Technical Report

Editor: Mikael Karlsson

Version 0.1

Status

Reviewed	Mikael Karlsson	2015-12-14
Approved	Hien Ngo	2015-12-XX

Project Identity

HT 2015
Linkoping University, ISY

Name	Responsibility	Phone	Email
Atheeq Ahmed	Group Member	070-4131687	athah298
Martin Andersson	Documentation Manager	073-3912657	maran703
Björn Ekman	Software Manager	070-2420137	bjoek586
Mikael Karlsson	Project Manager	076-2257939	mikka789
Valens Nsengiyumva	Hardware Manager	070-4131684	valns738
Oscar Silver	Test Manager	073-0589005	oscsi278

Customer: ISY, Linkoping University, 581 83, Linkoping

Customer contact: Hien Ngo, hien.ngo@liu.se
Examiner: Danyo Danev, danyo.danev@liu.se
Tutor: Antonios Pitarokoilis, antonios.pitarokoilis@liu.se

Contents

DOCUMENT HISTORY

Version	Date	Changes	Sign	Reviewed
0.1	2015-12-14	First Draft	Martin A	Mikael K

1 Introduction

The purpose of this document is to provide a detailed specification of the system, including explanations of both the functionality and the implementation. The system demonstrates the capabilities of Zero Forcing in the context of MIMO and Beamforming. The system demonstrates, by using Zero Forcing, that transmissions can be focused to a specific geographical point of interest and at the same time be suppressed at other specified points, to minimize the interference between different users in a wireless communication system.

1.1 MIMO

MIMO is an essential element of wireless communication that uses multiple antennas at both the transmitter and receiver to enhance the capacity of the radio link. Massive MIMO is a new innovative version of MIMO which uses a very large number of transmitter antennas that are operated in a completely coherent and adaptive manner to focus the transmission of signal energy into small regions of space.

1.2 Beamforming

Beamforming is one of the techniques that can be used in a MIMO system. The concept is that signals from the multiple antennas are transmitted in such a way that the signal energy gets focused at certain points in space by constructive and combining. In the case of Zero Forcing beamforming, the signal energy is focused at the relevant user while the other users essentially receive no signal at all. In the ideal case this would lead to zero interference between the different users of the communication system, a very valuable property to have in a communication system.

1.3 The Project

The project in the course TSKS05 CDIO Communication Systems has been to demonstrate the capabilities of Massive MIMO in an audio environment, building upon the work done by a previous project, Massive Audio Beamforming [7]. Using the hardware built in the previous project we have designed and implemented communication and Zero Forcing capabilities to the already existing system. The Zero Forcing and MIMO technique can ultimately be demonstrated in the implemented system by sending different data to two different terminals, simultaneously.

1.4 Aim and goals

See section 2.1 in the document Project Plan [5].

1.5 Definitions

See Table 1 for definitions of words used in this document.

Table 1: Definitions of words used in this document.

Word	Definition
MIMO	Multiple Input Multiple Output
A/D	Analog to Digital
D/A	Digital to Analog
L/M unit	Loudspeaker/Microphone unit
L/M-pair	A pair of Loudspeaker/Microphone units
MIMO-array	The array of L/M pairs used to generate the Zero Forcing beam
Terminal	One of the two units in the receiving L/M pair.
OS	Operating System
Subsystem	A part of the whole system

2 Overview of the System

The system implemented in this project is basically an upgrade of the existing system designed in the previous project. The system consists of two subsystems, the Software and the Hardware. The Hardware was not changed within this project and is therefore same as in the system given to us. On the other hand, the Software subsystem is completely redone from the last year's project. The new system is able to demonstrate Zero Forcing by simultaneous data transmission to two different terminals, both receiving different data at the same time. An overview of the system is depicted in Figure 1.

Briefly, the Hardware consists of eight L/M pairs of which seven pairs act like a MIMO-array while one pair receives the signals from the MIMO-array, a distribution box, A/D and D/A converters as well as the actual computer.

The Software subsystem consists of MatLab scripts and functions for controlling the interface to the A/D and D/A converters, channel estimation, MIMO combining, channel coding and calibration. It also includes the drivers for communication between a Windows operative system and the Hardware.

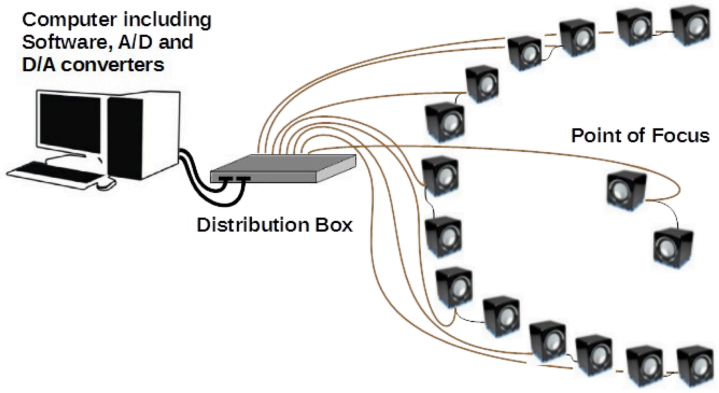

Figure 1: Overview of the Hardware in the system. [7, Fig. 1]

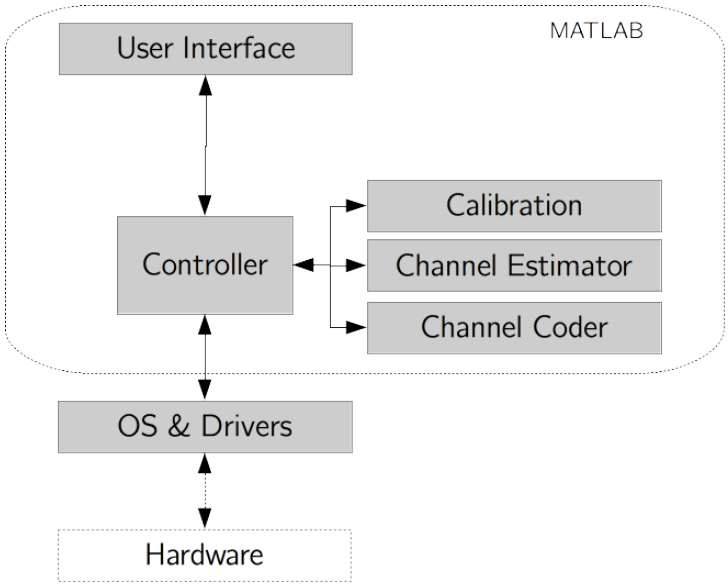

Figure 2: Overview of the software.

