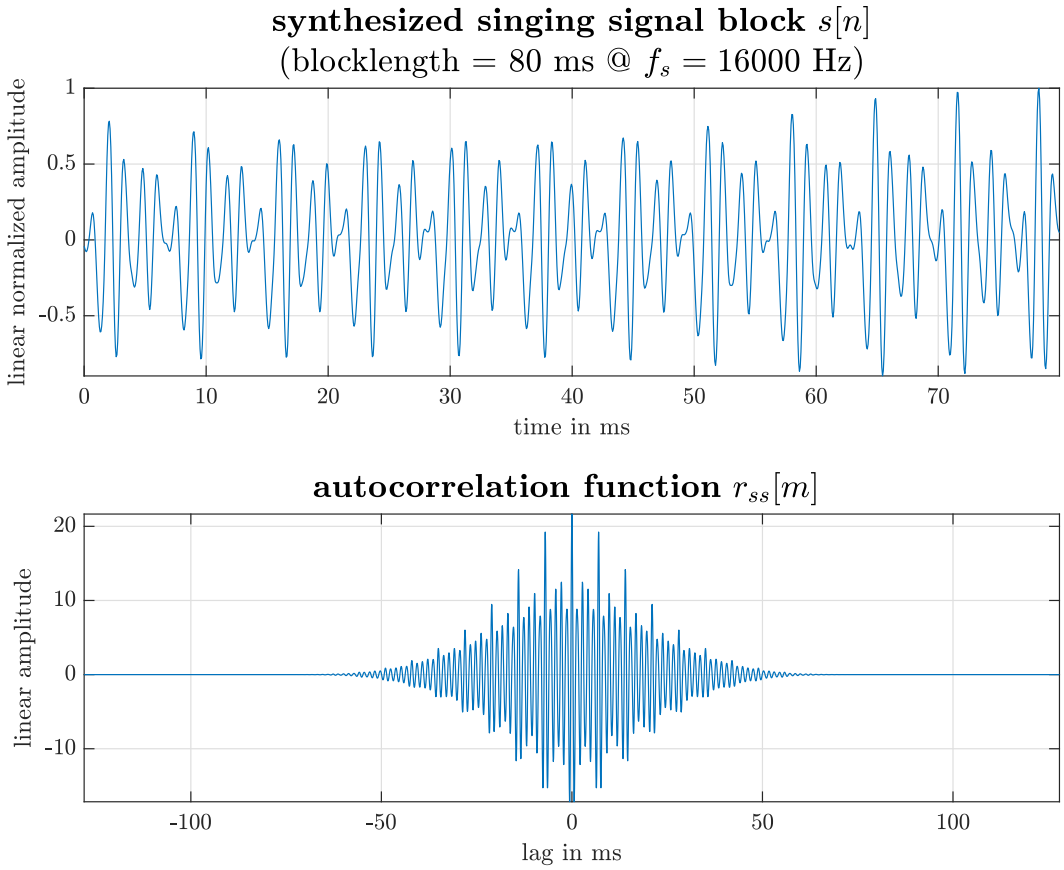


Figure 3.12 *Exemplary sung vocal signal block and corresponding autocorrelation function*

3.2.2 Autocorrelation Method with Cepstral Refinement

The theoretical background concerning the estimation of the filter coefficients for this method coincides with the autocorrelation method mentioned in subsection 3.2.2. The difference lies in the autocorrelation function used to solve Equation 3.40. In [60], Rahman and Shimamura proposed the usage of a *cepstral* refinement enabling harmonic suppresion of the fundamental frequency and its multiples, contained in the autocorrelation function, caused by the impulsive excitation signal. [2] The main advantage of a cepstrum is that the periodic source components can be separated from the vocal tract filter which as applied through a non-linear operation (convolution) [60, p.2]. Basically, the refinement can be interpreted as a smoothing of the autocorrelation function through so-called cepstral liftering. The autocorrelation is again calculated via the frequency domain as mentioned in 3.2.1. The cepstral refinement comprises, the computation of the autocorrelation function's cepstrum $c_{\text{rss}}[q]$, with [70, p. 66]

$$c_{\text{rss}}[q] = \mathcal{F}^{-1}_{k\mapsto q}\{\ln(|\mathcal{F}_{m\mapsto k}\{r_{\text{ss}}[m]\}[k]|)\}[q]\,. \tag{3.42}$$

. Afterwards in the cepstral domain a lifter window is applied. The liftered cepstrum is then transformed back to the time domain. The DFT resolution is calculated as mentioned in Equation 3.41, the same resolution is used for the inverse Fourier transform denoted in Equation 3.42.

[2] The term *cepstrum* was coined by Bogart, Healy and Tukey in 1963. It is a semantically playful adaption of the word spectrum. There are more semantical pairs, such as frequency/quefrency, filter/lifter and many more. A brief overview on the history of cepstrum is given in [57].

Lifter Window Computation. The next step is the computation of a lifter window, which is designed as a tapered cosine, or *Tukey* window. In contrast to the lengths of the lifter-window mentioned in [60], a length N_{wc} dependent on the estimated fundamental frequency $\hat{f}_0$ of the current signal is used, defined by

$$N_{\text{wc}} = \left\lfloor \frac{1}{\hat{f}_0} \cdot f_s \cdot 0.85 \right\rceil . \tag{3.43}$$

The window is then computed as a Tukey window with the length of N_{wc}. The Tukey window is defined in [5, eq. 6.9] for an interval of $0 \leq x \leq 1$. A discretized version for $x = \frac{n}{N_{\text{wc}}}$ can be formulated, such that

$$w_{\text{c}}[n] = \begin{cases} \frac{1}{2}\left(1 - \cos\left(\frac{2\pi}{\alpha_{\text{wc}}} \frac{n}{N}\right)\right), & 0 \leq n < \left\lfloor \frac{N \cdot \alpha_{\text{wc}}}{2} \right\rceil \\ 1, & \left\lfloor \frac{N \cdot \alpha_{\text{wc}}}{2} \right\rceil \leq n < \left\lfloor N - \frac{N \cdot \alpha_{\text{wc}}}{2} \right\rceil \\ \frac{1}{2}\left(1 - \cos\left(\frac{2\pi}{\alpha} - \frac{2\pi}{\alpha_{\text{wc}}} \frac{n}{N}\right)\right), & \left\lfloor N - \frac{N \cdot \alpha_{\text{wc}}}{2} \right\rceil \leq n \leq N \end{cases} . \tag{3.44}$$

The window length N is chosen as $N = 2 \cdot N_{\text{wc}}$, as the lifter window has to be a symmetric, allowing a correct reconstruction of the smoothed autocorrelation $\tilde{r}_{\text{ss}}[m]$ as formulated in Equation 3.45. The parameter α_{wc} defines, how much of the windowed data is tapered at the beginning end the end of the signal block [5, p. 69]. For the implemented algorithm $\alpha_{\text{wc}} = 0.5$ was chosen. In `Matlab`, the window is computed with the command `tukeywin()` [55].

Applying the Lifter Window. To apply the liftering operation on the calculated cepstrum $c_{\text{rss}}[q]$ from Equation 3.42, the Tukey lifter window has to be shifted into the correct position. Firstly, the window has to be zero-padded from length N to the length of the cepstrum, which coincides with the autocorrelation length N_{DFT}. This way, the zero-padded lifter window has the same length as the cepstrum. After zero-padding, the lifter window is then shifted circularly in the following way: The first half of the Tukey window is placed at the last N_{wc} samples of the autocorrelation function and the second half of the Tukey window is placed at the first N_{wc} samples of the autocorrelation blocks. The placement of the window over a exemplary normalized cepstrum is shown in Figure 3.13, where the cepstrum was only normalized for visualization purposes. The calculations in the algorithm implementation are executed for a *non-scaled* cepstrum. After the lifter window is shifted into the right position, the lifter operation reduces to an element-wise multiplication between the cepstrum $c_{\text{rss}}[q]$ and the window w_c.

Refined Autocorrelation Function. Following the liftering operation, a refined autocorrelation function $\tilde{r}_{\text{ss}}[m]$ can be calculated by reverting Equation 3.42 back to the time domain, such that

$$\tilde{r}_{\text{ss}}[m] = \mathcal{F}^{-1}_{k \mapsto m}\left\{ \left| \mathrm{e}^{\mathcal{F}_{q \mapsto k}\{c_{\text{rss}}[q] \cdot w_{\text{c}}[q]\}[k]} \right| \right\}[m] . \tag{3.45}$$

Figure 3.15 visualizes the cepstral refinement and its effects on the autocorrelation function. In the first subplot of Figure 3.15, the cepstrum of an exemplary signal block and the corresponding lifter window is visible. Through the element-wise multiplication, the low quefrency content of the spectrum is preserved. The first peak outside of the lifter window, can be interpreted as an more fluctuating spectral component due to the peak's position at a higher quefrency. The position of the peak corresponds to the estimated signal period $\hat{T}_0$, thus it can be assumed, that the cepstral peak represents the spectral fluctuations created by the periodic excitation signal.

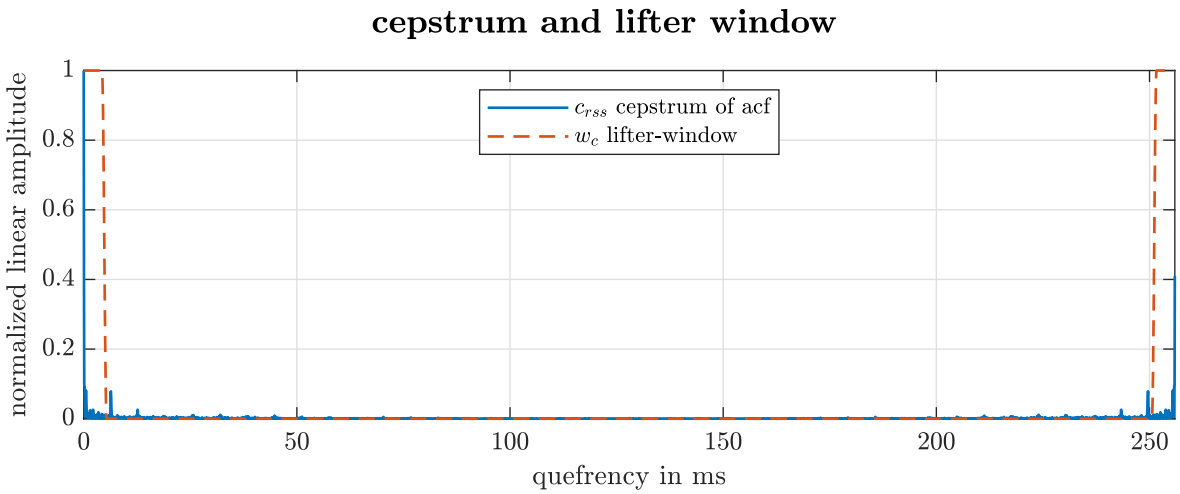


Figure 3.13 *Exemplary cepstrum and Tukey lifter window*

Low quefrencies correspond to low spectral fluctuations, which are caused for example by the vocal tract filter. These low-quefrency fluctiations are often described as the *spectral envelope* [60]. The spectral envelope of a sung vocal signal's autocorrelation shows the formant structure. This is also visible when comparing the spectrum of the liftered and unliftered autocorrelation, $\tilde{r}_{\mathrm{ss}}$ and r_{ss} respectively, which are shown in the third subplot of Figure 3.15.

The high dynamic in the autocorrelation's spectrum can be explained with the already high signal to noise ratio in the original signal, due to its synthetic nature. As mentioned in subsection 2.3.3, the only noise applied to the sung vocal signal is high-pass filtered additive Gaussian white noise, where the manually fixed signal to noise ratio amounts to $96\,\mathrm{dB}$, as defined in Equation 2.18. The additive noise's influence is drastically reduced in the autocorrelation domain, because due to its lack of covariance, it has only an effect on the autocorrelation bin for lag zero. This leads to an even higher signal to noise ratio and dynamic in the autocorrelation of a voiced sung vocal signal block.

To conclude the autocorrelation method using cepstral refinement for the vocal tract filter coefficient estimation, its signal flow is shown in Figure 3.14.

As shown in Figure 3.14, the signal flow starts with a voiced and pre-emphasis filtered signal block. Then its autocorrelation is calculated according to Equation 3.41. The next step is the cepstral refinement: By attenuating the excitation signals periodic influence with a lifter window in the cepstral domain, a smoothed autocorrelation is calculated according to Equation 3.45. In the implementation, this is done in the Matlab file `CepsLift.m` found in the folder `00_ABGABE_Matlab/V12b_LPA_JUCE_Matlab_Reference/`. The cepstrally refined autocorrelation function is then used to solve the Yule-Walker equation given in Equation 3.40, with the help of the Levinson-Durbin algorithm.

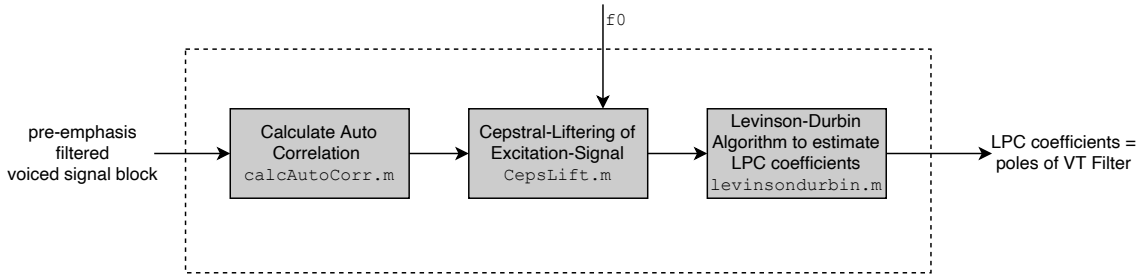


Figure 3.14 *Processing steps of the autocorrelation method with cepstral refinement*

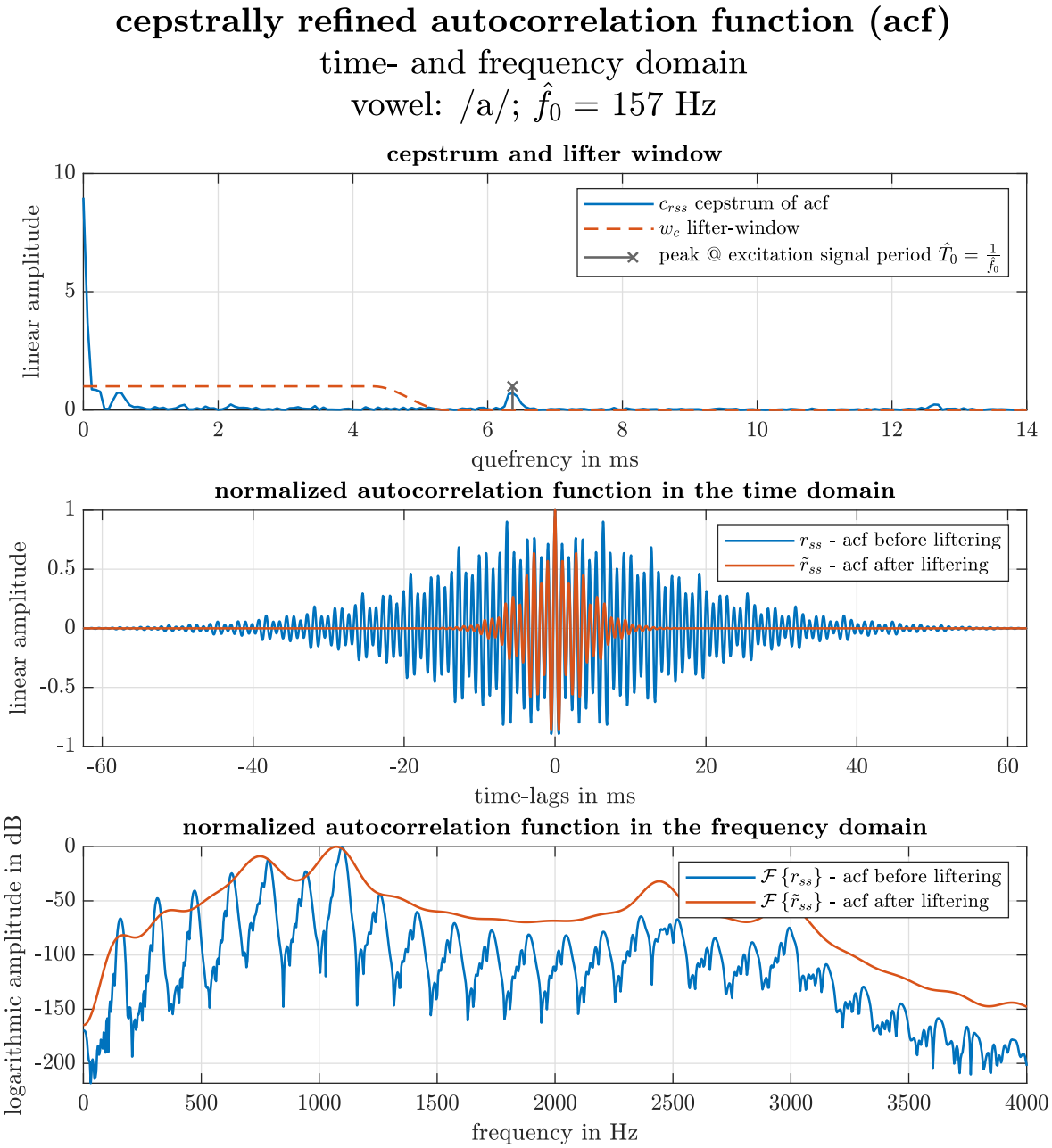


Figure 3.15 *Cepstral refinement on the autocorrelation function using liftering*

3.2.3 Covariance Method

The initial equation for the covariance method is Equation 3.29. The difference towards the autocorrelation method lies in the different averaging interval for the calculation of ϕ_{ij}. Where the averaging interval of the autocorrelation method was chosen to range over *all* available signal samples, the averaging interval of the covariance method is chosen to start at the p-th sample, such that

$$\phi_{ij} = \sum_{n=p}^{N-1} s[n-i]s[n-j]. \tag{3.46}$$

The averaging interval is predefined and ϕ_{ij} is called the covariant function in accordance to [38, p.12-

13]. Using Equation 3.29 and Equation 3.46, a matrix-vector equation system can be set up for the covariance method [38, p.13], such that

$$\sum_{i=1}^{p} \hat{a}_i \phi_{ij} = -\phi_{0j}, \qquad j = 0, 1, \ldots, p.$$

$$\begin{bmatrix} \phi_{1j} & \phi_{2j} & \ldots & \phi_{pj} \end{bmatrix} \begin{pmatrix} \hat{a}_1 \\ \hat{a}_2 \\ \hat{a}_3 \\ \vdots \\ \hat{a}_p \end{pmatrix} = -\phi_{0j}, \qquad j = 0, 1, \ldots, p \tag{3.47}$$

As already derived for the autocorrelation method, the sum rewritten as an inner vector product has to be exectued for all $j = 0, 1, \ldots, p$ and can therefore also be noted as a matrix-vector product in accordance to [38, p. 11], thus

$$\underbrace{\begin{bmatrix} \phi_{11} & \phi_{12} & \phi_{13} & \ldots & \phi_{1p} \\ \phi_{21} & \phi_{22} & \phi_{23} & \ldots & \phi_{2p} \\ \phi_{31} & \phi_{32} & \phi_{33} & \ldots & \phi_{3p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \phi_{p1} & \phi_{p2} & \phi_{p3} & \ldots & \phi_{pp} \end{bmatrix}}_{\boldsymbol{\Phi}} \underbrace{\begin{pmatrix} \hat{a}_1 \\ \hat{a}_2 \\ \hat{a}_3 \\ \vdots \\ \hat{a}_p \end{pmatrix}}_{\hat{\boldsymbol{a}}_{\text{opt}}} = - \underbrace{\begin{pmatrix} \phi_{01} \\ \phi_{02} \\ \phi_{03} \\ \vdots \\ \phi_{0p} \end{pmatrix}}_{\boldsymbol{\psi}}. \tag{3.48}$$

In contrast to the autocorrelation method, the matrix $\boldsymbol{\Phi}$ is neither an autocorrelation matrix, nor does it show Toeplitz structure. However, as the indices i and j iterate over the same values, they are interchangeable, thus

$$\phi_{ij} = \phi_{ji}, \qquad \forall\, i = 0, 1, \ldots, p \text{ and } \forall\, j = 0, 1, \ldots, p. \tag{3.49}$$

Therefore, $\boldsymbol{\Phi}$ is a symmetric matrix with the dimensions $[p \times p]$. Due to the given symmetry of $\boldsymbol{\Phi}$, Equation 3.48 can be solved for the optimal coefficients $\hat{\boldsymbol{a}}_{\text{opt}}$ with

$$\hat{\boldsymbol{a}}_{\text{opt}} = \boldsymbol{\Phi}^{-1} \boldsymbol{\psi}. \tag{3.50}$$

This matrix-vector equation can be solved using *Cholesky decomposition* [38, p. 13], which allows the factorization of a symmetric, positive semidefinite matrix into a diagonal matrix $\boldsymbol{D}$ and a lower triangular matrix $\boldsymbol{L}$, where

$$\boldsymbol{\Phi} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T, \tag{3.51}$$

as shown in [24, p. 143]. With this property, the matrix inversion occuring in Equation 3.50 can be solved more efficiently. The `Matlab` Code `cholesky.m` can be found in the folder `00_ABGABE_Matlab/V12b_LPA_JUCE_Matlab_Reference/` and was taken from [59]. It is used in the LP analysis implementation to execute the covariance method with the help of the Cholesky decomposition.

The steps a voiced signal block is processed through in the LP analysis algorithm with the covariance method are summarized in Figure 3.14. Similar to the autocorrelation method, the starting point is a whitened voiced signal block (whitened by means of the pre-emphasis filter discussed in subsection 3.1.4), afterwards the vocal tract filter coefficients are estimated with the covariance method using the Cholesky decomposition.

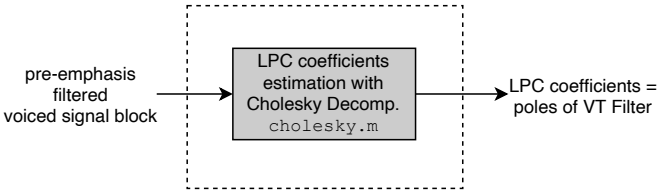


Figure 3.16 *Processing steps of the covariance method*

3.2.4 Windowed Covariance Method

In analogy to the relation between the *autocorrelation method* and the *autocorrelation method using cepstral refinement*, the *windowed covariance method* is the modified version of the covariance method. As visible in Figure 3.17, an additional block is present which modifies the previously known signal flow from Figure 3.16. In addition to the block representing vocal tract filter coefficient estimation using Cholesky decomposition, another process block is present. Within this block, a weight window is applied onto the whitened voiced signal block in time domain. The main goal of these weight windows is to weaken the signal's discontinuity occuring at the GCIs (i.e., the non-differentiable place in the dGF).

For the computation of the time windows, the information derived in the pre-processing stage mentioned in section 3.1 is used. The estimated fundamental frequency and estimated GCIs are needed for the window creations. In the `Matlab` implementation, the window creation is carried out in the file `createWeightWin.m`. The file is located in the folder `00_ABGABE_Matlab/V12b_LPA_JUCE_Matlab_Reference/`.

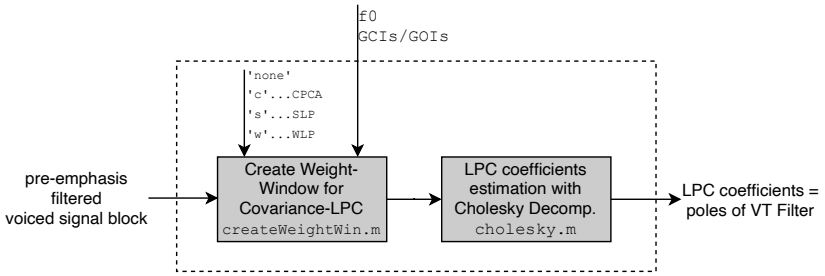


Figure 3.17 *Processing steps of the windowed covariance method*

Three possible types of windows were implemented according to [10], wherein different weightings define the names of the linear prediction analysis. The different analysis methods whose window creation were implemented in `createWeightWin.m` are

1. the closed phase covariance analysis (CPCA),
2. the weighted linear prediction (WLP) and
3. the sparse linear prediction (SLP).

In the following paragraphs, the different windows are distinguished using the corresponding abbreviations. The windows are created for all glottal cycles *inside* a voiced signal block.

Calculation of CPCA windows. The window used for *closed phase covariance analysis* is a rectangular window which opens for the closed phase of a glottal cycle. The closed phase is defined as the interval between a GCI $\hat{n}_e$ and a consecutive GOI $\hat{n}_0$. The estimated positions of the GIs are calculated as mentioned in subsection 3.1.3. Therefore, the discrete-time interval of the glottal closure phase can be assumed to be the time, which is included in the interval $t \in [\hat{t}_0,\ \hat{t}_e]$, which in samples corresponds to the interval $n \in [\hat{n}_0,\ \hat{n}_e]$. According to [10], a guarding delay of $0.9\,\mathrm{ms}$ is added to the GCI. If the time difference between the GOI and GCI reduces to a difference of less than $4.5\,\mathrm{ms}$, the guarding delay is decreased and fixed with $0.2\,\mathrm{ms}$ times of said difference. On a sample based calculation, the CPCA window $w_{\text{cpca}}[n]$ for one glottal cycle ($n \in [n_0,\ n_c]$) can be formulated as follows.

$$\begin{aligned} N_{\text{lim}} &= \left\lfloor 4.5 \cdot 10^{-3} \cdot f_s \right\rceil, \qquad N_{\text{cycle}} = |\hat{n}_e - \hat{n}_0| \\ n_{\text{guard,1}} &= \left\lfloor 0.9 \cdot 10^{-3} \cdot f_s \right\rceil, \qquad n_{\text{guard,2}} = \lfloor 0.2 \cdot |\hat{n}_e - \hat{n}_0| \rceil \\ w_{\text{cpca}}[n] &= \begin{cases} 1\ \forall n \in \left[\hat{n}_0 + n_{\text{guard,1}},\quad \hat{n}_e\right], & \text{if } N_{\text{cycle}} > N_{\text{lim}} \\ 1\ \forall n \in \left[\hat{n}_0 + n_{\text{guard,2}},\quad \hat{n}_e\right], & \text{if } N_{\text{cycle}} < N_{\text{lim}} \\ 0, & \text{else} \end{cases} \end{aligned} \tag{3.52}$$

The window creation is repeated for each pair of GCI and GOI detected inside the signal block.

Calculation of WLP windows. The windows used for *weighted linear prediction* analysis are created as piecewise linear functions as proposed in [10]. Thereby, two amplitude levels inside the window function are connected linearly. The maximum value of the window lies at one, and the minimum amplitude value amounts to 0.05. The value change from the maximum to the minimum value or vice versa is carried out in $0.45\,\mathrm{ms}$ with a linear ramp function. Therefore, a slope k_{wlp} can be calculated for the linear part of the window in the discrete time domain, such that

$$k_{\text{wlp}} = \frac{1 - 0.05}{\lfloor 0.45 \cdot 10^{-3} \cdot f_s \rceil} = \frac{0.95}{\lfloor 0.45 \cdot 10^{-3} \cdot f_s \rceil}. \tag{3.53}$$

The low amplitude value of the window is reached at 0.32 times the current fundamental period estimate $\hat{T}_0$ before a estimated GCI instant $\hat{n}_e$. The low amplitude value is then held for 0.4 times T_0. After this, the ramp-up process is started and the high amplitude value is reached within $0.45\,\mathrm{ms}$. Firstly, four sample-based time indices Δ_1, Δ_2, Δ_3 and Δ_4 need to be calculated, such that

$$\begin{aligned} \Delta_1 &= \left\lfloor \left(0.32 \cdot \hat{T}_0 + 0.45 \cdot 10^{-3}\right) \cdot f_s \right\rceil, \qquad \Delta_2 = \left\lfloor 0.32 \cdot \hat{T}_0 \cdot f_s \right\rceil \\ \Delta_3 &= \left\lfloor 0.08 \cdot \hat{T}_0 \cdot f_s \right\rceil, \qquad \Delta_4 = \left\lfloor \left(0.08 \cdot \hat{T}_0 + 0.45 \cdot 10^{-3}\right) \cdot f_s \right\rceil. \end{aligned} \tag{3.54}$$

Using the values of Δ_1, Δ_2, Δ_3 and Δ_4, the WLP window function $w_{\text{wlp}}[n]$ for one glottal cycle ($n \in [n_0,\ n_c]$) can be formulated as follows.

$$w_{\text{wlp}}[n] = \begin{cases} -k_{\text{wlp}} \cdot [n - (\hat{n}_e - \Delta_1)] + 1, & \forall\, n \in \left[\hat{n}_e - \Delta_1,\quad \hat{n}_e - \Delta_2\right] \\ 0.05, & \forall\, n \in \left[\hat{n}_e - \Delta_2,\quad \hat{n}_e + \Delta_3\right] \\ k_{\text{wlp}} \cdot [n - (\hat{n}_e + \Delta_3)] + 0.05, & \forall\, n \in \left[\hat{n}_e + \Delta_3,\quad \hat{n}_e + \Delta_4\right] \end{cases} \tag{3.55}$$

In `Matlab`, the linear sections connecting the maximum and minimum values of the windows are calculated with the `interp1()` command [49].

Calculation of SLP windows. In contrast to the previous windows, the *sparse linear prediction* window is defined for the whole signal block, meaning the window computation doesn't have to be repeated for each glottal cycle. All GCI estimates $\hat{\boldsymbol{n}}_e$ are included in the window computation. With the SLP window, a notch is placed at each detected GCI which weakens the discontinuities. With the fixed parameters κ and σ as well as $\hat{n}_e^{(l)}$ being the l-th GCI estimate, the SLP window $w_{\text{slp}}[n]$ can be defined according to [10] such that

$$w_{\text{slp}}[n] = 1 - \kappa \cdot \sum_{l=1}^{L} \mathrm{e}^{\frac{-\left(n-\hat{n}_e^{(l)}\right)^2}{2(\sigma f_s)^2}}. \tag{3.56}$$

As proposed in [10], the parameters were defined as $\kappa = 0.9$ and $\sigma = 0.25\,\mathrm{ms}$. κ defines the signal dampening at each estimated GCI, where a value of $\kappa = 0.9$ means that the minimum amplitude value for the SLP window is 0.1 at the GCIs $\hat{\boldsymbol{n}}_e$.

Comparison of Window Functions and Conclusion. A graphical comparison of each weight window is illustrated in Figure 3.18. The created windows for one signal block are plotted over the signal block's true excitation signal (dGF) for modal voice quality, in order to show the window positioning around the estimated glottal instants.

In the first subplot of Figure 3.18 it is visible, that the flawed GOI-detections mentioned in subsection 3.1.3 lead to misplaced CPCA-windows. There are respectively long phases where the window sets the signal to 0. This means a lot of energy is taken out of the signal block. The WLP window shown in the second subplot shows shorter dampening phases, but still $40\,\%$ of each glottal cycle are dampened. The most energy is preserved when using the SLP windows, shown in the third subplot. Also due to the robust GCI-detection the notches contained in the SLP window are placed very accurately.

Therefore, in order to ensure further estimation with as much signal energy as possible and due to more robust GCI-detection the SLP window is chosen for further representation of the windowed covariance method. This means for the forthcoming sections, when the windowed covariance method is mentioned section 3.4, always SLP weight windows were used. The terms *sparse linear prediction* and *windowed covariance method* are therefore used interchangeably in the following parts of this document.

After the window is computed for one sung vocal signal block and the pre-emphasis filtered sung vocal signal is multiplied with the window, the windowed covariance method then uses the same steps same steps to solve Equation 3.50, as the covariance method.

