TECHNICAL DOCUMENTATION

Simon Sörman

Version 1.0

Status

Reviewed	Mikael Olofsson	2015-12-11
Approved	Mikael Olofsson	2015-12-11

PROJECT IDENTITY

HT 2015

Linköping University, ISY

Participants of the group

Name	Responsible	Phone	E-mail
Miguel Abadia	Software Engineer	073 723 98 37	migab416@student.liu.se
Herman Molinder	Project Manager	076 823 81 96	hermo276@student.liu.se
Simon Pålstam	Software Engineer	076 803 17 06	simpa265@student.liu.se
Thiti Sookyoi	Software Engineer	073 573 66 63	thiso311@student.liu.se
Simon Sörman	Responsible for the documentation	070 954 78 41	simso657@student.liu.se

Customer: Mikael Olofsson, mikael.olofsson@liu.se, 013 – 28 13 43

Examiner: Danyo Danev, danyo.danev@liu.se, 013 – 28 13 35

Supervisor: Antonios Pitarokoilis, antonios.pitarokoilis@liu.se, 013 – 28 13 40

Contents

Document history

Version	Date	Changes	Performed by	Reviewed
0.1	2015-12-09	First draft	All project members	All project members
0.2	2015-12-11	Added information about channel coders	MA	HM
1.0	2015-12-12	Document approved		

1 INTRODUCTION

This document is the technical documentation of the project Parallel Data Transmission, made as part of the course TSKS05 – Communication Systems CDIO at Linköping University 2015.

The aim is that this documentation should be extensive enough to enable an outside person, with the same previous knowledge as the project group before the start of the project, to continue development of the system. This documentation should be read together with the User Manual to understand the complete usage of the product.

1.1 MIMO beamforming

Massive Multiple-Input Multiple-Output (MIMO) beamforming is the very foundation on which this project is built upon and is utilizing. This is a theory of wireless communication where the base station is equipped with a large amount of antennas (thus massive MIMO). The main result of the theory is that with knowledge of the channels between the base station and the terminals (users), the base station is able to "beamform" signals. The meaning of this is that the signal can be aimed at a specific point in space, due to constructive interference of waves. In classical systems, the space in one base station's cell and all therein positioned terminals is considered as one single big channel that needs to be split in time and/or frequency to enable communication with several users at the same (perceived) time. But by directing different signals to terminals at different positions, one has effectively split the single big channel in space rather than in time and/or frequency. This allows the base station to actually communicate with several terminals at the same carrier frequency and at the exact same time instant. The benefits of this is both potentially larger data rates for the users, and lower energy consumption by the base station since it doesn't have to spread signal energy everywhere, but only towards the users.

1.2 Project

This project is a continuation of the project in the same course that was conducted in 2014, where hardware was produced together with some small amount of software that produced a proof of concept of the massive MIMO technique. The hardware consists of A/D and D/A converters, 16 units of dual loudspeakers and microphones (L/M-units), and some circuitry. Thus the hardware does not use electromagnetic waves as means of communication as in commercial systems, but uses soundwaves instead.

The aim of the project was to create a demonstrational system that is able to use 14 of these L/M-units as the base station MIMO array, and the last 2 as terminals. The system should be able to communicate data to both of these terminals at the same frequency and the same time instant, and aim at maximizing throughput while keeping a reasonable bit error rate.

The final system is able to manage this communication at rates up to 300 bits per second and user, while still maintaining a bit error rate of less than 1 percent.

1.3 Definitions of terms

In table 1 below, abbreviations used throughout the document is listed and explained.

Table 1: Abbreviations used in the document

Word	**Definition**
A/D	Analog to Digital
BER	Bit Error Rate
BPSK	Binary Phase Shift Keying
D/A	Digital to Analog
DF	Decision Feedback
GUI	Graphical User Interface
ISI	Inter-Symbol Interference
L/M	Loudspeaker/Microphone
LSE	Least-Square Estimation
MIMO	Multiple Input Multiple Output
MLE	Maximum Likelihood Estimation
MMSE	Minimum Mean Square Error
MRT	Maximum-Ratio Transmission
QPSK	Quadratic Phase Shift Keying
TDD	Time Division Duplex
ZF	Zero-Forcing
MIMO-array	The array of L/M units that are used for beamforming
MIMO-transceiver	One L/M unit in the MIMO-array

2 SYSTEM OVERVIEW

The complete system consists of both hardware and software implementations. This project is only focused on the software part of the complete system, as the hardware that is used was created during the last year's project. The hardware consists of a computer, 8 L/M pairs, A/D and D/A converters and a distribution box, as described in last year's project documentation (Stenmark, 2014a). The software consists of a large amount of files almost exclusively containing matlab code; these files are listed in appendix A and are essential for the system to work properly. The software has been divided into three sub-systems, the channel estimator, the modulator and the channel coder. A simple overview of the sub-systems is shown in Figure 1.

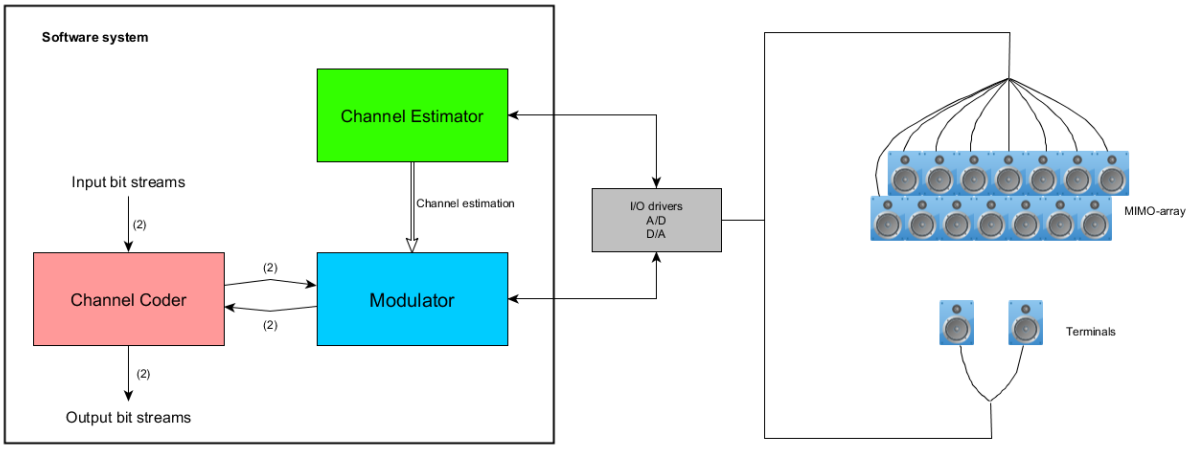

Figure 1: Overview of the sub-systems

The interaction of the various processing blocks for the communication is shown in Figure 2:

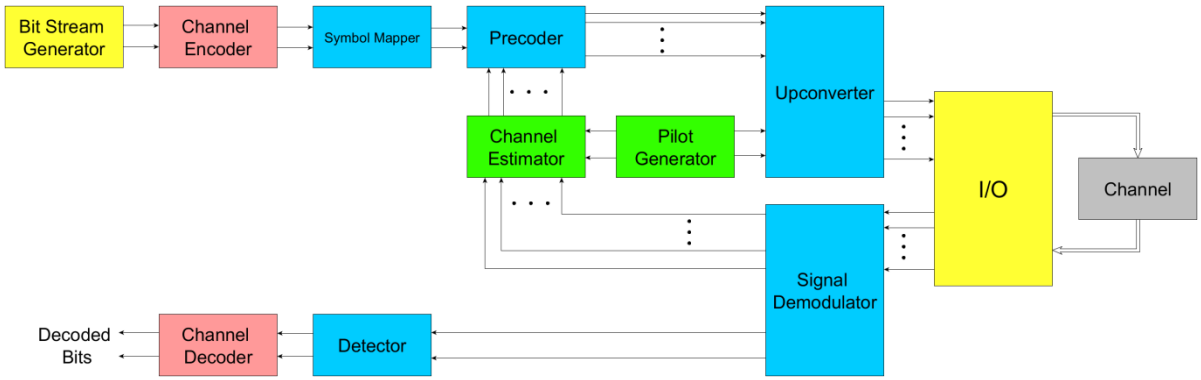

Figure 2: Overview of the software system

In the figures, the blocks are color-coded. Red blocks are part of the channel coder, the blue ones are the modulator and the green blocks are the channel estimator. Yellow blocks are not part of a sub-system.

As shown in Figure 2 the software system functions as follows. The channel encoder is presented with two bit streams that in the end are transmitted from the MIMO array to the two terminals. The coder encodes these bit streams and output two coded bit streams. The symbol mapper then maps these bit-streams onto appropriate baseband-signals. The channel estimator

estimates the channels. In order to do this each terminal sends one pilot signal to the array. The received signals during pilot transmission are passed on to the channel estimator and processed to get channel estimates. The precoder uses these estimates to perform prefiltering of the baseband signals. These signals are modulated by the upconverter to be the real valued signals that are transmitted. The signal demodulator does the reverse operation; it takes the real sampled signals and produces the complex baseband representation. The detector converts the baseband signals to fit the channel decoder. The channel decoder will use this information to estimate the most likely information bit sequences.

The bit stream generator's only task is to provide streams of bits that the system should use. This can be random bits or representations of text messages that the user can specify. The I/O-block is responsible for controlling the hardware, making sure that the upconverter/demodulator has an easy to use interface to the L/M-pairs.

2.1 Main program

The main program of the system is a script called *massive_MIMO_transmission.m*. This script cannot be used without setting a set of parameters beforehand. These parameters are set either by using the graphical interface for the system (see section 2.2) or by running the script *NO_GUI_transmission.m*. The main script can be divided into several different phases. The first phase sets a few constant parameters that have been decided either by testing or simply because of the limits of the system. Next phase is the initialization; this part initializes the hardware and creates the bit sequences that will be transmitted. The bit sequences are either randomized or created from an input text string. After initialization comes the coding, this phase performs channel coding, interleaving, and symbol mapping of the bit sequences.

The program is now ready to start transmitting the data. The transmission is divided into frames of maximum two seconds, thus will the transmission be divided into several frames if it takes more than two seconds to transmit. In each frame the following tasks are performed: send pilot signals from the terminals to the MIMO array, estimate the channel impulse responses, precode the information signals, send information signals from the MIMO array to the terminals, demodulate the signals received at the terminals.

The decoding phase starts when all frames have been transmitted and received; this part performs inverse interleaving and channel decoding, and eventually decodes a text message from the received bit sequence. The last part is the statistics phase where various statistics from the transmission are calculated.

2.2 GUI

The GUI is created with Matlab built in functionality for graphical interfaces. When using this application Matlab produces a fig-file (*GUI.fig)* that specifies the design of the GUI, and an m-file (*GUI.m*) consisting of auto generated code. The m-file has later been modified to perform security checks and set all parameters for the *massive_MIMO_transmission.m* script, as well as run the script or plot results when the corresponding icon is clicked. The basic structure of the m-file is that all objects in the GUI have a callback-function that is called each time the object is changed. The communication between the GUI objects and the callback-functions is done via handle objects stored in a structure called *handles*. A picture of the GUI is presented in Figure 3.

