Karlsruhe Institute of Technology

Combined Phase and Amplitude Analysis in Harmonic Acoustic Signals for Robust Speech Recognition

Studienarbeit
von

Ralf Huber

Institut für Anthropomatik
Fakultät für Informatik

Betreuer:	Prof. Dr. rer. nat. A. Waibel
Betreuender Mitarbeiter:	Dipl.-Inform. F. Kraft

Bearbeitungszeit: 01. November 2010 – 29. April 2011

Contents

1. Introduction

1.1 Reverberation in Automatic Speech Recognition

Modern automatic speech recognition (ASR) systems work very good in close-talk situations, i.e. if the microphone is placed directly in front of the speaker's mouth or at least within a short distance. However, as the distance from the speaker to the microphone increases the word error rate (WER) also increases dramatically. An example of this degradation can be seen in fig. 4 of [Pearson96], where the word error rate of an ASR system trained on close-talk recordings increased from 25% to 65% when the speaker distance was increased from 0.9m (3ft) to 1.8m (6ft). That result was achieved using a directional microphone and the result for an omnidirectional microphone was even worse.

The first and simplest solution for the problem is to train a speech recognition system on far-distant recordings. This way, reverberations are already considered during the learning-phase of the system, which makes it more robust against reverberations than when the system was trained on close-talk recordings. While this approach indeed reduces the scale of the problem, Kinoshita et al. state in [Kinoshita05a] that for situations where the reverberant time of the room is longer than 0.5s, even the performance of those ASR systems decreases, which were trained using distant-talk.

Therefore, Kinoshita et al. draw the conclusion that dereverberation should be achieved in a pre-processing stage, before a recording is actually processed in the ASR system.

1.2 Goal of this Work

The main intention of this work is to present an algorithm which decreases the amount of reverberations in a recording. Unlike other dereverberation methods (two of them are presented in chapter 2), the new approach should not need any knowledge about the room or the speaker or any other feature of the environment. The approach can therefore be considered as a form of "blind" dereverberation.

Soon after the work on this text began it became obvious that the goal of dereverberating real speech signals with the suggested approach could not be reached within the time limit. Instead, it seemed to be a better idea to go back to more simple signals before advancing to real speech. As a result of that, the dereverberation approach is indeed presented completely, but it is only implemented to a stage where single sine waves can be dereverberated. Experiments should then be carried out to find out if dereverberation of single sine waves can be achieved and if it is reasonable to continue and implement the full system.

For the dereverberation algorithm it is needed to track the characteristics (amplitude, frequency, ...) of the recorded signal as precisely as possible. A special kind of control-loops, named phase-locked loops (PLLs), will be used for this task. As phase-locked loops are a fairly new technique in the field of automatic speech recognition, another goal of this work is to give an introduction into PLL theory, which will be done in chapter 3. The word "new" in the previous sentence is not to be understood in the way of "current" or "up-to-date", because experiments with phase-locked loops in ASR date back to 2001 [Estienne01]. It is just that PLLs are not widely used for automatic speech recognition, which is why it is possibly a good idea to have a somewhat more detailed explanation of PLLs.

The third and last goal of this work is to get a better understanding of PLLs in order to find out if they can also be used for other speech-recognition-related tasks, such as extracting features for an ASR frontend. This is also a reason why the chapter covering PLL theory turns out relatively long.

1.3 Outline

The work at hand is organized as follows: In the next chapter (ch. 2), existing approaches to dereverberation for automatic speech recognition are first presented and then their drawbacks are pointed out in ch. 2.1. After that, chapter 2.3 contains a description of the new dereverberation method. This cannot be done without making assumptions about the speech and the environment, which are also included in that chapter.

While chapter 3 contains mostly mathematical descriptions of a PLL, the actual implementation is described in ch 4.1, followed by the implementation details of the rest of the dereverberation method (ch. 4.2). Actual code is not included in the text, but it can be found in appendix A.

Some experiments were carried out to test and measure the dereverberation performance of the new approach in order to compare it to existing approaches. These experiments and their results are described in ch. 5 at the end of the work.

2. Analysis of the Problem

2.1 Existing Approaches

Existing approaches for dereverberation include beamforming using microphone arrays or blind deconvolution methods, where the room impulse response (RIR) is first estimated and then applied inversely. Both of these methods will be presented in the following two chapters in order to understand their drawbacks.

2.1.1 Microphone Arrays

A common approach to dereverberation is the use of microphone arrays in order to carry out some kind of beamforming. Beamforming is a method which allows to "listen" primarily into a specific direction by increasing the magnitude of the sound that originates from this direction.

The easiest form of a beamformer is a delay-and-sum beamformer (DSB), which - as its name implies - delays the signal from each microphone by a certain amount of time and then adds all the delayed signals to form its output. The time the sound travels from the speaker to each of the microphones differs and this difference is supposed to be cancelled by the delay applied in the beamformer.

When the delay is cancelled well, the signals from all microphones are aligned in time and by adding them, constructive interference is simulated. However, the delays in the beamformer fit only for the target direction (where the direct sound comes from), but they don't fit for all other directions (where reverberations come from). Because of this, reverberations do not only interfere constructively, but also destructively, depending on the particular layout of the room. In total, the magnitude of the direct sound is increased compared to the magnitude of the reverberations.

N. Gaubitch and P. Naylor have analyzed the theoretical performance of delay-and-sum beamformers in [Gaubitch05]. According to them, the performance of a DSB depends solely on the number of microphones used and the distance from the sound source to the closest microphone of the array.

Gaubitch and Naylor measure the dereverberation performance by calculating how much the use of a DSB improves the direct-to-reverberant ratio (DRR) compared to

a single microphone. Similar to the signal-to-noise ratio (SNR), the DRR is calculated by dividing the power of the direct sound by the power of the reverberations in a given signal. The simulations in [Gaubitch05] show that a delay-and-sum beamformer with 5 microphones can improve the DRR up to around 6.5dB for recording distances (distance to the closest microphone) between 0.5m and 3m. At a distance of 2m, DRR improvements range from around 3dB when 2 microphones are used up to 7.5dB when 7 microphones are used.

2.1.2 Blind Dereverberation

Blind dereverberation (or deconvolution) methods try to utilize an inverse filter in order to reverse the effect the RIR had on the sound. Neither has the exact RIR to be known when the dereverberation system is trained, nor has it to be known during the operation of the system, which is why these methods are called 'blind'. An example of such a system is given in [Nakatani03a] and the following steps outline its basic functionality:

1. Estimate the fundamental frequency f_0 of the speech signal.
2. Estimate the direct sound using an adaptive harmonic filter, which enhances frequency components of the recorded signal at integer multiples of f_0. For this step, it is assumed that the direct sound is a complex of harmonic frequencies, which stay constant during short periods of time.
3. Estimate and apply a dereverberation filter, which is based on the recorded reverberant signal and the estimated direct sound from step 2.

In [Nakatani03a], this procedure is done twice in order to be able to estimate f_0 more precisely in the second run from the already dereverberated output of the first run. Besides this original method, the authors of [Nakatani03a] have also published several improvements to it: In [Kinoshita05b], a method is proposed which decreases the amount of data needed to train the dereverberation filter and in [Kinoshita05a], this amount is reduced even further.

In [Nakatani03b], a third run of the basic algorithm is added, but the estimation of the direct sound in run 2 is based on the original recorded sound rather than on the output of the first run. Two different approaches for the calculation of the dereverberation filter are compared in [Nakatani07].

2.1.3 Limitations of Existing Approaches

Both methods presented in the previous two chapters have drawbacks which make them difficult to use in certain applications.

Multiple microphones are needed for microphone arrays, which cost more than a single microphone and which is a drawback when limited space is available, for example in mobile phones. When a DSB is used, the direction of arrival of a speakers voice must be estimated in order to select correct delays for the microphones, otherwise the working principle implies that the direct sound is not necessarily enhanced. In some applications, the direction of arrival is mostly limited to a few possibilities:

One example would be a voice-activated in-car entertainment/communication system, where most of the commands are issued from the driver, whose location can be predicted very good. In other applications, for example in voice-activated TV-sets, the speaker location is not known in advance.

An advantage of a DSB is that is does not have to be "trained" in any way when the direction of arrival is known, in contrast to the blind dereverberation approach presented in chapter 2.1.2. In [Nakatani03a], utterances of 5240 words are used to train a single dereverberation filter. Although the total time for these utterances is not stated in the paper, it can be expected that it is more than a minute. The reason for this is that in [Kinoshita05b] the same authors compare the results which they achieved using 60 minutes of training data to results of using only a minute of training data.

However, a minute of training data is still too long to use the system in the real world, which is why the approach was improved even further in [Kinoshita05a]. In that paper, Kinoshita et al. present an algorithm which can successfully train a dereverberation filter on just 15 seconds of reverberant speech. However, the filter is specifically designed to tackle late reflections, whereas other methods (like for example Cepstral Mean Normalization) must be used to remove early reflections. Obviously, a single algorithm which can handle both early and late reflections would be better than having to use two methods.

2.2 Basic Definitions

Before continuing any further, it is important to establish conventions about the naming and writing of mathematic expressions throughout the rest of the work.

According to Pol, there is some confusion about what is called the "phase" of a trigonometric function [Pol46], so this is a good starting point. Evidently, in the term $u(t) = a \cdot \sin(2\pi ft + \phi)$, f is called the wave's frequency and it is measured in Hz $(= \frac{1}{s})$. The factor 2π can also be combined with f to create the angular frequency, denoted by $\omega = 2\pi f$ and measured in radians. a is, of course, the amplitude, which can have all sorts of dimensions, depending on what physical effect is described by the sine wave, for example meters or volts. To follow the advice of Pol (and the german standard [DIN1311-1]), the whole argument of a trigonometric function, namely $(2\pi ft + \phi)$, will be called its phase from now on, denoted by θ, whereas ϕ will be called the initial phase offset. Both values are given in radians unless noted otherwise.

Operators will be written in curved letters and their arguments will be contained within curly brackets, for example the Fourier transform of the function $m(t)$ would be $\mathcal{F}\{m(t)\}$. Variables in the Fourier-domain will be denoted in capital letters, so $M(f)$ would be the Fourier-transform of $m(t)$.

2.3 New Approach

As a summary from the section 2.1.3, the new dereverberation system should fulfill the following algorithmic prerequisites:

- "Blind" dereverberation, i.e. the system needs no knowledge about the speaker or the characteristics of the room.
- The system needs not to be trained in any way in advance.

Before the new approach will be presented in detail, it is first assumed that the room is not altered during a segment of speech and that both the speaker and the microphone do not change their position, so the room impulse response remains constant. This can be achieved for example by observing small segments of time, throughout which the RIR is almost constant. Additionally, is is assumed that voiced speech is composed out of a set of harmonic frequencies, so

$$Speech_{voiced}(t) = \sum_{h \in \mathbb{N}} a_h(t) \cdot \sin(2\pi \cdot h \cdot f_0(t) \cdot t + \phi_h)$$

Modeling voiced speech as a sum of harmonic frequencies has proven to be a reasonable approximation, for example in [McAulay86]. Using this assumption, the parameters $a_h(t)$, $f_0(t)$ and ϕ_h can be used to model voiced speech at a given time t (ϕ_h does actually not depend on t as it will be explained in the beginning of the next section). No assumptions are made for unvoiced speech since the proposed approach works only on voiced speech. To deal with unvoiced speech segments, one could for example estimate the room impulse response using the dereverberated voiced parts of the speech. This estimated room impulse response could then be applied inversely to the recorded signal to dereverberate the unvoiced parts of the speech, too.

2.3.1 Explanation for a Constant Sine Wave

To start with a simple explanation, the direct sound is first assumed to consist only of a single sine wave with a constant frequency f, amplitude a_0 and phase offset ϕ_0. Sections 2.3.2 and 2.3.3 will deal with time-varying frequencies and amplitudes, respectively. Concerning a possibly time-varying phase offset, it is obvious that a single real physical oscillating device (like the glottis) can never produce a phase step, because it would have to oscillate infinitely fast for an infinitely short amount of time to produce such a phase step. As a result, a constant phase offset is a reasonable approximation to real voiced speech.

In a room, each single sine wave that contributes to a voiced phoneme will travel directly from the speakers mouth to the microphone if no obstacle is in the way. The diaphragm of the microphone will then oscillate due to the pressure variations of the sound wave. Later, when the first reflected sound wave arrives at the microphone, both the direct sound and the first reflection interfere. Each additional reflected sound wave, which arrives at the microphone, interferes with those which have arrived earlier. Figure 2.1 shows this process in detail.

At $t = t_0$, the direct sound arrives at the microphone. Its frequency is f, its amplitude a_0 and its phase offset ϕ_0. The first reflection arrives at $t = t_1$ and since it is a reflection of the direct sound, its frequency is also f. After it has interfered with the direct sound, the total amplitude is a_1 and the total phase offset is ϕ_1. This process is repeated with each new reflection which arrives at the microphone

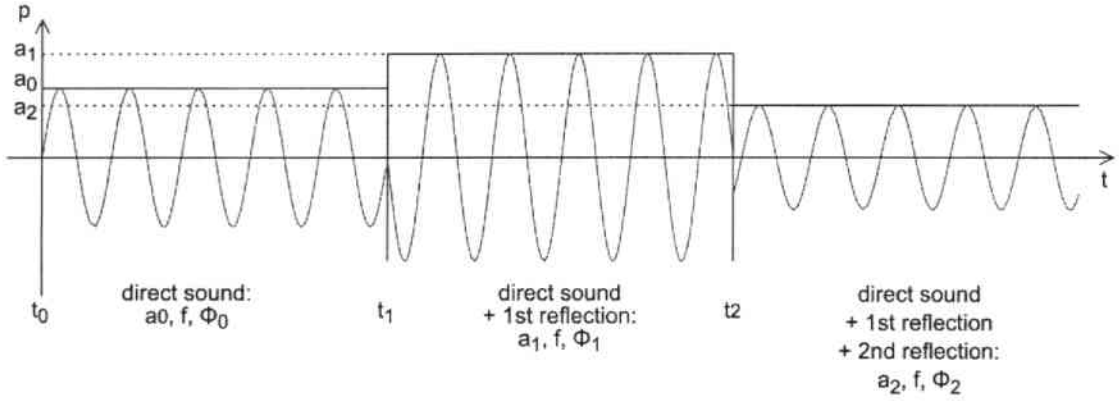


Figure 2.1: Superposition of waves at the microphone

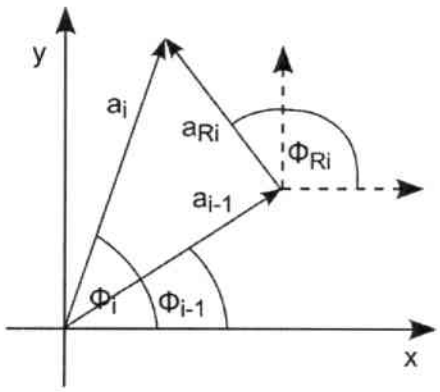


Figure 2.2: Addition of two phasors

When two waves are added using the phasor notation (see fig.2.2), the following equations hold:

$$a_{i-1} \cdot \sin(2\pi ft + \phi_{i-1}) + a_{Ri} \cdot \sin(2\pi ft + \phi_{Ri}) = a_i \cdot \sin(2\pi ft + \phi_i) \tag{2.1}$$

$$a_{i-1} \cdot \cos(2\pi ft + \phi_{i-1}) + a_{Ri} \cdot \cos(2\pi ft + \phi_{Ri}) = a_i \cdot \cos(2\pi ft + \phi_i) \tag{2.2}$$

Here, a_{Ri} and ϕ_{Ri} denote the amplitude and phase offset of the i-th. reflected wavefront, respectively. If it was possible to measure both a_{i-1}, ϕ_{i-1} and a_i, ϕ_i, eqn. 2.2 could be solved for a_{Ri}, resulting in:

$$a_{Ri} = \frac{a_i \cdot \cos(2\pi ft + \phi_i) - a_{i-1} \cdot \cos(2\pi ft + \phi_{i-1})}{\cos(2\pi ft + \phi_{Ri})} \tag{2.3}$$

After that, a_{Ri} can be substituted in eq. 2.1, which can be solved for ϕ_{Ri} afterwards:

$$\phi_{Ri} = \arctan\left(\frac{a_i \cdot \sin(2\pi ft + \phi_i) - a_{i-1} \cdot \sin(2\pi ft + \phi_{i-1})}{a_i \cdot \cos(2\pi ft + \phi_i) - a_{i-1} \cdot \cos(2\pi ft + \phi_{i-1})}\right) - 2\pi ft \tag{2.4}$$

Obviously, with the knowledge of a_{Ri} and ϕ_{Ri}, a corresponding sine wave could be generated and subtracted from the recorded signal in order to eliminate reflection i.

For this approach is it obviously needed to determine f, a and ϕ of the sine wave precisely for any given moment. Usually, a Fourier transform is used to obtain the parameters of an audio signal. However, the problem of the (discrete) Fourier

transform is that it trades temporal resolution for frequency resolution: If the Fourier transform is calculated based on a long period of time, the output "bins" of the Fourier transform are narrow, resulting in a good frequency resolution. If the Fourier transform is calculated based on only a small amount of samples, its frequency resolution is bad. For example: If each output "bin" of a FFT should represent a frequency bandwidth of 10Hz, a 4800-point FFT must be used when the sample rate is 48kHz. A 4800-point FFT at 48kHz corresponds to averaging over a timespan of 0.1s. Unfortunately, multiple reflections of a sound wave arrive at the microphone within a few milliseconds, not within hundreds of milliseconds. As a result, the FFT must be calculated based on fewer samples, for example just 48, in order to be able to measure a_i and ϕ_i between the arrival of two consecutive reflected wavefronts. A 48-point FFT however, results in output bins which are 1000Hz wide. This is clearly too much to separate multiple harmonic frequencies in voiced speech, because they are usually only between 80 and 200Hz apart.

To solve the problem of the measurement of f, a and ϕ, phase-locked loops are used in this work. Chapter 3 contains a detailed description of PLLs, for now it is enough to know that they can provide the required precision for the estimation of the sound's parameters. Their drawback is that they can only operate on a single frequency component, which is why a recorded signal has to be split up into its frequency components using bandpass filters and each band of the filter must then be passed to a single PLL.

When the parameters $f[k]$, $a[k]$ and $\phi[k]$ have been successfully estimated for each recorded sample k using a PLL, the actual dereverberation can start by determining the frequency f_0, amplitude a_0 and phase offset ϕ_0 of the direct sound. For this task, it is necessary to first detect the start of the direct sound in the recording before f, a_0 and ϕ_0 can simply be read out at that instant of time.

After the parameters of the direct sound are known, the parameters of each consecutive sample can be compared to those of the direct sound using equations 2.3 and 2.4. When the first reflection has not yet arrived at the microphone, the resulting reverb amplitude a_R will be close to zero.

However, if a reflected sound wave has arrived and interfered with the direct sound, the reverb amplitude a_R and reverb phase ϕ_R will rapidly change to accommodate the new situation. This change in reverb amplitude and phase can be detected, so the time t_1 for the arrival of the first reflected wavefront is known and the combined amplitude a_1 and phase offset ϕ_1 of the direct sound and the first reflection can be read out at t_1. With this information, the correct reverb amplitude a_{R1} and phase offset ϕ_{R1} can be calculated afterwards to produce a copy of the first reverberation, which can then be subtracted from the recording.

From then on, the parameters of all further samples can be compared to a_1 and ϕ_1, waiting for a change in reverb amplitude or reverb phase again. This procedure is then repeated until the end of the recording or until the reverberation amplitudes are so low that they don't change the sound anymore. This could be possibly the case for reflections that arrive very late after the direct sound, which means they travelled a long distance through the air and lost most of their energy on the way.

After the end of the direct sound, the measured amplitude and phase offset will still keep changing from time to time, but this does not happen due to newly incoming

reflections but due to discontinuing reflections. When the i-th. reflected wavefront arrives at the microphone at $t = t_i$, it took $t_i - t_0$ seconds to travel from the speaker to a reflective surface and to the microphone. As a result this very reflection will also disappear $t_i - t_0$ seconds after the direct sound has ended.

This can be used in the following way: At first, the end of the direct sound must be detected (for example by detecting the largest drop of the signal energy) and then, it is known that all "reflections" detected later are not new reflections but ending reflections. Because this is difficult to describe, fig. 2.3 depicts what is actually happening.