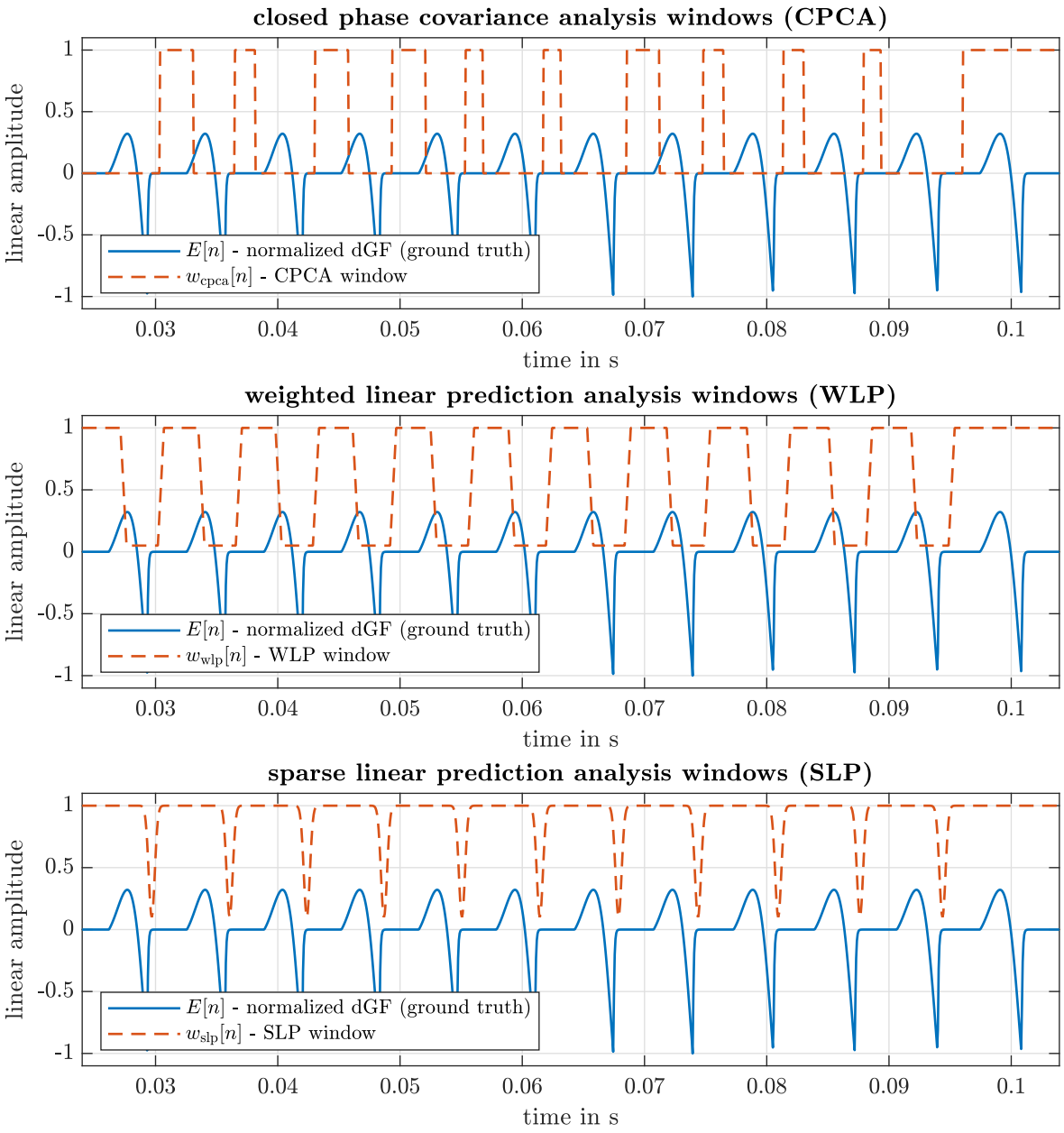


Figure 3.18 *Comparison of windows implemented for the windowed covariance method (CPCA, WLP and SLP)*

3.3 Post-Processing and Classification of Vowel and Voice Quality

This section describes, how an estimation of the dGF is computed, using estimated filter coefficients of the vocal tract filter, which have been calculated based on a *blocked signal* using a linear prediction method as described in section 3.2. Further, the estimated dGF is then used to calculate features, enabling voice quality classification, whereas the filter coefficients are used to determine which vowel was sung. On a general level, the post-processing and classification stage can be split up into four parts:

1. *inverse filtering* with the estimated vocal tract filter to calculate the estimated dGF, as described in subsection 3.3.1
2. *calculation of formant frequencies* from the estimated vocal tract filter coefficients, as described in subsection 3.3.2
3. *indication of vowel* based on the estimated formant frequencies, as described in subsection 3.3.3
4. *indication of voice quality* based on two skewness measures calculated from the dGF, as described in subsection 3.3.4

Each of these steps is discussed in detail in the following subsections. Note, that in principle the vocal tract filters of all four linear prediction methods discussed in section 3.2 can be used for the post-processing. If not stated otherwise, the figures displayed in the following subsections are created using the *autocorrelation method with cepstral refinement* from subsection 3.2.2.

3.3.1 Inverse Filtering

With any of the linear prediction algorithms, described in section 3.2, linear prediction coefficients $\hat{\boldsymbol{a}}_{\text{opt}} = [\hat{a}_0, \ldots, \hat{a}_p]^T$ and a estimated filter gain $\hat{G}$ can be derived (p is the linear prediction order). The coefficients $\hat{\boldsymbol{a}}_{\text{opt}}$ are defined in Equation 3.40 for the autocorrelation method, and in Equation 3.50 for the covariance method. Thus, the estimated vocal tract filter in z-domain is given as

$$\hat{H}_{\text{VT}}(z) = \frac{\hat{G}}{\sum\limits_{k=0}^{p} \hat{a}_k z^{-k}} = \frac{\hat{G}}{\nu(z)}, \tag{3.57}$$

where $a_0 = 1$, and $\nu(z)$ denotes the denominator for later reference.

To estimate the dGF of a sung vocal signal using the estimated vocal tract filter $\hat{H}_{\text{VT}}(z)$, the filter from Equation 3.57 must be inverted. In time domain, this results in the following relation

$$\hat{E}_{\text{full}}[n] = \frac{1}{\hat{G}} \sum_{k=0}^{p} \hat{a}_k s[n-k] \tag{3.58}$$

where $s[n]$ is the synthesized sung vocal signal as described in chapter 2 and $\hat{E}[n]$ denotes the estimated dGF. In `Matlab`, `filter()` is used for this operation [45]. After the inverse filtering, the first $2 \cdot p$ samples of $\hat{E}[n]$ are set to zero, because they contain the initial transient response of the linear prediction filter which is not meaningful in this context.

$$\hat{E}[n] = \begin{cases} 0 & \text{for} \quad 0 \leq n \leq (2 \cdot p) - 1 \\ \hat{E}_{\text{full}}[n] & \text{else} \end{cases} \tag{3.59}$$

From the estimated dGF $\hat{E}[n]$, an estimation of the glottal flow $\hat{E}_{\text{GF}}[n]$ can be calculated with the

cumulative sum of the dGF, such that

$$\hat{E}_{\mathrm{GF}}[n] = \sum_{k=0}^{n} \hat{E}[k]. \tag{3.60}$$

The inverse filtering algorithm delivering the estimated dGF as well as the estimation of the glottal flow is implemented in the `Matlab`-file `AllPoleInvFilt.m`, which can be found in the folder `V12b_LPA_JUCE_Matlab_Reference/`.

3.3.2 Calculation of Formant Frequencies from Estimated Filter Coefficients

The algorithm described in this subsection provides an estimate of the vocal tract filter by means of its coefficients $\hat{a}_k$, as defined in Equation 3.57. The fundamental theorem of algebra states, that the denominator $\nu(z)$ of the estimated vocal tract filter's transfer function can be rewritten by means of its complex zeros $z_{0,k}$, such that

$$\nu(z) = \sum_{k=0}^{p} \hat{a}_k z^{-k} = \frac{1}{z^p} \sum_{k=0}^{p} \hat{a}_k z^{p-k} = \frac{1}{z^p} \prod_{k=0}^{p} \left(z - z_{0,k}\right). \tag{3.61}$$

The zeros $z_{0,k}$ of $\nu(z)$ are in general complex-valued, and their location in the z-plane determines the bandwidth and the frequency of the estimated vocal tract's corresponding pole. To calculate the zeros, the *eigenvalue problem of the companion matrix* needs to be solved, as demonstrated in [72]. Therefore, the companion matrix $\boldsymbol{A}$ is set up in accordance to [52] such that

$$\boldsymbol{A} = \begin{bmatrix} -\hat{a}_1 & -\hat{a}_2 & -\hat{a}_3 & \cdots & -\hat{a}_{p-1} & -\hat{a}_p \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \tag{3.62}$$

This matrix $\boldsymbol{A}$ has size $p \times p$, and its eigenvalues are equivalent to the zeros $z_{0,k}$ of $\nu(z)$. The eigenvalues $z_{0,k}$ can be calculated by means of a Matlab command `eig()` [42], or the equivalent command from a C++ library like *Eigen* [26].

The noted matrix is a transposed form of the companion matrix, as used in `Matlab` [52], in contrast to this form, other literature (for example Werner [72]) often mentions a direct form of the companion matrix $\boldsymbol{A}$. However, due to the fact that $\hat{a}_k \in \mathbb{R}\ \forall k = 0, \ldots, p$, the eigenvalue calculation is invariant to a transposition of the companion matrix, because the zeros $z_{0,k}$ are either real-valued or occur in *complex conjugated* pairs.

From the set of complex zeros $z_{0,k}$, which are the poles of the estimated vocal tract filter $\hat{H}_{\mathrm{VT}}(z)$ only those with an imaginary part greater or equal to zero are further considered, i.e. if $\mathfrak{Im}\{z_{0,k}\} < 0$, the zero gets omitted, because it has a conjugate complex zero which carries the necessary information for the following steps. The formant frequencies F_i and the corresponding bandwidths B_i are determined in the following way:

$$\begin{aligned} F_i &= \tan^{-1}\left(\frac{\mathfrak{Im}\{z_{0,i}\}}{\mathfrak{Re}\{z_{0,i}\}}\right) \cdot \frac{f_s}{2\pi} \\ B_i &= -\ln(|z_{0,i}|) \cdot \frac{f_s}{\pi}, \end{aligned} \tag{3.63}$$

where f_s is the sampling frequency and i is the index for all zeros with *non-negative imaginary parts*. For each pair of F_i and B_i, denoted subsequently as $\langle F_i, B_i \rangle$, limitation parameters are checked, because it is not meaningful to consider all estimated poles $z_{0,i}$ of the vocal tract filter to be formants. Therefore, the following limitation parameters are introduced:

- upper bandwith limit B_{max}
- minimum formant frequency F_{min}
- maximum formant frequency F_{max}

The following values for the limitation parameters are used in the presented implementation:

$$B_i < B_{\text{max}} = 500\,\text{Hz}, \qquad F_i > F_{\text{min}} = 90\,\text{Hz}, \qquad F_i < F_{\text{max}} = 3.5\,\text{kHz}. \tag{3.64}$$

From all $\langle F_i, B_i \rangle$ that fulfill these criterions, the two pairs with the smallest formant frequencies F_i are considered the estimated formant $\hat{F}_1$ and $\hat{F}_2$, respectively. For example considering the estimation of the first formant $\hat{F}_1$

$$\hat{F}_1 = \min\{F_i\} \quad \forall \quad \langle F_i, B_i \rangle : B_i < B_{\text{max}} \wedge F_i > F_{\text{min}} \wedge F_i < F_{\text{max}}. \tag{3.65}$$

Based on the estimated first and second formants $\hat{F}_1$ and $\hat{F}_2$, a classification of the sung vowel can be obtained, as described in the following subsection.

3.3.3 Indication of Vowel based on Estimated Formant Frequencies

Based on the first two formants F_1 and F_2, Sendlmeier *et al.* have established a map of spoken German vowels in [63, 64]. Their study is based on $N_{\text{female}} = 58$ female and $N_{\text{male}} = 69$ male participients, which were asked to realize 16 different vowels embedded in two-syllable standard German words. The formant frequencies were measured with the software PRAAT [6, 63]. Before them, Hillenbrand *et al.* have executed a similar study for spoken English in [27] with $N_{\text{female}} = 48$ female and $N_{\text{male}} = 45$ male participients with $N_{\text{child}} = 46$ children. Due to the more recent research, the data of Sendlmeier *et al.* was used in the successive vowel classification and visualization process.

Sendlmeier *et al.* present their research data for each vowel by means of a mean value and a standard deviation with respect to F_1 and F_2 for each vowel. The mean is denoted as $\boldsymbol{\mu}_{\text{vow}}$ and the standard deviation as $\boldsymbol{\sigma}_{\text{vow}}$.

$$\boldsymbol{\mu}_{\text{vow}} = \begin{bmatrix} \mu_{F_1} \\ \mu_{F_2} \end{bmatrix}, \qquad \boldsymbol{\sigma}_{\text{vow}} = \begin{bmatrix} \sigma_{F_1} \\ \sigma_{F_2} \end{bmatrix} \tag{3.66}$$

In Figure 3.19, the ellipses indicate the vowel location within the F_1/F_2-plane.[3]

From Figure 3.19 one can assert, that the distinction of 16 different vowels is not feasible in the context of this project, because an extensive amount of overlap is given within the groups of vowels. Therefore, a subset $\mathcal{S}_{\text{vow}}$ of Sendlmeier's 16 vowels was selected to provide a vowel classification, such that

$$\mathcal{S}_{\text{vow}} = \Big\{ \text{/a:/}, \quad \text{/ɛ:/}, \quad \text{/i:/}, \quad \text{/ɔ/}, \quad \text{/u:/}, \quad \text{/y:/}, \quad \text{/ə/} \Big\} = \Big\{ v_i \Big\}_{i=1}^{7}, \tag{3.67}$$

where the elements v_i of $\mathcal{S}_{\text{vow}}$ are denoted with the IPA-symbols according to [64]. Each vowel in $\mathcal{S}_{\text{vow}}$ has a corresponding mean vector $\boldsymbol{\mu}_{\text{vow}}$ and standard deviation vector $\boldsymbol{\sigma}_{\text{vow}}$. Thus, for each vowel

[3] Note, that Sendlmeier *et al.* do not provide information about the covariance between F_1 and F_2. Therefore, the covariance is assumed to be zero.

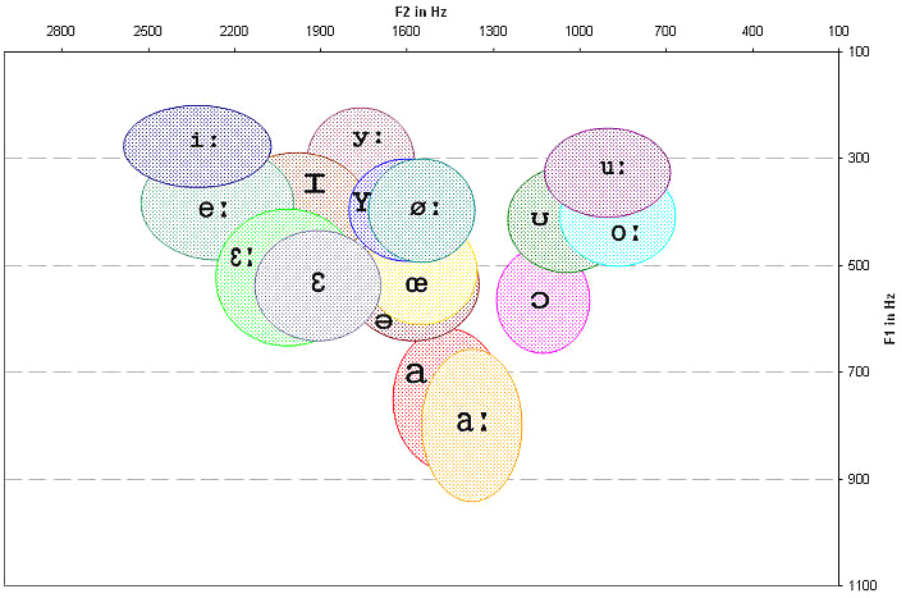


Figure 3.19 *Location of vowels in the formant plane defined by the first two formants F_1 and F_2 according to [63, Figure 1]*

in $\mathcal{S}_{\text{vow}}$, a bivariate normal probability density function (PDF) $p_{\text{vow},i}(\boldsymbol{x})$ is set up, which leaves for the i-th vowel in $\mathcal{S}_{\text{vow}}$

$$p_{\text{vow},i}(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^2 \det \boldsymbol{\Sigma}_{\text{vow},i}}} \mathrm{e}^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_{\text{vow},i})^T \boldsymbol{\Sigma}_{\text{vow},i}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_{\text{vow},i})}$$
$$\text{with} \quad \boldsymbol{\Sigma}_{\text{vow},i} = \begin{bmatrix} \sigma_{F_1,i}^2 & 0 \\ 0 & \sigma_{F_2,i}^2 \end{bmatrix}, \tag{3.68}$$

where $\boldsymbol{x}$ is a point of interest in the F_1/F_2-plane whose probability of membership to the i-th vowel in $\mathcal{S}_{\text{vow}}$ should be evaluated, where $\boldsymbol{\Sigma}_{\text{vow},i}$ is the covariance matrix of the i-th vowel. Commonly, the point $\boldsymbol{x}$ is a grid point of a mesh grid that discretizes the F_1/F_2-plane. Given a PDF $p_{\text{vow},i}(\boldsymbol{x})$ for each vowel in $\mathcal{S}_{\text{vow}}$, each mesh grid point $\boldsymbol{x}$ is classified to belong to the vowel with the highest probability, i.e.

$$v_{i,\text{opt}}(\boldsymbol{x}) = \underset{i \in \mathcal{S}_{\text{vow}}}{\operatorname{argmax}}\, p_{\text{vow},i}(\boldsymbol{x}), \tag{3.69}$$

where $v_{i,\text{opt}}(\boldsymbol{x})$ denotes the vowel in $\mathcal{S}_{\text{vow}}$, that has the highest probability for the mesh grid point $\boldsymbol{x}$.

Weighting of Gender. Sendlmeier *et al.* provide the data (mean $\boldsymbol{\mu}_{\text{vow},i}$ and standard deviation $\boldsymbol{\sigma}_{\text{vow},i}$ of the i-th vowel) for $N_{\text{male}} = 69$ males and $N_{\text{female}} = 58$ females seperately. This enables four different data weighting schemes, which are explained in the following. Let $\boldsymbol{\mu}_{\text{vow,male},i}$ and $\boldsymbol{\sigma}_{\text{vow,male},i}$ be the mean and standard deviation for males, and $\boldsymbol{\mu}_{\text{vow,female},i}$ and $\boldsymbol{\sigma}_{\text{vow,female},i}$ for females of the i-th vowel in $\mathcal{S}_{\text{vow}}$, respectively. Four different weighting options are possible:

- use the mean and standard deviation measured for *females only*
- use the mean and standard deviation measured for *males only*

- use a *weighted average* for mean and standard deviation, i.e.

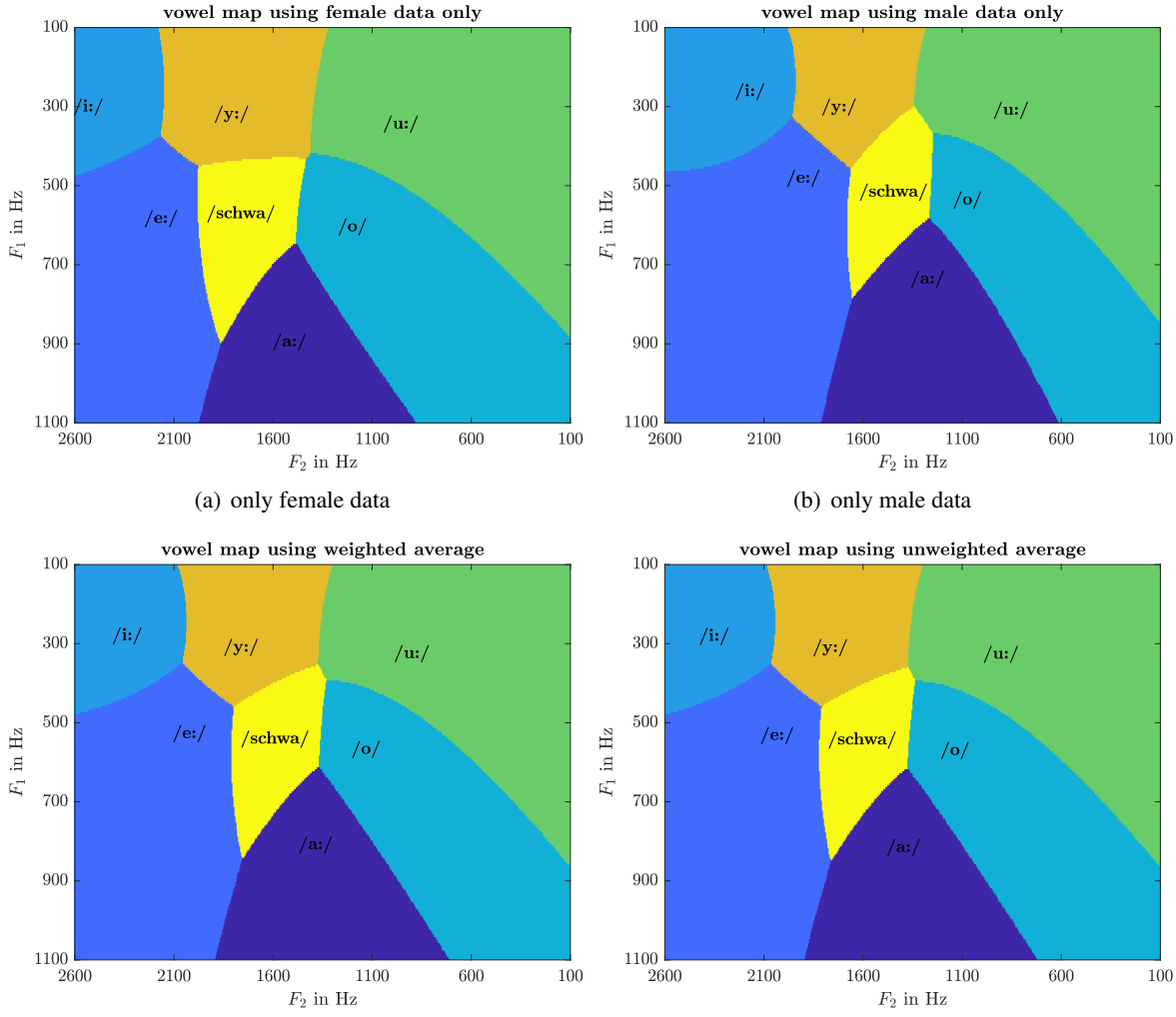
$$\begin{aligned} \boldsymbol{\mu}_{\text{vow},i} &= \frac{N_{\text{female}} \cdot \boldsymbol{\mu}_{\text{vow,female},i} + N_{\text{male}} \cdot \boldsymbol{\mu}_{\text{vow,male},i}}{N_{\text{female}} + N_{\text{male}}} \\ \boldsymbol{\sigma}_{\text{vow},i} &= \frac{N_{\text{female}} \cdot \boldsymbol{\sigma}_{\text{vow,female},i} + N_{\text{male}} \cdot \boldsymbol{\sigma}_{\text{vow,male},i}}{N_{\text{female}} + N_{\text{male}}} \end{aligned} \tag{3.70}$$

- use an *unweighted average* for mean and standard deviation, i.e.

$$\begin{aligned} \boldsymbol{\mu}_{\text{vow},i} &= \frac{\boldsymbol{\mu}_{\text{vow,female},i} + \boldsymbol{\mu}_{\text{vow,male},i}}{2} \\ \boldsymbol{\sigma}_{\text{vow},i} &= \frac{\boldsymbol{\sigma}_{\text{vow,female},i} + \boldsymbol{\sigma}_{\text{vow,male},i}}{2} \end{aligned} \tag{3.71}$$

The effects of the different weighting methods can be seen in Figure 3.20. From Figure 3.20 it becomes clear, that in general, women have higher formants than males, especially when looking at the central */schwa/* sound. In the synthesizer described in chapter 2, only male formant frequencies, as listed in Table 2.3, have been considered, due to the lack of data on the LF-model parameters for female speakers in [22]. Therefore, the vowel map displaying male data only, as showed in Subfigure (b), is used for further applications.

Graphical Indication of Present Vowel. To indicate the vowel of a given sound, a point is plotted in the coloured F_1/F_2-plane for male data, which is shown in Figure 3.20 (b). Based on the point's location, the user has a visual indication of the present vowel. For the five vowels of interest, examples of the visual vowel indication are presented in Figure 3.21. Note that in Figure 3.21 (e) it is indicated that the actual vowel */u/* is classified as being a */schwa/*, but the estimated formants F_1 and F_2 correspond with the synthesized vocal tract filter listed in Table 2.3. Therefore it is important to notice, that there are discrepancies in the literature on the exact location of formant frequencies, at least between [19] and [63] for the formant frequencies concerning the vowel */u/*.



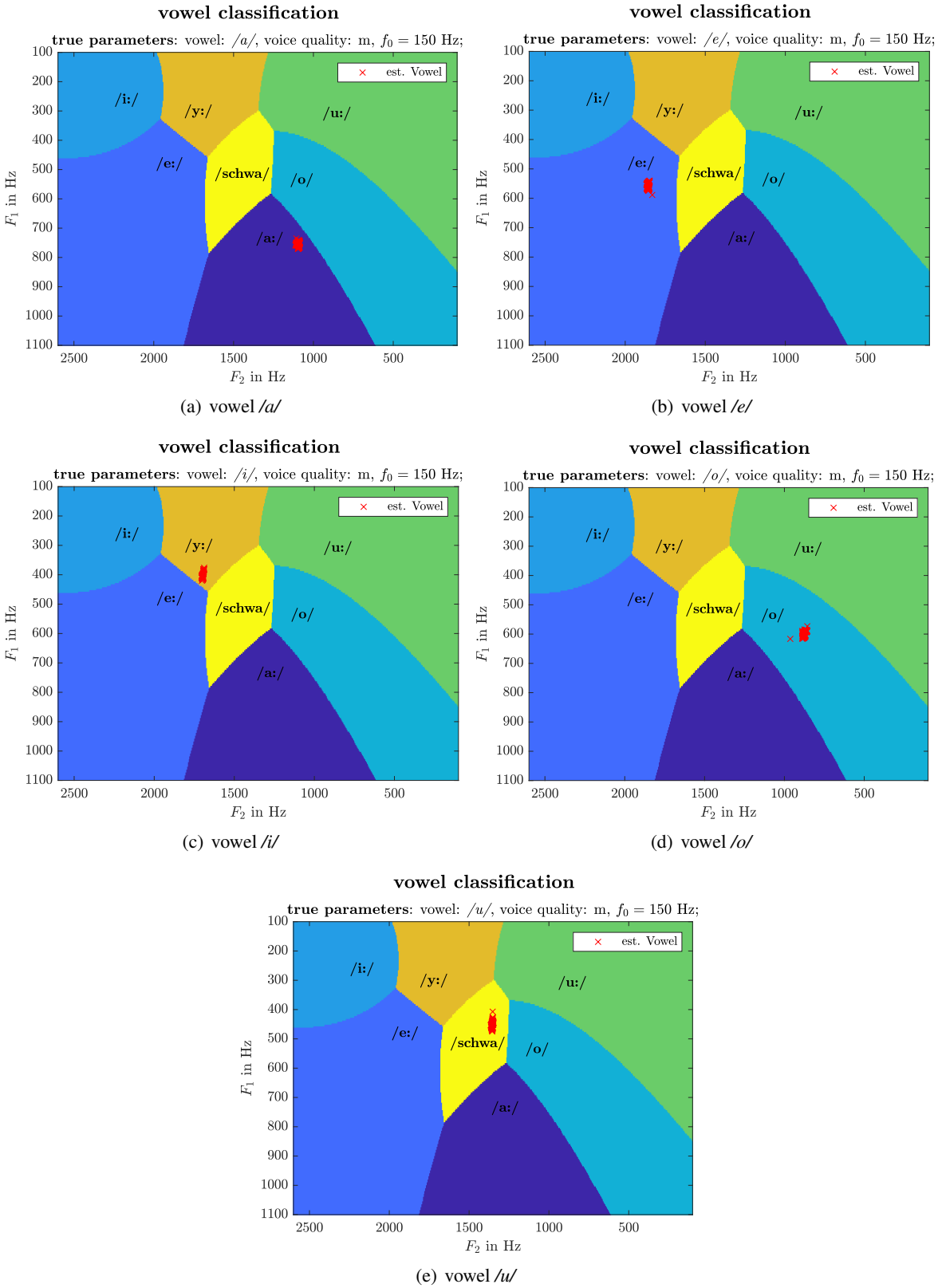
(a) only female data

(b) only male data

(c) weighted average

(d) unweighted average

Figure 3.20 *Comparison of vowel maps for different weighting methods of male and female formant frequencies. Data source [64]*



(a) vowel /a/ (b) vowel /e/

(c) vowel /i/ (d) vowel /o/

(e) vowel /u/

Figure 3.21 *Visual indication of the present vowel for a modal voice quality with a fundamental frequency of* $f_0 = 150\,\text{Hz}$

3.3.4 Classification of Voice Quality based on Skewness Measures

In this section, the classification of voice quality is explained. In order to classify the voice quality, the estimated dGF, denoted as $\hat{E}[n]$, obtained by inverse filtering as described in subsection 3.3.1, two skewness-related measures are used:

1. the *skewness of the dGF amplitude values* and
2. a *skewness-related measure of the estimated glottal flow* (GF).

In the following section, the two skewness-related measures are described in detail.

Skewness of dGF Amplitude Values. One of the two features used to enable an assignment of a synthesized speech sound to a voice quality is the skewness of the dGF amplitude values s_{dGF}. It is defined as

$$s_{\text{dGF}} = \mathbb{E}\left\{ \frac{\left(\hat{E}[n] - \mu_{\hat{E}}\right)^3}{\sigma^3_{\hat{E}}} \right\}, \tag{3.72}$$

where $\mathbb{E}\{\cdot\}$ denotes the expectation operator, $\hat{E}[n]$ is the estimated dGF of one signal block, $\mu_{\hat{E}}$ and $\sigma_{\hat{E}}$ are the mean and the standard deviation of the dGF amplitude values, respectively. This is the *third standardized central moment*, which is calculated by dividing the *third central moment* [58, eq. (5-68)] through the standard deviation $\sigma_{\hat{E}}$ to the power of three . For mean $\mu_{\hat{E}}$ and standard deviation $\sigma_{\hat{E}}$, the following estimators are used

$$\mu_{\hat{E}} = \frac{1}{N}\sum_{n=0}^{N-1} \hat{E}[n] \qquad\qquad \sigma_{\hat{E}} = \sqrt{\frac{1}{N}\sum_{n=0}^{N-1} \left|\hat{E}[n] - \mu_{\hat{E}}\right|^2}. \tag{3.73}$$

The mean estimator stems from [58, p. 307]. Note, that for the standard deviation, the *biased*[4] estimation was used, in contrast to [58, eq. (8-13)]. In Matlab, the skewness of the dGF amplitude values is calculated with `skewness()` [53]. Thus, we obtain one value for s_{dGF} for each signal block.

In Figure 3.22, histogram and skewness value for the three voice qualities are displayed for a fundamental frequency of $f_0 = 120\,\text{Hz}$.

Skewness-Related Measure of GF. The second skewness-related measure aims at the skewness of the glottal flow (GF) for each glottal cycle. To separate the glottal cycles, we need the estimated GCIs as described in subsection 3.1.3. Precisely, the vector $\hat{\boldsymbol{n}}_e$, that contains all estimated GCIs, is needed as defined in Equation 3.14. To estimate the glottal flow $\hat{E}_{\text{GF}}[n]$, the cumulative sum of the mean-excempt dGF $\hat{E}[n]$ is calculated as follows:

$$\hat{E}_{\text{GF}}[n] = \sum_{k=0}^{n} \left(\hat{E}[k] - \mu_{\hat{E}}\right) \tag{3.74}$$

The estimated GF is $\hat{E}_{\text{GF}}[n]$ scaled to the interval $[0, 1]$ by subtracting its minimum value and dividing by its maximum value, which results in the scaled GF $\hat{E}_{\text{GF,sc}}[n]$, such that

$$\hat{E}_{\text{GF,sc}}[n] = \frac{\hat{E}_{\text{GF}}[n] - \min\left(\hat{E}_{\text{GF}}[n]\right)}{\max\left(\hat{E}_{\text{GF}}[n] - \min\left(\hat{E}_{\text{GF}}[n]\right)\right)}. \tag{3.75}$$

[4] The bias is introduced by the factor $1/N$, in contrast to the factor of the unbiased standard deviation estimate, which is $1/(N-1)$.