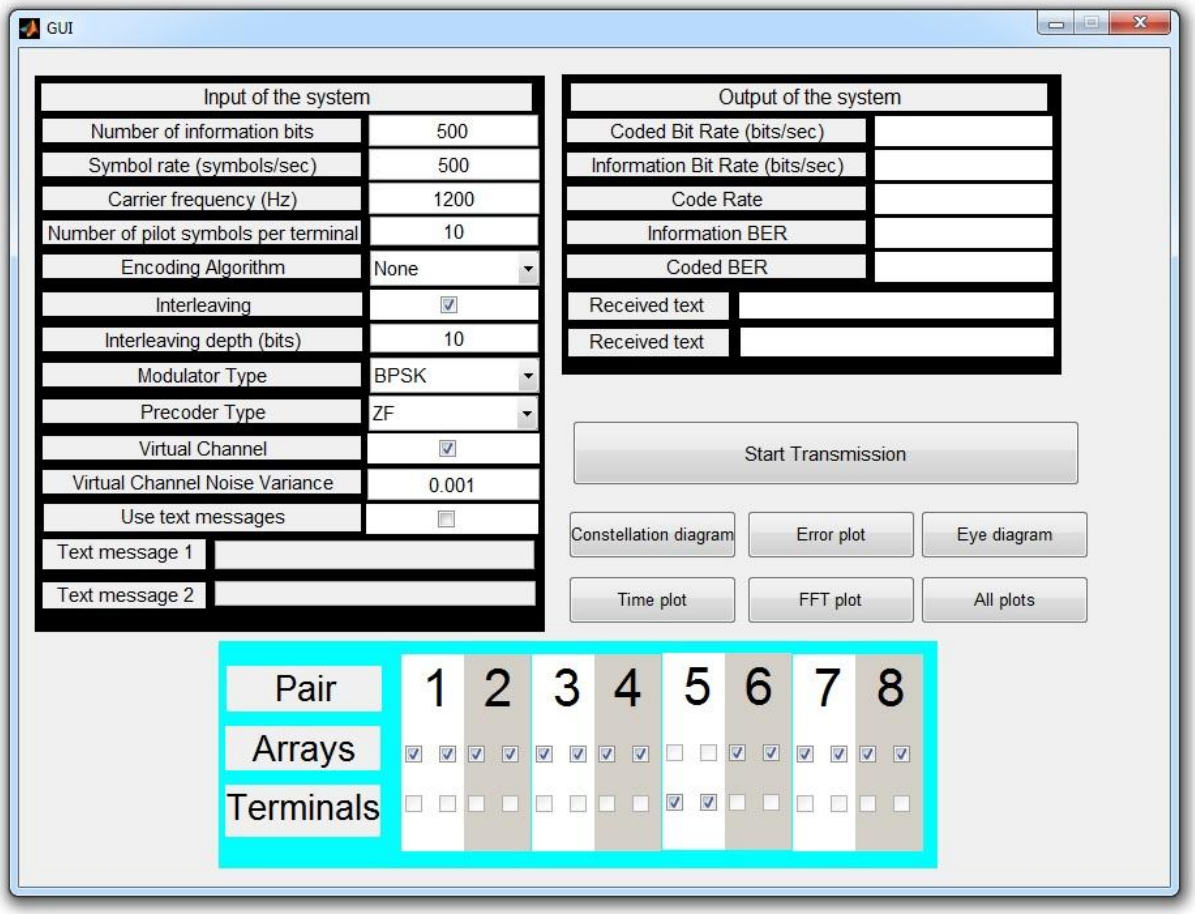

Figure 3: Graphical User Interface

The upper left box of the GUI displays the input parameters for the system, descriptions and limitations for all these input parameters are displayed in Table 2.

Table 2: Descriptions and limits of input parameters

Input	**Description**	**Min value**	**Max value**
Number of information bits	Input the number of information bits (integer).	1	-
Symbol rate	Input the symbol rate in unit symbols/sec (integer).	12	$3012 = \left(\frac{fs}{2}\right)$
Carrier frequency	Input the carrier frequency in unit Hz (integer).	1	$3011 = \left(\frac{fs}{2} - 1\right)$
Number of pilot symbols per terminal	Input the number of pilot symbols per terminal (integer).	1	$2 * (Symbol\ rate)$
Encoding Algorithm	Choose the error correcting coding algorithm from a list.	-	-
Interleaving	Choose to use interleaving.	-	-
Interleaving depth (Bits)	Input interleaving depth if interleaving is used.	-	-
Modulator Type	Choose the modulator type from a list.	-	-
Precoder Type	Choose precoder type from a list.	-	-
Virtual Channel	Choose to use a virtual channel	-	-
Virtual Channel Noise Variance	Input the variance of the AWGN that is applied to all simulated channels if virtual channel is used.	0	-
Use text messages	Choose to transmit text messages.	-	-
Text message 1	Input a US-ASCII text message for the first terminal. The text message must contain at least one printable US-ASCII character.	-	-
Text message 2	Input a US-ASCII text message for the second terminal. The text message must contain at least one printable US-ASCII character.	-	-

The lower box defines which L/Ms are used as terminals and which L/Ms are used in the MIMO array. It is possible to start a transmission by clicking the icon *start transmission*. The

upper right box displays statistical data from the latest transmission. There are five different types of plots that can be produced by clicking on the corresponding button in the GUI. The plots that are available are:

- *Constellation diagram* – Plots the received symbols and the constellation points in the complex plane.
- *Error plot* – Creates a discrete time plot that displays all bit errors received by each terminal.
- *Eye diagram* – Crates an eye diagram for the received symbols at each terminal.
- *Time plot* – Plots the received signal in the time-domain. It should be noted that the time axis not corresponds to the real time since the time for pilot transmission and coding/decoding is not included.
- *FFT plot* – Plots the received signal for each terminal in the frequency-domain.

For more information about how to use the system please read the user manual.

3 MODULATOR

The Modulator sub-system has two different main tasks. The first is to create signals to be transmitted by the MIMO-transceivers. The second is to demodulate signals received by the two terminals. The MIMO signals are created by mapping the bit streams to a chosen symbol constellation and precode the symbols into several different signals that are to be transmitted by the MIMO-transceivers.

3.1 External interface

The input to the modulator sub-system consists of bit streams from the channel encoder and channel impulse responses from the channel estimator. The MIMO signals are sent to the D/A card via the I/O drivers. Sub-system also receives signals from the I/O. These signals are either sent immediately to the channel estimator as a baseband signal or to the channel decoder as detected bit streams. A block scheme of the modulator sub-system is presented in Figure 4.

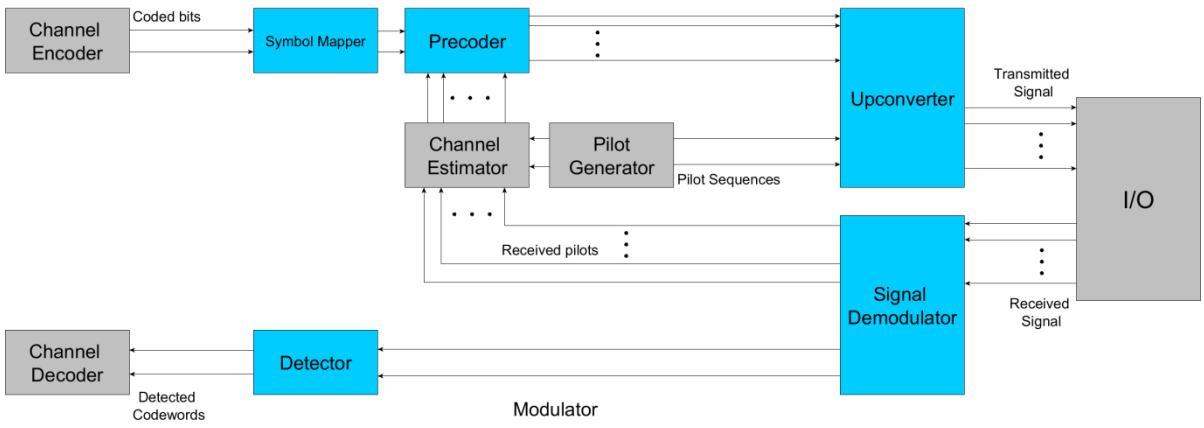

Figure 4: A simple block scheme of the modulator sub-system

3.2 Symbol Mapper

The symbol mapper sub system maps the bit streams onto a chosen symbol constellation to create a baseband. The constellations available are: BPSK, QQPSK and 8PSK. To accomplish this there is one mapping function for each constellation. There is also an overhead function that calls the other functions depending on the given input.

3.3 Detector

The detector sub system detects bits from a received baseband signal. The detection is performed with hard decision and the constellations available are: BPSK, QPSK and 8PSK. Just as for the symbol mapper there is one function for each constellation as well as an overhead function.

3.4 Precoder

The precoder filters the baseband signals to create one precoded signal for each transceiver in the MIMO array. The filtering consists of different scaling and phase shifting of the signals. In practice this comes down to a matrix multiplication as in equation 1.

$$\mathrm{x} = \sqrt{\alpha}\mathrm{Ws} \tag{1}$$

Here W is the filter matrix, x is a matrix where each row contains the precoded baseband signal to be sent from one MIMO transceiver, s is a matrix where each row contains a baseband signal intended for a terminal, this is visualized in equation 2. Finally, α is a normalization constant defined by equation 3.

$$s = \begin{pmatrix} s_{1,1} & \cdots & s_{1,N_s} \\ \vdots & \ddots & \vdots \\ s_{k,1} & \cdots & s_{k,N_s} \end{pmatrix} \tag{2}$$

Here $s_{i,j}$ is the j:th symbol to the i:th terminal.

$$\alpha = \frac{1}{\mathrm{E}\{\mathrm{tr}(\mathrm{WW}^{\mathrm{H}})\}} \tag{3}$$

Note that this is a theoretical expression, as it requires the knowledge of the channel statistics. Since the statistics is not known the expectation is estimated to be that of equation 4.

$$\widehat{\alpha} = \frac{1}{\mathrm{tr}(\mathrm{WW}^{\mathrm{H}})} \tag{4}$$

Two different kinds of filters can be used as W, these are: MRT and ZF.

The MRT algorithm only maximizes the signal gain at the intended user and is close to optimal in noise-limited systems, where the inter-user interference is negligible compared to

the noise. The MRT filter is defined by equation 5 where H is the channel matrix defined in section 4.2.

$$W_{MRT} = H^{*} \quad (5)$$

It is worth mentioning that this method does not perform well if there is high co-channel interference.

The ZF filter only compensates for co-channel interference and neglects thermal noise. The filter is defined by equation 6.

$$W_{ZF} = H^{*}(H^{T}H^{*})^{-1} \quad (6)$$

These linear precoders have very low complexity but give a sufficient result for this system.

3.5 Upconverter

The upconverter is responsible for transforming the complex baseband signals into passband signals with a chosen carrier frequency. These signals are then forwarded to the I/O for D/A conversion.

3.6 Signal Demodulator

The demodulation starts by transforming the received signal from passband to baseband, this transformation creates a copy of the signal at $2f_c$, this copy along with high frequency noise is removed from the baseband signal by low pass filtering it. The filter used is an 8th order digital Butterworth low-pass filter with cut-off frequency f_c.