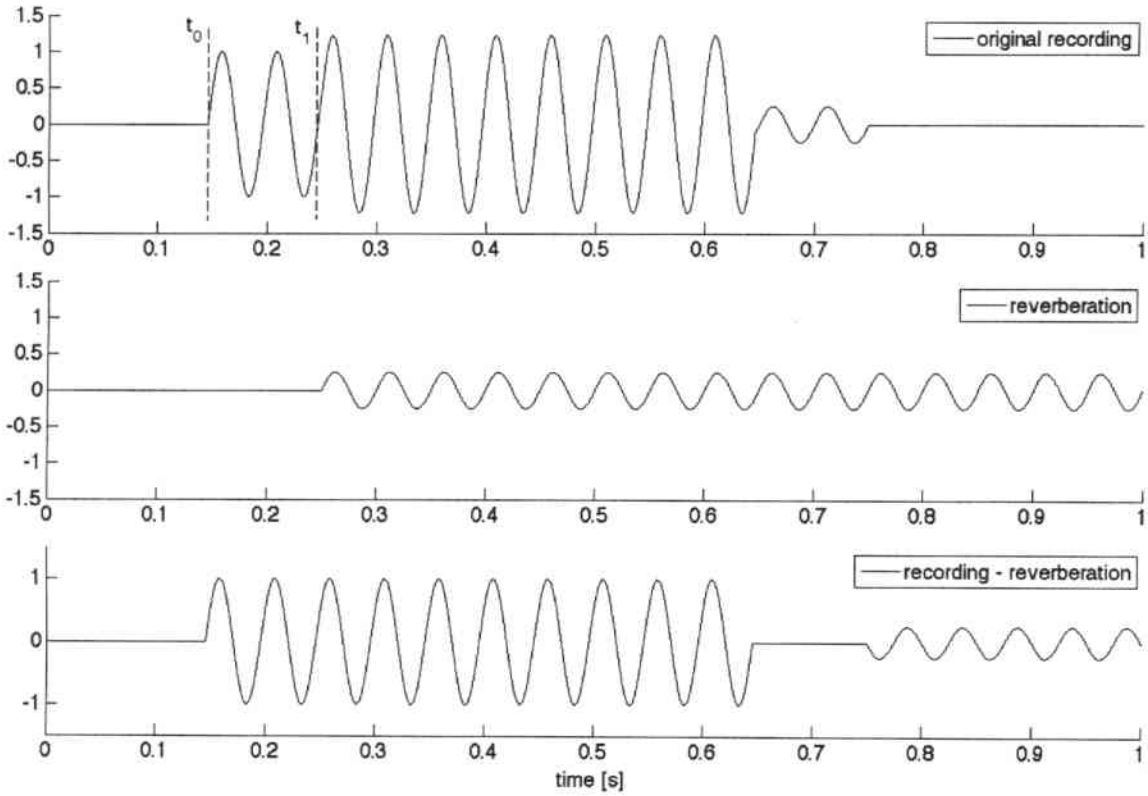


Figure 2.3: Dereverberation of a single reflection

After the direct sound has ended at $t = 0.65s$, another "reflection" would be detected at $t = 0.75s$ due to the ending of the actual reflection. After the end of the direct sound has been detected at $t = 0.65s$, it is known that the direct sound (which is the target signal for the dereverberation) is zero for $t > 0.65s$. The additional "reflection" can thus be removed in the same way as the actual reflection by producing and subtracting a corresponding "counter" sound wave.

2.3.2 Extension for a Time-Varying Frequency

When the frequency of the direct sound changes over time, equations 2.1 and 2.2 do not hold because they assume a constant frequency. However, assuming that the

PLL tracks the changing frequency, its value $f(t)$ is known for each instant of time, which allows for the following calculation:

$$\begin{aligned} s(t) &= a \cdot \sin(2\pi f(t)t + \phi) \\ &= a \cdot \sin(2\pi(f(0) + \Delta_f(t))t + \phi) \\ &= a \cdot \sin(2\pi f(0)t + \underbrace{2\pi\Delta_f(t)t + \phi}_{\phi(t)}) \end{aligned} \tag{2.5}$$

In this equations, $f(0)$ corresponds to the inital frequency of the sine wave, which it had before the frequency started to change. In situations where the frequency is permanently changing (like in speech), $f(0)$ would be the frequency at the beginning of the direct sound.

It can be seen from eq. 2.5 that is is possible to convert a signal with a time-varying frequency into a signal with a constant frequency and time-varying phase offset. With this conversion, equation 2.1 and 2.2 can be applied again. As a result, the phase offsets ϕ_{i-1} and ϕ_i in equation 2.1 and 2.2 must be replaced by time-varying functions $\phi_{i-1}(t)$ and $\phi_i(t)$, respectively. Consequently, the resulting reverb phase $\phi_{Ri}(t)$ will also be time-varying, but there is no problem with that.

2.3.3 Extension for a Time-Varying Amplitude

When a reflection is detected in a recorded signal, it is detected because the calculated reverb amplitude or phase change rapidly. However, the calculated amplitude of the reverb can change for two reasons: Firstly, the change can be really due to a reflected wavefront, which has arrived at the microphone. Secondly, it is also possible that the speaker has changed his/her voice, for example to form a new phoneme. Voiced phonemes are characterized by their formants, so two voiced phonemes can have the same fundamental frequency f_0, but they still sound different because the acoustic energy is concentrated on different frequencies. As a result, a change from one phoneme to another is actually a change of amplitude for each harmonic frequency. Such changes of amplitude would be mistaken for an incoming reverberation by the dereverberation method.

This brings up the question how an intentional change of amplitude can be distinguished from amplitude variations caused by reverberations. Luckily, the problem can be solved by considering all harmonic frequencies at once: The inverse distance law $\left(p \propto \frac{1}{r}\right)$ [Sengpiel11] describes the decrease of amplitude as a sound wave propagates away from its source. As it can be seen from the formula, the pressure p (which determines the amplitude) does not depend on the frequency of the sound, so the amplitude of a particular reflection decreases equally for all frequencies.

If the origin for a change in amplitude is not a reflection but a change in the speaker's voice, the amplitude of the wrongly calculated "reverb" would not decrease equally for all frequencies. Instead, the reverb amplitude which is calculated based on a certain frequency f_1 would not match the reverb amplitude calculated based on another frequency f_2. If such a situation is detected the algorithm should not attempt to dereverberate the recording but instead, it should adapt to the changing direct sound by changing the current reference amplitude.

Detecting the uniformity of the amplitude decrease is one way to distinguish between amplitude variations caused by reverberations and intentional amplitude changes caused by the speaker. The method fails, however, if a reflected wavefront arrives at the same time than the direct sound of a new phoneme. While this case probably does not happen very often, it might still occur from time to time. However, it is expected that after the beginning of a phoneme the speaker will not change to another phoneme for some time (unless he or she talks very fast). So, a phoneme change will most likely occur when the early reflections (which have the most energy) are already over. As a result, the amplitude of a reflection that occurs at the same time as a phoneme change will most likely be low, because the reflection is one of the late reflections. As a result of this, the intended amplitude change will be higher than the amplitude change caused by the late reflection, in which case the intended amplitude change dominates the reflection.

2.4 Summary

In this chapter, existing dereverberation methods have been presented like beamforming using microphone arrays or a blind dereverberation approach. Both of these methods have drawbacks which limit their use in real-world applications. This was the motivation to develop a new dereverberation method which works using a single microphone and without prior knowledge of the speaker or room.

The theory for the new approach was presented in section 2.3, first for the simple case of a sine wave with constant frequency, amplitude and phase offset. In the following sections, the theory has been extended to also handle signals with time-varying frequencies or amplitudes, which is needed if the algorithm is ever to be used on real speech signals.

3. Phase-Locked Loops

A phase-locked loop (PLL) is basically a feedback control system, which accepts a sinusoidal target signal as an input and then matches the frequency and initial phase of a local oscillator to the parameters of the target signal. When a PLL circuit has arrived in a state where the local replica oscillation matches the target signal (at least to a certain amount), it is considered to be "locked" or "in-lock". Furthermore, PLLs can be extended by an amplitude estimation circuit in order to find the amplitude of the target signal after the PLL has locked.

When a PLL is in-lock and the amplitude estimator has had enough time to measure the signal amplitude, all parameters of the sinusoidal target signal (frequency, amplitude and initial phase offset) are known to the user of the PLL. The purpose of this work is to find out if this information can be used to track the characteristics of sinusoidal sounds and their reverberations in a room.

According to [Best93, p. 5], Henri de Bellescize described the first PLL implementations in his work "La réception synchrone" in 1932. Today, PLLs are widely used in many electronic circuits such as FM or GPS receivers, TV sets or frequency synthesizers. Because of this, a lot of literature covering different aspects and use-cases of PLLs is available. Some of the standard works about PLLs are "Phase-Locked Loops" by R. Best [Best93] and "Phaselock Techniques" by F. Gardner [Gardner05]. "Phase-Locked Loops for Wireless Communications" by D. Stephens [Stephens02] is another book which - despite its name - also contains PLL basics and comprehensive mathematical descriptions of both single PLL components and PLL systems as a whole.

The goal of the following chapters is mainly to mathematically describe the PLL system which was used for this work. At the same time, an introduction to PLLs is given which also explains the basic working principles and the components PLLs are made of. Neither will the following chapters cover all types of PLLs, nor are they intended to give a comprehensive mathematical description of all aspects of PLL theory and design. The behavior of a PLL will only be discussed in detail for the situations where the PLL has already locked. Mathematical analyses of the unlocked state make use of nonlinear differential equations, for which no exact solution has been found until today [Best93, pp. 25]. In the textbooks mentioned above, usually

some approximations are made (such as $\sin(x) \approx x$ for small values of x) to be able to solve the equations. Nevertheless, findings derived from these approximations have proven to be reasonably valid so they will also be used in later chapters.

3.1 Types of Phase-Locked Loops

Before looking at the individual components a PLL is made of, it is important to know in which ways a PLL circuit can be implemented at all. In [Best93, p. 6 and further chapter introductions], the author R. Best describes four types of PLLs: analog PLLs (APLLs), digital PLLs (DPLLs), all-digital PLLs (ADPLLs) and software PLLs (SPLLs).

APLLs are also called linear PLLs in [Best93], hence another abbreviation for them is LPLLs. However, the term APLL will be used for an analog phase-locked loop in this work. As the name suggests, APLLs are built solely out of analog circuit components, such as capacitors, resistors, inductors and the like. As it is the case with all analog components, these parts suffer from production tolerances and spread, which would have to be taken into account if one wanted to be really precise. They are also not well applicable for experiments because it would be necessary to replace components and to rewire them for each new parameter setting. Their advantage is that they are not subject to the Nyquist sampling theorem and as a result of that, APLLs can operate on signals with very high frequencies without using expensive fast analog-digital converters.

ADPLLs on the other hand are built completely out of logical circuits like counters or flip-flops. ADPLLs can either operate on binary digital signals (square waves) or on discrete, digitally sampled analog signals, i.e. data words. A drawback of all digital systems is of course the effect of the sampling theorem which results in the fact that only signals with frequencys below $\frac{1}{2} \cdot f_{sampling}$ can be processed in an ADPLL. Additionally, sampling errors always arise when working with sampled analog signals. Like APLLs, classic ADPLLs are also built in hardware which reduces their suitability for experiments. However, reconfiguring an IC is easier than soldering new components on a printed circuit.

DPLLs are a combination of APLLs and ADPLLs, having some digital parts mixed with analog parts. Depending on which parts are digital and which are analog, they might have some of the (dis)advantages of APLLs and some of ADPLLs.

The last type of the four are software PLLs. As the name suggests, these PLLs are fully realised as software components and as such they can be programmed to act like any of the other PLL types, given the sample rate is high enough to fulfil the sampling theorem. They can easily be reconfigured or reprogrammed to suit different needs which makes them ideal for simulations and research. Furthermore, it is possible to use parts and operations in an SPLL which cannot be built as an analog circuits, for example the trigonometric functions. Depending on the complexity of a PLL design and the computer hardware used to run the software, execution can be slow and not suitable for real-time operation, even on modern computers.

For this paper, an SPLL system based on an analog PLL has been implemented using Matlab and Simulink from The Mathworks, Inc. Some of the reasons for this decision have been:

- The system can be implemented and tested without buying hardware and building actual circuits.
- Campus licenses for Matlab and Simulink are available free of charge at the KIT software shop.
- The PLL can be easily reconfigured to test different configurations and parameters.
- Additional components can be built in easily.
- Many audio test files can be processed automatically and evaluated using scripts.
- Simulink models can be automatically compiled to create programs for commercially available FPGA or DSP microchips.

3.2 Components of a PLL

Fig. 3.1 shows the three basic components of a PLL system:

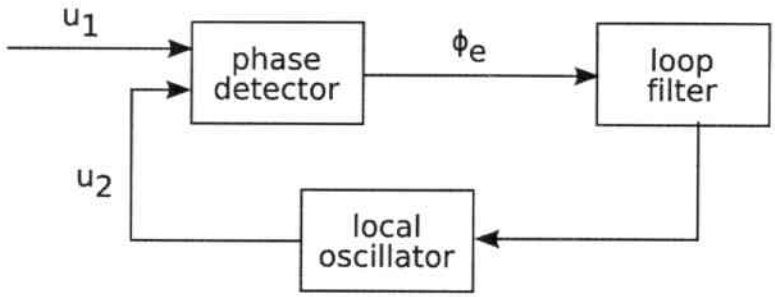


Figure 3.1: diagram of a classical PLL circuit

The target signal has a sinusoidal shape and it is denoted by u_1. The local oscillator outputs another sinusoidal signal, which will be named u_2 hereon. Mathematically, these signals can be described as follows:

$$u_1(t) = a_1 \cdot \sin(2\pi f_1 t + \phi_1) \tag{3.1}$$

$$u_2(t) = a_2 \cdot \sin(2\pi f_2 t + \phi_2) \tag{3.2}$$

u_1 and u_2 are compared in the phase detector and the output of an ideal phase detector would be $\phi_e = \phi_1 - \phi_2$. ϕ_e is called phase error or phase difference between u_1 and u_2. The phase error is then used as an input for the loop filter, which is for example a lowpass filter. The design of the loop filter is most crucial for the operation of the loop, because it determines the amount of signal noise a PLL can handle or the time it takes, until the PLL gets locked. The output of the loop filter is a control signal, which is used to increase or decrease the instantaneous frequency $\hat{f}_2$ of the local oscillator's sinusoidal output.

In traditional PLLs, only the frequencies f_1 and f_2 as well as the initial phase offsets ϕ_1 and ϕ_2 match when the PLL is locked. However, the amplitudes a_1 and a_2 are not equal and as a result, the PLL input $u_1(t)$ and its output $u_2(t)$ are not equal, too.

Usually, the local oscillator produces a sine wave with a default amplitude of $a_2 = 1$, whereas the amplitude of the waveform stored in a .wav audio file is within the interval $[0, 1]$, depending on the sound volume. As it was shown in chapter 2, signal amplitude is an important property when tracking reverberations in an audio recording, hence it needs to be estimated. This can be done by an additional amplitude estimator, as seen in fig. 3.2. The additional circuit estimates a_1, so it can be multiplied by the replica signal (whose amplitude a_2 equals 1) later. It can be seen in the diagram that the amplitude estimator is not part of the actual loop, so it doesn't contribute to the behavior of the loop in terms of control theory. As a result, the explanation of the amplitude estimator will be given at the end of chapter 3 in section 3.5.

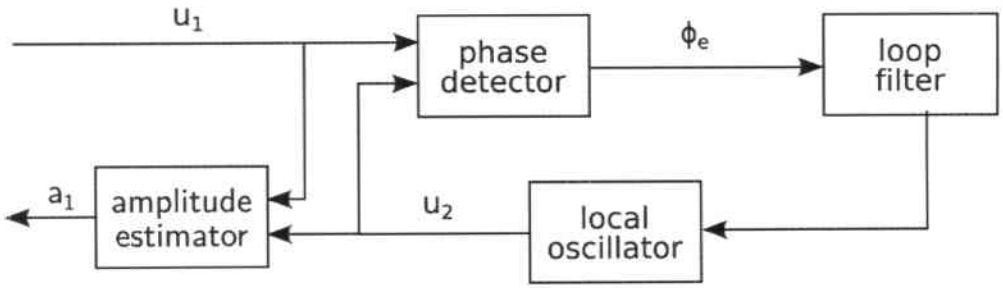


Figure 3.2: diagram of a PLL circuit including an amplitude estimator

3.2.1 Phase Detector

As already stated, the phase detector (usual abbrev.: PD) calculates the phase error of the two signals $u_1(t)$ and $u_2(t)$ (fig. 3.1). A very common phase detector (for APLLs), which is presented and analyzed in many books about PLLs, is the multiplying phase detector. In the next section, the multiplying phase detector will be analyzed and the reason why it has not been used will be presented.

For the analysis of any PD, it is assumed that the frequencies f_1 and f_2 already match, so the ideal output of the phase detector would simply be $\phi_e = \phi_1 - \phi_2$. As it can be seen in the next section, the output of the multiplying phase detector is rather different so the Hilbert Phase detector will be introduced in the next but one section to have an alternate solution.

3.2.1.1 Multiplying Phase Detector

As the name suggests, the multiplying PD simply consists of a multiplier which calculates the product $u_1(t) \cdot u_2(t)$. This yields the following result:

$$\begin{aligned} u_1(t) \cdot u_2(t) &= a_1 \cdot \sin(2\pi f t + \phi_1) \cdot a_2 \cdot \sin(2\pi f t + \phi_2) \\ &= a_1 \cdot a_2 \cdot \sin\left(\frac{4\pi f t + 2\phi_1 + \phi_2 - \phi_2}{2}\right) \cdot \sin\left(\frac{4\pi f t + 2\phi_2 + \phi_1 - \phi_1}{2}\right) \\ &= a_1 \cdot a_2 \cdot \sin\left(\frac{\alpha + \beta}{2}\right) \cdot \sin\left(\frac{\alpha - \beta}{2}\right) \end{aligned}$$

In the last line, the substitutions $\alpha = 4\pi f t + \phi_1 + \phi_2$ and $\beta = \phi_1 - \phi_2$ have been made, because after that it is possible to apply the following trigonometric identity:

$$\sin\left(\frac{\alpha + \beta}{2}\right) \cdot \sin\left(\frac{\alpha - \beta}{2}\right) = -\frac{1}{2} \cdot (\cos\alpha - \cos\beta)$$

After applying the identity and re-substituting the values for α and β, the result is:

$$\begin{aligned} u_1(t) \cdot u_2(t) &= -\frac{a_1 a_2}{2} \cdot (\cos(4\pi f t + \phi_1 + \phi_2) - \cos(\phi_1 - \phi_2)) \\ &= \frac{a_1 a_2}{2} \cdot \cos(\underbrace{\phi_1 - \phi_2}_{\phi_e}) - \frac{a_1 a_2}{2} \cdot \cos(\underbrace{4\pi f}_{\text{double frequency component}} t + \phi_1 + \phi_2) \end{aligned} \tag{3.3}$$

As it can be seen from eqn. (3.3), multiplying both signals does not directly result in ϕ_e but rather in the cosine of it as well as another cosine component having twice the frequency of the target signal. The first thing to do is to filter out the double frequency component using a lowpass filter. This filter must not be confused with the actual loop filter, although a loop filter could be designed in such a way, that it also filters out the double frequency component.

At first, the lowpass filter does not seem to be a huge problem, unless one recalls that the PLL design in this work is intended to operate over the whole freqency spectrum of human speech. Because of this, it has to work at input frequencies of about 100Hz as well as, for example, 2kHz. At 2kHz, the lowpass filter should therefore eliminate frequencies of 4kHz and above. If the same lowpass filter was used for an input signal of 100Hz, too, it would obviously not be able to remove the double frequency component at 200Hz. Because of the fact that a PLL can only track one frequency at once, many PLLs will be used to track each harmonic frequency in speech. With multiplying phase detectors, each of these PLLs would need a specifically designed lowpass filter. This is one of the reasons why the multiplying phase detector was not used.

When F_{LOW} denotes an appropriate lowpass filtering operation, the result is

$$F_{LOW}\{u_1(t) \cdot u_2(t)\} \approx \frac{a_1 a_2}{2} \cdot \cos(\phi_e) \tag{3.4}$$

$$\Rightarrow \phi_e \approx \arccos\left(\frac{2}{a_1 a_2} \cdot F_{LOW}(u_1(t) \cdot u_2(t))\right) \tag{3.5}$$

Now the biggest drawback of multiplying phase detectors gets obvious: As one can see, the right hand side of equation 3.4 depends on a_1 and a_2. The factor $a_2/2$ is down to the local oscillator (usually, $a_2 = 1$) and the PLL designer can therefore compensate for it. a_1, however, corresponds to the power of the target signal, which is probably not known by the designer (depending on the use-case of the PLL). Therefore, the multiplying phase detector yields an undesirable behavior when the power of u_1 changes. So, if it cannot be guaranteed that a_1 is constant (like in recordings of human speech), some sort of automatic gain control (AGC) needs to be used [Stephens02, p. 19] in order to keep a_1 at a certain known value.