3 Software

The Software is built in a modular fashion with well defined interfaces in between the different modules. As can be seen in Figure 2, the system has one main module called *Controller* which takes care of all communication with the different modules as well as with the drivers and interfaces to the A/D- and D/A-cards. This section will try to summarize and explain the main tasks as well as the functions and scripts of the different modules.

3.1 General Implementation of the Modules

The different modules in Figure 2 are all coded in *MatLab* in the form of different functions and scripts. Scripts are mainly used in two ways, as main scripts to run the system on a high level and as initialization scripts to set certain values needed through out the runs.

Functions are mainly used for functionality as well as readability in the main scripts.

Some coding guidelines have been used when writing the Software:

- Comments are written in such a way that the built in MatLab-functions "help" and "doc" can be used with each function.
- Uppercase is used for constants.
  Example: *CONSTANT_VARIABLE*.

- Global constant have the letter "G" added to them.
  Example: *CONSTANT_GLOBAL_G*.
- Camelcase is used for all other variable names, as well as functions and scripts
  Example: *variableName*.
- All script names ends with *Script*.
  Example: *mainScript*.

3.2 Controller

The Controller is responsible for the general flow of the program and supplies data to and from the other modules. It also contains the functions necessary to communicate with the hardware drivers. The controller contains the main loop of the system.

As for the controller functions, these are functions to communicate with the hardware drivers. Thus, the functions for sending and receiving data are in the Controller, since these are functions that need to interact with the drivers for the D/A and the A/D converters.

3.2.1 Implementation

The Controller is built up by several scripts and functions which are described below.

mainScript

This script is a wrapper for the `controllerMainScript` that provides the user with the choice to use the default parameters or to set the parameters manually. If the user chooses to set the parameters manually `uiScript` will be started.

controllerMainScript

The `controllerMainScript` manages the flow of a test run. It uses basically all other functions and scripts described in this document in order to estimate the channel, send data to the terminals and then process the received data. If one is fine with slight changes to the source code, then this script offers some more configuration possibilities than the UI.

initSWGlobalsScript

This script initiates all global variables that are not directly associated with the hardware and its drivers. Description of the different variables and their default values can be seen in Table 2.

Table 2: Global constants set in initSWGlobalsScript

Variable	Description	Default Value
CARRIER_FREQ_G	The passband carrier frequency used by the system.	502

SAMPLE_FREQ_G	The sample frequency of the AD and DA converters.	6060
ARRAY_PAIRS_G	Pair indices to the LM-pairs acting as array.	[1,2,3,4,6,7,8]
TERMINAL_PAIRS_G	Pair indices to the LM-pairs acting as terminals.	[5]
AMOUNT_LM_GROUPS_G	Number of different groups for the LM-pairs. Group selected through hardware switch.	2
TERMINAL_GROUP_G	ID for the terminal group, in our case this corresponds to switches in the down-position.	0
ARRAY_GROUP_G	ID for the array group, in our case this corresponds to switches in the up-position.	1
PSK_K_G	The k value of the PSK modulation, e.g. 1 for BPSK, 2 for QPSK.	1
PSK_M_G	The M value of the PSK modulation, e.g. 2 for BPSK, 4 for QPSK.	2
HAMMING_K_G	k value of the Hamming code used for error control.	4
HAMMING_N_G	n value of the Hamming code used for error control.	7

initHWGlobalsScript

This script initiates all global variables that are directly associated with the hardware and its drivers. This script can only be used on the lab computer. It uses some of the globals initialized in the `initSWGlobalsScript`, which must be run before this script. The variables set in this script are described in Table 3. The Data Acquisition Toolbox supplies functions and example to find these strings should the hardware be changed.

Table 3: Global constants set in initHWGlobalsScript

Variable	Description	Default Value
ADAPTOR_NAME_G	A string with the HW card suppliers name.	'contec'
AD_NAME_G	The model number of the AD card.	'AD12-64'
DA_NAME_G	The model number of the DA card.	'AIO001'
DIO_NAME_G	The model number of the AD card (this card also has four digital input/output lines).	'AD12-64'
AO_G	A Data Aqcuisition Toolbox analog output object.	-
AI_G	A Data Aqcuisition Toolbox analog input object.	-

importGlobalsScript

Imports the globals initiated in initSWGlobalsScript and initHWGlobalsScript.

sendReceive()

This function does the simultaneous sending and receiving using the Matlab Data Acquisition (DAQ) Toolbox objects and methods.

The input signal should contain an even number of rows between 2-16 (one row for each speaker).

Hard coded in this file are the scaling factors determined by the calibration scripts using a sound intensity meter. These values are used to make sure that each speaker sends equally strong for the same software input and receives the same software values for equally strong input audio signal.

Table 4: In and out parameters for the `sendReceive()` function.

In parameters	**Description**
sampleVec	A vector consisting of different streams of samples. Each row corresponds to one of the transmitting units.
analogOutput	AO Object used for sending through D/A converter.
analogInput	AI Object used for receiving data through A/D converter.
sendingLMPairsVec	Vector consisting of the number corresponding to the transmitting pairs.
receivingLMPairsVec	Vector consisting of the numbers corresponding to the receiving pairs.
sampleFreq	Sample frequency
sendingGroup	Which group that are sending, corresponds to variable in the importSWGlobals script.
Out parameters	**Description**
receivedSampleVec	A vector consisting of the different received streams of samples. Each row corresponds to one of the receiving units.
timeVec	A vector consisting of the point in time of each sample in receivedSignalVec.

sendReceiveData()

This function sends input data from the MIMO array and outputs what is received at the terminals.

Table 5: In and out parameters for the `sendReceiveData()` function.

In parameters	**Description**
sampleVec	A vector consisting of different streams of samples. Each row corresponds to one of the transmitting units.
Out parameters	**Description**
receivedSampleVec	A vector consisting of the different received streams of samples. Each row corresponds to one of the receiving units.

sendReceivePilots()

This function sends input data from the terminals and outputs what is received at the MIMO array.

Table 6: In and out parameters for the `sendReceivePilots()` function.

In parameters	**Description**
sampleVec	A vector consisting of different streams of samples. Each row corresponds to one of the transmitting units.
Out parameters	**Description**
receivedSampleVec	A vector consisting of the different received streams of samples. Each row corresponds to one of the receiving units.

simulateSendReceive()

This function emulates an AWGN channel with optional delay and echo. In short it:

- Adds AWGN noise to all streams
- Adds echo (set `echoGain` to zero to remove it)
- Attenuates the original signal
- Delays the input streams same amount for all streams

For the input variables they are all declared in the struct `SIMU` in the `controllerMainScript` and contains the fields mentioned in the Table 7. The output matrix will have `simu.syncError` + `simu.echoDelay` + `simu.delay` extra samples.