Next step is to find the time delay in the received signal; this is done by correlate the received signals with the known part of the transmitted signals. In the uplink case this refers to the non-zero part of the pilot signals and in the downlink case this refers to predefined synchronisation symbols added at the beginning of the information signals. Equation 7 gives a mathematical expression for the estimation of the time-delay for a single signal.

$$arg\max_n(r_i \star p_i)[n] \quad (7)$$

Here r_i is the received signal at receiver i; p_i is the known pilots (or synchronisation-symbols in the downlink), and $\star$ denotes convolution.

The phase-shift of the received signals compared to the transmitted signals are also estimated to complete the synchronisation, this is done as in equation 8.

$$\hat{\psi} = angle(r_i \cdot p_i) \quad (8)$$

Here r_i and p_i are the same as in equation 7 and $\cdot$ denotes the scalar product. The received signal is then phase-shifted with $-\hat{\psi}$ to get the same phase as in the transmitted signal.

The final part of the signal demodulation is to down sample the signal to symbols, taking the mean value of the samples in each symbol interval does this. A guard interval of 1/2 the symbol length is implemented to make the system more robust against small errors in the synchronisation and to minimize the errors produced by symbol alternation. The sampling is done in the centre of the symbol interval, thus are all symbol intervals split into 1/4 pre-guard, 1/2 symbol sampling, and 1/4 post-guard.

3.7 Functions

This section describes all functions that belong to the modulator sub-system.

3.7.1 symbol_mapper_bpsk()

This function assigns a complex symbol for each bit in the input vector according to a BPSK constellation.

Input:

bits – Matrix where each row contains the coded bits to be sent to each terminal.

Output:

symbols – Matrix where each row contains the complex symbols to be sent to each terminal.

3.7.2 symbol_mapper_qpsk()

This function assigns a complex symbol for each pair of bits in the input vector according to a QPSK constellation. The mapping is Gray coded to minimize the number of bit errors for each symbol error. This function adds a zero at the end of the input vector if the length of the vector is odd. This is to make sure that the last symbol is complete.

Input:

bits – Matrix where each row contains the coded bits to be sent to each terminal.

Output:

symbols – Matrix where each row contains the complex symbols to be sent to each terminal.

3.7.3 symbol_mapper_8psk()

This function assigns a complex symbol for each triplet of bits in the input vector according to an 8PSK constellation. The mapping is Gray coded to minimize the number of bit errors for each symbol error. This function adds zeros at the end of the input vector if the length of the vector is not divisible by 3. This is to make sure that the last symbol is complete.

Input:

bits – Matrix where each row contains the bits to be sent to each terminal.

Output:

symbols – Matrix where each row contains the complex symbols to be sent to each terminal.

3.7.4 mapper()

This is an overhead function that calls one of the symbol_*mapper_*()* functions depending on the input. It also calculates the number of added bits.

Input:

bits – Matrix containing bits for each terminal.

constellation – String that specifies the constellation that will be used. The inputs available are: 'BPSK', 'QPSK' and '8PSK'.

Output:

complexSymbols – Matrix where each row consists of symbols for each terminal.

extraMappingBits – The number of added bits at the end.

3.7.5 detector_bpsk()

This function detects the most probable bit for each complex symbol according to a BPSK constellation.

Input:

symbols – Matrix where the rows consists of complex symbols for each terminal

Output:

bits – Matrix where the rows consists of bits for each terminal.

3.7.6 detector_qpsk()

This function detects the most probable bit pair for each complex symbol according to a QPSK constellation. The detection assumes that Gray coding were used for the mapping.

Input:

symbols – Matrix where the rows consists of complex symbols for each terminal

Output:

bits – Matrix where the rows consists of bits for each terminal.

3.7.7 detector_8psk()

This function detects the most probable bit triplet for each complex symbol according to an 8PSK constellation. The detection assumes that Gray coding were used for the mapping.

Input:

symbols – Matrix where the rows consists of complex symbols for each terminal

Output:

bits – Matrix where the rows consists of bits for each terminal.

3.7.8 distances()

This function calculates the euclidean distance between the received symbols in one terminal and every complex symbol of the constellation. This function is only used by the *detector_8psk()* function.

Input:

s – Vector containing received complex symbols for one terminal.

constellation – Compared constellation

Output:

dist – Distance between the corresponding vector and the concrete constellation.

3.7.9 detector()

This is an overhead function that calls one of the *detector_*()* functions depending on the input.

Input*:*

complexSymbols – Matrix where the rows consists of complex symbols for each terminal.

constellation – String that specifies the constellation that will be used. The choices available are: 'BPSK', 'QPSK' and '8PSK'.

extraMappingBits – The number of bits added by the mapper at the end to get a complete symbol.

Output:

bits – Matrix where each row consists of bits for each terminal.

3.7.10 MRT()

This function precodes complex symbols with a MRT filter to enable a massive MIMO transmission. It is possible to reuse and update the normalization constant if the function is called several times. This might give a better estimate of the expression in equation 3, section 3.4.

Input:

complexSymbols – Matrix where the rows consists of complex symbols for each terminal

channelEstimations – Matrix containing all the channel estimations calculated by the channel estimator.

normConst – Normalisation constant that will be updated each time the function is used, it is denoted alpha in the mathematical description.

iterNmbr – The number of times that normConst has been updated, thus the number of times that the function has been used without a reset.

Output:

precodedSymbols – Matrix containing the precoded symbols for each transmitter in the MIMO array.

normConst – The updated normConst.

iterNmbr – The updated iterNmbr.

3.7.11 ZF()

This function precodes complex symbols with a ZF filter to enable a massive MIMO transmission. It is possible to reuse and update the normalization constant if the function is called several times. This might give a better estimate of the expression in equation 3, section 3.4.

Input:

complexSymbols – Matrix where the rows consists of complex symbols for each terminal

channelEstimations – Matrix containing all the channel estimations calculated by the channel estimator.

normConst – Normalisation constant that will be updated each time the function is used, it is denoted alpha in the mathematical description.

iterNmbr – The number of times that normConst has been updated, thus the number of times that the function has been used without a reset.

Output:

precodedSymbols – Matrix containing the precoded symbols for each transmitter in the MIMO array.

normConst – The updated normConst.

iterNmbr – The updated iterNmbr.

3.7.12 upconverter()

This function takes complex-baseband signals and converts them into real-valued passband signals specified by the input.

Input:

fs – The sampling frequency that should be used (Hz).

fc – The carrier frequency to convert into (Hz).

R – The symbol rate that should be used (symbols/sec).

s*ymbols* – A matrix where each row contains a complex-baseband signal to convert.

Output:

signals – A matrix where each row contains a converted passband signal.

3.7.13 uplink_demodulator()

This function performs several operations on the received uplink/pilot signals to retrieve the received symbols. The operations are: Transformation from passband to baseband, low pass filtering, synchronisation, phase estimation/modification, and finally down sampling.

Input:

fs – Sampling frequency [integer].

fc – Carrier frequency [integer].

R – Symbol rate [integer].

r – Matrix where each row contains the samples of the received signal for each transceiver.

p – Matrix where each row contains the pilot symbols for each terminal.

Nguard – Number of guard symbols before first pilot symbol.

Output:

symbols – Matrix where each row contains the received symbols for each transceiver.

3.7.14 downlink_demodulator()

This function performs several operations on the received downlink signals to retrieve the received symbols. The operations are: Transformation from passband to baseband, low pass filtering, synchronisation, phase estimation/modification, and finally down sampling.

Input:

fs – Sampling frequency [integer]

*f*c – Carrier frequency [integer]

R – Symbol rate [integer]

r – Matrix where each row contains the samples of the received signal for each terminal.

p – Matrix where each row contains an orthogonal synchronisation vector for each terminal.

Nguard - Number of guard symbols before first pilot symbol

Nsym - The number of symbols that are supposed to be demodulated, this number includes the synchronisation symbols in *p.*

Output:

symbols – Matrix where each row contains the demodulated symbols for each transceiver.

4 CHANNEL ESTIMATOR

The channel estimator estimates the channel impulse response between each MIMO transceiver and terminal. A TDD protocol is used where the terminals initially transmit pilots and subsequently the MIMO array transmits data to the terminals. At the beginning of a time frame the terminals transmit orthogonal pilots, which are received by the MIMO transceivers. By processing the received pilots the channel estimator is able to calculate and output channel estimates to the modulator. The estimation procedure is done regularly to adapt to changes in the channel.

4.1 External Interface

The Pilot Generator outputs pilot sequences to the upconverter. The Channel Estimator receives channel responses from the channels between each terminal and MIMO transceiver. The number of pilot sequences is equal to the number of terminals. The number of estimates is equal to the number of terminals multiplied by the number of MIMO transceivers. An example overview of the external interface of the channel estimator is shown in Figure 5.

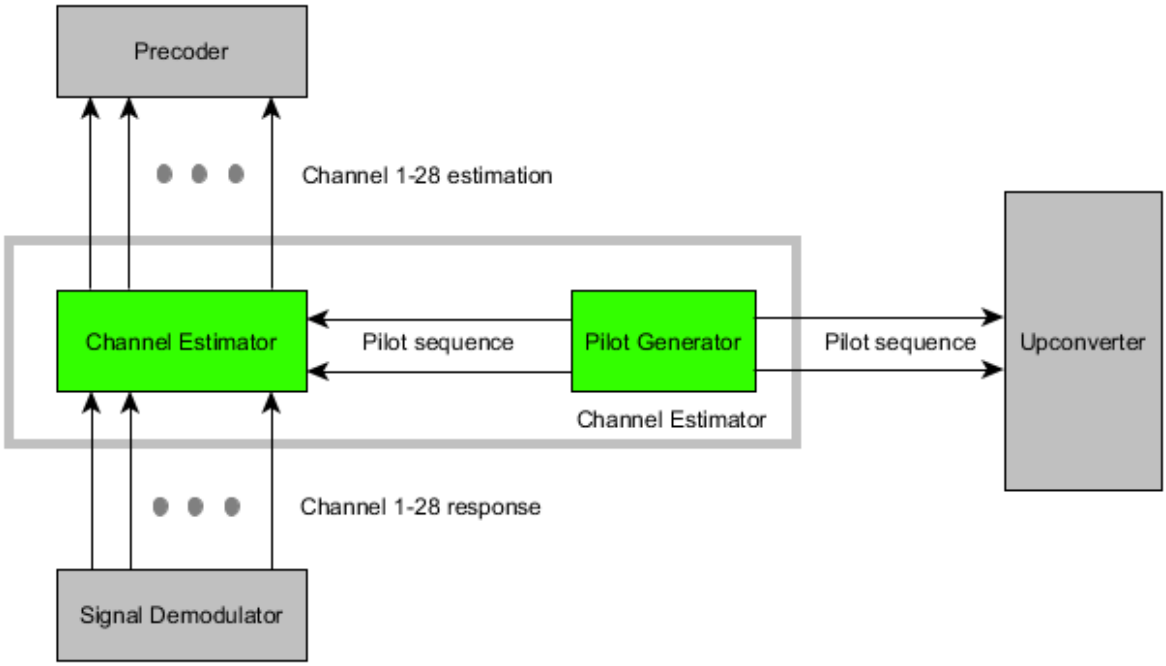

Figure 5: External interface of the channel estimator using 2 terminals and 14 MIMO transceivers

4.2 Mathematical Description

Let:

$\boldsymbol{p}_i = \begin{pmatrix} p_i^{(1)} & \cdots & p_i^{(N_p K)} \end{pmatrix}$ be the pilot sequence for the i:th terminal, where N_p is the number of pilot symbols and K is the number of terminals.

