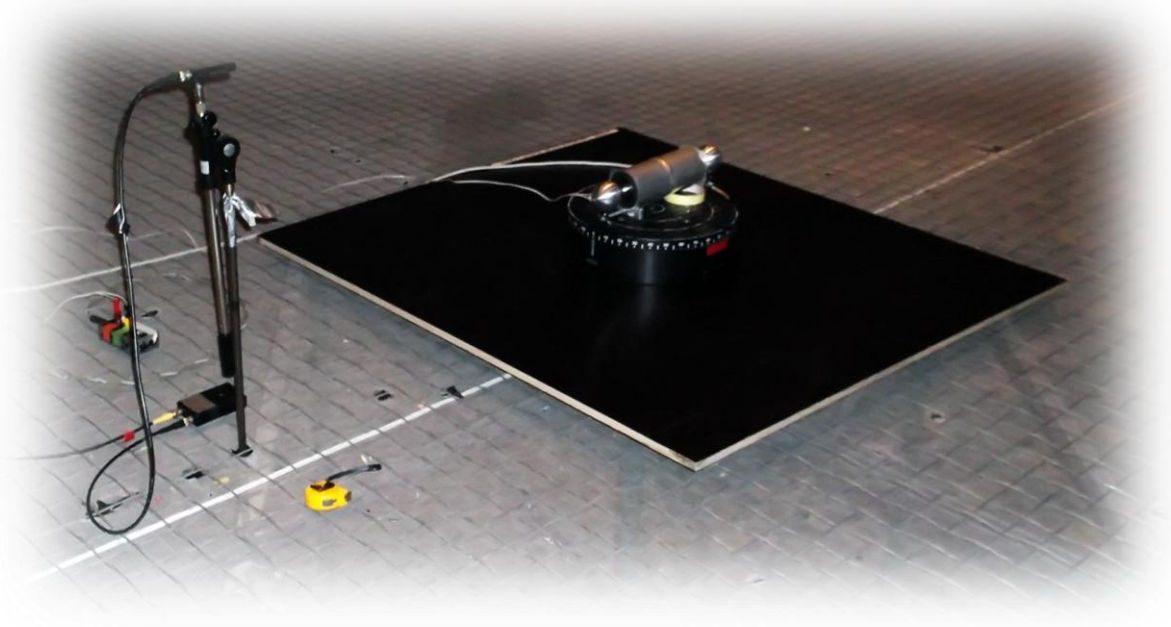
CHALMERS



Multipoint optimization of a loudspeaker impulse response

Master of Science Thesis

GUILLAUME PERRIN

Department of Civil and Environmental Engineering
Division of Applied Acoustics
Vibroacoustics Group
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2012
Master's Thesis 2012:137

MASTER'S THESIS 2012:137

Multipoint optimization of a loudspeaker impulse response

Master of Science Thesis

GUILLAUME PERRIN

Department of Civil and Environmental Engineering
Division of Applied Acoustics
Vibroacoustics Group
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2012

Multipoint optimization of loudspeaker impulse response

Master of Science Thesis
GUILLAUME PERRIN

Examensarbete / Institutionen för bygg- och miljöteknik,
Chalmers tekniska högskola 2012:137

Department of Civil and Environmental Engineering
Division of Applied Acoustics
Vibroacoustics Group
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: + 46 (0)31-772 1000

Cover:
Measurement setup for directivity measurements in the anechoic chamber.

Chalmers Reproservice / Department of Civil and Environmental Engineering
Göteborg, Sweden 2012

Multipoint optimization of loudspeaker impulse response

Master of Science Thesis

GUILLAUME PERRIN
Department of Civil and Environmental Engineering
Division of Applied Acoustics
Vibroacoustics Group
Chalmers University of Technology

ABSTRACT

It's relatively simple to obtain a very good impulse response at a single control point from a loudspeaker using an equalization filter. But in real life, an average impulse response over a larger geometrical area using multiple control point can be more interesting (e.g. for an audience). The purpose of this study is to investigate, analyse and simulate a multichannel equalization filter for a home-made loudspeaker. This filter must be able to optimize a loudspeaker impulse response over an area defined by at least three points.

To answer this problem, we apply an adaptive filtering method since it offers a simple and effective way to expand to multichannel filtering. It also allows us to implement the Volterra theory through a second-order Volterra (SOV) filter to take nonlinearities into account. After several tests and simulations the well-known normalized least mean square (NLMS) algorithm was chosen and implemented using a finite impulse response (FIR) filter and the filtered-x arrangement.

Simulations were conducted with Matlab using for beginning either small impulse responses, or real impulse responses from database. At the end, we took impulse responses from our own measurements of our loudspeaker made in an anechoic room. While we achieved good results with the multichannel linear filtering, we couldn't make the SOV filter to work and it kept diverging. We couldn't find the reason why but some leads are mentioned for further investigations.

The principal conclusion is that multichannel filtering is all about trade-off. The mean square error (MSE) performance is impacted by the size of the equalized area and the number of points to optimize. Moreover, this study should serve as a starting point for further projects that could lead to the integration in a product (e.g. loudspeaker and/or amplifier).

Keywords: Impulse response, Volterra filter, inverse filtering, multichannel, equalization, digital filter, adaptive algorithm, NLMS, FxLMS, MSE.

Contents

Preface

This study was conducted at the Applied Acoustics laboratory of Chalmers from February to June 2012.

Acknowledgements

This Master Thesis represents my last work for Chalmers and signs the end of an incredible exchange year in this beautiful country that is Sweden. I could say I had the time of my life!

It was literally a blast to be able to take my first steps into the research environment at the Applied Acoustics laboratory of Chalmers. I needed this experience for my own sake, and I thank my supervisor, Pontus Thorsson, for giving me this opportunity.

And of course, what could I be without my family and my friends. No need to name them, they know who they are and how I love them. Thanks for caring, thanks a ~~million~~ ~~billion~~ zillion!

Göteborg, Sweden - June 2012

Guillaume Perrin

Notations and abbreviations

(M)MSE	(Minimum) Mean Square Error
(N)LTI	(Non)Linear Time-Invariant
FIR	Finite Impulse Response
FR	Frequency Response
IIR	Infinite Impulse Response
IR	Impulse Response
LMS	Least Mean Square
RLS	Recursive Least Square
SOV	Second-Order Volterra
TF	Transfer Function

t	Time variable
n	Sample variable
s	Complex variable
$x(t)$	Continuous-time signal (causal when $t \geq 0$)
$x(n)$	Discrete-time signal (causal when $n \geq 0$)
$X(s)$	Complex frequency spectrum of signal $x(t)$

We point out to the reader that we should in fact write $x(n\Delta t)$ for a discrete-time signal. Δt being the sampling time with $\Delta t = \frac{1}{f_S}$ and f_S the sampling frequency. But to simplify, we normalize using $f_S = \Delta t = 1$ and thus can write $x(n)$.

1 Introduction

In sound-reproduction system, we usually want to reproduce a sound with the most fidelity to the original. We want to experience a song exactly as the composer made it. Thus there is a real need in loudspeaker optimization[1].

Equalization techniques have long been used to correct loudspeaker and room responses. Traditionally it involves the optimization of the frequency spectrum with graphic or parametric equalizers [1-3]. These techniques have obvious limitations (for example with the frequency resolution). Furthermore, they do not attempt to optimize the phase response, which is often disregarded whereas it contains important information [4]. A more complete tool is to use the impulse response which, as we are going to see, contains all the information we need.

But an impulse response represents only a linear system. In the real world, non-linearity transformation occurs and deforms the signal [5]. Thus it must be taken into account when filtering.

Common approach uses a linear single point equalization (i.e. using only one microphone). While this method yields to good results [6], the optimized location is fixed and quite small. If the listener is moving or if there is a group of person scattered around, then the optimization is close to worthless since only one listener will benefit from it. Moreover, equalization on one point can sometimes have bad effect on other points and then the experience would be worse for the audience [7]. In such cases, multipoint equalization would be better as it would enlarge the equalized zone to a whole area. Figure 1 depicts that concept. The goal is to apply a multipoint equalization process to produce a "roughly" good impulse response over a large area for a specific home-made loudspeaker.

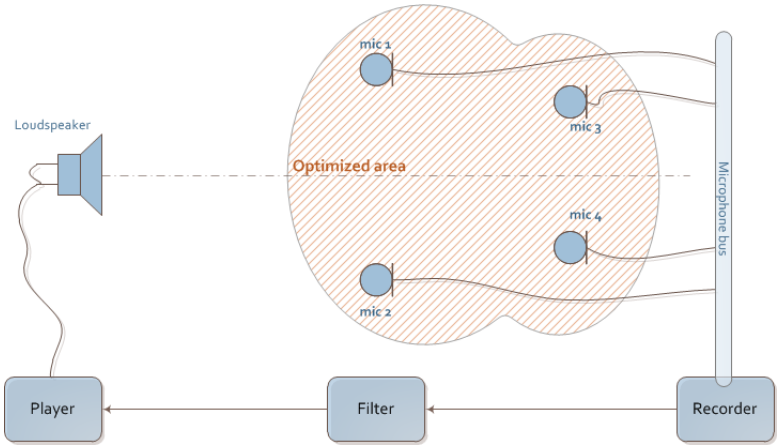


Figure 1 - Concept of the system.

Achieving a perfect inversion would be possible though by using MINT (*m*ultiple input/output *in*verse *t*heorem) as described in [8]. But it requires a number of loudspeakers greater than the number of microphones and is thus not applicable for our system.

[1] Equalization, inverse filtering or inversion have similar meaning and can be found indifferently through this report.

Still, many works can be found in literature regarding multipoint linear equalization, but dealing mainly with room acoustics [2, 3, 7, 9-11]. A multipoint frequency domain approach is used in [7], but allows to compensate the magnitude spectrum irregularities only. The same approach is used with a fuzzy c-mean clustering algorithm in [2]. Analysing the transfer function of the system, a multipoint equalization method by using common acoustical poles is proposed in [3] but only permits to suppress the common peaks. Regarding time domain equalization, [9] presents an common adaptive least square error method. [10, 11] use a more statistical approach based on a linear minimum mean squared error criterion.

As for non-linear equalization, the vast majority of work that can be found concerns fixed-point optimization with intensive use of Volterra Series [12-16] or a little of time-delay feedforward neural network [17]. However, it is possible to implement Volterra Filters to multi-point optimization using the adaptive least square error method.

1.1 Objective

A multichannel inverse filtering of a loudspeaker impulse response is implemented using an adaptive least square error method (see Figure 2). The goal is to obtain with an impulse signal at the input the very same net impulse at the outputs to have the best sound-reproduction chain.

Different adaptive filtering approaches and types are tested and analysed. The nonlinearities are taken into account using the Volterra theory framework. Finally, the filtering system was tested on a home-made loudspeaker characterized in an anechoic room.

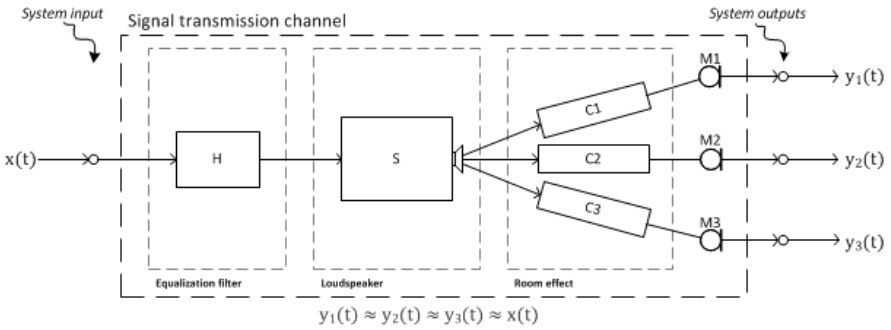


Figure 2 - Detailed concept of the system.

1.2 Structure of the report

Following the introduction, Chapter 2 presents the theoretical framework of this study, the system analysis theory, loudspeaker characterization, and inverse digital filtering including the multichannel approach. Chapter 3 gives the methodology used to conduct this thesis with the description of the simulations and the measurements processes and Chapter 4 presents the results. A discussion on them follows in Chapter 5, while Chapter 6 concludes the study by summing up our results. Chapter 7 offers some leads to further investigations and developments.

2 Background and theory

We present in this chapter the theories behind system analysis and what characterize a loudspeaker, then we move on to digital filters and the adaptive approach.

2.1 System analysis

System analysis is used to characterize an electrical system, analogue or digital. A system can be view as a black box with n inputs and m outputs, as in Figure 3.

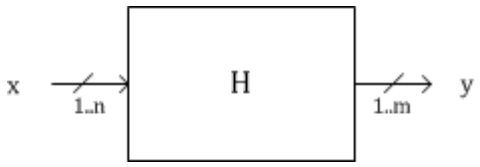


Figure 3 - Representation of a system with n inputs and m outputs

All the systems of this study are said to be Time-invariant[2] (TI) as the outputs do not depend explicitly on time (only on the inputs).

$$y(t) = f\big(x(t)\big) \;\equiv\; y(t+\Delta) = f\big(x(t+\Delta)\big) \quad \forall\, t, \Delta \tag{1}$$

It is a common simplification in signal processing [18], although one can argue that it is not true for loudspeakers with the aging of the mechanical components. But since this is a slow process, we can say with a fairly confidence that this is a Time-invariant system.

A system is characterized by how it responds to input signals. There is a lot of different ways to represent it, there are two main categories: the linear and the nonlinear approach.

2.1.1 Linear system

A system is linear when H can be described as a linear operator satisfying the properties of superposition, scaling and homogeneity summarize by equation (2).

$$\alpha y_1(t) + \beta y_2(t) = H\{\alpha x_1(t) + \beta x_2(t)\} \quad \forall\, \alpha, \beta \tag{2}$$

In other words, it is a system where the outputs are proportional to the inputs. From a frequency point of view, it's a system that cannot produce new frequency components that are not at the input. It can only alter the amplitude and phase.

With the property defined in this chapter's introduction, we obtain the well-known and well-study Linear Time-invariant (LTI) system.

A LTI system can be completely characterized by a single function, its impulse response (IR), noted $h(t)$ (also called the kernel, and is assumed to be causal, i.e. to satisfy $h(t) = 0 \quad \forall t < 0$).

[2] For a discrete (digital) system, the equivalent term is Shift-invariant system.

In other words, an IR is the reaction of the system to a short and strong excitation, the impulse. From a mathematical point of view, it's to feed the input with a Dirac delta function $\delta(t)$ (or a Kronecker delta function $\delta(n)$ in discrete-time). It's a function that can be seen as infinitely high and infinitely thin at the origin $t = 0$ with a total area of one (a discrete-time example can be seen Figure 4a). Furthermore, the Fourier Transform of this function gives one, i.e. an impulse in time has all possible excitation frequency in equal portion.

So equivalently in the frequency domain it can be completely characterized by its transfer function (TF), noted $H(s)$. The name frequency response can also be used, although loudspeaker's manufacturers usually think it only means magnitude over frequency (what about the phase?).

We can switch from the IR to the TF (or inversely) using the direct (or inverse) Laplace transform[3].

$$H(s) = \mathcal{L}\{h(t)\} = \int_{-\infty}^{+\infty} h(t)e^{-st}dt \tag{3}$$

The output of the system is the convolution of the input with the impulse response in time domain, see equation (4). In the frequency domain, the output will be simply the product between the input and the TF, see equation (5).

$$y(t) = h(t) * x(t) = \int_{-\infty}^{+\infty} h(\tau_1)x(t-\tau_1)d\tau_1 \tag{4}$$

$$Y(s) = H(s)\,X(s) \tag{5}$$

An illustration of impulse responses can be found Figure 4 below.

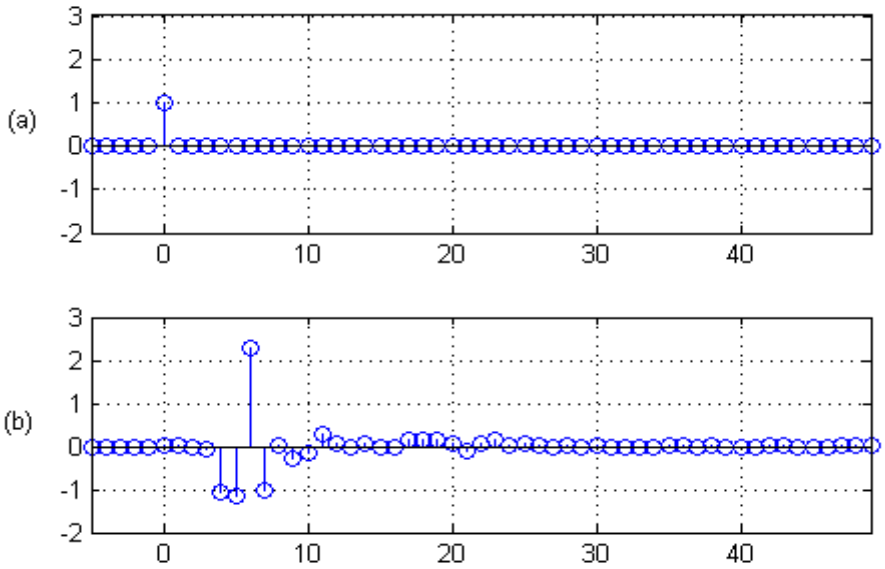


Figure 4 - Illustration of 50-tap impulse responses: (a) perfect theoritical IR, i.e. a Kronecker delta function; (b) simple IR with delay and ringing.

[3] The Fourier transform can also be used to obtain the transfer function $H(j\omega)$. But it is more commonly used with signals that are infinite in extent (like sinusoids). In the case of a discrete-time (sampled) system, we would use the Z-transform to switch between domains.

A LTI system is a powerful and simple tool to describe a lot of system, as usually nonlinearities can be easily disregarded. But when those nonlinearities became a little too prominent, using a linear filtering may increase them even more. So we have no choice but to take them into accounts.

2.1.2 Nonlinear system

A system is said nonlinear when the output signals are not directly proportional to the inputs, that is, there are no linear combinations that fully represent the system. We have then a nonlinear time-invariant (NLTI) system.

Nonlinearities distortions can be seen in two forms [19]: (i) harmonic distortions (HD), and (ii) intermodulation distortions (IMD). The first occurs when there is presence of harmonics which are not present in the original signal (see Figure 5). The second occurs when the input signal contains two or more frequency; the intermodulation between all those frequencies will produce new ones that are the sum and difference of the original frequencies (see Figure 6).

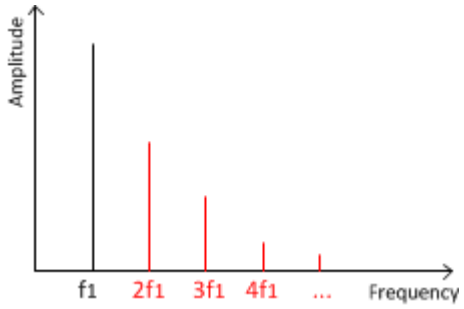


Figure 5 - Harmonic distortions of a single-frequency input signal (shown in red)

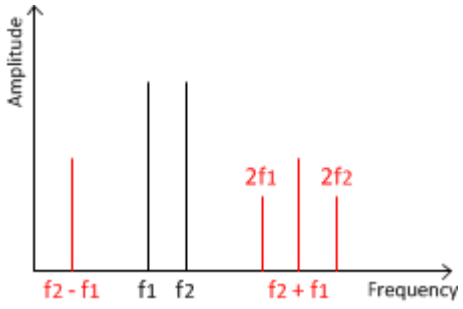


Figure 6 – 2nd Order IMD product of a dual-frequency input signal (shown in red)

Recently, the Volterra series expansion has been applied successfully to the analysis and identification of nonlinear systems [5, 15, 16, 20-23]. It's a multi-dimensional generalization of the impulse response function. There is no longer only one but an infinity of IR for the system. It can be seen as a Taylor series with memory effect [24].

Let H_p be the p-th order Volterra operator. We define the output as equation (6). A schematic of the Volterra system can be seen in Figure 7 below.

$$y(t) = \sum_{p} H_p\big(x(t)\big) \tag{6}$$

And a Volterra Operator is defined as the p-dimensional convolution of the input signal with the p-dimensional Volterra kernel h_p.

$$H_p\big(x(t)\big) = \int_{-\infty}^{+\infty} \dots \int_{-\infty}^{+\infty} h_p(\tau_1, \dots, \tau_p) x(t-\tau_1) \dots x\big(t-\tau_p\big) d\tau_1 \dots d\tau_p \tag{7}$$

If we insert equation (7) into (6), we obtain:

$$\begin{aligned} y(t) = & \int_{-\infty}^{+\infty} h_1(\tau_1)x(t-\tau_1)d\tau_1 \\ & + \int_{-\infty}^{-\infty}\int_{-\infty}^{+\infty} h_2(\tau_1,\tau_2)x(t-\tau_1)x(t-\tau_2)d\tau_1 d\tau_2 \\ & + \cdots \\ & + \int_{-\infty}^{+\infty} \dots \int_{-\infty}^{+\infty} h_p(\tau_1,\dots,\tau_p)x(t-\tau_1)\dots x(t-\tau_p)d\tau_1 \dots d\tau_p \end{aligned} \tag{8}$$

On the first line, we recognize a LTI system, it is the same equation as (4). This is the first-order Volterra operator H_1 corresponding to the linear part. On the second line is the second-order Volterra operator H_2, responsible for the bilinear combination of the input. We could go on and consider any higher orders *ad infinitum* but the complexity increases rapidly with the order. A second-order Volterra (SOV) system is said to be a good approximation for a loudspeaker [20].

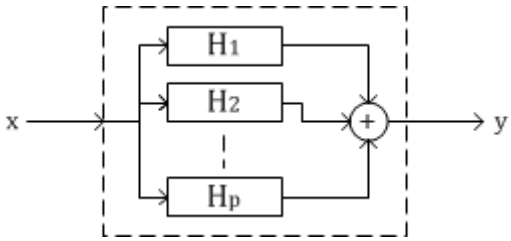


Figure 7 - A general Volterra system

Since we are going to work with discrete time, the sampled equivalent to equation (8) of a SOV system is given by:

$$\begin{aligned} y(n) = & \sum_{m_1=0}^{\infty} h_1(m_1)x(n-m_1) \\ & + \sum_{m_1=0}^{\infty}\sum_{m_2=0}^{\infty} h_2(m_1,m_2)x(n-m_1)x(n-m_2) \end{aligned} \tag{9}$$

2.2 Characterization of loudspeakers

Now we understand how to characterize a general system, let's look into the system of this study.

2.2.1 The dynamic loudspeaker

Dynamic loudspeakers convert audio coming from an electrical signal to a mechanical vibration of the air in order to recreate the sound we want to hear. Figure 8 is a simple sketch of a dynamic loudspeaker, also called driver.

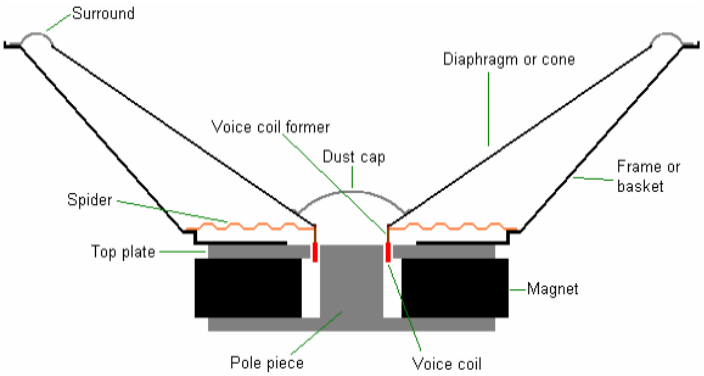


Figure 8 - Construction of a dynamic loudspeaker. Source [5].

First the electrical signal have to be amplified to a usable voltage by a Power Amplifier (PA - could be inside the loudspeaker). After that little pre-step, the signal is injected into a coil freely mounted on a magnet which generates a magnetic field. The signal variation in the coil will create a flux variation (resulting from the Faraday's law) which will create a corresponding force, the so-called Lorentz force, on the coil. Since the coil can move freely, it will drive the membrane (or diaphragm) attached to it. It is the variations of the cone that creates the sound waves.

Since it is hard for a single driver to reproduce the entire range of audible frequencies (usually defined as $20\ \text{Hz} - 20\ \text{kHz}$), many loudspeakers are often composed of several drivers [25]. The most common one, the so-called two-way system, has separate drivers for low and mid-high frequencies (respectively of big and small size).

Ideal loudspeakers produce acoustic waves that are a linear transformation of the electrical input signal [25]. Thus it could be seen as a LTI system and its main characteristic would be its impulse response. While this assumption relies on obvious oversimplification of the electro-mechanical, such analysis techniques still provide useful insight into loudspeaker performance.

Nonlinearities in loudspeakers are caused by various elements but the most dominants are ones related to the cone displacement and voice-coil excursion [26]: the force factor Bl, the electrical self-inductance and the mechanical stiffness of the suspension. Since the cone displacement is related to the input power or the frequency, we can say that nonlinear distortions are more important for low frequencies and/or large input power.

It is worth to add that the audio signal's travel doesn't stop after the loudspeaker; it has to reach the receiver (listener's ears or microphone) through the room. And once again the signal will be distorted, this time, by the presence of reflective wall that will cause echo and reverberation often undesirable. But hopefully, this room effect can be approximate to a linear system [27]. Therefore, room and loudspeaker equalization can be used with equal meaning as long as we restrict to the linear part.

2.2.2 Our specific loudspeaker

The loudspeaker we want to equalize is a specific one, as you can see Figure 9. It is composed of an empty steel tube with two drivers on each side. There is a plate

hermetically separating each driver's enclosure. The whole tube is hermetically sealed. Now the main features of this loudspeaker are the solid steel tear-shape reflectors in front of the drivers. They are carved in order to scattered the sound wave all around.

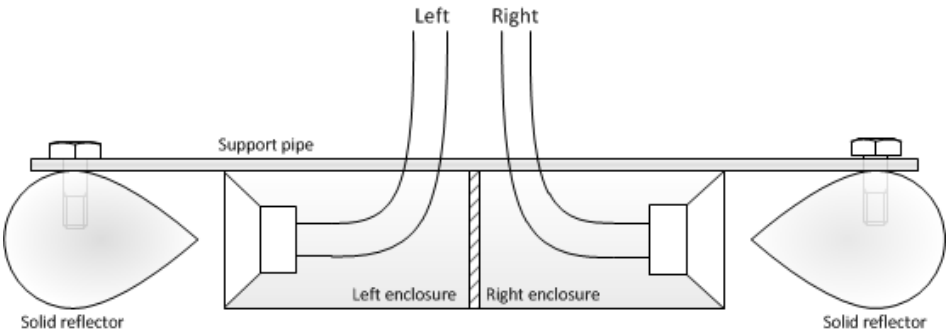


Figure 9 - Cutaway view of our loudspeaker

A thin support pipe holds the tube and the reflector together. Note that this loudspeaker needs to be hanging on a wall, since there is no stand to go on the floor. Furthermore, the loudspeaker will benefit from the wall's reflection.

Figure 10 - Photo of one side of the loudspeaker (driver and reflector).

2.2.3 Linear impulse response measurement

We want to acquire the impulse response as accurately as possible. In traditional impulse response measurement, periodic pulse and Maximal-Length Sequence (MLS) are often used as excitation signals [28]. Periodic pulse testing is the simplest and most intuitive method but usually results in a poor signal-to-noise ratio (SNR). MLS improve this method, yet it is shown that it is too much prone to nonlinearities [28].

The sine sweeps used as excitation signals provide a good way to measure linear impulse response [28]. In this technique, a log-swept[4] sine stimulus is employed (the frequency varies exponentially with time), see equation (10). The output presents, after deconvolution, a clean separation of linear response and harmonic distortion.

$$s(t) = \sin\left[K \cdot \omega_1 \left(e^{\frac{t}{K}} - 1\right)\right] \tag{10}$$

[4] Linear-swept has also been tried but with mitigated results, especially at low frequencies [28].

$$\text{With} \quad K = \frac{T}{\ln \frac{\omega_2}{\omega_1}} \tag{11}$$

With ω_1 and ω_2 respectively the lower and higher frequency limits, and T the time duration in seconds.

The energy of the signal as a function of $j\omega$ is then given by:

$$E(j\omega) \propto \frac{K}{\omega_1} \cdot \frac{1}{K + j\omega} \tag{12}$$

The energy drops while frequency increases (by 3dB/octave) as you can see Figure 11.