Table 7: In and out parameters for the `simulateSendReceive()` function.

In fields	**Description**
syncError	Possible to force the synchronization to be some samples off, though this requires the signals to be long enough. This many zeros are added at the end.
echoDelay	Delay the echo this many samples extra compared to the original signal delay.
snr	The SNR of the received signal (not taking echo into account).
chGain	The gain/attenuation added by the channel to both the echo and original signal.
echoGain	Extra gain/attenuation added to the echo.
Out parameters	**Description**
data	The simulated result of sending. The matrix has the same amount of rows as `passbandSig` (ie. not a correct MIMO simulation)

pairIndexs2UnitIndexs()

A function that returns the corresponding LM-unit-indices vector to an input pair-index vector.

Table 8: In and out parameters for the `pairIndexs2UnitIndexs()` function.

In parameters	**Description**
pairIndexs	A vector of pair indexes.
Out parameters	**Description**
unitIndexs	A vector of unit indexes corresponding to input pair indexes.

3.3 UI

The UI is text-based and is responsible for allowing the user to set system parameters manually. The usage of the UI is further described in the user manual [4].

3.3.1 Implementation

uiScript

This script provides the option to manually change the values of some global variables. The variables that can be changed are listed in Table 9.

Table 9: Variables possible to change in uiScript

Variable	**Description**	**Default Value**
CARRIER_FREQ_G	The passband carrier frequency used by the system.	505
SAMPLE_FREQ_G	The sample frequency of the AD and DA converters.	6060
ARRAY_PAIRS_G	Pair indices to the LM-pairs acting as array.	[1,2,3,4,6,7,8]
TERMINAL_PAIRS_G	Pair indices to the LM-pairs acting as terminals.	[5]
PSK_K_G	The k value of the PSK modulation, e.g. 1 for BPSK, 2 for QPSK.	1
PSK_M_G	The M value of the PSK modulation, e.g. 2 for BPSK, 4 for QPSK.	2
HAMMING_K_G	k value of the Hamming code used for error control.	4
HAMMING_N_G	n value of the Hamming code used for error control.	7

USE_INTERLEAVER_G	Boolean that decides whether to use interleaver or not in the channel coding.	True
USE_RRC_FILTER_G	Boolean that will use a root raised cosine filter if true, otherwise a rectangular filter.	False
USE_K_MEAN_CLUSTERING_G	Boolean that decides whether or not phase detection based on K-mean clustering will be used.	True
USE_POWER_CONTROL_G	Boolean that decides wheter or not to use power control.	True
USE_ZF$_G$	Boolean that decides wheter to use Zero Forcing (true) or Maximum Ratio Combining (false).	True

uiQuestion()

Helper function for uiScript that gets an answer to a yes/no question from the user. Parameters are described in Table 10.

Table 10: In and out parameters for the `uiQuestion()` function.

In parameters	**Description**
question	A string with a question asking for a yes/no answer.
Out parameters	**Description**
answer	A boolean that is 1 if the answer was yes, 0 if no.

uiValuePrompt()

Helper function to uiScript that prompts the user for an input value and returns that value. Parameters are described in Table 11.

Table 11: In and out parameters for the `uiValuePrompt()` function.

In parameters	**Description**
prompt	A string prompting for a value.
Out parameters	**Description**
answer	The input value.

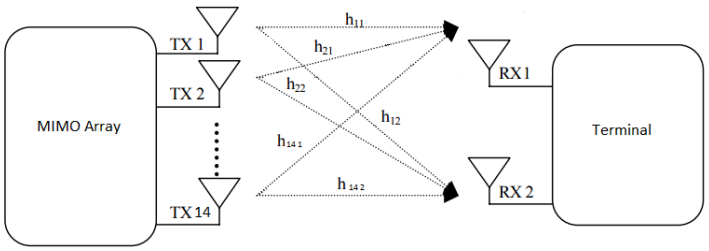

Figure 3: Channels in the MIMO system.

getAvailableTerminalPairs()

Helper function to uiScript that uses the set Array pairs and calculates what terminal pairs are available to use without collision. Parameters are described in Table 12.

Table 12: In and out parameters for the `getAvailableTerminalPairs()` *function.*

In parameters	**Description**
arrayPairs	A vector containing the array pairs currently in use.
Out parameters	**Description**
terminalPairs	A vector containing the available terminal pairs.

3.4 Channel Estimator

The Channel Estimator is used to estimate the impulse responses of each channel between the two terminals to each L/M-unit in the L/M-array, the channels are depicted in Figure 3. In essence it is used to estimate the true channel matrix H defined in (1).

$$H := \begin{bmatrix} h_{1\,1} & h_{1\,2} \\ h_{2\,1} & h_{2\,2} \\ \vdots & \vdots \\ h_{14\,1} & h_{14\,2} \end{bmatrix} \tag{1}$$

The information for each channel is represented in H with a complex number where its absolute value α represents the amplitude attenuation and its angle ϕ the phase shift introduced by the channel. The estimate of the channel between unit i and j, $\hat{h}_{i,j}$, is therefore defined as in (2).

$$\hat{h}_{i,j} = \hat{\alpha}_{i,j}^{\hat{\phi}_{i,j}} \tag{2}$$

The channels are estimated by sending a known signal from the two terminals and recording what the L/M-array receive. By comparing the known sent signal with the received one, the characteristics of the channel are estimated.