$\boldsymbol{P} = \begin{pmatrix} \boldsymbol{p}_1 \\ \vdots \\ \boldsymbol{p}_K \end{pmatrix}$ be the pilot matrix, where K is the number of terminals.

$\boldsymbol{H} = \begin{pmatrix} h_{1,1} & \cdots & h_{1,K} \\ \vdots & \ddots & \vdots \\ h_{M,1} & \cdots & h_{M,K} \end{pmatrix}$ be the channel matrix, where $h_{i,j}$ is the channel impulse response between terminal i and transceiver j, M is the number of array elements and K is the number of terminals.

$\boldsymbol{N}$ be a complex white Gaussian noise matrix.

Then the received matrix $\boldsymbol{Y}$ can be modeled as follows:

$$\boldsymbol{Y} = \boldsymbol{HP} + \boldsymbol{N}$$

4.3 Pilot Generator

Let $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ be the pilot sequence for terminal i and j, respectively. Then $\boldsymbol{p}_i \cdot \boldsymbol{p}_j = 0$ must be fulfilled for any $i \neq j$ to ensure that the pilot sequences are orthogonal. This is achieved by letting:

$$\boldsymbol{p}_1 = \begin{pmatrix} p^{(1)} \\ \vdots \\ p^{(N_p)} \\ 0 \\ \vdots \\ 0 \end{pmatrix}^T, \boldsymbol{p}_i = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ p^{((i-1)N_p+1)} \\ \vdots \\ p^{(iN_p)} \\ 0 \\ \vdots \\ 0 \end{pmatrix}^T \text{ and } \boldsymbol{p}_K = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ p^{((K-1)N_p+1)} \\ \vdots \\ p^{(KN_p)} \end{pmatrix}^T$$

The non-zero elements in the pilot sequences is created by letting $p^{(k)} = \exp(iU)$, where U is a continuous uniform distributed stochastic variable with boundaries $[0, 2\pi]$ and i is the imaginary unit. Therefore $p^{(k)}$ becomes a random point on the complex plane unit circle.

The randomness is used to distinguish phase and delay differences between pilot sequences by the demodulator. The stochastic distribution is created with the built in pseudo random functions in Matlab.

4.4 Channel Estimator

The channel estimates will be estimates of the channel impulse response within the coherence time interval and coherence bandwidth. If the channel can be assumed to be flat fading, the channel impulse response can be represented as a complex number. The LSE algorithm provides an MLE of $\boldsymbol{H}$ according to equation 9.

$$\boldsymbol{H}_{MLE} = \boldsymbol{Y}\boldsymbol{P}^H(\boldsymbol{P}\boldsymbol{P}^H)^{-1} \quad (9)$$

4.5 Functions

This section describes all functions that belong to the channel estimator sub-system.

4.5.1 pilot_generator()

This function returns a matrix of orthogonal pilot sequences. Each row of the matrix contains a pilot sequence.

Input:

Nterm – Number of pilot sequences/terminals

Nsym_per_term – Number of pilot symbols per terminal

Output:

p – A matrix with *Nterm* rows and *Nsym_per_term* columns, where each row contains the pilot sequence for a terminal.

4.5.2 channel_estimator()

This function returns a matrix of the estimated channel impulse responses. The channel impulse responses are estimated with the least squares estimation algorithm. *H_lse(i,j)* corresponds to the estimated impulse response between MIMO transceiver *i* and terminal *j*.

Input:

P – A matrix with *"*number of terminals*"* rows and "number of pilot symbols" columns, where each row contains the pilot sequence for a terminal.

Y – A matrix with *"*number of MIMO tranceivers*"* rows and "number of pilot symbols" columns, containing the received pilot sequences of each MIMO transceiver. *Y(i,j)* corresponds to received symbol *j* of transceiver *i*.

Output:

H_lse – A matrix with *"*number of MIMO transceivers*"* rows and "number of terminals" columns, containing the estimated channel impulse responses. *H_lse(i,j)* corresponds to the estimated impulse response between MIMO transceiver *i* and terminal *j*.

5 CHANNEL CODER

To be able to use the system for demonstration of real data transmission, channel coding is used to try to minimize errors in the information bits being transmitted.

The channel coder is designed to improve the system, meaning that compared to the system without channel coding it preserves the information bit rate while decreasing the information BER. An interleaver has also been implemented to enhance the performance of the channel coding. The channel coder sub-system takes several information bit streams as input, and outputs the same number of coded bit

5.1 External Interface

Figure 6 below shows a simple overview of how this sub-system functions in the complete system. The channel coder is able to do both encoding and decoding. The interface of this is two functions, one for encoding a number of sequences, and one for decoding a number of sequences. The outputs of the functions are the decoded/coded sequences.

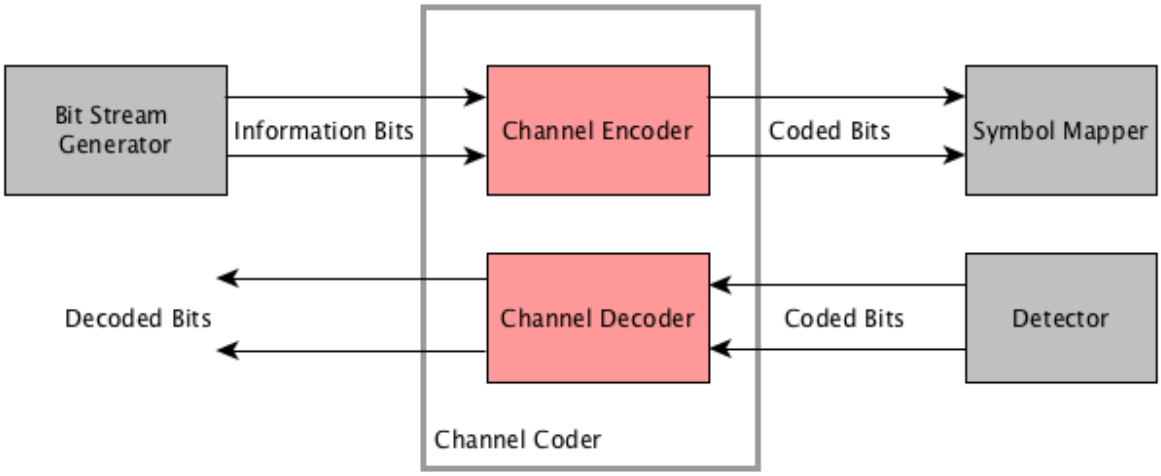

Figure 6: A block scheme of the channel encoder

5.2 Channel Encoder

The channel encoder encodes the information bits choosing between a convolutional code, block code and turbo code. The implementation of the channel encoder is performed using the existing functions in the Communication System Toolbox in Matlab.

5.3 Channel Decoder

The channel decoder decodes the received code words using the Viterbi algorithm, a block decoder or a turbo decoder depending on the channel encoder chosen previously. The implementation of the channel decoder is performed using the existing functions in the Communication System Toolbox in Matlab.

5.4 Functions

This section, describes all functions related to the channel coder. These functions are the basis for all channel coding methods that the system is able to use.

5.4.1 Convo_1_2 ()

This function channel codes the information using a convolutional encoder with a code rate of 1/2. This encoding gives an error correction capability of 2 bits in a block of 6 bits, therefore, its free distance is 5 bits. The generator polynomials are $G_1 = (1, 0, 1)$ and $G_2 = (1, 1, 1)$.

Input:

bits - Matrix containing information bits for each terminal.

Output:

codedBits - Matrix containing coded bits for each terminal.

5.4.2 Convo_2_3 ()

This function codes the information bits to minimize error by using ta convolutional encoder with a code rate of 2/3. This encoding has a free distance of 9 bits. The generator polynomials are $G_{1,1} = (0, 0, 1, 0, 0, 0, 0, 1)$, $G_{1,2} = (0, 1, 1, 1, 0, 1, 0, 1)$, $G_{1,3} = (1, 0, 0, 0, 0, 1, 0, 1)$, $G_{2,1} = (0, 1, 1, 1, 1, 0, 0, 0)$, $G_{2,2} = (0, 0, 0, 0, 1, 0, 1, 1)$ and $G_{2,3} = (0, 1, 1, 0, 1, 0, 1, 0)$.

This function will zero-pad the input before performing any encoding if the length of input is odd. This is due to the fact that convolutional encoding with rate 2/3 demands an input with even length. The output *padding* will notify if any padding has occurred to make it possible for the decoder to remove the redundant bits.

Input:

bits - Matrix containing information bits for each terminal.

Output:

codedBits - Matrix containing coded bits for each terminal.

padding - Number of zero-bits that were used to pad each message.

5.4.3 Blockcode_7_4 ()

This function codes the information bits using a Hamming (7, 4) linear block encoder. The minimum distance of this coder is 3 bits.

Input:

bits - Matrix where each row contains information bits for each terminal.

Output:

codedBits - Matrix where each row contains coded bits for each terminal.

5.4.4 Blockcode_15_11 ()

This function codes the information bits using a Hamming (15, 11) linear block encoder. This coder has a minimum distance of 3 bits.

Input:

bits - Matrix where each row contains information bits for each terminal.

Output:

codedBits - Matrix where each row contains coded bits for each terminal.

5.4.5 TurboEncoder ()

This function codes the information bits using turbo coding with a code rate of 1/3 .The turbo encoder adds a new output, this is a vector that contains the indices used for the turbo encoder to make its interleaving and that the decoder has to know when it make the de-interleaving.

Input:

data - Matrix where each row contains information bits for each terminal.

Output:

encodSignal - Matrix where each row contains coded bits for each terminal.

intrlvrIndices - Interleaver parameters that the turbo decoder has to know.

5.4.6 InterBlock ()

This function performs interleaving on bit streams in order to make the channel coding more robust to burst errors. The interleaving is performed by picking every *n*:th bit in the input and wrap around the input matrix until all input bits have been placed in the output matrix, *n* is called the interleaving depth.

Similarly, the permutation vector that indicates the position of every information bit in the interleaved stream (the same for all terminals) is given as output.

Input:

Data - Matrix where each row contains information bits for each terminal.

n - Interleaving depth

Output:

intData - Matrix where each row contains coded bits for each terminal.

Perm - A vector that indicates the positions of the bits in the interleaved stream. It is needed in the de-interleaving.

5.4.7 Deconvo_1_2 ()

This function decodes bit streams that have been encoded with the function *Convo_1_2*(). It uses the Viterbi algorithm in the decoding.

Input:

codedBits - Matrix where each row contains coded information bits for each terminal.

Output:

bits - Matrix where each row contains decoded information bits for each terminal.