3.2.1.2 Hilbert Phase Detector

The Hilbert phase detector works on digitally sampled versions of analog signals, so it is best used in an ADPLL or SPLL. Descriptions of this detector can be found for example in [Stephens02, p. 270], which refers to [Best93, p. 186]. The name of the Hilbert Phase detector comes from the Hilbert transform, on which it is based. The Hilbert transform (invented by the mathematician David Hilbert) will be denoted by $\mathcal{H}$ from now on. It is an operation which shifts the phase of all frequency components of a signal individually by $-\pi/2$ radians, i.e.

$$\mathcal{H}\{\sin x\} = -\cos x$$

For the Hilbert phase detector, both the signals u_1 and u_2 must be manipulated using a Hilbert transform which leads to the signals $\overline{u_1}$ and $\overline{u_2}$, respectively:

$$\begin{aligned} u_1(t) &= a_1 \cdot \sin(\omega_1 t + \phi_1) \\ \overline{u_1}(t) &:= \mathcal{H}\{u_1(t)\} = a_1 \cdot \sin\left(\omega_1 t + \phi_1 - \frac{\pi}{2}\right) = -a_1 \cdot \cos(\omega_1 t + \phi_1) \\ u_2(t) &= a_2 \cdot \sin(\omega_2 t + \phi_2) \\ \overline{u_2}(t) &:= \mathcal{H}\{u_2(t)\} = a_2 \cdot \sin\left(\omega_2 t + \phi_2 - \frac{\pi}{2}\right) = -a_2 \cdot \cos(\omega_2 t + \phi_2) \end{aligned} \tag{3.6}$$

In these equations, f_1 and f_2 have been removed by the substitution $\omega_i = 2\pi f_i$ to make the equations easier to read. Like for the analysis of the multiplying PD, $\omega_1 = \omega_2 = \omega$ is assumed, so the desired output of the phase detector is $\phi_e = \phi_1 - \phi_2$. Within the phase detector, the following operations are performed:

$$\begin{aligned} sig_1(t) &:= u_1(t) \cdot u_2(t) + \overline{u_1}(t) \cdot \overline{u_2}(t) \\ &= a_1 a_2 \cdot (\sin(\omega t + \phi_1) \cdot \sin(\omega t + \phi_2) + \cos(\omega t + \phi_1) \cdot \cos(\omega t + \phi_2)) \end{aligned}$$

Using the substitutions $\alpha = \omega t + \phi_1$ and $\beta = \omega t + \phi_2$ and the trigonometric identity $\sin\alpha \cdot \sin\beta + \cos\alpha \cdot \cos\beta = \cos(\alpha - \beta)$, this equation can be transformed to

$$\begin{aligned} sig_1(t) &= a_1 a_2 \cdot \cos\left(\omega t + \phi_1 - (\omega t + \phi_2)\right) \\ &= a_1 a_2 \cdot \cos(\phi_1 - \phi_2) \\ &= a_1 a_2 \cdot \cos(\phi_e) \end{aligned} \tag{3.7}$$

Eq. 3.7 still depends on the input amplitude a_1 and on its own, it would therefore be of no more use than the multiplying phase detector. However, another signal combination can be produced:

$$\begin{aligned} sig_2(t) :&= u_1(t) \cdot \overline{u_2}(t) - \overline{u_1}(t) \cdot u_2(t) \\ &= -a_1 a_2 \cdot (\sin(\omega t + \phi_1) \cdot \cos(\omega t + \phi_2) - \cos(\omega t + \phi_1) \cdot \sin(\omega t + \phi_2)) \\ &= -a_1 a_2 \cdot \sin(\omega t + \phi_1 - (\omega t + \phi_2)) \\ &= -a_1 a_2 \cdot \sin(\phi_1 - \phi_2) \\ &= -a_1 a_2 \cdot \sin(\phi_e) \end{aligned} \tag{3.8}$$

For this conversion, the arguments of the trigonometric functions can be substituted by α and β again, before applying the identity $\sin\alpha \cdot \cos\beta - \cos\alpha \cdot \sin\beta = \sin(\alpha - \beta)$.

After that, $sig_2(t)$ is divided by $sig_1(t)$, which results in:

$$-\frac{sig_2(t)}{sig_1(t)} = -\left(\frac{u_1 \cdot \overline{u_2} - \overline{u_1} \cdot u_2}{u_1 \cdot u_2 + \overline{u_1} \cdot \overline{u_2}}\right)(t) = -\frac{-a_1 a_2 \cdot \sin(\phi_e)}{a_1 a_2 \cdot \cos(\phi_e)} = \tan(\phi_e) \tag{3.9}$$

As sig_2 is divided by sig_1, the result gets independent of any signal amplitudes. Additionally, the Hilbert phase detector does not need an additional lowpass-filtering like the multiplying phase detector.

Besides the needed arctan function, which can easily be implemented in software, the only remaining difficulties in the implementation of the Hilbert phase detector are the actual Hilbert transforms needed to produce $\overline{u_1}$ and $\overline{u_2}$. However, there are local oscillators which produce not only a sine wave (u_2) of a given frequency, but also a corresponding cosine wave. This cosine wave can simply be multiplied by -1 to get the Hilbert transform of the sine wave, i.e. $\overline{u_2}$.

As a result, only $\overline{u_1}$ must be calculated by a Hilbert transform effectively. The actual implementation of a Hilbert filter will be discussed in ch. 4.1.2.

3.2.2 Local Oscillator

The local oscillator of a PLL has to generate a sine wave with a certain instantaneous frequency (and also a cosine wave at the same frequency if a Hilbert phase detector is to be used). In APLLs, the local oscillator is typically a voltage-controlled oscillator (VCO). This means, that the voltage applied at the input of a VCO determines the instantaneous frequency of its output. As the SPLL presented in this work resembles an APLL, a discrete-time VCO has been used. This is simply a VCO whose output is digitally sampled at a certain sampling rate. Other PLL implementations might also feature a current-controlled oscillator, for example, but this depends on the application the PLL is used for.

In addition to the discrete-time VCO, Simulink also provides a so-called numerically-controlled oscillator (NCO), which can be used (almost) like a VCO. Mathematically, they should both perform the same task and so the formulas describing the discrete-time VCO should also fit the NCO. In the Simulink NCO block, the sin function is realized using a precalculated lookup table, which stores the result of $\sin(x)$ for any given x. This way of calculating trigonometric functions is usually faster than using

the respective functions from a software math library, which is what the discrete-time VCO block does. Additionally, the amount of entries in the lookup table can be controlled explicitly, which allows/forces the designer to decide between memory consumption and mathematical precision. However, Simulink's NCO block has not been used in this work, because there is possibly a flaw in its implementation, which will be explained later, after the mathematical description of a VCO.

A VCO in Simulink has only one input port and the value $u_{in}(t)$ applied at it determines the instantaneous frequency of the sinusoidal VCO output. In a real physical circuit, the unit of the input would be volt, so it makes sense to specify the sensitivity (or gain) κ of a VCO in $\frac{Hz}{V}$. Thus, κ_0 controls by how many Hz the instantaneous frequency of the output changes when u_{in} is changed by one (volt).

The behavior of a VCO can be further adjusted by another parameter, $f_0[= Hz]$. It is called the quiescent frequency and it specifies the VCO output frequency for an input of 0V. In some applications, for example in FM demodulation, where there is a known carrier frequency f_1, f_0 can be set to f_1. As a result, the PLL only has to control the phase shift of the VCO output in these cases, but not its frequency.

Given specific values for κ_0 and f_0 and a certain function $u_{in}(t)$, the purpose of the VCO is to produce a sine wave with an instantaneous frequency $\hat{f}(t) = f_0 + \kappa_0 u_{in}(t)$. By definition, the instantaneous frequency $\hat{f}(t)$ of a sinusoid like $g(t) = \sin(\theta(t))$ is the derivative of its phase:

$$\hat{f}(t) = \frac{1}{2\pi}\frac{\mathrm{d}\theta}{\mathrm{d}t} \tag{3.10}$$

Thus, it is not sufficient to simply calculate $u_2(t) = \sin(2\pi(f_0 + \kappa u_{in}(t))t)$ in the VCO. Using eq. 3.10 and the product rule, the instantaneous frequency of the resulting sine wave would be:

$$\hat{f}_2(t) = \frac{1}{2\pi}\frac{\mathrm{d}}{\mathrm{d}t}\big(2\pi\big(f_0 + \kappa u_{in}(t)\big)t\big) = f_0 + \kappa u_{in}(t) + \kappa u'_{in}(t) \cdot t$$

Obviously, this is not the desired output of $\hat{f}_2(t) = f_0 + \kappa u_{in}(t)$. To obtain the correct output, the calculation in the VCO must be:

$$u_2(t) = \sin\left(2\pi \int_0^t \underbrace{f_0 + \kappa u_{in}(t)}_{\hat{f}_2(t)} \mathrm{d}t + \phi_0\right) \tag{3.11}$$

Deriving the phase of eq. 3.11 by means of eq. 3.10 yields the proper value for the instantaneous frequency, so eq. 3.11 is the correct equation for a VCO.

In the beginning of this section it was mentioned that the Simulink NCO and VCO blocks differ. One of these differences is that the VCO block implements eq. 3.11 directly, i.e. the input to the VCO is actually u_{in}. The NCO block, however, does not expect u_{in} as its input, but $\hat{f}_2 = f_0 + \kappa u_{in}$. As a result, $\hat{f}_2$ would have to be calculated outside of the NCO block.

In eq. 3.11, ϕ_0 is an optional (constant) initial phase offset for the VCO sinusoid. Since the initial phase offset ϕ_1 of the target signal is not known, it is impossible to set the VCO's initial phase offset ϕ_0 close to ϕ_1. Thus, ϕ_0 can be set to 0, because

0 is no better or worse choice than any other value. With this, eq. 3.11 can be transformed as follows.

$$\begin{aligned} u_2(t) &= \sin\left(2\pi \int_0^t f_0 + \kappa u_{in}(t)\mathrm{d}t + 0\right) \\ &= \sin\left(2\pi \int_0^t f_0 \mathrm{d}t + 2\pi \int_0^t \kappa u_{in}(t)\mathrm{d}t\right) \\ &= \sin\left(2\pi f_0 t + \underbrace{2\pi \int_0^t \kappa u_{in}(t)\mathrm{d}t}_{\phi_2(t)}\right) \end{aligned} \tag{3.12}$$

Obviously, u_2 can be seen as a sine wave at a frequency of f_0 Hz and with a changing phase offset of $\phi_2(t)$ radians. Now, it is important to recall the purpose of a PLL: A PLL is ought to be a servo loop for the phase of the VCO output signal. As a result, the important property of eq. 3.12 is not the actual resulting signal $u_2(t)$, nor the instantaneous frequency $\hat{f}_2(t)$, but only the phase offset $\phi_2(t)$. This is the only information that matters in the phase detector afterwards.

Eq. 3.12 models the VCO output in time domain, but as the goal of this work is to build a digital feedback control system, it is important to know the z-domain transfer function of each component. In section 3.3, these transfer functions will then be used to analyze the stability of the feedback control system.

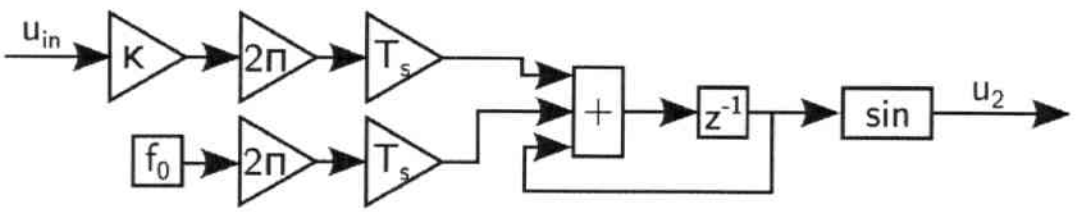


Figure 3.3: Simplified diagram of the Simulink discrete-time VCO block

Figure 3.3 is a simplified schematic drawing of the internals of a (Simulink) VCO. Obviously, for every sampling step, the integration result is calculated as the sum of the integration result from the previous sampling step plus the input to the integrator from the previous sampling step. Before adding the current input to the total sum, it has to be scaled by T_s (the sampling time), because otherwise changing the sampling rate would also change the result of the integration. For instance, if the sampling rate was increased, the VCO input would be added to the sum more often in a given period of time, which would lead to a wrong result. According to [Stephens02, p. 216], this kind of discrete integration is also known as "Forward Euler" integration (in contrast to "Backward Euler" integration, where the delay block would be placed in the backward part of the loop).

As an equation, the integration in fig. 3.3 looks like:

$$\mathrm{integrator}_{\mathrm{out}}[t] = \mathrm{integrator}_{\mathrm{out}}[t-1] + T_s \cdot \mathrm{integrator}_{\mathrm{in}}[t-1]$$

Here, t specifies the index of the sampling step, so $t-1$ corresponds to the sampling step just before step t. Now that the integration method is known, the z-domain transfer function $F_{VCO}[z]$ can be derived. As stated before, only the phase offset $\phi_2(t)$ is important, so f_0 must not be taken into account in the transfer function.

$$\begin{aligned} & \Phi_2[z] = z^{-1}\Phi_2[z] + 2\pi\kappa T_s \cdot z^{-1}U_{in}[z] \\ \Rightarrow \quad & \Phi_2[z] - z^{-1}\Phi_2[z] = 2\pi\kappa T_s \cdot z^{-1}U_{in}[z] \\ \Rightarrow \quad & \Phi_2[z](1 - z^{-1}) = 2\pi\kappa T_s \cdot z^{-1}U_{in}[z] \\ \Rightarrow \quad & F_{VCO}[z] := \frac{\Phi_2[z]}{U_{in}[z]} = 2\pi\kappa T_s \frac{z^{-1}}{1 - z^{-1}} \end{aligned} \tag{3.13}$$

This result differs from the equation given in [Stephens02, p. 216, eq. 7-57] by a factor of $2\pi\kappa$ due to the fact that the Simulink NCO expects the unit for the input to be Hz rather than radians and Stephens has not included the gain κ in the transfer function for the digital VCO, as opposed to the analog VCO.

3.2.3 Loop Filter

From the three components of a classical PLL, the phase detector and the local oscillator have already been discussed in the previous sections. This section will therefore cover the loop filter. However, before covering specific filter designs, the definition of the order of a PLL must be given.

3.2.3.1 PLL Order

In terms of control theory, a PLL in total can be regarded as a servo loop for the phase of the incoming target signal. As a result, it is possible to set up a transfer function which describes the effect the loop has on the phase of the target signal. This will be done in chapter 3.3.

For now, it is enough to recall eq. 3.13 from chapter 3.2.2. The transfer function of a VCO has one pole for $z = 1$, so the order of the VCO's transfer function is one. With respect to control theory, the phase detector realizes a negative feedback, because its output is $\phi_e = \phi_1 - \phi_2$. It does therefore not introduce any further poles to the system as a whole.

As a result, the total order of the PLL is determined by the order or the loop filter plus one for the pole of the VCO. A PLL will consequently be considered a n^{th} order PLL if the order of the loop filter is $n-1$.

According to [Stephens02, p. 16-18], loop filters of order zero, i.e. simple linear gain factors, manage to adjust the phase offset ϕ_2 of the local oscillator if the initial phase offset of the target signal follows a step function. Thus, when ϕ_1 changes to another value instantaneously, ϕ_2 will get equal to ϕ_1 after a certain amount of time. However, PLLs with zero order loop filters cannot compensate input changes of higher order, which means that zero order PLLs are not able to follow the target signal if the change of ϕ_1 can be described by a ramp function, a quadratic equation or any polynomial of even higher order.

On page 29 of [Stephens02], the author explains that by increasing the order of the phase locked loop, "it tends to compensate for an instantaneous change in the next

higher derivative of the input". For example, PLLs of order two are able to follow the target signal even if its initial phase offset is constantly increasing [Stephens02, p. 38]. According to eq. 3.10, a linear variation of the target signal's phase offset amounts to a step change of its instantaneous frequency. This can be seen in fig. 3.4(b) on page 24.

Phase variations of the next higher order can be described by quadratic equations, so by using eq. 3.10 again, they are equal to a linear ascent of the target signal's frequency, as depicted in figure 3.4(c).

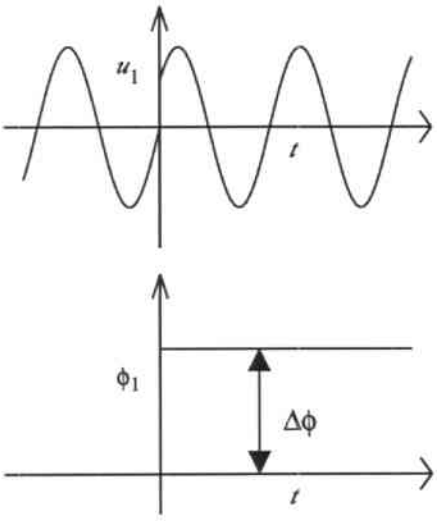
At first sight, it seems desirable to increase the order of a PLL as much as possible for the tracking of harmonic frequencies in human speech, so it can follow even the most complex input characteristics. However, by looking at the filter design equations following in section 3.4, it can be seen that a specific design of a PLL is always a trade-off between tracking stability and tracking speed. So, using a higher filter order makes the loop more stable when the input changes slowly, but it also increases its reaction time, which might render the loop too slow to follow the fast frequency changes occuring in human speech. Another drawback of high order loops is the increased computational load, which should usually be kept low.

The biggest problem is finally not the computational load, but the complexity of the filter design. For 1^{st} order loop filters, the filter coefficients can be calculated by equations which have a real physical meaning, like for example the time it takes for the PLL to lock or maximum signal-to-noise ratio at which the PLL can lock at all. Loop filters of order two and higher must be designed by parameters without a direct physical meaning, like the filter's phase margin, its unity gain crossover frequency or the position of the filter's poles in a root locus plot. On page 103 of [Stephens02], the author even suggests iterating some design parameters until a good result is achieved.

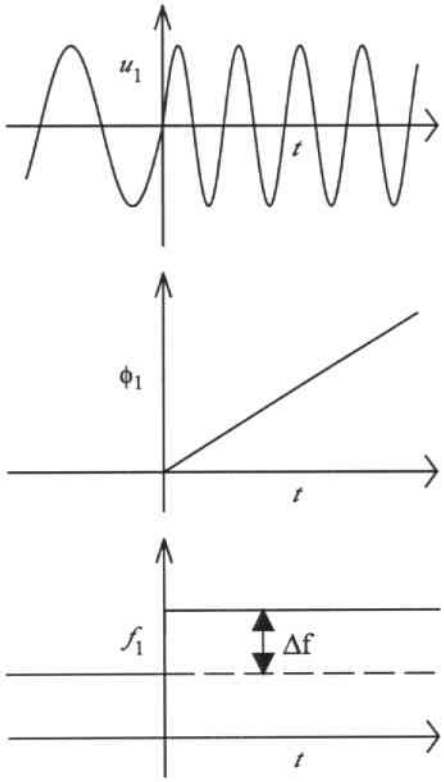
As a result, 1^{st} order loop filters have been used in this work. The equations which can be used to specify their parameters will be presented in section 3.4, after the transfer functions for both the loop filter and the PLL as a whole have been derived.

3.2.3.2 First Order Loop Filter

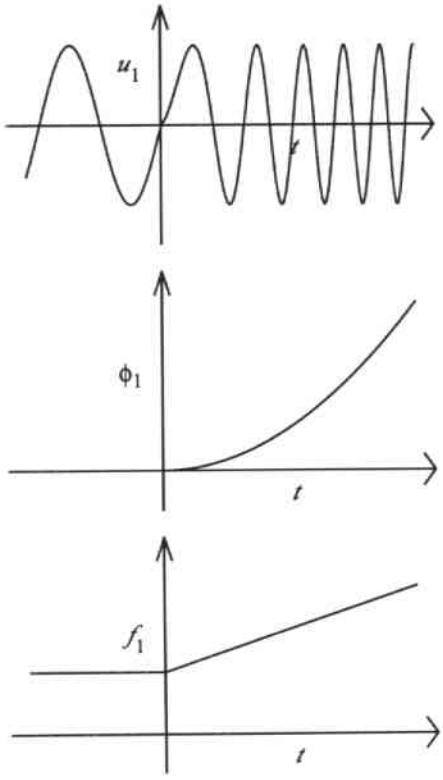
There are many ways how a first order loop filter can be designed, for example as a simple first order lowpass, as a passive lead-lag filter or as an active lead-lag filter [Stephens02, p. 31]. For this work, the active lead-lag filter (= PI controller) has been chosen, because a loop with an active lead-lag filter has an infinite pull-in range [Best93, p. 41]. This means that such a PLL can theoretically lock onto any frequency, no matter what the current instantaneous frequency $\hat{f}_2$ of the VCO is. Depending on the frequency offset $\Delta f = \hat{f}_1 - \hat{f}_2$, the locking process might take some time, but sooner or later the PLL will get locked. PI is actually an abbreviation for "proportional and integrating", which means that the output of the filter depends both linearly on its input but also on the integral of its input over time. This integrating characteristics allows the output of a digital PI controller to increase infinitely (at least theoretically, as long as there is no overflow in any variable), which, in turn, allows for an infinite pull-in range. On the other hand, PLLs with first order lowpass filters or passive lead-lag filter have limited pull-in ranges.



(a) phase step



(b) frequency step = phase ramp



(c) frequency ramp

Figure 3.4: Possible characteristics of a PLL input.

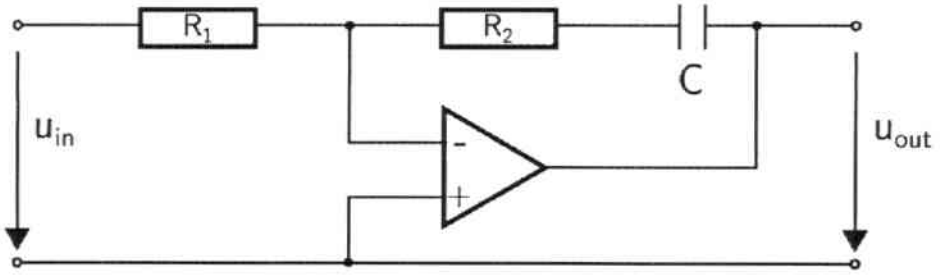


Figure 3.5: Schematic diagram of an analog PI-controller [Best93, p. 8].