Estimation

Amplitude is estimated by simply taking an average of the peaks of the received signals. While cross-correlation can be used to estimate the phase shift introduced by the channel it is not accurate enough for our purposes. The cross-correlation between two signals, s_0 and s_1, is given in Equation (3) for the time discrete case. The phase shift τ between two signals can then be estimated as the argument of the maximum of the cross-correlation, as stated in Equation (4). The accuracy of the cross-correlation is improved if the signal is not periodic, therefore we add a BPSK part in the pilot along with the sinusoid signal.

$$(s_0 \star s_1)[n] = \sum_m s_0^*[m] s_1[m+n] \tag{3}$$

$$\tau = \arg\max_n \ (s_0 \star s_1)[n] \tag{4}$$

Phase Transfrom

A way to sharpen the peak of the cross-correlation and to better handle correlated noise is to use a generalized cross correlation involving spectral weighting. The phase transform (PHAT) method [2] whitens the signal spectrum and has been shown to be optimal for minimizing the variance of the time delay estimate [3]. Applying PHAT improves our results when compared with just using simple cross-correlation. Mathematically the PHAT corresponds to equation (7).

$$Y(f) = \mathscr{F}(s_0 \star s_1) \tag{5}$$

$$\theta(f) = \frac{Y(f)}{|Y(f)|} \tag{6}$$

$$\tau = \arg\max_n \ \mathscr{F}^{-1}(\theta(f)) \tag{7}$$

Zero Crossings

The PHAT method will give an estimate of the delay with a maximum 1 sample precision. Due to the constraints introduced by hardware even this is not enough for our system. Therefore a second method, comparing zero-crossings (ZC) of the signals, is also used. A zero-crossing is simply when the signal crosses zero. For this estimation the sinusoid part of the pilot is used due to its periodicity. By interpolating between samples and comparing positive zero-crossings between two signals we are able to get sub-sample precision of the phase. Note that due to the periodicity of the signal (which

increases accuracy since we can average over multiple measurements) the ZC method will only output values in the range $[0, 2\pi)$.

Our Solution

We combine these two methods to get an estimate of both the number of multiples of 2π and the phase within 2π. The PHAT method is used to estimate the multiples of 2π in the phase shift by simply flooring the result to the nearest multiple of 2π. Then the ZC method is used to estimate the phase within the period (2π). Due to noise we can end up in the situation depicted in Figure 4, where the PHAT method estimates the phase to be just under 2π, and the ZC method estimates it to be just over 2π. This would result in the combined estimate to be just over 0 since no multiples of 2π were detected by the PHAT method. From the PHAT method we can calculate what phase we should expect to get from the ZC method by taking the PHAT result modulo 2π. If this expected phase differ too much from the result from the ZC method then we use the expected value instead since we probably have the situation seen in Figure 4.

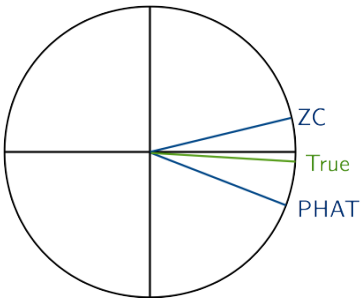

Figure 4: Potential problem due to noise in the phase estimation.

To avoid transients a window of the signals is used for the respective methods. Note that we are only interested in relative phase shifts between the signals and not in shifts introduced due to delays in software or hardware, therefore we subtract all phase estimates with the minimum before returning from the function.

Training Pilot

The Channel Estimator module is also responsible for generating the pilot to be sent during training. This is achieved with the `constructPilot()` function detailed in section 3.4.1. Sending the pilot with a high and constant amplitude will ease the amplitude estimation and hopefully overpower most of the noise. By assuming reciprocity for the channel, the pilot only has to be sent from the two terminals instead of all the units in the MIMO array. The pilot consists of two different types of signals, the first half is a pure sinusoid and the second half is a BPSK modulated signal. The pure sinusoid is used for the Zero Crossing method of the phase estimation since it has the required periodicity. The BPSK signal is suitable for the PHAT method since it is based on a random data stream and thereby the cross-correlation gives a clear peak.

3.4.1 Implementation

The Channel Estimator module is called in the main script using just the function `performChannelEst()`.

performChannelEst()

This function performs the estimation for *all* terminals using the `channelEstMain()` function and combines the results from the different runs into two matrices containing information about the amplitude and phase. Table 13 details its in and out parameters.

Table 13: In and out parameters for the `performChannelEst()` function.

In parameters	**Description**
carrierFreq	The carrier frequency for the pilot.
sampleFreq	The sampling frequency used for sending and receving the pilot.
Out parameters	**Description**
ampEst	Estimates of the amplitude for the different channels, matrix [terminals x units].
phaseEst	Estimates of the phase for the different channels, matrix [terminals x units].

channelEstMain()

The function `channelEstMain()`, see Table 14 for parameters, is used to estimate the amplitude and phase respectively as perceived from *one* terminal. It has two subfunctions, `channelEstAmp` and `channelEstPhase`.

Table 14: In and out parameters for the `channelEstMain()` function.

In parameters	**Description**
pilot	The signal used as training pilot, row vector [samples]
recordedSignals	The signals recorded when the pilot was sent, matrix [units x samples]
carrierFreq	The carrier frequency for the pilot.
sampleFreq	The sampling frequency used for sending and receiving the pilot.
Out parameters	**Description**
amplitudeEst	Estimates of the amplitude for the different channels, row vector [units].
phaseEst	Estimates of the phase for the different channels, row vector [units].

channelEstAmp()

This function, see Table 15 for parameters, is used to estimate the amplitudes of the received signals.The amplitudes are estimated using the samples from a selected window of the received signals and taking an average of the peaks. The window used is [0.1, 0.4] of the pilot if the full pilot is seen as the interval [0,1]. This window is used to avoid transients and the sinusoid part of the pilot is used as it is more consistent in amplitude.

Table 15: In and out parameters for the `channelEstAmp()` function.

In parameters	**Description**
recordedSignals	The signals recorded when the pilot was sent, matrix [units x samples]
Out parameters	**Description**
ampEst	Estimates of the amplitude for the different recorded signals, row vector [units].

channelEstPhase()

This function, see Table 16 for parameters, is used to estimate the phase of the received signals. It in turns calls `channelEstPhasePHAT()` and `channelEstPhaseCrossing()`.

Table 16: In and out parameters for the `channelEstPhase()` function.

In parameters	**Description**
pilot	The signal used as training pilot, row vector [samples]
recordedSignals	The signals recorded when the pilot was sent, matrix [units x samples]
carrierFreq	The carrier frequency for the pilot.
sampleFreq	The sampling frequency used for sending and receiving the pilot.
Out parameters	**Description**
phaseEst	Estimates of the phase for the different channels, row vector [units].

channelEstPhasePHAT()

This function performs a phase estimation using the phase transform (PHAT) and its parameters are detailed in Table 17.

Table 17: In and out parameters for the `channelEstPhasePHAT()` function.

In parameters	**Description**
pilot	The signal used as training pilot, row vector [samples]
recordedSignals	The signals recorded when the pilot was sent, matrix [units x samples]
carrierFreq	The carrier frequency for the pilot.
sampleFreq	The sampling frequency used for sending and receiving the pilot.
Out parameters	**Description**
phaseEst	Estimates of the phase for the different channels, row vector [units].

channelEstPhaseCrossing()

This function performs a phase estimation using zero crossings and its parameters are detailed in Table 18.