5.4.8 Deconvo_2_3 ()

This function decodes bit streams that have been encoded by the function *Convo_2_3*(). It uses the Viterbi algorithm in the decoding, and all zeros added by the encoder at the end of message are removed.

Input:

codedBits - Matrix where each row contains coded information bits for each terminal.

padding - Number of zero-bits that were used to pad each message in the encoder.

Output:

bits - Matrix where each row contains decoded information bits for each terminal.

5.4.9 DeBlockcode_7_4 ()

This function decodes bit streams encoded by the function *Blockcode_7_4*().

Input:

codedBits - Matrix where each row contains coded information bits for each terminal.

Output:

bits - Matrix where each row contains decoded information bits for each terminal.

5.4.10 DeBlockcode_15_11 ()

This function decodes bit streams encoded by the function *Blockcode_15_11*().

Input:

codedBits - Matrix where each row contains coded information bits for each terminal.

Output:

bits - Matrix where each row contains decoded information bits for each terminal.

5.4.11 TurboDecoder ()

This function decodes bit streams encoded by the function *TurboEncoder*().

Input:

data - Matrix where each row contains coded information bits for each terminal.

intrlvrIndices - Interleaver parameters outputted from the *TurboEncoder()* function, when the data was encoded.

Output:

decSignal - Matrix where each row contains decoded information bits for each terminal.

5.4.12 DeInterBlock ()

This function inverse the interleaving performed by the function *InterBlock*().

Input:

intData - Matrix containing the interleaved information bits.

Perm - Permutation used by the interleaver. This is acquired from the *InterBlock()* function.

Output:

Data - Matrix containing the de-interleaved information bits.

5.4.13 channelCode()

Proxy function that will take bit streams and code them with the appropriate channel coding function chosen from the functions in sections 5.4.1 to 5.4.5.

Input:

bits – A matrix of the bits to be coded, each row is treated as an independent stream.

codeType – String which chooses the channel coder, the available options are: ‘block code (7,4)’, ‘block code (15,11)’, ‘convolutional code 1/2’, ‘convolutional code 2/3’, ‘turbo code’ and ‘no channel code’.

Output:

codedBits – A matrix of coded bit streams, each row is one of the streams.

extraBits – Not used.

dim – Not used.

intrlvrIndices – The interleaving indices that are used by the turbo coder (if chosen).

padding – The amount of padding bits used on each stream.

5.4.14 channelDecode()

Proxy function that will take coded bit streams (that might contain errors) and decode them with the appropriate channel decoding function chosen from the functions in sections 5.4.7 to 5.4.11.

Input:

codedBits – A matrix of coded bit streams, each row is one of the streams.

extraBits – Not used.

dim – Not used.

intrlvrIndices – The interleaving indices that are used by the turbo coder (if chosen).

noiseVar – Not used.

padding – The amount of padding bits used on each stream.

codeType – String which tells which coding was used, the available options are: 'block code (7,4)', 'block code (15,11)', 'convolutional code 1/2', 'convolutional code 2/3', 'turbo code' and 'no channel code'.

Output:

decodedBits – The decoded bit streams, each row is one stream.

6 MISCELLANEOUS

This chapter presents the parts of the software that doesn't fit into one of the sub-systems, but are considered as other utilities.

6.1 I/O HAL

To be able to work with the hardware in an easily manageable manner, a Hardware Abstraction Layer is created. This abstraction layer provides an interface to make so called transactions. A transaction is simply a transmission that is sent from either the MIMO-array or the terminals, and that is recorded on the other end. The interface also provides means of setting which L/M-modules that are part of MIMO array and terminals respectively, setting sample- and transmission frequency on both ends individually and specifying which control group of the hardware that is used as MIMO array.

The interface is implemented with a Matlab class called IOobject. An instance of this object provides the functions described in sections 6.1.1 to 6.1.10.

6.1.1 IOobject() (Constructor)

To obtain an instance of the IOobject class, one should call the constructor of the class. This should be provided with the A/D and D/A card adaptors and hardware identifiers, which makes it possible to switch to another set of A/D and D/A cards without having to change this code. The IOobject is inherited from Matlabs class handle, which means that the constructor returns a Matlab-handle of the created instance.

Input:

ad_adaptor – The adaptor of the A/D card, in our case 'contec'.

ad_id – The hardware ID of the A/D card, in our case 'AD12-64'.

da_adaptor – The adaptor of the D/A card, in our case 'contec'.

da_id – The hardware ID of the D/A card, in our case 'AIO001'.

Output:

obj – Handle of the created IOobject instance.

6.1.2 setTerminalChannels()

This function is used to set which channels are used for the terminals. The number of channels specifies the amount of terminals. It is very important to note that duplicate channel numbers will be removed, and that channel numbers will be sorted in ascending order.

Input:

channels – Array of the channel numbers that is going to be used for the terminals.

6.1.3 setArrayChannels()

This function works exactly as *setTerminalChannels*(), but sets the channel used for the MIMO-array elements.

Input:

channels – Array of the channel numbers that is going to be used for the terminals.

6.1.4 setTerminalSampleFreq()

Sets what sampling frequency to use when recording on the terminals. Due to hardware restrictions of the current A/D card, there is a maximum limit of

$$terminal\_sample\_freq * (\max(terminal\_channels) + 1) \leq 100\,000$$

which arises from that when the A/D card is sampling, it samples from all channels up to and including the highest channel number that one wants to sample from (e.g. if one wants to only sample channel 4, then the A/D card has to sample on channels 0, 1, 2, 3 and 4). Together with the fact that the A/D card has a sampling frequency of 100 kHz that is split on all sampled channels, this gives the requirement above.

It is not certain that all choices of frequencies are compatible with the hardware; therefore this function returns the value that will actually be used.

Input:

freq – The sampling frequency that the user wants to use for the terminals.

Output:

sample_freq – The sampling frequency that will actually be used for the terminals.

6.1.5 setTerminalSendFreq()

This sets what sending frequency to use when transmitting from the terminals. Due to hardware restrictions of the current D/A card, there is a maximum limit of

$$terminals\_send\_freq * (\max(terminal\_channels) + 1) \leq 100\,000$$

which comes from that the D/A card has the exact same restrictions as the A/D card described in section 6.1.4.

Input:

freq – The sending frequency that the user wants to use for the terminals.

Output:

send_freq – The sending frequency that will actually be used for the terminals.

6.1.6 setArraySampleFreq()

Does the same as *setTerminalSampleFreq*() but for the MIMO array, with the transformed requirement of

$$array\_sample\_freq * (\max(array\_channels) + 1) \leq 100\,000$$

Input:

freq – The sampling frequency that the user wants to use for the MIMO array.

Output:

sample_freq – The sampling frequency that will actually be used for the MIMO array.

6.1.7 setArraySendFreq()

Does the same as *setTerminalSendFreq*() but for the MIMO array, with the transformed requirement of

$$terminals\_send\_freq * (\max(terminal\_channels) + 1) \leq 100\,000$$

Input:

freq – The sending frequency that the user wants to use for the MIMO array.

Output:

send_freq – The sending frequency that will actually be used for the MIMO array.

6.1.8 setArrayControlGroup()

This sets which control group that is used as MIMO array (either 1 or 2). This is a property of the hardware, and some logical control signals must be used during transactions to ensure proper functioning.

Input:

group – Which control group is used as MIMO array (1 or 2).

6.1.9 sendTerminalToArray()

This function is used to perform a transaction from the terminal to the MIMO array. It is important to ensure that one signal per terminal is provided for transmission. This function then takes care of sending control signals to the hardware, it makes sure to give the correctly formatted data to the A/D card and it also starts sampling of the D/A card.

It should be noted that signals that are to be transmitted are scaled with a constant scaling that is independent for each channel. This is done due to the fact that different speakers have different transmission characteristics.

These characteristics have been measured for all speakers by sending a pure sine of frequency 1600 Hz on each speaker in turn, and recording this with the same device. The recorded signal amplitude was then used to create a scaling of each transmission channel that produces the same amplitude out of each speaker for the same signal.

The recorded signals are likewise scaled with a constant scaling since the microphones also have different characteristics. The measurements were made in the same manner; a device was used to transmit a pure sine to each microphone in turn (with the exact same setup for each). The recorded amplitudes were used to create a scaling that should produce the same recorded amplitude for each microphone, if they receive the exact same signal.

Another important thing to note is that the D/A and A/D card has limited amount of memory, therefore this function emits a warning whenever there is a risk of exceeding this limit. To be entirely certain that this will not happen, one should never transmit signals that are longer than 2 seconds (if using a sending frequency of 6250 Hz).

Lastly, it is also important to have set all parameters (with the functions described in sections 6.1.1 through 6.1.8) before starting a transaction.

Input:

input_signals – Matrix where each row is a signal to be sent from a terminal; the first row will be sent on the first terminal (in ascending number order) etc.

samples – The number of samples to take on each MIMO array element. This parameter is optional and will correspond to the transmission time if left out.

Output:

output_signals – Matrix where each row is a signal recorded on one MIMO array element; the first row is recorded on the first array element (in ascending order) etc.

6.1.10 sendArrayToTerminal()

This function is used to perform a transaction from the MIMO array to the terminals. It works exactly the same as *sendTerminalToArray*(), with the same scaling and limitations.

Input:

input_signals – Matrix where each row is a signal to be sent from an array element; the first row will be sent on the first array element (in ascending number order) etc.

samples – The number of samples to take on each terminal. This parameter is optional and will correspond to the transmission time if left out.

Output:

output_signals – Matrix where each row is a signal recorded on one terminal; the first row is recorded on the first terminal (in ascending order) etc.

6.2 IO simulation

Since the hardware had to be shared between two project groups during the development phase, and because it was noticed that an easy way of testing the software without the actual system would be beneficial, it was decided that an IO simulation should be created. This simulation should function in the exact same way as the regular IO HAL, but it should not use the actual hardware, but a channel model instead.

The result was a new class named IOobject_sim, which provides the same functions as the IOobject with one addition.

6.2.1 init_channels()

The one added function that is provided is an initialization of the virtual channel. This creates a random channel matrix $H = (h)_{i,j}$, with the model that the physical channel between terminal i and MIMO array element j is a simple multiplication in the frequency domain with $h_{i,j} \in \mathbb{C}$.

This function also lets the user choose the variance of the AWGN that will be added on each channel, h_{min} and h_{max} such that $h_{min} \leq |h_{i,j}| \leq h_{max}\ \forall\ i,j$, and a "channel noise parameter" n such that the actual used channels during transactions are $h_{i,j}^{used} = h_{i,j} + n_{i,j}$ where $n_{i,j} \sim \mathcal{CN}(0,n),\ |h_{i,j}| \sim \text{uniform}(h_{min}, h_{max}),\ \arg(h_{i,j}) \sim \text{uniform}(0, 2\pi)\ \forall\ i,j$.

Input:

num_terminals – The number of terminals that will be used.

num_array_elements – The number of MIMO array elements that will be used.

noise_level – The variance of the AWGN that is added to each channel.

min_ampl_scale – The lowest value on amplitude scaling that a channel may have (h_{min}).

max_ampl_scale – The highest value on amplitude scaling that a channel may have (h_{max}).

channel_noise_ampl – The "channel noise parameter" n.