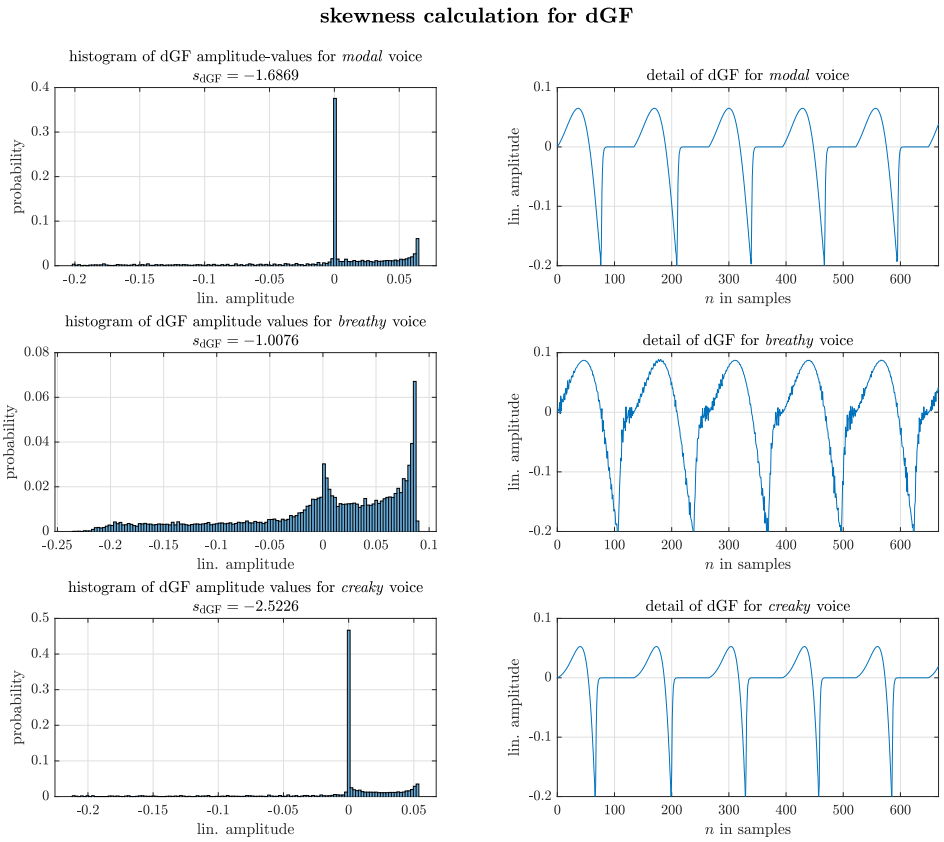


Figure 3.22 *Comparison of the dGF skewness values for different voice qualities using a fundamental frequency of* $f_0 = 120\,\mathrm{Hz}$

The process of calculating the GF is illustrated in Figure 3.23.

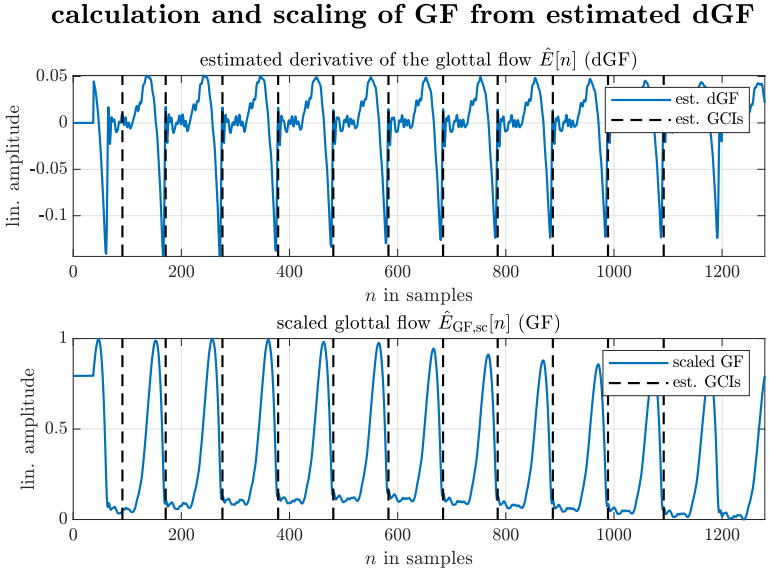


Figure 3.23 *Calculation of scaled GF* $\hat{E}_{GF,sc}[n]$ *using the dGF's cumulative sum, scaled to the interval* $[0, 1]$ *for one signal block of the synthesized vowel /a/ with a fundamental frequency* $f_0 = 150\,\mathrm{Hz}$ *and modal voice quality*

Now the scaled GF is considered for each glottal cycle separately. Let i be the index of GCIs in one audio block with the range $i = 1, \ldots, N_c$, where N_c is the number of GCIs in that audio block. From subsection 3.1.3, it is clear that

$$\hat{\boldsymbol{n}}_e = \begin{bmatrix} \hat{n}_{e,1} & \cdots & \hat{n}_{e,N_c} \end{bmatrix}^T, \tag{3.76}$$

where $\hat{n}_{e,i}$ it the sample index of the i-th GCI in the audio block. Implicitly, we assume that $\hat{n}_{e,i} \neq \hat{n}_{e,j} \neq 0$ for all $i \neq j$, which means that all detected GCIs are different and non-zero. Eventually, this must be assured in an actual implementation e.g. through the usage of commands like `unique()` in `Matlab`. Thus, the duration $d_{\mathrm{c},i}$ (in samples) of the i-th glottal cycle is the difference between two successive GCI indices, i.e.

$$d_{\mathrm{c},i} = \hat{n}_{e,i+1} - \hat{n}_{e,i} \qquad \text{where} \qquad i = 1, \ldots, N_c - 1. \tag{3.77}$$

Each glottal cycle is then interpolated with a cubic *spline* interpolation, which is implemented in `Matlab`'s `interp1()` [49]. Therefore, $N_{\text{interp}} = 500$ query points are used in the interval $[0, 1]$. The sample points are defined as $d_{\mathrm{c},i}$ linearly spaced points in the interval $[0, 1]$ for the i-th glottal cycle in the audio block. This way, the scaled and interpolated GF has an abscissa range in the interval $[0, 1]$. Now the skewness s_{GF} of the scaled and interpolated GF's amplitude distribution is calculated for each glottal cycle contained in the signal block. For instance this can be done with Matlab's `skewness()` [53]. Figure 3.24 shows the interpolation of the scaled GF and provides a histogram of the interpolated GF amplitude values with the resulting skewness $\tilde{s}_{\text{GF},i}$ for the i-th cycle.

This operation is performed for each glottal cycle in the signal block, thus we obtain $N_c - 1$ GF skewness values, of which median value is taken as a representative value for one signal block. This value is denoted as s_{GF}, which concludes the calculation of the GF's skewness-related measure.

$$s_{\text{GF}} = \text{median}\left\{ \begin{bmatrix} \tilde{s}_{\text{GF},1} & \cdots & \tilde{s}_{\text{GF},i} & \cdots & \tilde{s}_{\text{GF},N_c-1} \end{bmatrix} \right\} \tag{3.78}$$

Justification of Voice Quality Features. The two skewness measures, which are (1) the skewness of the dGF amplitude values s_{dGF} and (2) the skewness-related measure s_{GF} of the GF, are considered as *features* of a signal block. They span a two-dimensional feature space, in which different voice quality clusters can be determined. Figure 3.25 shows the voice quality clustering of the synthesized ground truth. The true dGF and the true GCIs of $N_{\text{it}} = 1000$ signal realizations for each voice quality with a duration of $0.5\,\text{s}$ were used for the skewness calculations. Therefore, all of the following fundamental frequencies f_0 have been used for the synthesis.

$$f_0 \in \mathcal{F}_0 = \Big\{ 70, \quad 120, \quad 170, \quad 220, \quad 270, \quad 320, \quad 370, \quad 420, \quad 470, \quad 520 \Big\}\,\text{Hz} \tag{3.79}$$

From Figure 3.25 it is clear, that a voice quality classification based on the chosen features is possible. The different excitation signal forms determining the voice quality of a sung vocal signal according to the LF-model mentioned in subsection 2.1.1 seems to be represented by the proposed features and a classification within this feature space should be possible, at least if the skewness features are evaluated for the *ground truth* dGF and GCIs. In the next section, a the algorithm performance is analyzed in order to evaluate the clustering using estimations for both dGF and GCIs. In section 3.4 the dataset containing the skewness features calculated from the *ground truth* dGF and GCIs are referred to as the *ground truth* dataset.

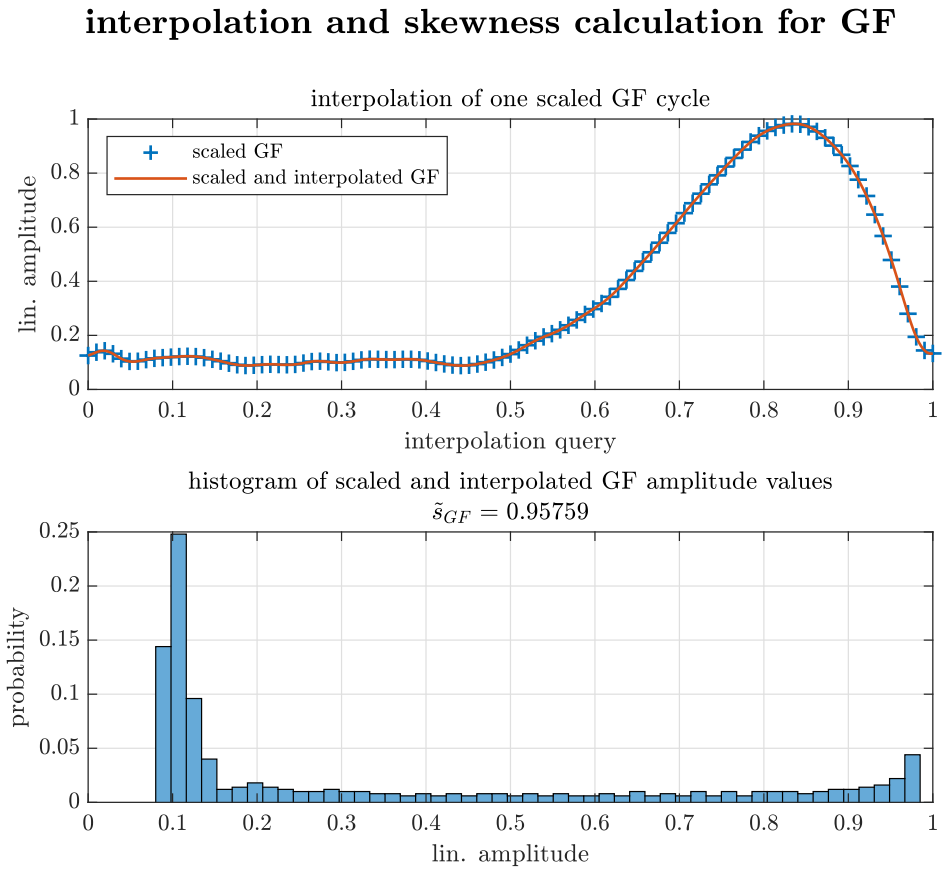


Figure 3.24 *Calculation of skewness $\tilde{s}_{GF}$ of scaled and interpolated GF for the fourth cycle from 3.23*

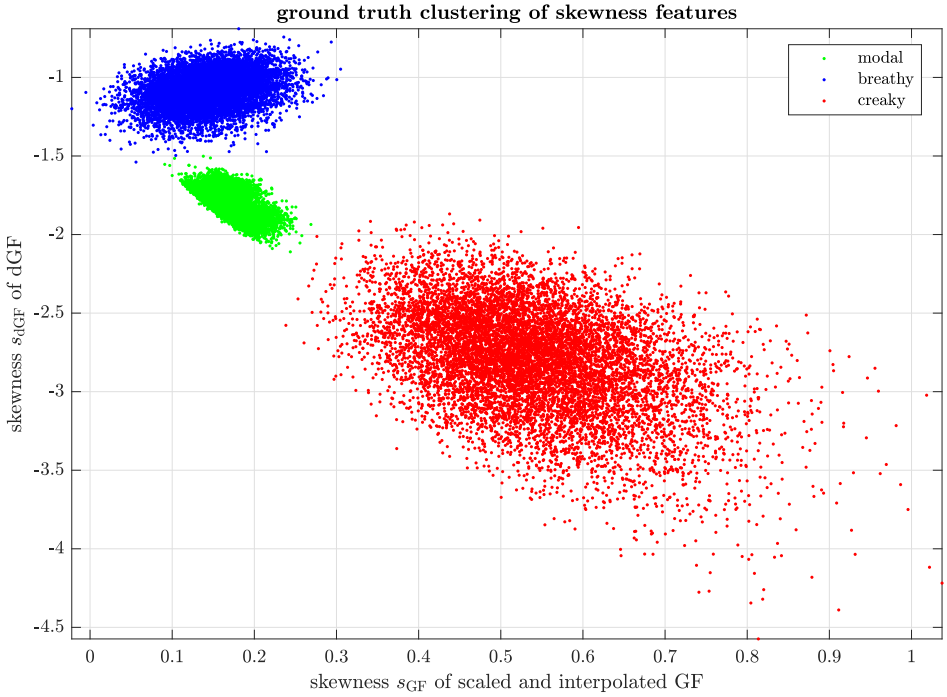


Figure 3.25 *Feature space clustering of the ground truth for all $f_0 \in \mathcal{F}_0$*

3.4 Performance Analysis of the LP Algorithms

The aim of the performance analysis is to achieve a educated decision on which analysis algorithm is selected for the implementation in C++/JUCE. It is expected that the analysis algorithms perform differently for varying prescribed parameters of fundamental frequency, vowel and voice quality. Therefore, this section aims to quantify the performance of each analysis algorithm based on (i) the *estimation error for formant frequencies* as a measure for the estimation quality of the vocal tract filter and (ii) the *clustering of voice quality features* by means of the prediction performance for voice quality. The following analysis algorithms from section 3.2 are compared performance-wise:

- *autocorrelation method*, as described in subsection 3.2.1,
- *autocorrelation method with cepstral refinement*, as described in subsection 3.2.2,
- *covariance method*, as described in subsection 3.2.3 and
- *windowed covariance (SLP)*, as described in subsection 3.2.4.

To account for the synthesizer variability as defined in Table 2.1, a dataset is needed. The dataset consists of $N_{\text{it}} = 100$ realizations with a duration of $0.5\,\text{s}$ for each *combination* of

- voice quality (modal, breathy, creaky)
- vowel (*/a/*, */e/*, */i/*, */o/*, */u/*)
- fundamental frequency $f_0 \in \mathcal{F}_0$ where $\mathcal{F}_0$ is a set of 10 fundamental frequency values defined as

$$\begin{aligned}\mathcal{F}_0 &= \Big\{70,\quad 120,\quad 170,\quad 220,\quad 270,\quad 320,\quad 370,\quad 420,\quad 470,\quad 520\Big\}\,\text{Hz} \\ &= \Big\{f_{0,i}\Big\}_{i=1}^{10}\,.\end{aligned} \tag{3.80}$$

Table 3.1 lists the size parameters of the dataset.

Table 3.1 *Size of the dataset*

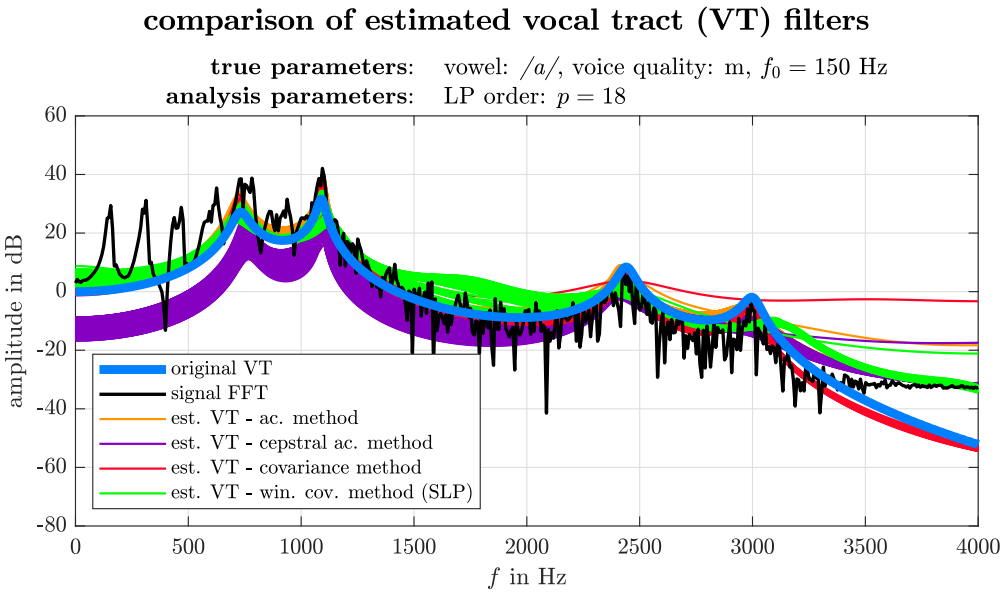
Parameter	**Amount**
number of fundamental frequencies f_0 (corresponds to the number of elements in $\mathcal{F}_0$)	$N_{f_0} = 10$
number of voice qualities	$N_{\text{VQ}} = 3$
number of vowels	$N_{\text{vow}} = 5$
number of realizations for a single combination of f_0, vowel and voice quality	$N_{\text{it}} = 100$
total number of realizations (each with a duration of $0.5\,\text{s}$)	$N_{\text{dataset}} = 15000$

3.4.1 Algorithm Performance on Formant Estimation

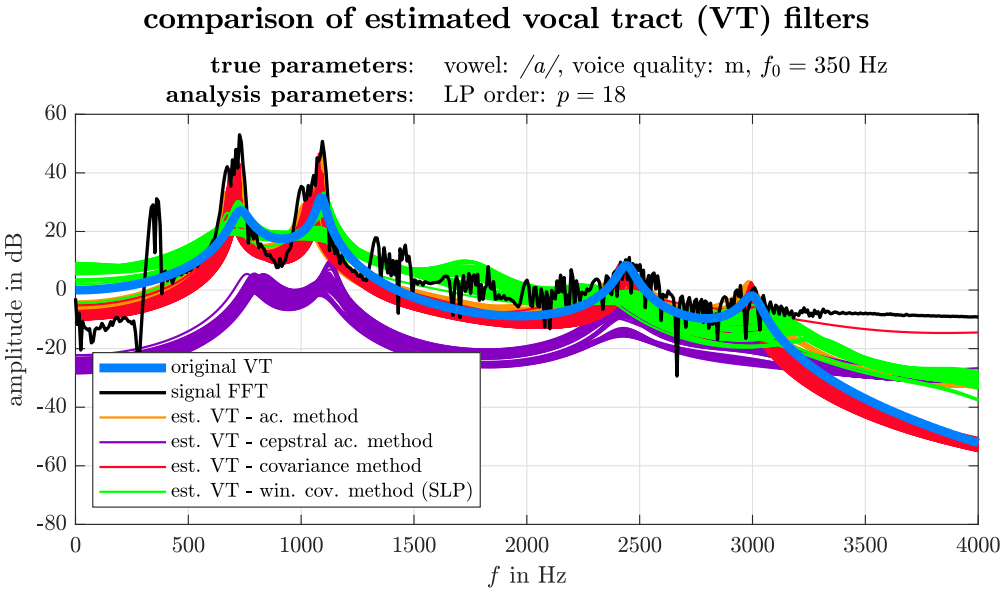
Each estimation algorithm results in a set of vocal tract filter coefficients which define the location (i.e. frequency) and shape (i.e. bandwidth) of poles of the vocal tract filter, that are interpreted as the formants of the present vocal tract. In Figure 3.26 (a) we see the amplitude of the estimated

vocal tract filters in frequency domain for a sung vocal signal with vowel */a/*, modal voice quality and fundamental frequency $f_0 = 150\,\text{Hz}$. The estimated vocal tracts for other vowels are displayed in section A.2.

In Figure 3.26 (b), the vocal tract filter estimation results for the vowel */a/* with modal voice quality and a fundamental frequency $f_0 = 350\,\text{Hz}$ are displayed. Looking at Figure 3.26 (b) it is clear that the vocal tract filter estimation for $f_0 = 350\,\text{Hz}$ did not work as well as the estimations visualized in Figure 3.26 (a), where $f_0 = 150\,\text{Hz}$ was used. This already indicates a possible limitation of the analysis algorithms, which will be discussed in subsection 3.4.3.



(a) fundamental frequency $f_0 = 150\,\text{Hz}$



(b) fundamental frequency $f_0 = 350\,\text{Hz}$

Figure 3.26 *Comparison of estimated vocal tract filters for the vowel /a/ with modal voice quality and fundamental frequencies* $f_0 \in \{150,\ 350\}\,\text{Hz}$

In order to quantify the error of the vocal tract filter estimation results compared to the true vocal tract filter, a formant error measure is introduced. This error measure considers the estimated vocal tract filter's formants, which are calculated with the algorithm described in subsection 3.3.2.

Formant Error Measure. The post-processing stage provides an estimate of the formants $\hat{F}_i$, as described in subsection 3.3.2. Using the ground truth formants F_i set in the synthesizer, a formant error measure e_F in percent can be defined, such that

$$e_F = \frac{100}{N_F} \sum_{i=1}^{N_F} \frac{\left| F_i - \hat{F}_i \right|}{F_i} \quad \%, \tag{3.81}$$

where N_F is the number of estimated formants. The formant error measure e_F is used for both variants of the performance analysis in the following paragraphs.

It can be expected that the algorithms exhibit a certain amount of variability, originating from inacurracies and uncertainties in the pre-processing stage as described in section 3.2. Additionally, due to the variability in the dGF synthesis parameters (see Table 2.1), variations between different realizations of one parameter set (voice quality, vowel and fundamental frequency) are expected. Therefore, two variants of the performance analysis are carried out in this subsection:

(i) analysis of the algorithm variability *within single realizations* using long signal durations and

(ii) analysis of the algorithm variability *between multiple realizations* with short signal durations.

Variability within single realization. In order to analyze the algorithm variability within single realizations a dataset, consisting of a single, $10\,\mathrm{s}$ long realization for each combination of (i) fundamental frequency in the set $\mathcal{F}_0$ as defined in Equation 3.80, (ii) voice quality and (iii) vowel is created. The formant error measure is evaluated for each signal block of each sung vocal signal in this dataset.

To visualize the distribution of formant errors for each fundamental frequency, *violin plots* are created using [28]. The formant error measure violin plots for the vowel */a/* are shown in Figure 3.27. The violin plots of the other vowels, can be found in the appendix, section A.3.

From Figure 3.27 and the corresponding figures for other vowels in section A.3 it can be concluded, that the within-signal variability shows the least effects for the autocorrelation method using cepstral refinement. Therefore, regarding this aspect, the *autocorrelation method with cepstral refinement* is preferred.

Variability between different realizations. The formant error measure e_F as defined in Equation 3.81 is evaluated using the first $N_F = 4$ estimated formants. The formant estimation is calculated for all realizations of the test dataset described in Table 3.1. For each test data realization, the formant error is evaluated.

The formant error measure violin plots for the vowel */a/* are shown in Figure 3.28. The violin plots of the other vowels, can again be found in the appendix, section A.4.

From the formant error measure evaluations for multiple realizations displayed in Figure 3.28 and the corresponding figures in section A.4, we see that the autocorrelation method with cepstral refinement and the covariance method perform best in the range of $f_0 \in [70, 320]$. Therefore, regarding the variation between signal realizations, these two algorithms show the best results.

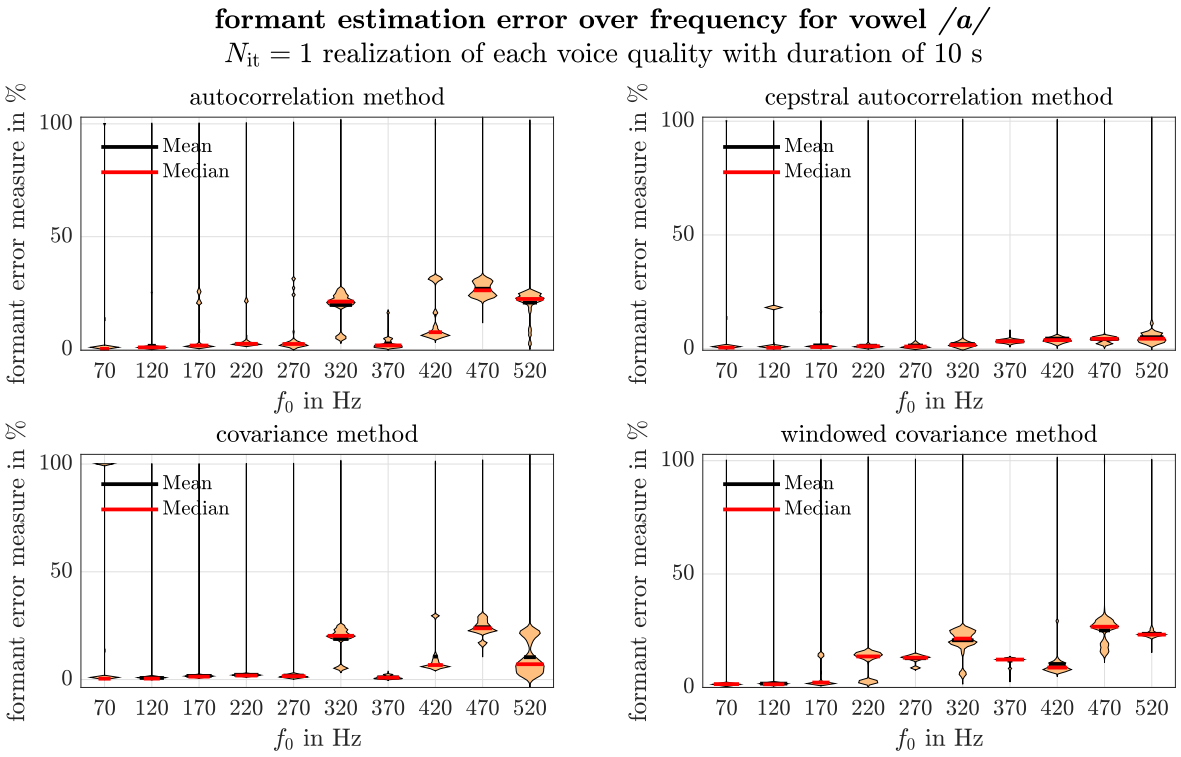


Figure 3.27 *Error of formant estimation depending on the fundamental frequency f_0 for the four algorithms with vowel /a/ (single realization)*

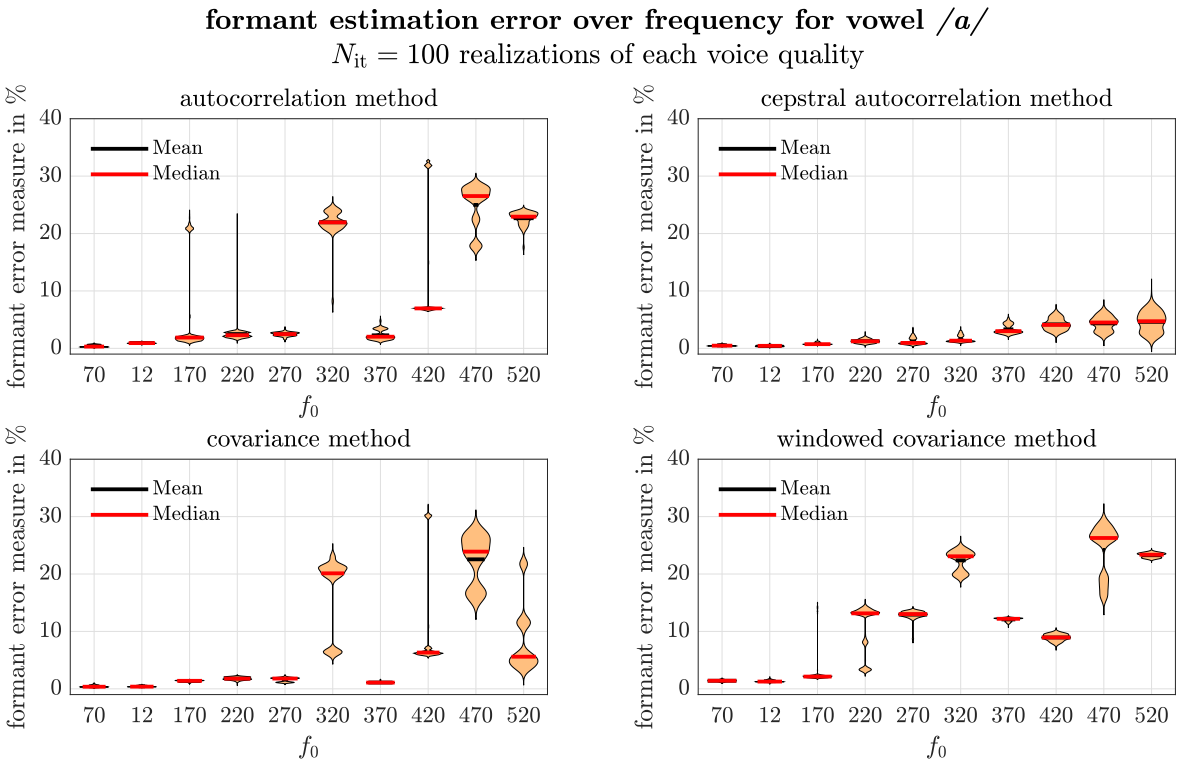


Figure 3.28 *Error of formant estimation depending on the fundamental frequency f_0 for the four algorithms with vowel /a/ (multiple realizations)*

3.4.2 Algorithm Performance on Voice Quality Variability

The algorithm, which calculates the voice quality features (i) *skewness of dGF amplitude values* and (ii) *skewness-related measure of GF* is described in subsection 3.3.4. In the following subsection, the clustering of the estimated features within the two-dimesional feature space are evaluated. Essentially, the clusters obtained by calculating the skewness features for the ground truth dGF as displayed in Figure 3.25 are compared with the clusters given by the features calculated with the proposed analysis methods from section 3.2. Based on the frequency dependent clustering, a frequency range is selected for which the clusters are deemed sufficiently compact, in order to achieve a meaningful classification. Therefore, the following steps are executed:

1. train a supervised machine learning model (e.g. *support vector machine*) for a given frequency range
2. evaluate the prediction error on the training and test set
3. choose a frequency range, that is as large as possible, while at the same time maintains a high percentage of correctly classified data samples

Supervised Learning with Support Vector Machines. Due to the input parameters of the proposed synthesizer, there is full knowledge of the synthesized vocal sound's voice quality. Therefore, one can create a large enough number of sung vocal signals with defined voice quality and evaluate the two voice quality features. This method is known as a *Monte Carlo* simulation, as described in [58, p. 285], and it allows to approximate the features' probability density function (PDF) by evaluating the features for a large number of sung vocal signal realizations. For this large number of sung vocal signal realizations, a classification is obtained by a *support vector machine* (SVM) with a binary classification for each class [3, Ch. 7.1.3]. An implementation of SVMs is provided by `Matlab`'s `fitcsvm()`-command [48]. For the SVM training using the training dataset (see below), the following parameters were used:

- second order polynomial kernel function[5]
- `Standardize` $= 1$
- `BoxConstraint` $= 1$

Subsequently, the predicted class was evaluated for the training and test dataset seperately using `predict()`. Comparing the predicted label with the true label enables the calculation of the percentage of samples that have been classified correctly. Depending on the data the percentages are called the *score on the training set* or *score on the test set*.