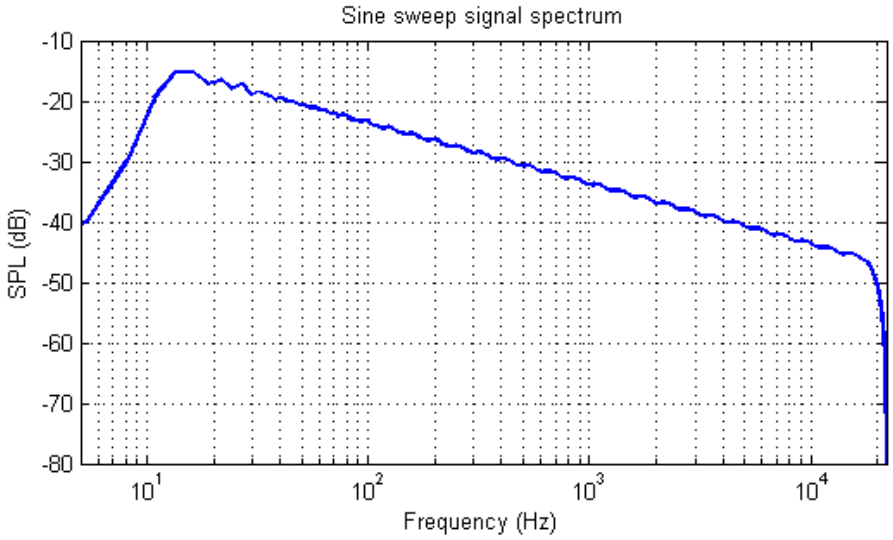


Figure 11 - Spectrum of the stimuli (sine sweep) signal.

Using an exponential sweep has thus two major advantages: (i) more energy is present in the measurement signal at low frequencies (which is beneficial as it improves the accuracy in that region); and (ii) nonlinearities result in anti-causal components in the impulse response, which can be easily separated (by taking only the causal part).

2.3 Digital filter

Signal processing can be done with an analogue (continuous-time) or a digital (discrete-time) filter. Many signals are analogue by nature (e.g. signal from a microphone) and one can wonder why we don't use analogue filter. Those filters use electrical components like resistors, capacitors to achieve their goals. But there are two major flaws associated with analogue filters: (i) the electrical component's value will drift with temperature and age; and (ii) the available functions are quite limited for complex filtering (e.g. adaptive filtering impossible due to the lack of analogue memory unit). Furthermore, modifying a digital filter is ridiculously easy compared to an analogue one since they are programmable. Thus, the use of a *Digital Signal Processor* (DSP) is attractive.

But digital filtering imposes some extra steps, since it deals with different type of signals. The conversion between a continuous-time signal $x(t)$ to a discrete-time

signal $x(n)$ is done by an *Analogue to Digital Converter* (ADC). The opposite equivalent is the *Digital to Analogue Convert* (DAC). See Figure 12 for an illustration.

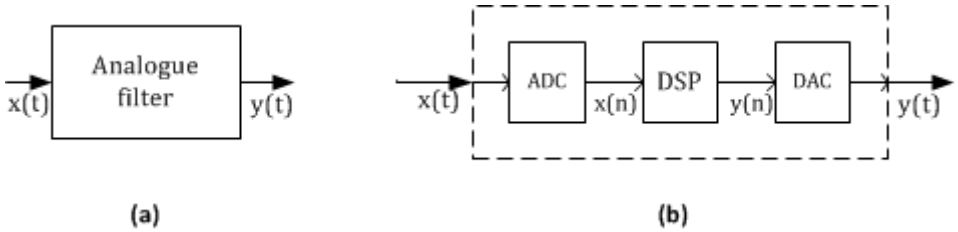


Figure 12 - (a) Analogue filter; (b) sampled data equivalent: digital filter.

Of course what we consider in this chapter 2.2 is filtering of LTI system (nonlinear filters are based on the concept of linear filters, enhanced with p-order combinations). As we explained in Chapter 2.1.1, a linear filter is completely characterized by its impulse response.

Let's now focus on the digital signal processing part. There are two alternative ways to do it: using recursive or non-recursive realisation.

2.3.1 Finite impulse response (FIR) filter

A finite impulse response (FIR) filter is the simplest form one can think of and pretty straightforward. The impulse response is of finite duration N and is equal to zero after (and before, since it is causal). It is non-recursive as it depends only of the input.

The different values h_i of the impulse response are called filter coefficients, or filter weights. If the filter is of size N, the filter order is of size $N-1$.

$$h = [h_0\ h_1\ \dots\ h_i \dots\ h_{N-1}] \tag{13}$$

The output is calculated as the discrete-time convolution:

$$\begin{aligned} y(n) &= h_0 x(n) + h_1 x(n-1) + \cdots + h_{N-1} x(n-N-1) \\ &= \sum_{i=0}^{N-1} h_i x(n-i) \end{aligned} \tag{14}$$

The major advantages of FIR filters are that they are unconditionally stable and can offer a linear phase response [18]. They are also simpler to implement. However, they usually require more delay and computation than their recursive counterparts.

2.3.2 Infinite impulse response (IIR) filter

For an IIR filter, the impulse response is non-zero after an infinite amount of time. To achieve this, the use of output feedback is necessary. Thus, there are two different coefficients type: the N feedforward filter coefficient a_i and the M feedback filter coefficient b_i.

$$a = [a_0\ a_1\ \dots\ a_i \dots\ a_{N-1}] \tag{15}$$

$$b = [b_0\ b_1\ \dots\ b_i \dots\ b_{M-1}] \tag{16}$$

And the output is calculated as (see the recursive part on the second line):

$$\begin{aligned} y(n) = {} & a_0 x(n) + a_1 x(n-1) + \cdots + a_{N-1} x(n-N-1) \\ & + b_0 y(n) + b_1 y(n-1) + \cdots + b_{M-1} y(n-M-1) \\ = {} & \sum_{i=0}^{N-1} a_i x(n-i) + \sum_{i=0}^{M-1} b_i y(n-i) \end{aligned} \tag{17}$$

The major advantage of IIR filter is that thanks to the recursion they require fewer coefficients and thus fewer computing resources [18]. However the recursion implies that the filter can be easily unstable, so one has to be careful when dealing with IIR filters.

2.4 Inverse filtering

Let S be a generic system (e.g. an acoustic path, a transmission line, a loudspeaker …) that we want to reduce the effect on our input signal x. Then, a filter H is placed in the forward path and will perform an inverse filtering of S in order to get $x = x'$. Figure 13 shows the schematic of this concept.

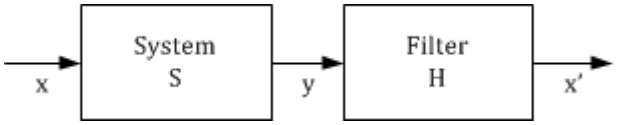


Figure 13 - Inverse filtering of a system.

There are several methods to calculate the impulse response of the filter. First of all, if one know exactly the transfer function of S, then we can simply take the inverse: $H = S^{-1}$. But this method has some flaws. The system S is not always known, and furthermore, the direct inverse is often unstable since most of loudspeaker and room IR are non-minimum phase function[5] [27].

Achieving a perfect inversion without knowing S is possible though by using MINT (*m*ultiple input/output *in*verse *t*heorem) as described in [8]. But it requires a number of inputs greater than the number of outputs and is thus not applicable for our system.

Another solution is to use the well-known method of adaptive filtering.

2.4.1 Single channel adaptive inverse filtering

An adaptive filter is a filter that can adjust its coefficients according to an optimization algorithm driven by an error signal.

Let S be an unknown system. There are basically two modes of operation of an adaptive filter: (i) direct system modelling to get the same IR as the unknown system and (ii) inverse system modelling to obtain the inverse IR. The second mode is the one we are interested in and is illustrated in Figure 14.

[5] A non-minimum phase function of a stable and causal system has one or more zeros in the right side of the Laplace Domain (or outside the unit circle in discrete-time). Therefore, the inverse of this function would be causal but unstable.

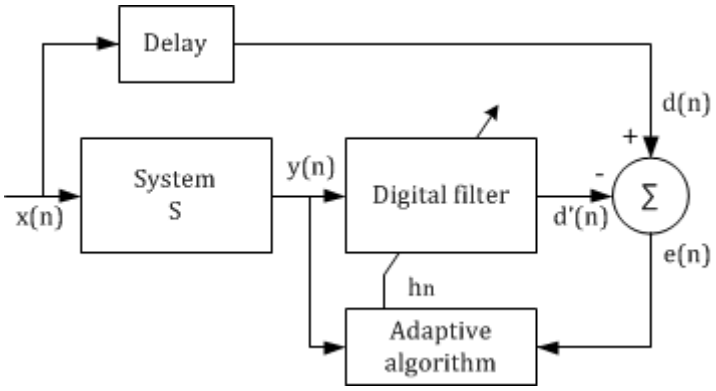


Figure 14 – Schematic of the single channel adaptive inverse filtering.

The idea behind this is to minimize a cost function C by appropriately selecting the filter coefficients h_n and updating the filter as new data arrives. This cost function is always related to the error signal, as we try to minimize it. The adaptive algorithm has always the form of equation (18), with Δh the correction factor.

$$h(n+1) = h(n) + \Delta h(e(n), \dots) \tag{18}$$

Now the problem with this actual layout is that the filter is placed after the unknown system. In our application, the unknown system is formed by the loudspeaker, the room and the microphone. This means the effect of the filter will not be heard by a listener. In order to do that, we would need to stop the filtering process, and copy the filter coefficients to a new filter placed before the unknown system.

But there is a layout where the adaptive filter is placed before the system. This is the so-called filtered-x (Fx) algorithm, illustrated in Figure 15.

Since we still need a signal processed by the system S to inverse, an estimated model S' is used to feed the algorithm. The algorithm is robust to errors in the estimate of the system [29, 30].

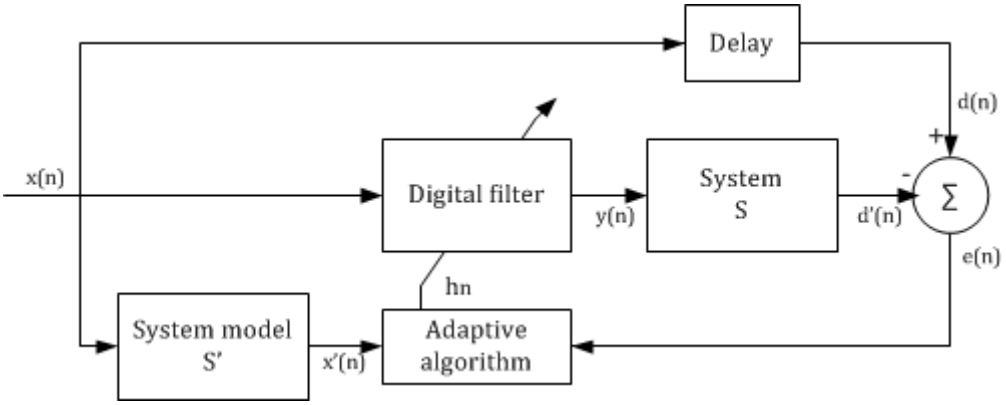


Figure 15 - Schematic of the single channel filtered-x adaptive inverse filtering.

The filtered-x layout has been widely used for active noise control system [29-32] but also for more conventional channel equalization [9, 31].

2.4.2 Multichannel adaptive inverse filtering

A common approach for adaptive multichannel filtering is to minimize the sum of the square of the errors between the equalized responses [7, 9, 30]. Figure 16 presents it with the filtered-x layout.

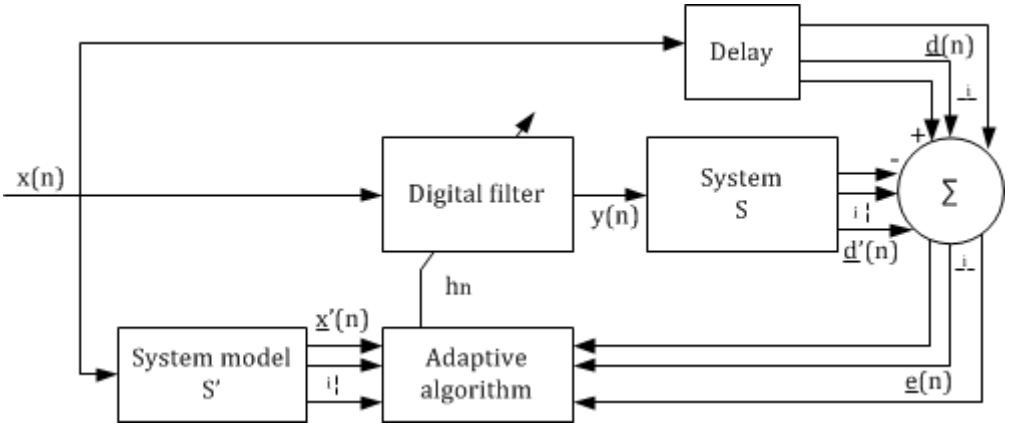


Figure 16 - Schematic of the multichannel filtered-x adaptive inverse filtering.

We have one input (i.e. loudspeaker) and i output (i.e. microphones) of the unknown system S, thus i modelling delay, i error signals and i estimated signals.

The update equation will then be of the form [30]:

$$h(n+1) = h(n) + \sum_{i} \Delta h_i(e_i(n), x_i'(n), \dots) \tag{19}$$

2.4.3 Algorithms for adaptive filtering

Adaptive algorithms have been extensively studied in the past few decades and have been widely used in many fields [33]. The most popular adaptive algorithms are the Recursive Least Square (RLS) algorithm and the stochastic gradient (SG) algorithms.

But before continuing, here are some definitions of symbols used:

$$x(n) = [x(n), x(n-1), \dots x(n-N+1)]^T \tag{20}$$

$$h(n) = [h_0(n), h_1(n), \dots, h_{N-1}(n)]^T \tag{21}$$

$$d'(n) = h(n)\, y^T(n) \tag{22}$$

$$e(n) = d(n) - d'(n) \tag{23}$$

Note that the equations referred to the conventional single channel adaptive inverse filtering layout as we can see Figure 14. To apply the filtered-x layout, the signal $y(n)$ must be replaced by $x'(n)$ in the update equations and equation (22) must be replaced by:

$$y(n) = h(n)\, x^T(n) \tag{24}$$

To get the multichannel layout, we can treat each signal separately and use equation (19) as the overall update equation.

2.4.3.1 Recursive Least Squares algorithm

The RLS algorithm uses a least squares (LS) estimate, that is to try to minimize the sum of the squared errors, hence the cost function:

$$C = \sum_{n=0}^{N} \alpha^{N-n}\, e(n)^2 \tag{25}$$

With $0 < \alpha \leq 1$ the forgetting factor which gives exponentially less weight to older samples error. N is the filter order.

The update algorithm is then defined by equations (26) to (28).

$$k(n) = \frac{P(n\ -\ 1)\, y(n)}{\alpha\ +\ y^T(n)\, P(n\ -\ 1)\, y(n)} \tag{26}$$

$$P(n)\ = \frac{P(n\ -\ 1) -\ P(n\ -\ 1)\, y^T(n)\, k(n)}{\alpha} \tag{27}$$

$$h(n+1) =\ h(n) +\ \ e(n)\, k(n) \tag{28}$$

Where ^{T} indicates matrix transposition.

The initial conditions are:

$$\begin{aligned} h_0 &= 0 \\ P(0) &= \frac{1}{\delta}\, I_N \end{aligned} \tag{29}$$

With δ a small positive number and I_N the N-rank identity matrix.

The RLS algorithm is known to have a fast convergence but a high computational load [18, 33, 34].

2.4.3.2 Stochastic gradient algorithms

The stochastic gradient algorithms use the steepest descent (or gradient descent) method in order to find the Wiener optimum, i.e. to minimize the mean-square error (MSE), hence the cost function [18]:

$$C =\ E\{\mathrm{e}^2(\mathrm{n})\} \tag{30}$$

With $E\{\cdot\}$ denoting the expectation.

This leads to the well-known least mean-squares (LMS) algorithm defined by:

$$h(n+1) =\ h(n)\ +\ 2\mu\, y(n)e(n) \tag{31}$$

With μ a small positive constant called the step size (or sometime learning rate or convergence factor). Its value is strongly dependent to the input signal (for instance a white noise input signal could allow to use a higher step size value than a pop-rock music signal). A small value ensures a result closer to the optimum, but slows down the algorithm. The convergence of the algorithm depends of μ as well, as a too large value will misses the optimum and completely diverge. So we have to be careful when choosing the step size value.

The stochastic gradient algorithms are known to have a lower computational load but a slower convergence than the RLS one [18, 33-36].

Based on the LMS algorithm, several variants have been proposed [33-37] and differ by their update equation.

2.4.3.2.1 The Normalized LMS (NLMS) algorithm

The main drawback of the LMS algorithm is that it is sensitive to the input signal type, and therefore it can be really hard, if not impossible, to find one suitable for every type. The NLMS algorithm solves that problem by normalizing the signal power, hence the new update equation [36]:

$$h(n+1) = h(n) + \mu \frac{y(n)e(n)}{\|y(n)\|^2 + \delta} \quad (32)$$

With δ a small value to avoid division by 0.

The step size is now normalized: $0 < \mu < 2$.

2.4.3.2.2 The Time-Varying LMS (TV-LMS) algorithm

The TV-LMS algorithm works in the same manner as the LMS algorithm, except for the convergence factor, which became time-dependant: it starts from a large value and decrease to a smaller value. This allows to have a fast convergence at the beginning, and then to have a smaller and more precise convergence towards the end. The step size is now expressed as [33]:

$$\mu_n = \mu_0 A^{\frac{1}{1+\alpha n^\beta}} \quad (33)$$

Where A, α and β are positive constants that will determine the magnitude and the rate of decrease for the convergence factor, see Figure 17 for an example. The second term starts from A and converges to 1.

According to equation (9), A has to be larger than 1. If $A = 1$, the learning rate is no longer time-dependant and we fall back on the conventional LMS algorithm.

Note that μ_0 can be normalized as well, like in equation (32): $\mu_0 = \frac{1}{\|y(n)\|^2 + \delta}$.

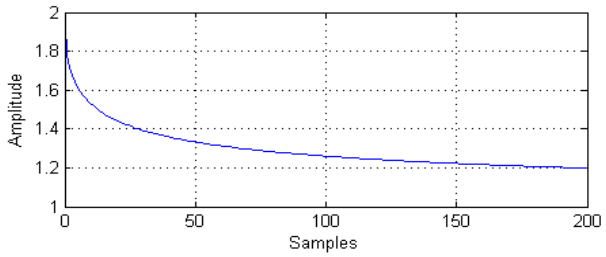


Figure 17 - Example of the evolution of the time-dependant step size value over time ($\mu_0 = 1$, $A = 2$, $\alpha = 0.2$ and $\beta = 0.5$).

The new update equation is then:

$$h(n+1) = h(n) + \mu_n y(n)e(n) \quad (34)$$

2.4.3.2.3 The Normalized Least Mean p-Norm (NLMP) algorithm

The LMP algorithm is the p-norm generalization of the LMS algorithm for α-stable signal[6], hence the generalized cost function [37]:

$$C = E\{|\mathrm{e(n)}|^p\} \tag{35}$$

With p a strictly positive constant, usually $0 < p \leq \alpha$.

This cost function yield to a new generalized update equation:

$$h(n+1) = h(n) + \mu \frac{|e(n)|^{p-1} sgn\big(e(n)\big)}{\|y(n)\|_p^p + \delta} y(n) \tag{36}$$

With $sgn(\cdot)$ the sign function, and δ a small value to avoid division by 0.

Note that if $p = 2$, the 2-norm NLMP algorithm is reduce to the usual NLMS algorithm. On the other hand, if $p = 1$, we obtain the Normalized Least Mean Absolute Deviation (NLMAD) algorithm with the update equation:

$$h(n+1) = h(n) + \mu \frac{sgn\big(e(n)\big)}{\|y(n)\|_1 + \delta} y(n) \tag{37}$$

For our test we use $p = 1.5$ (to have a value half-way from NLMAD to NLMS).

[6] α-stable characterize the type of probability distribution of the signal. α values close to 0 indicate impulsive nature and α values close to 2 indicate a more Gaussian type of behaviour. There is two special values: the $\alpha = 1$ and 2 cases correspond to the Cauchy and Gaussian distributions respectively

3 Methods

This study has been conducted in two times: simulations with Matlab and then real measurements in an anechoic room.

3.1 Simulations

3.1.1 Matlab

Matlab (for *ma*trix *lab*oratory) is a programming environment developed by MathWorks[7] for algorithm development, data analysis, visualization, and numerical computation. Chalmers has unlimited site licence for this software. The simulations were performed using the R2012a version of Matlab.

We also used the Matlab Simulink tool. Simulink is an environment for multidomain (continuous or discrete-time) simulation and Model-Based Design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries to design, simulate, implement, and test a variety of time-varying systems (e.g. communications, signal/image processing …). The version used for this study is 7.9.

Matlab scripts and Simulink models can be found in Annexes.

3.1.2 Acoustic model

At the beginning of the study, no loudspeaker model or measured impulse response were available for simulating our system so we needed to find some impulse responses.

3.1.2.1 Simplified multichannel impulse responses

We employed in first place impulse responses used by Elliot and Nelson [9]. They defined four simple impulse responses of 50 coefficients that we can use for our tests.

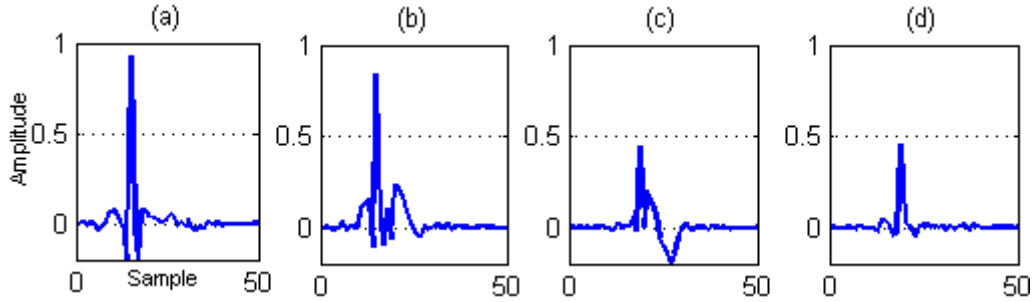


Figure 18 – Simplified impulse responses.

[7] https://www.mathworks.com/

3.1.2.2 Real multichannel impulse responses from MARDY

In order to have more realistic simulations, we needed realistic impulse responses. That is why we used the Multichannel Acoustic Reverberation Database at York[8] (MARDY). This database contains real multichannel room and loudspeaker impulse responses (see Figure 19).

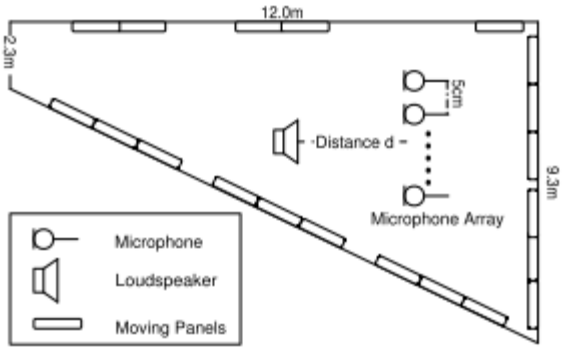


Figure 19 - Diagram of the MARDY recording room dimensions and setup (Room height = 3 m). Both loudspeaker and microphones were elevated to 1m above the floor. Source [38].

The original data contains over 60000 samples per IR but the interesting part is mainly located at the 1000 first samples, as you can see in Figure 20.

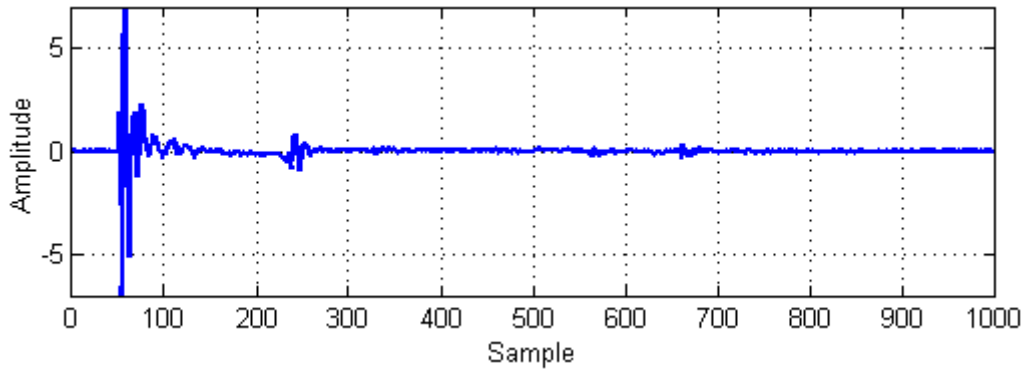


Figure 20 - Real impulse responses from MARDY

We used only the four extreme points for our simulations. It allowed us to simulate the multichannel equalization over an area of $2 \times 0.8\ m^2$, as depicts in Figure 21.

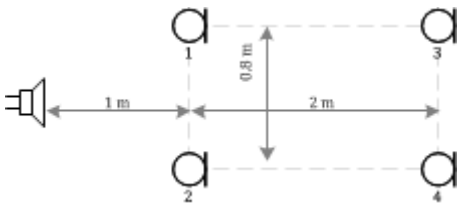


Figure 21 - Points used from MARDY in our simulation.

[8] http://www.commsp.ee.ic.ac.uk/~sap/resources/mardy-multichannel-acoustic-reverberation-database-at-york-database/

3.1.2.3 Nonlinear acoustic model

For the nonlinear simulations, we used in the acoustic path a simple nonlinear system defined by Singh and Chatterjee [39]:

$$y(n) = x(n)^2 + \frac{y(n-1)}{1 + y(n-1)^2} \tag{38}$$

Thus we have a quadratic part $x(n)^2$ and a recursive part decreasing with time.

This nonlinear system is then added to an existing linear system (like we defined before).

3.2 Measurements in laboratory

The Chalmers Department of Applied Acoustics has an anechoic chamber where we can conduct our measurement.

3.2.1 The anechoic chamber

Anechoic rooms are used to mimic free-field conditions, i.e. a condition where only the direct sound field is present. In such rooms one can for example calibrate microphones and measure the frequency response and directivity characteristics of loudspeakers and microphones. The room can also be used to reproduce binaural recordings or simulations via loudspeakers.

In order to have a high sound absorption in an anechoic chamber, the room's bounding surfaces (ceiling, walls and floor) are covered with mineral wool (we walk on a net hanging at the middle of the room). The absorption material is wedge-shaped to achieve an impedance matching, i.e. a slow transition between the specific sound impedance of air and the impedance of the porous absorber. The absorption coefficient and the length of the wedges define the frequency limitation of the room. The anechoic chamber at the Department of Applied Acoustics has a volume of $10 \times 10 \times 8$ m^3 and a lower frequency limit of $f_L = 75\ Hz$.

An anechoic chamber should also be well isolated to vibration and sound since measurements often are made at low levels (e.g. distortion measurements on loudspeakers). This requirement can be fulfilled by using a double shell construction. For example, the anechoic room is built as a box, physically separated from the rest of the building. Vibrations from the ground are eliminated by mounting the box on steel coil springs. The anechoic measurement room at the Department of Applied Acoustics weighs about 800 tons and has a resonance frequency of $f_0 \approx 10$ Hz.

As one can guess from the explanation above, an anechoic room with reasonable measurement conditions is extremely expensive.

3.2.2 The measurement setup

Measurements were conduct on Windows XP using HOLMImpulse, a freeware program developed by Holm Acoustics[9] for frequency and impulse response measurement. The sine sweep technique was used (see chapter 2.2.3 for explanation).