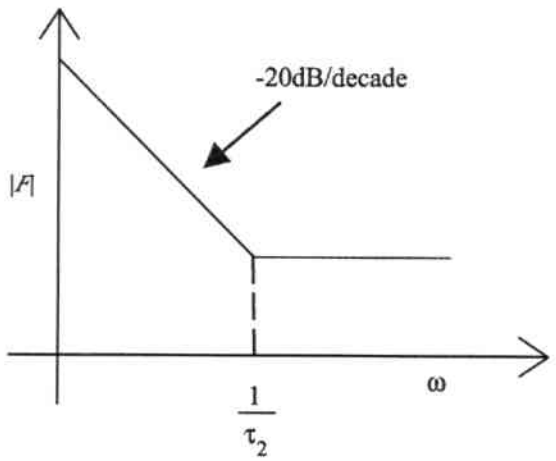


Figure 3.6: Frequency response of a PI controller (logarithmic axes). In theory, the filter gain is infinite for $\omega = 0$. Figure is based on [Best93, p. 9].

Figure 3.5 shows the components of an analog PI controller and fig. 3.6 its frequency response curve. Obviously, there are 3 design parameters in the circuit, namely the resistors R_1, R_2 and the capacitor C. Luckily, when the circuit is converted to an s-domain transfer function, these parameters can be combined to $\tau_1 = R_1 \cdot C$ and $\tau_2 = R_2 \cdot C$, which results in two degrees of freedom for the designer. The transfer function of a PI controller can be found for example in [Best93, p. 10] and it is:

$$F(s) = \frac{1 + \tau_2 s}{\tau_1 s} \tag{3.14}$$

There are several possible ways of converting a transfer function from s-domain to z-domain, like the impulse-invariant z-transform or the bilinear z-transform. While the impuls-invariant z-transform ensures that the impulse response of the resulting digital filter resembles the analog impulse response as much as possible, the bilinear z-transform allows the digital frequency response to match the analog frequency response better. The following substitution must be done in an s-domain transfer function to get the equivalent z-domain transfer function:

$$s = \frac{2}{T_s} \frac{1 - z^{-1}}{1 + z^{-1}} \tag{3.15}$$

With this substitution, the z-domain transfer function of a loop filter that has the form of a PI controller can be derived as follows:

$$\begin{aligned} F_{LF}[z] &= \frac{1+\tau_2\frac{2}{T_s}\frac{1-z^{-1}}{1+z^{-1}}}{\tau_1\frac{2}{T_s}\frac{1-z^{-1}}{1+z^{-1}}} \\ &= \frac{\frac{T_s}{2}+\tau_2\frac{1-z^{-1}}{1+z^{-1}}}{\tau_1\frac{1-z^{-1}}{1+z^{-1}}} \\ &= \frac{\frac{1}{1+z^{-1}}}{\frac{1}{1+z^{-1}}}\cdot\frac{\frac{T_s(1+z^{-1})}{2}+\tau_2(1-z^{-1})}{\tau_1(1-z^{-1})} \\ &= \frac{\frac{T_s}{2}+\frac{T_s}{2}z^{-1}+\tau_2-\tau_2 z^{-1}}{\tau_1(1-z^{-1})} \\ &= \frac{\frac{T_s}{2\tau_1}+\frac{\tau_2}{\tau_1}+\left(\frac{T_s}{2\tau_1}-\frac{\tau_2}{\tau_1}\right)z^{-1}}{1-z^{-1}} \end{aligned} \tag{3.16}$$

Unfortunately, transforming an analog filter to a digital one by means of a bilinear z-transform changes the corner frequencies of the resulting digital filter. The reason for this is, that the bilinear z-transform maps the whole (infinite) frequency range of the analog filter to the (digital) frequency range $[0, \frac{f_s}{2})$, where f_s denotes the sampling rate of the digital system. Low frequencies, which are far away from $\frac{f_s}{2}$, are mapped (more or less) "exactly", but the closer a frequency is to $\frac{f_s}{2}$, the bigger the mapping error gets.

If the corner frequencies of the resulting digital filter should match those of the corresponding analog filter, a procedure called "prewarping" has to be done before the filter is transformed by the bilinear z-transform. To prewarp a given filter, all of its corner frequencies ω_i must be transformed by eq. 3.17 [Best93, p. 348]. This moves all corner frequencies in such a way, that the corner frequencies of the resulting digital filter equal their analog counterparts.

$$\omega_{i,\text{prewarped}} = \frac{2}{T_s}\tan\left(\frac{\omega_i T_s}{2}\right) \tag{3.17}$$

From fig. 3.6, it can be seen that the only corner frequency of a PI controller is determined by $\frac{1}{\tau_2}$. Using eq. 3.17 yields:

$$\begin{aligned} \frac{1}{\tau_{2,\text{prewarped}}} &= \frac{2}{T_s}\tan\left(\frac{1}{\tau_2}\cdot\frac{T_s}{2}\right) \\ \Rightarrow \qquad \tau_{2,\text{prewarped}} &= \frac{T_s}{2\tan\left(\frac{T_s}{2\tau_2}\right)} \end{aligned} \tag{3.18}$$

Until now, the transfer functions of all individual components of a PLL have been examined. These transfer functions will be combined in the next section to get the transfer function of a PLL as a whole. After that, in chapter 3.4, a set of equations will be presented which can be used to specify the loop filter parameters τ_1 and τ_2.

3.3 Transfer Function and Stability Analysis

To create the z-domain transfer function of a PLL as a whole, it is almost sufficient to simply combine the transfer functions of the phase detector, the loop filter and the VCO like the model of fig. 3.7 suggests. However, one problem is arising from fig. 3.7: for any given sample index t, $\phi_2[t]$ depends on $\phi_e[t]$, which, in turn, depends on $\phi_2[t]$. To deal with this problem, a one-sample delay should be inserted in the feedback loop, so that $\phi_2[t]$ does not depend on itself, but on $\phi_2[t-1]$.

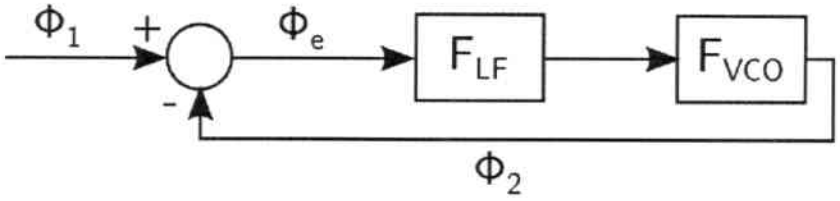


Figure 3.7: Z-domain block diagram of a PLL.

Luckily, it can be seen from fig. 3.3, that there is already a delay in the forward looking part of the Forward Euler integrator in the VCO, so $\phi_2[t]$ actually depends on $\phi_e[t-1]$. In a Backward Euler integrator, the delay would be part of the VCO's backward looking signal path. So, with a Backward Euler integrator, an additional delay block would be needed somewhere in the loop. This would effectively turn the Backward Euler integrator into a Forward Euler integrator, as it is pointed out in [Stephens02, p. 216/217].

Including an additional delay would actually not be any problem at all, so why is this issue mentioned here in detail? At first, the simulations for this work have been done with a Simulink NCO block. By looking at its help page in Simulink, it can be seen that it also features a delay in the forward looking signal path, which is why it can be considered a Forward Euler integrator. However, when a simulation was tried to be started, Simulink always reported an error occuring from an "algebraic loop". For some unknown reason this could be fixed by adding another delay block, but then again more delays as actually needed make the system slower as it could be. In the end, the NCO was replaced by a discrete-time VCO block, which provides the same (mathematical) functionality as the NCO, but does not require an additional delay.

With the help of fig. 3.7, the z-domain transfer function can be obtained easily:

$$\begin{aligned}
\Phi_2[z] &= F_{VCO}[z] \cdot F_{LF}[z] \cdot \Phi_e[z] \\
&= F_{VCO}[z] \cdot F_{LF}[z] \cdot (\Phi_1[z] - \Phi_2[z]) \\
&= F_{VCO}[z] \cdot F_{LF}[z] \cdot \Phi_1[z] - F_{VCO}[z] \cdot F_{LF}[z] \cdot \Phi_2[z] \\
\Rightarrow \quad & \Phi_2[z] + F_{VCO}[z] \cdot F_{LF}[z] \cdot \Phi_2[z] = F_{VCO}[z] \cdot F_{LF}[z] \cdot \Phi_1[z] \\
\Rightarrow \quad & \Phi_2[z] \cdot (1 + F_{VCO}[z] \cdot F_{LF}[z]) = F_{VCO}[z] \cdot F_{LF}[z] \cdot \Phi_1[z]
\end{aligned}$$

In the standard form $F[z] = \frac{Output[z]}{Input[z]}$, this looks like:

$$F_{PLL}[z] := \frac{\Phi_2[z]}{\Phi_1[z]} = \frac{F_{VCO}[z] \cdot F_{LF}[z]}{1 + F_{VCO}[z] \cdot F_{LF}[z]} \tag{3.19}$$

Now, $F_{VCO}[z]$ and $F_{LF}[z]$ can be substituted by eqs. 3.13 and 3.16, respectively:

$$\begin{aligned}
F_{PLL}[z] &= \frac{\frac{2\pi\kappa T_s \cdot z^{-1}}{1-z^{-1}} \cdot \frac{\frac{T_s}{2\tau_1}+\frac{\tau_2}{\tau_1}+\left(\frac{T_s}{2\tau_1}-\frac{\tau_2}{\tau_1}\right)z^{-1}}{1-z^{-1}}}{1+\frac{2\pi\kappa T_s \cdot z^{-1}}{1-z^{-1}} \cdot \frac{\frac{T_s}{2\tau_1}+\frac{\tau_2}{\tau_1}+\left(\frac{T_s}{2\tau_1}-\frac{\tau_2}{\tau_1}\right)z^{-1}}{1-z^{-1}}} \\
&= \frac{\cancel{\frac{1}{(1-z^{-1})^2}}}{\cancel{\frac{1}{(1-z^{-1})^2}}} \cdot \frac{2\pi\kappa T_s \cdot \left(\frac{T_s}{2\tau_1}+\frac{\tau_2}{\tau_1}+\left(\frac{T_s}{2\tau_1}-\frac{\tau_2}{\tau_1}\right)z^{-1}\right)z^{-1}}{(1-z^{-1})^2+2\pi\kappa T_s \cdot \left(\frac{T_s}{2\tau_1}+\frac{\tau_2}{\tau_1}+\left(\frac{T_s}{2\tau_1}-\frac{\tau_2}{\tau_1}\right)z^{-1}\right)z^{-1}} \\
&= \frac{2\pi\kappa T_s \cdot \left(\left(\frac{T_s}{2\tau_1}+\frac{\tau_2}{\tau_1}\right)z^{-1}+\left(\frac{T_s}{2\tau_1}-\frac{\tau_2}{\tau_1}\right)z^{-2}\right)}{1-2z^{-1}+z^{-2}+2\pi\kappa T_s \cdot \left(\left(\frac{T_s}{2\tau_1}+\frac{\tau_2}{\tau_1}\right)z^{-1}+\left(\frac{T_s}{2\tau_1}-\frac{\tau_2}{\tau_1}\right)z^{-2}\right)} \\
&= \frac{(\pi\kappa T_s^2+2\pi\kappa\tau_2 T_s)\,z^{-1}+(\pi\kappa T_s^2-2\pi\kappa\tau_2 T_s)\,z^{-2}}{\tau_1+(\pi\kappa T_s^2+2\pi\kappa\tau_2 T_s-2\tau_1)\,z^{-1}+(\pi\kappa T_s^2-2\pi\kappa\tau_2 T_s+\tau_1)\,z^{-2}}
\end{aligned}$$

The terms in parenthesis can be be substituted now to make the function more handy. After that, the location of the poles can be determined by setting the denominator to zero and using the quadratic formula.

$$a := \pi\kappa T_s^2 \tag{3.20}$$

$$b := 2\pi\kappa\tau_2 T_s \tag{3.21}$$

$$\begin{aligned}
0 &\stackrel{!}{=} \tau_1+(a+b-2\tau_1)z^{-1}+(a-b+\tau_1)z^{-2} \\
&= z^2+\left(\frac{a+b}{\tau_1}-2\right)z+\left(\frac{a-b}{\tau_1}+1\right) \\
\Rightarrow z_{1/2} &= 1-\frac{a+b}{2\tau_1} \pm \sqrt{\frac{(a+b)^2}{4\tau_1^2}-\frac{2a}{\tau_1}}
\end{aligned} \tag{3.22}$$

So, after choosing a sampling time T_s, a VCO gain κ and filter coefficients τ_1 and τ_2, their values can be used in equation 3.22 to determine the position of the transfer functions' poles. A digital control circuit is stable, if all the poles of its z-domain transfer function are within the unit circle. Thus, the PLL will be stable, if the absolute value of its poles is smaller than 1. At this point, the last remaining task in the implementation of a PLL is to specify its filter coefficients, which will be done in the next chapter.

3.4 Design Equations for 2nd order PLLs

In chapter 3.3, the transfer function for a PLL was established in the z-domain. When the transfer function is calculated in the Laplace-domain, it looks like this:

$$F_{PLL}(s) = \frac{\frac{2\pi\kappa\tau_2}{\tau_1}s+\frac{2\pi\kappa}{\tau_1}}{s^2+\frac{2\pi\kappa\tau_2}{\tau_1}s+\frac{2\pi\kappa}{\tau_1}}$$

The denominator of this transfer function can be converted to the standard form of $s^2 + 2\zeta\omega_n s + \omega_n^2$ by the substitutions 3.23 [Best93, p. 18]. ω_n is called the natural frequency and ζ is called the damping factor.

$$\omega_n = \sqrt{\frac{2\pi\kappa}{\tau_1}} \qquad \zeta = \frac{\omega_n \tau_2}{2} \tag{3.23}$$

$$\Rightarrow \qquad \tau_1 = \frac{2\pi\kappa}{\omega_n^2} \qquad \tau_2 = \frac{2\zeta}{\omega_n} \tag{3.24}$$

In the following paragraphs, equations from [Best93] will be presented, which can be used to specify ω_n and ζ. These, in turn, can then be used to calculate the filter coefficients. Obviously, ω_n and ζ are defined based on the analog transfer function and so will be the equations used to actually calculate these parameters. Estimations obtained from APLLs are not directly applicable for digital PLLs, but this problem will be dealt with in section 3.4.3. There, it can be seen that an SPLL behaves like the corresponding APLL if its sampling frequency is high enough. Unfortunately, the sampling frequency must be a lot higher than the "usual" value of two times the maximum frequency occuring the system.

At first, the parameter ζ will be chosen and according to [Best93, p. 19-23], it is best to select $\zeta = \frac{\sqrt{2}}{2} \approx 0.707$. For higher values of ζ the damping is too high and the PLL reacts too slowly. For smaller values of ζ the loop is not damped enough, so it reacts too fast and the PLL is affected too much by noise and other disturbances.

Now, as ζ has been chosen, ω_n is the only free parameter left. There are several possibilities to specify ω_n. An overview of all APLL design equations can be found in [Best93, p. 58/59] and three of them will be presented in the following sections.

3.4.1 Design by Pull-Out Range

If a PLL is locked onto a certain frequency, the pull-out range $\Delta\omega_{PO}(= 2\pi\Delta f_{PO})$ is defined by [Best93, p. 42/43] as "that frequency step, which causes a lock-out if applied to the reference input of the PLL". On the same page in that book, the pull-out range is approximated by

$$\Delta\omega_{PO} \approx 1.8\omega_n(\zeta + 1) \qquad \Rightarrow \qquad \omega_n \approx \frac{\pi \cdot \Delta f_{PO}}{0.9(\zeta + 1)} \tag{3.25}$$

So, by specifying Δf_{PO}, ω_n can easily be obtained and with that, eq. 3.24 can be used to specify the filter parameters. However, because the frequency transitions in human speech are mostly continuous, it is hard to actually define Δf_{PO}. This is the reason why the filter parameters have not been defined by the PLL's pull-out range in this work.

3.4.2 Design by Lock Range

The lock range $\Delta\omega_L(= 2\pi\Delta f_L)$ of a PLL is defined as follows: if the local oscillator currently produces a sine wave at a frequency of ω_0 rad/s (this can be its quiescent frequency, but is does not need to be) and the target signal has a frequency of $\omega_1 = \omega_0 + \Delta\omega_L$ rad/s, what can the maximum value of $\Delta\omega_L$ be, so that the PLL

locks within one beat-note between ω_0 and ω_1? [Best93, p. 38] gives the value for the lock range as follows:

$$\Delta\omega_L \approx 2\zeta\omega_n \quad \Rightarrow \quad \omega_n \approx \frac{\pi \cdot \Delta f_L}{\zeta} \tag{3.26}$$

Eq. 3.26 is useful especially for the design of a multi-PLL system. As the voiced parts of human speech are made up of a fundamental frequency f and its harmonics $2f$, $3f$, etc., multiple PLLs are needed to track all of the frequencies. To accomplish this task, the target signal is split into many frequency bands using band-pass filters. Because the bandwidth of the band-pass filters is known, equation 3.26 can be used to build a PLL which is able to lock onto any frequency in its frequency band within one beat-note.

3.4.3 Design by Noise Bandwidth

The target signal for a PLL is ideally a perfect sine wave at only one certain frequency. However, if there are other frequencies present in the input signal, the PLL will still try to lock on the frequency component with the highest amplitude. In doing this, frequencies of lower amplitude are suppressed and as a result, the VCO output signal has an increased signal-to-noise ratio compared to the PLL input.

When noise is added to the target signal, the zero crossings of the target sine function will be displaced forward or backward in time [Best93, p. 47]. This means that noise of a certain bandwidth induces phase jitter, which will be reduced by the PLL. According to [Best93, p. 49], the following equation holds:

$$(SNR)_L = (SNR)_i \cdot \frac{B_i}{2B_L} \tag{3.27}$$

In this expression, $(SNR)_i$ and $(SNR)_L$ denote the signal-to-noise ratio at the input of the PLL and at the VCO output, respectively. These values are calculated using the frequency noise and not the phase noise. Furthermore, the (double-sided) bandwidth of the noise at the input is supposed to be B_i and B_L is called the loop bandwidth or noise bandwidth. As an example, if a frequency of 1000Hz is to be tracked by a PLL, but there are further, equally distributed, frequency components from 950Hz to 1050Hz in the target signal, B_i would be 1050Hz − 950Hz = 100Hz. Dividing the signal power of the 1000Hz component by the rest of the power within the range from 950Hz to 1050Hz yields $(SNR)_i$. Concerning $(SNR)_L$, experience has shown that it has to be at least 4 (= 6dB), otherwise the PLL will not be working stably [Best93, p. 50].

From the variables of eq. 3.27, $(SNR)_i$ can be measured or estimated (at least in the controlled environment established for the tests in this work), $(SNR)_L$ must be greater than 6dB and B_i can be limited by filtering the PLL input with a band-pass filter. With these values, B_L can be calculated by solving eq. 3.27 for it:

$$B_L = \frac{(SNR)_i}{(SNR)_L} \cdot \frac{B_i}{2} \quad \overset{(SNR)_L > 6dB}{\Rightarrow} \quad B_L < \frac{(SNR)_i \cdot B_i}{12} \tag{3.28}$$

After that, B_L can be used to calculate ω_n using the equation from [Best93, p. 48]:

$$B_L = \frac{\omega_n}{2}\left(\zeta + \frac{1}{4\zeta}\right) \quad \Rightarrow \quad \omega_n = \frac{2B_L}{\zeta + \frac{1}{4\zeta}} \tag{3.29}$$

In the beginning of ch. 3.4, it was mentioned that design equations for analog PLLs can normally not be used for digital PLLs directly. A description and a graph can be found in [Stephens02, p. 243/244], which explain how much the noise bandwidth of a digital loop differs from its analog counterpart given a specific sampling rate. It is suggested that the sampling rate is at least seven times the analog loop bandwidth. Luckily, the loop bandwidths of the PLLs presented in this work are no higher than 2kHz, so a sampling rate of 48kHz is easily high enough to fulfill this criterion.

3.5 Amplitude Estimator

From the previously presented components of a PLL, none keeps track of the amplitude of the target signal. However, it is necessary to have an estimation for the amplitude in order to implement the dereverberation method presented in chapter 2.3. For this reason the amplitude estimation circuit from [Karimi-Ghartemani01] was used in this work. Fig. 3.8 is a schematic drawing of the circuit:

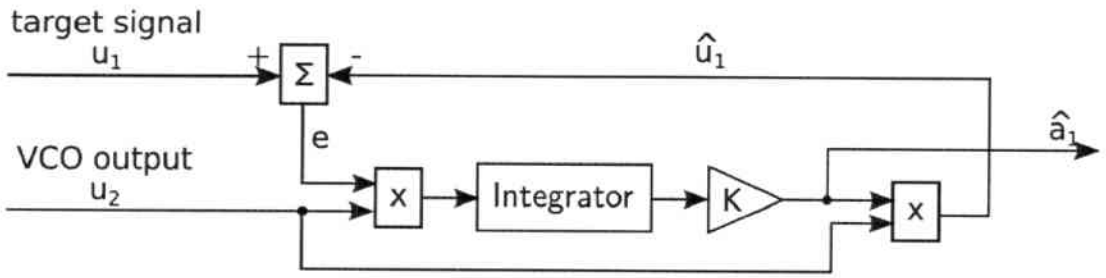


Figure 3.8: Amplitude estimation circuit

In [Karimi-Ghartemani01], the amplitude estimation is used in a combined amplitude/phase detector for a new type of PLL, but in this work only the amplitude estimation capability of the circuit is needed. A detailed mathematical derivation of the amplitude estimator will not be given here, but then again it is quite easy to understand how it works: The output of the VCO (u_2) is multiplied by the current estimate of the amplitude ($\hat{a_1}$) to produce a (possibly) correctly scaled replica of the target signal ($\hat{u_1}$). This replica is subtracted from the target signal in order to get the error, which is then accumulated in the integrator. If the current amplitude estimation is too low, the error will be big and the accumulator output will increase rapidly, which also increases the estimate of the amplitude. The design is rather simple and the only design parameter is the integrator gain k. The higher k, the faster the amplitude estimator will react to amplitude changes, but the more it will also be disturbed by noise. For this work, a gain of $k = 5000$ has proven to be a reasonable value, although no mathematical optimization has been carried out, like for instance minimizing the mean square error.