Table 18: In and out parameters for the `channelEstPhaseCrossing()` function.

In parameters	**Description**
pilot	The signal used as training pilot, row vector [samples]
recordedSignals	The signals recorded when the pilot was sent, matrix [units x samples]
carrierFreq	The carrier frequency for the pilot.
sampleFreq	The sampling frequency used for sending and receiving the pilot.
Out parameters	**Description**
phaseEst	Estimates of the phase for the different channels, row vector [units].

constructPilot()

This function is used to generate the pilot, the parameters are given in Table 19. The pilot consists of two different types of signals, the first half is a pure sinusoid and the second half is a BPSK modulated signal.

Table 19: In and out parameters for the `constructPilot()` function.

In parameters	**Description**
carrierFreq	The carrier frequency for the pilot.
timeDuration	How long the resulting pilot should be in seconds.
sampleFreq	The sampling frequency used for sending and receiving the pilot.
amplitude	The amplitude of the pilot.
Out parameters	**Description**
pilot	The generated pilot, row vector.

3.5 Channel Coder

The channel coder includes all functions that have to do with the actual channel coding. This include functions for creating bit streams, modulation, filtering, interleaving, error control and so on. A high level description of how the channel coder functions are used can be seen in Figure 5. The MIMO block will be described in some more detail before the implementation of all blocks are presented.

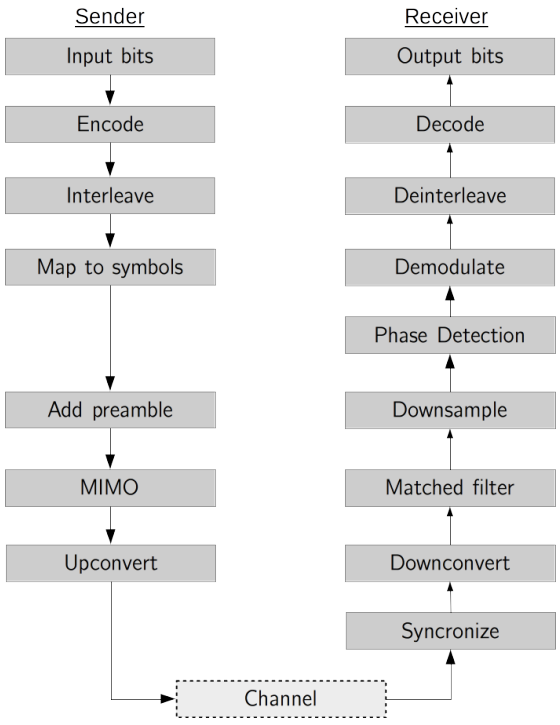

Figure 5: Overview of the flow of the channel coding.

3.5.1 MIMO

This part of the Channel Coder performs Zero Forcing precoding for an antenna array. This module will combine the results from the Channel Estimator with the data from the Channel Coder according to a MIMO-combining technique, in this case Zero-Forcing, in order to send signals to each terminal. The output will be the time signals for each L/M-unit in the MIMO array. An overview of this setup is depicted in Figure 6.

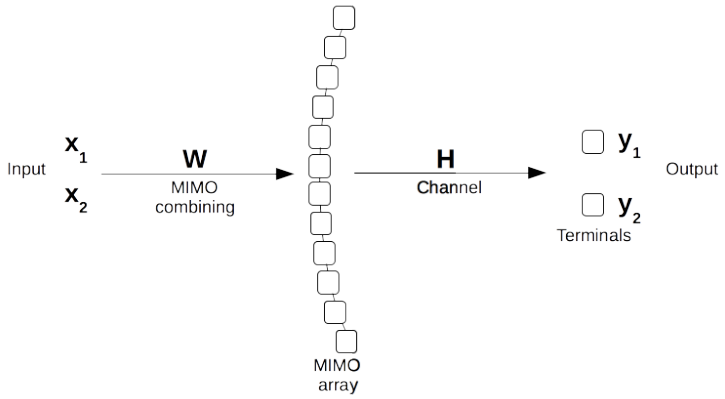

Figure 6: Overview of MIMO system with Zero Forcing.

3.5.2 Implementation

The implementation of the Controller module consists of several different functions and a script used for plotting. They are all described below.

createBitStream()

Creates a matrix of random bits. The size of the matrix is nrOfStreams $\times$ streamLength.

Table 20: In and out parameters for the `createBitStream()` function.

In parameters	**Description**
nrOfStreams	The number of streams in the output vector.
streamLength	The length of the streams in the output vector.
Out parameters	**Description**
bitVector	A vector of random bits.

hammingEncode()

Adds redundancy bits to input bit vector by using a Hamming code with input n and k values. For the input bit vector to work with the Hamming code, zeros are added at the

end of input vector. The number of zeros added are specified in the `added` variable. Each row of the input and output bit vectors represents different streams.

Table 21: In and out parameters for the `hammingEncode()` function.

In parameters	**Description**
bitVec	A vector of bits where each row represents different stream.
hammingN	The n value of the used Hamming code.
hammingK	The k value of the used Hamming code.
Out parameters	**Description**
errCtrlBitVec	The Hamming encoded bit vector. Each row represents different streams.
zerosAdded	The number of zeros added at the end of the input bit vector to make its dimensions match with the Hamming code.

mapToSymbols()

Maps a vector of binary numbers to a vector of complex numbers. The mapping uses an M-ary, graycoded PSK constellation where $M = 2^k$ and k is input to the function. The initial phase is set to 0.

Table 22: In and out parameters for the `mapToSymbols()` function.

In parameters	**Description**
bitVector	A vector of bits.
k	k value of the PSK modulation.
Out parameters	**Description**
symbolVector	A vector of complex symbols, mapped from the input bit vector.

createPreamble()

This function creates a preamble to be sent on the channels, see Table 23 for parameters. The preamble consists of 18 bits (this is long enough and is also divisible by both two and three as required by the `mapToSymbols()` function) which are then mapped to symbols using the current modulation scheme.

The preambles for the first two streams are hardcoded in this file. Stream number three and higher uses 18 random bits.

Table 23: In and out parameters for the `createPreamble()` function.

In parameters	**Description**
nStreams	Number of streams supposed to be in the output.
k	Bits per symbol.
Out parameters	**Description**
preambleSymb	The output symbol vector representing the 18 bits for each stream. Each row represents one stream.

MIMOCombining()

This function performs Zero-Forcing precoding on the input data to achieve parallel data transmission. The return value is the data to be sent from each unit in the antenna array. Note that the output data is scaled to range between 0 and 1.