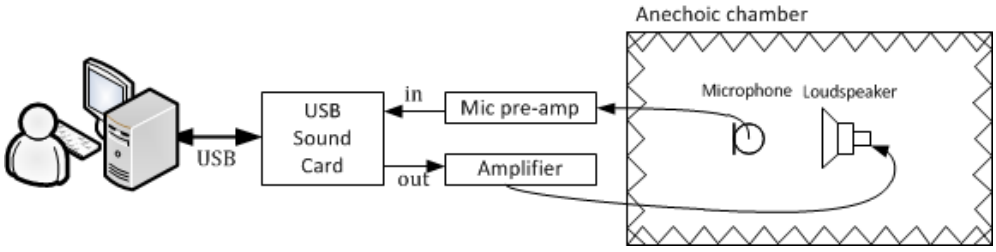


Figure 22 - The measurement system

Harmonic distortion measurements using pure tone was conduct with the trial version of the software DSSF3 by Ymec[10].

We used an external USB soundcard with 48 kHz sampling and an omnidirectional electret condenser microphone (see Table 1 for specifications).

Table 1 - Microphone WM-063 specifications

Sensitivity	-62 ± 3dB
Frequency range	$20 - 20\,000$ Hz
SNR	> 38 dB

3.2.3 Calibrations

There is of course no good measure without a proper calibration of the instruments.

The first step is to is to calibrate the microphone. For this, we need to look at what is given by the manufacturer. Only a typical frequency response is given in the datasheet, as you can see Figure 23. It seems pretty flat all over the desired frequency range, with only small ripples less than $\pm$ 3 dB above 5 kHz. For our measure, we can consider it good enough.

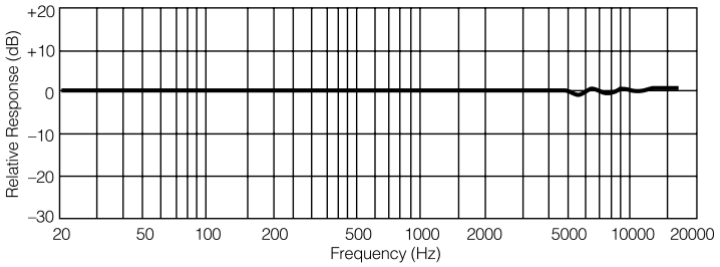


Figure 23- Typical frequency response curve of the microphone.

For the next step we calibrate the soundcard by measuring and adjusting the loopback, i.e. measuring between the output and the input. As we define in Chapter 2.3, a digital

[9] http://www.holmacoustics.com
[10] http://www.ymec.com

system has converters to switch from continuous-time to discrete-time signal (and vice-versa). Those converters (DAC and ADC) can modify the signal and therefore need calibration.

As you can see Figure 24, not much calibration was necessary for our soundcard since it is already pretty flat.

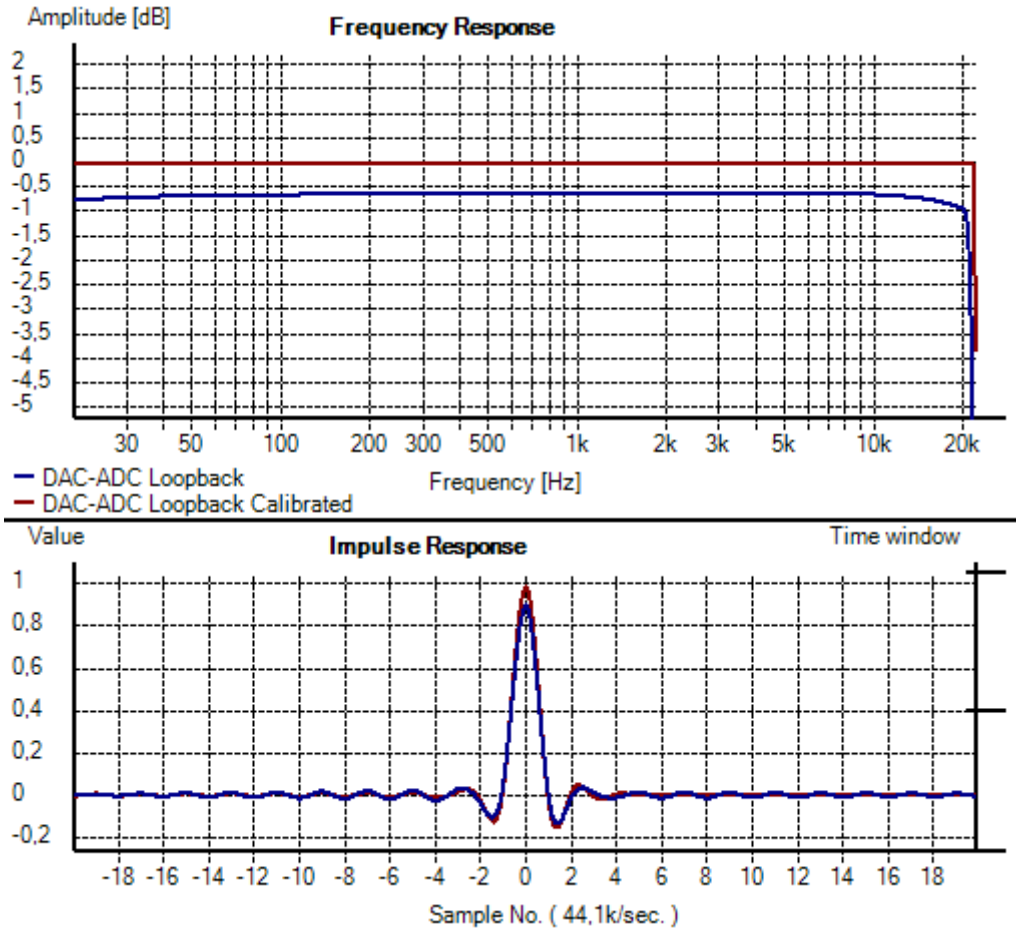


Figure 24 - Soundcard frequency and impulse response before and after calibration.

3.2.4 The measurements

First, the impulse response has been measured at various points around the loudspeaker as you can see in Figure 25. The loudspeaker was installed on a wooden square plank of 1.25 m^2 for reflection purpose.

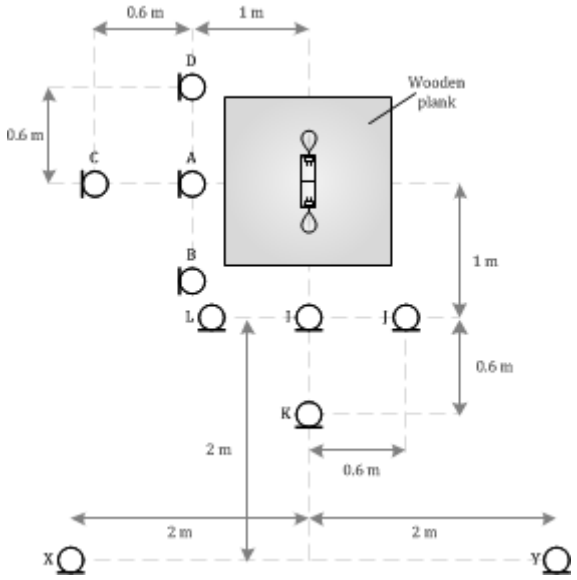


Figure 25 – Measurement points ABCD – IJKL – XY. Microphones are 1m above the loudspeaker.

Then we measured the directivity pattern of the loudspeaker on two positions (side and front), see Figure 26. We used an automatic turntable to orient the loudspeaker.

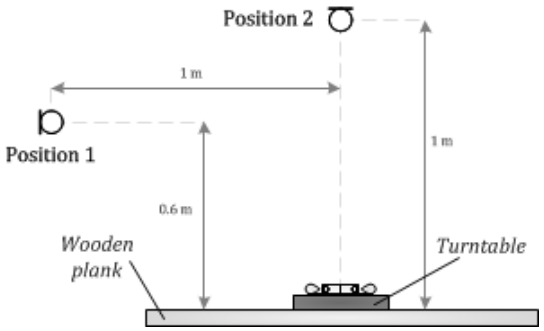


Figure 26 - Positions for the directivity measurement.

4 Results

The results of the various simulations and measurements are presented in this chapter and will be discussed in the next one.

4.1 Simulations

The simulations were conduct in three steps: adaptive algorithms simulations, linear simulations and nonlinear simulations.

4.1.1 Adaptive algorithms

The first simulations concern all the adaptive algorithms defined in Chapter 2.4.3: LMS, NLMS, TV-LMS, NLMAD, NLMP and RLS. The goal is to find the most suitable algorithm for our needs.

Figure 27 plots the convergence of the algorithms by looking at the evolution of the mean squared error (MSE) over time. The MSE for each algorithm is calculated as equation (39). It shows how quickly an algorithm can converge to a stable and low value. We see that the RLS algorithm is the best in this category: fast convergence to an extremely low value.

$$MSE = \frac{1}{N}\sum_{n=0}^{N} e(n)^2 \tag{39}$$

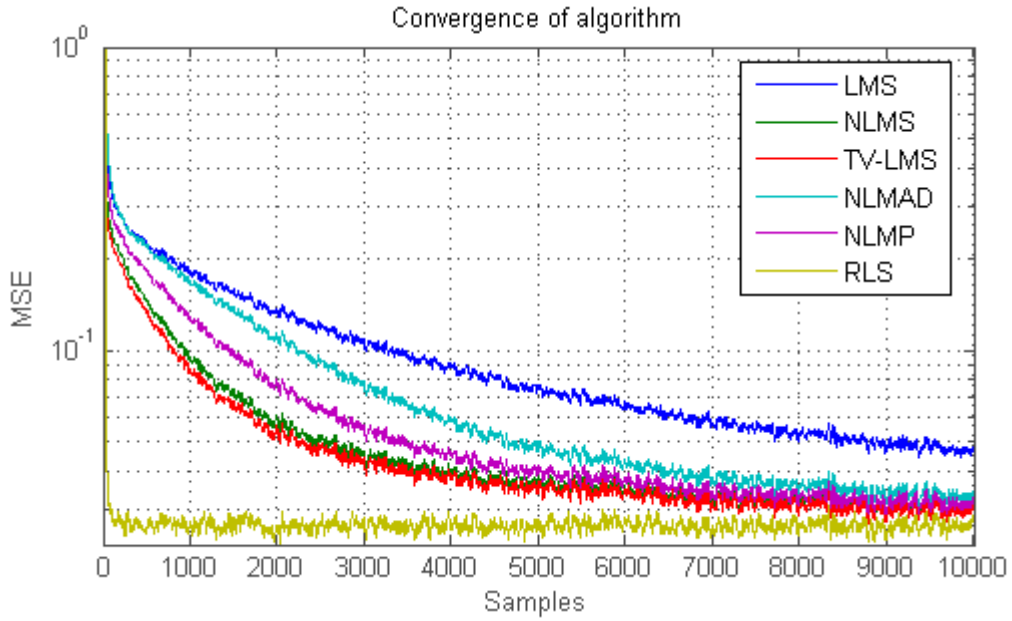


Figure 27 – Learning curves for various algorithms
(1000-tap MARDY impulse response, 300-tap filter length, 100-tap delay, and no noise).

The next simulations used the simplified impulse response of 50 coefficients in order to accelerate the simulations for each point. We remind the reader that the signal-to-

noise ratio is defined as the ratio between a signal power and the noise power, see equation (40).

$$SNR = \frac{P_{Signal}}{P_{Noise}} \tag{40}$$

Figure 28 and Figure 29 show the MSE evolution for different SNR and different filter length respectively. No algorithm really stands out; they all have the same sensibility to noise and filter length.

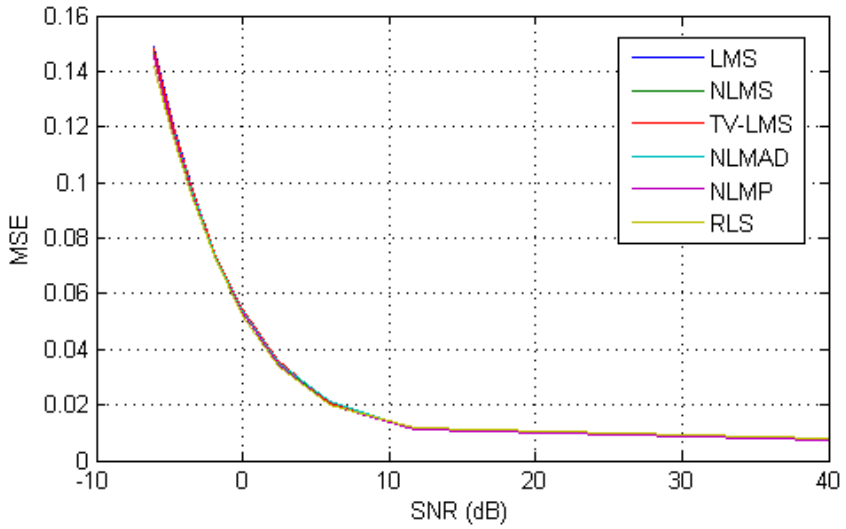


Figure 28 – MSE performance for various signal-to-noise ratios (50-tap impulse response, 50-tap filter length, and 25-tap delay).

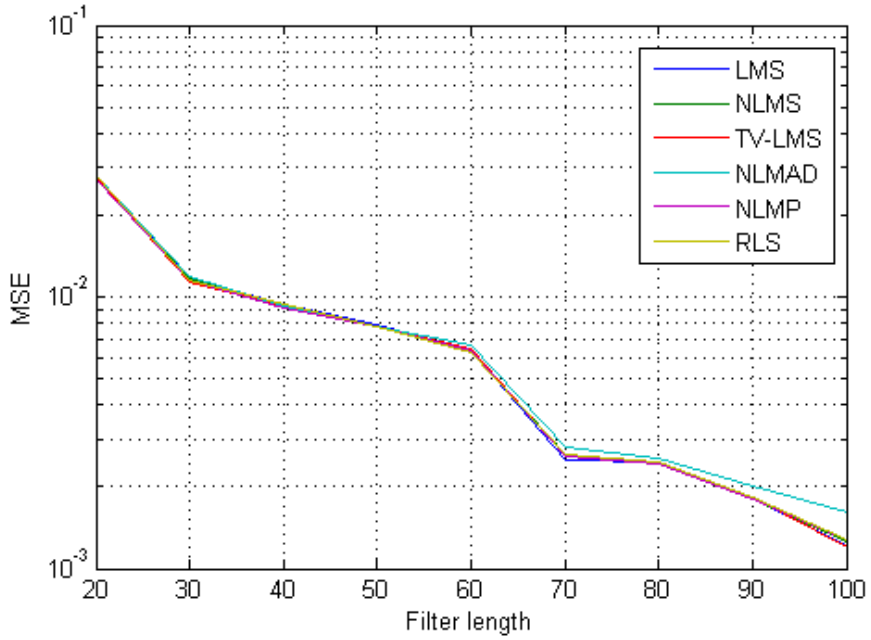


Figure 29 - MSE performance for various filter length (50-tap impulse response, 25-tap delay, and no noise).

For the last one, Figure 30 shows the computation time of the algorithms for increasing filter length. This is an important parameter for the choice of the algorithm. And we clearly see that the RLS algorithm take a lot of increasing computation time whereas the other ones stay quite low.

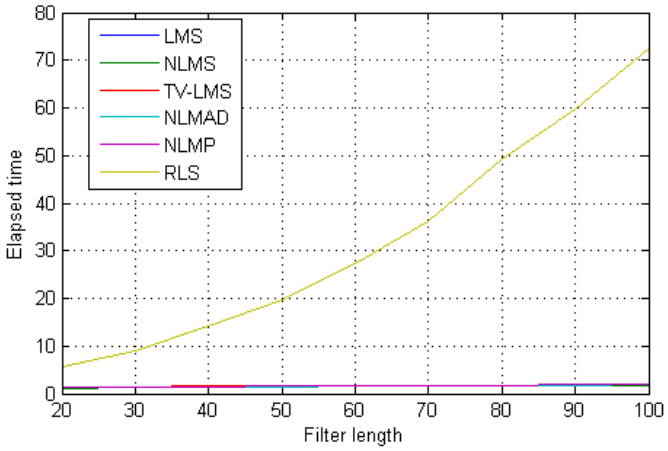


Figure 30 – Elapsed time for various filter length (50-tap impulse response, 25-tap delay, no noise).

4.1.2 Linear implementation

The filtered-x implementation of the NLMS algorithm was used for the linear implementation, since it offers the best trade-off between convergence speed and computation time. This choice is extensively detailed in Chapter 5.1.

4.1.2.1 Single channel

We started with the single channel equalization as described in Chapter 2.4.1.

4.1.2.1.1 Channel equalization

We looked at the channel equalization performance by simulating with an optimal delay and no noise. Figure 31 shows the acoustic path impulse response and the corresponding inverse filter we generate. Figure 32 shows the convolution between them. The equalization is effectively quite good and we almost get a perfect IR. Finally, Figure 33 gives their respective frequency spectrums. We see a big improvements for the magnitude spectrum (except in the low frequencies, due to the size of the filter).

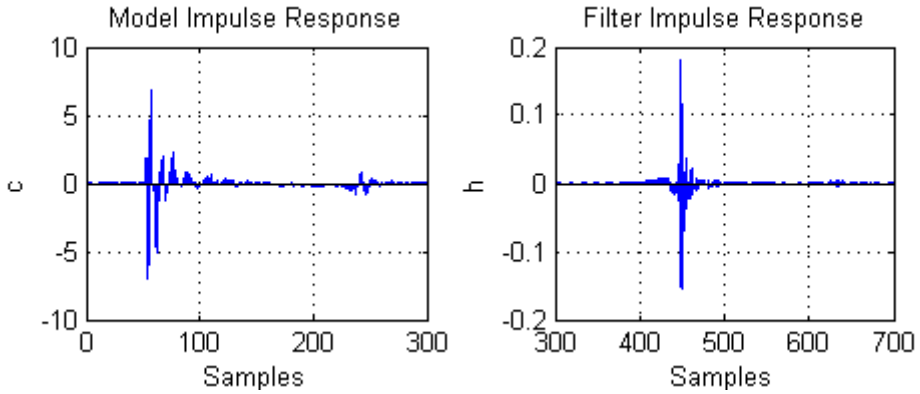


Figure 31 – Left is 1000-tap MARDY impulse response (zoom on the first 300 samples) and right is 1000-tap filter impulse response (zoom on 300 to 700 samples). (150-tap delay and no noise).

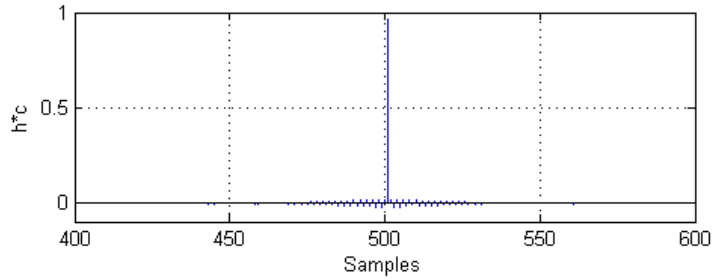


Figure 32 – Resulting impulse response acoustic path + filter (zoom on 400 to 600 samples) (1000-tap MARDY impulse response, 1000-tap filter length, 150-tap delay, and no noise).

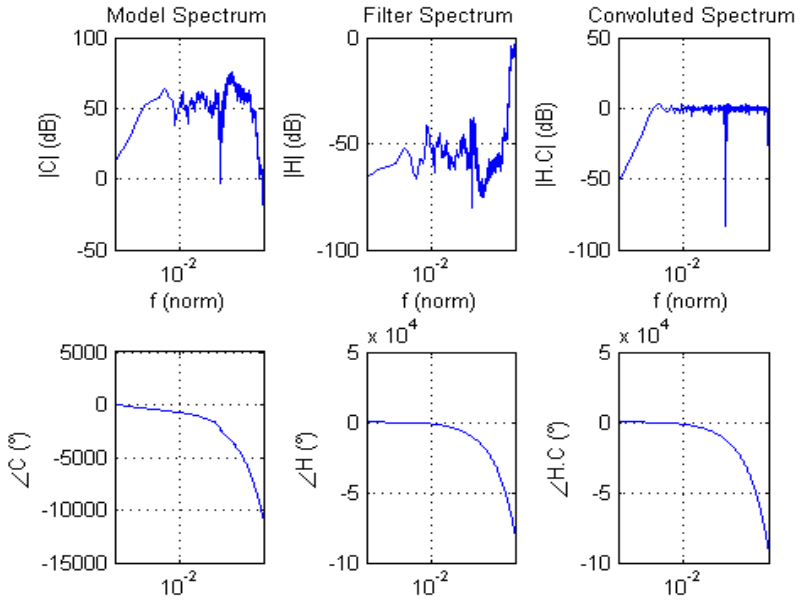


Figure 33 – Frequency spectrums (magnitude and phase) of the various blocs: acoustic path (or Model), filter and model + filter (1000-tap MARDY impulse response, 1000-tap filter length, 150-tap delay, and no noise).

4.1.2.1.2 Influences on the delayed path

Next, we focused on the parameters influencing the algorithm in the delayed path. Figure 34 shows how the delay for $x(n)$ affects the MSE convergence. We can understand that there is a minimum delay which corresponds roughly at the first peak position in the acoustic path impulse response (which is around 50 for this one). This will be a little more detailed with the multichannel simulation.

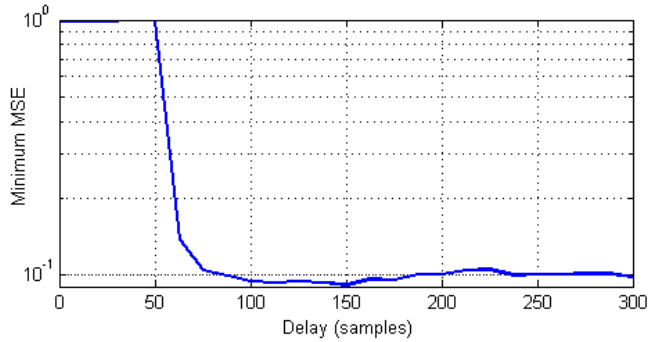


Figure 34 – Minimum MSE for various delay in the forward path. (1000-tap MARDY impulse response, 1000-tap filter length, and no noise).

4.1.2.1.3 Influences on the estimated path

It could be interesting to know how good the estimated path has to be (S' in Figure 15). We start by adding some white noise to see if the convergence is impacted, see Figure 35. We notice that even with a SNR of -20 dB (i.e. the signal is ten times weaker than the noise) the convergence is not really impacted. This means that we don't need to have a very precise model for the estimated path.

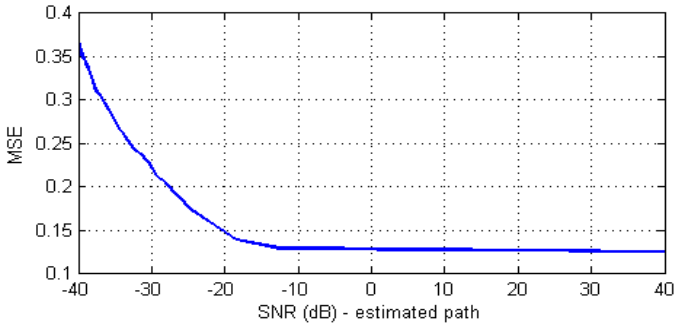


Figure 35 - MSE performance for various SNR in the estimated path.
(1000-tap MARDY impulse response, 1000-tap filter length, 150-tap delay, and no noise).

Then we add more delay to the path and see how the convergence goes, see Figure 36. Here we observe that even for a small variation of the delay, the system is hugely impacted and diverge completely. This means the delay is what matter the most for the quality of the estimated path.

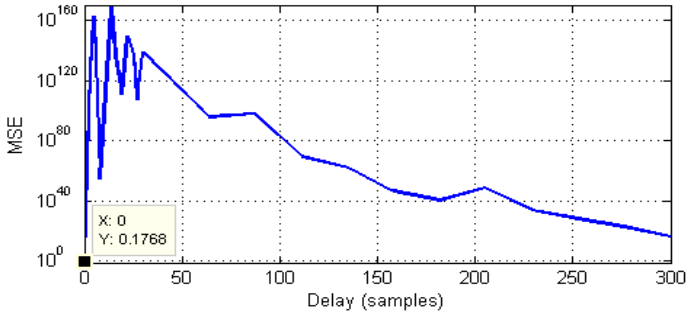


Figure 36 - MSE performance for various extra delay in the estimated path.
(1000-tap MARDY impulse response, 1000-tap filter length, 150-tap delay, and no noise).

4.1.2.1.4 Single channel equalization on multichannel path

It can be interesting to look at why multichannel equalization could be useful instead of single channel. For that, we equalize a single point (number 1) and look at how three other points in the room (2 to 4) are influenced by this filtering. Indeed, the first channel is well equalized, but the other channels don't benefit from it.

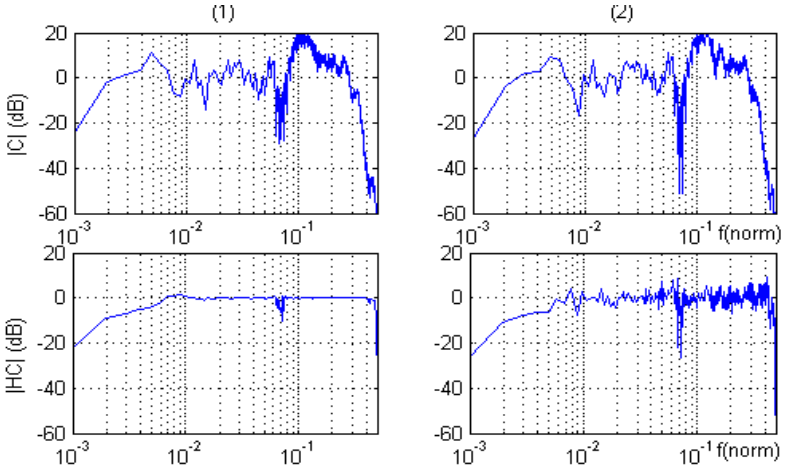


Figure 37 - Spectrums of acoustic path before and after single channel equalization for point 1 and 2. (1000-tap MARDY impulse response, 1000-tap filter length, 150-tap delay, and no noise).

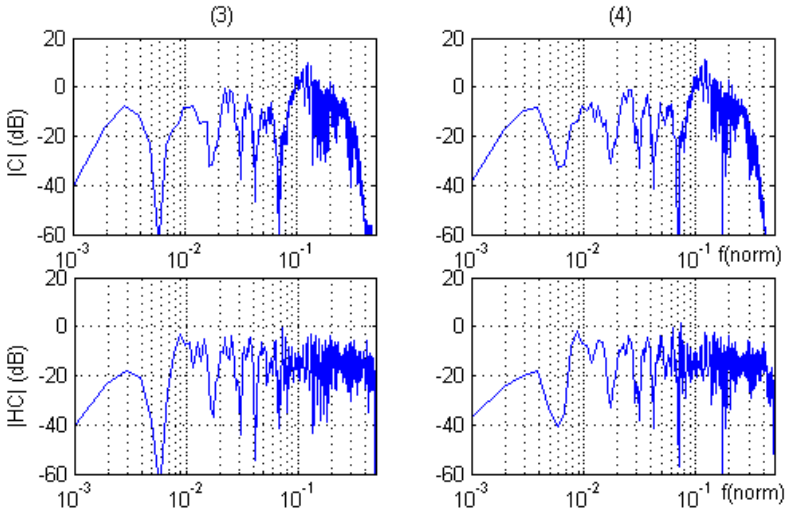


Figure 38 - Spectrums of acoustic path before and after single channel equalization for point 3 and 4. (1000-tap MARDY impulse response, 1000-tap filter length, 150-tap delay, and no noise).

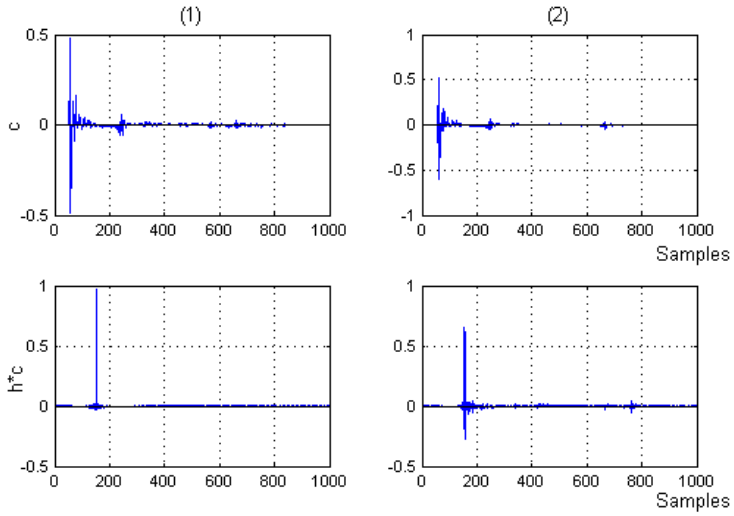


Figure 39 – Impulse responses of acoustic path before and after equalization for point 1 and 2. (1000-tap MARDY impulse response, 1000-tap filter length, 150-tap delay, and no noise).