The original design presented in [Karimi-Ghartemani01] includes a 90° phase-shift of the VCO output u_2, which is needed because the target signal is modelled as a cosine-function in [Karimi-Ghartemani01], whereas a sine-function is used to calculate the

VCO output. This difference in the model introduces a 90° phase-shift between u_1 and u_2, which is not needed in this work because all signals are modelled as sine-functions.

A disadvantage of this amplitude detection circuit is the fact that it only works when the PLL is already phase-locked onto the target signal. If u_1 and $\hat{u}_1$ are not in phase, the difference e might be large not only because of different amplitudes of u_1 and $\hat{u}_1$ but also because of the phase shift between the signals. As a result, the time it takes to aquire a good estimate for the ampliude is determined by both the integrator gain k of the amplitude estimator and the lock-in time of the PLL itself. This is a problem especially at the beginning of the target signal: In "idle" mode, when there is no dominant frequency in u_1 (i.e. the target signal contains unvoiced speech or just noise), the PLL is not locked onto any certain frequency. When a voiced speech segment starts afterwards, the PLL must first lock the phase (and frequency) of a sinusoid before the amplitude estimation can get more precise. Unfortunately, the dereverberation method presented in this work relies on a good estimate of the amplitude of the direct sound, which must be approximately correct before the first reflected sound wave arrives at the microphone. This leaves only a very short timespan of typically one to three milliseconds to measure the amplitude of the direct sound (if the reflection from the floor of the room is considered the first reflection).

One to three milliseconds turned out to be too short for the experiments in this work, which is why the amplitude estimation has been carried out by processing the microphone signal backwards in time. Instead of tracking the time between the start of the direct sound and the start of the first reflection, the time between the discontinuation of the first reflection and the discontinuation of the direct sound is tracked in the reverse signal. The discontinuation of the first reflection changes the phase and amplitude of the total signal (according to equations 2.2 and 2.1), but not its frequency. As a result, the PLL must overcome only the changes of phase and amplitude in the reversed signal compared to changes of frequency, phase and amplitude when processing the signal in the "right" direction. The results of the experiments for this work showed that the amplitude of the direct sound can be estimated precisely enough using the backwards-processing technique.

3.6 Summary

In chapter 3, it has been explained what a phase-locked loop is, which components it is made up of and that it can be built using software, hardware, or a mixture of both. The components of a PLL are the phase detector, the loop filter and the local oscillator. Each of these components has been analyzed and a z-domain transfer function has been presented. After the definition of the order of a PLL, the transfer functions have been combined to form the transfer function of a PLL as a whole, which, in turn, is useful to analyze the dynamic stability of the PLL circuit. In addition to the analysis of a PLL, equations have been presented to calculate all necessary parameters based on the desired performance of the PLL. Finally, a short introduction to the EPLL amplitude estimator was given.

4. Implementation

In this chapter, a more detailed view of the actual implementation of the dereverberation algorithm is presented. The chapter is divided into two sections, the first one describing the implementation of the PLL which is needed to measure the parameters of the recorded signal (frequency, amplitude, phase), and the second one describing the actual dereverberation method. The following chapter covers the PLL implementation and it will focus on implementation details instead of PLL theory, as this was already presented in the previous chapter.

4.1 Matlab/Simulink PLL Model

The implementation of the PLL was done using a combination of Matlab and Simulink: the actual circuit was built as a Simulink model, whereas a Matlab script (further referred to as the initialization script) was used to load the microphone waveforms and to set parameters. When line numbers are given in this section, they always refer to the listing of the initialization script in appendix A.1.

Being a graphical programming language, one main advantage of Simulink is that the program is represented graphically and not as text, which is good when discussing it with other people. Additionally, the graphical representation also helps to keep an overview of the data flow. Furthermore, Simulink models (as the programs are called) can easily be vectorized to process multiple channels of data at once. Single-channel data, like the waveform-output of a microphone, is represented using a vector of length t (containing t samples of the signal) and multi-channel data is stored in a $n \times t$ matrix (containing n channels with t samples each). When multiple input channels are to be processed in Matlab or Simulink, it is often computationally faster to use a vectorized approach than to use a loop-structure that iterates over all of the channels.

Fig. 4.1 shows the Simulink PLL model, which is made up of multiple subsystems: the phase detector, the VCO, the amplitude estimator and a subsystem which loads the signals from the Matlab workspace where they were already setup by the initialization script. The spectrum visualization subsystem to the left of the model is for visualization purposes only. It plots lines in the power spectrum of the input signal

showing the current instantaneous frequency of each VCO channel. However, it is not used in this work, so it will not be discussed in detail here.

4.1.1 Bandpass Filterbank Design

In chapter 2 it was pointed out that all harmonics of a segment of voiced speech have to be considered for the dereverberation process, because changes of all frequency components indicate reflections whereas changes in only some of the frequency components indicate a change of the spoken phoneme. As a single PLL can only track a single sine-wave, multiple PLLs are needed to keep track of all harmonics. Therefore, a bandpass filterbank is implemented in the initialization script (line 129 and the following) to split the microphone signal into its main frequency components, one for each PLL.

To determine the width of a single filters' passband and transition band, speech recorded in an anechoic chamber was analyzed by hand. Among these recordings of a male adult speaker, the fundamental frequency in voiced parts was usually in a range between 100Hz and 150Hz, so the distance between successive harmonics is at least 100Hz. This means that the stopband of each bandpass filter must begin at most 100Hz below and above the filters' passband center frequency, respectively. As a result, the passband bandwidth was set to 100Hz (line 47) and the width of the transition band adjacent to the passband was set to 50Hz on each side of the passband (line 48).

With these specifications the filters could be implemented using either a FIR or an IIR design. The advantage of FIR filters over IIR filters is that their group delay is constant, so all frequency components are delayed equally. For the dereverberation approach presented in this work this is a crucial property because if the group delay was not constant, the frequency components would be shifted with respect to each other. This, in turn, would make it impossible to compare the times when a reflected wavefront is detected at various frequencies. The disadvantage of FIR filters is the fact that higher filter orders are needed to meet the specifications (passband bandwidth, transition bandwidth) compared to IIR filters.

To solve this problem, IIR filters were implemented in combination with zero-phase filtering (lines 146-149). A mathematically precise description of zero-phase filtering can be found in [Smith07] but it basically involves filtering the signal twice: one time forward in time and another time backward in time. The result of this is that there is no time-delay between the original signal and the double-filtered signal (because the group delay of a single filter is applied both forward and backward, resulting in a total group delay of zero).

For the actual filter design, Chebyshev filters were used because of the steep transition between their pass- and stopband compared to other filter types of the same order. More precisely, the filters were designed as type II (inverse) Chebyshev filters because they exhibit stopband ripple instead of passband ripple. Stopband ripple was considered less important because the stopband contains only irrelevant information for a single PLL anyway.

Besides the elimination of the group delay, zero-phase filtering has another effect: the total magnitude transfer function of the zero-phase filtering equals the square of the transfer function of a single filter. In the passband, where the transfer function

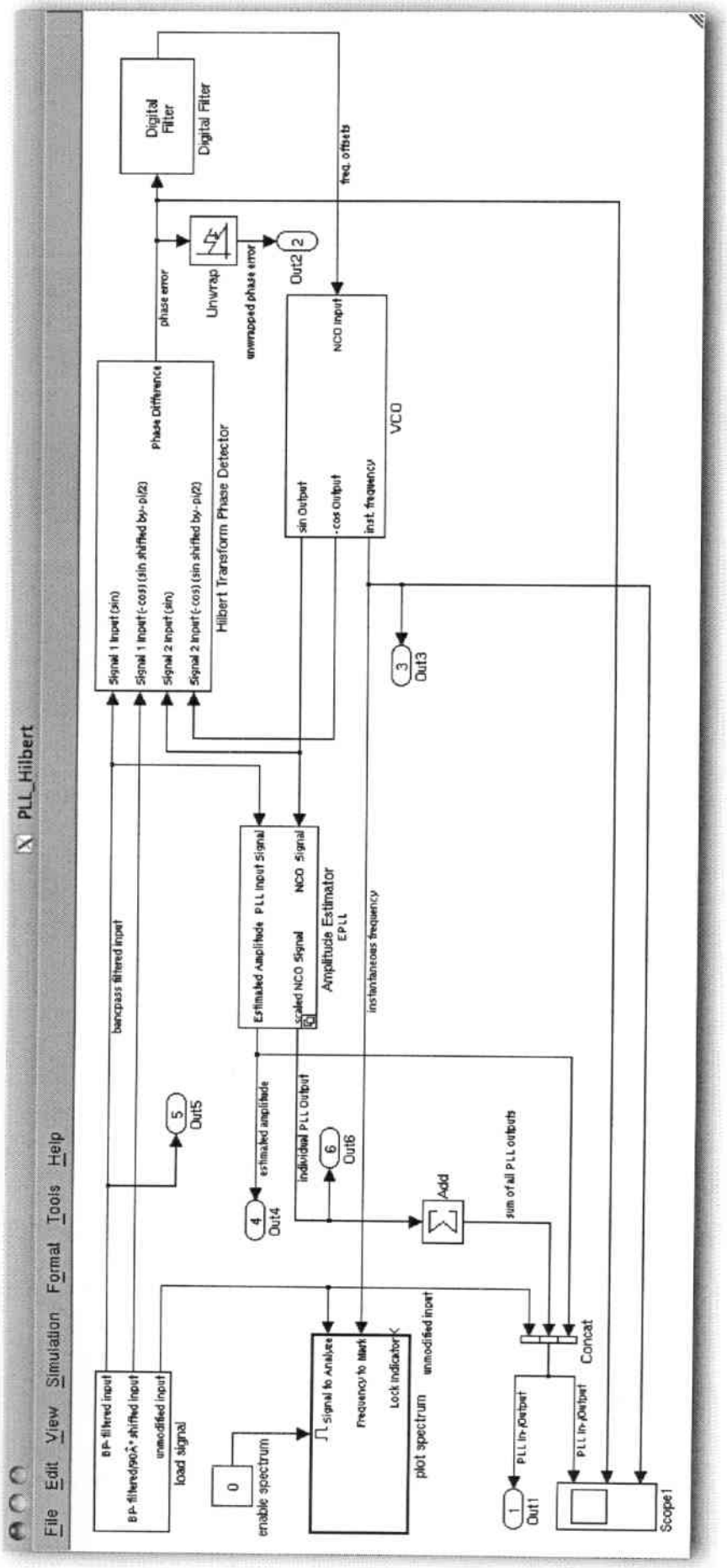


Figure 4.1: PLL modeled in Simulink

of the type II Chebyshev filters is approximately one, this has nearly no effect. In the stopband, however, where the transfer function is lower than 1, squaring the transfer function reduces it even more. This results in an additional attenuation in the stopband.

4.1.2 Hilbert Transform Phase Detector

After the bandpass filterbank each bandpass-filtered signal must also be Hilbert-transformed to produce the 90° phase-shifted versions for the Hilbert-Transform phase detector. A perfect Hilbert transformer cannot be implemented, as it would have an acausal impulse response. Thus, the transfer function is approximated by a high-order ($n = 1200$) FIR filter in lines 167 and 168, which works like a Hilbert-filter within a certain passband. For this work, the filter was designed to act like a Hilbert transformer for frequencies from 100Hz to 23900Hz ($= f_s/2 - 100\text{Hz}$). After the filtering, the new signals must be shifted back in time in order to compensate for the group delay of the Hilbert filter. The group delay of a Hilbert-Transform FIR filter of order n is always $\frac{n}{2}$ samples, which is why a shift of 600 samples was implemented in line 179.

In the end, two matrices are produced from all these signals. The first one is named "wavesignal" and it contains both the originally recorded signal and all bandpass filtered versions of it. The second one is named "wavesignal_hilbert" and it contains all the Hilbert-transformed versions of the first matrix. These two matrices are loaded into the Simulink model in the "load signals" subsystem in the top left corner of the model.

The bandpass- and Hilbert-filtered versions of the recorded signal are then processed in the phase detector, which is depicted in fig. 4.2. It is simply a Simulink representation of eqn. 3.9. Simulink automatically detects the number of channels, i.e. bands of the bandpass filterbank, and simulates an equal amount of individual phase detectors (and PLLs as a whole), as it was explained in section 4.1.

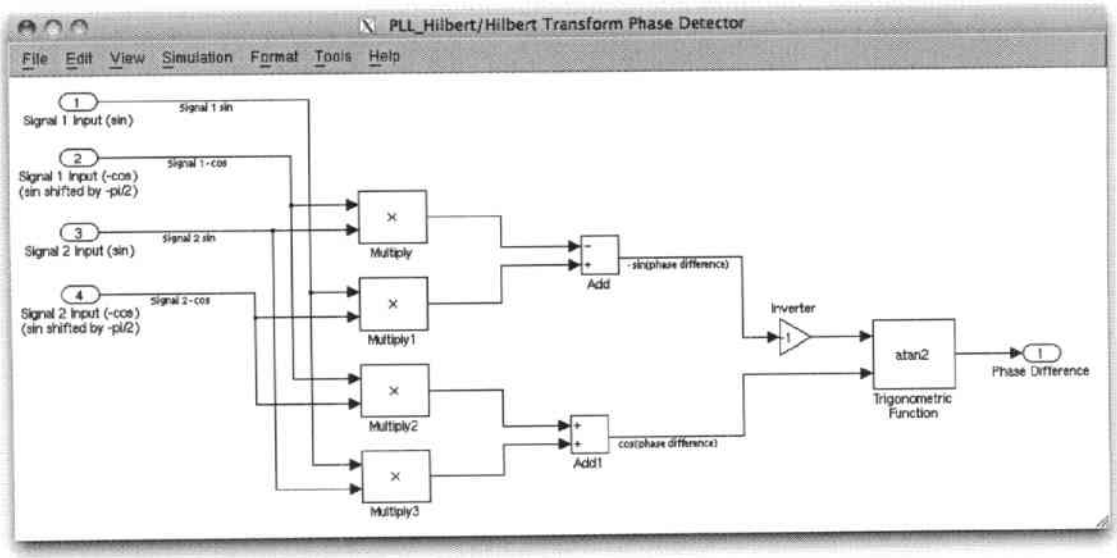


Figure 4.2: Hilbert Transform Phase Detector modeled in Simulink

4.1.3 Loop Filter

In order to design the loop filter, the noise bandwidth approach presented in section 3.4.3 was used for two reasons.

The first reason for choosing the design by noise bandwidth was that all variables needed to calculate the filter coefficients could be calculated deterministically instead of just arbitrarily setting a certain lock range or pull-out range. For the design by noise bandwidth, only the bandwidth of the target signal B_i and its signal-to-noise ratio $(SNR)_i$ must be known. The bandwidth of the target signal is limited by the bandpass filterbank, so an estimate for B_i was already at hand. $(SNR)_i$ was also easy to determine because it could simply be calculated from the recordings: All recorded files contain a short period of silence, i.e. noise, at the beginning, before the actual sound starts. So, k samples of noise (s_{noise}) could be taken from the beginning of each recording, whereas l samples of the actual signal plus the noise ($s_{signal+noise}$) could be taken from the rest of each recording. The SNR for a single file could then be easily calculated using the following equation:

$$SNR = 20 \cdot \log_{10} \left(\frac{\sqrt{\frac{1}{l} \sum_l s^2_{signal+noise}} - \sqrt{\frac{1}{k} \sum_k s^2_{noise}}}{\sqrt{\frac{1}{k} \sum_k s^2_{noise}}} \right)$$

This calculation was done for each recording and then the mean was calculated for all recordings of the same recording distance. The results can be seen in table 4.1 alongside with the corresponding z-domain filter coefficients $\tau 1$ and $\tau 2$.

distance [m]	$(SNR)_i$ [dB]	τ_1	τ_2
0.1	46.4	1.1819e-05	0.0019
0.6	34.3	2.1630e-05	0.0026
1.2	29.1	3.0050e-05	0.0031
1.8	21.5	5.5050e-05	0.0042

Table 4.1: Mean signal-to-noise ratio and filter coefficients for different recording distances

In the initialization script, the filter coefficients are first calculated in the Laplace-domain (lines 195 to 199) before they are converted to the Z-domain by a bilinear transformation (line 213). Additionally, the corner frequency of the filter is pre-warped between these two steps to keep the corner frequency of the digital filter at the same frequency as for the analog filter.

The second reason for choosing the noise bandwidth approach is that the bandwidth of the bandpass filters B_i can be found directly in equation 3.28, so the filter is automatically adjusted depending on the bandwidth of the bandpass filters. In the design by lock range or pull-out range, the width of the passband must be taken into account manually when specifying the lock range or the pull-out range, respectively.

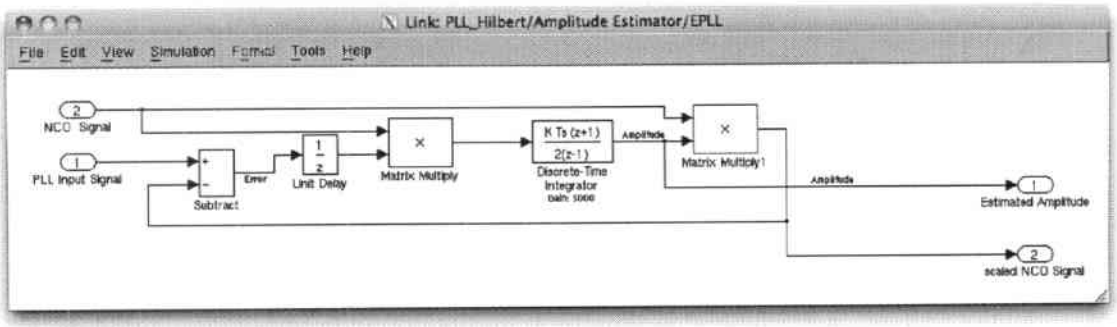


Figure 4.3: Amplitude estimator modeled in Simulink

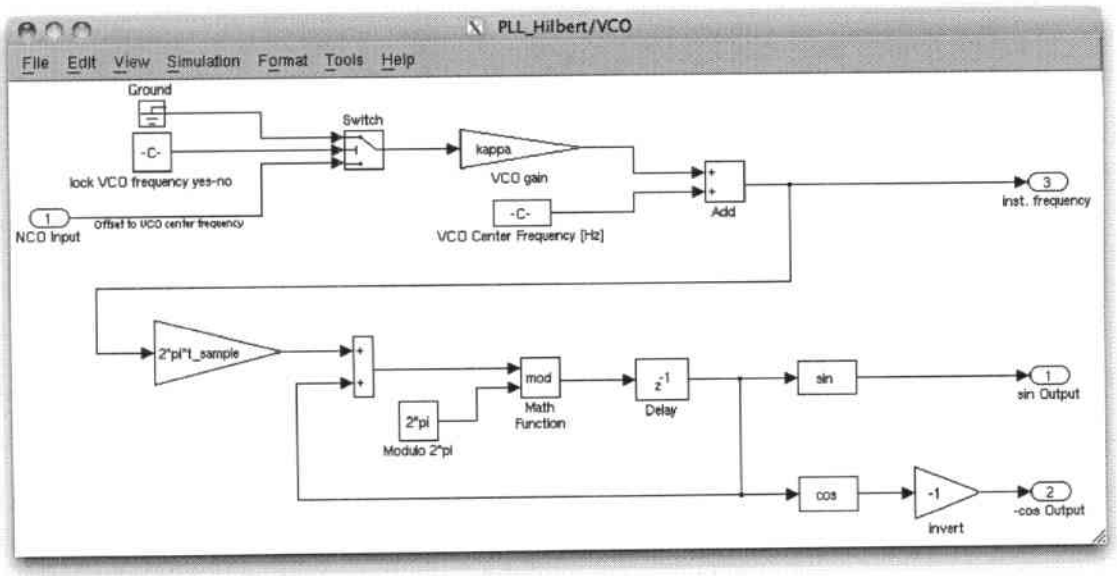


Figure 4.4: VCO modeled in Simulink

4.1.4 Amplitude Estimator and VCO

The amplitude estimator and the VCO are quite easy to understand as they are simply modeled after fig. 3.8 and fig. 3.3, respectively.

For a single-channel PLL, the VCO could simply be modeled using the "Discrete-Time VCO" block provided by Simulink. Unfortunately, that block does not support vectorized multi-channel data, probably because of the special blocks contained in it, which check the format of the input (for example if the input is a row or a column vector, or what its data type is) and make sure, that the output has the same format. In the original Simulink VCO block, this is needed to make it compatible with various input formats, but it seems to break the compatibility for vectorized models.

As a result, the contents of the Simulink "Discrete-Time VCO" block were rebuilt by hand without the unnecessary blocks, so the model supports vectorized data now. Furthermore, it can be seen that the custom system provides both a sine and cosine output for the Hilbert-Transform phase detector, unlike the original "Discrete-Time VCO" block, where it was needed to choose between a sine output and a cosine output.