6.2.2 Transactions

This subsection will explain how the simulated result of a transaction is calculated.

Let us assume that the transaction is sent from the MIMO array to the terminals. The terminal channel numbers are $\boldsymbol{t} = (t_1 \ldots t_K)$, and the array element channel numbers are $\boldsymbol{a} = (a_1 \ldots a_M)$.

The transmitted signals from the array are $S = (S_1[n], \ldots, S_M[n])^T$, where $S_i[n]$ is the signal that should be transmitted on array element i, $1 \leq n \leq s_{len}$.

To simulate the clicking sound that appears at the start of a transmission, a small "burst" signal is created. The length of this signal is first decided to be b_{len}, which is uniformly distributed between 5 and 10 ms. The burst is then created as WGN with a variance that decreases linearly from being 5 at the start of the burst, to being 3.5 at the end of it. This "burst" signal is then added to the beginning of all $S_i[n]$. This behavior of the burst is entirely modeled after the appearance of several tests on the actual hardware.

To simulate that there is a non-deterministic delay between the start of the 'recording' (sampling on A/D card) and the start of the transmission, a delay k_d is calculated. This delay is uniformly distributed to represent a time delay between 0 and 5 ms. The resulting delayed signals are then $D = (D_1[n], \ldots, D_M[n])^T = (S_1[n-k_d], \ldots, S_M[n-k_d])^T,\ 1 \leq n \leq s_{len}$.

The received signal on terminal i is then:

$$R_i[n] = \sum_{j=1}^{M} IFFT\left(h_{i,j}^{used} * FFT\left(D_j[n]\right)\right) \qquad (10)$$

Observe that the multiplication in equation 10 really is a multiplication of $h_{i,j}^{used}$ with one side of the FFT spectrum, and $(h_{i,j}^{used})^*$ with the other side of it, to ensure that this corresponds to a real filter so that it gives a real-valued output.

The last thing that is done is to resample the received signals $R_i[n]$ to correspond to the used sampling frequency for the receiving side.

6.3 Sequence generators

There are two different kinds of sequence generators implemented in this project. The first one is a bit stream generator, which is capable of creating random bit streams used as data. The second one is capable of generating orthogonal preambles used in downlink data transmission. These functions are presented in more detail in sections 6.3.1 and 6.3.2.

6.3.1 bit_stream_generator()

This function produces a matrix with bit streams as rows. Each element of the matrix is either 0 or 1, and is created through a Bernoulli process with equal probability of each value.

Input:

Nstreams – The number of bit streams to generate (the amount of rows in the matrix).

Nbits – The number of bits in each stream (the amount of columns in the matrix).

Output:

bits – The matrix containing the streams.

6.3.2 preamble_generator()

This function produces a matrix *M* of orthogonal rows, which should be used as preambles. There are two variants of the generating algorithm; 'BPSK' where each row consist of only the numbers 1 and -1, and 'QPSK' where each row consists of only the numbers $1, i, -1$ and $-i$.

The algorithm uses the matrices that are generated from the following formulas:

$$A_1^{BPSK} = A_1^{QPSK} = (1) \tag{11}$$

$$B_{2j}^{BPSK} = \begin{pmatrix} A_j^{BPSK} & A_j^{BPSK} \\ A_j^{BPSK} & -A_j^{BPSK} \end{pmatrix} \tag{12}$$

$$B_{2j}^{QPSK} = \begin{pmatrix} A_j^{QPSK} & iA_j^{QPSK} \\ iA_j^{QPSK} & A_j^{QPSK} \end{pmatrix} \tag{13}$$

$$A_j^{type} = B_j^{type} \; for \; j = 2, and \; otherwise$$

$$A_j^{type} = \left(B_{j,col\,1}^{type}, \dots, B_{j,col\left(\frac{j}{2}-1\right)}^{type}, B_{j,col\left(\frac{j}{2}+1\right)}^{type}, B_{j,col\left(\frac{j}{2}+2\right)}^{type}, B_{j,col\left(\frac{j}{2}\right)}^{type}, B_{j,col\left(\frac{j}{2}+3\right)}^{type}, \dots, B_{j,col\,j}^{type} \right) \tag{14}$$

Equation 14 simply means that A_j^{type} is B_j^{type} but with a cyclic shift of three of the columns in the middle region of it (if this can be done).

The updating procedures (12) and (13) ensures that all rows of B_{2j}^{type} are orthogonal, and since A_j^{type} only has some columns switched, the rows of this matrix will also be orthogonal.

The reason behind the switching of columns is to prevent rows having a repeating pattern. E.g. without this there would in the case of BPSK exist rows such as $(1, -1, 1, -1, \dots, 1, -1)$ and $(1, -1, -1, 1, 1, -1, -1, \dots, 1, 1, -1, -1, 1)$. These repeating patterns are not good for delay estimations since this makes a delayed and phase-shifted version of the preamble resemble the original very much.

Let the wanted length of the preambles be $n = \sum_{j=K}^{N} b_j 2^j, \ \ with\ b_N, b_K \neq 0$. Then each row of M is a concatenation of one random row of each A_j^{type} such that $b_j \neq 0$, and each row of A_j^{type} may only appear in one row of M. For preambles of type 'BPSK' there is one more requirement; the first row of A_N^{BPSK} may never appear in M. This is to prevent rows of M to be constant, which also is bad for delay estimation for the same reasons as above.

Each row of M is as a final step multiplied with a random choice from the allowed numbers, i.e. $1, -1$ for 'BPSK' and $1, i, -1, -i$ for 'QPSK'.

The above described process imposes a restriction on the maximum number of preambles m for a certain choice of n:

- **QPSK**: $m \leq 2^K$
- **BPSK**: $m \leq \min(2^K, 2^N - 1)$

Input:

nPreambles – The number of wanted preambles (the amount of rows in M).

nSymbols – The length of each preamble (the amount of columns in M).

type – 'BPSK' or 'QPSK' algorithm. This input is optional; 'BPSK' will be used as default.

Output:

preambles – The matrix M containing the orthogonal preambles.

6.4 Other functions

In this chapter all functions that haven't been described yet are presented. These are not considered part of any sub-system, but are used as utilities in the final system.

6.4.1 initIO()

Initializes and returns an IOobject or IOobject_sim depending on the input. The IOobject_sim gets initialized with some fixed values. This function is a nice abstraction of the IO, the user can act completely independent of the type of IO.

Input:

terminals – The channel numbers of the terminals.

arrayTransceivers – The channel numbers of the MIMO transceivers.

fs – The sampling frequency to use on microphones.

useVirtualChannel – Boolean, should a virtual channel be used instead of real?

virtualNoiseVariance – The variance of the AWGN of the IOobject_sim (if used).

Output:

io – The IOobject/IOobject_sim, initialized and ready for transactions.

6.4.2 checkForNonASCII()

Checks if there is a non US-ASCII character in a string.

Input:

str – A string of characters

Output:

result – True if the string contains a non US-ASCII character, false otherwise.

6.4.3 stringToBits()

Encodes a string of US-ASCII characters into a one-dimensional matrix of binary numbers.

Input:

string – A string containing US-ASCII characters, non-printable characters at the end will be removed.

Output:

bits – A one-dimensional matrix containing the binary encoded characters.

6.4.4 createMessages()

Creates two binary messages from two strings containing US-ASCII characters. This function calls the *stringToBits()* function.

Input:

str1 – A string containing only US-ASCII characters where at least one character must be a printable character

str2 – A string containing only US-ASCII characters where at least one character must be a printable character

Output:

A matrix with two rows where the first row is a binary representation of *str1* and the second row is a binary representation of *str2*. The shorter string will be zero-padded if the strings are different in length.

6.4.5 bitsToString()

Decodes a matrix with binary numbers into a string containing US-ASCII characters.

Input:

bits – A one-dimensional matrix containing binary representation of US-ASCII characters. All characters must be represented by 7 binary numbers, thus must the length of the input matrix be divisible by 7.

Output:

string – A string containing the decoded US-ASCII characters.

6.4.6 readMessages()

Recreates two strings of US-ASCII characters from a matrix containing binary numbers. This function calls the *bitsToString()* function.

Input:

bits – A matrix with two rows containing binary representations of US-ASCII characters. All characters must be represented by 7 binary numbers, thus must the length of the input matrix be divisible by 7.

Output:

str1 – A string of US-ASCII characters

str2 – A string of US-ASCII characters

6.4.7 multiPlot()

Plots every row of a data matrix in separate subplots.

Input:

xData – A one dimensional matrix containing the data that is to be plotted on the x-axis.

yData – A matrix with an arbitrary number of rows containing the data that is to be plotted on the y-axis.

terminals – A vector that specifies which transceivers are as terminals.

xMax – The maximum value of the x-axis

figureTitle – A title that specifies what kind of plots are presented in the figure (optional).

xLabel – Label of the x-axis (optional).

yLabel – Label of the y-axis (optional).

6.4.8 multiStem()

Stem plots every row of a data matrix in separate subplots.

Input:

data – A matrix with an arbitrary number of rows containing the data that is to be plotted on the y-axis.

terminals – A vector that specifies which transceivers are as terminals.

figureTitle – A title that specifies what kind of plots are presented in the figure (optional).

xLabel – Label of the x-axis (optional).

yLabel – Label of the y-axis (optional).

6.4.9 constellationPlot()

Scatter plots symbols for each terminal in separate subplots. As well as the constellation points as a reference.

Input:

symbols – Matrix where each row contains symbols from different terminals.

constellation – A vector containing all the constellation points for the symbol mapping.

7 RESULTS

This section depicts the results from measurements and tests of the system. The measurements of section 7.1 were made using the hardware in a laboratory environment over a real channel.

7.1 System Performance

This section presents the performance of the final system.

7.1.1 Pilot test

To get a hint of the optimal number of pilot symbols per terminal several measurements were made keeping all parameters except the number of pilot symbols fixed according to table 3. The terminals were placed in a line approximately 1 m apart and 1 m from the array. The array was placed in a line with approximately 2 dm between each element. The result of the test is shown in table 4.

Table 3: Pilot test input data

Number of information bits	10000
Symbol rate (symbols/sec)	192
Carrier frequency	800
Number of pilot symbols per terminal	{5,10,20,40}
Encoding Algorithm	None
Interleaving	None
Interleaving depth	-
Modulator Type	QPSK
Precoder Type	ZF
Number of terminals	2
Number of array elements	14
Number of synchronization symbols	16

Table 4: Pilot test result

	Test 1	**Test 2**	**Test 3**	**Test 4**
Number of pilot symbols	5	10	20	40
Information bit rate	299	299	277	259
Information BER	0.0483	0.0282	0.02995	0.0702

7.1.2 Synchronization test

To get a hint of the optimal number of synchronization symbols several measurements were made keeping all parameters except the number of synchronization symbols fixed according to table 5. The terminals and array were placed in the same way as in the pilot test. The result of the test is shown in table 6.