Creation of Training and Test Sets. We expect the clustering to be dependent on the fundamental frequency range, more precisely it is expected that the clustering will be more unstable with an increasing upper frequency limit of the fundamental frequency range. Therefore, ten different datasets were used iteratively, where in each iteration the dataset was expanded with the data for the next higher fundamental frequency.

The a dataset of the k-th iteration consideres all fundamental frequencies contained in the set $\mathcal{F}_{0,k}$,

[5] Alternatively, *radial basis functions* could be used while achieving a similar classification quality, but they were not considered in this project.

which is a subset of the set $\mathcal{F}_0$ containing all fundamental frequencies, as defined in Equation 3.80.

$$\begin{aligned} \mathcal{F}_{0,k} &= \left\{ f_{0,i} \right\}_{i=1}^{k} \qquad \text{with} \qquad k = 1, \ldots, 10 \\ \text{for } k = 10: \qquad & \mathcal{F}_{0,10} = \mathcal{F}_0 \end{aligned} \tag{3.82}$$

In total, the size of the dataset N_{tot} can be calculated with

$$N_{\text{tot}} = N_{f_0} \cdot N_{\text{VQ}} \cdot N_{\text{vow}} \cdot N_{\text{it}}, \tag{3.83}$$

where $N_{f_0} = k$ is the number of fundamental frequencies considered for the k-th iteration, $N_{\text{VQ}} = 3$ is the number of different voice qualities, $N_{\text{vow}} = 5$ is the number of different vowels, and $N_{\text{it}} = 100$ the number of realizations according to Table 3.1.

For example, the dataset for $f_0 = 320\,\text{Hz}$ uses estimated features of the following fundamental frequencies

$$\mathcal{F}_{0,6} = \left\{ f_{0,i} \right\}_{i=1}^{6} = \left\{ 70, \quad 120, \quad 170, \quad 220, \quad 270, \quad 320 \right\} \text{Hz}. \tag{3.84}$$

Therefore, the dataset for $f_0 = 320\,\text{Hz}$ has the following size

$$N_{\text{tot}} = N_{f_0} \cdot N_{\text{VQ}} \cdot N_{\text{vow}} \cdot N_{\text{it}} = 2 \cdot 6 \cdot 3 \cdot 5 \cdot 100 = 9000 \text{ data points} \tag{3.85}$$

This dataset consisting of skewness values evaluated from the estimated dGF (so-called *measurements*) was subdivided into $80\,\%$ training data and $20\,\%$ test data.

In addition to the feature data samples derived from the estimated dGF, the same amount of feature data samples from the *ground truth* dataset were used in order to stabilze the clustering. The data samples taken from the *ground truth* skewness dataset, mentioned in subsection 3.3.4, were also split into $80\,\%$ training data and $20\,\%$ testing data.

Both the *measurement* and the *ground truth* training datasets together are used to train the SVM, whereas both test datasets together are used to evaluate the SVM prediction performance.

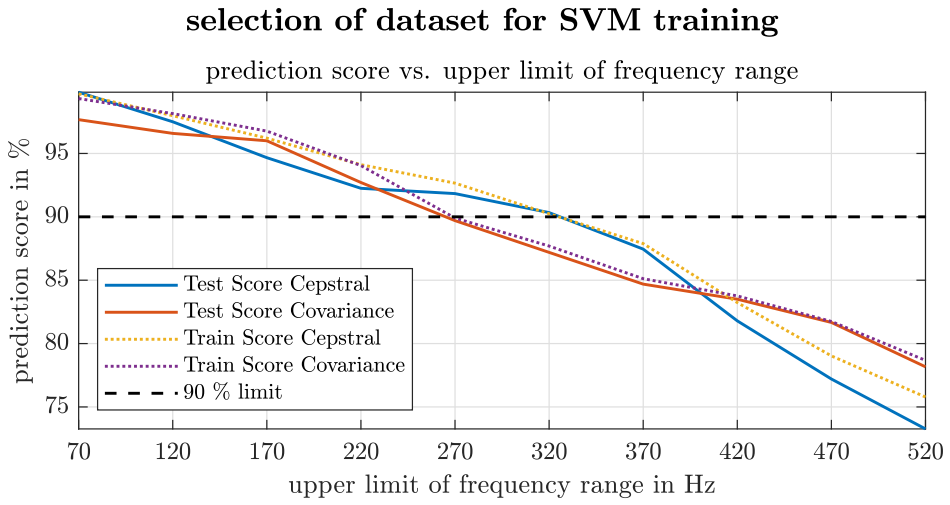


Figure 3.29 *Prediction score of test and training datasets in dependence on the frequency range's upper limit. Comparison of cepstral autocorrelation method and covariance method.*

Evaluation of Prediction Score of SVM. In Figure 3.29, the score on the training and test set is displayed for varying upper limits of the frequency range. Ideally, all frequencies should yield an optimal training and test score. However, due to algorithm limitations, the clustering becomes more unstable, as the clusters start to rotate and show multimodal behaviour within the feature space for higher upper frequency limits of the fundamental frequency range. This can be observed, especially for the *creaky* voice's estimated skewness features. Therefore, a test score limit of more than $90\,\%$ was considered to be sufficient. The cepstral autocorrelation method achieves this score for fundamental frequencies up to $f_0 = 320\,\mathrm{Hz}$, as it can be seen in Figure 3.29.

From Figure 3.29 it is clear, that in the frequency range of $f_0 \in [70, 320]$, the *covariance method* is outperformed by the *autocorrelation method with cepstral refinement* with respect to the prediction performance on both training and test datasets. Thus, the latter is considered best concerning this aspect.

The SVM's clustering result for fundamental frequencies of the set $\mathcal{F}_{0,6}$ is displayed in Figure 3.30. The clustering results for the other fundamental frequency sets can be found in section A.5.

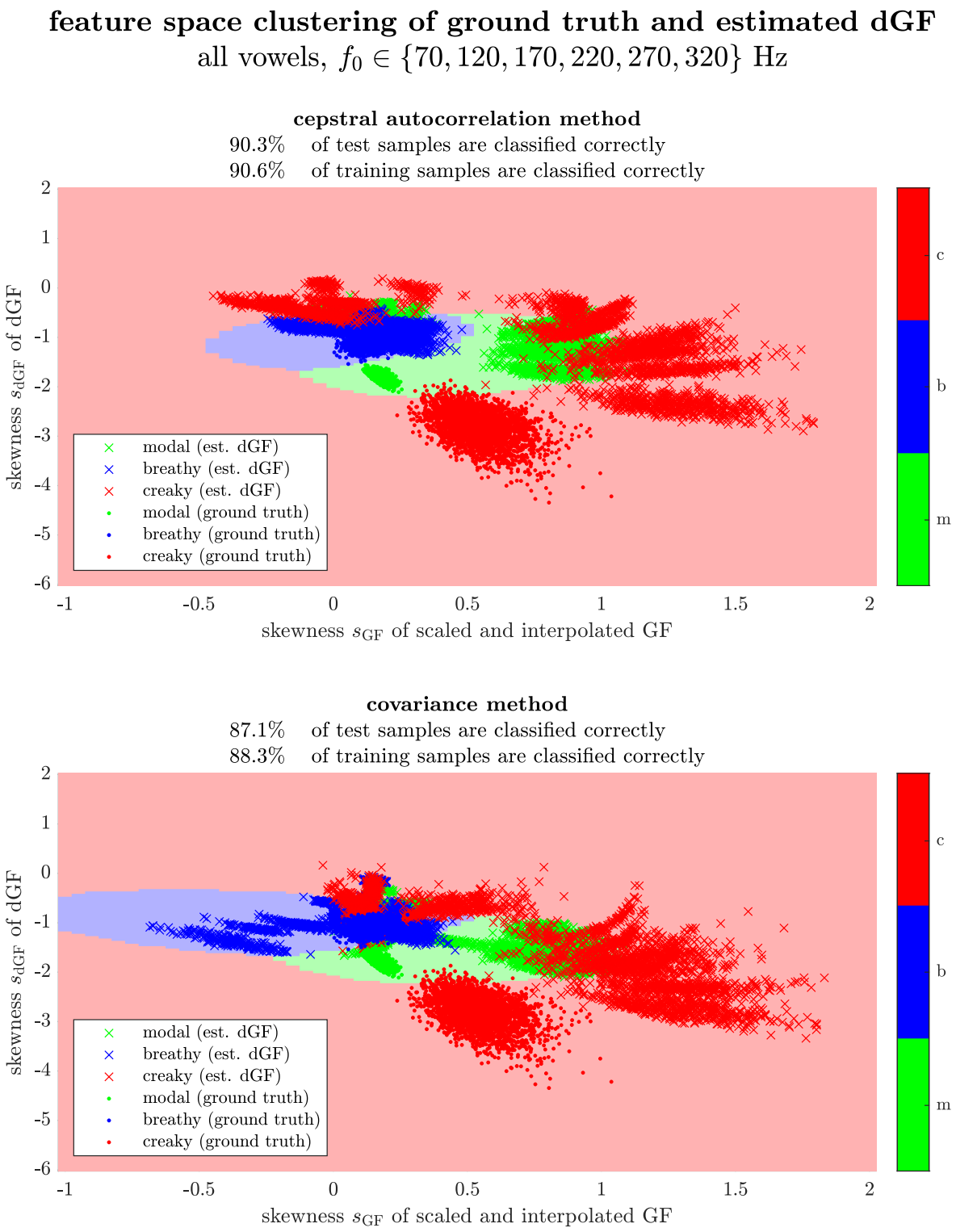


Figure 3.30 *Clustering of voice quality features considering fundamental frequencies up to* $f_0 =$ 320 Hz. *Comparison of cepstral autocorrelation method and covariance method.*

3.4.3 Conclusion on Performance Analysis

The performance analysis was carried out with two goals in mind. The algorithm chosen for the implementation in C++/JUCE should show (i) a good ability in the formant frequency estimation as well as a good performance concerning the voice quality classification, while (ii) these analysis capabilities should be provided over a broad range of fundamental frequencies. These goals are contradictory, meaning that for a increasing range of fundamental frequencies, the estimation and classification performance decreases. This leads to a typical trade-off situation, where a frequency range needs to be limited, in order to ensure a *sufficiently good* algorithm performance. The following conclusions are made based on the previous performance analysis.

In subsection 3.4.1, the algorithms have been analyzed with regard to their variability on the formant estimation *within* long signal realizations, and *between* short signal realizations. From this analysis it can be concluded, that the *autocorrelation method with cepstral refinement* performs best, followed by the *covariance method*.

Looking at the performance analysis with respect to the the voice quality classification and the clustering within the feature space in subsection 3.4.2 it can be concluded that a clustering can be considered stable within the fundamental frequency range of $f_0 \in [70, 320]\,\mathrm{Hz}$. By evaluating the SVM's score on the training data as shown in Figure 3.29, for the skewness features calculated using the *autocorrelation method with cepstral refinement* and the *covariance method*, it can be concluded that the *autocorrelation method with cepstral refinement* outperforms the *covariance method* within the limited frequency range.

Decision for Implementation. Based on the performance analysis of the LPC algorithms, it can be concluded that the *autocorrelation method with cepstral refinement* performs best in the frequency range $f_0 \in [70, 320]\,\mathrm{Hz}$. Therefore, this algorithm is chosen for the implementation in JUCE, which is described in the following chapter.

4 Implementation in C++ using the JUCE-Framework

From the considerations following the performance analysis in section 3.4, we conclude that the autocorrelation method using the cepstral refinement, as described in subsection 3.2.2 performs best, which is why this method has been chosen to be implemented as a VST-plugin using the C++ framework *JUCE* by Storer *et al.* [65].

The analysis and classification algorithm proposed in chapter 3 is based on a block-wise process chain. Due to the block-wise nature of the algorithm an implementation within the JUCE-framework is possible. In order to enable a block-wise signal flow the buffer structure of a recent plugin implementation by Holzmüller *et al.* [29], concerning a real-time capable Constant-Q-transformation was used. Their code includes a buffer structure, enabling block processing which is reused in this project for the execution of the linear prediction algorithm. In contrast to Holzmüller *et al.*, the proposed implementation is not focused on performance and resource-efficiency, but rather on a proof of concept, to implement a real-time application of the proposed algorithm.

In the following sections each vital part of the analysis algorithm implemented using the JUCE-framework is discussed. Also the differences between the implementation in `Matlab` and the difficulties that arose using the JUCE-framework are highlighted.

Before the algorithm implementation is discussed the necessary libraries as well as all implemented classes and code files are listed and shortly discussed.

4.1 Necessary libraries, functions, C++ Classes and Code Files

Besides the libraries and functions already included in the JUCE plug-in template and the folder "resources" provided by Institute of Electronic Music and Acoustics (IEM) in [61], there are additional libraries and functions necessary as listed in Table 4.1.

Table 4.1 *listing of necessary libraries and functions*

Function/Library	Area of Application	Reference
`Eigen` library	formant detection based on polynomial roots	[26]
`peakfinder` function	analysis pre-processing - glottal instant detection	[73]
`fftw` library	Fast Fourier Transform for Windows operating systems	[20]
`spline.h` interpolation function	spline interpolation of glottal flow amplitude values	[37]

To provide an overview of the used C++ classes, they are listed in the following. Thereby, one C++ class is implemented using a header file (`*.h`) and an implementation file (`*.cpp`).

- `PluginProcessor`: This is a JUCE class template and represents the entry point of the plug-

in [30, p.12]. It creates all the other classes and also executes the anti-aliasing filtering necessary for downsampling.

- `PluginEditor`: Is also a JUCE class template and is responsible for the creation of the GUI and the visualizer components, indicating the classification results.
- `LPThread`: This class constitutes the main routine of the plug-in. In it, the block-wise signal processing chain concerning the analysis and low-level feature calculation is completed.
- `ClassifyVisualizer`: Creates the classificiation result visualization.

In addition to the functions libraries and class files, it has to be ensured that the file `DecFiltCoeff.h` is placed in the code file's parent work directory. It contains the anti-aliasing filter coefficients and down-sampling factor as mentioned in subsection 4.2.1.

The C++ code files are available at IEM Cloud in the folder `00_ABGABE_JUCE/LPAVoiceQualEval` (restricted access) or via IEM GIT at `https://git.iem.at/PAB/lpa_voice_qual_eval` (unrestricted access).

4.2 Pre-Processing, Analysis and Classification implementation

In the following section the implementation of the voice-quality and formant analysis algorithm proposed in chapter 3 within the C++ based framework JUCE is discussed. Firstly, an overview on the data flow through the plugin and on the implementation of downsampling and signal blocking is given. As parts of the downsampling are executed in the `PluginProcessor` class, the subsequent subsections dealing with the pre-processing, analysis and classification of sung vocal signals are all executed in `LPThread`.

4.2.1 Data Flow, Signal Blocking and Downsampling

Data Flow. Nearly the same buffer structure as in [29] was chosen to process the audio samples into classification results. Thus, similar to [30, fig. 5], the data flow structure is given in Figure 4.1.

In contrast to the buffer structure implemented by Holzmüller *et al.* in [29], where only *one* multichannel output buffer was implemented, now *three* output buffers for the estimation results are present, as shown in Figure 4.1. The output buffer containing the fundamental frequency estimates for each signal block, is a single channel structure, whereas the formant and voice-quality buffer contain two channels. The data pushed onto the formant buffer, are the two estimated formant frequencies, calculated as mentioned in subsection 3.3.3. The two skewness measures from subsection 3.3.4 are pushed onto the voice quality buffer.

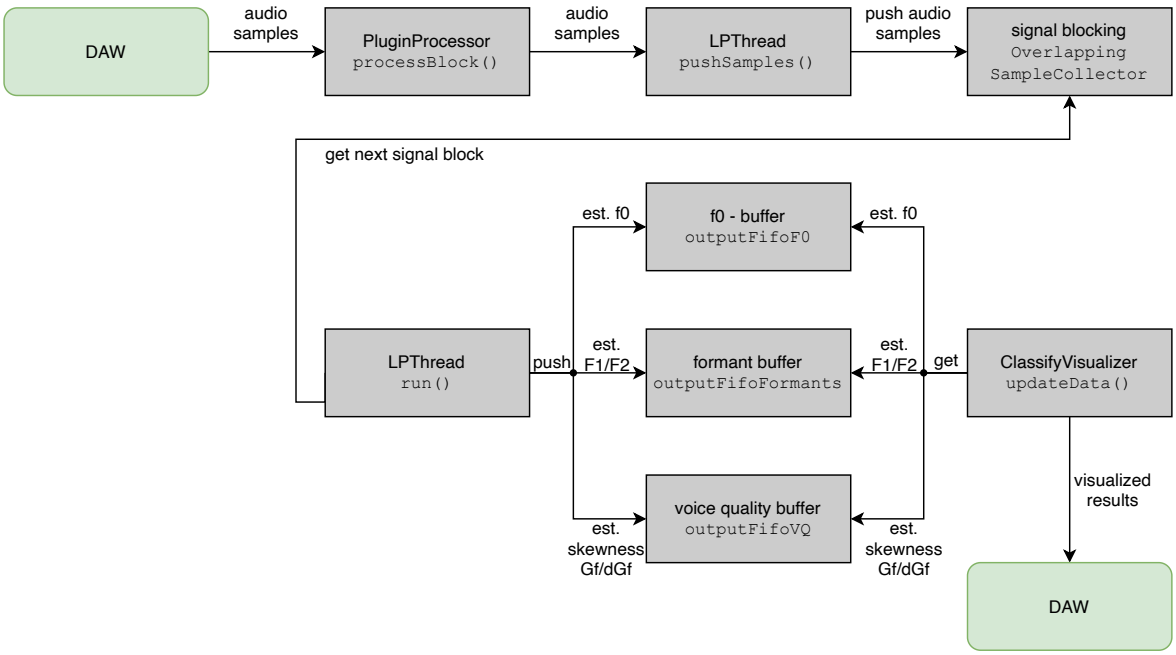


Figure 4.1 *Data flow through the plug in implementation*

Downsampling and Signal Blocking. The block-wise signal processing chain starts with the anti-aliasing filter, which is implemented in the `processBlock()` routine of the `PluginProcessor` class. The audio samples handed to `processBlock()` via the passed `buffer` object, are processed through a 30-th order FIR-Filter. The FIR filter object is instantiated in the constuctor of the `PluginProcessor` class, where also the filter coefficients are loaded. Its coefficients were calculated in `Matlab` using the `fir1()` command [47] and copied into a `*.h`-file. The filter coefficients and the

downsampling factor are stored in the file `DecFiltCoeff.h` located in the folder `00_ABGABE_JUCE/LPAVoiceQualEval/`, from which the filter coefficients are read in and passed to the filter object using the `.coefficients()` member variable of the `dsp::FIR::Filter<float>` object [34]. From the `PluginProcessor`, the audio-samples are then pushed onto a `OverlappingSampleCollector` object, in which the signal gets blocked into blocks with a duration of $80\,\mathrm{ms}$ and an overlap of $70\,\%$. Note, that the blocking is still executed using the initial sampling rate. The actual downsampling is the first step of the `LPThread` class' `.run()` routine. The downsampling factor stored in `DecFiltCoeff.h` determines, which samples of a signal block are kept and which are neglected. In the implementation, a downsampling factor of three was used, meaning every third sample of a signal block is used and stored into a new `std::vector<float>` object, called `LPdataVec`.

`LPdataVec` represents a downsampled signal block which is then passed through the various processing steps discussed in chapter 3. The following subsections provide a short introduction on how the analysis steps introduced in section 3.1, section 3.2 and section 3.3 were realized in C++/JUCE.

4.2.2 Pre-Processing

The pre-processing stage comprises the computation of the LP residual, the fundamental frequency estimation, and the voiced/unvoiced decision as well as the GCI detection. Before the LP residual is computed, the downsampled signal block is freed of its mean and normalized. For the normalization, the separate routine `LPThread::normalizeVec()` was created, which returns the normalized downsampled signal block. The following paragraphs deal with the remaining pre-processing steps, which are all executed, within `LPThread::run()`.

Computation of the LP residual. In accordance to subsection 3.1.1, a *rough LP analysis* of the downsampled signal block is obtained. The rough LP includes the calculation of the non-whitened signal block's autocorrelation function, and the execution of the Levinson-Durbin recursion mentioned in subsection 3.1.1. The subroutine `LPThread::calcAutoCorr()` is, where the autocorrelation function is calculated in JUCE according to Equation 3.41 using the JUCE class `dsp::FFT` [33]. The subroutine `LPThread::calcAutoCorr()` is JUCE's analogue of the `Matlab` function `calcAutoCorr()` found in the folder `00_ABGABE_Matlab/V12b_LPA_JUCE_Matlab_Reference/` and its core is shown in Listing 4.5. The rough estimation of the vocal tract is then obtained from the autocorelation function by solving the Levinson-Durbin recursion. The `Matlab` implementation's function `levinsondurbin.m` was replicated in C++/JUCE with the subroutine `LPThread::levinsonDurbin()`.

It is important to note, that for the C++/JUCE implementation of the Levinson-Durbin algorithm a regularization is necessary. For the rough pre-processing linear prediction a regularization term of $\epsilon = 0.01$ is added onto the main diagonal of the autocorrelation matrix $\boldsymbol{R}_{ss}$. This means, in the practical implementation the following adapted version of the Yule-Walker equation system from Equation 3.40 is solved to estimate the filter coefficients $\hat{\boldsymbol{a}}_{\mathrm{opt}}$.

$$\hat{\boldsymbol{a}}_{\mathrm{opt}} = \left(\boldsymbol{R}_{ss} + \boldsymbol{I}\epsilon\right)^{-1} \boldsymbol{r}_{ss+1}, \tag{4.1}$$

where $\boldsymbol{I}$ is the identity matrix. The regularization is necessary, because the estimated filter coefficients surpassed the boundary of numeric stability due to the *single precision* float processing in JUCE. This leads to estimated filter coefficients, whose roots are located *outside* the complex unit circle, resulting in unstable vocal tract filter estimates. Especially the limitation of JUCE's `dsp::FFT` object to single precision lead to more inaccurate autocorrelation function estimates than in `Matlab`, where the sample type default is *double precision* float. Another driving factor of those instabilities is the

fact, that for the rough LP within the pre-processing, no signal whitening by means of a pre-emphasis filter is executed. Therefore, the used synthesized sung vocal signals including its excitation signal are far from the linear prediction's ideal excitation signal, which would be a Gaussian white noise signal [71, p.230].

After solving the regularized Yule-Walker equation system with the `LPThread` class' subroutine `levinsonDurbin()`, which returns the estimated VT filter coefficients and the VT filter gain. The downsampled signal block is inversely filtered using `LPThread::inverseFiltering`, in order to obtain the *residual signal*. The inverse filtering application is done with an FIR filter object using the class `dsp::FIR::Filter<SampleType>` [34]. Initially, two FIR filter objects are instantiated, one to apply the inverse filtering of the rough VT filter estimate within the pre-processing stage, and the other one for the more accurate VT filter estimation discussed in subsection 4.2.3. The core of the inverse filtering and the handling of the FIR Filter objects in JUCE is shown in Listing 4.1.

Listing 4.1 *inverse filtering implemented in `LPThread::inverseFiltering()`, line 223–232 of `LPThread.cpp`*

```
223     dsp::AudioBlock<float> audioBlock (tempbuffer);
224     dsp::ProcessContextReplacing<float> context (audioBlock);
225
226     if (roughFlag)
227     {
228         InvVTFiltRough.reset();
229         dsp::FIR::Coefficients<float>::Ptr FIRCoeffs = new dsp::FIR::Coefficients<float> (a, numCoeffs+1);
230         *InvVTFiltRough.coefficients = *FIRCoeffs;
231         InvVTFiltRough.process(context);
232     }
233     else
234     {
235         InvVTFilt.reset();
236         dsp::FIR::Coefficients<float>::Ptr FIRCoeffs = new dsp::FIR::Coefficients<float> (a, numCoeffs+1);
237         *InvVTFilt.coefficients = *FIRCoeffs;
238         InvVTFilt.process(context);
239     }
```

The first half of the `if`-condition shown in Listing 4.1 shows the implementation of the inverse filtering using the rough VT filter estimate with the FIR filter object. Firstly, the new coefficients are assigned to the FIR Filter object. Then, the audio block, which has to be available in the form of a `dsp::ProcessContextReplacing` structure [35] named `context` is inversely filtered using the `.process()` method. The same method is used for the if-condition's second half, shown in Listing 4.1, which processes the inverse filtering with the more accurate VT filter estimate derived in subsection 4.2.3.

The final step of `LPThread::inverseFiltering()` is to store the residual signal into the output vector, whose memory address is handed to the subroutine. Back in the `run()` method, the residual signal is normalized before the next pre-processing steps are executed.

Fundamental Frequency Estimation and Voiced/Unvoiced Detection. The estimation of the fundamental frequency $\hat{f}_0$ of the signal block and the voiced/unvoiced decision that comes with it is performed in `LPThread::getF0andVUV()`. The calculations are the same as implemented in subsection 3.1.2, and the corresponding C++/JUCE implementation is listed in Listing 4.2.

Listing 4.2 *fundamental frequency estimation with `LPThread::getF0andVUV()`, line 297–337 of `LPThread.cpp`*

```
297     fftF0.performRealOnlyForwardTransform(fftInOut.data(),false);
298     float specSum = 0;
299     for (int ii = 0; ii < specMat.size(); ++ii)
300     {
301         specMat[ii] = std::sqrt(std::pow(fftInOut[2*ii],2)+std::pow(fftInOut[2*ii+1],2));
302         specSum = specSum+std::pow(specMat[ii],2);
303     }
304     specSum = std::sqrt(specSum);
305
306     for (int ii = 0; ii < specMat.size(); ++ii)
307         specMat[ii] = specMat[ii]/specSum;
```

```
308
309     auto f0minIdx = getFreqIdx(f0min);
310     auto f0maxIdx = getFreqIdx(f0max);
311     std::vector<float> SRHvec;
312     SRHvec.resize(f0maxIdx-f0minIdx);
313
314     for (int ii = f0minIdx; ii<f0maxIdx; ++ii)
315     {
316         float addTerm = 0.0f;
317         if ((nHarmonics*ii) < params.fsDec/2)
318         {
319             for (int k=2; k <= nHarmonics; ++k)
320             {
321                 addTerm = addTerm + specMat[k*ii]-specMat[juce::roundToInt((k-0.5f)*ii)];
322             }
323             SRHvec[ii-f0minIdx] = specMat[ii]+addTerm;
324         }
325         else
326         {
327             SRHvec[ii-f0minIdx] = specMat[ii];
328         }
329     }
330
331     auto F0idx = std::max_element(SRHvec.begin(),SRHvec.end()) - SRHvec.begin();
332
333     float alphaSRH = 0.5f;
334     SRHcurr = *std::max_element(SRHvec.begin(), SRHvec.end());
335     SRHsmooth = (1.0f-alphaSRH) * SRHprev+alphaSRH*SRHcurr;
336     SRHprev = SRHsmooth;
337     VUVDecision = (SRHsmooth > VUVThresh);
```

As it can be seen from Listing 4.2, the residual signal is written into the `fftInOut` array and transformed to frequency domain with the `.performRealOnlyForwardTransform()` method of a JUCE FFT object, see line 297. The normalization of the spectrum to the overall signal energy is applied in the first `for`-loop between line 299 and 303. The summation according to Equation 3.4 is executed within the `for`-loop implemented from line 314 to 329. Note that JUCE FFT objects only support FFT lengths in the form of $2^{o_{\text{FFT}}}$, where $o_{\text{FFT}} \in \mathbb{N}$ is known as the FFT order. This makes it impossible to use a FFT-length of $N_{\text{FFT}} = f_s$ to obtain the proposed frequency resolution $\Delta f = 1\,\text{Hz}$ mentioned in [14]. For example if the signal is downsampled to $f_s = 16\,000\,\text{Hz}$ the FFT length in JUCE is calculated with $N_{\text{FFT}} = 2^{14} = 16384$. This results in a frequency resolution of:

$$\Delta f = \frac{f_s}{N_{\text{FFT}}} = \frac{16000}{16384} = 0.9766\,\text{Hz} \tag{4.2}$$

for the JUCE implementation.

The maximum value of the corresponding index of the `SRH` array is evaluated and smoothed according to Equation 3.6, in order to obtain the fundamental frequency estimate $\hat{f}_0$ for the current signal block and the corresponding SRH value.

The voiced/unvoiced decision is calculated in the last line of Listing 4.2. Due to the fact, that the estimated fundamental frequency is also made visible in the plug-in implementation, there exists also a smoothed version of the fundamental frequency estimate $\hat{f}_0$ – analogously to Equation 3.6 – in order to prevent extensive fluctuations in the GUI's frequency display. For the frequency smoothing, a smoothing factor of $\alpha = 0.08$ is chosen. If the V/UV decision deems a signal block to be unvoiced, the frequency estimate is set to `NAN`, thus no further processing steps are performed and the next signal block is popped from the buffer queue. The V/UV decision is visible in the algorithm flow chart shown in Figure 4.4.

Glottal Closure Instant Detection. If a signal block is considered to be voiced, the next processing step is the GCI detection. The main routine concerning the GCIs is `LPThread::getGCIs()`, which is executed within the `.run()` method. The subroutine `LPThread::getGCIs()` is the JUCE pendant to the `Matlab` function `getGIs.m`. They are both based on the calculations discussed in subsection 3.1.3. The only difference that occurs, is that the JUCE variant only includes a GCI detection,

because there is no need for GOIs, when using the *autocorrelation method with cepstral refinement*. The first step of the GCI detection is the calculation of the mean-based signal. According to Equation 3.7, the mean-based signal is computed by convoluting the voiced sung vocal signal block with a Blackman window. The JUCE implementation of this operation is shown in Listing 4.3.

Listing 4.3 *calculation of the mean based signal in `LPThread::getGCIs()` , line 377–389 of `LPThread.cpp`*

```
377     auto T0Est = juce::roundToInt(params.fsDec/F0);
378     auto winLen = juce::roundToInt((1.7*T0Est)/2);
379     BlackmanBuf.setSize(1, 2*winLen+1);
380
381     std::vector<float> blackmanWindow;
382     blackmanWindow.clear();
383     blackmanWindow.resize (2*winLen+1);
384     WindowingFunction<float>::fillWindowingTables (BlackmanBuf.getWritePointer(0), 2*winLen+1,
          ↪ WindowingFunction<float>::blackman, false);
385     BlackmanConv.loadImpulseResponse(std::move(BlackmanBuf), params.fsDec, juce::dsp::Convolution::Stereo::no,
          ↪ juce::dsp::Convolution::Trim::no, juce::dsp::Convolution::Normalise::yes);
386
387     dsp::AudioBlock<float> audioBlockGI (tempbufferGI);
388     dsp::ProcessContextReplacing<float> contextGI (audioBlockGI);
389     BlackmanConv.process(contextGI);
```

As it can be seen from Listing 4.3, the convolution is performed with a `dsp::Convolution` object, which is more efficient than doing the same operation with a `dsp::FIR::Filter` object for impulse responses larger than 128 samples [32]. A Blackman window can be created using the JUCE framework's class `dsp::WindowingFunction<float>` and its function `.fillWindowingTables` [36]. Using the method `.loadImpulseResponse()`, the Blackman window is loaded into the convolution object as an impulse response. The convolution itself is again computed with the `.process` method, in conjunction with the audio samples transformed into a `dsp::ProcessContextReplacing` structure [35]. Using the peak finder function `Peaks::peakfinder()` from [73], the minima and maxima of the mean-based signal are evaluated. Afterwards, the refinement using the residual signal is carried out, in order to obtain median relative GCI-positions λ_{GCI} from Equation 3.10. To calculate a median value, the helper function `LPThread::calcMedian()` was written, following [21].