Unfortunately, it is not possible to measure both the target signals' amplitude and phase offset in a a single PLL run. The reason for that is that the PLL continuously tries to eliminate any phase difference between the target signal and the VCO output. As a result, the output of the phase detector tends to converge to zero when the PLL has locked.

In order to measure both the phase and the amplitude of the target signal, the PLL must be run twice: At first, the target signal must be processed by the PLL in a normal way, so the amplitude and frequency can be tracked. To increase the accuracy of the amplitude estimation it has already been pointed out in section 3.5 that it's best to process the recording backwards and reverse the result after that to recreate its original order.

For the second run, the connection between the loop filter and the VCO needs to be cut, so the VCO cannot change its frequency. Additionally, the quiescent frequency must be set to the frequency of the target signal which was estimated in the first run. By doing this, the VCO already produces a sine wave at the same frequency as the target signal and according to eqn. 3.9, the output of the Hilbert-Transform phase detector will then be an estimate of the phase difference between both signals.

For the purpose of phase measurements there is a switch at the top left of the VCO subsystem, which allows to route a constant zero to the VCO instead of the loop filter output. By setting the corresponding variable "lock_nco = 1" in the startup script (line 69), the VCO frequency can thus be held constant at the quiescent frequency.

4.2 Dereverberation Script

In appendix A.2 there is a script which performs the actual dereverberation once the target signal's amplitude, frequency and phase offset have been measured by the PLL.

At first, both the amplitude and phase measurements from the two PLL runs are loaded and smoothed using a median filter with a length of 48 samples, which corresponds to a window of 1ms at 48kHz sampling rate (lines 36 to 38). A median filter was chosen instead of a moving average because firstly the median-filter is less sensitive to outliers and secondly it has a smoothing effect while also preserving steep slopes in the signal. This feature is advantageous for the dereverberation approach presented in this work, because a rapidly changing amplitude and/or phase offset indicates an incoming reverberation. These precise moments should not be smoothed by a moving average, because then it's harder to detect them.

4.2.1 Estimation of the Reference Signal

For the new dereverberation method the amplitude and phase offset must be compared to the amplitude and phase of the direct sound at any given time by means of eqs. 2.1 and 2.2. As a result, the amplitude and phase of the direct sound must be estimated from the recorded signal. In this work, the algorithm will only be tested on direct sound consisting of single sine waves with constant frequency, amplitude and phase, so it is certain that all variations occuring in the recordings are due to reverberations and/or noise. In section 2.3 it is explained how direct sound consisting of multiple varying frequencies with varying amplitudes and phase offsets can be handled.

At first, the start and end of the direct sound have to be detected in the recording (lines 47 and 48). These timestamps will be called "startSample" and "stopSample", respectively. It is very important to estimate the "startSample" as exactly as possible (+/- 1ms), because the amplitude and phase offset of the direct sound can only be estimated precisely in the short timespan between the arrival of the direct sound and the arrival of the first reflection.

For the test signals used in this work it was sufficient to simply detect the highest (positive) derivative of the microphone amplitude for the location of the "startSample" and the lowest (negative) derivative for the location of the "stopSample". If the system is to be used in a real-world application, a more robust detection scheme is needed, though. It could be seen from the experiments that reflections arriving at the microphone can have roughly the same effect on the measured amplitude as the direct sound itself. This could lead to misdetection when incoming reverberations change the microphone amplitude more than the arrival of the direct sound, for example if the highest derivative of the amplitude would not be found at the beginning of the direct sound.

Knowing the starting and ending time of the direct sound, its frequency can be determined by calculating the mean of instantaneous frequency of the VCO between the "startSample" and the "stopSample" (line 68). To suppress outliers a 10%-trimmed mean is used instead of a pure mean.

To estimate the amplitude of the direct sound for the sinusoidal test signals, a window of 48 samples (=1ms) is cut out of the PLL amplitude measurement immediately following the "startSample". After that, the median of the amplitude is calculated within that window (see fig. 4.5). The reference phase is obtained in a similar way using the phase measurement of the PLL instead. These values will be further referred to as the reference amplitude and phase, respectively. However, the reference

phase, frequency and amplitude are only valid between the "startSample" and the "stopSample", where there is an actual direct sound. Before the "startSample" and after the "stopSample", the reference amplitude is zero, because there is no direct sound.

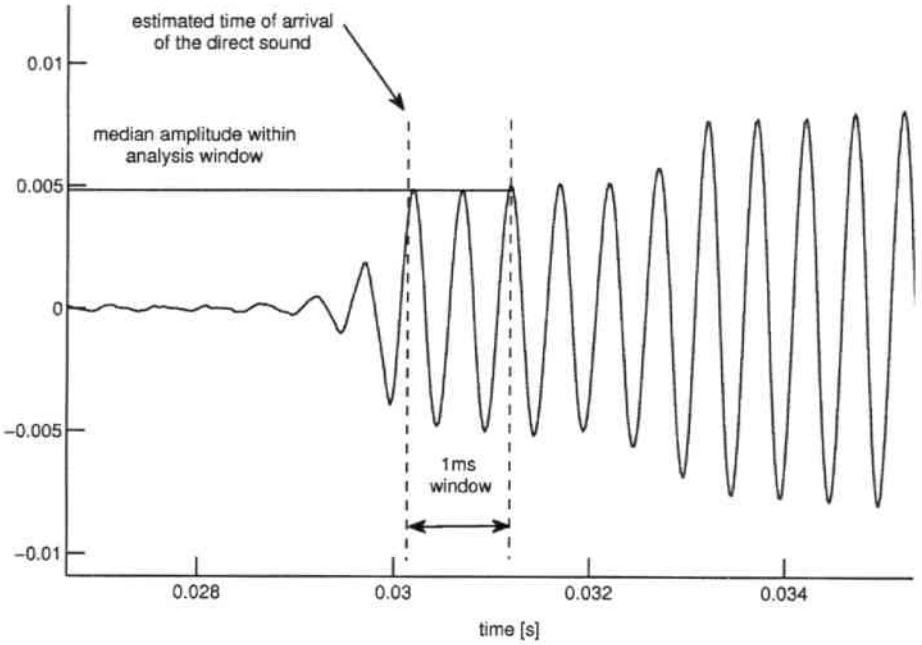


Figure 4.5: example for the identification of the amplitude of the direct sound

4.2.2 Calculation of Reverb Amplitude and Phase

After the reference amplitude, phase and frequency have been calculated, the rest of the algorithm is quite straightforward. At first, the amplitude and phase of the reverberation is calculated for each sample by using eqs. 2.3 and 2.4. To calculate the arctangent it is important to use the atan2 function provided by Matlab. In contrast to the normal atan $\left(\frac{y}{x}\right)$ function, atan2(y, x) analyzes the signs of y and x in order to return a value with a correct sign within the range of $[-\pi, \pi]$ instead of $[-\pi/2, \pi/2]$

In section 2.3 it is suggested to calculate the amplitude and phase of each single reflection. This means that for a given time t_1, the recording is composed of the direct sound and a sum of k reverberations. At time $t_2 > t_1$, another reflected wavefront arrives at the microphone, so the total signal is then made up of the direct sound and $k+1$ reflections. Because of this, the amplitude and phase offset of each recorded sample must be compared to the values after the last known reflection. If they are similar, it is assumed that no new reflection has arrived and if they differ, an additional reflection is likely to have arrived.

However, instead of comparing each sample to one from the last known reflection it is also possible to compare each sample straight to the direct sound. By this method, eqs. 2.4 and 2.3 are not used to calculate the amplitude and phase offset

for a single additional reflection but for the sum of all reflections. The equations can be rewritten as follows:

$$a_{SR} = \frac{a_C \cdot \cos(2\pi ft + \phi_C) - a_D \cdot \cos(2\pi ft + \phi_D)}{\cos(2\pi ft + \phi_{SR})}$$

$$\phi_{SR} = \arctan\left(\frac{a_C \cdot \sin(2\pi ft + \phi_C) - a_D \cdot \sin(2\pi ft + \phi_D)}{a_C \cdot \cos(2\pi ft + \phi_C) - a_D \cdot \cos(2\pi ft + \phi_D)}\right) - 2\pi ft$$

The subscript "D" means "direct sound", "C" means "current value" (for a given time) and "SR" stands for "sum of reflections". These equations can be simplified even further by omitting all "$2\pi ft$" terms, which is effectively the same as setting $t = 0$. In the end, however, it doesn't matter at what time two sine waves are compared as their amplitude and phase difference are the same at all times.

$$a_{SR} = \frac{a_C \cdot \cos(\phi_C) - a_D \cdot \cos(\phi_D)}{\cos(\phi_{SR})} \tag{4.1}$$

$$\phi_{SR} = \arctan\left(\frac{a_C \cdot \sin(\phi_C) - a_D \cdot \sin(\phi_D)}{a_C \cdot \cos(\phi_C) - a_D \cdot \cos(\phi_D)}\right) \tag{4.2}$$

In lines 99 to 106, eqs 4.1 and 4.2 are used on each recorded sample to calculate the amplitude and phase offset of the sum of all reflected sound waves at any given time (simply called reverb phase and amplitude from now on).

4.2.3 Detection of Reflection Times

When a reflected wavefront arrives at the microphone, it can be expected that the reverb amplitude and phase change rapidly, like it is descibed by equation. 2.2 and 2.1. In order to detect these moments in time (further referred to as reflection times), the reverb amplitude and phase is differentiated in lines 114 and 115, respectively. After that, both derivatives are normalized (lines 121 and 122) and smoothed (lines 125 and 126), before maxima are searched for in lines 129 and 130. The result of this are two vectors containing the reflection times. The first vector is solely calculated using the reverb amplitude and the other one is based only the reverb phase, which is why both vectors must be merged afterwards (line 132).

The Matlab function "findpeaks" has been used to detect peaks of the derivatives of the reverb amplitude and phase in combination with a "MINPEAKDISTANCE" parameter of 24. The result of this is that two consecutive peaks must be at least 24 samples (=0.5ms) apart if both of them are to be found. This, in turn, causes that two reflected sound waves cannot be distinguished if their times of arrival differ by less than 0.5ms. While a limit of 0.5ms is reasonably low for early reflections, it might be too high for late reflections, because there are so many late reflections and it is very likely that several of them arrive at the microphone within less than 0.5ms. While it is not possible to distinguish all of the late reflections, this has only a small effect on the dereverberation performance: When multiple reflections arrive at the microphone at roughly the same time the algorithm just combines them to their resultant and takes the resultant for the reverb which is to be cancelled. The experimental results in chapter 5 show that this simplification works.

Figure 4.6 shows an example of this detection scheme with the peaks (thick line) indicating the detected reflection times.

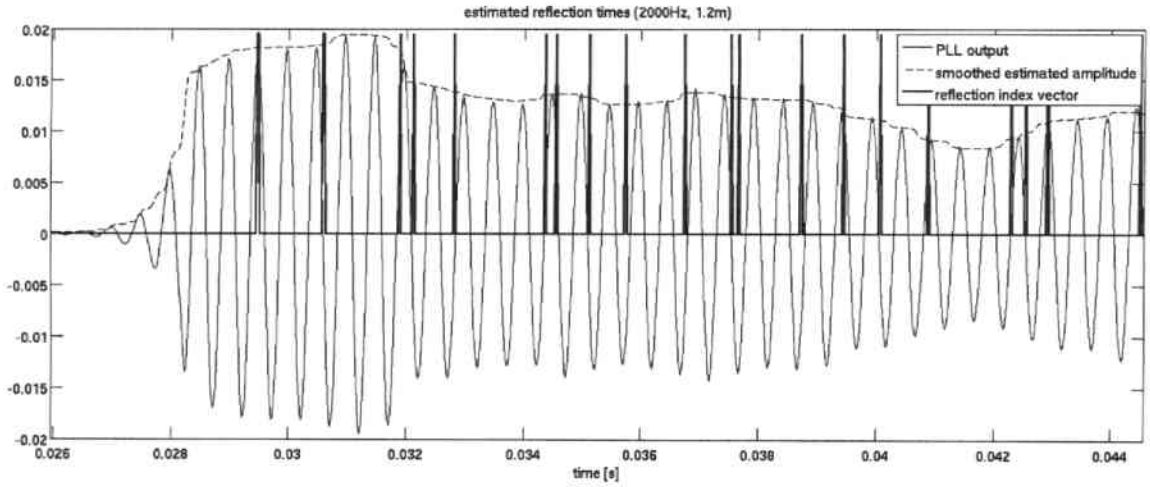


Figure 4.6: detected times of arriving reflections

4.2.4 Subtraction of Reverb

The actual subtraction of the reverberations from the microphone signal is done in lines 142 to 166. It is important to recall that the reverberation amplitude and phase remain constant in the time between the arrival of a reflected wavefront and the arrival of the following reflection. As a result, it is possible to calculate the median reverberation amplitude and phase offset between two consecutive reflection times (lines 154 and 155). With these values, a sine wave can be generated that should equal the reverberation wavefront if all amplitude, frequency and phase offset measurements were correct. After such a sine wave has been generated in lines 158 to 161 it is subtracted from the recording (line 156) to receive the dereverberated signal.

4.3 Summary

As a summary for this chapter the following list briefly describes the dereverberation algorithm:

1. Process recorded signal backwards in PLL to measure its frequency and amplitude.

2. Process recorded signal forwards in PLL with the VCO locked to the previously measured frequency to measure the phase offset.

3. Detect beginning and end of direct sound in the recording.

4. Estimate frequency, amplitude and phase offset of the direct sound during the period of time determined in step 3.

5. Calculate reverb amplitude and phase with respect to the estimated direct sound.

6. Detect rapid changes in reverb amplitude or phase, which characterize the arrival of a reflected sound wave.

7. Use values from step 5 to produce a signal which equals the sum of reverberations between two consecutively arriving reverberations.

8. Subtract the previously generated signal from the recording.

5. Evaluation

In order to evaluate the theoretical assumptions described in chapter 2 and to test the implementation of chapter 4, experiments have been carried out. The type and execution of the experiments is described in the following section, whereas the results are discussed afterwards.

5.1 Description of Evaluation Data and Recordings

For all experiments, specific signals were first generated by Matlab scripts. These files were then played via loudspeakers and recorded again. A sample rate of 48kHz was used for all processing steps, not only for the files themselves but also the simulation of the PLL in Simulink, for example.

The recordings were done in a normal living room, which was neither extremely reverberant nor acoustically damped, in order to reflect an environment in which automatic speech recognition might be used in the future, for example in voice controlled consumer electronic devices or domestic robots.

To test the performance of the new approach on various frequencies, .wav files were generated, each containing 1s of a single sine wave at one of the following frequencies: 150Hz, 200Hz, 250Hz, 300Hz, 350Hz, 400Hz, 450Hz, 500Hz, 600Hz, 800Hz, 1000Hz, 1050Hz, 1200Hz, 1500Hz, 2000Hz, 2400Hz, 3000Hz, 3500Hz and 4000Hz.

The amplitude, frequency and phase was constant (amplitude = 1, phase = 0) throughout each file to simulate single harmonic frequency components during a single voiced phoneme. Obviously, during a single phoneme it is impossible to have a phase step in the direct sound, because the glottis would have to oscillate infinitely fast for a short amount of time to produce such a phase step. As a result, the constant phase during each file is a reasonable approximation to real voiced speech.

The amplitude and frequency, however, are normally not fixed throughout a series of real phonemes, but due to lack of time recordings with time-varying frequency and amplitude have been carried out but not yet analyzed. In chapter 2.3 it is described how varying frequencies and amplitudes can be dealt with.

Furthermore, natural voiced speech contains not only a single frequency component (like the test files) but a set of harmonic frequencies. Each PLL can only measure the phase and amplitude of a single frequency component. For this reason, bandpass filters are introduced in chapter 4.1.2 to separate the frequency components. However, the bandpass filters also smooth the signal due to their prolonged impulse response, which makes it harder to detect rapid changes of the reverb amplitude and phase, i.e. reflection times. The detection scheme for the reflection times was presented in section 4.2.3, but it still has to be improved to work with the smoothed output of a bandpass filter. As a result, the bandpass filters were deactivated for the experiments with single sine waves. This can be done because there are no other dominant frequency components in the recordings which could disturb the tracking of the PLL.

To find out, how the dereverberation performance differs depending on the recording distance, all recordings were carried out at distances of 0.1m, 0.6m, 1.2m and 1.8m from the speakers to the microphone. However, only the speakers were moved during the experiments, whereas the microphone was left at the same position during all recordings. The recording volume was set to the maximum volume possible, while the playback volume was adjusted so that the sounds were about as loud as a normal conversation. After that, the playback and recording volume were kept constant for all four recording distances in order to not wrongly compensate for the decrease in loudness at increasing distances of the sound source. It cannot be expected that users of automatic speech recognition systems speak up as they move away from the microphone of the system.

The following hard- and software was used for the recordings:

Computer: Macbook Pro running Mac OS X 10.6.6

Playback and recording software: ecasound v2.7.2

Audio interface: Roland Edirol UA-25EX

Speakers: Heco Concerto B15, driven by a NAD C325 BEE stereo amplifier

Microphone: Sony ECM-T145 omnidirectional electret condenser lavalier microphone

The microphone was not especially chosen for any of its features, because the purpose of the work was to achieve dereverberation without using special calibrated hardware. In the making of this work, the signals were also recorded using a Behringer ECM-8000 measurement microphone. The only reason why this was done is because the dereverberation did not work out as intended and the microphone had to be eliminated as a possible source of inaccuracy. However, the original recordings were used again after the error had been found in the implementation.

In order to make sure that all recordings were done in the same way, a makefile was used to batch-process all files in combination with an ecasound script which carried out the actual recordings. The ecasound script starts a recording, plays one of the .wav files and stops the recording automatically after 1.5s. As a result, the recordings contain 1s of direct sound and 0.5s of decaying reflections after the direct

sound has ended. If the dereverberation is to work good, it must not only be able to cancel reflections which occur while the direct sound is still present, but it must also cancel any remaining reflections after the end of the direct sound.

5.2 Evaluation Results

As an index for the dereverberation performance of the new approach the direct-to-reverberant ratio (DRR) was used. This allows to compare the results with these from [Gaubitch05], where delay-and-sum beamformers were tested using the same measure. The DRR is simply the ratio of the power of the direct sound compared to the power of the reverberations. To calculate the DRR it is obviously necessary to know the direct sound, so it can be subtracted from the recorded sound to get the power of the reverberations.

In [Gaubitch05] artificial reverberations were added to the direct sound, so an exact calculation of the DRR was possible because the direct sound was known exactly. In this work, however, real signals were used which generates the problem that it is not possible to specify exactly what portion of a recording is direct sound and which part is made up of reverberations. Of course, the .wav file used for each recording is known, so the direct sound is also known to some extent. What is not known is the exact time alignment of the recording and the .wav file that was played.

The problem is that for each recording, the ecasound script starts the recording and then starts the playback afterwards. There is a slight time delay between these two actions so the direct sound can not be observed in the recording until about 30ms have passed. The exact delay varies depending on the recording distance and also the CPU load of the PC in the moment when the ecasound script was run. Under heavy load it must be expected that it took a little bit longer for the script to start the playback than under low CPU load. As a result of that, a method has been searched to find a good alignment of the direct sound and the corresponding recording.

To achieve this, the first 3ms of direct sound are cut out in each recording. The beginning of the direct sound must be estimated for the dereverberation anyway, so this information is already known. After that, the cross-correlation between one oscillation of the direct sound and the 3ms segment is calculated (line 73 in the dereverberation script in appendix A.2). The peak of this correlation gives the time delay at which the direct sound in the recording and the direct sound from the .wav file match best. Figure 5.1 gives an example of this computation:

After the direct sound has been identified in each recording by the method described above, the reverberation part of the original recording can be calculated by subtracting the direct sound from the recorded signal. When n denotes the amount of samples in a recording, the DRR is calculated as follows:

$$DRR = \frac{\sqrt{\frac{1}{n}\sum_{k=1..n} directSound[k]^2}}{\sqrt{\frac{1}{n}\sum_{k=1..n}(recordedSound[k] - directSound[k])^2}}$$

This gives the DRR of the original reverberated recording. Additionally, the DRR is calculated after the dereverberation has been performed in order to find out how

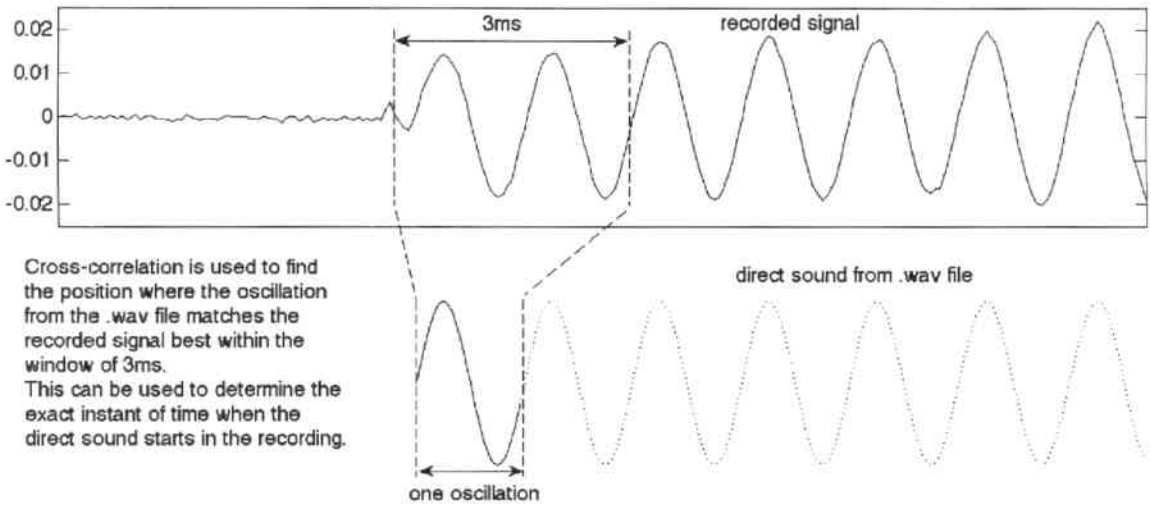


Figure 5.1: correlation-based identification of the direct sound's time delay in a recording

much the DRR is improved by the dereverberation. The following graphs show this improvement in dB for each recording distance and frequency.