Table 5: Synchronization test input data

Number of information bits	10000
Symbol rate (symbols/sec)	192
Carrier frequency	800
Number of pilot symbols per terminal	10
Encoding Algorithm	None
Interleaving	None
Interleaving depth	-
Modulator Type	QPSK
Precoder Type	ZF
Number of terminals	2
Number of array elements	14
Number of synchronization symbols	{16,32,64}

Table 6: Pilot test result

	Test 1	**Test 2**	**Test 3**
Number of synchronization symbols	16	32	64
Information bit rate	299	283	259
Information BER	0.03005	0.03085	0.2007

7.1.3 Precoder test

To compare the performance of the precoders several measurements were made keeping all parameters except the modulator type and the position of the terminal and array fixed according to table 7. The different terminal and array positions tested are explained in table 8. In table 9 and 10 the resulting BER for BPSK and QPSK is given, respectively. The rest of the measurements are done using position 4.

Table 7: Precoder test input data

Number of information bits	4000
Symbol rate (symbols/sec)	192
Carrier frequency	800
Number of pilot symbols per terminal	10
Encoding Algorithm	None
Interleaving	None
Interleaving depth	-
Modulator Type	{BPSK,QPSK}
Precoder Type	{MRT,ZF}
Number of terminals	2
Number of array elements	14

Table 8: Positions of terminals and array

Position 1	Lined terminals approx. 1 m apart, approx. 1 m from array. Lined array approx. 2 dm between each element.
Position 2	One terminal approx. 0.75 m from array and one terminal approx. 1.25 m from array. Approx. 1 m between terminals. Lined array approx. 2 dm between each element.
Position 3	One terminal approx. 0.75 m from array and one terminal approx. 1.25 m from array. Approx. 1 m between terminals. Lined array approx. 1 dm between each element.
Position 4	Lined terminals approx. 1 m apart, approx. 1 m from array. Lined array approx. 1 dm between each element.

Table 9: Precoder results using BPSK

	MRT	**ZF**
Position 1	0.0005	0.01025
Position 2	0.005	0.003
Position 3	0.00375	0.018
Position 4	0	0

Table 10: Precoder results using QPSK

	MRT	**ZF**
Position 1	0.039875	0.019125
Position 2	0.038875	0.0545
Position 3	0.059875	0.034125
Position 4	0.008625	0.005625

7.1.4 Modulator test

To compare the performance of the modulators several measurements were made keeping all parameters except the modulator type, symbol rate and number of information bits fixed according to table 11. The number of information bits was altered to have an approximately equal transmission time for each test. The resulting information bit rate and BER is shown in figure 7.

Table 11: Modulator test input data

Number of information bits	{3000,6000,9000,12000,15000}
Symbol rate (symbols/sec)	32-480
Carrier frequency	1200
Number of pilot symbols per terminal	10
Encoding Algorithm	None
Interleaving	None
Interleaving depth	-
Modulator Type	{BPSK,QPSK,8PSK}
Precoder Type	ZF
Number of terminals	2
Number of array elements	14

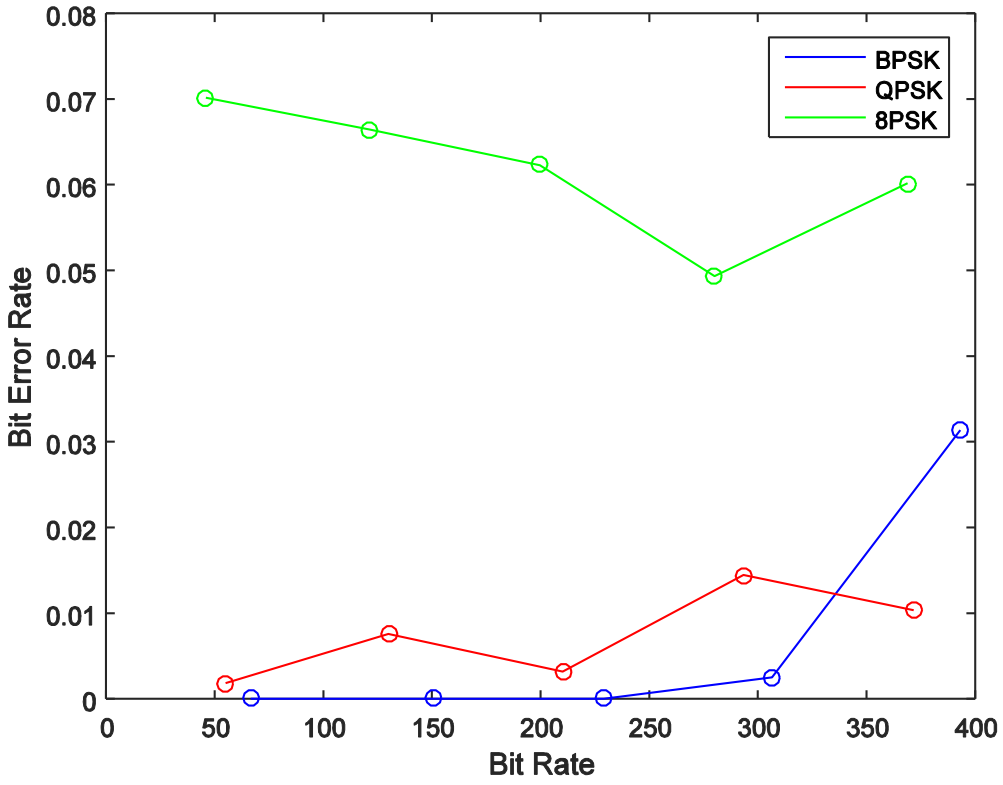

Figure 7: Bit rate and BER of different modulators

7.1.5 Channel coder test

To compare the performance of the channel coders several measurements were made keeping all input parameters except the encoding algorithm, interleaving, symbol rate and number of information bits fixed according to table 12. The number of information bits was altered to have an approximately equal transmission time for each test. The resulting information bit rate and BER is shown in figure 8. The information bit rate and BER of the channel coders that performed better than without channel coding is shown in figure 9.

Table 12: Channel coder test input data

Number of information bits	{3000,6000,9000,12000,15000}
Symbol rate (symbols/sec)	48-432
Carrier frequency	1200
Number of pilot symbols per terminal	10
Encoding Algorithm	{No code, Block code (7,4), Block code (15,11), Convolutional 1/2, Convolutional 2/3, Turbo code}
Interleaving	{Yes, No}
Interleaving depth	10
Modulator Type	QPSK
Precoder Type	ZF
Number of terminals	2
Number of array elements	14

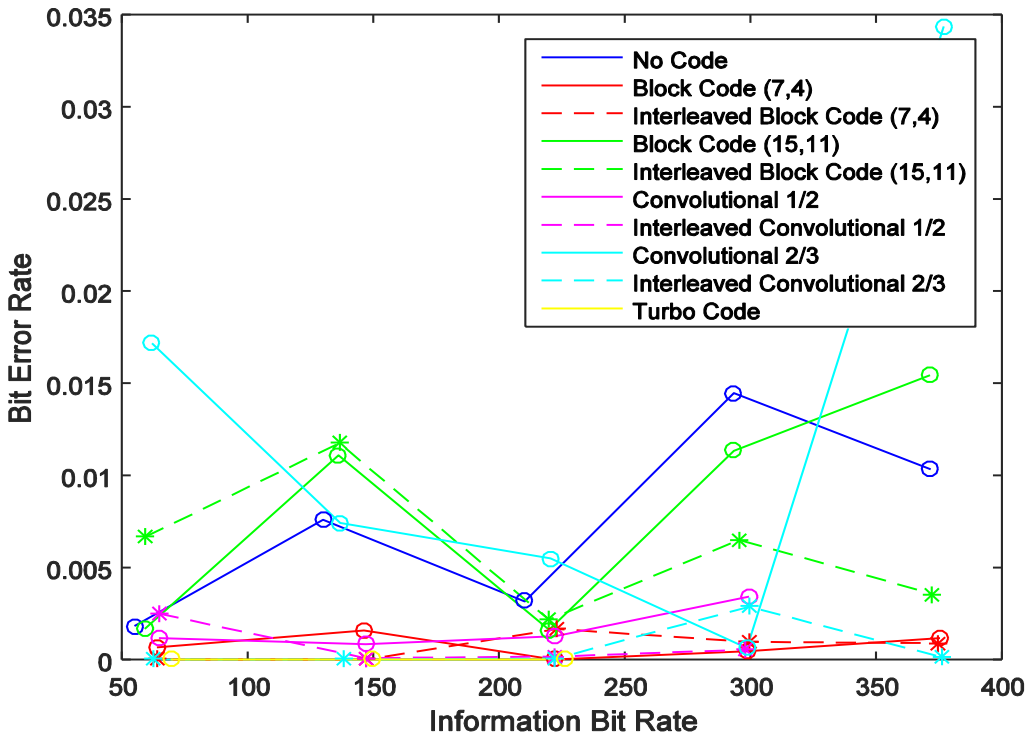

Figure 8: Resulting information bit rate and BER of channel coder test

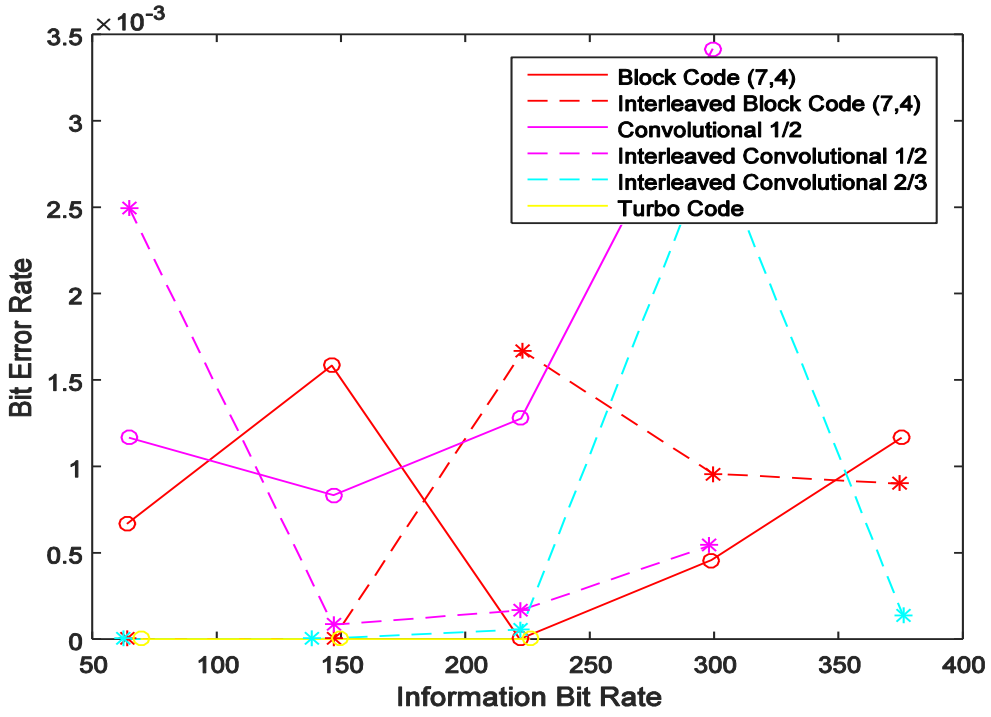

Figure 9: Resulting information bit rate and BER of the best performance in channel coder test

7.1.6 Several terminals

A test was performed to measure the system performance when expanding the number of terminals. All input parameters were kept fixed except the number of information bits, symbol rate, number of terminals and the positioning of the terminals according to table 13. The number of information bits was altered to have an approximately equal transmission time for each test. The position of the array and terminals was according to position 4 in table 8 except of the spacing between the terminals. When using 3 terminals the spacing between them were approximately 0.75 m and when using 4 terminals the spacing between them were approximately 0.5 m. The resulting information bit rate and BER is shown in figure 10.