Listing 4.4 *GCI-detection within derived presence intervals in `LPThread::getGCIs()` , line 509–546 of `LPThread.cpp`*

```
509     for (int ii= 0; ii<MinPeaks.size(); ++ii)
510     {
511         float alpha = RatioGCI - 0.35f;
512         start = MinPeaks[ii]+std::round(alpha*(MaxPeaks[ii]-MinPeaks[ii]));
513         alpha = RatioGCI + 0.35f;
514         stop = MinPeaks[ii]+std::round(alpha*(MaxPeaks[ii]-MinPeaks[ii]));
515         if (start < 0)
516             start = 0;
517         else if (start > (res.size()-1))
518             break;
519         if (stop > (res.size()-1))
520             stop = int(res.size()-1);
521         if (stop > 0)
522         {
523             if (start<stop)
524             {
525                 vec.clear();
526                 vec.resize(stop - start + 1);
527                 std::copy(res.begin() + start, res.begin() + stop + 1, vec.begin());
528                 int posi = int(std::max_element(vec.begin(), vec.end())-vec.begin());
529                 //In matlab for -1 is added for GCIPos estimation due to array-indices starting at 1.
530                 GCIPos[Ind] = start+posi;
531                 Ind++;
532             }
533             else
534             {
535                 start = 0;
536                 stop = int(res.size()-1);
537                 vec.clear();
538                 vec.resize(stop - start + 1);
539                 std::copy(res.begin() + start, res.begin() + stop + 1, vec.begin());
540                 int posi = int(std::max_element(vec.begin(), vec.end())-vec.begin());
541                 //In matlab for -1 is added for GCIPos estimation due to array-indices starting at 1.
542                 GCIPos[Ind] = start+posi;
543                 Ind++;
544             }
545         }
546     }
```

Equations 3.11 – 3.14 are then executed in the for-loop shown in Listing 4.4. To ensure a proper GCI detection, there are some `if`-conditions included to be safe, that in case of unfortunate circumstances no negative start positions or stop positions exceeding the signal block length occur.

Pre-Emphasis Filtering. The pre-emphasis filter is simply created as a FIR-filter object in JUCE. The subroutine `LPThread::preEmphFiltering()` processes the signal block in form of a `dsp::ProcessContextReplacing` structure, using the coefficients calculated as mentioned in Equation 3.19.

4.2.3 Linear Prediction Analysis with the Autocorrelation Method Using Cepstral Refinement

The C++/JUCE implementation's linear prediction analysis stage, which in contrast to chapter 3 only comprises the *autocorrelation method with cepstral refinement*, as reasoned in subsection 3.4.3, consists of the following methods executed within the `LPThread::run()` routine:

1. `LPThread::calcAutoCorr()`
2. `LPThread::fftshift()`
3. `LPThread::CepsLift()`
4. `LPThread::levinsonDurbin()`
5. `LPThread::inverseFiltering()`
6. `LPThread::normalizeVec()`

Calculation of the Autocorrelation Function. The first step, which is the calculation of the autocorrelation function, is executed in the same routine, as used in the computation of the rough LP residual in the pre-processing as discussed in subsection 4.2.2. The core of the autocorrelation calculation's implementation in C++/JUCE is shown in Listing 4.5, which makes it clear that a direct implementation of Equation 3.41 was applied.

Listing 4.5 *calculation of the autocorrelation function in `LPThread::calcAutoCorr()`, line 125–140 of `LPThread.cpp`*

```
125     for (int ii = 0; ii < numSamples; ++ii)
126     {
127             fftInOut[ii] = sigBlock[ii]*hannWindow[ii];
128             hannSum = hannSum+hannWindow[ii];
129     }
130
131     float W = hannSum/hannWindow.size(); // window correction term.
132
133     fft.performRealOnlyForwardTransform(fftInOut.data(),false);
134
135     for (int ii = 0; ii < fft.getSize(); ++ii)
136     {
137         fftInOut[2*ii] = std::pow(fftInOut[2*ii]*1/W,2)+std::pow(fftInOut[2*ii+1]*1/W,2);
138         fftInOut[2*ii+1] = 0.0f;
139     }
140     ifft.performRealOnlyInverseTransform(fftInOut.data());
```

Cepstral Refinement of the Autocorrelation Function. The next step is the shifting of the autocorrelation function's zero-lag component into the center of the array. The `Matlab` function `fftshift()` [44] was rebuilt in C++/JUCE using the `std::reverse()` method. By placing the zero-lag component into the autocorrelation array's center, the function is ready to be transformed into the cepstral domain. In order to avoid array entries showing true zeros, an `if`-condition is implemented

in line 608 of Listing 4.6. Within the condition, true zeros are replaced by the smallest numerical floating point value `epsilon()`. This is done in order to ensure, that the logarithm computation included in the cepstrum calculation, according to Equation 3.42, does not result in an overflow (i.e. infintely large negative values), as it is not defined for arguments which equal zero. Line 613 of Listing 4.6 shows the inverse N-point Fourier transformation leading to the cepstrum of the current signal block's autocorrelation function.

Listing 4.6 *cepstral transformation of the autocorrelation function in `LPThread::CepsLift()` , line 604–613 of `LPThread.cpp`*

```
604     fft.performRealOnlyForwardTransform(fftInOut.data(),false);
605     for (int ii = 0; ii < numSamples; ++ii)
606     {
607         fftInOut[2*ii] = std::sqrt(std::pow(fftInOut[2*ii],2)+std::pow(fftInOut[2*ii+1],2));
608         if (fftInOut[2*ii]==0.0f)
609             fftInOut[2*ii] = std::numeric_limits<float>::epsilon();
610         fftInOut[2*ii] = std::log(fftInOut[2*ii]);
611         fftInOut[2*ii+1] = 0.0f;
612     }
613     ifft.performRealOnlyInverseTransform(fftInOut.data());
```

The next step is the creation of the lifter window. As mentioned in subsection 3.2.2, the lifter window is computed as a Tukey window. The function `LPThread::tukeyWin()` returns the Tukey window calculated according to Equation 3.44. Within the `for`-loop implemented in lines 626–630 of Listing 4.7, the window is shifted into the correct position to exhibit the symmetric form exemplarily shown in Figure 3.13. The next for-loop then applies the cepstral lifter, and the routine is concluded by reversing the cepstrum calculation and returning to time domain as formulated in Equation 3.45.

Listing 4.7 *lifter computation as Tukey-window in `LPThread::CepsLift()` line 616–644 of `LPThread.cpp`*

```
616     int LiftLen = juce::roundToInt(1/F0Est*params.fsDec*0.85);
617     std::vector<float> TukeyWin;
618     TukeyWin.clear();
619     TukeyWin.resize(2*LiftLen);
620     tukeyWin(TukeyWin.data(),2*LiftLen, 0.2f );
621
622     // compute Cepstral-Lifter:
623     std::vector<float> CepsLifter;
624     CepsLifter.clear();
625     CepsLifter.resize(fft.getSize());
626     for (int ii = 0; ii<LiftLen; ++ii)
627     {
628         CepsLifter[ii] = TukeyWin[LiftLen+ii];
629         CepsLifter[CepsLifter.size()-1-ii] = TukeyWin[LiftLen-1-ii];
630     }
631
632     // apply Cepstral-Lifter
633     for (int ii = 0; ii<CepsLifter.size(); ++ii)
634         fftInOut[ii] = fftInOut[ii]*CepsLifter[ii];
635
636     // transform back to frequency domain
637     fft.performRealOnlyForwardTransform(fftInOut.data(),false);
638     for (int ii = 0; ii < numSamples; ++ii)
639     {
640         fftInOut[2*ii] = std::exp(fftInOut[2*ii])*std::exp(fftInOut[2*ii+1]);
641         fftInOut[2*ii+1] = 0.0f;
642     }
643
644     ifft.performRealOnlyInverseTransform(fftInOut.data());
```

Vocal Tract Filter Coefficient Estimation and Inverse Filtering. The refined autocorrelation is passed through `LPThread::levinsonDurbin()`, which solves the regularized Yule-Walker equation formulated in Equation 4.1 and delivers the estimated VT filter coeffcients. For the Levinson-Durbin recursion solved at this point of the implementation, a regularization term of $\epsilon = 0.001$ is used. The estimated filter coefficients are then used within the `LPThread::inverseFiltering()` method shown in Listing 4.1 to obtain an estimate on the exciation signal (dGF) in correspondance with Equation 3.58. In contrast to the inverse filtering process executed during the pre-processing stage, the `roughFlag` is now set to `false`, yielding in the usage of the non-rough FIR filter object shown in Listing 4.1.

Finally, the estimated dGF is normalized using `LPThread::normalizeVec()`. The estimated and normalized dGF $\hat{E}[n]$ is then further processed to evaluate the voice quality of the sung vocal signal block, and the estimated VT filter coefficients are used to determine which vowel was sung. The processing steps enabling the evaluation of the voice-quality and vowel are discussed in the following subsection 4.2.4.

4.2.4 Formant Frequency Calculation and Voice Quality Classification

The final stage of the implementation is the classification of voice quality and the detection of the sung vowel based on the estimated formant frequencies F_1 and F_2. This subsection provides an overview on the C++/JUCE implementation's specific topics concerning the classification stage. For the formant frequency calculation and voice quality classification, the following methods are executed within the `LPthread::run()` routine:

1. `LPThread::calcFormants()`
2. `LPThread::calcSkewnessGF()`
3. `LPThread::calcSkewness()`

Formant Frequency Calculation. The estimated vocal tract filter coefficients are used to calculate the formants F_1 and F_2 according to subsection 3.3.2. In order to set up the companion matrix $\boldsymbol{A}$ (see Equation 3.62) and for the calculation of its eigenvalues, the script `roots()` from [9] was used in a modified version. The applied modification includes the transposition of the companion matrix, as described in subsection 3.3.2. To evaluate the eigenvalues of $\boldsymbol{A}$, `Eigen`'s function `eigenvalues()` is used, just as in [9]. These eigenvalues are the zeros $z_{0,k}$ of the estimated the vocal tract filter's denominator polynomial $\nu(z)$, as defined in Equation 3.61. From the complex-valued zeros $z_{0,k}$, the formant frequencies $\hat{F}_i$ and their respective formant bandwidths $\hat{B}_i$ are calculated. It is checked, whether the formants and bandwidths fulfill the criteria defined in Equation 3.65. The two formants with the lowest frequencies (i.e. $\hat{F}_1$ and $\hat{F}_2$), that fulfill the criteria, are smoothed with `LPThread::smoothF()` and the smoothed formant frequency values are pushed onto the output FIFO buffer `outputFifoFormants`.

This algorithm is implemented in `LPThread::calcFormants()`, which itself relies on `LPThread::roots()` for the calculation of the polynomial roots via the companion matrix's eigenvalues.

Voice Quality Classification. Based on the estimated dGF derived by inverse filtering, the two skewness values described in subsection 3.3.4 are calculated. The two skewness measures are (i) the skewness of dGF amplitude values and (ii) the skewness-related measure of the GF, as described in subsection 3.3.4. The *skewness of dGF amplitude values* was implemented in C++/JUCE in the function `LPThread::calcSkewness()`, which relies on `LPThread::calcStd()` and `LPThread::calcMean()`. Essentially, Equation 3.72 is evaluated for the estimated dGF $\hat{E}[n]$ for each signal block, as it can be seen in Listing 4.8.

Listing 4.8 *skewness of dGF amplitude values, line 871–885 of `LPThread.cpp`*

```
871 float LPThread::calcSkewness(std::vector<float> vec, float mu, float sigma)
872 {
873   float skew = 0.0f;
874   std::vector<float> vec3mu;
875   float tempVar1 = 0.0f;
876   float tempVar2 = 0.0f;
877   for (int ii = 0; ii < vec.size(); ii++)
878   {
879     tempVar1 = vec[ii] - mu;
880     tempVar2 = pow(tempVar1, 3.0f) / pow(sigma, 3.0f);
881     vec3mu.push_back(tempVar2);
```

```
882   }
883   skew = calcMean(vec3mu);
884   return skew;
885 }
```

For the second feature used in the voice quality classification, the *skewness-related measure of the GF*, the spline-interpolation implemented in the class `tk::spline` from [37] is necessary. Fundamentally, the C++ function `LPThread::calcSkewnessGF()` is used for the skewness calculation of the GF. In this function, the scaled estimate of the GF $\hat{E}_{\mathrm{GF,sc}}[n]$ is calculated for each glottal cycle using the cumulative sum as defined in Equation 3.75. In line 1029 in Listing 4.9, the interpolation of the scaled GF using the `tk::spline`-object is performed.

Listing 4.9 *skewness of scaled and interpolated GF using `tk::spline` , line 1012–1045 of `LPThread.cpp`*

```
1012  for (int ii = 0; ii < GCIPos.size() - 1; ii++)
1013  {
1014    GCIdist = GCIPos[ii + 1] - GCIPos[ii] + 1; // distance between 2 successive GCIs
1015    if (GCIdist < 3)
1016    {
1017      continue;
1018    }
1019
1020    std::vector<float> xFloat = linspaceVQ(0.0f, 1.0f, GCIPos[ii + 1] - GCIPos[ii] + 1); // sample points
1021    std::vector<double> x(xFloat.begin(), xFloat.end()); // convert to double
1022    std::vector<double> val(GFest.begin() + GCIPos[ii], GFest.begin() + GCIPos[ii + 1]+ int(1)); // sample values;
        ↪ Data type double needed for spline interpolation
1023
1024    tk::spline spline;
1025    spline.set_points(x, val);
1026
1027    for (int jj = 0; jj < nInterp; jj++)
1028    {
1029      interpResult = spline(xq[jj]);
1030      GfInterp.push_back(float(interpResult));
1031    }
1032
1033    scalingFact = trapz(spacing, GfInterp);
1034    for (int jj = 0; jj < GfInterp.size(); jj++)
1035    {
1036      GfInterp[jj] = GfInterp[jj] / scalingFact;
1037    }
1038
1039    muGF = calcMean(GfInterp);
1040    sigmaGF = calcStd(GfInterp, true);
1041    skewGFtemp = calcSkewness(GfInterp, muGF, sigmaGF);
1042    skewGfVec.push_back(skewGFtemp);
1043
1044    GfInterp.clear();
1045  }
1046
1047  if (skewGfVec.size() != 0)
1048  {
1049    skewGF = calcMedian(skewGfVec);
1050  }
```

The scaling to achieve unit area under the GF-curve is performed using `LPThread::trapz()`, which is a replica of `Matlab`'s `trapz()`-command [54]. This can be seen in line 1033 and 1036 of Listing 4.9. In accordance to Equation 3.78, the skewness of the GF is calculated with `LPThread::calcMedian()` in line 1050 of Listing 4.9.

4.2.5 Visual Indication of Voice Quality and Vowel

With the C++ class `ClassifyVisualizer`, the visualization of both vowel and voice quality is handled. The member variable `ClassifyVisualizer::visType` is used to switch between the vowel and voice quality map. If it is set to zero, the `ClassifyVisualizer` object is used as a *vowel* visualizer, and if it is set to one, the `ClassifyVisualizer` object is used to visualize the *voice quality*.

Vowel Visualization for `visType` $= 0$. The `ClassifyVisualizer` objects pops the formant frequency estimation results from the FIFO buffer `outputFifoFormants`. According to this results,

a point is plotted in the F_1/F_2-plane. The background of the F_1/F_2-plane is a `.png`-figure, which provides a coloured map of different vowels according to subsection 3.3.3 and [63, 64]. The `.png`-figure has been generated using the `Matlab`-file `dataSendlmeier.m` in the folder `00_ABGABE_Matlab/Databases/vowels_Sendlmeier/`. Based on the point's location on the background figure, it can be visually assigned to a vowel by the user. In order to provide some insight into the prevoiusly estimated vowels, the current formant estimate is supplemented by the 14 previously computed estimation results.

Voice Quality Visualization for `visType` $= 1$. In order to visualize the voice quality, the `ClassifyVisualizer` object pops the two skewness values s_{GF} and s_{dGF} from the FIFO buffer `outputFifoVQ`. Similar to the vowel visualization, a point is plotted in the $s_{\text{GF}}/s_{\text{dGF}}$-plane. In the background, a `.png`-figure showing the voice quality regions, which were evaluated in advance using the `Matlab` file `00_ABGABE_Matlab/V14_LPA_Matlab_Classification/Main_featureSpace_clustering_SVM_freqRange.m`, is displayed. Based on the point's location on the background figure, a visual indication of voice quality is provided. Again, the current point is complemented by the last 14 points in order to provide an insight into the previous block's voice quality estimates.

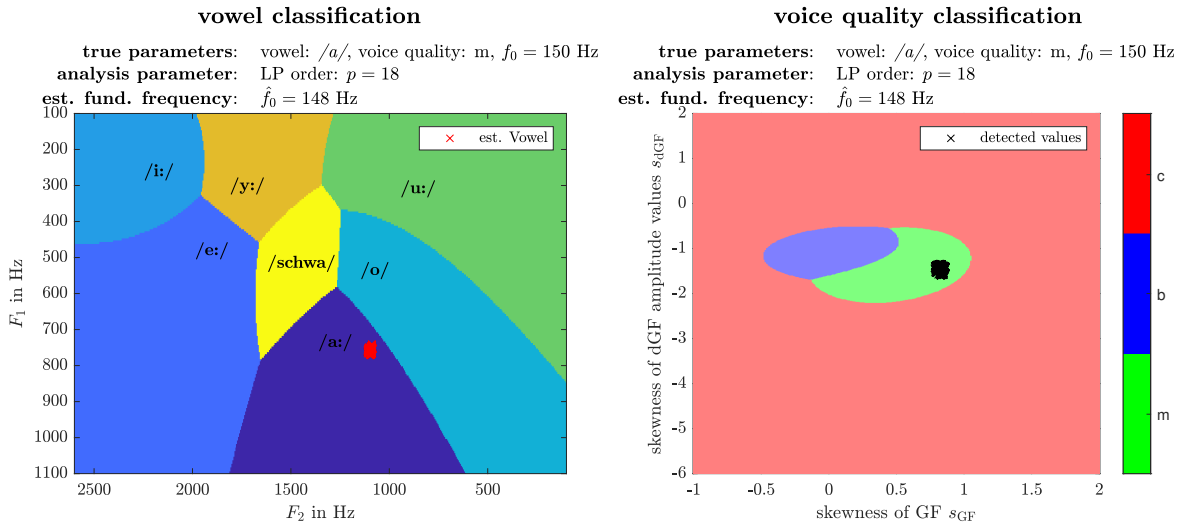
4.3 Process Summary and GUI Screenshots

In order to conclude the LP analysis algorithm implementation in C++/JUCE, a short summary is given. Figure 4.4 shows an overview of the signal flow and the control structure inside `LPThread::run()`'s core implementation. Comparing Figure 4.4 with Figure 1.2 it is visible that only one LP analysis algorithm is implemented, namely the *autocorrelation method with cepstral refinement*. Figure 4.4 is organized in such a way, that the blocks indicated by the dashed line (these are *Pre-Processing*, *Linear Prediction Analysis* and *Post-Processing/Classification*) match the eponymous blocks of Figure 1.2 with respect to their functionality.

In Figure 4.2, the estimation results for vowel and voice quality classification are shown using the `Matlab` implementation, discussed in subsection 3.2.2. Thereby, a $15\,\text{s}$ long sung vocal signal with vowel */a/*, modal voice quality and a fundamental frequency of $f_0 = 150\,\text{Hz}$ was synthesized and analyzed. The same sung vocal signal was processed through the VST implementation, whose results are shown in Figure 4.3. However, while Figure 4.2 shows the estimation results for all signal blocks, Figure 4.3 only shows the results of 15 blocks. This is due to the real-time processing of the implementation in C++/JUCE.

It is visible from Figure 4.2 in conjunction with Figure 4.3, that the results from the `Matlab`-implementation coincide with VST-plugin's results for the given set of true parameters used in the synthesis algorithm.

Further discussion on both implementations and their limitations can be found in chapter 5.



(a) vowel classification in `Matlab` (b) voice quality classification in `Matlab`

Figure 4.2 *Result plots for vowel and voice quality classification using* `Matlab`

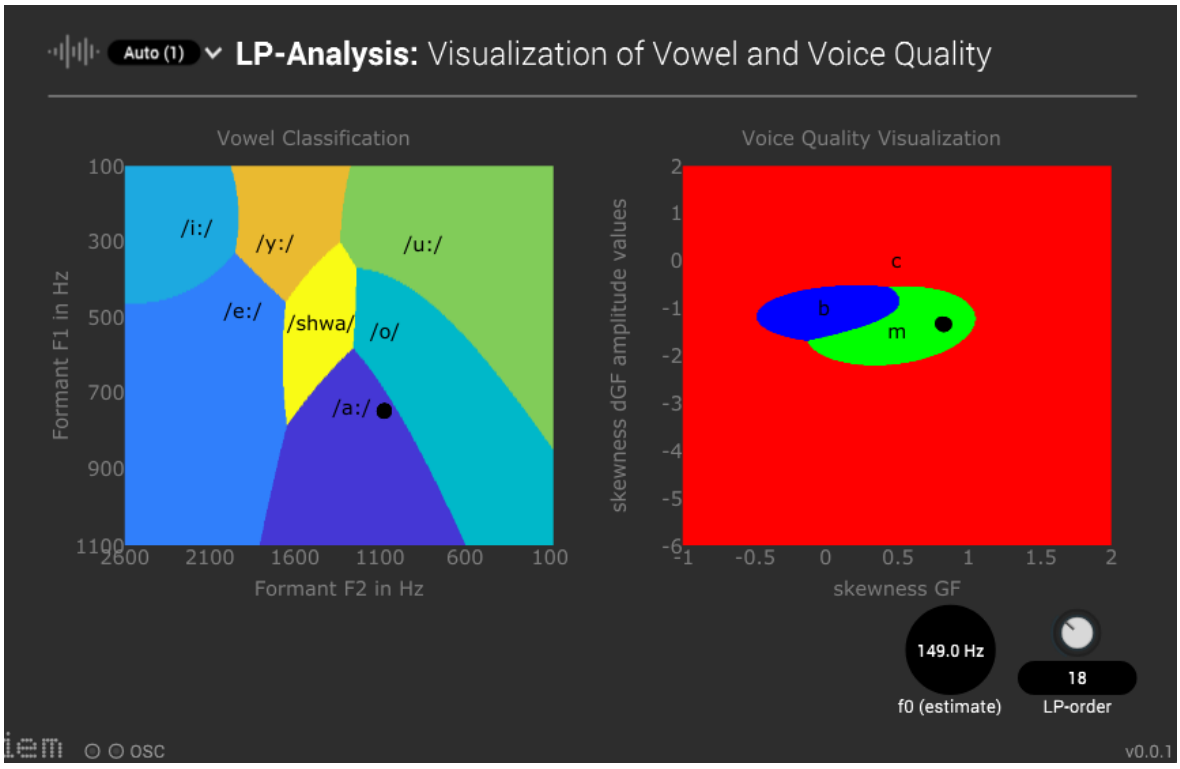


Figure 4.3 *Screenshot of proposed VST-plugin*

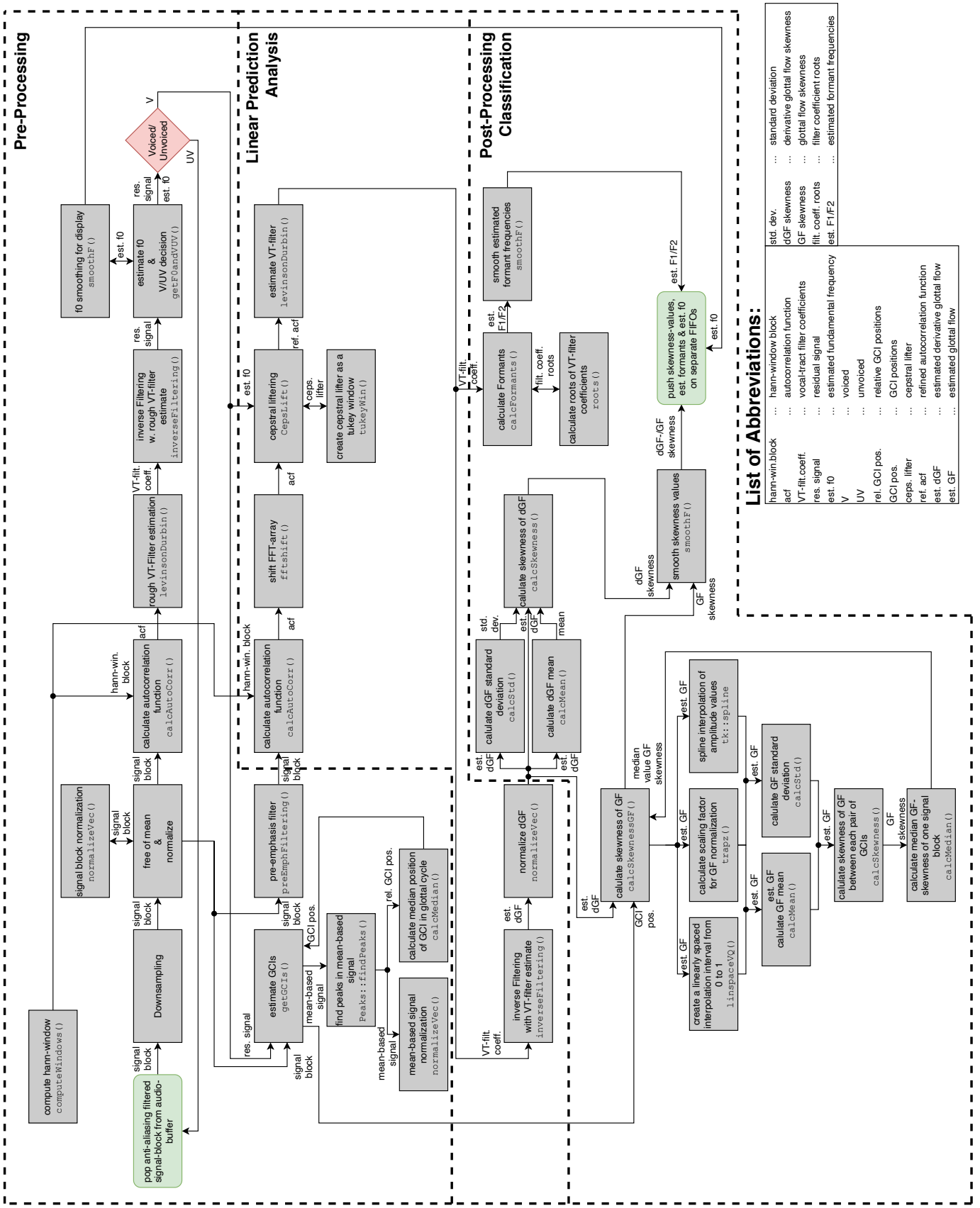


Figure 4.4 *Signal flow overview on the plug-in implementation's core: LPThread::run()*

5 Conclusion

On a general level, this project consisted of three parts:

(i) a synthesis algorithm as described in chapter 2,

(ii) a comparison of four linear prediction-based analysis algorithms as described in chapter 3 and

(iii) a proof-of-concept implementation as a VST-plugin in C++ using the JUCE framework as described in chapter 4.

The main findings of each project part are described in the following.

The *synthesis algorithm* described in chapter 2 built on the Liljencrants-Fant model to create a glottal signal (dGF) with a defined voice quality and singing vibrato. The dGF signals have been filtered with an all-pole filter modelling the human vocal tract, which results in a synthesized sung vocal signal with defined parameters for *fundamental frequency*, *voice quality* and *vowel*.

In chapter 3 we described, how the four different *linear prediction analysis algorithms* work, and how they have been compared with regard to their ability to approximate an all-pole vocal tract filter. The output of the linear prediction algorithms are estimated coefficients of the vocal tract filter, which are used to perform inverse filtering in order to compute the dGF signal. Based on the estimated filter coefficients, the formant frequencies are calculated, which are the definitive features for the vowel indictation. The estimated dGF-signal is used to evaluate two skewness-related measures for which a voice quality distinction was shown to be possible. A Monte-Carlo simulation has been used to create a dataset which accounts for the variability in the synthesizer parameters. For the formant frequency estimation, an error measure was evaluated for the *within-signal* formant estimation error and the *between-signal* estimation error, where the latter was evaluated for the whole Monte-Carlo dataset. Additionally, the dataset was used to evaluate the voice quality features. Based on the evaluated voice quality features and the ground truth, a support vector machine was trained to differantiate between the three voice qualities. Taking all results into account, in subsection 3.4.3 it was concluded, that the *autocorrelation method with cepstral refinement* performs best and is able to generate meaningful results for a frequency range of $f_0 \in [70, 320]$ Hz.

Finally, in chapter 4 an *implementation in C++ using the JUCE-framework* has been described for the chosen linear prediction algorithm. The aim of the implementation in C++/JUCE is to create a VST-plugin for digital audio workstations. Building onto an existing buffer structure from a recent project by Holzmüller *et al.* [29], the implementation was in general straightforward. Nevertheless, the differences between the `Matlab` prototype and the C++/JUCE-implementation are detailedly highlighted in chapter 4. The results calculated with the VST-plugin match the results of the Matlab prototype.

5.1 Limitations of the Proposed Synthesis and Analysis Algorithms

During the course of this project different limitations arose in different parts of the covered processing steps. The following paragraphs deal with the limitations that occurred during the synthesis, preprocessing, analysis and classification stage.

Synthesis of Sung Vocal Signals. Beginning at the first covered chapter, the sung vocal signal synthesis as proposed in page 5, a limitation is given through the simplified assumption that sung vocal signals are essentially speech signals with a higher fundamental frequency and vibrato. More complex processes occuring whilst singing, such as sophisticated laryngeal mechanisms or formant tuning as discussed in [2, p.39-41], are *not* covered by the implemented synthesis algorithm. Thus, signals whose fundamental frequency exceed the first formant frequency are not realistic and therefore lead to difficulties when it comes to estimating the VT filter of such a signal.

A small technical limitation worth noticing is, that the signal length of a synthesized signal has to be chosen larger than $0.4\,\mathrm{s}$, if signals with a fundamental frequency below $700\,\mathrm{Hz}$ are synthesized with vibrato. Nevertheless, for the execution of the Monte-Carlo simulation mentioned in section 3.4, the signal length was chosen to be $0.5\,\mathrm{s}$, thus, for the presented project this limitation was of no practical relevance.

Pre-Processing. The two main pre-processing steps which are limited due to certain parameters, are the *fundamental frequency estimation* and the *GOI/GCI-detection* discussed in subsection 3.1.2 and subsection 3.1.3, respectively. The main drawback discussed in the context of the f_0-estimation is the fixation of a comparatively large block length with respect to a real-time implementation. In subsection 3.1.2, it was mentioned, that the block length of $80\,\mathrm{ms}$ is necessary to ensure the occurance of at least five glottal cycles in one signal block for the lowest frequency of interest. The original routine proposed by Drugman and Alwan in [14] even presented a block-length of $100\,\mathrm{ms}$ as "[...] a good compromise for being efficient in any environment." [14, p. 1975]. The original routine from [14] was meant to be applied offline on whole signals (in contrast to signal blocks). For the purpose of this project, it was modified in such a way, that the calculations are executed block-wise, without any further sub-blocking. Therewith, it was possible to reduce the block length from $100\,\mathrm{ms}$ to the proposed $80\,\mathrm{ms}$, which is still relatively large considering a real-time application.

Analyzing the statistical results concerning the GI detection presented in Figure 3.7, 3.8 and Figure 3.9, it becomes clearly visible that the GOI detection as proposed in [15] and executed on the synthesized sung vocal signals is less precise than the GCI detection. As already mentioned in [15], the reason for this circumstance can be found in the weaker non-linearity occuring at the GOIs, or as formulated by Drugman and Dutoit: "[...] while the impulse at the GCI significantly emerges from its neighborhood, the behaviour at the GOI is more regular since the excitation presents a discontinuity more spread out and with a weaker strength." [15, p. 2] The weaker non-linearity leads to a less distinctive peak in the residual signal, which does not always exceed the other peaks in its presence interval. Another assumption that can be drawn from evaluating the first percentage measure shown in Figure 3.7 (b) is, that for signals with a fundamental frequency larger than $320\,\mathrm{Hz}$–$370\,\mathrm{Hz}$, the anticipated GOI presence intervals derived from the mean based signal are not valid. On the other hand, the GCI detection is more robust, as visible in the performance analysis of Figure 3.7 (a). Only for vowels with a low first formant frequency F_1, namely */i/* and */u/*, the GCI presence intervals are wrongly positioned, which could be traced back to a flawed rough VT filter estimation, which influences the computation of the residual signal, and therefore has indirect effects on the refinement of the GCI presence intervals proposed by Drugman and Dutoit in [15].