At a recording distance of 10cm it can be seen that the DRR improvement is between 0dB and 3dB for most of the frequencies. For 1000Hz and 1050Hz the DRR is decreased very much by the dereverberation algorithm. The reason for that is that the phase of the direct sound is not correctly estimated at the beginning of the direct sound and as a result, the dereverberation algorithm uses a wrong target signal. The amplitude variations which occur in the recording and which are a caused by reflections are removed correctly, but the phase is slightly wrong. Figure 5.3 shows the problem: The direct sound is slighly shifted compared to the dereverberation output which leads to a low DRR. However, it is more important to remove the amplitude variations in the recording than to achieve a perfectly matching phase, which is why the result at 1000Hz and 1050Hz are actually not as bad as they seem to be.

Another result for the comparatively low performance of the dereverberation at a distance of 10cm is the fact that the DRR of the original recording is already quite high. The reason for this is the small distance which increases the amplitude of the direct sound compared to the amplitude of the reverberations. As a result, the achievable improvement is not as big as it is for higher distances.

Last but not least another explanation for the comparatively poor performance might be that the parameter tuning for the algorithm was done using the 2000Hz recording at a distance of 1.2m. Because of this, the parameters (length of smoothing windows, detection scheme for the beginning of the direct sound, etc.) might be not perfectly suited for lower recording distances.

At a recording distance of 0.6m (fig. 5.4) the DRR improvement is mostly over 4dB. In [Gaubitch05], a DRR improvement of around 4.5dB at a distance of 0.5m was achieved using a delay-and-sum beamformer with 5 microphones. For half of the frequencies the DRR improvement achieved with the new algorithm was even over 6dB, which is clearly more than the results from [Gaubitch05]. To be fair, it must be said however that real speech was used as a test signal in [Gaubitch05], not just

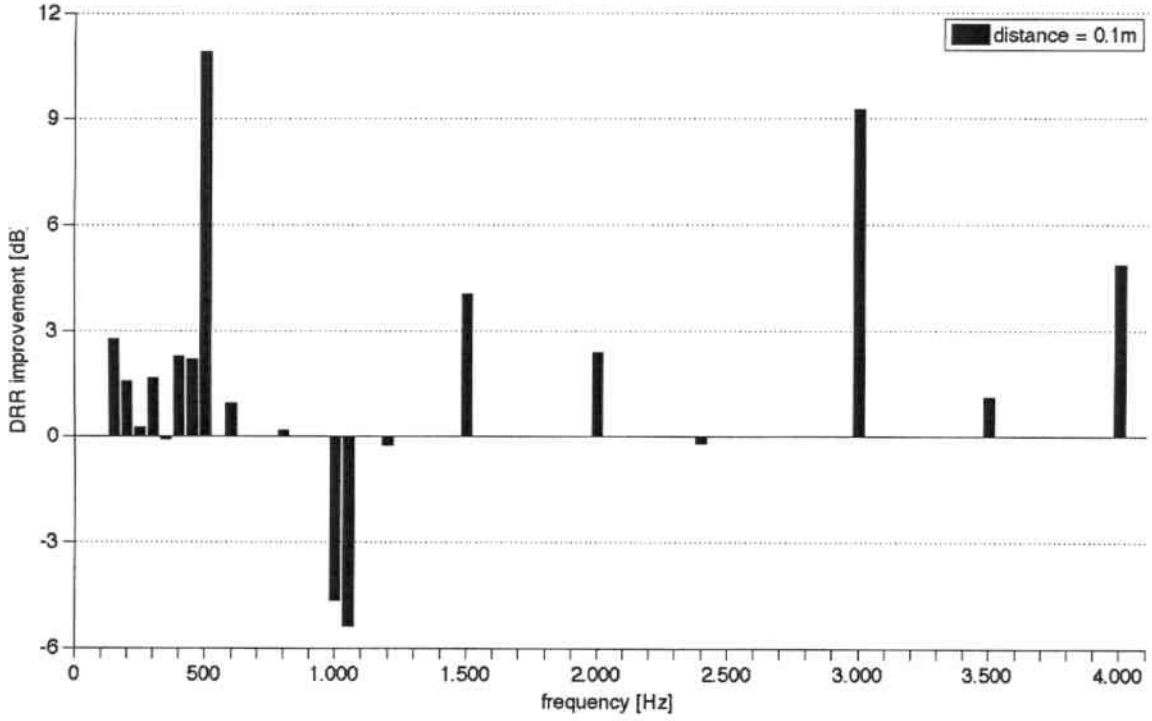


Figure 5.2: dereverberation performance at 0.1m recording distance

single sine waves. On the other hand, Gaubitch and Naylor used only simulated room impulse responses instead of real recordings with noise.

The outlier at 1000Hz can be explained simply by the fact that the frequency estimation obtained from the PLL is slightly lower than 1000Hz. Because of this, the calculated reverb is not correct and so is the output of the dereverberation after the reverb has been subtracted from the recorded sound. From fig. 5.5 it can be seen that the amplitude variations at the beginning of the direct sound are suppressed, but the amplitude of the dereverberation output increases with respect to time due to the wrongly measured frequency.

The results for recording distances of 1.2m and 1.8m were very good with a DRR improvement of more than 6dB for most of the frequencies. According to [Gaubitch05] a delay-and-sum beamformer with 4 to 5 microphones would be needed to achieve the same DRR improvement. If only a microphone array with two microphones was used, the expected DRR improvement is only about 3dB.

At a distance of 1.8m there is an outlier at 250Hz. Fig. 5.8 shows that the time-alignment of the direct sound as it is determined by the cross-correlation-method (described at the beginning of this section) is wrong. If it was right, the dashed line in figure 5.8 would be in phase with the beginning of the direct sound between 0.03s and 0.04s

5.3 Summary

Two main conclusions can be drawn from the experiments: The first one is of course the positive fact that the dereverberation actually works on single sine waves. Throughout the making of this work it was not always sure if the algorithm would ever work at all. The main concern was if the PLL can track the signals frequency,

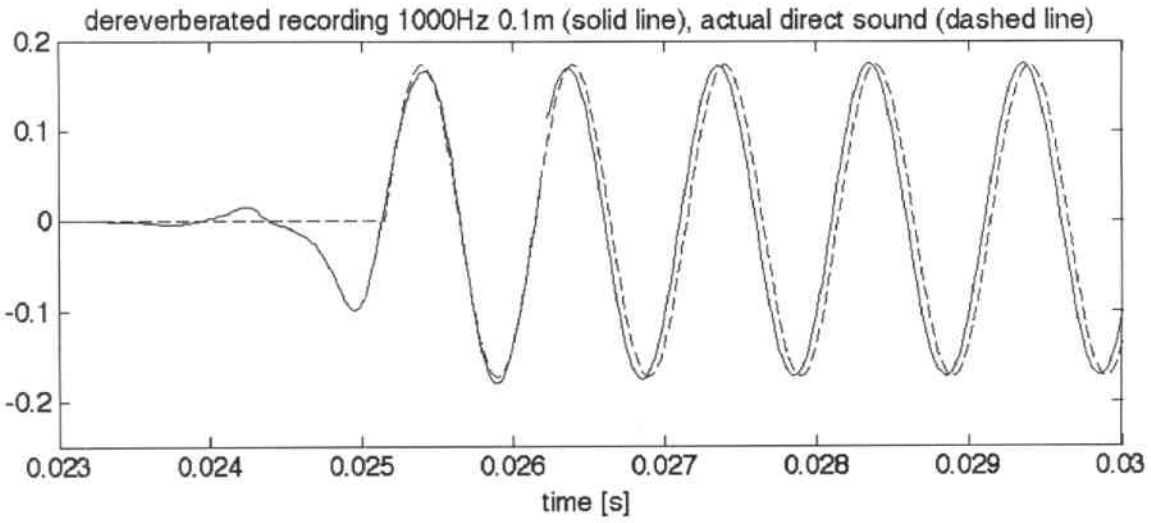


Figure 5.3: dereverberation of a 1000Hz signal at 0.1m recording distance

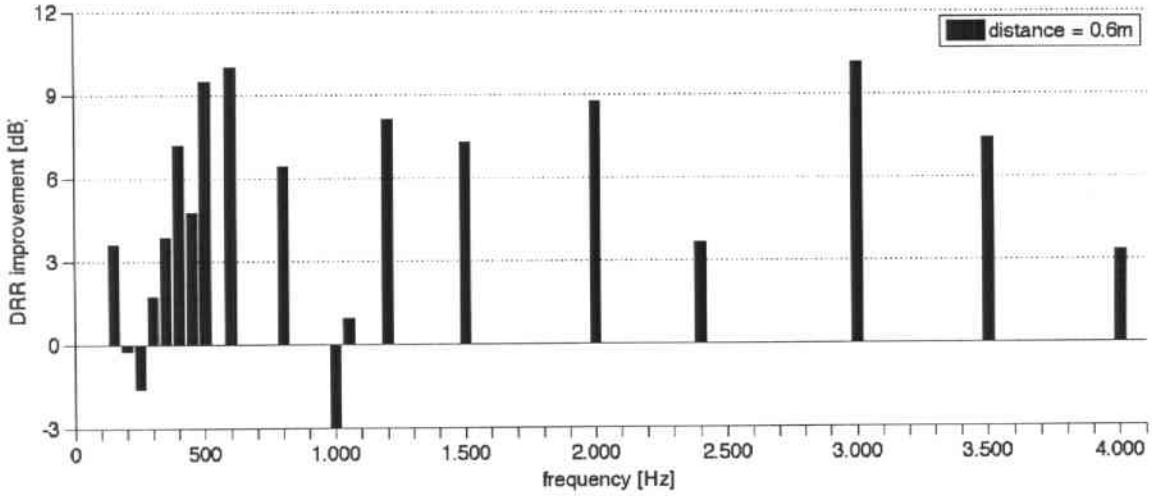


Figure 5.4: dereverberation performance at 0.6m recording distance

amplitude and phase offset precisely enough. From fig. 5.5 it can be seen that even a slight misdetection of the frequency (in this particular case the frequency was less than 0.5Hz away from the real value) can render the whole approach useless. Because of this, it is important to increase the precision and robustness of the PLL even further, when the transition from the dereverberation of single sine waves to the dereverberation of real speech shall be successful.

The second conclusion is actually a surprise: Due to the detection scheme for the reverb times introduced in section 4.2.3 it was expected that the dereverberation would only work properly for early reflections. The late reflections were considered to arrive within too short intervals of time so that they could not possibly be tracked and removed from the recording. As it turned out, the dereverberation of late reflections works too. The real reason for this has yet to be found out, because it is unlikely that each arriving late reflection can actually be tracked. The main reason is supposed to be the fact that due to the low amplitude of the late reflections, their contribution to the recorded signal is rather small. Furthermore, if multiple late reflections arrive at the microphone at the same time, they arrive from various directions with various phase delays, which is why they probably cancel each other

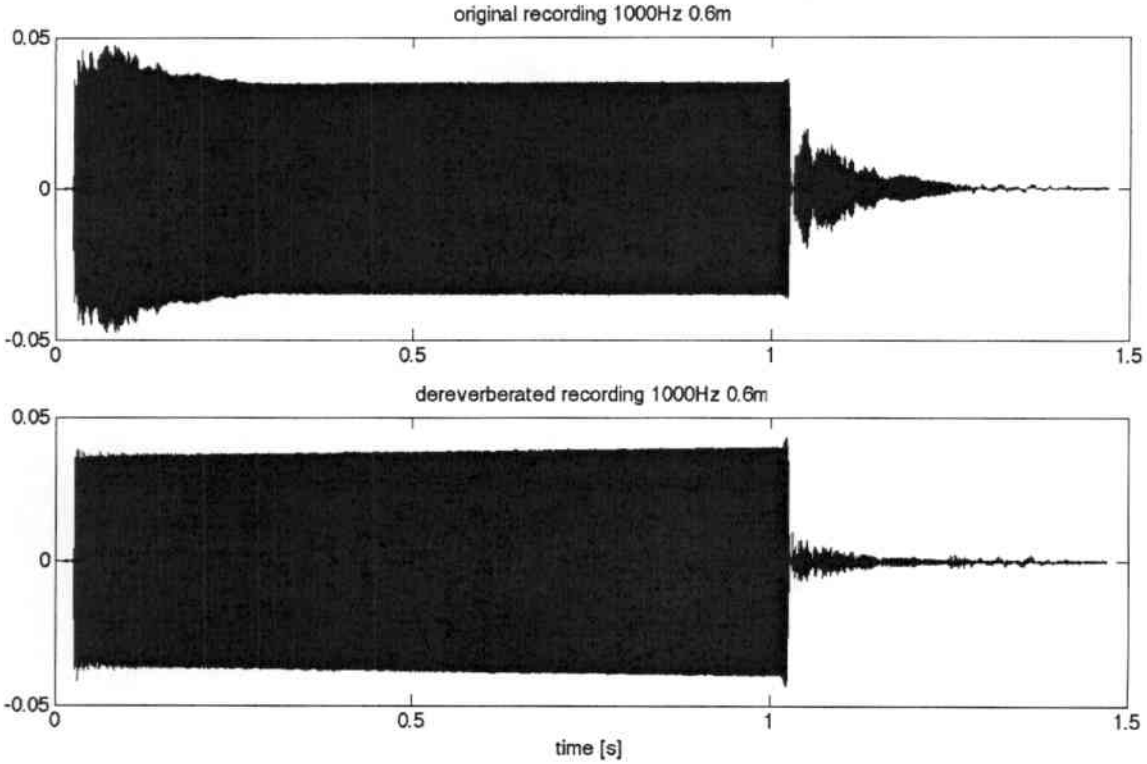


Figure 5.5: dereverberation of a 1000Hz signal at 0.6m recording distance

out statistically. To further investigate this issue it could be tried to remove only the early reflections and compare the result to a "full" dereverberation.

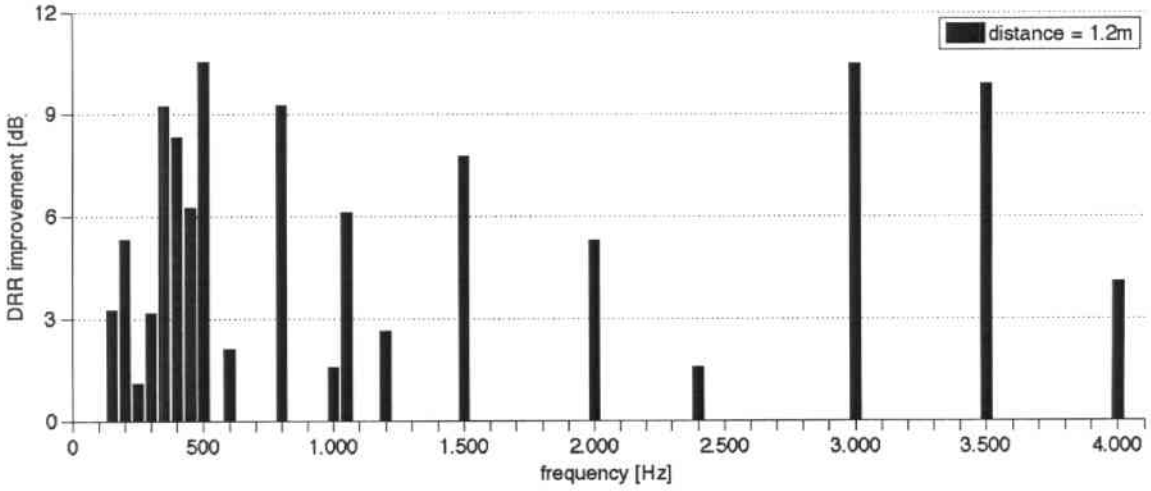


Figure 5.6: dereverberation performance at 1.2m recording distance

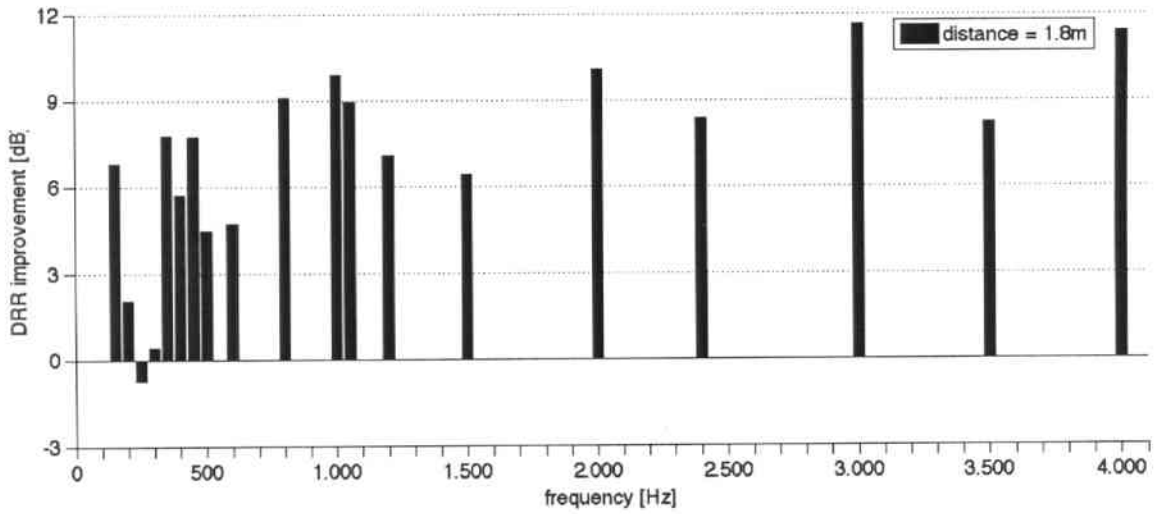


Figure 5.7: dereverberation performance at 1.8m recording distance

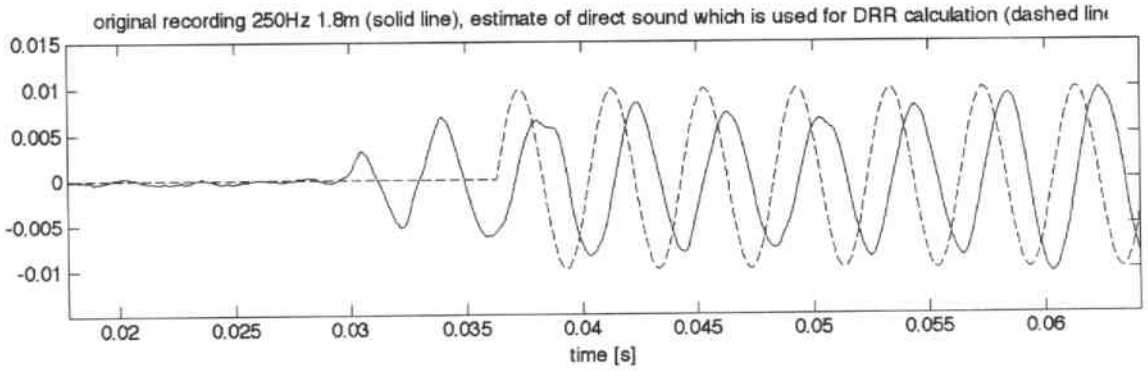


Figure 5.8: estimated direct sound of a 250Hz signal at 1.8m recording distance

6. Summary

To conclude the work, this chapter is intended to give a brief overview and rating over what has been found out and where possible "next steps" could lead to.

6.1 Conclusion

At first, the goal of demonstrating a dereverberation method for single sine waves which does not rely on any assumptions about the environment, has been achieved. The improvements in the direct-to-reverberant ratio for the various test recordings are mostly equal or better than those achieved with beamforming algorithms and multiple microphones in [Gaubitch05]. However, the restriction to single sine waves is a strong simplification and so the new approach is far from being applicable in the real world. Whether the proposed method will ever be useful for the dereverberation of real-world signals highly depends on the practicability of the extensions presented in the chapters 2.3.2 and 2.3.3.

While the overall approach looks promising due to the comparably high tracking accuracy of the PLLs, this accuracy must be increased even further. In fig. 5.5 it can be seen what happens if the measurement of a frequency, for example, is just slightly wrong. Additionally, the experience gained throughout this work suggests that improvements must also be made in the robustness of both the PLL tracking and the detection scheme which is used to detect the arrival of a wavefront.

Finally, the transition from single sine waves to multiple sine waves will also be a difficult step, but one that is absolutely necessary in order to tell reflections and phoneme-changes apart. To analyze multiple frequency components, the recorded signal must be splitted using bandpass filters which results in a smoothing of the recorded signal. In a smoothed signal, however, it will be more difficult to detect rapid changes which, in turn, is needed to find out when a reflected wavefront arrives at the microphone.