The idea is to use the estimated knowledge of the channel between each unit in the antenna array and each terminal pair and invert that to achieve signal separation to each of the terminal units. The inversion of the channel estimation is calculated by using the pseudo-inverse as described in equation (8). If Maximum Ratio Combining is performed instead then equation (9) is used. The precoding is then done by multiplying the data to be sent with the pseudo-inverse W. Assuming perfect channel estimation and no noise in the transmission this would achieve parallel data transmission. But even though noise is present and with an imperfect channel estimation this technique is good enough to achieve error-free parallel data transmission in most setups.

$$W = H^*(HH^*)^{-1} \quad \text{for Zero Forcing} \tag{8}$$

$$W = H^* \quad \text{for Maximum Ratio combining} \tag{9}$$

Power control is then applied to W columnwise, as per equation (10), in order to normalize the power each terminal receive. The parameters of the MIMOCombining function are described in table 24.

$$W_i = \frac{W_i}{||W_i||} \tag{10}$$

Table 24: In and out parameters for the `MIMOCombining()` function.

In parameters	**Description**
data	Data to be sent through the Zero-Forcing precoder.
Hamp	Matrix of amplitudes from channel estimation.
Hphase	Matrix of phases from channel estimation.
Out parameters	**Description**
MIMOComplexVec	Matrix containing MIMO precoded data.

createFilters()

This function creates two Root Raised Cosine (RRC) filters, one transmit filter and one receiver filter. Used together these filters should minimize ISI and concentrate signal power to the passband. If useRRCFilter is false, the transmit filter will be destroyed in the end, signaling to other functions/scripts not to use it.

Table 25: In and out parameters for the `createFilters()` function.

In parameters	**Description**
useRRCFilter	A boolean. If false the transmit filter is "destroyed" by setting to a numeric value.
samplesPerSymbol	The upsampling factor.
span	The length of the filter in number of symbols.
rolloff	The rolloff factor of the RRC filter. 1 gives sharp edges, 0 gives very smooth edges.
Out parameters	**Description**
hRxFilter	Communication toolbox object for a receive RRC filter.
hTxFilter	Communication toolbox object for a transmit RRC filter. Or a numeric (if useRRCFilter)

upConvert()

Up-converts a baseband signal to passband. First an up-sampling is done, either using a rectangular filter or a RRC-filter. Then up-conversion is done by multiplying with a complex exponential at the carrier frequency and keeping the real part of that multiplication.

Table 26: In and out parameters for the `upConvert()` function.

In parameters	**Description**
symbolMatrix	Symbol vectors to be upconverted. Each row corresponds to different streams.
carrierFreq	The carrier frequency for the upconversion.
sampleFreq	The sample frequency.
samplesPerSymbol	Amount of samples per symbol in the passband signal.
hTxFilter	Optional. A compsys toolbox filter or a numeric. If numeric, or not supplied at all, rectangular pulseshaping will be applied.
Out parameters	**Description**
sampleMatrix	The created passband signal.
carrierMatrix	A corresponding carrier matrix (the carrier repeats in all rows).
timeVec	The corresponding time lags for each sample in sampleMatrix.

syncSignal()

This function looks for the preamble in the `receivedData` and returns all samples after the identified preamble.

It works row by row through the `preambleMatrix` and `receivedData` finding the lag of the preamble using the `xcorr` function. The maximum delay is connected to how much extra the system needs to stop and listen after each send.

Table 27: In and out parameters for the `syncSignal()` function.

In parameters	**Description**
preambleMatrix	The set of preambleSymbols upconverted to the passband.
receivedData	Data received from the channel.
maxLags	Maximum number of lags.
signalLength	Number of samples with actual information content.
Out parameters	**Description**
syncedData	The received signal but with lag and preamble samples removed from the beginning and only signalLength samples long.
allCorrelations	The result of each xcorr call.
allLags	The lag vectors.
delays	The number of samples the received signal lagged.

filterSignal()

Implements both a LP plus a moving average or a RRC filter. Uses the variable `useRRCFilters` to determine which filter output to use as output for this function.

Table 28: In and out parameters for the `filterSignal()` function.

In parameters	**Description**
signal	Either the inphase or quphase component of an unfiltered received signal.
samplesPerSymbol	The number of samples per symbol.
hRxFilter	A Communication Systems Toolbox RRC receiver filter object.
span	Number of symbols that the filter spans.
useRRCFilters	A boolean, if true the RRC filter is used, otherwise a LP filter plus a moving average filter is applied.
Out parameters	**Description**
filtSig	The filtered output.
startOffset	The delay introduced by the filters (in number of samples). Can also be viewed as the number of samples before the first sampling point.

kMeanClustering()

This function performs k-Mean clustering on the recevied symbols and outputs the detected phase shift of the centroids, the found centroid's locations as well as a vector of all input data points moved to their respective centroid.

Table 29: In and out parameters for the `kMeanClustering()` function.

In parameters	**Description**
data	Should be a complex vector of input data points.
nrClusters	Number of clusters, i.e. 2 for BPSK, 4 for QPSK and so on.
Out parameters	**Description**
phase	The phase of shift of the clusters.
centroids	The centroids' locations, each row is a centroid where column 1 is the real part and column 2 is the imaginary part.
centroidData	A vector of all data points moved to their respective centroid.

hammingDecode()

Decodes an input Hamming encoded bit vector. Also deletes any additional zeros added in `hammingEncode()`. Each row of the input and output bit vectors represents different streams.

Table 30: In and out parameters for the `hammingDecode()` function.

In parameters	**Description**
codedBitVec	A Hamming encoded bit vector. Each row represents different streams.
hammingN	The n value of the used Hamming code.
hammingK	The k value of the used Hamming code.
zerosAdded	The number of zeros added to the original unencoded to make its dimensions match with the used Hamming code.
Out parameters	**Description**
decodedBitVec	The decoded bit vector. Each row represents different streams.

calcBitError()

Calculates the number of bit errors as well as the ratio of bit errors between the two input bit vectors. Each row of the vectors represents a stream which is compared to corresponding stream of the other vector, e.g. The first row of the first vector is compared to the first row of the second vector, and so on.

Table 31: In and out parameters for the `calcBitError()` function.