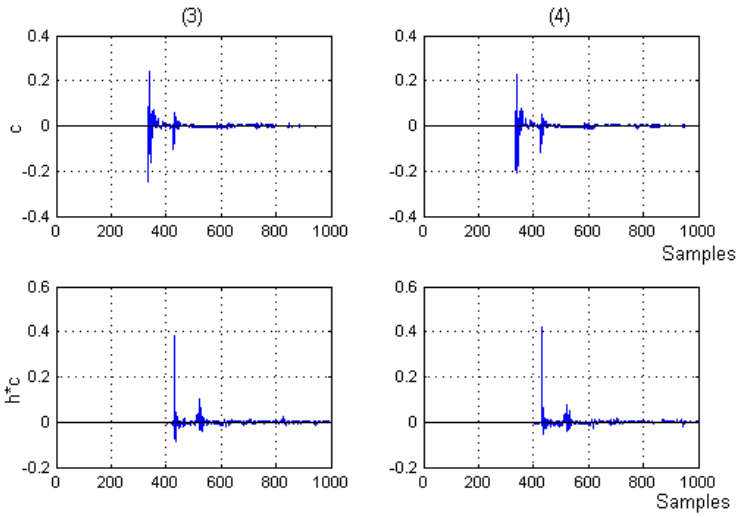


Figure 40 - Impulse responses of acoustic path before and after equalization for point 3 and 4. (1000-tap MARDY impulse response, 1000-tap filter length, 150-tap delay, and no noise).

4.1.2.2 Multichannel

We continue with the multichannel equalization as detailed in Chapter 2.4.2.

4.1.2.2.1 Find the optimal delay

The main problem when dealing with multichannel part is to choose the values for the delayed path (i.e. from $x(n)$ to $d(n)$, see in Figure 16). It turns out that the delay value of each path is linked to the others, and a bad choice for only one will limit the convergence. So before moving on well working simulation, we have to define the optimal delays.

To find the relation between those paths is quite easy. One has to look at the index of the first peak of each impulse responses. Then, taking one of those paths as the reference, the subtraction on each indexes gives the relation between them: it's the optimal offset (or delta) for the delays. Table 2 and Table 3 give the values for both of our IR type.

Table 2 – Peaks and Optimal deltas for the delayed path (50-tap simple IR).

	First peak	Optimal delta
Channel 1 (reference)	15	0
Channel 2	15	0
Channel 3	19	4
Channel 4	19	4

Table 3 - Peaks and Optimal deltas for the delayed path (1000-tap MARDY IR).

	First peak	Optimal delta
Channel 1 (reference)	57	0
Channel 2	61	4
Channel 3	338	281
Channel 4	339	282

To illustrate this idea, we use the 50-tap impulse responses with a 25-tap default delay and we vary only one delay value at a time. The results are shown Figure 41.

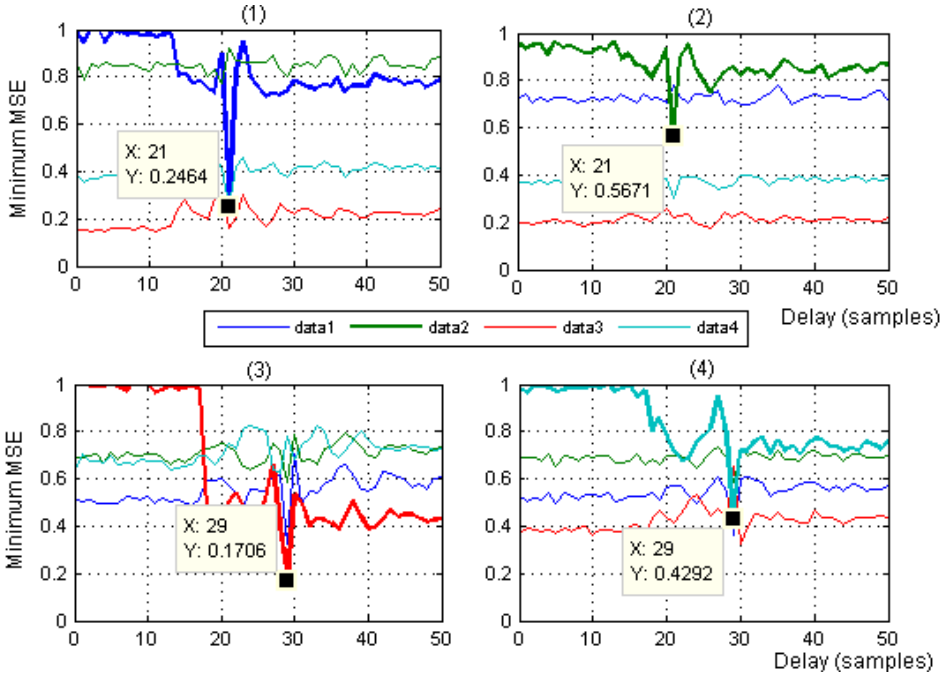


Figure 41 - MMSE performance for single delay variation (only one delay varying at the same time) with 25-tap default delay (50-tap simple impulse response, 50-tap filter length, and no noise).

We can easily see that the lowest MSE values are the ones 4 samples from 25, as calculated in Table 2. Mind that the MSE are not that really low because the optimal offset is not used. We also notice, as we said before, that the delay should be at least higher than the first peak index.

Now if we use those optimal offsets with the MARDY impulse responses and simulate a full delay variation, we get the Figure 42. The working range is then from 100 to 700, i.e. we have to let enough room for the filter impulse response before and after otherwise it gets truncated.

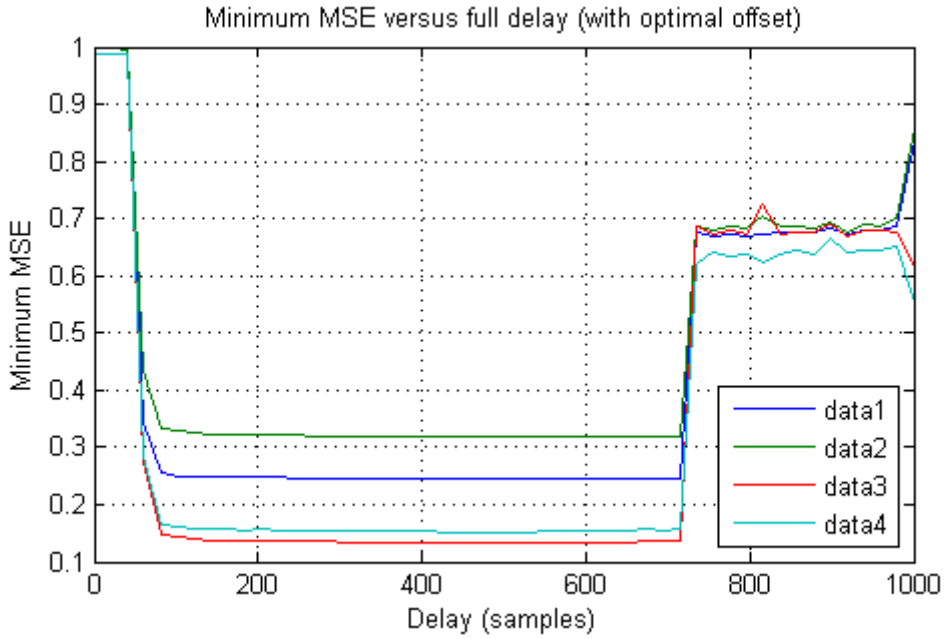


Figure 42 - MMSE performance for full delay variation (all delay varying at the same time) with optimal offset from Table 3 (1000-tap MARDY impulse response, 1000-tap filter length, and no noise).

4.1.2.2.2 Channels equalization

The optimal offset defined, we can equalize the four channels. After simulation is completed, we obtain the filter Figure 43. The effect of this filter on each channel can be seen Figure 44 - Figure 45 (impulse responses) and Figure 46 - Figure 47 (frequency spectrum). We clearly see that all points benefit from the equalization.

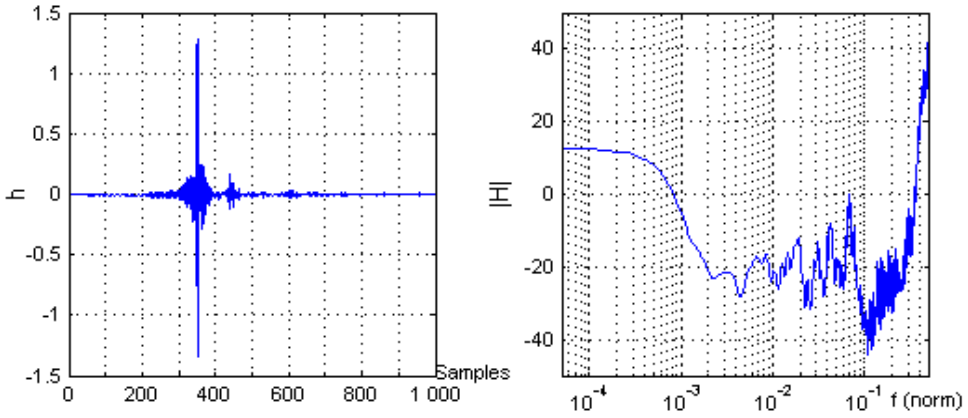


Figure 43 - Filter impulse response and frequency spectrum
(1000-tap MARDY impulse response, 1000-tap filter length, optimal delays, and no noise).

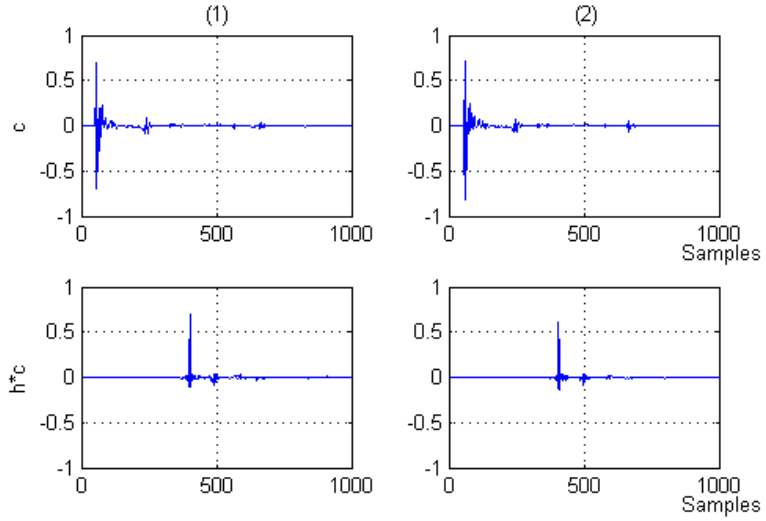


Figure 44 - IR of acoustic path before and after multichannel equalization for point 1 and 2.
(1000-tap MARDY impulse response, 1000-tap filter length, optimal delays, and no noise).

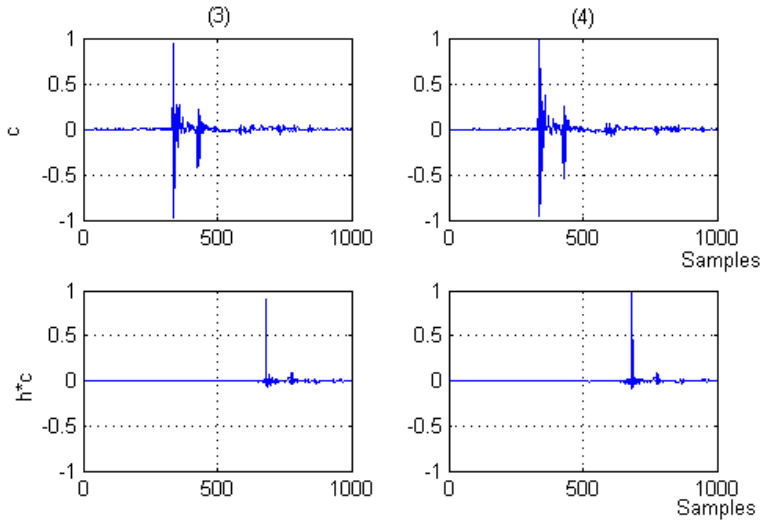


Figure 45 - IR of acoustic path before and after multichannel equalization for point 3 and 4. (1000-tap MARDY impulse response, 1000-tap filter length, optimal delays, and no noise).

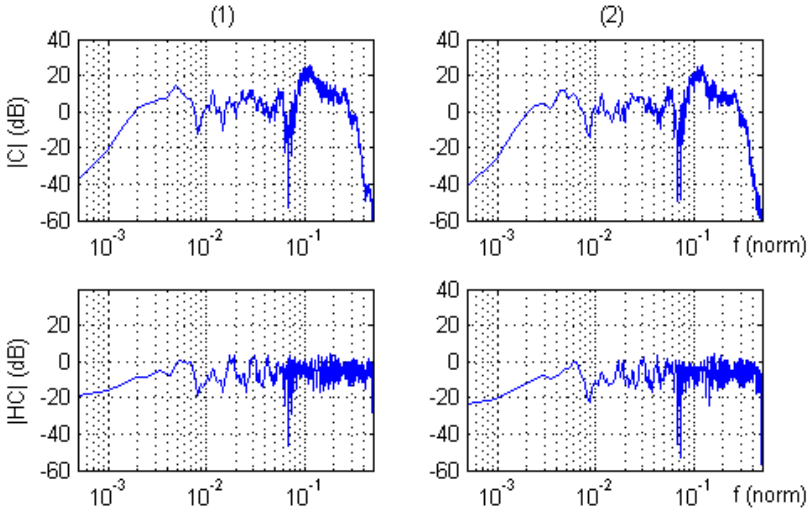


Figure 46 – Frequency spectrum of acoustic path before and after multichannel equalization for point 1 and 2 (1000-tap MARDY impulse response, 1000-tap filter length, optimal delays, and no noise).

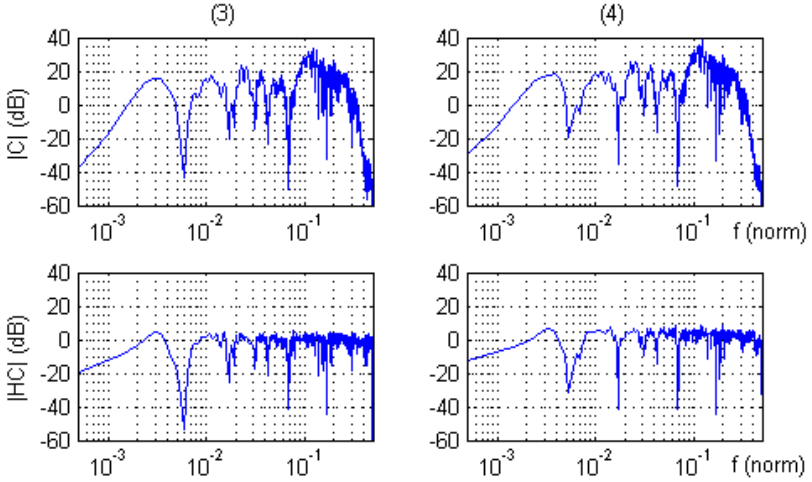


Figure 47 - Frequency spectrum of acoustic path before and after multichannel equalization for point 3 and 4 (1000-tap MARDY impulse response, 1000-tap filter length, optimal delays, and no noise).

4.1.2.2.3 Influence of the number of channels

It is interesting to see how the number of channels influences the equalization process. We simulate the progressive variation of the number of optimization point from one to four.

First, only channel 1 is taken into accounts (from 0 to 1), then channel 2 is progressively added (from 1 to 2), and it goes on until channel 4. Figure 48 plots the results.

We remark that at the beginning, the first channel is well equalized and the others benefit of it a little bit. Then adding the second channel decreases its error value but increase the first one. The same happened when the third and fourth channels are added. This is quite logical: we can't equalize multiple channel as good as only one channel. There will be a trade-off between all channels.

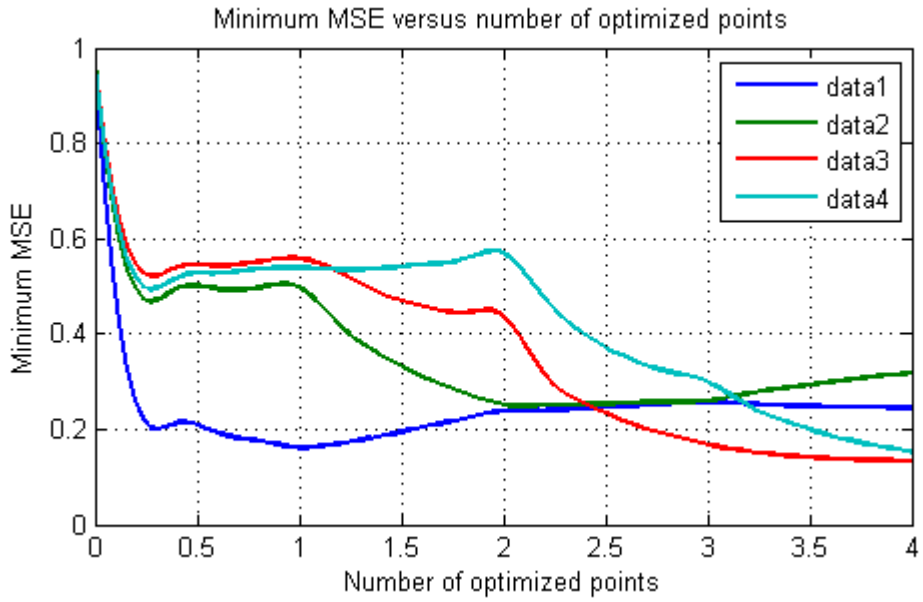


Figure 48 – MMSE performance for increasing number of optimized points (1000-tap MARDY impulse response, 1000-tap filter length, optimal delays, and no noise).

4.1.3 Nonlinear implementation

The second-order Volterra (SOV) implementation of the NLMS algorithm was used for the nonlinear implementation, as defined in Chapter 2.1.2.

With this nonlinear simulation we ran into a main problem: the simulation would always diverge. No matter the parameters, the MSE would basically converge normally then rise to huge values after a certain amount of time, as you can see in Figure 49. Figure 50 and Figure 51 show the filter coefficients diverging.

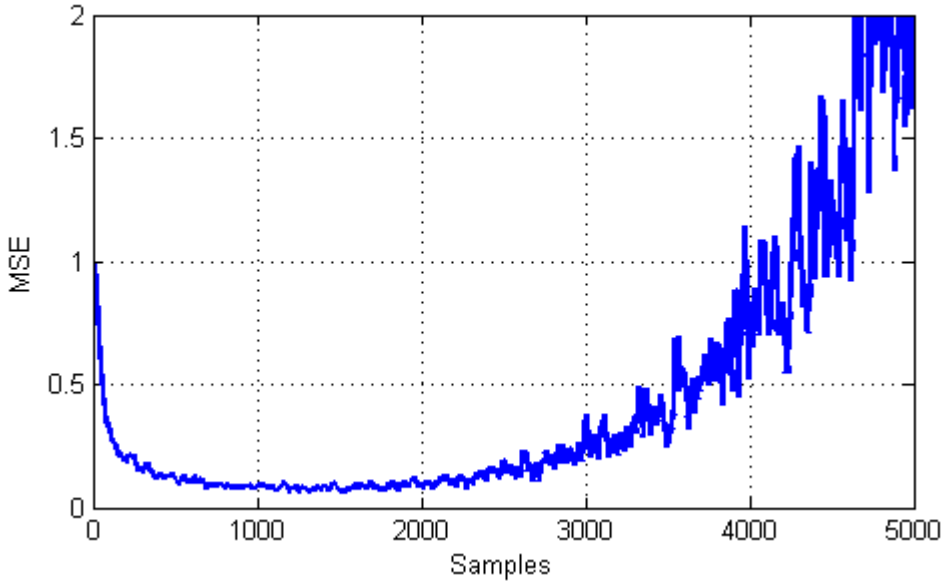


Figure 49 - Learning curves of the SOV filter (50-tap simple impulse response, 50-tap linear filter length, 325-tap SOV filter length, nonlinearities amplitude of 0.5, and no noise).

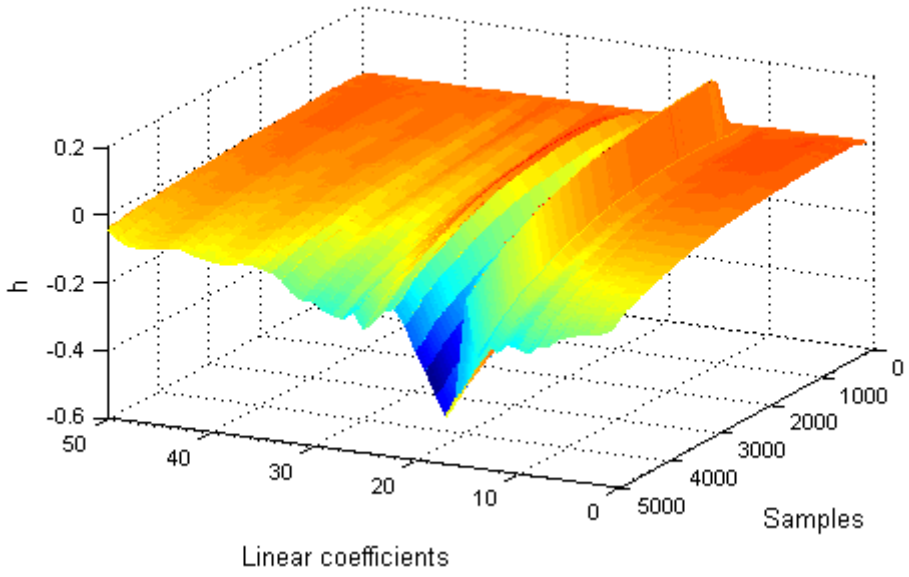


Figure 50 – Linear filter h evolution (50-tap simple impulse response, 50-tap linear filter length, 325-tap SOV filter length, nonlinearities amplitude of 0.5, and no noise).

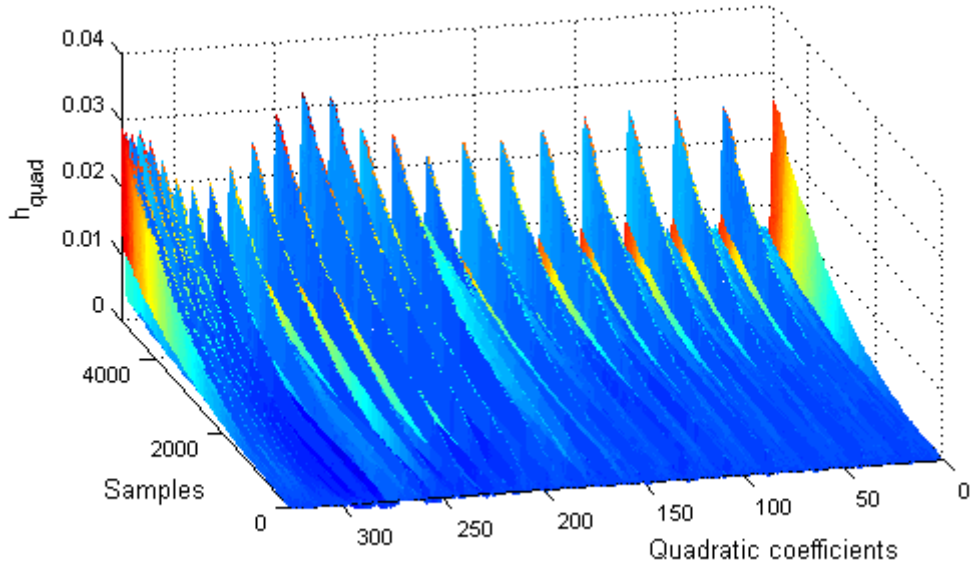


Figure 51 – quadratic filter evolution (50-tap simple impulse response, 50-tap linear filter length, 325-tap SOV filter length, nonlinearities amplitude of 0.5, and no noise).

Finally, Figure 52 shows the difference whether the Volterra filter is enabled or not. The reason why we have this divergence will be discussed next chapter.

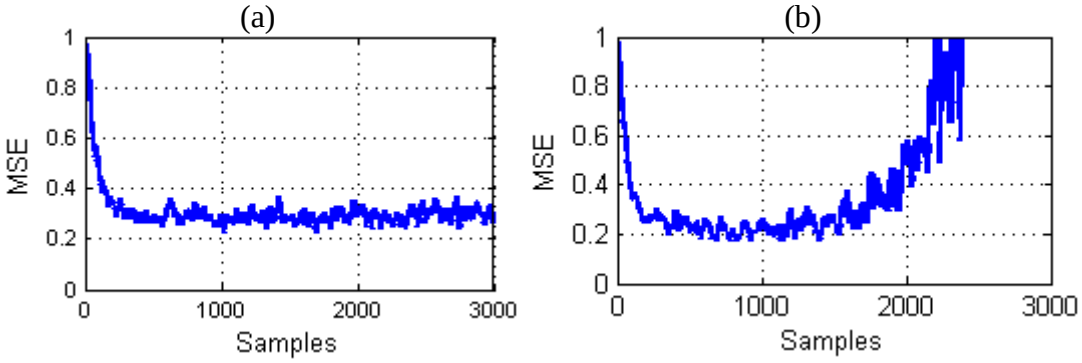


Figure 52 – Learning curves with (a) Volterra filter not enabled and (b) Volterra filter enabled (50-tap simple impulse response, 50-tap linear filter length, 325-tap SOV filter length, nonlinearities amplitude of 1, and no noise).

4.2 Loudspeaker characterization

Measurements in the laboratory were conducted as explained in Chapter 3.2.

4.2.1 Impulse and frequency responses

Impulse and frequency responses were measured in various positions. Figure 53 shows the two positions A and I (difference of 90° degrees between them).

We can see that the loudspeaker lacks of bass power. This is quite logical looking at the size of the drivers. In any way, the frequency response is not very flat and could be improved.

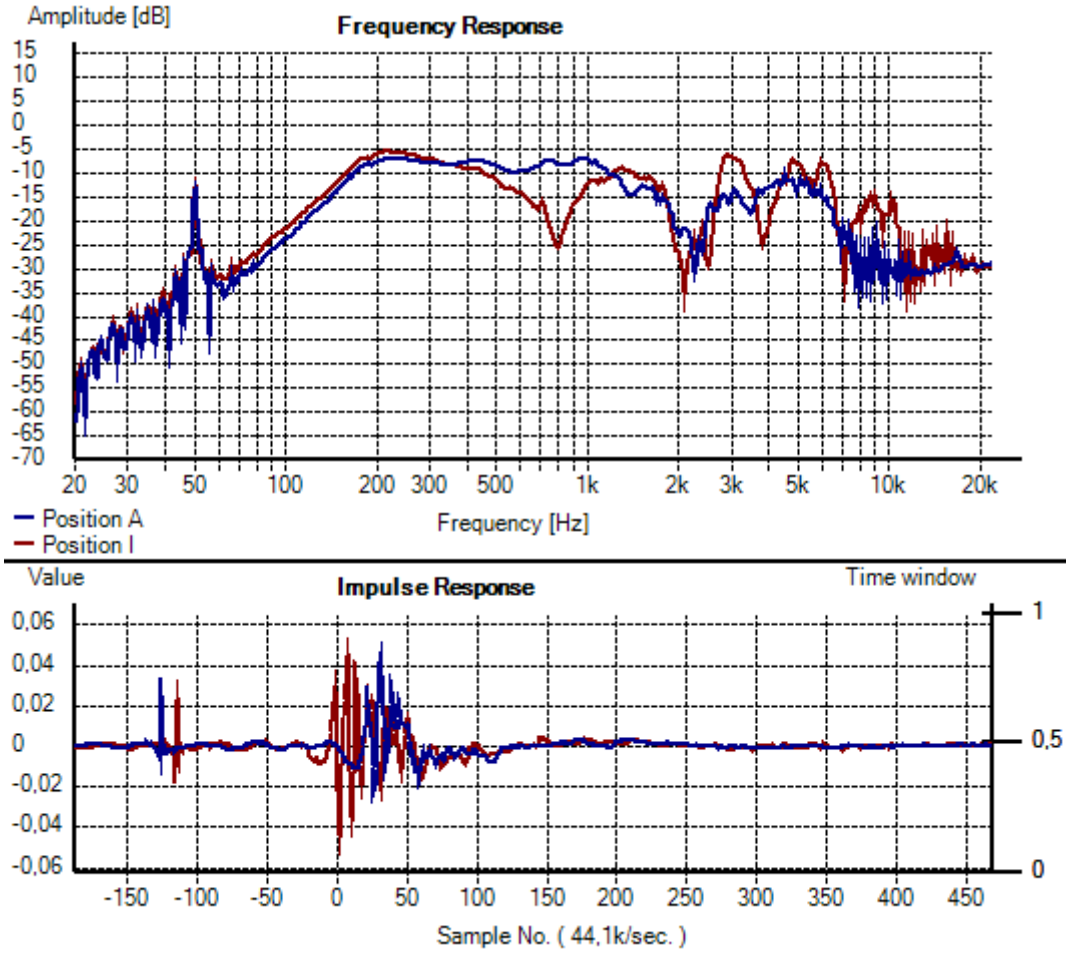


Figure 53 - Frequency and raw impulse responses for position A and I.