6.2 Future Work

The next steps to develop the dereverberation algorithm further are the implementation of the two methods presented in the chapters 2.3.2 and 2.3.3, in order to

find out if and how well the proposed approach can handle signals with multiple time-varying frequencies. The transition from artificial harmonic complexes to real speech can only be made after the algorithm works very well on artificial signals.

A. Matlab Code

A.1 PLL Initialization Script

```
%-------------------------------------------------------------------------
%       PLL Parameters
%-------------------------------------------------------------------------
addpath('../Simulink_Models');

% define audio file if this script is not run in batch mode
if exist('batchJob','var') == 0 || batchJob == 0
    audioFile = ['../data/sineRecordings.48.120cm/' ...
        'single_frequencies/1000Hz.wav'];
end

% extract frequencies played in the file out of the filename
if exist(audioFile,'file') == 2
    [¬, filename, ¬] = fileparts(audioFile);

    % parse filename to find PLL center frequencies
    % PLLBase: center frequency of first PLL
    % PLLSpacing: distance between center frequencies
    % PLLCount: number of PLLs to use
    if ¬isempty(strfind(audioFile,'harmonics'))
        PLLBase = str2double(filename(1:3));
        PLLSpacing = PLLBase;
        PLLCount = (str2double(filename(5:8))-PLLBase)/PLLSpacing;
    else
        pos_Hz = strfind(filename,'Hz');
        PLLBase = str2double(filename(1:pos_Hz-1));
        PLLSpacing = 0;
        PLLCount = 1;

        clear pos_Hz;
    end
else
    error('File does not exist');   % exit if file could not be found
end

```

```
clear filename;

% general parameters
f_sample = 48000;        % sample rate [Hz]
t_sample = 1/f_sample;   % sample time [s]

% calculate center frequencies for all PLLs
PLLCenter = PLLBase + (0:PLLCount-1) * PLLSpacing;

% prefilter parameters
use_prefilter = 0;       % use bandpass-prefilter yes/no
B_i= 100;                % PLL input bandwidth (= BW of prefilter) [Hz]
prefilterSlope = 50;     % undef. range between pass- and stopband [Hz]

% phase detector parameters
hilbertLength = 1200;    % length of the phase detector hilbert filter

% loop filter parameters
zeta = sqrt(2)/2;        % damping factor

% set SNR_i according to measured values
if ~isempty(strfind(audioFile,'10cm'))
    SNR_i = 46.4;
elseif ~isempty(strfind(audioFile,'60cm'))
    SNR_i = 34.3;
elseif ~isempty(strfind(audioFile,'120cm'))
    SNR_i = 29.1;
elseif ~isempty(strfind(audioFile,'180cm'))
    SNR_i = 21.5;
end

% VCO parameters
kappa = 1;                   % loop filter gain
lock_nco = 0;                % set NCO to fixed frequency yes/no

%-------------------------------------------------------------------------
%       Loading PLL Input Signal
%-------------------------------------------------------------------------

disp('Loading input audio file');

% first check if the file exists at all
if exist(audioFile,'file') == 2
    [pathname, filename, ext] = fileparts(audioFile);

    % replace path with '.' if it is empty (current folder)
    if strcmp(pathname,'') == 1
        pathname = '.';
    end

    % check if the file is a .wav file
    if strcmp(ext,'.wav') == 1
        % convert .wav to .mat if no corresponding .mat exists
        if exist(strcat(pathname,filesep,filename,'.mat'),'file') ~= 2
            wavToMat(f_sample,audioFile);
        end

        % change file name to the same file using .mat as ending
        audioFile = strcat(pathname,filesep,filename,'.mat');
```

```
95      end
96
97      % finally load the file
98      load(strcat(pathname,filesep,filename,'.mat'));
99  else
100     error('File does not exist');   % exit if file could not be found
101 end
102
103 % reverse vector if needed for backwards processing:
104 if exist('reverseProc','var') == 1 && reverseProc == 1
105     wavesignal(:,2) = wavesignal(end:-1:1,2);
106 end
107
108 % create empty vector for filtered versions of the original signal
109 ws_append = zeros(length(wavesignal(:,1)),PLLCount);
110 wavesignal = horzcat(wavesignal,ws_append);
111
112 % copy unfiltered version of input to each channel
113 for i = 1:PLLCount
114     wavesignal(:,i+2) = wavesignal(:,2);
115 end
116
117 clear pathname filename ext ws_append;
118
119 %-------------------------------------------------------------------------
120 %       Calculating Prefilter Coefficients
121 %-------------------------------------------------------------------------
122
123 if use_prefilter == 1
124
125     disp('Applying bandpass filterbank');
126
127     str = 'Filter lengths:';
128
129     for i = 1:PLLCount
130
131         % calculate passband edges
132         fp1 = PLLCenter(i) - B_i / 2;
133         fp2 = PLLCenter(i) + B_i / 2;
134
135         % calculate stopband edges
136         fst1 = fp1 - prefilterSlope;
137         fst2 = fp2 + prefilterSlope;
138
139         % design filter
140         d = fdesign.bandpass('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2', ...
141             fst1,fp1,fp2,fst2,60,1,60,f_sample);
142         hd = design(d,'cheby2','matchexactly','passband');
143
144         % zero-phase filter the data: filter forward, reverse in time
145         % filter backward and reverse in time again
146         wavesignal(:,i+2) = filter(hd,wavesignal(:,2));
147         wavesignal(:,i+2) = wavesignal(end:-1:1,i+2);
148         wavesignal(:,i+2) = filter(hd,wavesignal(:,i+2));
149         wavesignal(:,i+2) = wavesignal(end:-1:1,i+2);
150
151         str = [str ' ' num2str(hd.order)];
152
153         clear fst1 fst2 fp1 fp2 d hd;
```

```
154     end
155
156     % plot orders of the generated filters:
157     disp(str);
158     clear str i;
159 end
160
161 %-------------------------------------------------------------------------
162 %       Calculating Hilbert Filter Coefficients
163 %-------------------------------------------------------------------------
164
165 disp('Applying hilbert filter');
166
167 d = fdesign.hilbert('n,tw',hilbertLength,100,f_sample);
168 hd = design(d,'firls');
169
170 % create empty vector for hilbert filtered input for each PLL channel
171 wavesignal_hilbert = zeros(length(wavesignal(:,1)),PLLCount + 1);
172 wavesignal_hilbert(:,1) = wavesignal(:,1);  % copy timestamps
173
174 for i = 1:PLLCount
175     % filter signals
176     wavesignal_hilbert(:,1+i) = filter(hd,wavesignal(:,2+i));
177
178     % move them back in time to correct hilbert filter group delay
179     wavesignal_hilbert(1:end-hilbertLength/2,1+i) = ...
180         wavesignal_hilbert(1+hilbertLength/2:end,1+i);
181 end
182
183 % truncate vectors of samples (to remove the unnecessary ending)
184 wavesignal_hilbert = wavesignal_hilbert(1:end-hilbertLength/2,:);
185 wavesignal = wavesignal(1:end-hilbertLength/2,:);
186
187 clear d hd i;
188
189 %-------------------------------------------------------------------------
190 %       Calculating Loop Filter Coefficients
191 %-------------------------------------------------------------------------
192
193 disp('Calculating loop filter coefficients');
194
195 B_L = SNR_i * B_i / 12;                  % loop bandwidth
196 w_n = 2 * B_L / (zeta + 1/(4*zeta));     % natural frequency [rad/s]
197
198 tau1 = 2 * pi * kappa / (w_n * w_n);
199 tau2 = (2 * zeta / w_n);
200
201 % prewarp filter to compensate for bilinear transform warping
202 f_stop = 1 / tau2;
203 f_stop_prewarped = 2*f_sample*tan(f_stop/(2*f_sample));
204 tau2 = 1 / f_stop_prewarped;
205
206 % transfer function in Laplace domain
207 nums = [tau2 1];
208 dens = [tau1 0];
209
210 clear tau1 tau2 f_stop f_stop_prewarped;
211
212 % transfer function in z-domain
```

```
213 [numz, denz] = bilinear(nums,dens,f_sample);
214 numz = real(numz);
215 denz = real(denz);
216
217 disp('+--------------------------');
218 disp('| Loop Parameters:');
219 disp(['| w_n          = ', num2str(w_n / (2*pi)), ' Hz']);
220 disp(['| zeta         = ', num2str(zeta)]);
221 disp(['| Δ_f_po = ', num2str(1.8*w_n*(zeta+1)/ (2*pi)), ' Hz']);
222 disp(['| Δ_f_L  = ', num2str(2*zeta*w_n / (2*pi)), ' Hz']);
223 disp(['| T_L          = ', num2str(1000 * 2*pi/w_n), ' ms']);
224 disp(['| B_L          = ', num2str(B_L), ' Hz']);
225 disp('+--------------------------');
226
227 clear nums dens zeta f_lock w_n B_i B_L SNR_i;
228
229 %-----------------------------------------------------------------------
230 %       Load model and set simulation duration
231 %-----------------------------------------------------------------------
232
233 if exist('batchJob','var') == 0 || batchJob == 0
234     disp('Loading Simulink model');
235     load_system('../Simulink_Models/PLL_Hilbert');
236     open_system('../Simulink_Models/PLL_Hilbert');
237 end
238
239 % set duration of simulation
240 set_param('PLL_Hilbert','StopTime',num2str(wavesignal(end,1),15));
```

A.2 Dereverberation Script

```
1  f_sample = 48000;
2
3  amplFolder = 'sine_funlocked/single_frequencies.120cm/';
4  phaseFolder = 'sine_flocked/single_frequencies.120cm/';
5
6  freq = 1000;
7
8  axesList = [];  % list for axes to be synchronized afterwards
9
10 %% ##########################################################################
11 % load and smooth phase and amplitude
12
13 % load both files
14 fileName = strcat(num2str(freq),'Hz.out.mat');
15 amplObj = ProcessedSineRecording(strcat(amplFolder,fileName));
16 phaseObj = ProcessedSineRecording(strcat(phaseFolder,fileName));
17
18 % load amplitude and phase
19 a = amplObj.getSignalByIndex(4);
20 pll = amplObj.getSignalByIndex(1);
21 pll = pll(:,2);
22 mic = phaseObj.getSignalByIndex(1);
23 mic = mic(:,1);
24 p = phaseObj.getSignalByIndex(2);
25 f = amplObj.getSignalByIndex(3);
26
```

```
27 % move amplitude forwards in time due to the backwards processing
28 a(601:end) = a(1:end-600);
29 f(601:end) = f(1:end-600);
30 pll(601:end) = pll(1:end-600);
31 a(1:600) = 0;
32 f(1:600) = 0;
33 pll(1:600) = 0;
34
35 % smooth amplitude and phase by a median filter
36 p_s = medfilt1(p,48);
37 a_s = medfilt1(a,48);
38 f_s = medfilt1(f,48);
39
40 clear a p f out;
41
42 %% ########################################################################
43 % determine start and end of direct sound
44
45 % find maximum and minimum of derivative of the microphone signal
46 % (add 600 to compensate for backwards processing)
47 startSample = amplObj.getStartSample() + 600;
48 stopSample = amplObj.getStopSample() + 600;
49
50 disp(['start of direct sound at ' num2str(freq) 'Hz at sample ' ...
51     num2str(startSample)]);
52 disp(['end of direct sound at ' num2str(freq) 'Hz at sample ' ...
53     num2str(stopSample)]);
54
55 clear peaks peakTimes maxIndex diffA d hd;
56
57 %% ########################################################################
58 % initialize vectors for calculated reverb amplitude and phase
59
60 times = 0:length(a_s)-1;
61 times = times ./ f_sample;
62
63 a_ref = zeros(length(a_s),1);
64 f_ref = zeros(length(a_s),1);
65 p_ref = zeros(length(a_s),1);
66 a_ref(startSample:stopSample) = median(a_s(startSample: ...
67     startSample+48));
68 f_ref(startSample:stopSample) = trimmean(f_s(startSample: ...
69     stopSample),10);
70 p_ref(startSample:stopSample) = mod(median(p_s(startSample: ...
71     startSample+48)),2*pi);
72
73 %% ########################################################################
74 % create reference signal to measure the DRR
75
76 % cut beginning of direct sound out of the microphone signal
77 searchFrame = smooth(mic(startSample-48:startSample+96),5);
78
79 % generate one period of the reference sine wave
80 tref = 0:1/48000:1/freq;
81 refFrame = a_ref(startSample) * sin(2*pi*f_ref(startSample)*tref);
82
83 % calculate time of arrival by finding the correlation peak
84 [~,corrpeak] = max(xcorr(refFrame,searchFrame));
85
```

```
86 t_o_a = length(searchFrame) - corrpeak;
87 t_o_a = t_o_a + startSample - 48;
88
89 tref = 0:1/48000:1;
90 sineRef = a_ref(startSample) * sin(2*pi*f_ref(startSample)*tref);
91 signalRef = zeros(length(a_s),1);
92 signalRef(t_o_a:t_o_a+length(tref)-1) = sineRef;
93
94 clear searchFrame tref refFrame t_o_a sineRef;
95
96 %% ########################################################################
97 % calculate reverb amplitude and phase
98
99 reverbY = a_s .* sin(p_s) - a_ref .* sin(p_ref);
100 reverbX = a_s .* cos(p_s) - a_ref .* cos(p_ref);
101
102 reverbP = atan2(reverbY,reverbX);
103 reverbA = reverbX ./ cos(reverbP);
104
105 reverbP(1:startSample+48) = 0;
106 reverbA(1:startSample+48) = 0;
107
108 %% ########################################################################
109 % calculate derivative of calculated reverb amplitude and phase
110
111 d = fdesign.differentiator('n',9,f_sample);
112 hd = design(d,'firls');
113
114 diffA = abs(filter(hd,reverbA));
115 diffP = abs(filter(hd,reverbP));
116
117 %% ########################################################################
118 % determine times for incoming reflections
119
120 % root mean square normalization over a sliding windows (1ms)
121 arms = diffA'./rms(diffA, 48, 47, 1);
122 prms = diffP'./rms(diffP, 48, 47, 1);
123
124 % moving average (0.5ms)
125 arms = smooth(arms,24,'moving');
126 prms = smooth(prms,24,'moving');
127
128 % detect peaks with a minimum distance of 0.5ms
129 [¬,refTimesA] = findpeaks(arms,'MINPEAKDISTANCE',24);
130 [¬,refTimesP] = findpeaks(prms,'MINPEAKDISTANCE',24);
131
132 refTimes = union(refTimesA,refTimesP) - 4;
133
134 %% ########################################################################
135 % iterate over reflections
136
137 derevCount = length(refTimes)-1;     % number of reflections to remove
138 detectionMargin = 0.3;               % single sided
139
140 derev = p11;
141
142 for i = 1:derevCount
143     % calculate 'duration' of current reflection
144     refTimestamp = refTimes(i);      % unit = sample number
```

```
145     nextTimestamp = refTimes(i+1);
146
147     dist = nextTimestamp - refTimestamp;
148
149     % consider margin for the reflection 'duration'
150     t1 = ceil(refTimestamp + detectionMargin * dist);
151     t2 = floor(nextTimestamp - detectionMargin * dist);
152
153     % calculate average amplitude and phase inside for the reflection
154     reverbAmplitude = median(reverbA(t1:t2));
155     reverbPhase = median(reverbP(t1:t2));
156
157     % generate reverberation signal
158     rev = reverbAmplitude * sin(2*pi*f_ref(startSample)*times ...
159         + reverbPhase);
160     rev(1:refTimestamp-1) = 0;
161     rev(nextTimestamp:end) = 0;
162
163     % subtract reverberation from recorded signal
164     rev = rev';
165     derev = derev - rev;
166  end
167
168 % plot dereverberated signal
169 fig = figure();
170 ax = gca();
171 plot(times,[pll, derev, signalRef]);
172
173 title(ax,['dereverberation step ', num2str(i)]);
174 legend(ax,'PLL output','dereverberated PLL output', ...
175     'reference direct sound');
176 plotbrowser(fig,'on');
177 axesList = [axesList ax];
178 set(ax,'DeleteFcn',@ProcessedRecording.removeAxesFromList);
179
180 %% ########################################################################
181 % calculate DRR
182
183 direct = sqrt(sum(signalRef .* signalRef) / length(signalRef));
184
185 revBefore = (mic-signalRef) .* (mic-signalRef);
186 revBefore(isnan(revBefore)) = 0;
187 revBefore = sqrt(sum(revBefore) / length(signalRef));
188
189 revAfter = (derev-signalRef) .* (derev-signalRef);
190 revAfter(isnan(revAfter)) = 0;
191 revAfter = sqrt(sum(revAfter) / length(signalRef));
192
193 DRRbefore = direct/revBefore;
194 DRRafter = direct/revAfter;
195
196 DRRimprovement = 10 * log10(DRRafter/DRRbefore); % unit = dB
197 disp(['DRR improved by ' num2str(DRRimprovement) 'dB']);
```

Bibliography

[Best93] Roland Best. *Phase-locked loops: theory, design and applications.* 2nd ed. New York: McGraw-Hill, 1993. ISBN: 0-07-911386-9.

[DIN1311-1] DIN German Institute for Standardization: Fundamental Technical Standards Committee. *DIN 1311-1: (Mechanical) vibrations, oscillation and vibration systems - Part 1: Basic concepts, survey.* Beuth Verlag GmbH, Feb. 2000.

[Estienne01] Claudio Estienne, Patricia Pelle, and Juan Pablo Piantanida. "A Front-end for Speech Recognition Systems Using Phase-Locked Loops". In: *Proceedings of the Workshop in Information Processing and Control (RPIC).* Santa Fe, 2001, pp. 88–91.

[Gardner05] Floyd M. Gardner. *Phaselock techniques.* 3rd ed. Hoboken, New Jersey: Wiley-Interscience, 2005. ISBN: 978-0-471-43063-6.

[Gaubitch05] Nikolay Gaubitch and Patrick Naylor. "Analysis of the dereverberation performance of microphone arrays". In: *Proceedings of the International Workshop on Acoustic Echo and Noise Control (IWAENC).* Eindhoven, 2005.

[Karimi-Ghartemani01] Masoud Karimi-Ghartemani and M. Reza Iravani. "A new phase-locked loop (PLL) system". In: *Proceedings of the IEEE Midwest Symposium on Circuits and Systems (MWSCAS).* Vol. 1. Dayton, Ohio, 2001, pp. 421–424. DOI: 10.1109/MWSCAS.2001.986202.

[Kinoshita05a] Keisuke Kinoshita, Tomohiro Nakatani, and Masato Miyoshi. "Efficient Blind Dereverberation Framework for Automatic Speech Recognition". In: *Proceedings of the European Conference on Speech Communication and Technology (INTERSPEECH).* 2005, pp. 3145–3148.

[Kinoshita05b] Keisuke Kinoshita, Tomohiro Nakatani, and Masato Miyoshi. "Fast Estimation of a Precise Dereverberation Filter based on Speech Harmonicity". In: *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* Vol. 1. 2005, pp. 1073–1076. DOI: 10.1109/ICASSP.2005.1415303.

[McAulay86] Robert McAulay and Thomas Quatieri. "Speech analysis/Synthesis based on a sinusoidal representation". In: *IEEE Transactions on Acoustics, Speech and Signal Processing* 34.4 (1986), pp. 744–754. ISSN: 0096-3518. DOI: 10.1109/TASSP.1986.1164910.

[Nakatani03a] Tomohiro Nakatani and Masato Miyoshi. "Blind dereverberation of single channel speech signal based on harmonic structure". In: *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Vol. 1. 2003, pp. 92–95. DOI: 10.1109/ICASSP.2003.1198724.

[Nakatani03b] Tomohiro Nakatani, Masato Miyoshi, and Keisuke Kinoshita. "Implementation and effects of single channel dereverberation based on the harmonic structure of speech". In: *Proceedings of the International Workshop on Acoustic Echo and Noise Control (IWAENC)*. Hong Kong, 2003, pp. 91–94.

[Nakatani07] Tomohiro Nakatani, Keisuke Kinoshita, and Masato Miyoshi. "Harmonicity Based Blind Dereverberation for Single-Channel Speech Signals". In: *IEEE Transactions on Audio, Speech and Language Processing* 15.1 (2007), pp. 80–95. ISSN: 1558-7916. DOI: 10.1109/TASL.2006.872620.

[Pearson96] Qiguan Lin et al. "Robust distant-talking speech recognition". In: *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Vol. 1. May 1996, pp. 21–24. DOI: 10.1109/ICASSP.1996.540280.

[Pol46] Balth. van der Pol. "The Fundamental Principles of Frequency Modulation". In: *Journal of the Institution of Electrical Engineers - Part III: Radio and Communication Engineering* 93.23 (May 1946), pp. 153–158.

[Sengpiel11] Eberhard Sengpiel, Berlin University of the Arts. *Sound pressure p and the inverse distance law 1/r*. 2011. URL: http://www.sengpielaudio.com/calculator-distancelaw.htm (visited on 01/24/2011).

[Smith07] Julius O. Smith. *Introduction to Digital Filters with Audio Applications*. online book. 2007. URL: https://ccrma.stanford.edu/~jos/fp/Zero_Phase_Filters_Even_Impulse.html (visited on 01/25/2011).

[Stephens02] Donald R. Stephens. *Phase locked loops for wireless communications: digital, analog and optical implementations*. 2nd ed. Boston: Kluwer Academic Publishers, 2002. ISBN: 0-7923-7602-1.