In parameters	**Description**
bitVec1	A vector of bits.
bitVec2	A vector of bits
Out parameters	**Description**
nrOfErrors	Vector consisting of the number of errors for each respective stream.
errorRatio	Vector consisting of the error ratio for each respective stream.

plotResultsScript

A plotting script. It is very dependant on variables set in `controllerMainScript`, but does provide a neat way of replotting all results (if one forget to add a plot at the start of the program).
There are five different plots available to the user (examples can be found in [4]). Below is a list of the appropriate booleans in `controllerMainScript` and a short description of what is plotted should that variable be set to `true`:

Table 32: Boolean name and plot description for `plotResultsScript`

Boolean	**Plot Description**
PLOT_BEFORE	Plots the sent passband signals both in time and frequency
PLOT_FREQ	Plots the received signal: the time domain of the passband signal and then the frequency contents during different phases of the receiver process
PLOT_IQ	Plots the unfiltered In and Quadrature phase parts of the received signal together with the filtered versions. The filtered version also have the sampling points marked out on the curve. The sent bits are also plotted in a binary fashion
PLOT_SCATTER	Plots the scatter diagram of the received signal
PLOT_EYE	Plots the eye diagram of the received signal

createCarrier()

Outputs a carrier vector for given values of sample frequency, carrier frequency, number of samples, number of streams and phase offset.

Table 33: In and out parameters for the `createCarrier()` function.

In parameters	**Description**
sampleFreq	Sample frequency
carrierFreq	Carrier frequency
nrOfSamples	The number of samples in the output carrier vector
nrOfStreams	The number of streams in the output vector, i.e. number of rows. Note that all streams look the same.
phaseOffset	Phase offset for the initial phase of the output carrier vector.
Out parameters	**Description**
carrierVec	The output carrier vector.
timeVec	A time vector representing the times of each sample in the carrier vector.

repeatSample()

Upsamples each row in a matrix `upSampleFactor` times. Increases the number of columns `upSampleFactor` times.

Table 34: In and out parameters for the `repeatSample()` function.

In parameters	**Description**
symbolMatrix	The matrix to upsample.
upSampleFactor	Integer. The factor by which each symbol is repeated.
Out parameters	**Description**
repeatSampleMatrix	The upsampled matrix.

3.6 Calibration

Unfortunately, the LM-units, if not calibrated in the software, differ a lot in input and output values when receiving the same audio signal and trying to send the same signal. Therefore, the system has been calibrated in the software by using two scripts in the Calibration module. If the system changes, and someone wants to mixture with the volume knobs of the LM-units, there will be a need for new calibration to make the system work optimally. How to do the calibration is described in much detail in the script files. A sound intensity meeter have been used to calibrate the delivered system, and would also be useful for any future calibrations. The delivered system is calibrated for a carrier frequency of 502 Hz.

3.6.1 Implementation

The calibration is used to set the scaling factors in the `sendReceive` function. It uses the two scripts described below.

calibrationReceiveScript

This scripts calibrates the constant variable `REC_SF` in *sendReceive*. The script is used to make sure that each LM-unit, when getting an equally strong audio signal, samples an approximately equal level sampled signal.

calibrationSenderScript

This scripts calibrates the constant variable `SEND_SF` in *sendReceive*. The script is used to make sure that each LM-unit outputs an approximately equal level audio signal for a given sampled input signal.

3.7 Operating System and Drivers

In order to control the L/M-units the A/D and D/A card are used. These uses PCI-connections and are controlled using their respective Windows drivers. The drivers are accessed from MatLab (32-bit version) using the legacy interface of the "Data Acquisition Toolbox" (DAQ) as well as a MatLab library called "MatLab-compliant Data Acquisition Library" (MLDAQ) [7]. Preferably a 32-bit version of Windows should also be used, but during this project a 64-bit Windows 7 was used, with the cost of some random blue screens.

Figure 7: The computer back panel with A/D and D/A slots. [8, Fig. 2]

4 Hardware

The hardware from the previous project has not been altered and the description given here is therefore heavily based on the technical report produced by the previous project [7]. The hardware is only described to make this document complete but no development has been done regarding the hardware.

The existing hardware consists of a computer, an A/D converter, a D/A converter, a distribution box and eight L/M-pairs. All these entities are described in this chapter. An overview of the hardware is given in Figure 1.

4.1 Computer

The computer is an electronic device used for manipulating information or data, which can be stored, retrieved and processed. The model of the computer used in the project is a Hp Compaq Elite 8300 running Windows 7 as its operating system.

4.2 L/M pairs

The L/M units are modified loudspeakers that can also act as microphones. An L/M unit can operate either as a loudspeaker or a microphone based on the input from the user. A L/M pair is a set of two L/M units in which one is a master unit and the other a slave unit. The system has eight L/M pairs and one of these pairs is depicted in Figure 8. The master unit is made of an original amplifier board and an additional detection board designed by Mikael Olofsson.

The detection makes it possible for the L/M pair to operate as both loudspeaker and a microphone. The 9-pole D-sub connector (DB-9) mounted on top of the master unit serves as interface for signal to, or from, the L/M pair. The power supply to the L/M pair is done through a USB cable, connected to the wall via a USB adapter.

4.3 A/D and D/A converters

The A/D converter is an electronic circuit used to convert an electrical signal into binary numbers to be used in a digital controller (computer). The D/A converter circuit is used to convert binary numbers to analog voltage or current. The A/D converter (Contec AD12-64) and the D/A converter (Contec DA12-16) are attached to the computer

motherboard through PCI slots to ensure the communication between the computer and the distribution box. With the help of the internal sample clock in each card, resolution of 12V and then highest conversion speed of 100 kilosamples/s are attained for both converters. The voltage levels for both converters are in the interval [-10, 10] V.

Only 16 of the, in total, 64 analog input channels are used in the product. The A/D converter also has 4 digital inputs and 4 digital outputs for TTL level signals (Transistor-Transistor-Logic). In the product only two of the digital outputs of the A/D converter are utilized and these are responsible for switching between the two operation modes of the L/M pairs.

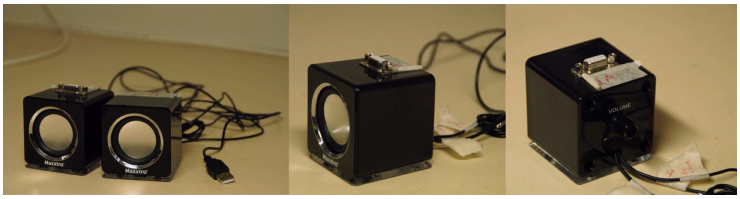

Figure 8: The L/M pair and the master unit from different angles. [8, Fig. 6]

4.3.1 Detection Board

Microphone mode operation for the L/M pair is possible by means of the detection board. When the L/M pair is working in this mode the circuit amplifies the audio signals received using a differential-in-differential-out amplifier with a voltage gain of 23 dB. Thereafter, the amplified signals are forwarded to the collection board for further amplification.