Analysis of Sung Vocal Signals. In general, when looking at the analysis algorithm's performance evaluation in section 3.4, it can be asserted that the misestimations concerning the VT filter are kept in a reasonable scope for fundamental frequencies $f_0 \leq 320\,\mathrm{Hz}$. For signal realizations containing vowels with a low first formant, e.g. */i/* and */u/*, the results shown in Figure A.14 and A.16, indicate misestimations for signals with fundamental frequencies that reach the frequency range of the first formant, i.e. $f_0 \approx F_1$. If the fundamental frequency f_0 reaches the vicinity of the first formant

frequency F_1, the spectral envelope is falsified by the harmonic structure brought in by the excitation signal. Thus, the VT filter estimation using linear prediction leads to misestimations, as fundamental frequency peaks are mistaken for formant sturctures. Especially the vowels */i/* and */u/* are affectd by this phenomenon, because of their relatively low F_1 being $F_1 = 360\,\text{Hz}$ for */i/* and $F_1 = 409\,\text{Hz}$ for */u/*. Taking a closer look at the estimated VT filter's frequency responses for sung vocal signal containing the vowel */i/* at $f_0 = 150\,\text{Hz}$ with modal voice quality, displayed in Figure A.6, the misinterpretation of the second/third harmonic as the signal's first formant F_1 is clearly visible. The harmonic structure produced by the excitation signal interferes with the spectral envelope created by the VT filter. It would not be fair to solely blame this circumstance on the analysis stage, as the analysis algorithms were only evaluated using signals synthesized with the synthesis algorithm proposed in chapter 2. As already mentioned, the modelling of sung vocal signals with this synthesis algorithm is not a realistic replication of sung vocal signals by real signers, especially for fundamental frequency reaching the range of the formant frequencies.

Taking a closer look at the *autocorrelation method with cepstral refinement*, the most distinctive limitation arises with the estimated filter gain. Due to the cepstral liftering, signal energy is removed from the autocorrelation function which – especially for signals with higher fundamental frequencies – leads to a misestimated filter gain as shown in Figure 3.26 (b). As visible in the VT filter frequency response, it is not just a gain *offset* that occurs, but rather a *trend* that leads to a filter gain decrease towards lower frequencies, so a simple additive gain would probably not be sufficient. By further investigating this behaviour, with e.g. a frequency dependent gain error evaluation, one could try to compensate the decrease with an additional filter or maybe even a more sophisticated pre-emphasis filter stage would suffice.

Post-Processing and Classification of Vowel and Voice Quality. The last step of the proposed analysis algorithm only comes with one drawback which can not be blamed on the chosen classification method, namely the usage of *support vector machines*. As the chosen classification algorithm with respect to the voice quality, is only as good as the data that was used to train it, thus the derived class boundaries are only valid for the proposed synthesizer. The generalization to other signals or real sung vocal samples is not given, as there was no other labeled training data included.

Another limitation of the voice quality classification comes with the used skewness features. As discussed in subsection 3.4.3, the skewness features only provide a meaningful clustering for fundamental frequencies of $f_0 \in [70\,\text{Hz},\ 320\,\text{Hz}]$. The clustering might be improved by using additional features, e.g. mel-frequency cepstral coefficients (MFCCs) or the estimated fundamental frequency $\hat{f}_0$.

The data used for the indication of the sung vowel was taken from Sendlmeier and Seebode, whose data is based on human speech samples [63]. Therefore, it can be assumed, that the vowel indication generalizes well for real samples, at least for human speech in German language. But then again, besides the already mentioned complex VT processes occuring in singing [2], the formant frequencies for sung vocal signals might also differ from the formant frequencies of speech, which was analyzed by Fleischer *et al.* in [19].

5.2 Suggestions for Future Research

Based on the experience obtained over the course of this thesis and the limitations of the proposed synthesis and analysis algorithms discussed in section 5.1, some questions in context with the presented project remain open. In the following, a list of open questions and topics for each chapter of this project is presented. Note that this list is neither complete nor definitive. It should rather serve as

an inspiration for the reader to continue the work in this vibrant, ever-expanding research field.

Synthesis of Sung Vocal Signals.

- How can the singing voice synthesis be improved? What are additional distinct features of singing voice compared to speech?
- Where are the formant frequencies of different vowels for singing voice? Is it even reasonable to model singing voice with formant data derived from speech signals? An investigation on the vocal tract filter formants during singing, by studying real, non-generic sung vocal signals, could be beneficial to refine the synthesizer.
- Is the modelling of female vocal tracts using the all-pole model valid enough? Are there additional *gender-specific aspects* that need special consideration?
- Implementation of a *real-time* synthesis algorithm for sung vocal signals.
- *Virtualization of a singer* with selectable vowel and voice quality in a 3D-environment using recent research on the directivity of sung vocals [4, 7, 8].

Analysis of Sung Vocals Signals.

- To shed further light on the different LP-analysis algorithm's performance a filter-gain error evaluation would be beneficial. Especially for the autocorrelation method using cepstral refinement it was shown that the proposed gain estimation is not beneficial for higher fundamental frequencies. This effect is visible in the misestimated gain of Figure 3.26. A frequency dependent gain-filter error evaluation could provide information on how to tackle the misestimation of the filter gain for different lp analysis algorithms.
- Concerning the voice quality classification: What other features are informative in order to to provide a distinction between voice qualities? For example can spectrum based features provide an improvement for voice quality classification?
- Can the proposed algorithm be used for the qualitative analysis of *real singers* producing different voice qualities? In which aspects does a real singer differ from the proposed sung vocal signal synthesizer?

Implementation in C++ using the JUCE-Framework.

- Optimize the implementation with respect to resource efficiency and performance.

Appendix A Additional Plots

On the following pages, additional plots are included that demonstrate the effect of synthesis parameter variation, for example different vowels, voice qualities or fundamental frequencies, on preprocessing, analysis and classification.

A.1 SRH and Estimation of Fundamental Frequency

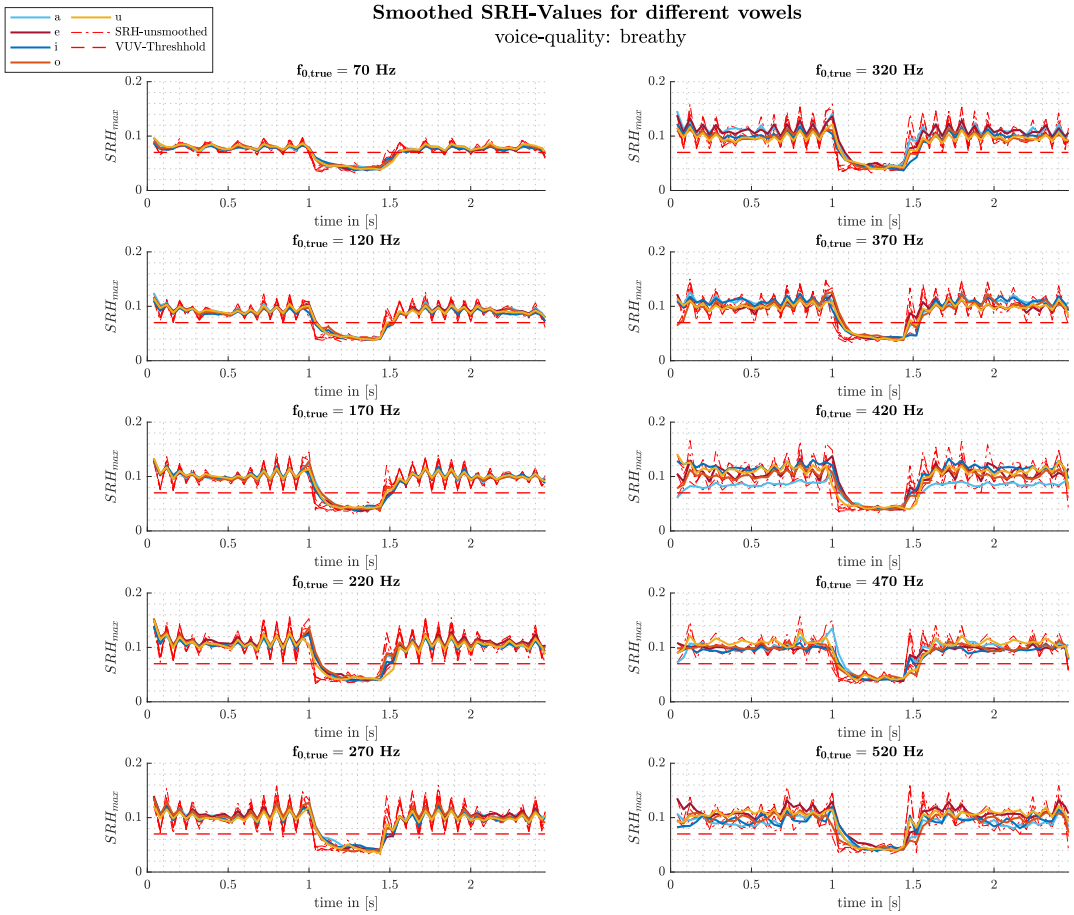


Figure A.1 *SRH_{max} smoothed for different vowels, fundamental frequencies and breathy voice*

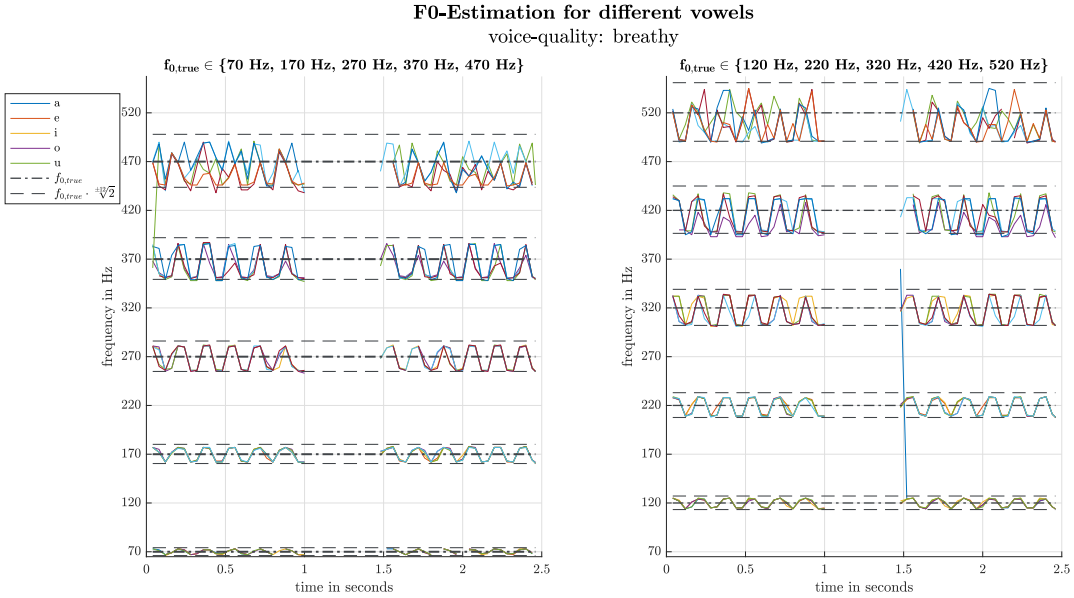


Figure A.2 *f_0-Estimation analysis over frequency for breathy voice*

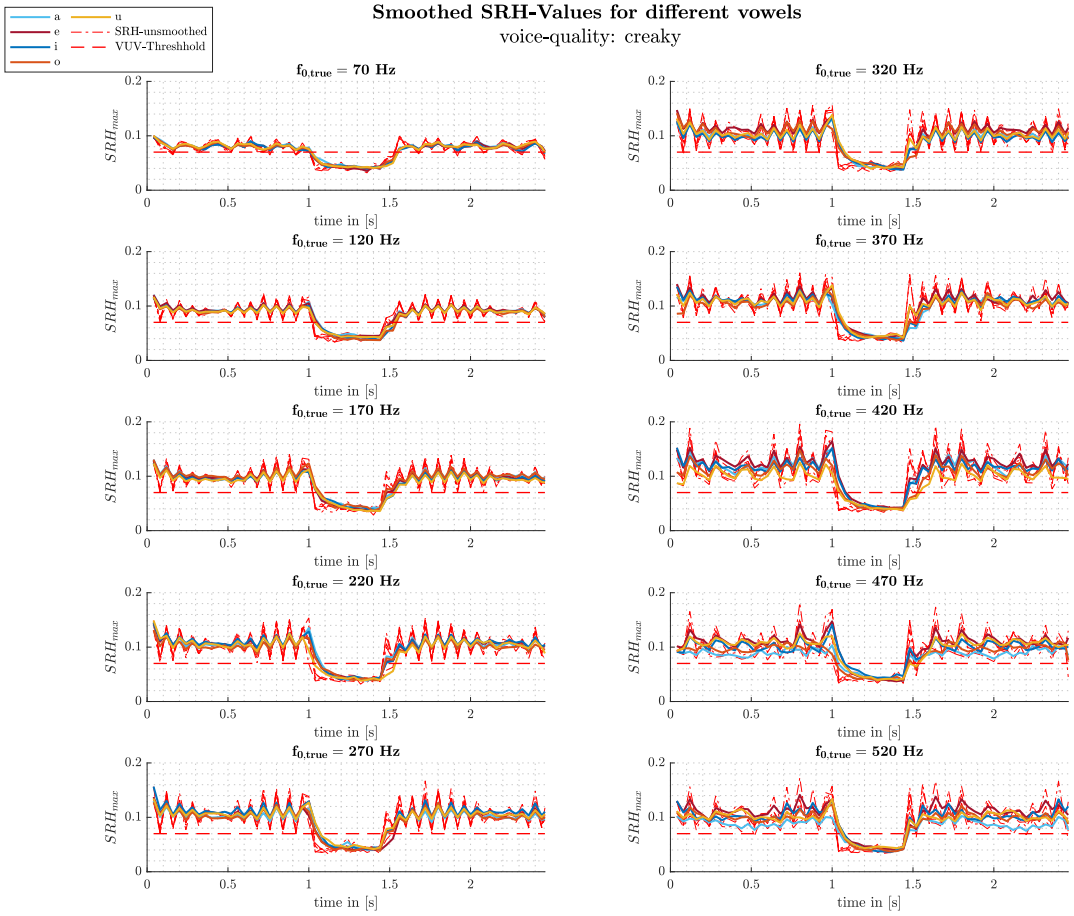


Figure A.3 *SRH_{max} smoothed for different vowels, fundamental frequencies and creaky voice*

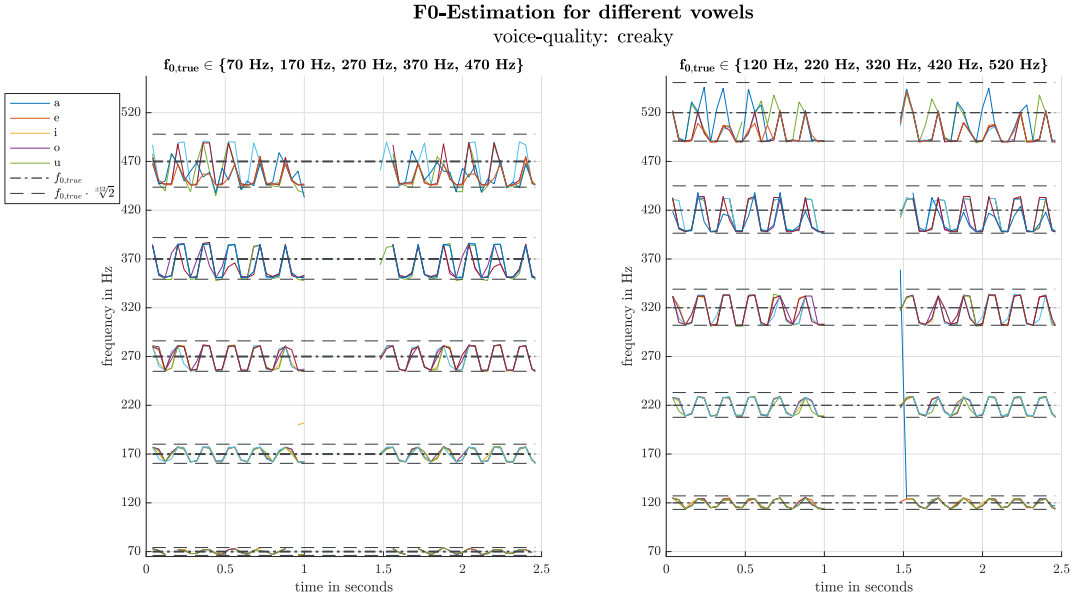


Figure A.4 *f_0-Estimation analysis over frequency for creaky voice*

A.2 Estimated Vocal Tract Filters

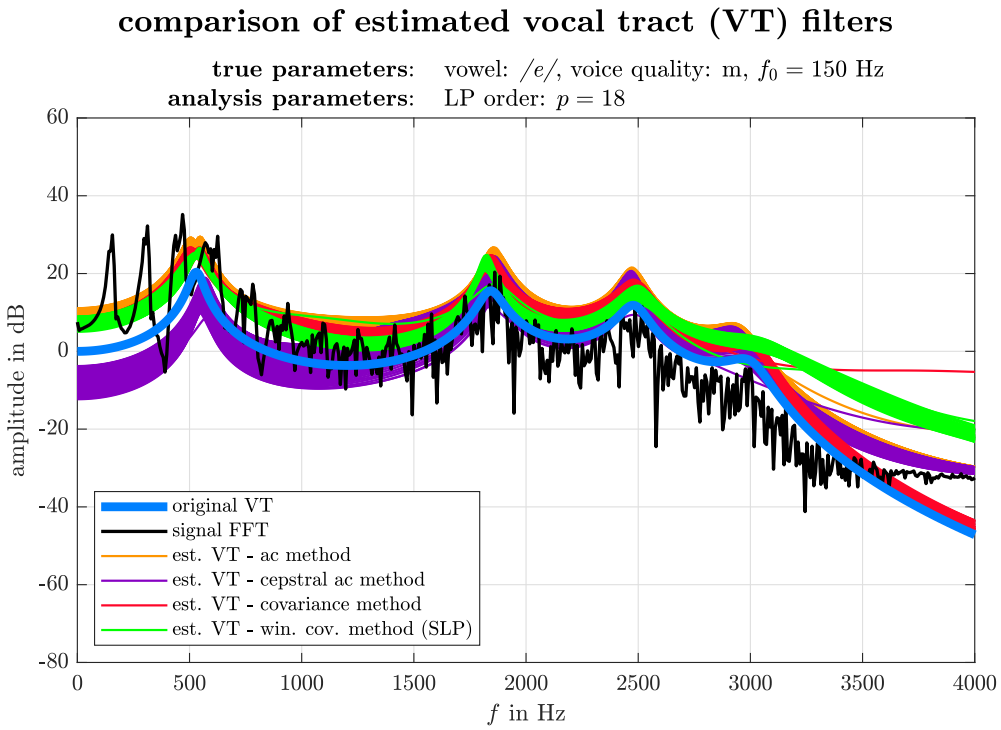


Figure A.5 *Comparison of estimated vocal tract filters for the vowel /e/ with voice quality modal and fundamental frequency* $f_0 = 150\,\mathrm{Hz}$

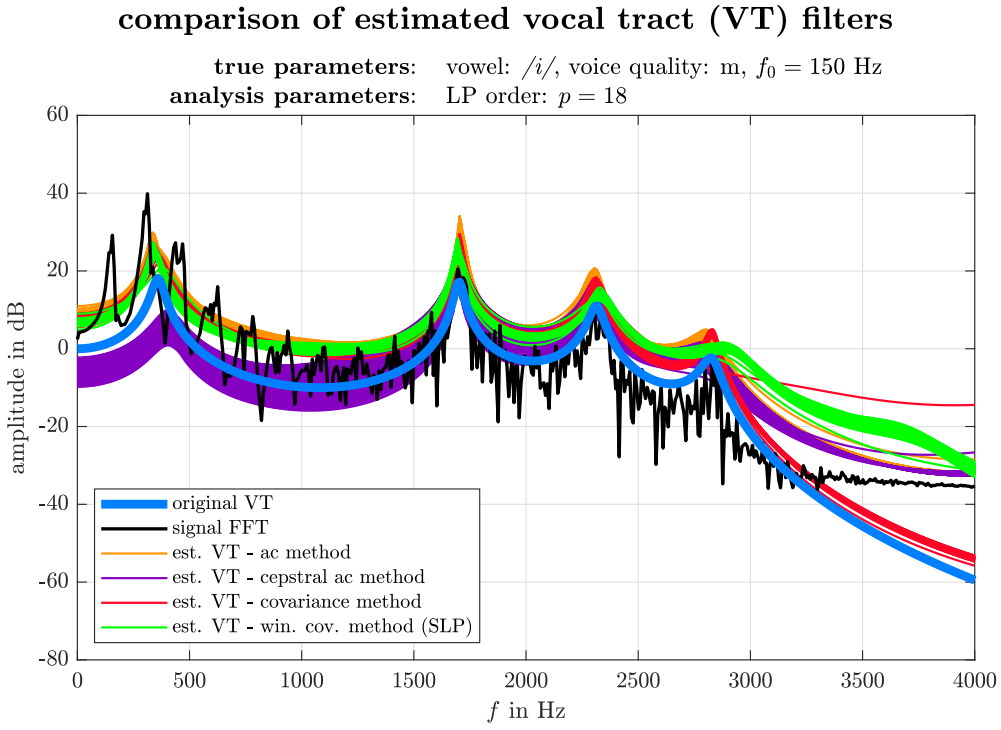


Figure A.6 *Comparison of estimated vocal tract filters for the vowel /i/ with voice quality modal and fundamental frequency* $f_0 = 150\,\mathrm{Hz}$

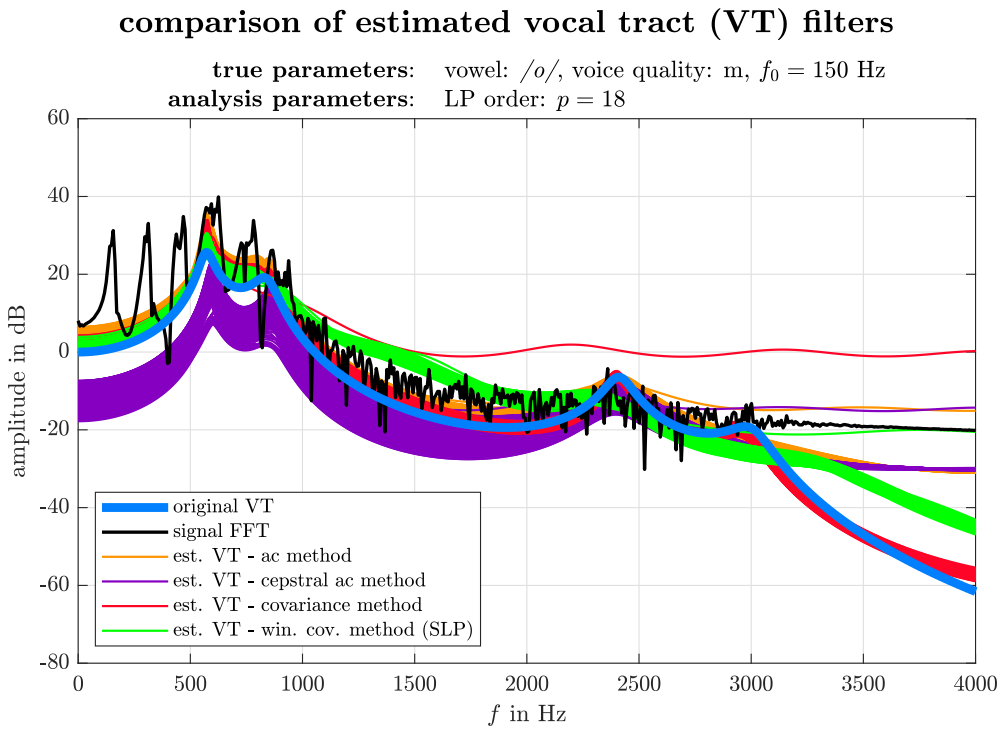


Figure A.7 *Comparison of estimated vocal tract filters for the vowel /o/ with voice quality modal and fundamental frequency* $f_0 = 150\,\mathrm{Hz}$

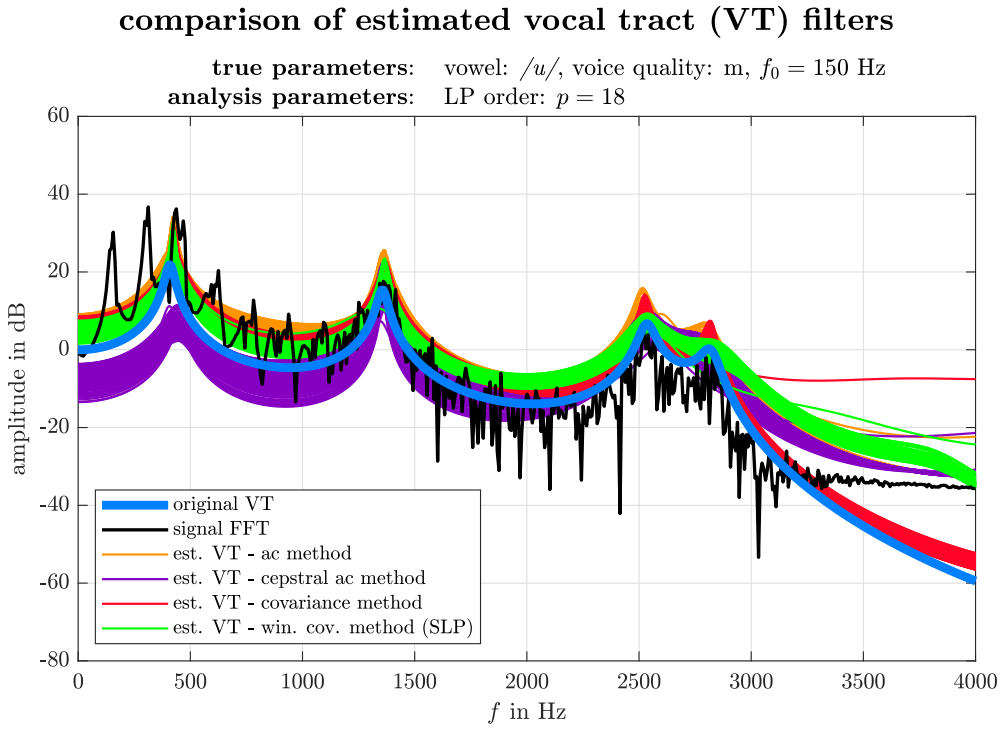


Figure A.8 *Comparison of estimated vocal tract filters for the vowel /u/ with voice quality modal and fundamental frequency* $f_0 = 150\,\mathrm{Hz}$

A.3 Formant Estimation Error Measure within Single Realizations

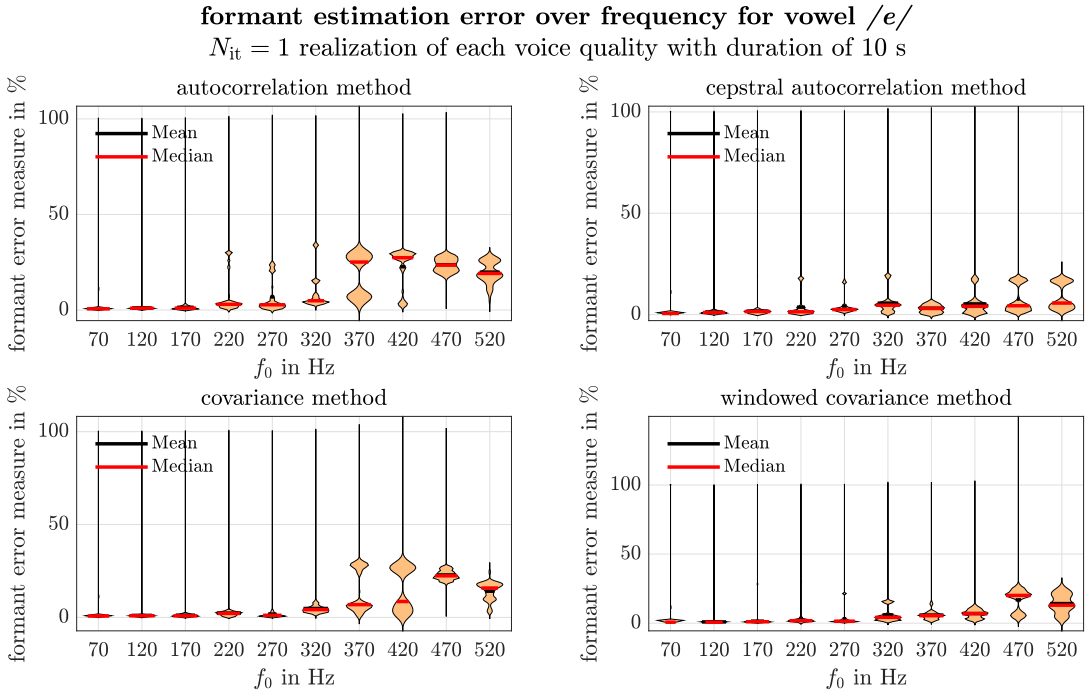


Figure A.9 *Error of formant estimation depending on the fundamental frequency f_0 for the four algorithms with vowel /e/ (single realization)*

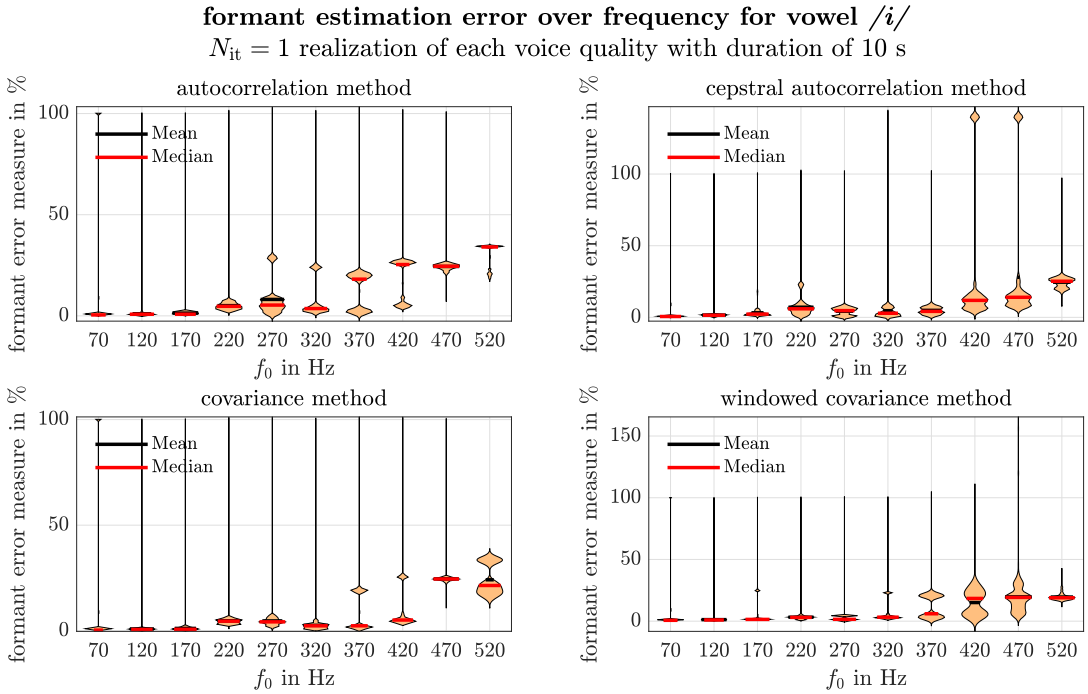


Figure A.10 *Error of formant estimation depending on the fundamental frequency f_0 for the four algorithms with vowel /i/ (single realization)*