Table 13: Several terminals input data

Number of information bits	{3000,6000,9000}
Symbol rate (symbols/sec)	{144,288,432}
Carrier frequency	1200
Number of pilot symbols per terminal	10
Encoding Algorithm	Turbo code
Interleaving	Yes
Interleaving depth	10
Modulator Type	QPSK
Precoder Type	ZF
Number of terminals	{3,4}
Number of array elements	12

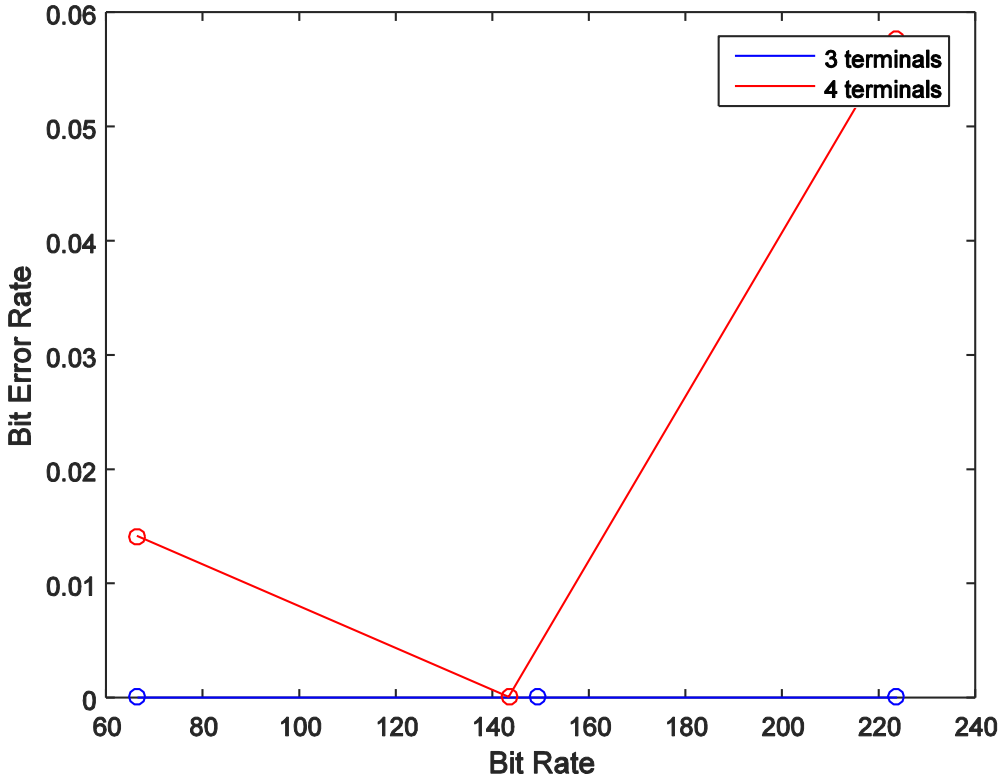

Figure 10: Resulting information bit rate and BER using 3 and 4 terminals

8 PROBLEMS AND LIMITATIONS

This chapter will present the limitations of the final product, and the problems that unfortunately exist.

8.1 Hardware

The big problem with the hardware was that all L/M units have quite different characteristics. Both the loudspeakers and the microphones have different sensitivities. As described in the IO HAL section, an attempt at mitigating this is implemented, where a constant scaling is used on each channel for loudspeakers and microphones. However, it seems as though the sensitivity is frequency dependent, and therefore the scaling might be off. Another thing that was noted when measurements on the characteristics were conducted was that the results wasn't very consistent. Either way it was decided to keep using the constant scaling, but a better solution would probably be to make even better measurements that incorporate the frequency dependency.

A smaller problem with the microphones is that the slave devices are quite hollow, such that sound might resonate inside of them, adding noise to the transmissions.

A third problem is with the A/D card, which sometimes malfunctions with the message "Sample Clock Error". When this happens the current A/D conversion is stopped, which for the system means that the sound transmission is cut off. The system will keep running and

start a new transmission in the next frame, but because no signal was sent for some amount of time, a large number of bit errors are to be expected. The project has not tried to implement retransmission if this situation happens, and has not been able to find any information about how to avoid this problem.

8.2 Software

The software system that was created is mostly designed with the aim to have as few limitations as possible. None of the sub-systems or the IO HAL restricts the number of devices used as terminals and as MIMO array (and they don't assume that the total is 16). The massive_MIMO_transmission script is actually functional for a configurable number of terminals and array elements with very few restrictions. The limitation on the number of terminals would be the point where the pilot signals needs to be longer than 2 seconds, where the A/D card's memory might not be sufficient.

However, the GUI program imposes some extra restrictions on the input parameters to avoid strange situations. These restrictions are described in the section about the GUI.

The absolute biggest software problem is the library that is distributed by Contec (the A/D and D/A card provider), which makes it possible to use the Matlab Data Acquisition Toolbox to interact with the A/D and D/A cards. This library is unfortunately only supporting 32 bit systems, while the computer used in the project is running 64 bit. This causes the computer to sometimes crash in a Blue Screen Of Death (BSOD). This problem was already known before the start of the project and therefore the design of the IOobject supports a change of implementation. The idea was to create a driver wrapper in C that could be called from Matlab and completely avoid using the Data Acquisition Toolbox. Since the BSOD problem occurs once every 3-6 hours on average, and since it would require a lot of time, this was never implemented. It should however be considered in future projects that use the same hardware.

9 CONCLUSIONS

This chapter presents the conclusions that can be drawn about the system.

9.1 System Performance

These are conclusions drawn from the measurement results of the system performance. Because of sampling clock errors and blue screens all testing needed to be supervised and couldn't be automated. A transmission takes time and the project has limited resources. Therefore we had to limit the amount of bits sent and number of tests done. The statistical results should therefore be seen as a weak estimate of the system performance. For a more accurate result and backing of conclusions a larger data set should be used.

9.1.1 Bandwidth

The system performance is limited by the bandwidth of the loudspeaker. Too low and too high carrier frequency results in poor system performance. From the results of informal testing we recommend keeping the carrier frequency between 500 Hz and 1200 Hz to get the best system performance.

9.1.2 Pilots

The pilot test showed that using 10 pilot symbols gave the highest bit rate together with the lowest BER and therefore the best system performance. Therefore the rest of the testing was performed using 10 pilot symbols per terminal.

9.1.3 Synchronization

The synchronization test showed that using 16 synchronization symbols gave the highest bit rate together with the lowest BER and therefore the best system performance. The number of synchronization symbols used was therefore hardcoded to be 16.

9.1.4 Precoder

The precoder test showed that position 4 gives the best system performance. MRT seems to perform best using BPSK and ZF seems to perform best using QPSK. More informal tests showed that MRT seems to be more dependent of the positioning of the terminals and the array. This is reasonable because MRT doesn't account for reducing the co-channel interference, in contrast to ZF. This is the reason for ZF being used in most of the other tests, to minimize the risk of co-channel interference.

9.1.5 Modulator

The modulator test shows that 8PSK performs worse than BPSK and QPSK. For lower bit rates BPSK performs best. Although when using higher bit rate QPSK performs better than BPSK. This is probably due to the bandwidth limitations of the loudspeaker. BPSK requires higher bandwidth than QPSK and therefore exceeds the bandwidth of the system at a certain rate. Another reason for QPSK performing better at high rates could be limitations of the sampling frequency. QPSK uses twice the amount of samples per symbols compared to BPSK. To utilize the bandwidth QPSK was used in most of the other tests.

9.1.6 Channel coder

It is hard to tell which channel code performs best. The bandwidth limitations of the system becomes even more of a problem when using channel coding, because the symbol rate needs to be increased to match the information bit rate of an uncoded transmission. The turbo code, using the lowest code rate of 1/3, performs best but can only reach information bit rates of approximately 225 bits/s before exceeding bandwidth. The same problem occurs using the convolutional code of rate 1/2 when reaching an information bit rate of about 300 bits/s. For higher bit rates the block code (7,4) and interleaved convolutional code of rate 2/3 performs best. The block code (15,11) and convolutional code of rate 2/3 without interleaving performed worst and made no obvious improvements of the system performance. The use of interleaving seems to have little effect on the system performance when combined with block code. However, interleaving seems to improve the system performance when combined with convolutional coding.

9.1.7 More terminals

When using 3 or 4 terminals the number of array elements can only be a maximum of 12, because of hardware limitations. Another hardware limitation is the short length of the cables connected to the terminals forcing the terminals to share a limited space in the laboratory environment and therefore decreases the maximum space between the terminals. These limitations make the beamforming less accurate. However, the measurements show that using

3 terminals works great together with turbo coding. When using 4 terminals some errors are introduced.

10 REFERENCES

Published references

Svensson, T. & Krysander, C. (2011). Projektmodellen Lips, Upplaga 1:1, Studentlitteratur AB, Lund, ISBN 978-91-44-07525-9

Electronic references

Sörman, S. (2014). User Manual Version 1.0 (PDF) Available: <http://www.isy.liu.se/edu/projekt/kommunikationssystem/2015/pdt/documents/User%20Manual%20v1.0.pdf >

Ngo, H, Q. (2015). Massive MIMO: Fundamentals and System Designs (PDF) Available: < http://liu.diva-portal.org/smash/get/diva2:772015/FULLTEXT01.pdf>

11 APPENDIX

Below follows a list of files and folders that should exist in the PDT folder for the system to work.

```
bit_stream_generator.m
bitsToString.m
checkForNonASCII.m
constellationPlot.m
createMessages.m
massive_MIMO_transmission.m
multiPlot.m
multiStem.m
NO_GUI_transmission.m
preamble_generator.m
readMessages.m
stringToBits.m

GUI/
   GUI.fig
   GUI.m

Modulator/
   detector_8psk.m
   detector_bpsk.m
   detector_qpsk.m
   detector.m
   distances.m
   downlink_demodulator.m
   mapper.m
   MRT.m
   symbol_mapper_8psk.m
   symbol_mapper_bpsk.m
   symbol_mapper_qpsk.m
   upconverter.m
   uplink_demodulator.m
   ZF.m

Channel coding/
   Blockcode_15_11.m
   Blockcode_7_4.m
   channelCode.m
   channelDecode.m
   Convo_1_2.m
   Convo_2_3.m
   DeBlockcode_15_11.m
   DeBlockcode_7_4.m
   DeInterBlock.m
   Deconvo_1_2.m
   Deconvo_2_3.m
   InterBlock.m
   TurboDecoder.m
   TurboEncoder.m
```

```
Channel_estimator/
   channel_estimator.m
   pilot_generator.m

IO/
   IOobject.m
   IOobject_sim.m
   initIO.m
```