4.2.2 Directivity pattern

We created the directivity patterns using the position defined in Figure 26. We obtained the sonograms Figure 54 and Figure 55.

Although the frequency response is not very flat, it is pretty constant over the orientation, meaning that two people right-angled from the loudspeaker will have approximately the same sound experience.

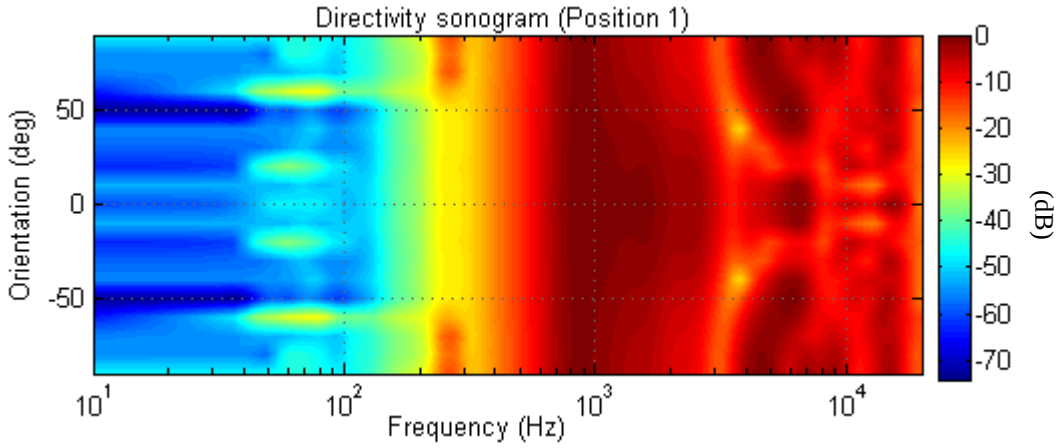


Figure 54 - Directivity sonogram for position 1 (side).

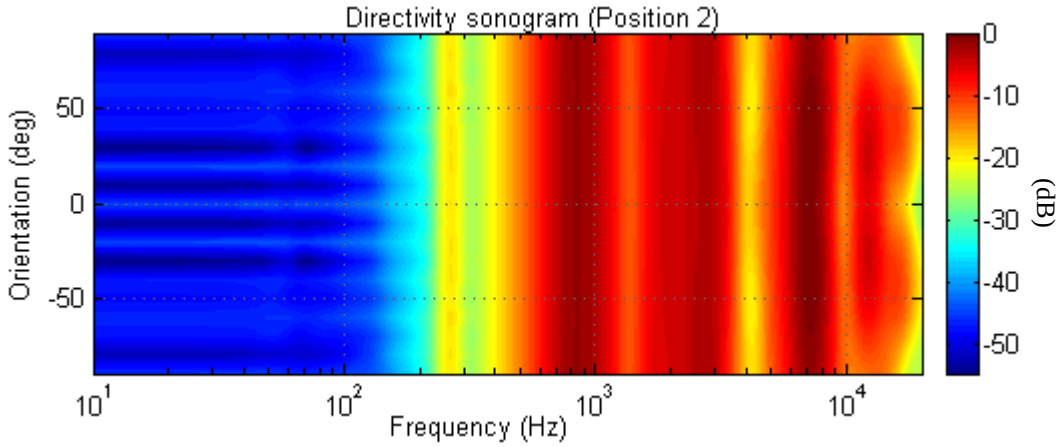


Figure 55 - Directivity sonogram for position 2 (top).

4.2.3 Harmonic distortions

We characterized the amount of harmonic distortion for only one pure tune of 1 kHz. It was more a small test for illustration purpose than a real characterization of the nonlinearities. Figure 56 shows the frequency spectrum. We clearly see the tone at 1 kHz and some harmonics at 2 and 3 kHz. Figure 57 sums up the distortion for each harmonics. The total is less than 1% at 1 kHz, which is quite good.

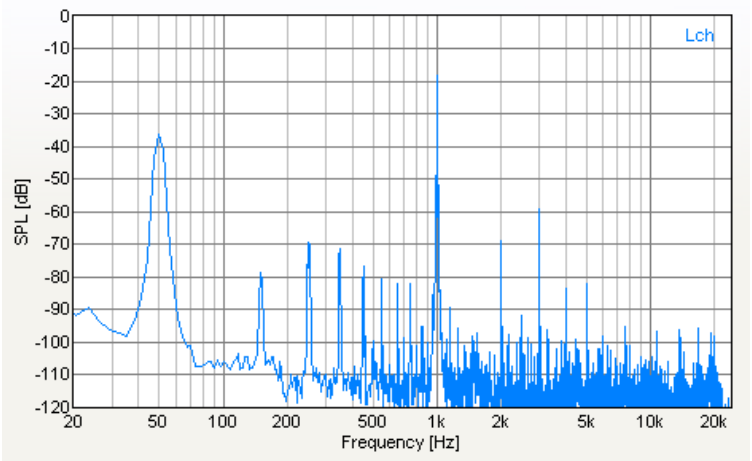


Figure 56 - Frequency response when stimulated with a 1 kHz pure tone.

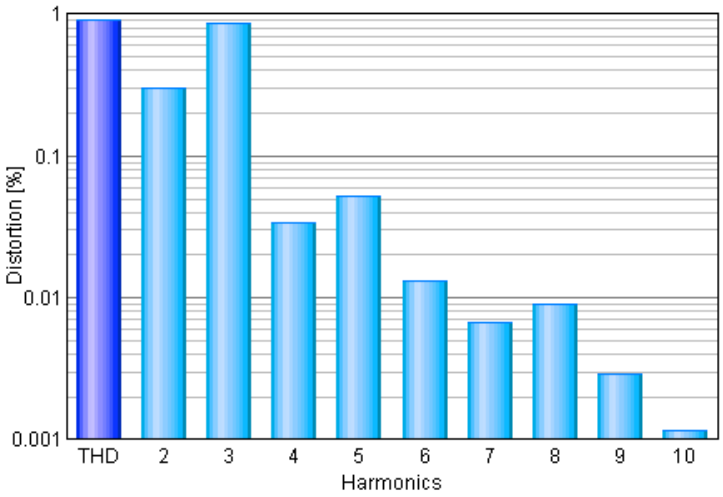


Figure 57 - Harmonics distortions when stimulated with a 1 kHz pure tone.

4.3 Loudspeaker equalization

The next step was to test our algorithm on our specific loudspeaker detailed in Chapter 2.2.2. The best way would have been to insert our filter after the soundcard and install multiple microphones across the room in order to conduct new tests in real time on the loudspeaker. However, due to the lack of time and easy solution we were obliged to perform those tests offline, i.e. to simulate the system using measured impulse responses. These simulations are not perfect, since it doesn't take into account the loudspeaker nonlinearities.

We selected the positions J, L, X and Y, forming a surface of 6.4 m² for equalization (see Figure 25 for exact dimensions).

4.3.1 Optimal delays

The first simulation concerned the optimal offset for the delayed path. Table 4 sums the value gathered.

Next step was to simulate the MSE evolution while varying the delays with optimal offset, allowing us to obtain the working range. The results are plotted in Figure 58, giving a working range between 200 and 500 samples.

Table 4 - Optimal delays for loudspeaker measured IR.

	First peak	Optimal delta
Position J (reference)	156	0
Position L	160	4
Position X	524	368
Position Y	528	372

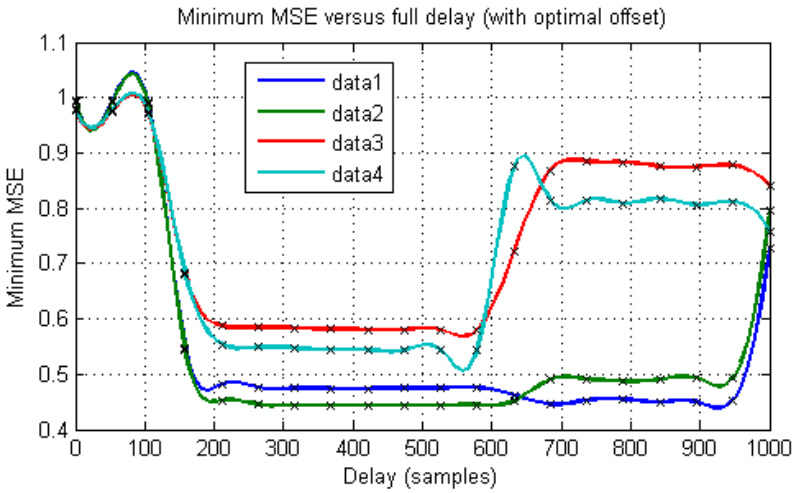


Figure 58 - MMSE performance for full delay variation (all delay varying at the same time) with optimal offset from Table 4 (1000-tap measured impulse response, 1000-tap filter length, and no noise).

4.3.2 Impulse and frequency responses

The optimal delays defined, we went on to equalize the four channels. Figure 59 and Figure 60 plot the acoustic path impulse responses before and after the equalization (zoomed for a better view) whereas Figure 61 and Figure 62 plot the resulting frequency responses.

We can see that the IR are compacted and show less ringing around their main peak. Looking at the FR, we notice that the magnitude become compacted around 0 dB and shows less big transitions between levels.

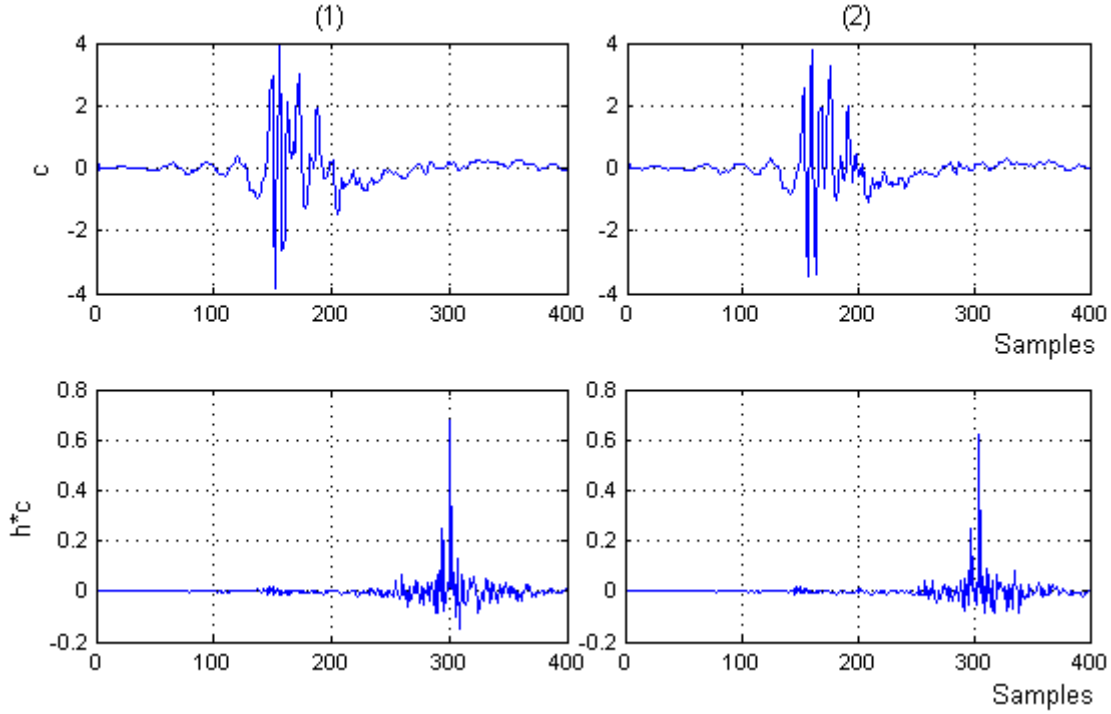


Figure 59 – Impulse responses for acoustic paths before and after equalization for point 1 and 2 (1000-tap measured impulse response, 1000-tap filter length, optimal delays, and no noise).

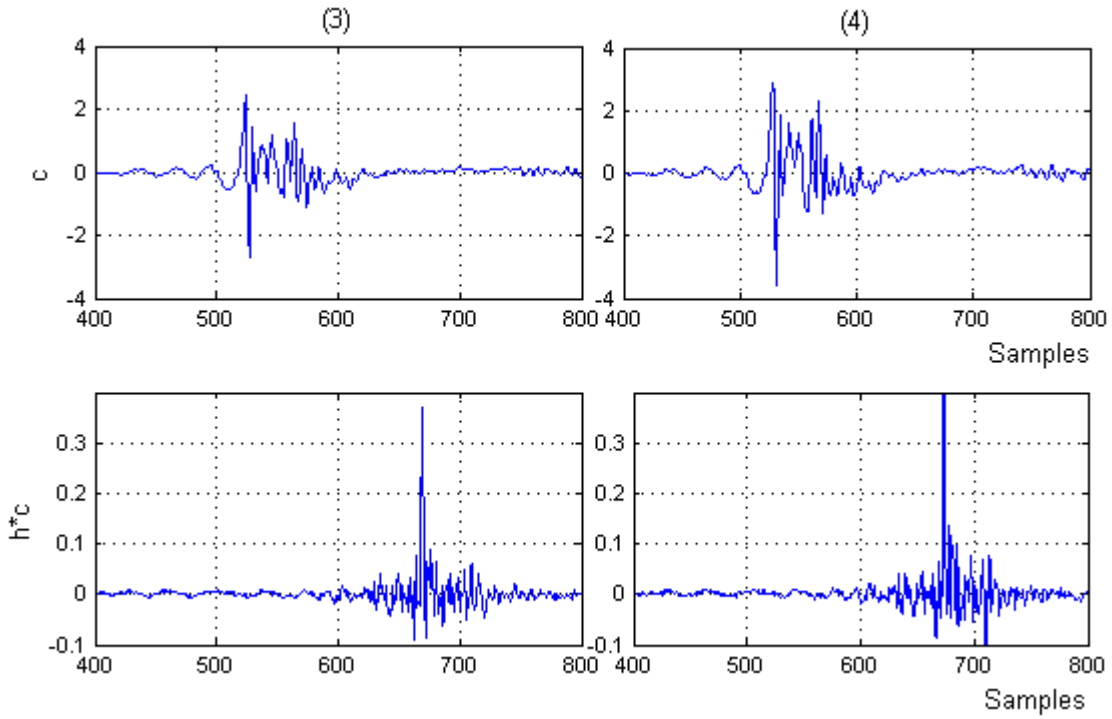


Figure 60 - Impulse responses for acoustic paths before and after equalization for point 3 and 4. (1000-tap measured impulse response, 1000-tap filter length, optimal delays, and no noise).

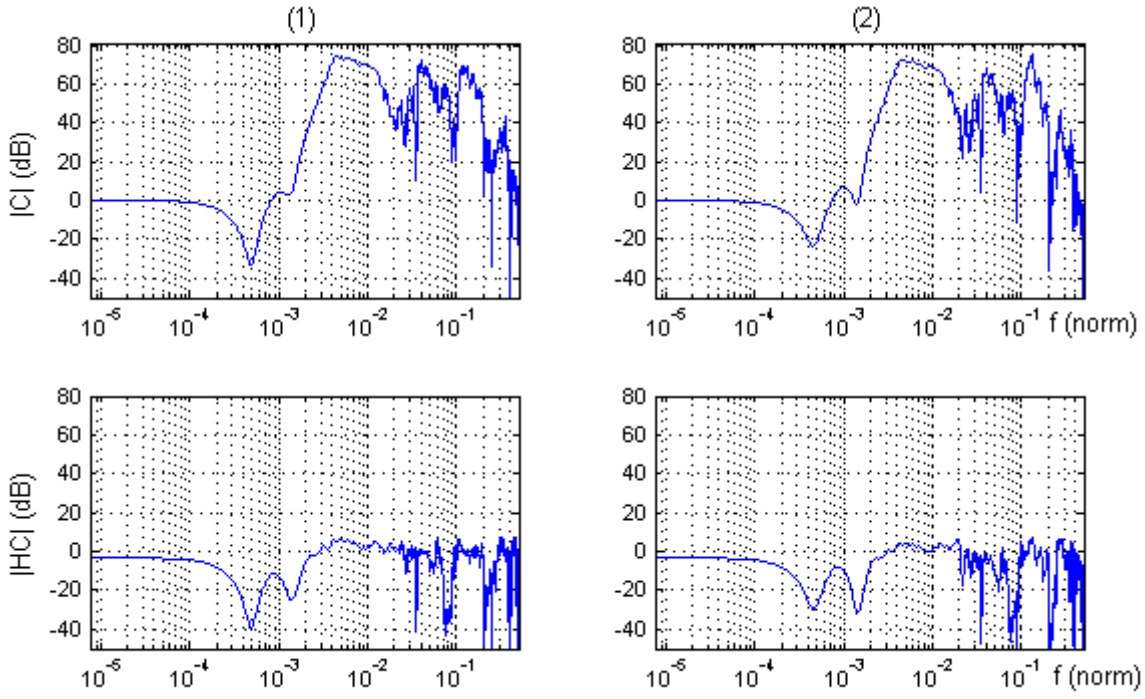


Figure 61 - Frequency responses for acoustic path before and after equalization for point 1 and 2. (1000-tap measured impulse response, 1000-tap filter length, optimal delays, and no noise).

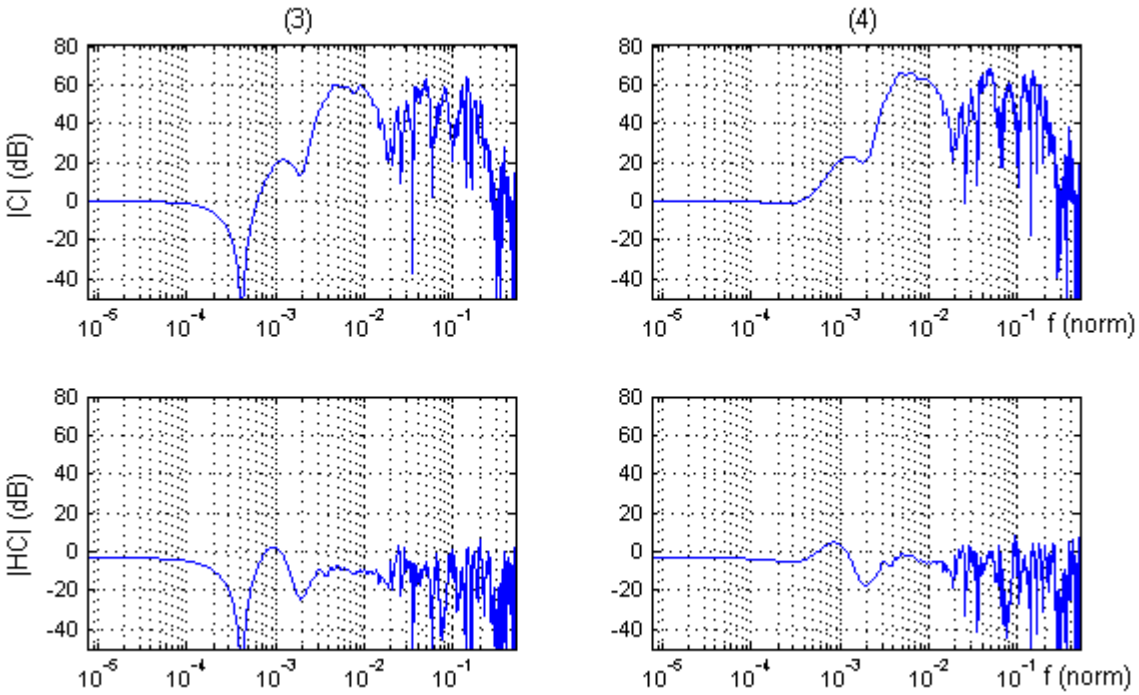


Figure 62 - Frequency responses for acoustic path before and after equalization for point 3 and 4. (1000-tap measured impulse response, 1000-tap filter length, optimal delays, and no noise).

4.3.3 MSE performance

Figure 63 plots the MSE convergence. The algorithm is stable and converges pretty quickly.

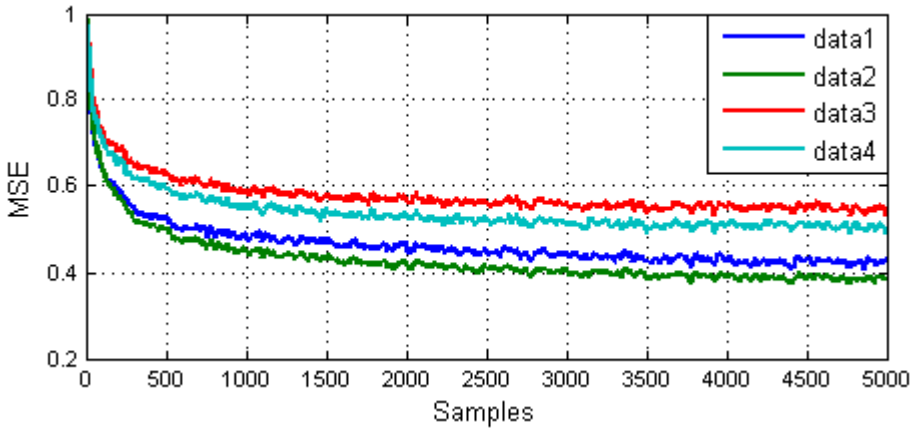


Figure 63 – Learning curves for the four channels (1000-tap measured impulse response, 1000-tap filter length, optimal delays, and no noise).

Finally, Figure 64 shows how the number of optimized points affects the MSE value. Once again, we see that the more point we add, the less we can achieve a good equalization.

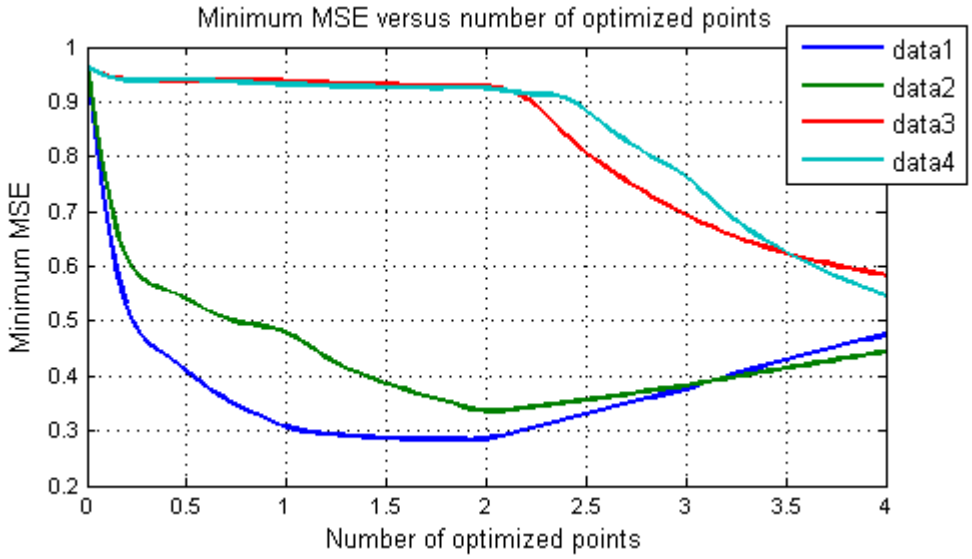


Figure 64 - MMSE performance for increasing number of optimized points. Channel 1 from 0 to 1, channel 2 from 1 to 2, and so on (1000-tap measured impulse response, 1000-tap filter length, optimal delays, and no noise).

5 Discussion

5.1 Choosing the algorithm

The choice of an algorithm isn't an easy task and relies on well-defined constraints. We would want the best and fastest MSE performance with the lowest complexity. This is however an idealistic point of view which can't be realized in the real world, hence the constraints we have to define. In our case, we want to equalize a loudspeaker for multiple channels. The generated filter will be used as a static built-in filter afterwards. Thus we can say that the convergence rate is not an important factor since we can afford to wait a little bit more before stopping the filter update. On the other hands, the most important factor would be the MSE performance since we expect to have a good equalization on, neither one nor two, but at least three channels. Regarding the complexity, the constraint is in between since we don't mind waiting more time for the convergence, but it still should be easy and low-cost to implement.

Now, looking at the simulation results in Chapter 4.1.1 (from page 23), we clearly see the different behaviour between the RLS and Steepest Descent (SD) algorithm class. The RLS algorithm completely outperforms every algorithm when looking at convergence rate. Concerning the MSE performance, the difference after convergence is not huge, except for the LMS algorithm which always requires a fine tuning of the step size. Regarding the sensitivity to noise and filter length, all algorithms show a similar comportment.

About the complexity, we can see with the elapsed time that the RLS algorithm has a high computational load and increases rapidly and nonlinearly with the filter order. This is normal since the RLS algorithm has a complexity of $\sim 3N^2$ operations[11] per iterations whereas a stochastic gradient-based algorithm has one of $2N$. However, the fast Kalman implementation of the RLS algorithm can reduce the complexity to $\sim 10N$ [18]. Still, the RLS will always have higher computational complexity than SD algorithms.

Applying our constraints to what we just said, we can rule out the RLS algorithm since the useless extremely fast convergence comes at the cost of high complexity. Now the question remains on which SD-based algorithm we should choose. They all offer the same complexity since they're based on the same method.

First we can easily rule out the LMS algorithm since we don't want to manually tune the step size. From here, we can rule out the NLMP and NLMAD algorithms since they show slower convergence and slightly worse MSE performance. The choice is then narrowed down to the NLMS and the TV-LMS algorithms which show similar performance in every simulation.

The final choice will be the NLMS algorithm. It's a well-known and well-study algorithm and slightly simpler to implement than the TV-LMS. There is also less parameter to tune.

[11] An operation represents one addition (or subtraction) and one multiplication. It's the so-called Multiply–accumulate (MAC) operation well-known on DSP system.

5.2 The multipoint approach

Multichannel inverse filtering using the sum of the square of the errors presents some nice advantages. It is easy to understand, simple to realise and freely implementable with every existing adaptive algorithm. It could also perfectly work with nonlinear filtering like SOV filters.

However, one has to be careful when using this approach. It can offer mixed to bad results if not handled and implemented correctly.

First of all, special care must be taken for the delayed paths. It all revolves around the largest peak index of the acoustic path. For a single channel, the delay must be at least higher than this index (see Figure 34 p. 26), and low enough to give enough place for the whole impulse response (remember that low frequencies need a long IR tail). For a multichannel system, every channel must satisfy this but are also in relation with each other. This means that the delayed paths must keep the same time (or sample) difference between them as the acoustic paths. This is mandatory to assure a convergence of the algorithm (see Figure 42 p. 30 and Figure 58 p. 39).

Secondly, the nature (e.g. mainly the point's position) and the number of channels play an important role. The more channels we add, the lower the MSE performance is (see Figure 48 p. 33 and Figure 64 p. 42). The same applies to point's position. The further they are from each other, i.e. the larger the area to equalize is, the lower the MSE performance is. It can be seen when comparing the simulations with MARDY data and our measured data (compare Figure 48 with Figure 64). The first one achieves a better performance than the second one. It can be partly explained by the size of the area to equalize: 1.6 m^2 against 6.4 m^2. Hence, the MSE performance will result in a trade-off between the number and the position of the points.