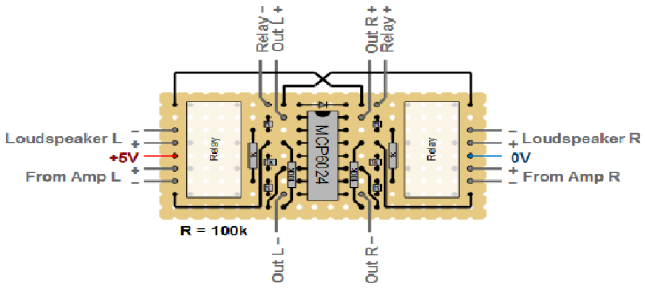

Figure 9: Detection board. [7, Fig. 3]

4.3.2 Maxxtro Mini Speaker 4W

A master loudspeaker and a slave loudspeaker constitute the main parts of the L/M pair. Both speakers, joint by a stereo cable of length 0.3m, are supplied 5V by the USB cable

of 1m length connected to an adapter. The adapter, called *Euro-USB-laddare*, has the stock number 25-249-98 at ELFA. The USB cable and volume controller are attached only to the master loudspeaker.

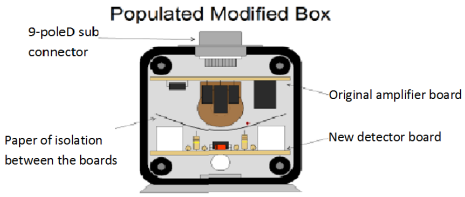

Figure 10: Overview of the L/M master unit. [7, Fig. 7]

4.4 Distribution Box

The distribution box depicted in Figure 11 works as a hub and distributes the signals between the computer and the L/M pairs. It has 11 connections: eight for the L/M pairs, one for the A/D and D/A converters respectively and a power supply connection. For the purpose of letting the user choose which control group a L/M pair is part of the distribution box also has a control board consisting of eight physical switches.

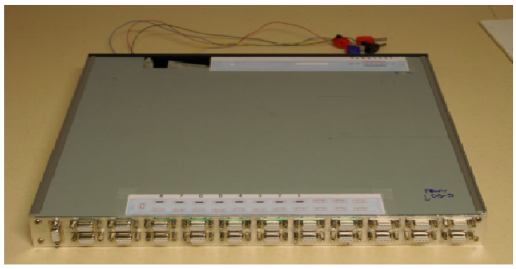

Figure 11: The Distribution Box. [8, Fig. 3]

4.5 Limitations on available hardware

4.5.1 Sampling frequency

The maximum sampling frequency that can be used for the A/D and D/A converter is 100 kHz. These 100 kHz are divided among all channels in use. Thus, when all 16 L/M

units are in use the maximum sampling frequency is limited to 6250 Hz in theory. But the sampling frequency is fixed to certain values, so practically the sampling frequency 6060Hz is used to get a good carrier frequency for an easy implementation.

4.5.2 Data transmission

The amount of data that can be put through the D/A engine is approximately 262144 samples,which translates to about 2.5 seconds of continuous sound, when using a sampling frequency of 6060 Hz and 16 channels.

4.5.3 A/D converters

The A/D converter has a voltage range of [-10,+10] V. However the power supply can deliver voltages in the interval [-11,11] V which means that the A/D converter can be damaged by a too high sound level from the loudspeakers.

5 Limitations and Problems

The system has some limitations and problems that have been discovered and taken into consideration in the development of the product. These are in addition to the inherent hardware limitations already mentioned in section 4.5.

5.1 First Run

The values obtained from the first run using the system after switching on the computer are inconsistent. We are not sure exactly why this happens but we suspect it may have to do with initializations for the hardware that occur during the first run. Not a major problem as long as you ignore the first run.

5.2 Blue Screen

While running the system, one might encounter a system crash and blue-screen which we think is likely due to the Contect drivers. The Contect drivers, used for running the A/D and D/A hardware, are specified for use with a 32-bit system while the system we use is a 64-bit one.

5.3 Sampling Clock Error

One might also occasionally encounter a sampling clock error. We are not sure why it occurs but we think it may also have something to do with the Contect drivers not being specified for a 64-bit system.

5.4 Nonuniform L/M Unit Characteristics

The L/M units have different characteristics for both sending and receiving. The receiving characteristics differ greatly for different carrier frequencies as well. Hence calibration has to be done for proper functioning of the system. We have calibrated our system for the carrier frequency 505 Hz and have compensated for these differences in the software using hard-coded values in the *sendReceive()* function. So in order to use the system at other carrier frequencies the calibration would have to be done again for those frequencies.

5.5 Phase interval

Since the channel state matrix is represented with complex numbers it is only possible to represent phases in the interval $[0, 2\pi)$. Due to this fact the distance from *one* terminal to the different units should not differ by more than one wavelength for correct synchronisation to be possible when sending. We use this assumption in the phase estimation to correct outliers that are off from the expected values by multiples of 2π.

5.6 Distance constraints

The L/M units sensitivity is quite weak at long distances. Hence we can not operate the system reliably with units placed long distances apart. We recommend using a distance less than 3m.

5.7 Background Noise

Our system performance is very sensitive to noise especially in the audible range. Hence for reliable functioning of the system it can only be used in a quite environment with minimum audible background noise.

References

[1] Tomas Svensson, Christian Krysander, *Projektmodellen LIPS*. Studentlitteratur, 2011.

[2] C. H. Knapp and G. C. Carter, *The generalized correlation method for estimation of time delay,* IEEE Trans. Acoust., Speech, Signal Processing, vol. 24, no. 4, pp. 320326, Aug. 1976.

[3] T. Gustafsson, B. Rao, M. Triverdi, *"Source localization in reverberant environments: modeling and statistical analysis."* IEEE Trans. Acoust., Speech, Signal Processing, vol. 11, no. 6, pp. 791-803. 2003.

Unpublished References:

[4] Mikael Karlsson et al., *User Manual*. ISY, Linkoping University, 2015.

[5] Mikael Karlsson et al., *Project Plan*. ISY, Linkoping University, 2015.

[6] Mikael Karlsson et al., *Requirement Specification*. ISY, Linkoping University, 2015.

[7] Fredrik Stenmark et al., *Technical Report*. ISY, Linkoping University, 2014.

[8] Fredrik Stenmark et al., *User Manual*. ISY, Linkoping University, 2014.

[9] Fredrik Stenmark et al., *Project Plan*. ISY, Linkoping University, 2014.