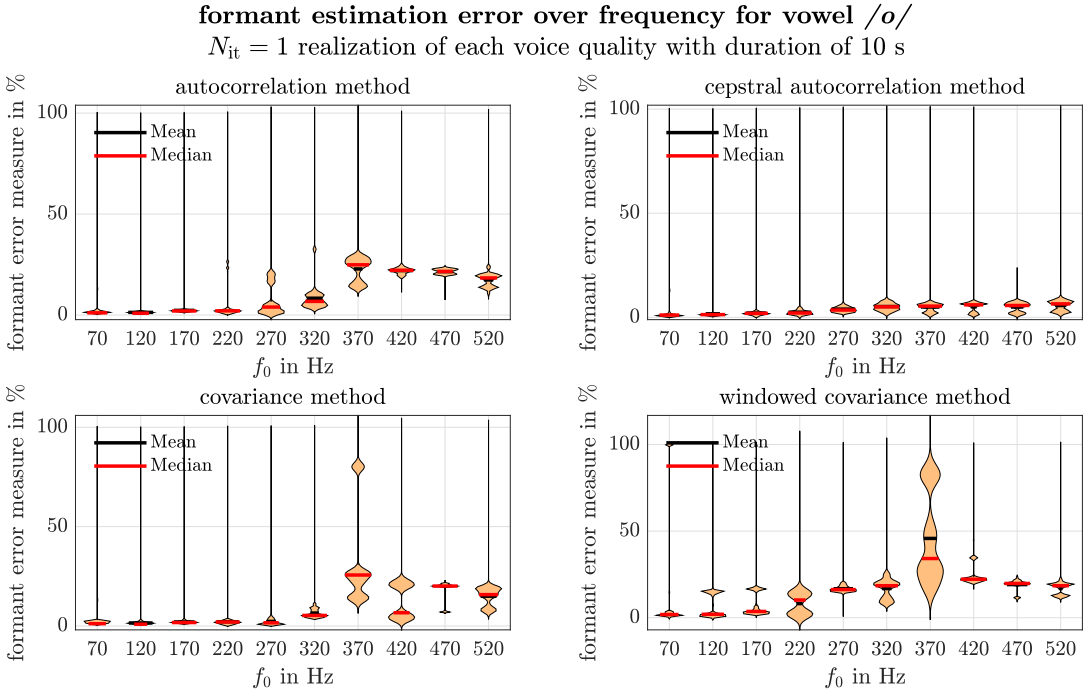


Figure A.11 *Error of formant estimation depending on the fundamental frequency f_0 for the four algorithms with vowel /o/ (single realization)*

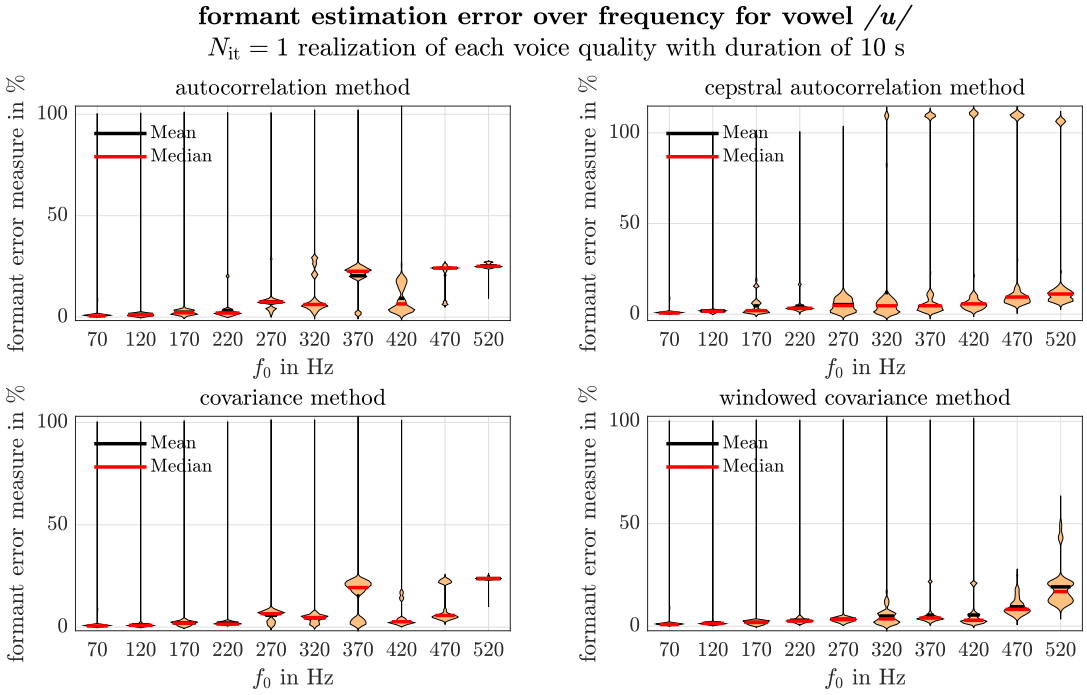


Figure A.12 *Error of formant estimation depending on the fundamental frequency f_0 for the four algorithms with vowel /u/ (single realization)*

A.4 Formant Estimation Error Measure between Multiple Realizations

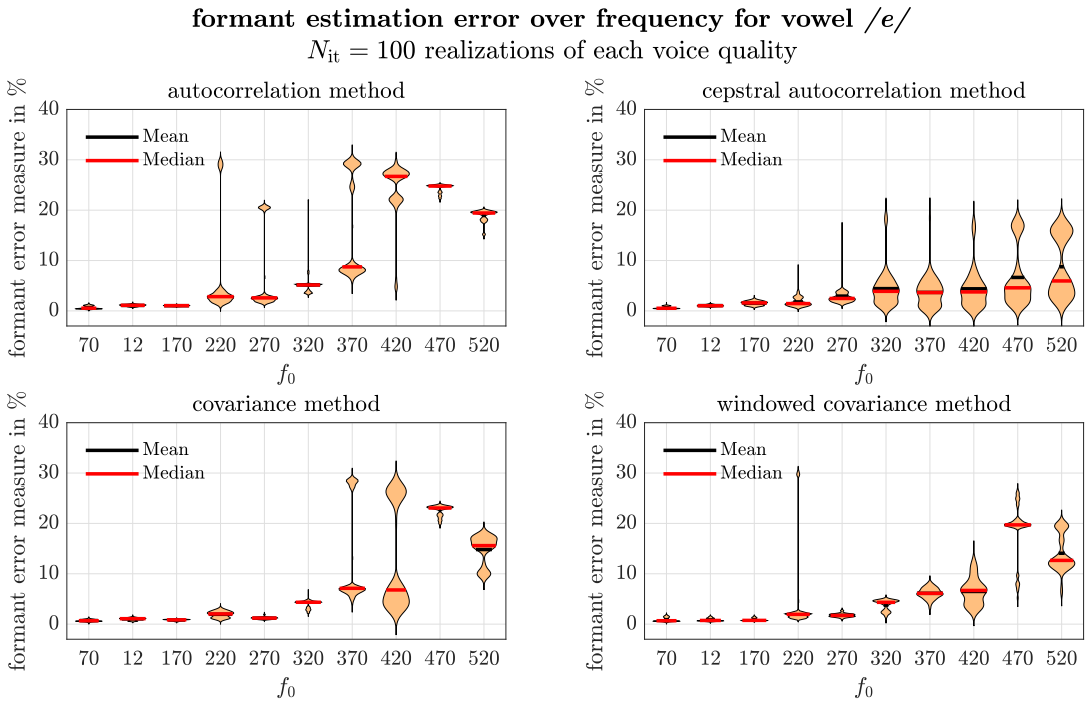


Figure A.13 *Error of formant estimation depending on the fundamental frequency f_0 for the four algorithms with vowel /e/ (multiple realizations)*

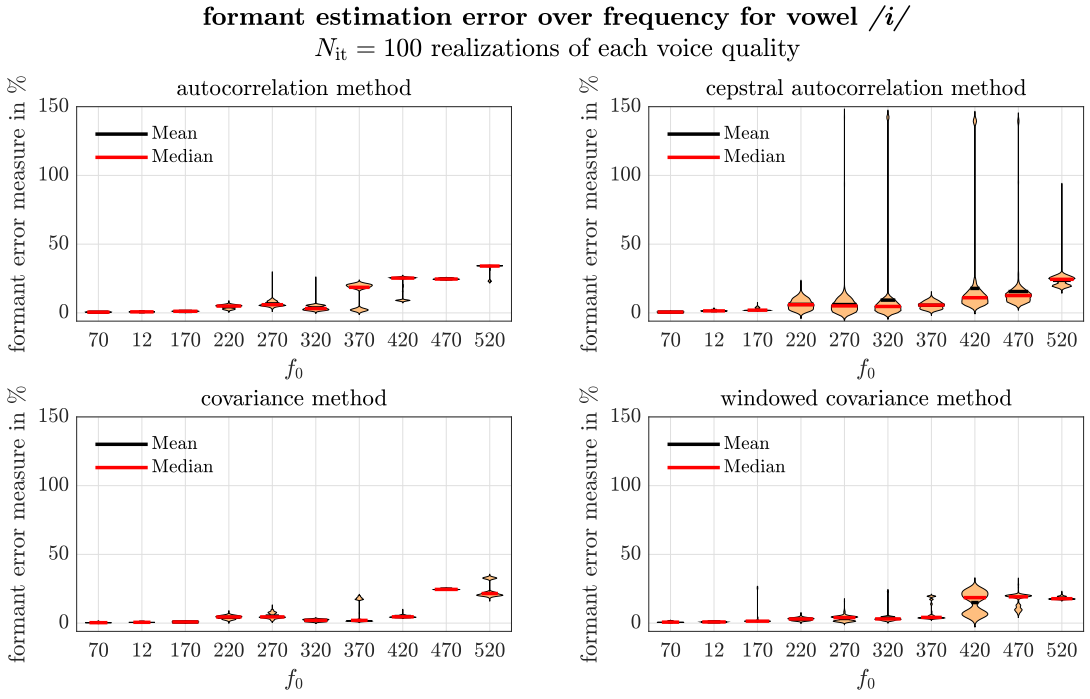


Figure A.14 *Error of formant estimation depending on the fundamental frequency f_0 for the four algorithms with vowel /i/ (multiple realizations)*

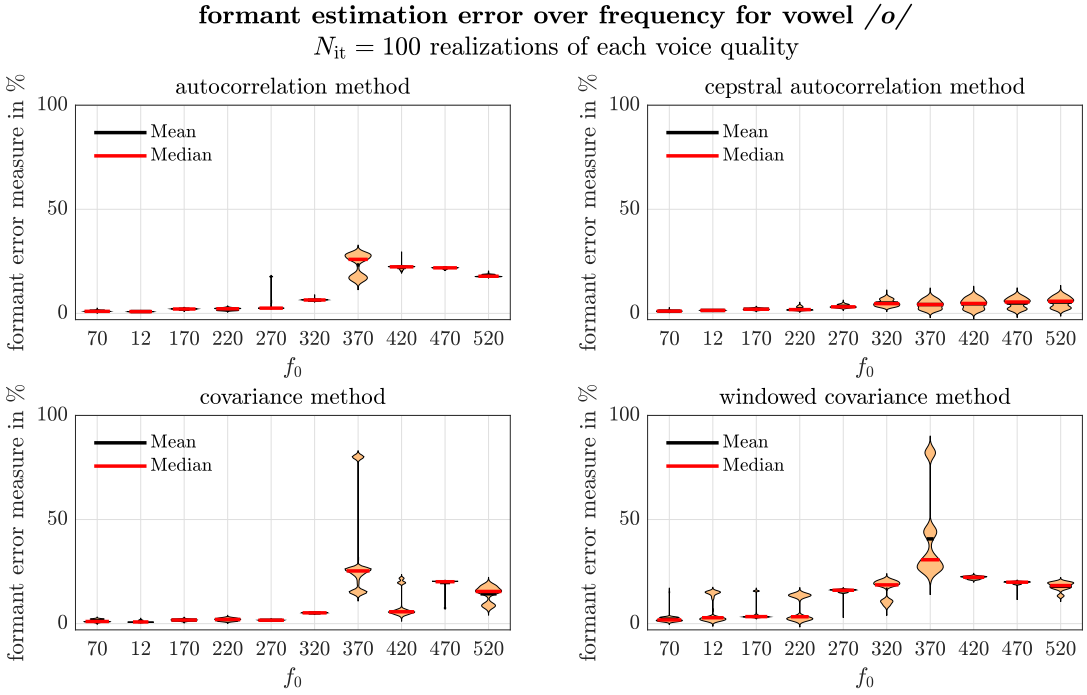


Figure A.15 *Error of formant estimation depending on the fundamental frequency f_0 for the four algorithms with vowel /o/ (multiple realizations)*

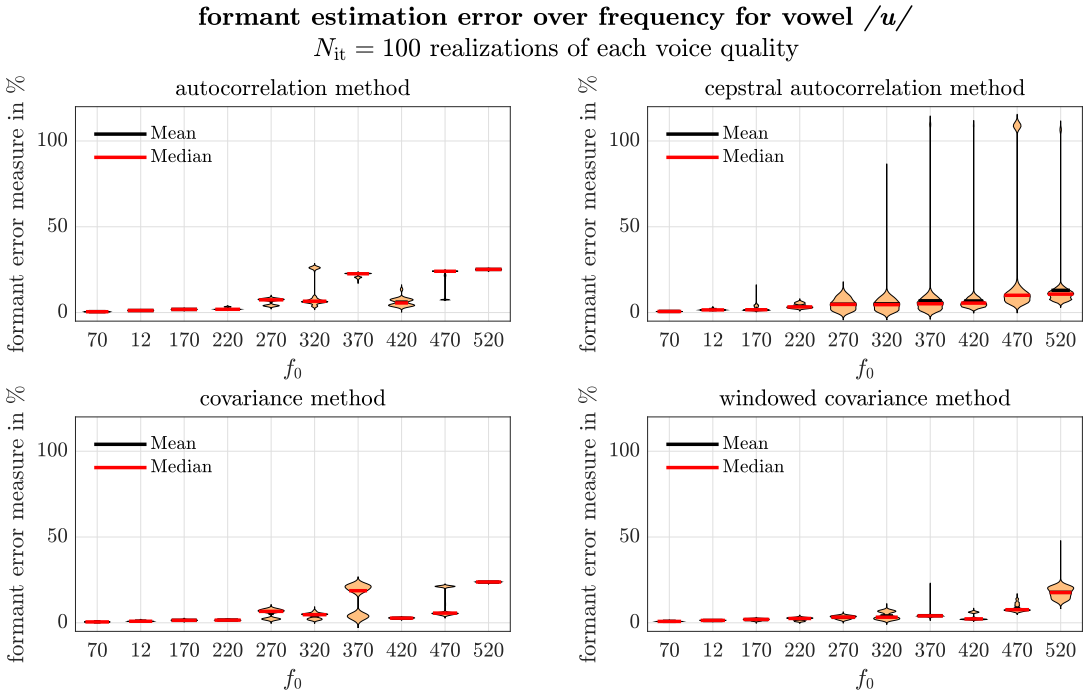


Figure A.16 *Error of formant estimation depending on the fundamental frequency f_0 for the four algorithms with vowel /u/ (multiple realizations)*

A.5 Clustering Analysis of Voice Quality Features

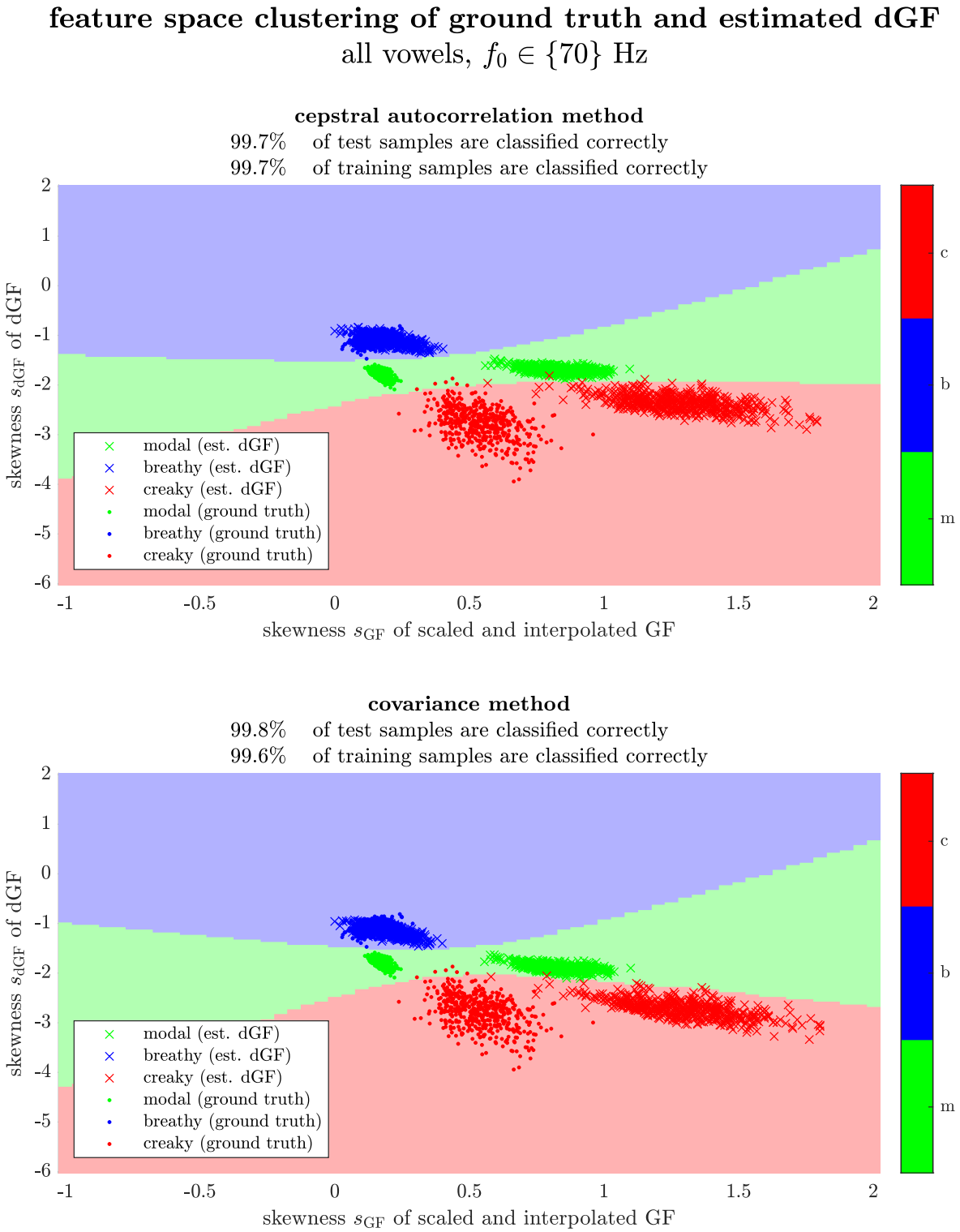


Figure A.17 *Clustering of voice quality features considering fundamental frequencies up to* $f_0 = 70\,\mathrm{Hz}$. *Comparison of cepstral autocorrelation method and covariance method.*

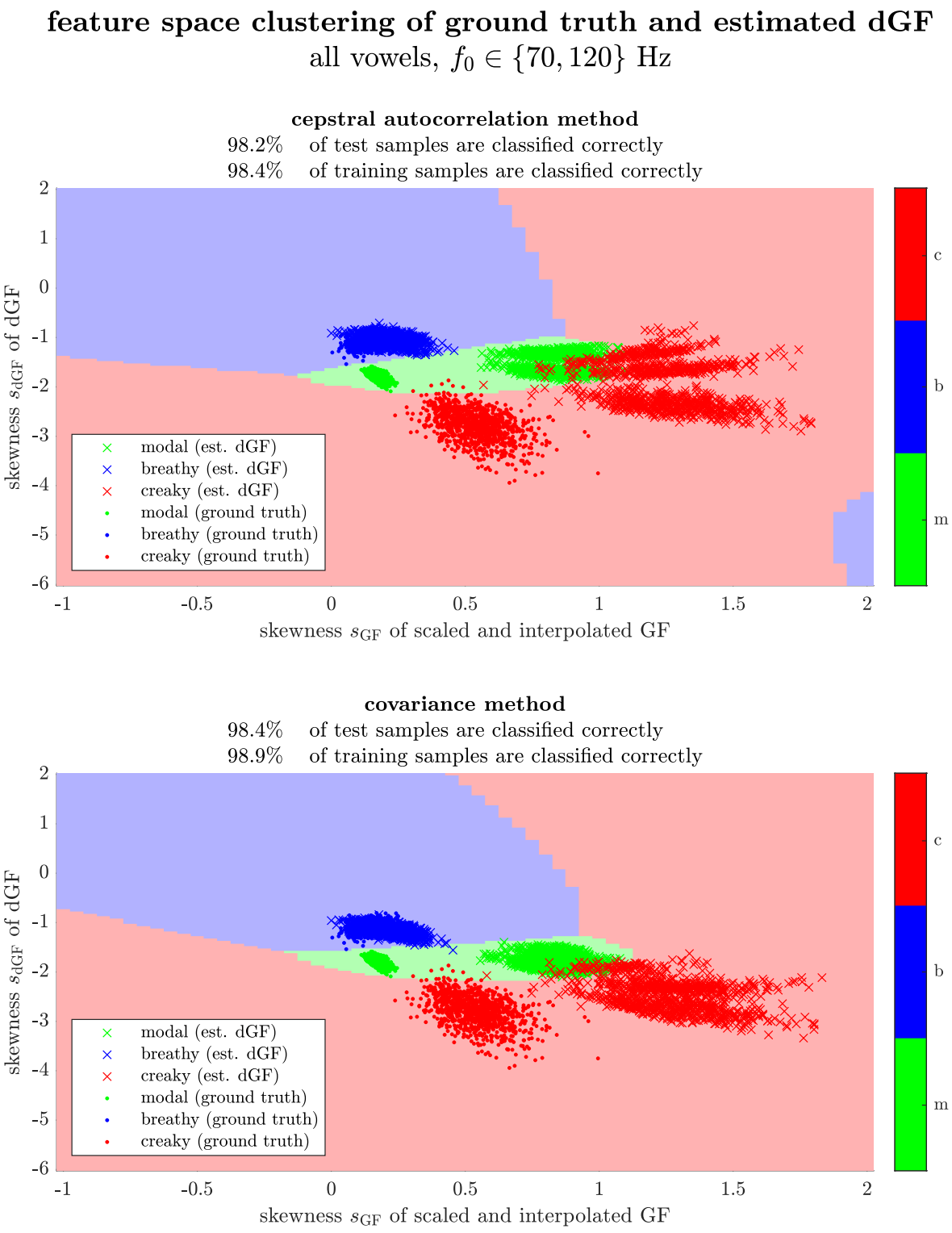


Figure A.18 *Clustering of voice quality features considering fundamental frequencies up to* $f_0 =$ 120 Hz. *Comparison of cepstral autocorrelation method and covariance method.*

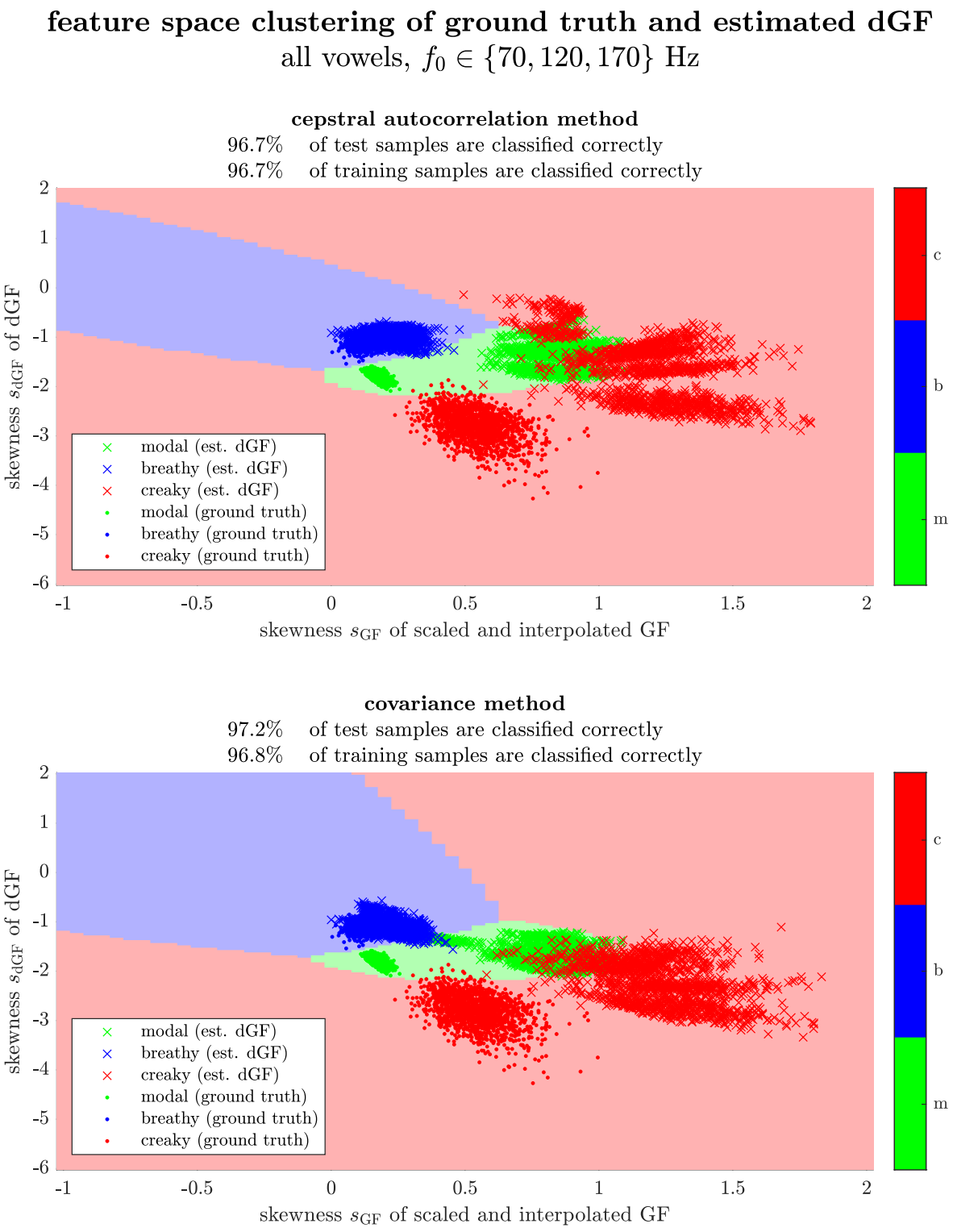


Figure A.19 *Clustering of voice quality features considering fundamental frequencies up to* $f_0 =$ 170 Hz. *Comparison of cepstral autocorrelation method and covariance method.*

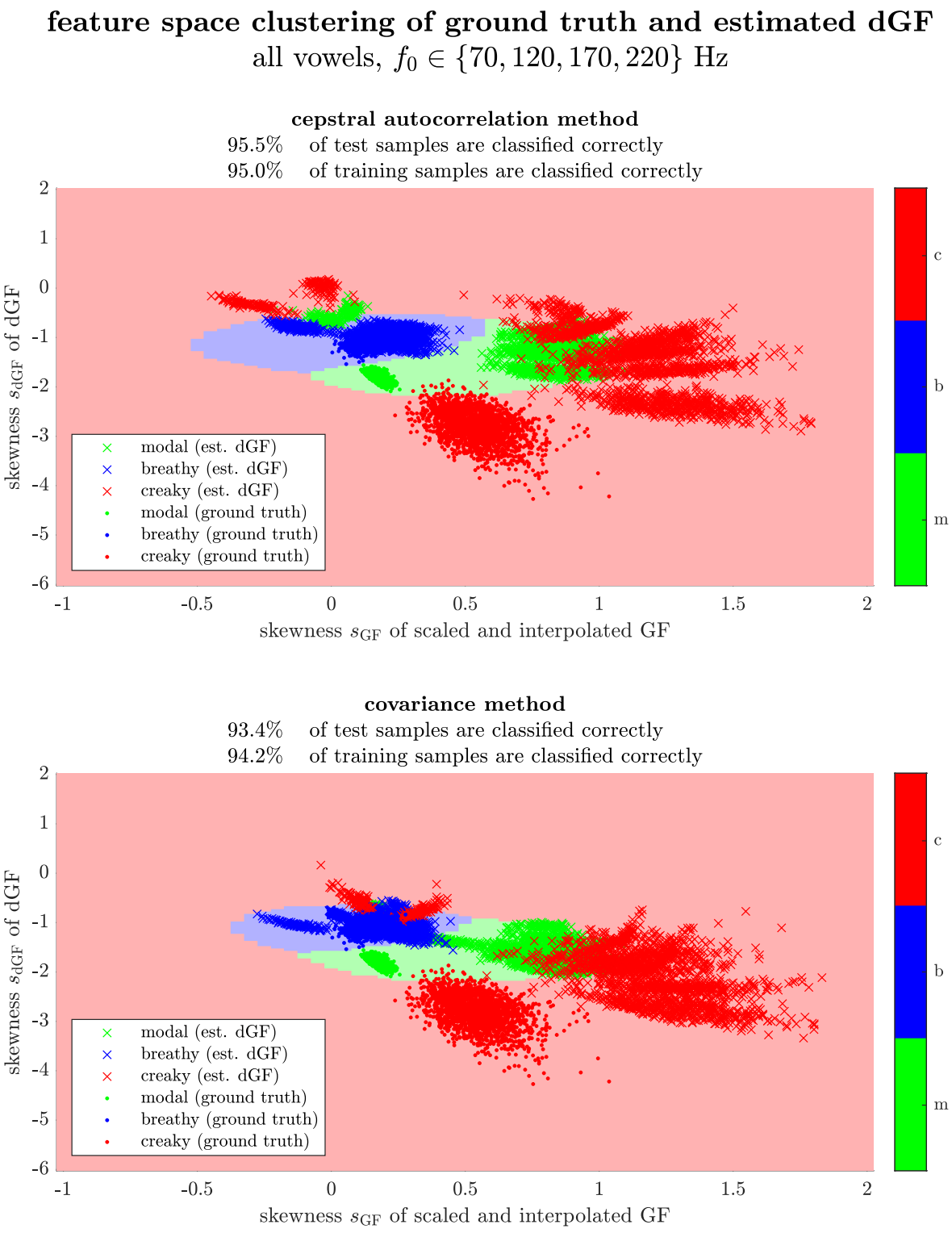


Figure A.20 *Clustering of voice quality features considering fundamental frequencies up to* $f_0 =$ 220 Hz. *Comparison of cepstral autocorrelation method and covariance method.*

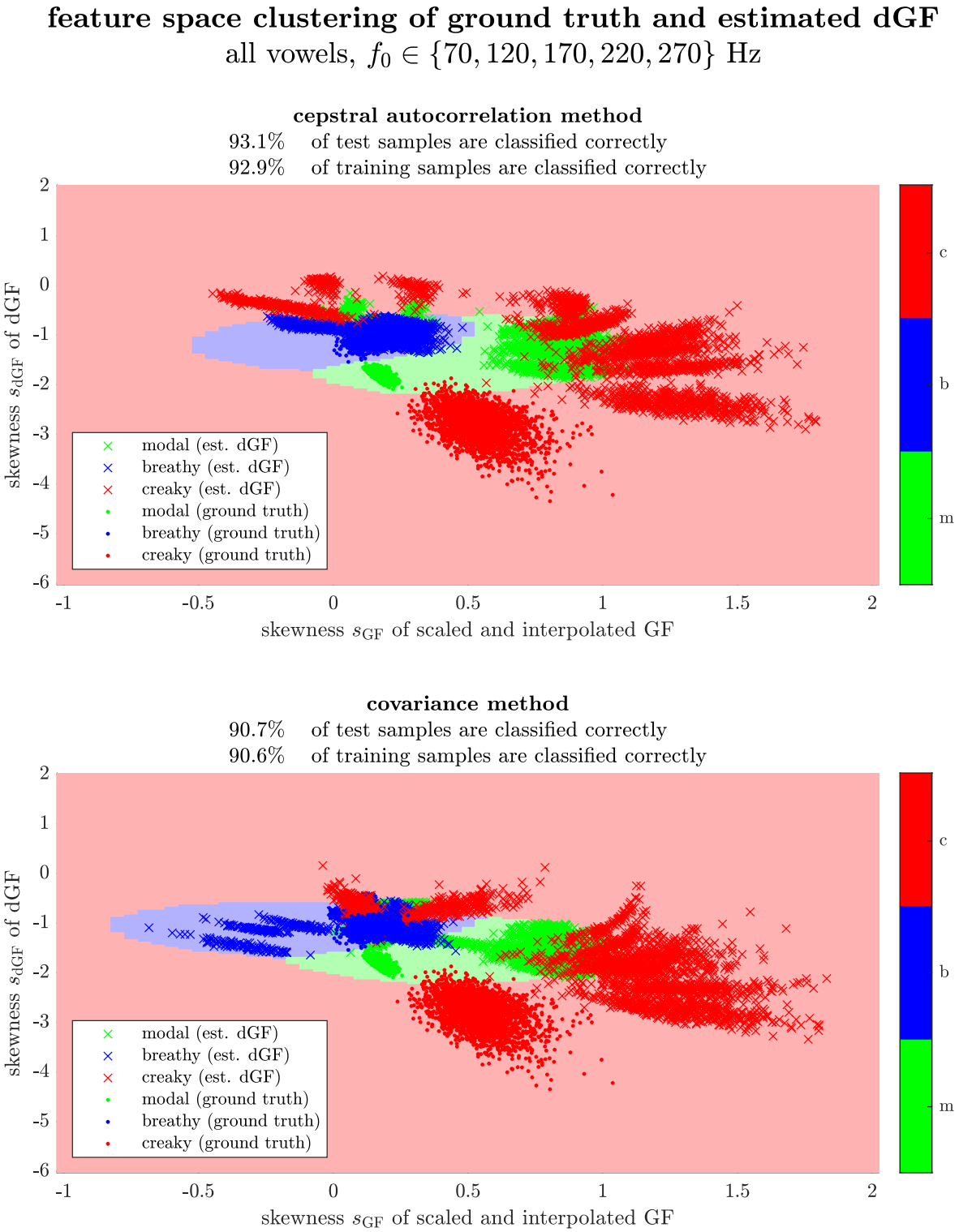


Figure A.21 *Clustering of voice quality features considering fundamental frequencies up to* $f_0 = 270\,\mathrm{Hz}$. *Comparison of cepstral autocorrelation method and covariance method.*

The clustering of voice quality features considering fundamental frequencies up to $f_0 = 270\,\mathrm{Hz}$ is displayed in Figure 3.30.