5.3 The Volterra implementation problem

The Second Order Volterra (SOV) filter has been implemented as thoroughly explained in multiple papers [22, 23, 39-43]. In any cases, the learning curve diverges after a certain time even with a small amount of nonlinearities (can be seen Figure 49 p. 34). There is no mention of this behaviour in any paper cited. Furthermore, we can say that this behaviour is completely related to the SOV filter since divergence happened only when it is enabled (see Figure 52 p. 35).

And yet, SOV implementation of the LMS algorithm isn't a big change. The existing part is entirely conserved for the linear filtering, and the quadratic combination $x.x'$ of the input x is used to form the bilinear coefficients h_{quad}, as detailed in equation (9). But without logical reasons the algorithm keeps diverging. We decided then to put the nonlinearity approach aside in order to complete the study.

But we can think of some leads on the cause of the problem. The first and most obvious reason would be a misunderstanding of the Volterra theory. It could also be a bad implemented nonlinear path which leads to self-amplification between the linear and the nonlinear coefficients. As can be seen Figure 50 and Figure 51 (p. 35), both filters diverge.

6 Conclusion

The goal of this thesis was to develop a system that can perform a multichannel equalization of a loudspeaker using Matlab.

After a literature study on inverse filtering, we settled down to linear adaptive filtering since it offers a well-known framework than can be easily expanded to nonlinear and multichannel filtering. We selected a handful adaptive algorithm to test: the RLS, LMS, NLMS, TV-LMS, NLMAD and NLMP. The choice went to the LMS algorithm for its low computational complexity and good MSE performance.

Then we prepared the ground for multichannel by implementing the single channel filtered-x method with the NLMS algorithm. This allowed us to easily expand it to the multichannel theory. Simulation showed good and expected results with both simple 50-tap and MARDY 1000-tap IR.

Next we tried to implement the Volterra nonlinear model with a SOV filter. We sadly ran into divergence problems beyond understanding which prevented us to continue. This was a real disappointment at this time of the study.

We moved on anyway to linearly characterize our home-made loudspeaker in an anechoic room. We measured its IR and FR from different positions. We then used those IR to simulate our algorithm on the loudspeaker in Matlab. This yields to "roughly" good results, partly due to the size of the equalized area. Still, the equalized IR show less ringing and more compact, impulse-like responses, which is basically what we wanted.

7 Future works

First of all it could be interesting to find why we have a divergence with our SOV filter. Linear filtering works, but using nonlinear filtering can definitely improve the MSE performance of the algorithm. Leads discussed in Chapter 5.3 can be used as a starting point, but maybe it is better to start from scratch. The bilinear approach [39, 42, 43] to nonlinear filtering could also be considerate, properly tested and compared with the SOV filter. After obtaining a working nonlinear filter, the multichannel approach can be applied and simulated.

Another important work would be to actually try the algorithm on the loudspeaker, i.e. by not using simulation but real-time filtering. This way, a proper validation of the algorithm can be achieved with real measurements of the filtered impulse responses. Note that this can be done with or without the nonlinear implementation. For example the measurement system Figure 22 can become the one Figure 65.

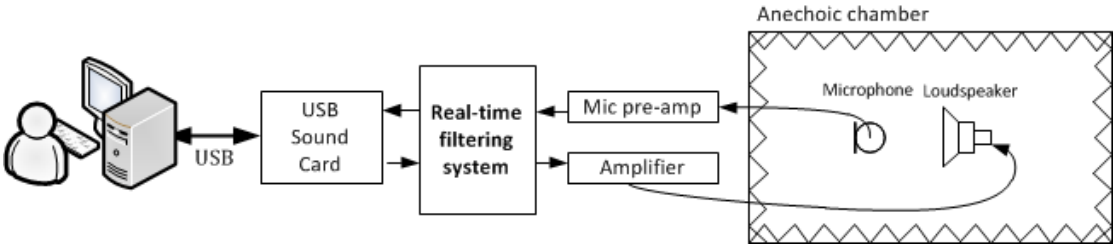


Figure 65 - Measurement system with real-time filtering.

The real-time filtering system can be realized with another computer so the existing scripts and models in Matlab can be used (with proper modification of the inputs and outputs). A DSP can also be used, but will require a development step to port the algorithm to the target architecture.

Furthermore, it could be interesting to perform these measurements in various real-life environments like bedroom or classroom, to see if problems arise from reverberations.

Last but not least, the ultimate stage would be to realize this filter as built-in either in the loudspeaker or in a pre-amplifier. The later would be easier since power source would be directly available. This extensive work would require hardware study (e.g. what system to use? what architecture? etc…), programming study (e.g. which language to use? etc…), mechanical study (e.g. what are the maximum dimensions? etc…) and electrical study (e.g. how much power is available? what are the input and output impedances? should a matching circuit be used? etc…). And of course a cost study will drive the choices all along the project.

References

[1] O. Kirkeby and P. A. Nelson, "Digital filter design for inversion problems in sound reproduction," *Journal of the Audio Engineering Society*, vol. 47, pp. 583-595, Jul-Aug 1999.

[2] A. Carini, S. Cecchi, F. Piazza, I. Omiciuolo, and G. L. Sicuranza, "Multiple Position Room Response Equalization in Frequency Domain," *Ieee Transactions on Audio Speech and Language Processing*, vol. 20, pp. 122-135, Jan 2012.

[3] Y. Haneda, S. Makino, and Y. Kaneda, "Multiple-point equalization of room transfer functions by using common acoustical poles," *Ieee Transactions on Speech and Audio Processing*, vol. 5, pp. 325-333, Jul 1997.

[4] S. G. Norcross, G. A. Soulodre, and M. C. Lavoie, "Subjective investigations of inverse filtering," *Journal of the Audio Engineering Society*, vol. 52, pp. 1003-1028, Oct 2004.

[5] S. Brown, "Linear and Nonlinear Loudspeaker Characterization," Worcester Polytechnic Institute 2006.

[6] W. C. Zhang, A. W. H. Khong, and P. A. Naylor, "ADAPTIVE INVERSE FILTERING OF ROOM ACOUSTICS," in *42nd Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, 2008, pp. 788-792.

[7] S. Cecchi, L. Palestini, P. Peretti, F. Piazza, and A. Carini, "Multipoint Equalization of Digital Car Audio Systems," *2009 Proceedings of 6th International Symposium on Image and Signal Processing and Analysis (Ispa 2009)*, pp. 656-661, 2009.

[8] M. Miyoshi and Y. Kaneda, "INVERSE FILTERING OF ROOM ACOUSTICS," *Ieee Transactions on Acoustics Speech and Signal Processing*, vol. 36, pp. 145-152, Feb 1988.

[9] S. J. Elliott and P. A. Nelson, "Multiple-Point Equalization in a Room using Adaptive Digital-Filters," *Journal of the Audio Engineering Society*, vol. 37, pp. 899-907, Nov 1989.

[10] F. Lingvall and L. J. Brannmark, "Multiple-point statistical room correction for audio reproduction: Minimum mean squared error correction filtering," *Journal of the Acoustical Society of America*, vol. 125, pp. 2121-2128, Apr 2009.

[11] L. J. Brannmark and A. Ahlen, "Spatially Robust Audio Compensation Based on SIMO Feedforward Control," *Ieee Transactions on Signal Processing*, vol. 57, pp. 1689-1702, May 2009.

[12] E. U. Angelo Farina, Alberto Bellini, Gianfranco Cibelli, Carlo Morandi, "Inverse numerical filters for linearisation of loudspeaker's response," University of Parma, 2000.

[13] F. X. Y. Gao, W. M. Snelgrove, and Ieee, "ADAPTIVE LINEARIZATION OF A LOUDSPEAKER," *Icassp 91, Vols 1-5*, pp. 3589-3592, 1991.

[14] T. Ishikawa, K. Nakashima, Y. Kajikawa, and Y. Nomura, "A consideration on elimination of nonlinear distortion of the loudspeaker system by using digital Volterra filter," *Electronics and Communications in Japan Part Iii-Fundamental Electronic Science*, vol. 83, pp. 110-118, 2000.

[15] H. F. Niklas Agevik, Henrik Grunell, Daniel Hasselqvist, Patrick Jakiel and Henrik Lundin, "On Loudspeaker Linearization Using Pre-Distortion," KTH Royal Institute of Technology, Signals, Sensors and Systems, 2004.

[16] Y. Nomura, Y. Kajikawa, and Ieee, "Linearization of loudspeaker systems using mint and volterra filters," in *30th IEEE International Conference on Acoustics, Speech, and Signal Processing*, Philadelphia, PA, 2005, pp. 457-460.

[17] P. R. Chang, C. G. Lin, and B. F. Yeh, "INVERSE FILTERING OF A LOUDSPEAKER AND ROOM ACOUSTICS USING TIME-DELAY NEURAL NETWORKS," *Journal of the Acoustical Society of America*, vol. 95, pp. 3400-3408, Jun 1994.

[18] P. G. B. Mulgrew, J. Thompson, *Digital Signal Processing : Concepts and Applications*, Second ed.: Palgrave Macmillan, 2003.

[19] W. J. Rugh, *Nonlinear System Theory - The Volterra/Wiener Approach*, 2002 Web ed.: The Johns Hopkins University Press, 1981.

[20] A. J. M. Kaizer, "Modeling of the nonlinear response of an electrodynamic loudspeaker by a Volterra series expansion," *Journal of the Audio Engineering Society*, vol. 35, pp. 421-433, Jun 1987.

[21] K. N. Tomokazu Ishikawa, Yoshinobu Kajikawa, and Yasuo Nomura, "A Consideration on Elimination of Nonlinear Distortion of the Loudspeaker System by Using Digital Volterra Filter," *Electronics and Communications in Japan*, vol. 83, 2000.

[22] Y. Kajikawa and Y. Nomura, "Design of nonlinear inverse systems by means of adaptive Volterra filters," *Electronics and Communications in Japan Part Iii-Fundamental Electronic Science*, vol. 80, pp. 36-45, Aug 1997.

[23] S. L. Chang, T. Ogunfunmi, and Ieee, "LMS/LMF and RLS Volterra system identification based on nonlinear Wiener model," *Iscas '98 - Proceedings of the 1998 International Symposium on Circuits and Systems, Vols 1-6*, pp. D206-D209, 1998.

[24] L. Carassale and A. Kareem, "Modeling Nonlinear Systems by Volterra Series," *Journal of Engineering Mechanics-Asce*, vol. 136, pp. 801-818, Jun 2010.

[25] M. Kleiner, *Audio Technology and Acoustics*, 2nd ed. Chalmers University of Technology: Division of Applied Acoustics, 2008.

[26] N. Quaegebeur and A. Chaigne, "Mechanical resonances and geometrical nonlinearities in electrodynamic loudspeakers," *Journal of the Audio Engineering Society*, vol. 56, pp. 462-472, Jun 2008.

[27] S. T. Neely and J. B. Allen, "Invertibility of a room impulse-response," *Journal of the Acoustical Society of America*, vol. 66, pp. 165-169, 1979.

[28] Q. Meng, D. Sen, S. Wang, and L. Hayes, "IMPULSE RESPONSE MEASUREMENT WITH SINE SWEEPS AND AMPLITUDE MODULATION SCHEMES," *Icspcs: 2nd International Conference on Signal Processing and Communication Systems, Proceedings*, pp. 617-621, 2008.

[29] L. Hakansson, "The Filtered-x LMS Algorithm," University of Karlskrona/Ronneby, 2004.

[30] S. J. Elliott, I. M. Stothers, and P. A. Nelson, "A MULTIPLE ERROR LMS ALGORITHM AND ITS APPLICATION TO THE ACTIVE CONTROL OF SOUND AND VIBRATION," *Ieee Transactions on Acoustics Speech and Signal Processing*, vol. 35, pp. 1423-1434, Oct 1987.

[31] E. Bjarnason, "ANALYSIS OF THE FILTERED-X LMS ALGORITHM," *Ieee Transactions on Speech and Audio Processing*, vol. 3, pp. 504-514, Nov 1995.

[32] I. T. Ardekani and W. H. Abdulla, "Filtered weight FxLMS adaptation algorithm: Analysis, design and implementation," *International Journal of Adaptive Control and Signal Processing*, vol. 25, pp. 1023-1037, Nov 2011.

[33] Y.-S. Lau, Z. M. Hussian, and R. Harris, "Performance of Adaptive Filtering Algorithms: A Comparative Study," School of Electrical and Computer Engineering, RMIT University, Melbourne, Australia., 2003.

[34] L. Ferdouse, N. Akhter, T. H. Nipa, and F. T. Jaigirdar, "Simulation and Performance Analysis of Adaptive Filtering Algorithms in Noise Cancellation," *IJCSI International Journal of Computer Science*, vol. 8, pp. 185-192, 2011.

[35] O. Arikan, M. Belge, A. E. Cetin, E. Erzin, and Ieee, "ADAPTIVE FILTERING APPROACHES FOR NON-GAUSSIAN STABLE PROCESSES," *1995 International Conference on Acoustics, Speech, and Signal Processing - Conference Proceedings, Vols 1-5*, pp. 1400-1403, 1995.

[36] A. B. Sankar, D.Kumar, and K.Seethalakshmi, "Performance Study of Various Adaptive Filter Algorithms for Noise Cancellation in Respiratory Signals " *Signal Processing : An International Journal (SPIJ)*, vol. 4, pp. 267-278, 2010.

[37] J. Kivinen, M. K. Warmuth, and B. Hassibi, "The p-norm generalization of the LMS algorithm for adaptive filtering," *Ieee Transactions on Signal Processing*, vol. 54, pp. 1782-1793, May 2006.

[38] J. Y. C. Wen, N. D. Gaubitch, E. A. P. Habets, T. Myatt, and P. A. Naylor, "Evaluation of speech dereverberation algorithms using the MARDY database," in *Proc. Intl. Workshop Acoust. Echo Noise Control (IWAENC)*, Paris, France, 2006.

[39] T. S. D. Singh and A. Chatterjee, "A comparative study of adaptation algorithms for nonlinear system identification based on second order Volterra and bilinear polynomial filters," *Measurement*, vol. 44, pp. 1915-1923, Dec 2011.

[40] L. Z. Tan and J. Jiang, "Filtered-X second-order Volterra adaptive algorithms," *Electronics Letters*, vol. 33, pp. 671-672, Apr 1997.

[41] Y. Takahama, Y. Kajikawa, and Y. Nomura, "A formulation of the convergence property for the second-order adaptive Volterra filter using NLMS algorithm," *Electronics and Communications in Japan Part Iii-Fundamental Electronic Science*, vol. 85, pp. 40-50, 2002.

[42] S. M. Kuo and H. T. Wu, "Nonlinear adaptive bilinear filters for active noise control systems," *Ieee Transactions on Circuits and Systems I-Regular Papers*, vol. 52, pp. 617-624, Mar 2005.

[43] V. J. Mathews, "Adaptive polynomial filter," *IEEE Signal Processing Magazine*, vol. 8, pp. 10 - 26 June 1991.

Annexes

Please note that the whole project (files, Matlab scripts and models, pictures …) will be available to download at http://guillaume.perrin74.free.fr/ChalmersMT2012 for at least one year.

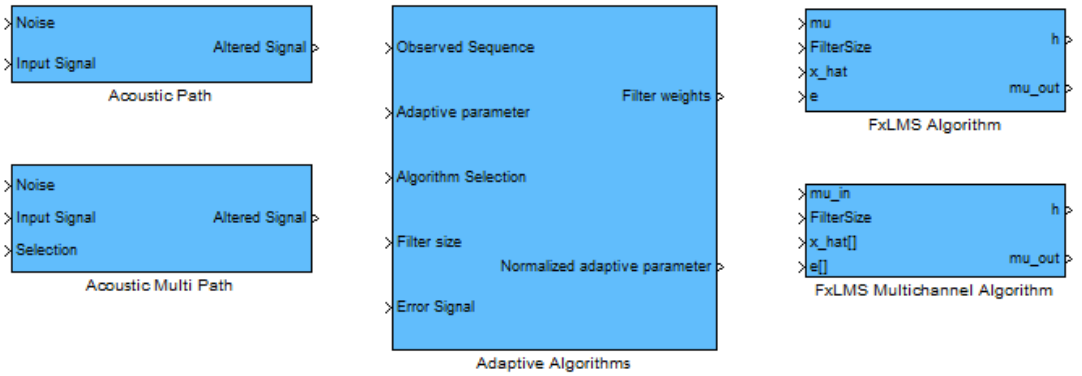
Matlab File structure

The Matlab related files are stored using this directory structure:

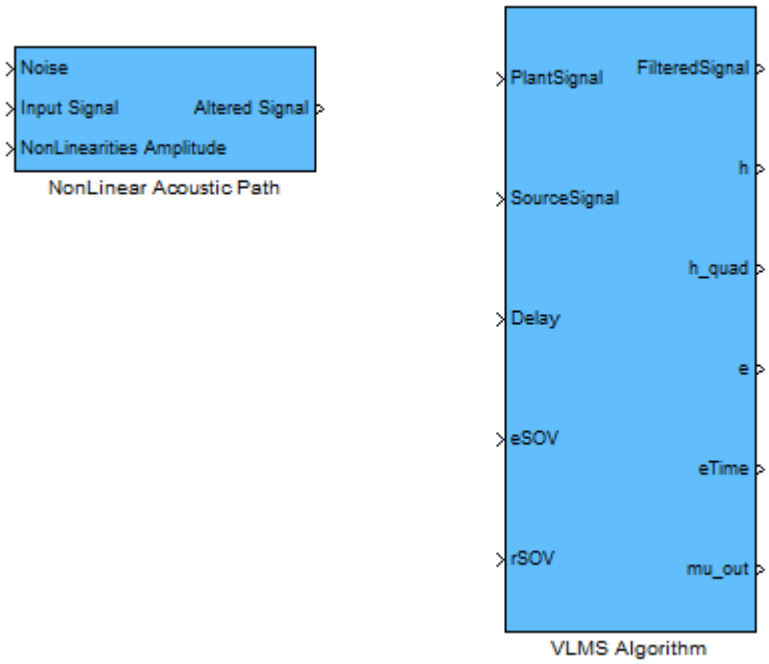
- **global:** contains general functions and files (IR …) used by all Matlab script.
- **libraries:** contains libraries Simulink models (acoustic path, algorithms …)
  - **lib_Linear:** linear library models.
  - **lib_NonLinear:** nonlinear library models.
- **LinearImplementation:** contains all scripts and Simulink models for the linear implementation.
  - **AdaptiveAlgTest:** contains scripts and models for the adaptive algorithms tests.
    - **Results:** simulation results.
  - **SingleChannel:** scripts and models for the single channel simulations.
    - **Results:** simulation results.
  - **MultiChannel:** scripts and models for the multichannel simulations.
    - **Results:** simulation results.
- **VolterraImplementation:** contains all scripts and Simulink models for the nonlinear implementation using Volterra series.
- **Measurements:** contains all scripts for measurements processing (directivity pattern …).

Matlab models

LibLinear

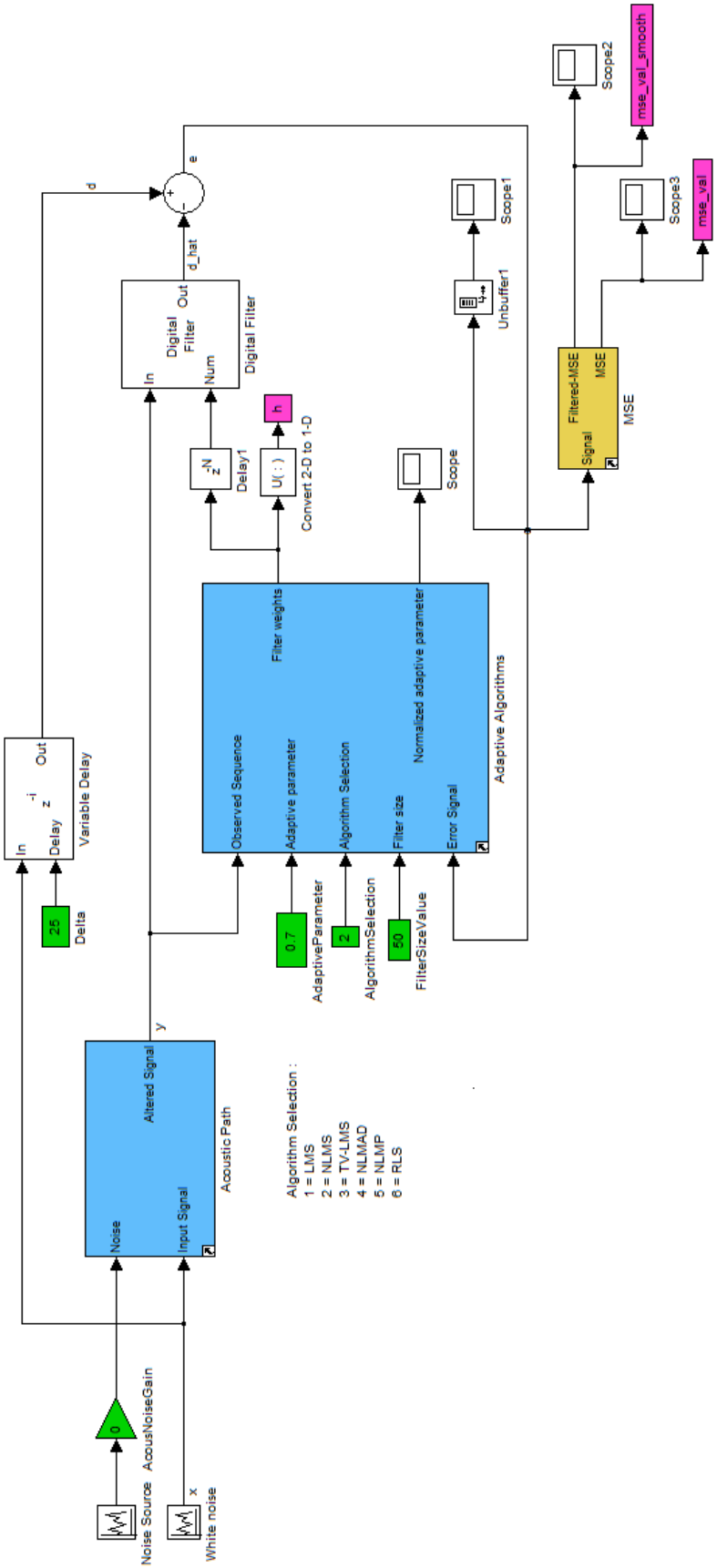


LinNonLinear

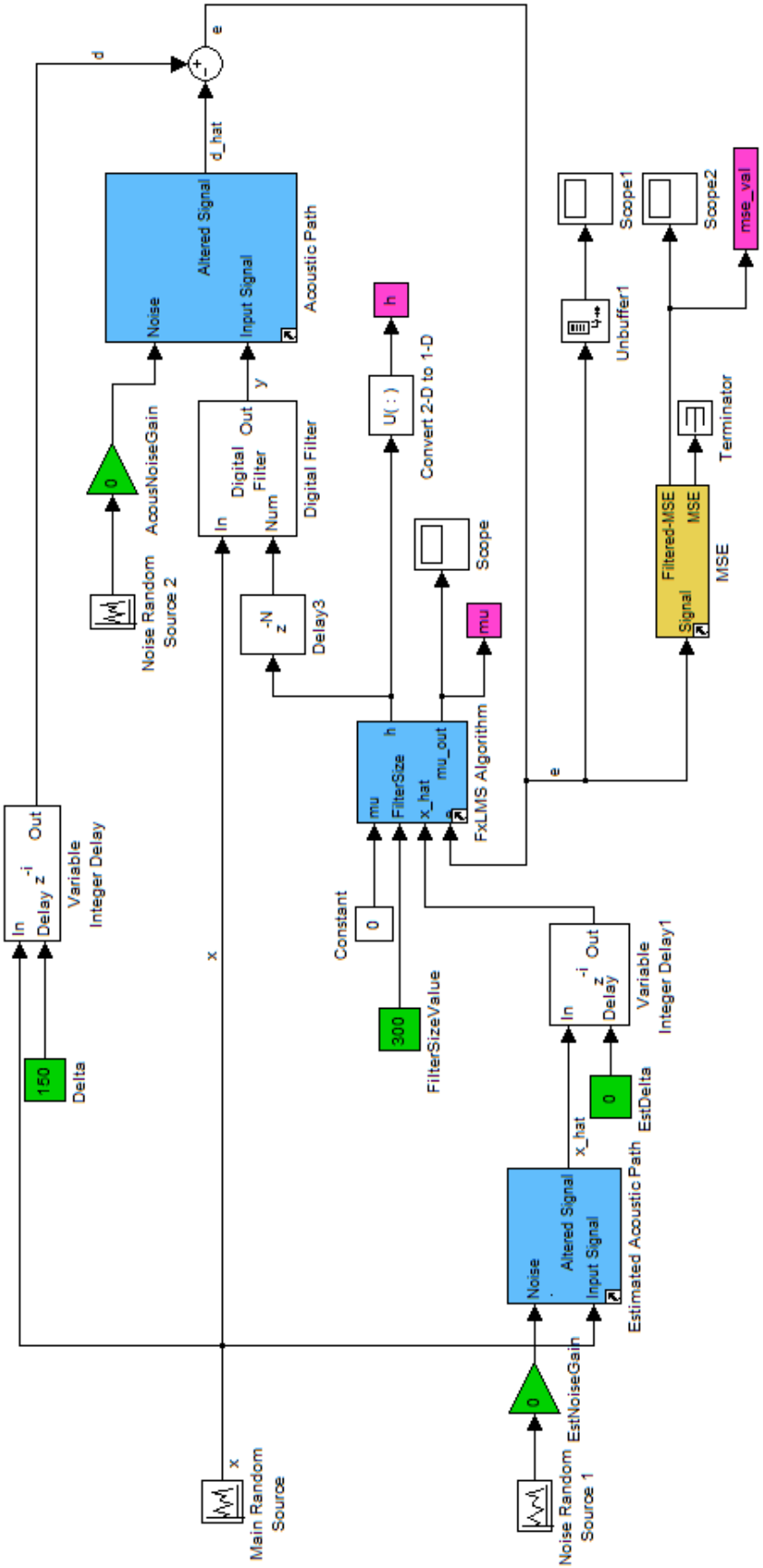


LibGlobal

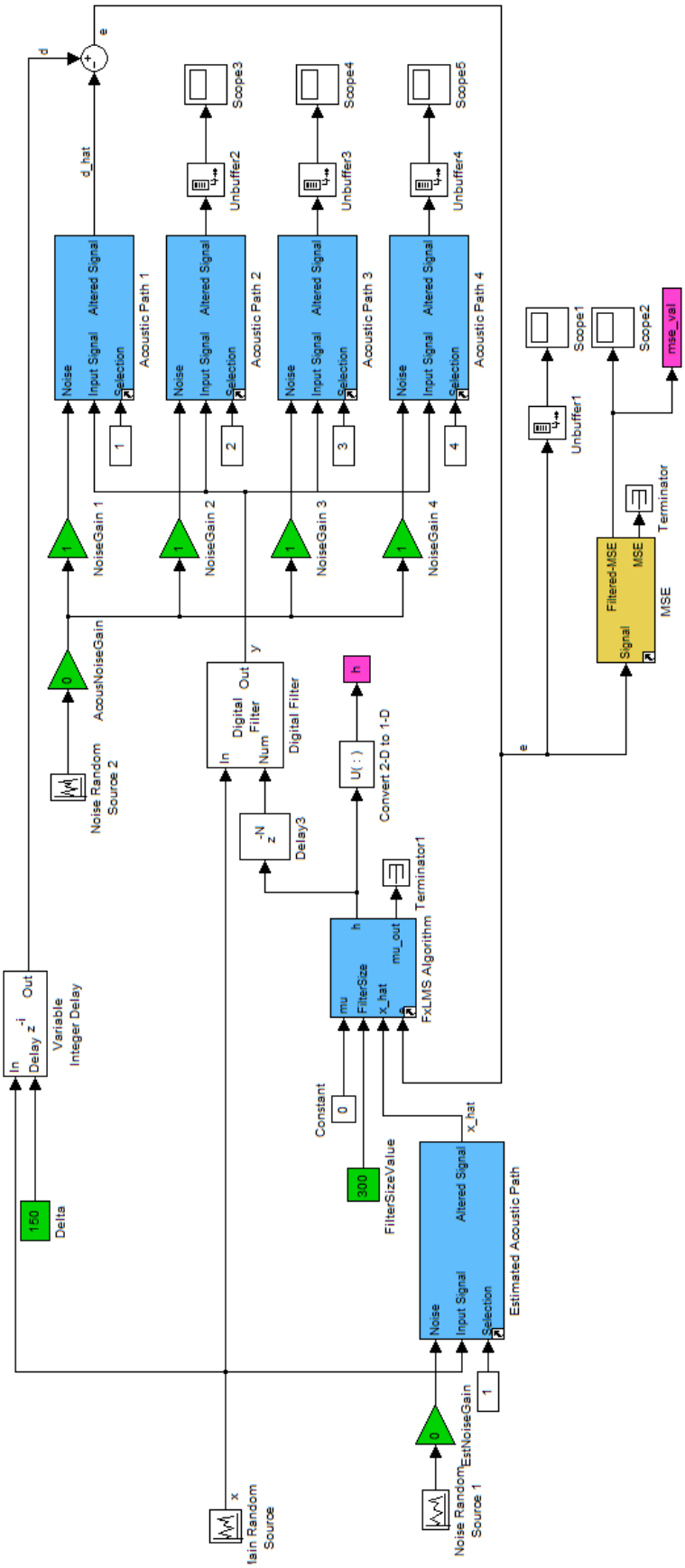
AdaptiveAlg_FIR_SC_IRTest



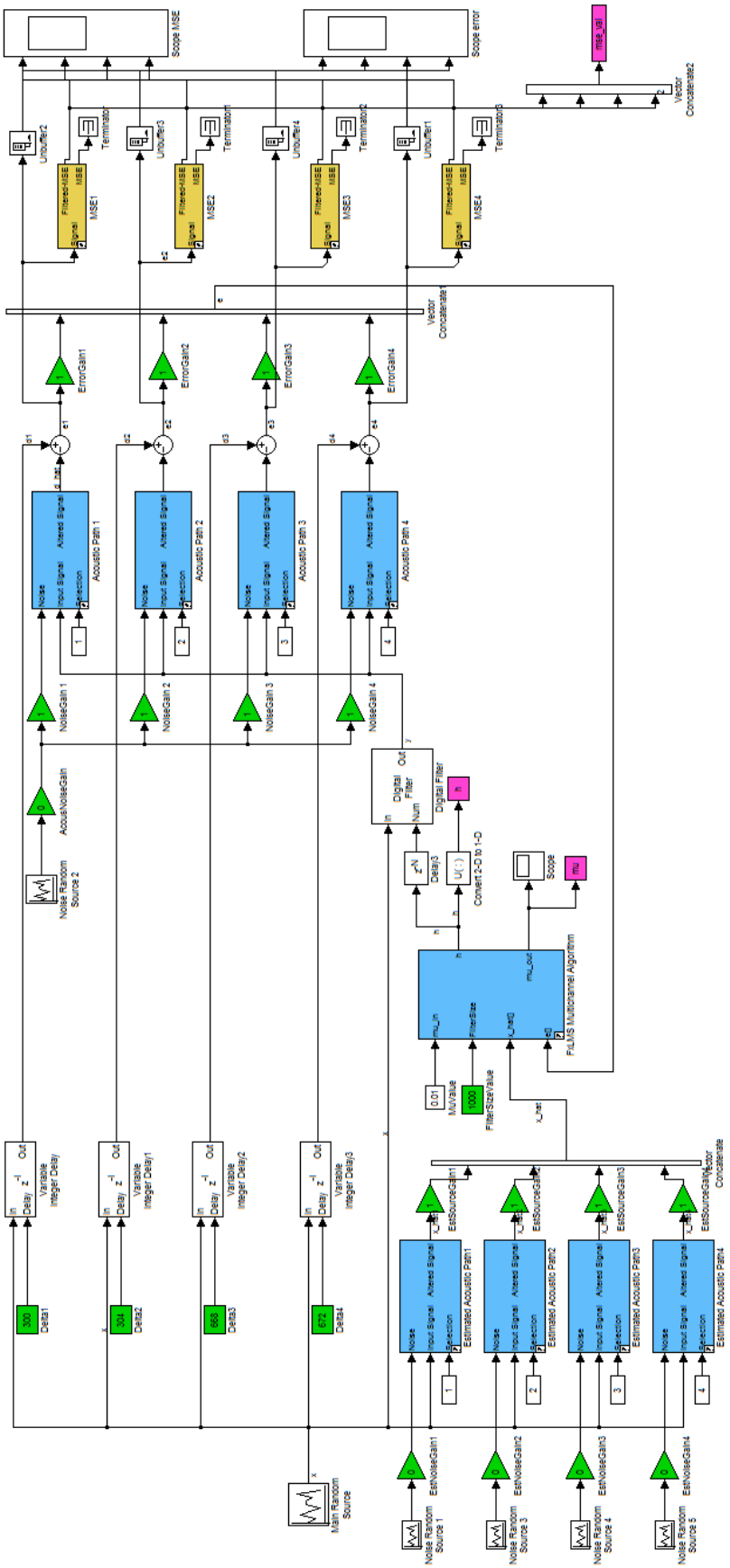
FxLMS_SC_IRTest



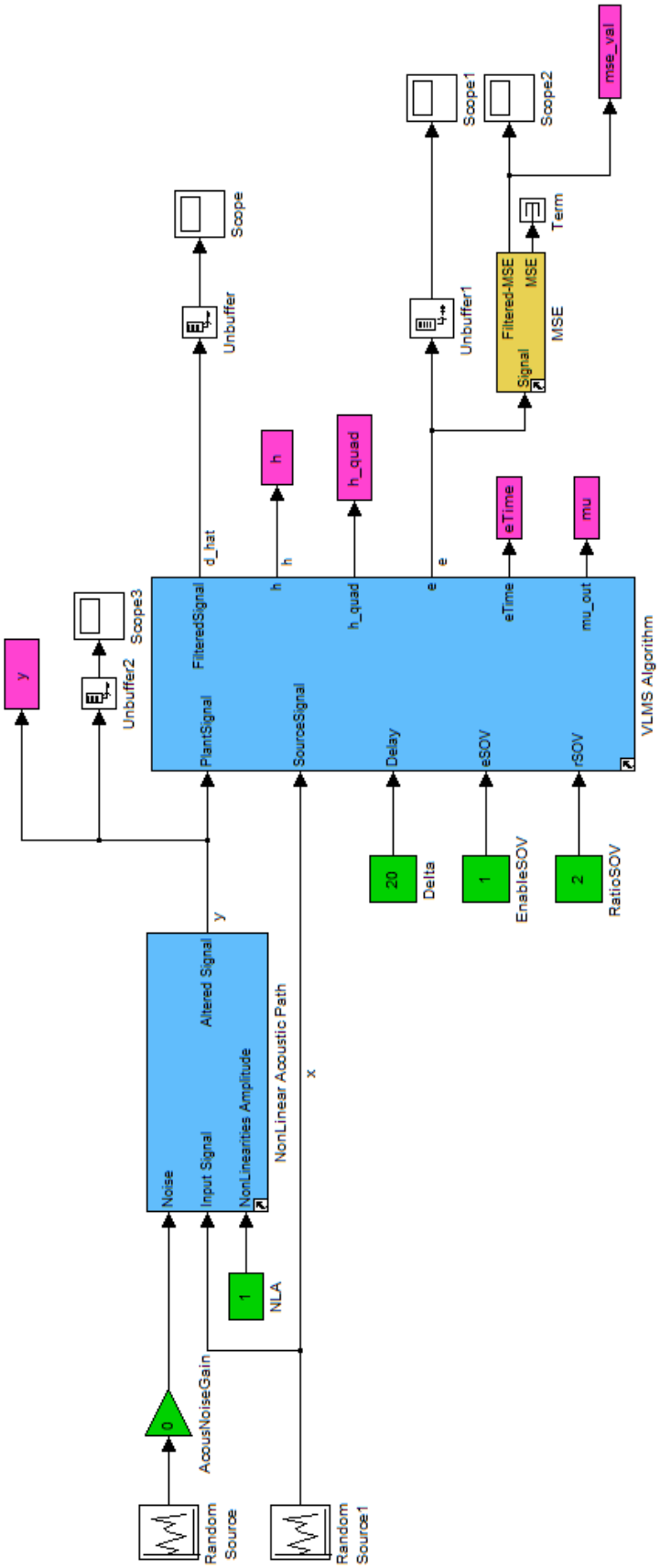
FxLMS_SC_IRTest_multichannel



FxLMS_MC_IRTest



VLMS_SCNL_IRTest



Matlab script

Function: AdaptFilters (used in bloc Adaptive Algorithms)

```
function [w, a_param_out] = AdaptFilters(y, a_param_in,
algorithm_index, L, e)
%#codegen

persistent weights;
persistent fifo;
persistent absolute_n ;

persistent rls_P ;
persistent mu_0 ;

N = length(y);

% Small value
lambda = 1E-30 ;

if (length(a_param_in) >= 1)
    mu = a_param_in(1);
end


if isempty(weights)
    % Filter coefficients:
    weights = zeros(N,1);
    % FIFO Shift Register:
    fifo = zeros(N,1);
    % RLS recursion matrix
    delta = sqrt(mean(y.^2));
    rls_P = 0.1*eye(N) / delta ;
    % Absolute value of n
    absolute_n = 1 ;
    mu_0 = 0 ;
end

if (L ~= 0)

for n = 1:N
    % Update the FIFO shift register:
    fifo(2:L) = fifo(1:L-1);
    fifo(1) = y(n);

    if (algorithm_index == 1)
        %% LMS ALGORITHM

        if (mu == 0)
            mu_max = 1/((lambda + mean(y.^2))*L) ;
            mu = 0.1*mu_max;
        end

        weights(1:L) = weights(1:L) + mu * e(n) * fifo(1:L) ;

    elseif (algorithm_index == 2)
        %% NLMS ALGORITHM

        if (mu == 0)
            mu = 0.1;
        end
```

```
        if (n == 1)
            mu = mu / (lambda + norm(y).^2);
        end

        weights(1:L) = weights(1:L) + mu * e(n) * fifo(1:L) ;

    elseif (algorithm_index == 3)
        %% TV-LMS ALGORITHM
        C = 20 ;
        a = 0.01 ;
        b = 0.7 ;

        if(mu_0 == 0)
            if(mu == 0)
%                   mu_0 = 0.5E-6;
                mu_0 = 0.1 / (lambda + norm(y).^2);
            else
                mu_0 = mu ;
            end
        end

        mu = mu_0 * C^(1/(1+a*absolute_n^b));
        weights(1:L) = weights(1:L) + mu * e(n) * fifo(1:L) ;

    elseif (algorithm_index == 4)
        %% NLMAD ALGORITHM

        if (mu == 0)
            mu = 0.001;
        end

        if (n == 1)
            mu = mu / (lambda + norm(y,1));
        end

        weights(1:L) = weights(1:L) + mu * sign(e(n)) * fifo(1:L) ;


    elseif (algorithm_index == 5)
        %% NLMP ALGORITHM

        if (length(a_param_in) >= 2)
            p = a_param_in(2);
            assert(p<=2 && p>=1);
        else
            p = 1.5 ;
        end

        if (mu == 0)
            mu = 0.01;
        end

        if (n == 1)
            mu = mu / (lambda + norm(y,p).^p);
        end

        weights(1:L) = weights(1:L) + mu * sign(e(n)) * abs(e(n)).^(p-1)
* fifo(1:L) ;

    elseif (algorithm_index == 6)
        %% RLS ALGORITHM

        if (mu == 0)
            mu = 1;
        end
```

```
        phi = fifo(1:L)' * rls_P(1:L,1:L) ;
        k = phi'/(mu + phi * fifo(1:L) );

        weights(1:L) = weights(1:L) + k * e(n) ;

        rls_P(1:L,1:L) = (rls_P(1:L,1:L) - k * phi) / mu ;

    end

    absolute_n = absolute_n + 1 ;
end
end

a_param_out = mu ;

% Output the filter weights:
w = weights;

end
```

Function: NLMS_filter (used in bloc FxLMS algorithm)

```
function [h, mu] = NLMS_filter(y, e, L, mu_in) %#codegen
% Generate the NLMS filter
%   Input
%       y (float array) : source signal
%       e (float array) : error signal
%       L (int) : filter length (in samples)
%       mu (float) : convergence coefficient (set to 0 for default
value)
%   Output
%       h (float array) : filter coefficients
%       mu (float) : calculated convergence coefficient
%

persistent weights;
persistent fifo;

N = length(y);

% Small value
lambda = 1E-30 ;

if isempty(weights)
    % Filter coefficients:
    weights = zeros(N,1);
    % FIFO Shift Register:
    fifo = zeros(N,1);
end

% Convergence coefficient
if (mu_in == 0)
    mu = 0.1 / (lambda + norm(y).^2);
else
    mu = mu_in / (lambda + norm(y).^2);
end

for n = 1:N
    % Update the FIFO shift register
    fifo(2:L) = fifo(1:L-1);
    fifo(1) = y(n);
    % Update filter coefficients
```

```
    weights(1:L) = weights(1:L) + mu * e(n) * fifo(1:L) ;
end

% Output the filter weights
h = weights;

end
```

Function: FxLMS_MC_filter (used in bloc FxLMS multichannel algorithm)

```
function [h, mu] = FxLMS_MC_filter(y, e, L, mu_in) %#codegen
% Generate the LMS filter
%   Input
%       x_hat (4 float array) : approximate source signal (from model
forward
%       path)
%       e (4 float array) : error signal
%       L (int) : filter length (in samples)
%       mu (float) : convergence coefficient (set to 0 for automatic
%                       detection)
%   Output
%       float array
%

persistent weights;
persistent fifo;

N = length(y);

% Small value
lambda = 1E-30 ;

% Number of signal
K = size(y,2);

if isempty(weights)
    % Filter coefficients:
    weights = zeros(N,1);
    % FIFO Shift Register:
    fifo = zeros(N,K);
end

% Convergence coefficient
if (mu_in == 0)
    mu = 0.1 / (lambda + mean(norm(y)).^2);
else
    mu = mu_in / (lambda + mean(norm(y)).^2);
end

% if (mu_in == 0)
%     mean_x_hat = 0 ;
%     for k = 1:K
%         mean_x_hat = mean_x_hat+mean(x_hat(:,k).^2);
%     end
%     mu_max = 2/(mean_x_hat*L) ;
%     mu = 0.05*mu_max;
% else
%     mu = mu_in;
% end
```

```
for n = 1:L
    % Sum over the channel
    for k=1:K
        % Update the FIFO shift registers
        fifo(2:L,k) = fifo(1:L-1,k);
        fifo(1,k) = y(n,k);
        % Update weights values
        weights(1:L) = weights(1:L) + mu * e(n,k) * fifo(1:L,k);
    end
end

% Output the filter weights:
h = weights;

end
```

Function: VLMS_filter (used in bloc VLMS algorithm)

```
function [ d_hat, h, h_quad, e, elaspedTime, mu_out ] = VLMS_filter( d,
x, L, EnableSOV, ratioSOV)
% Generate the LMS filter
%   Input
%       d (float array) : source signal (delayed)
%       x (float array) : modified signal from plant
%       L (int) : filter length (in samples)
%       mu (float) : convergence coefficient (set to 0 for automatic
%                    detection)
%   Output
%       float array
%
coder.varsize('weights_all');
coder.varsize('fifo_all');

persistent weights;
persistent weights_quad;
persistent fifo;
% persistent fifo_quad;

elaspedTime = 0 ;

% Start time measurement
coder.extrinsic('tic','toc');
tic;

coder.inline('never');

% Non linear size
if (ratioSOV < 1)
    ratioSOV = 1 ;
end
Lq = round(L / ratioSOV) ;

if isempty(weights)
    % Filter coefficients:
    weights = zeros(L,1);
    % Filter quadratic coefficients:
    weights_quad = zeros(Lq*(Lq+1)/2,1);
    % FIFO Shift Register:
    fifo = zeros(L,1);
end

% Pre-allocate output and error signals:
```

```
d_hat = coder.nullcopy(zeros(L,1));
e = coder.nullcopy(zeros(L,1));
fifo_quad = coder.nullcopy(zeros(Lq*(Lq+1)/2,1));

mu_out = coder.nullcopy(zeros(2,L));


for n = 1 : L
    %## Update the Linear Fifo
    fifo(2:L) = fifo(1:L-1);
    fifo(1) = x(n);

    % Approximation for plant signal
%      d_hat(n) = weights' * fifo;
    d_hat(n) = fifo' * weights;

    if (EnableSOV == 1)
        %##Quadratic combinaison Fifo
        fifo_quad = buildSOV_Vector(fifo(1:Lq));
        % Add quadratic plant approximation
%          d_hat(n) = d_hat(n) + weights_quad' * fifo_quad;
        d_hat(n) = d_hat(n) + fifo_quad' * weights_quad;
    end

    % Calculate the error
    e(n) = d(n) - d_hat(n) ;

    % Calculate mu
    mu_out(1,n) = 0.01 / (norm(fifo).^2+1E-30) ;

    % Update the linear filter
     weights = weights + mu_out(1,n) * e(n) * fifo;

    if (EnableSOV == 1)
        % Calculate mu
        mu_out(2,n) = 0.01 / (norm(fifo_quad).^2+1E-30) ;
        % Update the filter
        weights_quad = weights_quad + mu_out(2,n) * e(n) * fifo_quad ;

    end

end

% Output the filter weights:
h = weights;
h_quad = weights_quad ;

% End time measurement
elaspedTime = toc;

end
```

Function: buildSOV_Vector (used in the function above)

```
function [ x_quad ] = buildSOV_Vector( x )
% Build the Second Order Volterra Vector (quadratic
% combination of the input).
%
%   Input
%       x (vector) : linear source signal
%   Output
%       x_quad (vector) : quadratic combination
```

```
%

% Input size
N = length(x);

% Build quadratic combination matrix
x_matrix = x * x.';

% fyllo = [1:N+1:N^2]';
% x_matrix(fyllo)=abs(diag(x_matrix));

% Preallocating output
x_quad = zeros(N*(N+1)/2,1) ;
% Iterative loop to create the volterra vector
index=1;
for i=1:N
    x_quad(index:index+N-i) = x_matrix(i,i:N).';
    index = index + N - i + 1;
end

end
```

Script: AdaptiveAlg_FIR_SC_main_script

```
%
%   MAIN SIMULATION SCRIPT
%       ADAPTIVE ALGORITHM
%       LINEAR SINGLE CHANNEL
%       FIR
%
%% Initialize
clear all
close all
%clc

MDL_Init_Script
mdl = 'AdaptiveAlg_FIR_SC_IRTest';

disp('****** Parameters settings');

%====================================%
%====== Simulation Parameters ======%
%====================================%

% Simulation Algorithm Selection :
%     0 = All algorithm comparison
%     1 = LMS
%     2 = NLMS
%     3 = TV-LMS
%     4 = NLMAD
%     5 = NLMP
%     6 = RLS
sim_algo = 0;

% Simulation type
%     0 = No variation - (just simulate convergence with default
values)
%     1 = delay
%     2 = noise
%     3 = filter length
%     4 = Input signal source
```

```
%     5 = convergence factor
sim_type = 0;

% Ignore RLS Algorithm (useful for testing, RLS can take time)
%     0 = Don't ignore RLS
%     1 = ignore RLS
sim_algo_no_RLS = 0;

% Number of simulation points
sim_N_points = 9 ;

% Number of frame
sim_N_frame = 2000 ;

% Frame size
%   [2x1] vector (see AccPath_IR)
sim_frame_size = [50 ; 1000] ;

% Acoustic path Impulse Response
%     1 = Simple (FIR - 50 coeff)
%     2 = Real (FIR - 1000 coeff)
sim_AccPath_IR = 1 ;

%====== Default Parameters ======%

% Default Noise amplitude in Acoustic Path
default_AP_noise = 0;
% Default Delay
%   [2x1] vector (see AccPath_IR)
default_delay = [25 ; 100];
% Default filter length
%   [2x1] vector (see AccPath_IR)
default_Filterlength = [50 ; 300];

%====== Variation Parameters ======%

% Delay Simulation Parameters
sim_delay = ...

round(linspace(0,default_Filterlength(sim_AccPath_IR),sim_N_points));
% Noise Simulation Parameters
sim_noise_amplitude = linspace(0.01,2,sim_N_points);
% Filter length Simulation Parameters
if (sim_AccPath_IR == 1)
   sim_filter_length = round(linspace(20,100,sim_N_points));
else%if (AccPath_IR == 2)
   sim_filter_length = round(linspace(100,1000,sim_N_points));
end
% Convergence factor Simulation Parameters
sim_conv_factor(1,:) = ...
            linspace(nthroot(1E-6,4),nthroot(1E-
4,4),sim_N_points).^4;
sim_conv_factor(2,:) = ...

linspace(nthroot(0.001,4),nthroot(0.7,4),sim_N_points).^4;
sim_conv_factor(3,:) = ...
            linspace(nthroot(1E-6,4),nthroot(1E-
4,4),sim_N_points).^4;
sim_conv_factor(4,:) = ...
            linspace(nthroot(1E-4,4),nthroot(1E-
2,4),sim_N_points).^4;
sim_conv_factor(5,:) = ...

linspace(nthroot(0.001,4),nthroot(0.1,4),sim_N_points).^4;
```

```
% sim_conv_factor(6,:) = linspace(1,0.9999,sim_N_points);

%% Algo names

sim_algo_name(1,:) = 'LMS    ';
sim_algo_name(2,:) = 'NLMS   ';
sim_algo_name(3,:) = 'TV-LMS';
sim_algo_name(4,:) = 'NLMAD ';
sim_algo_name(5,:) = 'NLMP   ';
sim_algo_name(6,:) = 'RLS    ';

%% Simulink Model init

disp('****** Initialization');

MDL_Init_Script;

load_system(mdl);

FrameSize = sim_frame_size(sim_AccPath_IR);

if (sim_algo_no_RLS == 0 && sim_type == 0) || (sim_AccPath_IR == 2)
    set_param(mdl,'SimulationMode','rapid')
else
    set_param(mdl,'SimulationMode','normal')
end
set_param(mdl, 'StopTime', num2str(sim_N_frame*FrameSize));
set_param([mdl '/AcousNoiseGain'],'Gain',num2str(default_AP_noise));
set_param([mdl
'/Delta'],'Value',num2str(default_delay(sim_AccPath_IR)));
set_param([mdl '/AdaptiveParameter'],'Value', '0');
set_param([mdl '/AlgorithmSelection'],'Value', '1');

if (sim_AccPath_IR == 2)
    load('c_linear_1000_m1-4.mat');
    c = c1(1:FrameSize);
end

% Normalized energy
 c = c / abs(sum(c));

 % Calculate SNR and SNR db
 if (default_AP_noise ~= 0)
    SNR = (1/default_AP_noise)^2 ;
    SNR_dB = 10*log10(SNR) ;
 end

% Blank simulation
set_param([mdl '/FilterSizeValue'],'Value','0');
sim(mdl);
set_param([mdl '/FilterSizeValue'],'Value', ...

num2str(default_Filterlength(sim_AccPath_IR)));

%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% Simulations Time ! %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

e_mse = zeros(sim_N_points,1) ;
index = 1 ;

%% All algorithm simulation type
%
if (sim_algo == 0)
```

```
%

if sim_type == 0 %% No variation : Convergence simulation

    disp('****** Starting convergence simulation (all algorithm)');

    res_mse = zeros(sim_N_frame+1,6) ;
    res_mse_sm = zeros(sim_N_frame+1,6) ;
    res_time = zeros(1,6) ;
    res_mean_stable_mse = zeros(1,6) ;

    for i = 1:6
        disp(['Algorithm ' num2str(i)]);
        % Set algorithm selection
        set_param([mdl '/AlgorithmSelection'],'Value', num2str(i));

        if (i == 6 && sim_algo_no_RLS == 1)
            disp('ignored');
        else
            if (i == 6 && sim_AccPath_IR == 2)
                set_param(mdl,'SimulationMode','rapid')
            end

            % Start time measure
            tic;

            % Launch simulation
            sim(mdl);

            % Stop time measure
            toc;
            res_time(1,i) = toc ;

            % Save mean value (stable part)
            res_mean_stable_mse(1,i) = ...
                            mean(mse_val(round(length(mse_val)/3):end));

            % Save MSE
            res_mse(:,i) = mse_val;
            % Save MSE Smooth
            res_mse_sm(:,i) = mse_val_smooth;
        end
    end


    figure;
    semilogy(res_mse_sm);
    grid on;
    title('Convergence of algorithm')
    xlabel('Samples');
    ylabel('MSE');
    xlim([0 size(res_mse_sm,1)]);
    ylim([0 1]);
    legend(sim_algo_name,'Location','NorthEastOutside');
    set(gca,'Xcolor',[0.3 0.3 0.3]);
    set(gca,'Ycolor',[0.3 0.3 0.3]);

    % Copy axes to change color back to black
    Caxes = copyobj(gca,gcf);
    set(Caxes, 'color', 'none', 'xcolor', 'k', 'xgrid', 'off',...
              'ycolor','k', 'ygrid','off');
    delete(Caxes);
    % use delete(Caxes) if you want to get rid of the copy object
```

```
    % Can be useful for redraw purpose

elseif sim_type == 1 %% DELAY SIMULATION
    disp('****** Starting delay simulation (all algorithm)');

elseif sim_type == 2 %% NOISE SIMULATION

    disp('****** Starting noise simulation (all algorithm)');

    res_mse_sm = zeros(sim_N_frame+1,6,sim_N_points) ;
    res_mean_stable_mse = zeros(1,6,sim_N_points) ;
    res_SNR = zeros(1,sim_N_points) ;
    res_SNR_dB = zeros(1,sim_N_points) ;

    for w = sim_noise_amplitude

        % Calculate SNR and SNR_db
        if (w ~= 0)
           res_SNR(1,index) = (1/w)^2 ;
           res_SNR_dB(1,index) = 10*log10(res_SNR(1,index)) ;
        else
           res_SNR(1,index) = inf ;
           res_SNR_dB(1,index) = inf ;
        end

       disp(['*** SNR = ' num2str(res_SNR_dB(1,index)) ' dB']);
       % Set delay parameter to the model
       set_param([mdl '/AcousNoiseGain'],'Gain',num2str(w));

       for i = 1:6
          disp(['Algorithm ' num2str(i)]);
          % Set algorithm selection
          set_param([mdl '/AlgorithmSelection'],'Value', num2str(i));

          if (i == 6 && sim_AccPath_IR == 2)
             set_param(mdl,'SimulationMode','rapid')
          else
             set_param(mdl,'SimulationMode','normal')
          end

          % Launch simulation
          sim(mdl);

          % Save mean value (stable part)
          res_mean_stable_mse(1,i,index) = ...
                       mean(mse_val(round(length(mse_val)/3):end));

          % Save MSE Smooth
          res_mse_sm(:,i,index) = mse_val_smooth;

       end

       index = index + 1 ;
    end

    for n = 1:2:sim_N_points
       figure;
       semilogy(res_mse_sm(:,:,n));
       grid on;
       title(['Convergence of algorithm with SNR = ' ...
             num2str(res_SNR_dB(1,n)) ' dB'])
       xlabel('Samples');
       ylabel('MSE');
       xlim([0 size(res_mse_sm(:,:,n),1)]);
```

```
        ylim([0 1]);
        legend(sim_algo_name,'Location','NorthEastOutside');
        set(gca,'Xcolor',[0.3 0.3 0.3]);
        set(gca,'Ycolor',[0.3 0.3 0.3]);
    end

    res_mean_stable_mse = reshape(res_mean_stable_mse,6,sim_N_points);

    figure;
    plot(res_SNR_dB,res_mean_stable_mse');
    grid on;
    title('Mean MSE for the stable part')
    xlabel('SNR (dB)');
    ylabel('MSE');
    legend(sim_algo_name,'Location','NorthEastOutside');

elseif sim_type == 3 %% FILTER length SIMULATION

    disp('****** Starting filter length simulation (all algorithm)');

    res_mean_stable_mse = zeros(1,6,sim_N_points) ;
    res_time = zeros(1,6,sim_N_points) ;

    % Update StopTime to the max to avoid rebuilding at each iteration

set_param(mdl,'StopTime',num2str(sim_N_frame*max(sim_filter_length)))
;


    for j = sim_filter_length

        % Set new FrameSize value only if larger than default one
        if (j > default_Filterlength(sim_AccPath_IR))
            FrameSize = j;
            % Blank simulation (to not impact time measurements)
            set_param([mdl '/AlgorithmSelection'],'Value', '1');
            set_param([mdl '/FilterSizeValue'],'Value','0');
            set_param(mdl,'SimulationMode','normal')
            sim(mdl);
        end

        set_param([mdl '/Delta'],'Value',num2str(round(j/2)));
        set_param([mdl '/FilterSizeValue'],'Value',num2str(j));

        disp(['*** Filterlength = ' num2str(j)]);

        for i = 1:6
            disp(['Algorithm ' num2str(i)]);
            % Set algorithm selection
            set_param([mdl '/AlgorithmSelection'],'Value', num2str(i));

            if (i == 6 && sim_AccPath_IR == 2)
                set_param(mdl,'SimulationMode','rapid')
            end

            % Start time measure
            tic;

            % Launch simulation
            sim(mdl);

            % Stop time measure
            toc;
            res_time(1,i,index) = toc ;
```

```
            % Save mean value (stable part)
            res_mean_stable_mse(1,i,index) = ...
                            mean(mse_val(round(length(mse_val)/3):end));
            if (res_mean_stable_mse(1,i,index) > 1)
                res_mean_stable_mse(1,i,index) = NaN ;
            end
        end

        index = index + 1 ;
    end

    res_mean_stable_mse = reshape(res_mean_stable_mse,6,sim_N_points);
    res_time = reshape(res_time,6,sim_N_points);

    figure;
    semilogy(sim_filter_length,res_mean_stable_mse');
    grid on;
    title('Mean MSE for the stable part')
    xlabel('Filter length');
    ylabel('MSE');
    legend(sim_algo_name,'Location','NorthEastOutside');
    figure;
    plot(sim_filter_length,res_time');
    grid on;
    title('Elapsed time for algorithm')
    xlabel('Filter length');
    ylabel('Elapsed time');
    legend(sim_algo_name,'Location','NorthEastOutside');

end

%% Unique algorithm simulation type
%
elseif (sim_algo > 0)


if sim_type == 5 %% Convergence factor variation
    disp('**** Starting convergence factor simulation (unique
algorithm)');

    % Set algorithm selection
    set_param([mdl '/AlgorithmSelection'],'Value', num2str(sim_algo));

    res_mse = zeros(sim_N_frame+1,sim_N_points) ;
    res_mse_sm = zeros(sim_N_frame+1,sim_N_points) ;
    res_mse_val = zeros(2,sim_N_points) ;

    index = 1 ;
    for mu = sim_conv_factor(sim_algo,:)

        disp(['*** mu = ' mu]);

        set_param([mdl '/AdaptiveParameter'],'Value', num2str(mu));

        % Launch simulation
        sim(mdl);

        % Save MSE values
        res_mse_val(1,index) = mean(mse_val);
        res_mse_val(2,index) = min(mse_val);

        % Test for divergence
        if (res_mse_val(1,index) < 2)
            % Save MSE
```

```
            res_mse(:,index) = mse_val;
            % Save MSE Smooth
            res_mse_sm(:,index) = mse_val_smooth;
        end

        index = index + 1 ;
    end

    % Delete empty column
    for i = index-1:-1:1
        if (res_mse_val(1,i) > 2)
            res_mse(:,i) = [];
            res_mse_sm(:,i) = [];
        end
    end

    figure;
    semilogy(res_mse_sm);
    grid on;
    title(['Convergence of algorithm ', sim_algo_name(sim_algo,:),...
          ' (', num2str(sim_algo), ')'])
    xlabel('Samples');
    ylabel('MSE');
    xlim([0 size(res_mse_sm,1)]);
    ylim([0 1]);
    legendCell = cellstr(num2str( ...
          sim_conv_factor(sim_algo,1:size(res_mse,2))','mu = %-
.2e'));
    legend(legendCell,'Location','NorthEastOutside');
    set(gca,'Xcolor',[0.3 0.3 0.3]);
    set(gca,'Ycolor',[0.3 0.3 0.3]);

    % Copy axes to change color to black
    Caxes = copyobj(gca,gcf);
    set(Caxes, 'color', 'none', 'xcolor', 'k', 'xgrid', 'off',...
            'ycolor','k', 'ygrid','off');
    delete(Caxes);
    % use delete(Caxes) if you want to get rid of the copy object
    % Can be useful for redraw purpose

end

end

disp('****** END OF SCRIPT');
```