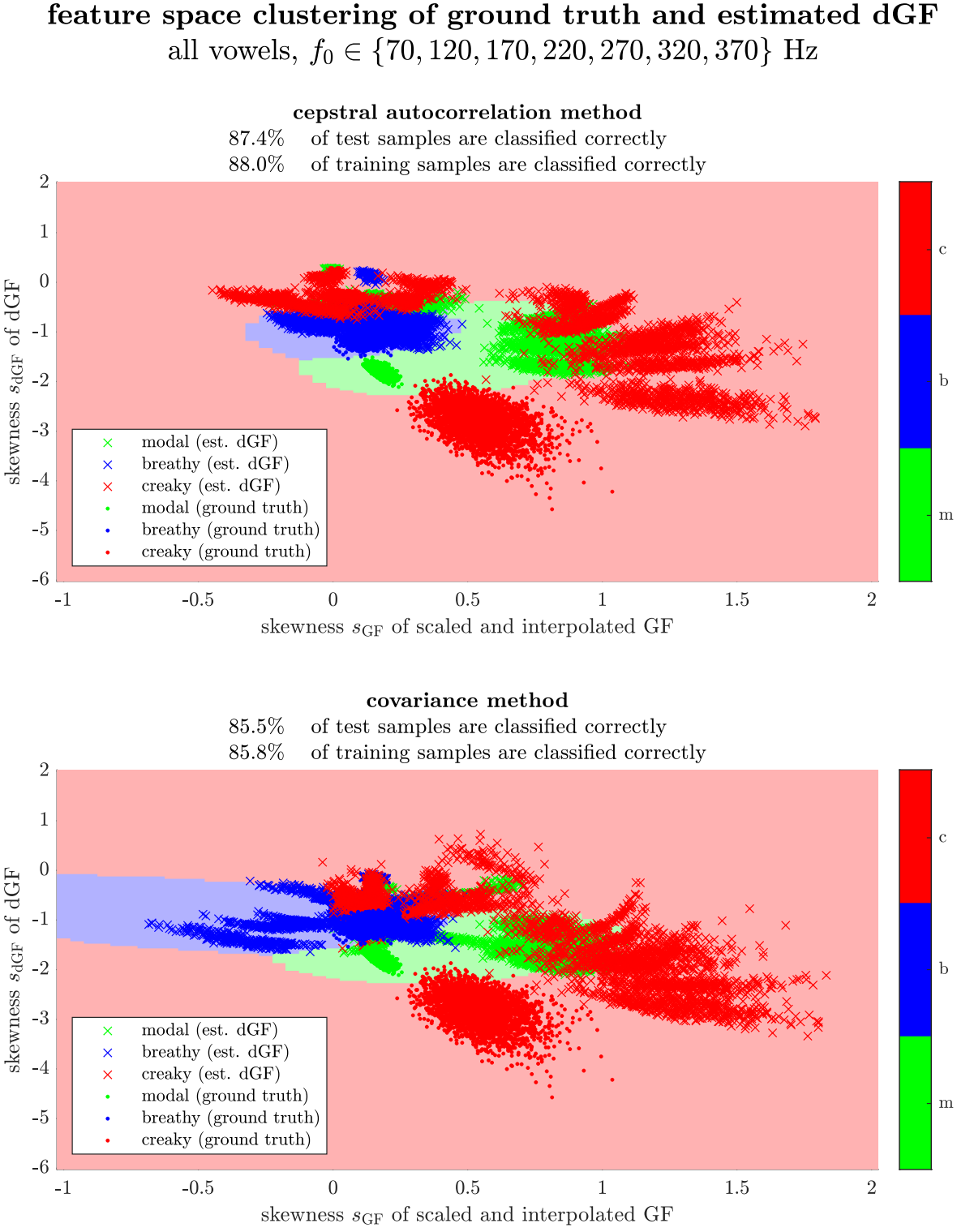


Figure A.22 *Clustering of voice quality features considering fundamental frequencies up to* $f_0 =$ 370 Hz. *Comparison of cepstral autocorrelation method and covariance method.*

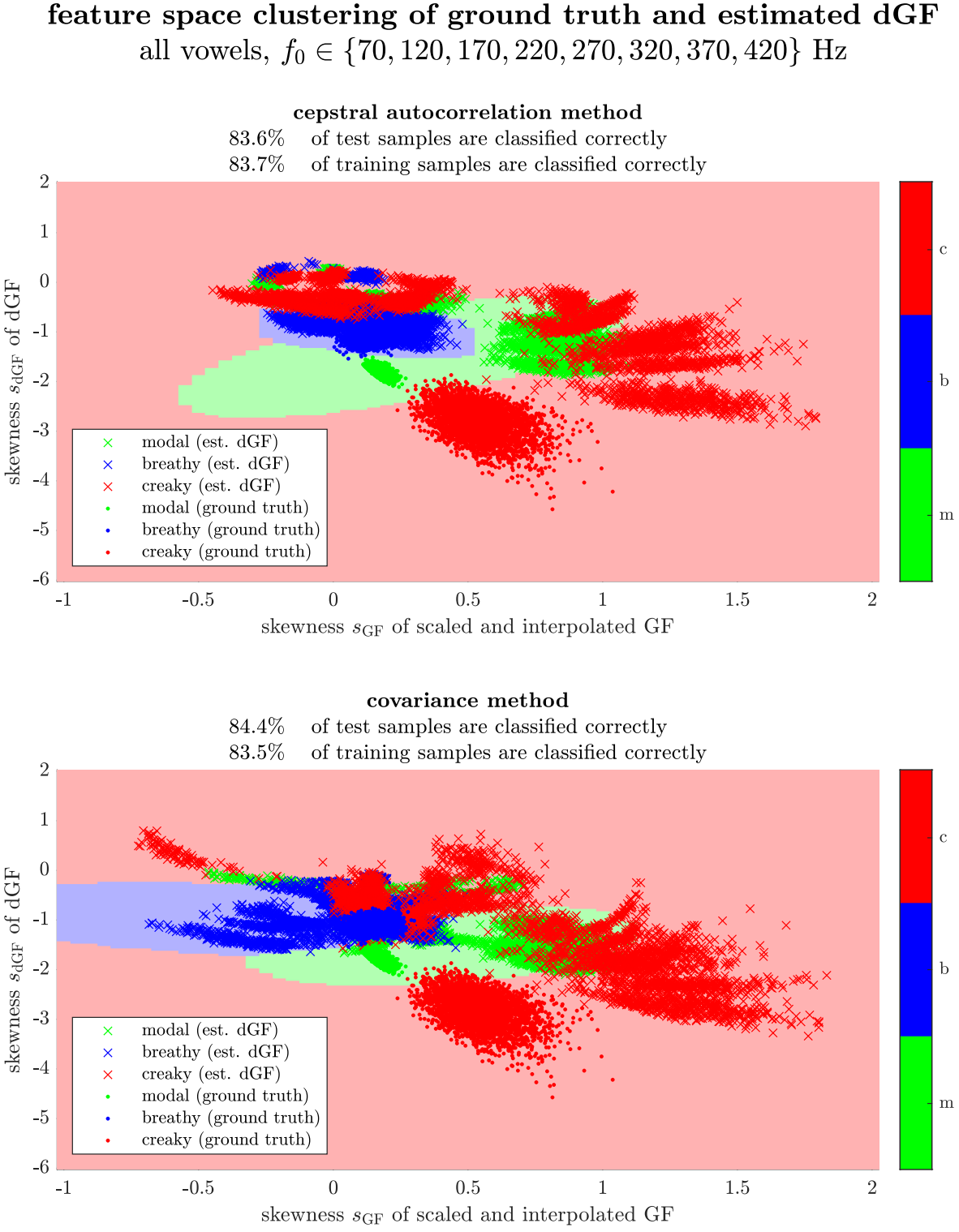


Figure A.23 *Clustering of voice quality features considering fundamental frequencies up to* $f_0 =$ 420 Hz. *Comparison of cepstral autocorrelation method and covariance method.*

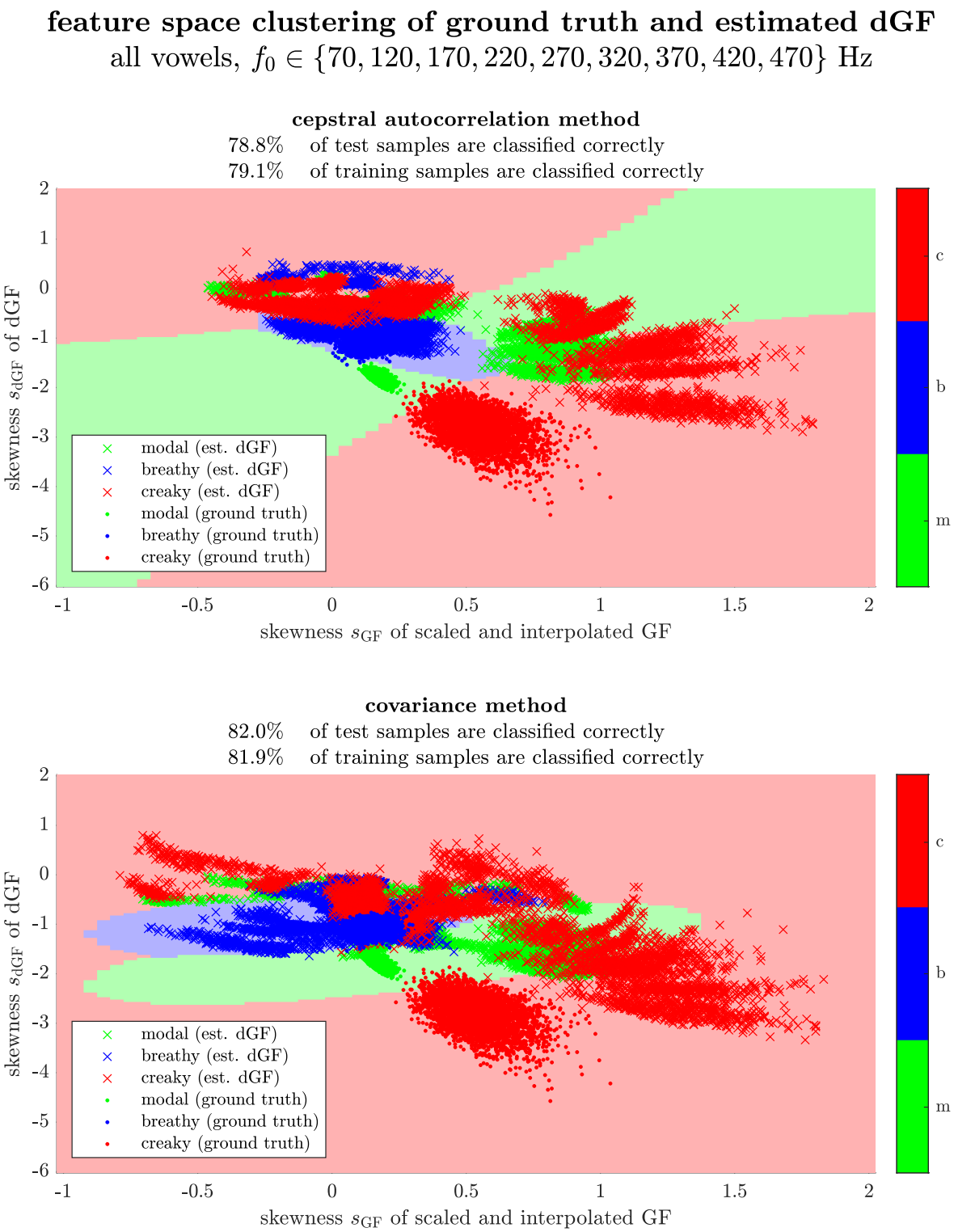


Figure A.24 *Clustering of voice quality features considering fundamental frequencies up to* $f_0 =$ 470 Hz. *Comparison of cepstral autocorrelation method and covariance method.*

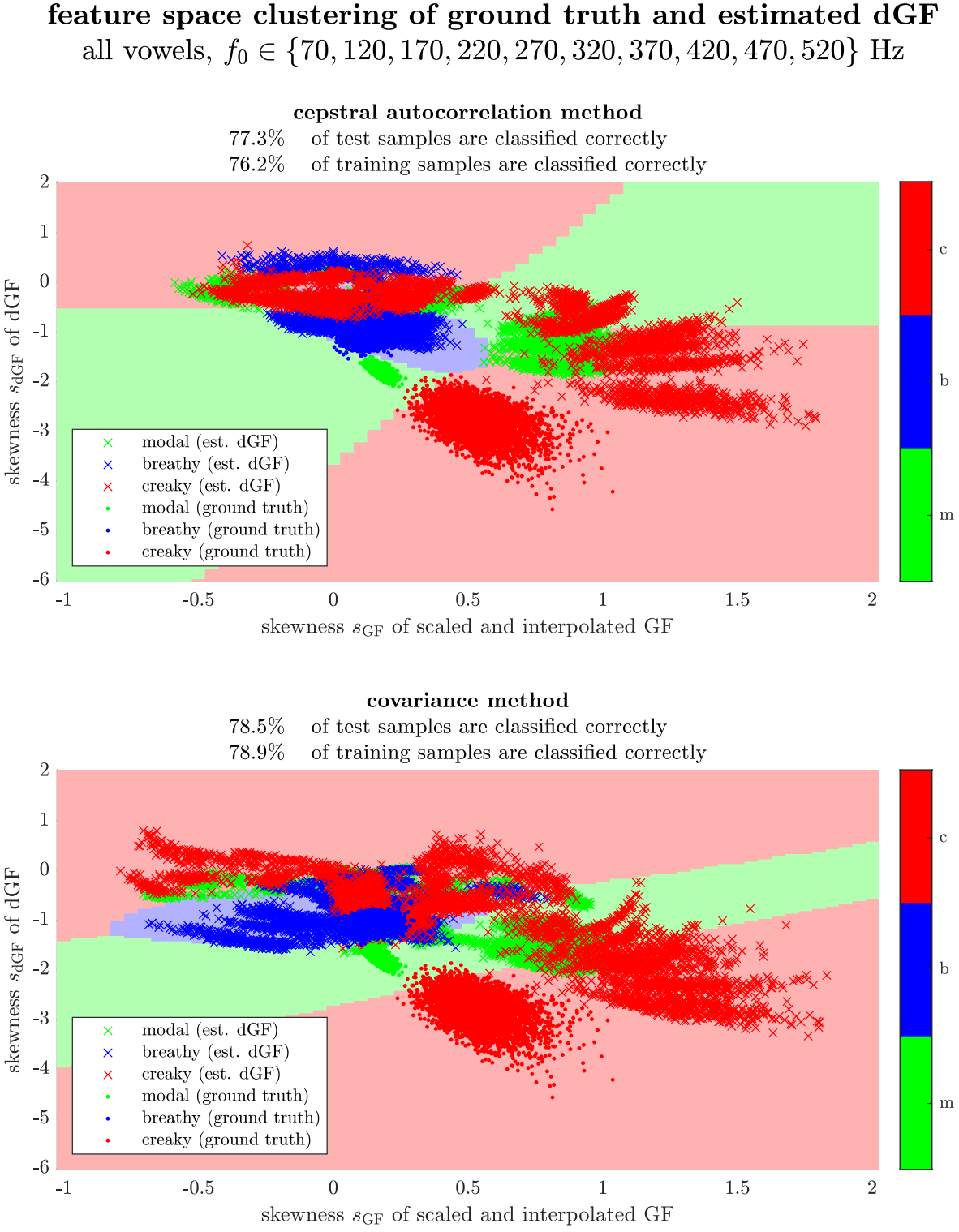


Figure A.25 *Clustering of voice quality features considering fundamental frequencies up to* $f_0 =$ 520 Hz. *Comparison of cepstral autocorrelation method and covariance method.*

Bibliography

[1] Paavo Alku, Tiina Murtola, Jarmo Malinen, Juha Kuortti, Brad Story, Manu Airaksinen, Mika Salmi, Erkki Vilkman, and Ahmed Geneid. OPENGLOT - an open environment for the evaluation of glottal inverse filtering. *Speech Communication*, 107:38 – 47, 2019.

[2] Luc Ardaillon. *Synthesis and expressive transformation of singing voice*. PhD thesis, Université Pierre et Marie Curie - Paris VI, 11 2017.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, Berlin, Heidelberg, 2006.

[4] Rémi Blandin and Manuel Brandner. Influence of the vocal tract on voice directivity. In *Proceedings of the 23rd International Congress on Acoustics*, pages 1795–1801. RWTH Aachen, September 2019.

[5] Peter Bloomfield. *Fourier Analysis of Time Series: an Introduction*. Wiley Series in Probability and Statistics. Wiley, New York, NY, second edition, 2000.

[6] Paul Boersma and David Weenink. PRAAT, a system for doing phonetics by computer. *Glot international*, 5:341–345, 01 2001.

[7] Manuel Brandner, Rémi Blandin, Matthias Frank, and Alois Sontacchi. A pilot study on the influence of mouth configuration and torso on singing voice directivity. *The Journal of the Acoustical Society of America*, 148(3):1169–1180, 2020.

[8] Manuel Brandner, Alois Sontacchi, and Matthias Frank. Real-Time Calculation of Frequency-Dependent Directivity Indexes in Singing. In *Fortschritte der Akustik*, Rostock, April 2019. DAGA.

[9] Carl. A C++ Polynomial Root Finder Using Eigen. Online Resource, July 2014. [accessed 24.02.21] , `http://www.sgh1.net/posts/cpp-root-finder.md`.

[10] Yu-Ren Chien, Daryush D. Mehta, Jón Guðnason, Matías Zañartu, and Thomas F. Quatieri. EGIFA - Evaluation of Glottal Inverse Filtering Algorithms Using a Physiologically Based Articulatory Speech Synthesizer. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 25(8):1718–1730, Aug 2017.

[11] Gilles Degottex, John Kane, Thomas Drugman, Tuomo Raitio, and Stefan Scherer. Covarep — a collaborative voice analysis repository for speech technologies. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 960–964, May 2014.

[12] Thomas Drugman. Residual excitation skewness for automatic speech polarity detection. *IEEE Signal Processing Letters*, 20(4):387–390, April 2013.

[13] Thomas Drugman, Paavo Alku, Abeer Alwan, and Bayya Yegnanarayana. Glottal source processing: From analysis to applications. *Computer Speech & Language*, 28(5):1117–1138, 2014.

[14] Thomas Drugman and Abeer Alwan. Joint robust voicing detection and pitch estimation based on residual harmonics. In *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH 2011*, pages 1973–1976, Florence, Italy, January 2011.

[15] Thomas Drugman and Thierry Dutoit. Glottal closure and opening instant detection from speech signals. In *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH 2009*, pages 2891–2894, Brighton, UK, January 2009.

[16] Thomas Drugman, Mark Thomas, Jón Guðnason, Patrick Naylor, and Thierry Dutoit. Detection of glottal closure instants from speech signals: A quantitative review. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(3):994–1006, March 2012.

[17] Michael Döllinger and Manfred Kaltenbacher. Preface: Recent Advances in Understanding the Human Phonatory Process. *Acta Acustica united with Acustica*, 102(14):195–208, February 2016.

[18] Gunnar Fant, Johan Liljencrants, and Qiguang Lin. A Four-Parameter Model of Glottal Flow. In *Quarterly Progress and Status Report*, volume 26(4) of *STL-QPSR*, pages 1–13. KTH School of Computer Science and Communication, Dept. for Speech, Music and Hearing, 1985.

[19] Mario Fleischer, Silke Pinkert, Willy Mattheus, Alexander Mainka, and Dirk Mürbe. Formant frequencies and bandwidths of the vocal tract transfer function are affected by the mechanical impedance of the vocal tract wall. *Biomechanics and modeling in mechanobiology*, 14, 11 2014.

[20] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3, 2005. `http://www.fftw.org`.

[21] GeeksforGeeks. Finding median of unsorted array in linear time using c++ stl: `calcMedian()`. Software documentation, GeeksforGeeks, 2020. [accessed 24.02.21] , `https://www.geeksforgeeks.org/finding-median-of-unsorted-array-in-linear-time-using-c-stl/`.

[22] Christer Gobl. A preliminary study of acoustic voice quality correlates. In *Quarterly Progress and Status Report*, volume 30(4) of *STL-QPSR*, pages 9–22. KTH School of Computer Science and Communication, Dept. for Speech, Music and Hearing, 1989.

[23] Bernard Gold and Lawrence R. Rabiner. Analysis of digital and analog formant synthesizers. *IEEE Transactions on Audio and Electroacoustics*, 16(1):81–94, March 1968.

[24] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[25] Gerhard Graber. *Digitale Audiotechnik 1, Skriptum*. Signal Processing and Speech Communication Lab, Graz University of Technology, Graz, 2016. Lecture Script, Version 8.61.

[26] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. Online-Resource, 2010. [accessed 24.02.21] , `http://eigen.tuxfamily.org`.

[27] James Hillenbrand, Laura Getty, Michael Clark, and Kimberlee Wheeler. Acoustic characteristics of american english vowels. *The Journal of the Acoustical Society of America*, 97:3099–3111, 06 1995.

[28] Holger Hoffmann. Matlab Central File Exchange: Violin Plot, 2021. [accessed 24.02.21] , `https://www.mathworks.com/matlabcentral/fileexchange/45134-violin-plot`.

[29] Felix Holzmüller, Paul Bereuter, Philipp Merz, Daniel Rudrich, and Alois Sontacchi. Computational Efficient Real-Time Capable Constant-Q Spectrum Analyzer. *Journal of the Audio Engineering Society*, May 2020. Convention e-Brief 567.

[30] Felix Holzmüller, Paul Bereuter, Philipp Merz, Daniel Rudrich, and Alois Sontacchi. Documentation - computational efficient real-time capable constant-q spectrum analyzer. Software documentation, Institue of Electronic Music and Acoustics, 2020. [accessed 24.02.21] , `https://git.iem.at/audioplugins/cqt-analyzer/-/blob/master/References/documentation_CQTanalyzer.pdf`.

[31] IRCAM. Linear predictive coding. Technical report, IRCAM, 2021. [accessed 28.02.21] , `http://support.ircam.fr/docs/AudioSculpt/3.0/co/LPC.html`.

[32] JUCE. Juce documentation: `dsp::Convolution`. Software documentation, Jules Storer & ROLI, 2021. [accessed 24.02.21] , `https://docs.juce.com/master/classdsp_1_1Convolution.html`.

[33] JUCE. Juce documentation: `dsp::FFT`. Software documentation, Jules Storer & ROLI, 2021. [accessed 24.02.21] , `https://docs.juce.com/master/classdsp_1_1FFT.html`.

[34] JUCE. Juce documentation: `dsp::FIR::Filter< SampleType >`. Software documentation, Jules Storer & ROLI, 2021. [accessed 24.02.21] , `https://docs.juce.com/master/classdsp_1_1FIR_1_1Filter.html`.

[35] JUCE. Juce documentation: `dsp::ProcessContextReplacing< ContextSampleType >`. Software documentation, Jules Storer & ROLI, 2021. [accessed 24.02.21] , `https://docs.juce.com/master/structdsp_1_1ProcessContextReplacing.html`.

[36] JUCE. Juce documentation: `dsp::WindowingFunction< FloatType >`. Software documentation, Jules Storer & ROLI, 2021. [accessed 24.02.21] , `https://docs.juce.com/master/classdsp_1_1WindowingFunction.html`.

[37] Tino Kluge. C++ spline interpolation: `spline.h`. Software documentation, Private, 2014. [accessed 24.02.21] , `https://kluge.in-chemnitz.de/opensource/spline/`.

[38] Branko Kovacevic, Milan Milosavljević, Mladen Veinović, and Milan Markovic. *Robust Digital Processing of Speech Signals*. Springer International Publishing, June 2017.

[39] Hui-Ling Lu and Julius Orion Smith. Glottal source modeling for singing voice synthesis. CCRMA, Stanford University, 01 2000.

[40] MathWorks. Matlab documentation: `buffer()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://de.mathworks.com/help/signal/ref/buffer.html`.

[41] MathWorks. Matlab documentation: `cumsum()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://de.mathworks.com/help/matlab/ref/cumsum.html`.

[42] MathWorks. Matlab documentation: `eig()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://de.mathworks.com/help/matlab/ref/eig.html`.

[43] MathWorks. Matlab documentation: `fft()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://www.mathworks.com/help/matlab/ref/fft.html`.

[44] MathWorks. Matlab documentation: `fftshift()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://www.mathworks.com/help/matlab/ref/fftshift.html`.

[45] MathWorks. Matlab documentation: `filter()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://www.mathworks.com/help/matlab/ref/filter.html`.

[46] MathWorks. Matlab documentation: `findpeaks()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://de.mathworks.com/help/signal/ref/findpeaks.html`.

[47] MathWorks. Matlab documentation: `fir1()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://de.mathworks.com/help/signal/ref/fir1.html`.

[48] MathWorks. Matlab documentation: `fitcsvm()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://de.mathworks.com/help/stats/fitcsvm.html`.

[49] MathWorks. Matlab documentation: `interp1()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://de.mathworks.com/help/matlab/ref/interp1.html`.

[50] MathWorks. Matlab documentation: `length()`. Software documentation, The MathWorks, Inc., 2020. [accessed 14.04.21] , `https://www.mathworks.com/help/matlab/ref/length.html`.

[51] MathWorks. Matlab documentation: `levinson()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://www.mathworks.com/help/signal/ref/levinson.html`.

[52] MathWorks. Matlab documentation: `roots()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://de.mathworks.com/help/matlab/ref/roots.html`.

[53] MathWorks. Matlab documentation: `skewness()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://de.mathworks.com/help/stats/skewness.html`.

[54] MathWorks. Matlab documentation: `trapz()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://de.mathworks.com/help/matlab/ref/trapz.html`.

[55] MathWorks. Matlab documentation: `tukeywin()`. Software documentation, The MathWorks, Inc., 2020. [accessed 24.02.21] , `https://www.mathworks.com/help/signal/ref/tukeywin.html`.

[56] Alan V. Oppenheim, John R. Buck, and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice-Hall signal processing series. Pearson Education Limited, Edinburgh Gate, Harlow, Exxex CM20 2JE, 3rd edition, 2014.

[57] Alan V. Oppenheim and Ronald W. Schafer. From frequency to quefrency: A history of the cepstrum. *IEEE signal processing Magazine*, 21(5):95–106, 2004.

[58] Athanasios Papoulis and S. Unnikrishna Pillai. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill, New York, fourth edition, January 2002.

[59] Lawrence Rabiner. Digital speech processing course - matlabcode: `cholesky.m`. Software documentation, University of California - Santa Barbara, Department of Electrical and Computer Engineering,

2014. [accessed 24.02.21] , https://web.ece.ucsb.edu/Faculty/Rabiner/ece259/digital%20speech%20processing%20course/Matlab%20Code/cholesky.m.

[60] M. Shahidur Rahman and Tetsuya Shimamura. Linear prediction using refined autocorrelation function. *EURASIP Journal on Audio, Speech, and Music Processing*, 2007(1):045962, 2007.

[61] Daniel Rudrich. Iem plugin suite repository. Software documentation, Institute of Electronic Music and Acoustics, 2020. [accessed 24.02.21] , https://git.iem.at/audioplugins/IEMPluginSuite/-/tree/master.

[62] Peter Sciri. Singing Voice Vibrato: Measurement and Modification. Master's thesis, Institute of Electronic Music and Acoustics, University of Music and Performing Arts, Graz, June 2011.

[63] Walter F. Sendlmeier and Julia Seebode. Formantkarten des deutschen Vokalsystems. *TU Berlin, Institut für Sprache und Kommunikation*, 2006.

[64] Walter F. Sendlmeier and Julia Seebode. Formantkarten des deutschen Vokalsystems — Mittelwerte und Standardabweichungen der weiblichen und männlichen Sprecher. Online-Resource, 2006. [accessed 24.02.21] , https://www.kw.tu-berlin.de/fileadmin/a01311100/Formantkarte_Standardabwch.pdf.

[65] Jules Storer and ROLI. JUCE: Jules' Utility Class Extensions. Online-Resource, 2020. Version 6.0.3, [accessed 24.02.21] , https://juce.com/.

[66] Johan Sundberg. Acoustic and Psychoacoustic Aspects of Vocal Vibrato. In *Quarterly Progress and Status Report*, volume 35(2–3) of *STL-QPSR*, pages 45–68. KTH School of Computer Science and Communication, Dept. for Speech, Music and Hearing, 1994.

[67] Li Tan and Jean Jiang. *Digital Signal Processing: Fundamentals and Applications*. Academic Press, Inc., USA, 2nd edition, 2013.

[68] William Trench. An algorithm for the inversion of finite toeplitz matrices. *Siam Journal on Applied Mathematics - SIAMAM*, 12, 09 1964.

[69] Gautam K. Vallabha and B. Tuller. Choice of filter order in lpc analysis of vowels. 2004.

[70] Peter Vary and Rainer Martin. *Digital Speech Transmission: Enhancement, Coding And Error Concealment*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2006.

[71] Saeed V. Vaseghi. *Linear Prediction Models*, chapter 8, pages 227–262. John Wiley & Sons, Ltd, 2000.

[72] Wilhelm Werner. A Generalized Companion Matrix of a Polynomial and Some Applications. *Linear Algebra and its Applications*, 55:19–36, 1983.

[73] Nathanael Yoder and Clayder Gonzalez. A c++ peak finder: Peaks::findPeaks(). Software documentation, MATLAB Central File Exchange & GitHub, 2020. [accessed 24.02.21] , https://github.com/claydergc/find-peaks.

[74] Julia Ziegerhofer. Excitation Signal Analysis - Gender Differences. Master's thesis, Signal Processing and Speech Communication Lab, Graz University of Technology, Graz, March 2018.