Script: main_script_LSC

```
%
%   MAIN SIMULATION SCRIPT
%       LMS IMPLEMENTATION
%       LINEAR SINGLE CHANNEL
%       FIR
%
%% Initialize
clear all
close all
%clc

tic

MDL_Init_Script
```

```
mdl = 'FxLMS_SC_IRTest';

disp('****** Parameters settings');
%====== Simulation Parameters ======%
% Simulation type
%   0 = Unique IR Response
%   1 = delay
%   2 = noise
%   3 = filter length
%   4 = estimated delay
%   5 = estimated noise
sim_type = 2;

% Number of simulation points
sim_N_points = 25 ;

% Number of frame
sim_N_frame = 2000 ;

% Frame size
%   [2x1] vector (see AccPath_IR)
sim_frame_size = [50 ; 1000] ;

% Acoustic path Impulse Response
%     1 = Simple (FIR - 50 coeff)
%     2 = Real (FIR - 1000 coeff)
sim_AccPath_IR = 2 ;

%====== Default Parameters ======%
% Noise amplitude
default_AP_noise = 0;
default_estAP_noise = 0;
% Default Delay
%   [2x1] vector (see AccPath_IR)
default_delay = [25 ; 150];
% Default estimated Delay
%   [2x1] vector (see AccPath_IR)
default_est_delay = [0 ; 0];
% Default filter length
%   [2x1] vector (see AccPath_IR)
default_Filterlength = [50 ; 300];

%====== Delay Simulation Parameters         ======%
sim_delay = ...

round(linspace(0,default_Filterlength(sim_AccPath_IR),sim_N_points));
%====== Noise Simulation Parameters         ======%
sim_noise_amplitude = linspace(0.01,2,sim_N_points);
%====== Estimated Delay Simulation Parameters         ======%
sim_est_delay = round( ...

[linspace(0,default_Filterlength(sim_AccPath_IR)/10,sim_N_points/2),.
..
    linspace(default_Filterlength(sim_AccPath_IR)/10+10, ...
            default_Filterlength(sim_AccPath_IR),sim_N_points/2)  ]);
%====== Estimated Noise Simulation Parameters         ======%
sim_est_noise_amplitude = linspace(0.01,100,sim_N_points);
%====== Filter length Simulation Parameters ======%
% Filter length Simulation Parameters
if (sim_AccPath_IR == 1)
   sim_filter_length = round(linspace(20,100,sim_N_points));
else%if (AccPath_IR == 2)
   sim_filter_length = round(linspace(100,1000,sim_N_points));
```

```
end

%% Simulink Model init
disp('****** Initialization');

load_system(mdl);

FrameSize = sim_frame_size(sim_AccPath_IR);

set_param(mdl, 'StopTime', num2str(sim_N_frame*FrameSize));

set_param([mdl '/AcousNoiseGain'],'Gain',...
                          num2str(default_AP_noise));

set_param([mdl '/EstNoiseGain'],'Gain',...
                          num2str(default_estAP_noise));

set_param([mdl
'/Delta'],'Value',num2str(default_delay(sim_AccPath_IR)));

set_param([mdl '/EstDelta'],'Value',...
                          num2str(default_est_delay(sim_AccPath_IR)));

set_param([mdl '/FilterSizeValue'],'Value', ...

num2str(default_Filterlength(sim_AccPath_IR)));

if (sim_AccPath_IR == 2)
    load('c_linear_1000_m1-4.mat');
    c = c1(1:FrameSize);
end

% Normalized energy
 c = c / abs(sum(c));

 % Calculate SNR and SNR_db
 if (default_AP_noise ~= 0)
    SNR = (1/default_AP_noise)^2 ;
    SNR_dB = 10*log10(SNR) ;
 end
 if (default_estAP_noise ~= 0)
    SNR_est = (1/default_estAP_noise)^2 ;
    SNR_est_dB = 10*log10(SNR_est) ;
 end
%% Simulations

e_mse = zeros(sim_N_points,1) ;
index = 1 ;

if sim_type == 0 %% UNIQUE IR SIMULATION
    disp('****** Starting unique IR simulation');

    % Launch simulation
    sim(mdl);

    % Calculate MSE
    e_mse = mean(mse_val(500:end))
    e_mmse = min(mse_val)

    % Plot IR and FR
    stem_MF_IR(c,h(sim_N_frame,:))
    plot_MF_spectrum(c,h(sim_N_frame,:))

    % Plot IR evolution
    figure;
```

```
    plot_gradient(h,round(logspace(0.1,2,10)))
    grid on;
    title('Filter Impulse Response evolution')
    xlabel('Samples');
    ylabel('h');

elseif sim_type == 1 || sim_type == 4 %% DELAY SIMULATION
    disp('****** Starting delay simulation');

    delta_values = 0 ;
    bloc_name = 'Delta' ;
    if sim_type == 1
       delta_values = sim_delay ;
    elseif sim_type == 4
       delta_values = sim_est_delay ;
       bloc_name = 'EstDelta' ;
    end

    e_mmse = zeros(sim_N_points,1) ;

    for delta = delta_values

       disp(['*** Delay = ' num2str(delta)]);

       % Set delay parameter to the model
       set_param([mdl '/' bloc_name],'Value',num2str(delta));

       % Launch simulation
       sim(mdl);

       % Calculate MSE
       e_mmse(index) = min(mse_val) ;
       e_mse(index) = mean(mse_val(round(length(mse_val)/3):end));
       index = index + 1 ;
    end


    figure;
    semilogy(sim_delay,e_mmse);
    grid on;
    title('Minimum MSE versus delay')
    xlabel('Delay (samples)');
    ylabel('Minimum MSE');

    figure;
    semilogy(sim_est_delay,e_mse);
    grid on;
    title('MSE (stable part) versus delay')
    xlabel('Delay (samples)');
    ylabel('MSE');

elseif sim_type == 2 || sim_type == 5 %% NOISE SIMULATION
    disp('****** Starting noise simulation');

    res_mse_sm = zeros(sim_N_frame+1,sim_N_points) ;
    res_mean_stable_mse = zeros(1,sim_N_points) ;
    res_SNR = zeros(1,sim_N_points) ;
    res_SNR_dB = zeros(1,sim_N_points) ;

    noise_values = 0 ;
    bloc_name = 'AcousNoiseGain' ;
    if sim_type == 2
       noise_values = sim_noise_amplitude ;
    elseif sim_type == 5
```

```
        noise_values = sim_est_noise_amplitude ;
        bloc_name = 'EstNoiseGain' ;
    end

    for w = noise_values

        % Calculate SNR and SNR_db
        if (w ~= 0)
           res_SNR(1,index) = (1/w)^2 ;
           res_SNR_dB(1,index) = 10*log10(res_SNR(1,index)) ;
        else
           res_SNR(1,index) = inf ;
           res_SNR_dB(1,index) = inf ;
        end

        disp(['*** SNR = ' num2str(res_SNR_dB(1,index)) ' dB']);

        % Set delay parameter to the model
        set_param([mdl '/' bloc_name],'Gain',num2str(w));

        % Launch simulation
        sim(mdl);

        % Save mean value (stable part)
        res_mean_stable_mse(1,index) = ...
                    mean(mse_val(round(length(mse_val)/3):end));

        % Save MSE Smooth
        res_mse_sm(:,index) = mse_val;

        % Calculate MSE
        e_mse(index) = min(mse_val) ;
        index = index + 1 ;
    end

%      plot(sim_noise_amplitude,e_mse);
%      grid on;
%      title('Minimum MSE versus noise')
%      xlabel('Noise amplitude');
%      ylabel('Minimum MSE');

    figure;
    plot(res_SNR_dB,res_mean_stable_mse);
    grid on;
    title('Mean MSE for the stable part')
    xlabel('SNR (dB)');
    ylabel('MSE');

elseif sim_type == 3 %% FILTER length SIMULATION

    disp('****** Starting filter length simulation');

    % Update StopTime to the max to avoid rebuilding at each iteration
    set_param(mdl, 'StopTime', ...

num2str(sim_N_frame*max(sim_filter_length)*2));

    for i = sim_filter_length

        disp(['*** Filterlength = ' num2str(i)]);

        % Set new delay parameter to the model
        set_param([mdl '/Delta'],'Value',num2str(round(i/2)));

        % Set FilterSize parameter
```

```
        set_param([mdl '/FilterSizeValue'],'Value', num2str(i));

        % Launch simulation
        sim(mdl);

        % Calculate MSE
        e_mse(index) = min(mse_val) ;
        index = index + 1 ;
    end

    plot(sim_filter_length,e_mse);
    grid on;
    title('Minimum MSE versus filter length')
    xlabel('Filter length (samples)');
    ylabel('Minimum MSE');

end

toc
disp('****** END OF SCRIPT');
```

Script: main_script_LSC_multichannel

```
%
%   MAIN SIMULATION SCRIPT
%       LMS IMPLEMENTATION
%       LINEAR SINGLE CHANNEL (with multichannel test)
%       FIR
%
%% Initialize
clear all
close all

tic

MDL_Init_Script
mdl = 'FxLMS_SC_IRTest_multichannel';

disp('****** Parameters settings');
%====== Simulation Parameters ======%
% Number of frame
sim_N_frame = 5000 ;

% Frame size
%   [2x1] vector (see AccPath_IR)
sim_frame_size = [50 ; 1000] ;

% Acoustic path Impulse Response
%     1 = Simple (FIR - 50 coeff)
%     2 = Real (FIR - 1000 coeff)
sim_AccPath_IR = 2 ;

%====== Default Parameters ======%
% Noise amplitude
default_AP_noise = 0;
default_estAP_noise = 0;
% Default Delay
%   [2x1] vector (see AccPath_IR)
default_delay = [25 ; 150];
% Default filter length
%   [2x1] vector (see AccPath_IR)
default_Filterlength = [50 ; 1000];
```

```
%% Simulink Model init
disp('****** Initialization');

load_system(mdl);

FrameSize = sim_frame_size(sim_AccPath_IR);

set_param(mdl, 'StopTime', num2str(sim_N_frame*FrameSize));

set_param([mdl '/AcousNoiseGain'],'Gain',...
                          num2str(default_AP_noise));

set_param([mdl '/EstNoiseGain'],'Gain',...
                          num2str(default_estAP_noise));

set_param([mdl
'/Delta'],'Value',num2str(default_delay(sim_AccPath_IR)));

set_param([mdl '/FilterSizeValue'],'Value', ...

num2str(default_Filterlength(sim_AccPath_IR)));

if (sim_AccPath_IR == 2)
    load('c_linear_1000_m1-4.mat');
    c1 = c1(1:FrameSize);
    c2 = c2(1:FrameSize);
    c3 = c3(1:FrameSize);
    c4 = c4(1:FrameSize);
end


%% Simulations
disp('****** Starting multichannel simulation');

% Launch simulation
sim(mdl);

% Calculate MSE
e_mse = mean(mse_val)

c = [c1 c2 c3 c4] ;
plots_C_spectrum(c,h(sim_N_frame,:)');
stems_C_IR(c,h(sim_N_frame,:)');

toc
disp('****** END OF SCRIPT');
```

Script: main_script_LMC

```
%
%   MAIN SIMULATION SCRIPT
%       LMS IMPLEMENTATION
%       LINEAR MULTI CHANNEL
%       FIR
%
%% Initialize
clear all
close all
% clc

tic
```

```
MDL_Init_Script

mdl = 'FxLMS_MC_IRTest';

disp('****** Parameters settings');
%====== Simulation Parameters ======%
% Simulation type
%   0 = 'Unique IR Response'
%   1 = 'Unique delay variation'
%   2 = 'Double delay variation'
%   3 = 'Full delay variation'
%   4 = 'Number of optimized points variation'
sim_type = 0;

% Number of simulation points
sim_N_points = 25 ;

% Number of frame
sim_N_frame = 10000 ;

% Frame size
%   [3x1] vector (see AccPath_IR)
sim_frame_size = [50 ; 1000 ; 1000] ;

% Number of acoustic path
sim_K = 4 ;

% Acoustic path Impulse Response
%     1 = Simple (FIR - 50 coeff)
%     2 = Real (FIR - 1000 coeff)
%     3 = Measured (FIR - 1000 coeff)
sim_AccPath_IR = 3 ;

%====== Default Parameters ======%
% Mu value
%   [3x1] vector (see AccPath_IR)
default_mu_value = [0 0 0.01];
% Error Gain
default_error_gain = [1 1 1 1];
% Estimate source Gain
default_est_source_gain = default_error_gain;
% Noise amplitude
default_AP_noise = 0;
default_estAP_noise = 0;
% Delay
default_delay = [300 304 668 672];
% Size of the filter
%   [3x1] vector (see AccPath_IR)
default_FilterLenght = [50 ; 1000 ; 1000];

% Delay
default_Unique_delay_var = [25 25 25 25];
% Delay
default_Double_delay_var = default_delay;
% Delay (optimal offset)
%   [3x4] vector (see AccPath_IR)
default_Full_delay_var = [0 0 4 4 ; 0 4 281 282 ; 0 4 368 372];

%====== Delay Simulation Parameters         ======%
sim_delay =
round(linspace(0,default_FilterLenght(sim_AccPath_IR),sim_N_points));
%====== Point Simulation Parameters         ======%
sim_point_amplitude = linspace(0,1,(sim_N_points/sim_K));
```

```
sim_point_amplitude =
sim_point_amplitude(2:length(sim_point_amplitude));

%% Simulink Model init
disp('****** Initialization');

load_system(mdl);

FrameSize = sim_frame_size(sim_AccPath_IR);

if (sim_AccPath_IR > 1)
    set_param(mdl,'SimulationMode','rapid')
else
    set_param(mdl,'SimulationMode','normal')
end

set_param(mdl, 'StopTime', num2str(sim_N_frame*FrameSize));
set_param([mdl '/AcousNoiseGain'],'Gain',num2str(default_AP_noise));
set_param([mdl '/FilterSizeValue'],'Value', ...

num2str(default_FilterLenght(sim_AccPath_IR)));

set_param([mdl
'/MuValue'],'Value',num2str(default_mu_value(sim_AccPath_IR)));

for k = 1:sim_K
    set_param([mdl '/EstNoiseGain'
num2str(k)],'Gain',num2str(default_estAP_noise));
    set_param([mdl '/Delta'
num2str(k)],'Value',num2str(default_delay(k)));
    set_param([mdl '/ErrorGain'
num2str(k)],'Gain',num2str(default_error_gain(k)));
    set_param([mdl '/EstSourceGain'
num2str(k)],'Gain',num2str(default_est_source_gain(k)));
end

if (sim_AccPath_IR == 2)
    load('c_linear_1000_m1-4.mat');
    c1 = c1(1:FrameSize);
    c2 = c2(1:FrameSize);
    c3 = c3(1:FrameSize);
    c4 = c4(1:FrameSize);
elseif (sim_AccPath_IR == 3)
    load('c_linear_meas_1000_m1-4.mat');
    c1 = c1(1:FrameSize);
    c2 = c2(1:FrameSize);
    c3 = c3(1:FrameSize);
    c4 = c4(1:FrameSize);
end

% Normalize
c1 = c1 / abs(sum(c1)) ;
c2 = c2 / abs(sum(c2)) ;
c3 = c3 / abs(sum(c3)) ;
c4 = c4 / abs(sum(c4)) ;

%% Simulations

e_mse = zeros(sim_N_points,1) ;
index = 1 ;

if sim_type == 0 %% UNIQUE IR SIMULATION
    disp('****** Starting unique IR simulation');
```

```
    % Launch simulation
    sim(mdl);

    % Calculate MSE
    e_mse = min(mse_val)

    % Plot results
    c = [c1 c2 c3 c4] ;
    plots_C_spectrum(c,h(sim_N_frame,:)');
    stems_C_IR(c,h(sim_N_frame,:)');

elseif sim_type == 1 %% UNIQUE DELAY SIMULATION
    disp('****** Starting unique delay simulation');

    e_mse = zeros(sim_N_points,sim_K,sim_K);
    for k = 1:sim_K
        disp(['*** Channel = ' num2str(k)]);
        % Default parameters
        index = 1 ;
        for kk = 1:sim_K
            set_param([mdl '/Delta'
num2str(kk)],'Value',num2str(default_Unique_delay_var(kk)));
        end
        % Delay variation
        for delta = sim_delay
            disp(['Delay = ' num2str(delta)]);
            % Set delay parameter to the model
            set_param([mdl '/Delta' num2str(k)],'Value',num2str(delta));
            % Launch simulation
            sim(mdl);
            % Calculate MSE
            for kk = 1:sim_K
                e_mse(index,kk,k) = min(mse_val(:,kk)) ;
            end
            % Update index
            index = index + 1 ;
        end
    end

    figure
    for k=1:sim_K/2
        subplot(2,sim_K/2,k);
        plot(sim_delay,e_mse(:,:,k));
        grid on;
        title(['Minimum MSE versus delay ' num2str(k)])
        xlabel('Delay (samples)');
        ylabel('Minimum MSE');
        subplot(2,sim_K/2,k+(sim_K/2));
        plot(sim_delay,e_mse(:,:,k+(sim_K/2)));
        grid on;
        title(['Minimum MSE versus delay ' num2str(k+(sim_K/2))])
        xlabel('Delay (samples)');
        ylabel('Minimum MSE');
    end
elseif sim_type == 2 %% DOUBLE DELAY SIMULATION
    disp('****** Starting double delay simulation');

    e_mse = zeros(sim_N_points,sim_K,sim_K);
    for k = 1:sim_K
        disp(['*** Channel = ' num2str(k)]);
        % Default parameters
        index = 1 ;
        for kk = 1:sim_K
```

```
            set_param([mdl '/Delta'
num2str(kk)],'Value',num2str(default_Double_delay_var(kk)));
        end
        % Delay variation
        for delta = sim_delay
            disp(['Delay = ' num2str(delta)]);
            % Set delay parameter to the model
            set_param([mdl '/Delta' num2str(k)],'Value',num2str(delta));
            if (k ~= 4)
                set_param([mdl '/Delta'
num2str(k+1)],'Value',num2str(delta));
            else
                set_param([mdl '/Delta'
num2str(1)],'Value',num2str(delta));
            end
            % Launch simulation
            sim(mdl);
            % Calculate MSE
            for kk = 1:sim_K
                e_mse(index,kk,k) = min(mse_val(:,kk)) ;
            end
            % Update index
            index = index + 1 ;
        end
    end

    figure
    for k=1:sim_K/2
        subplot(2,sim_K/2,k);
        plot(sim_delay,e_mse(:,:,k));
        grid on;
        title(['Minimum MSE versus delay ' num2str(k) '/'
num2str(k+1)])
        xlabel('Delay (samples)');
        ylabel('Minimum MSE');
        subplot(2,sim_K/2,k+(sim_K/2));
        plot(sim_delay,e_mse(:,:,k+(sim_K/2)));
        grid on;
        if (k+(sim_K/2) ~= 4)
            title(['Minimum MSE versus delay ' num2str(k+(sim_K/2)) '/'
num2str(k+(sim_K/2)+1)])
        else
            title(['Minimum MSE versus delay 4/1'])
        end
        xlabel('Delay (samples)');
        ylabel('Minimum MSE');
    end

elseif sim_type == 3 %% FULL DELAY SIMULATION
    disp('****** Starting full delay simulation');

    e_mse = zeros(sim_N_points,sim_K);

    % Delay variation
    for delta = sim_delay
        disp(['Delay = ' num2str(delta)]);

        % Set delay parameters to the model
        for k = 1:sim_K
            set_param([mdl '/Delta' num2str(k)],'Value', ...
                num2str(delta +
default_Full_delay_var(sim_AccPath_IR,k)));
```

```
        end

        % Launch simulation
        sim(mdl);

        % Calculate MSE
        for k = 1:sim_K
            e_mse(index,:) = min(mse_val) ;
        end

        % Update index
        index = index + 1 ;
    end

    d = sim_delay ;
    d_new =
linspace(0,default_FilterLenght(sim_AccPath_IR),5*sim_N_points);
    e_mse_int = zeros(length(d_new),sim_K);
    for k = 1:sim_K
        e_mse_int(:,k) = interp1(d,e_mse(:,k),d_new,'spline');
    end

    figure;
    plot(d_new,e_mse_int,d,e_mse,'xk');
    grid on;
    title(['Minimum MSE versus full delay (with optimal offset)'])
    xlabel('Delay (samples)');
    ylabel('Minimum MSE');

elseif sim_type == 4 %% Point optimization
    disp('****** Starting point optimization simulation');

    e_mse = zeros(length(sim_point_amplitude)*sim_K+1,sim_K);

    % Everything at zero
    for k = 1:sim_K
        set_param([mdl '/ErrorGain' num2str(k)],'Gain',num2str(0));
    end

    % Launch simulation
    sim(mdl);
    % Calculate MSE
    e_mse(1,:) = min(mse_val) ;

    index = 2;
    % Point variation
    for k = 1:sim_K
        disp(['*** Channel = ' num2str(k)]);
        for value = sim_point_amplitude
            disp(['****** Point amplitude = ' num2str(value)]);
            % Set delay parameter to the model
            set_param([mdl '/ErrorGain'
num2str(k)],'Gain',num2str(value));
            % Launch simulation
            sim(mdl);
            % Calculate MSE
            e_mse(index,:) = min(mse_val) ;
            index = index+1;
        end
    end
    x = linspace(0,4,length(sim_point_amplitude)*sim_K+1) ;
    x_new = linspace(0,4,5*(length(sim_point_amplitude)*sim_K+1));
    e_mse_int = zeros(length(x_new),sim_K);
```

```
    for k = 1:sim_K
        e_mse_int(:,k) = interp1(x,e_mse(:,k),x_new,'spline');
    end


    figure
    plot(x_new,e_mse_int);
    grid on;
    title('Minimum MSE versus number of optimized points')
    xlabel('Number of optimized points');
    ylabel('Minimum MSE');
end

toc
disp('****** END OF SCRIPT');
```

Script: delay_estimator_script

```
%
%   MAIN SIMULATION SCRIPT
%       Delay Estimator
%       FIR
%
%% Initialize
clear all
close all
% clc

addpath('../global');

disp('****** Parameters settings');
%====== Simulation Parameters ======%
% Frame size
%   [2x1] vector (see AccPath_IR)
sim_frame_size = [50 ; 1000 ; 1000] ;

% Acoustic path Impulse Response
%     1 = Simple (FIR - 50 coeff)
%     2 = Real (FIR - 1000 coeff)
%     3 = Measured (FIR - 1000 coeff)
sim_AccPath_IR = 3;

%% Simulink Model init
disp('****** Initialization');

FrameSize = sim_frame_size(sim_AccPath_IR);

if (sim_AccPath_IR == 1)
    load('c_linear_50_m1.mat');
    load('c_linear_50_m2.mat');
    load('c_linear_50_m3.mat');
    load('c_linear_50_m4.mat');
elseif (sim_AccPath_IR == 2)
    load('c_linear_1000_m1-4.mat');
    c1 = c1(1:FrameSize);
    c2 = c2(1:FrameSize);
    c3 = c3(1:FrameSize);
    c4 = c4(1:FrameSize);
elseif (sim_AccPath_IR == 3)
    load('c_linear_meas_1000_m1-4.mat');
    c1 = c1(1:FrameSize);
    c2 = c2(1:FrameSize);
```

```
    c3 = c3(1:FrameSize);
    c4 = c4(1:FrameSize);
end

%% Simulations
disp('****** Start simulation');

index = zeros(4,1);

index(1) = find(c1 == max(c1));
index(2) = find(c2 == max(c2));
index(3) = find(c3 == max(c3));
index(4) = find(c4 == max(c4));

optimal_delta = [   0 ; ...
                 abs(index(1)-index(2)) ; ...
                 abs(index(1)-index(3)) ; ...
                 abs(index(1)-index(4)) ];


disp(' Max peaks at index = ');
disp(index);

disp(' Optimal Delta (with channel 1 as reference) = ');
disp(optimal_delta);
```