

# 1 ICT Laboratory Overview - CIT Master

## 1.1 Introduction

The ANT part of the ICT laboratory held in the summer term is meant to be solved in groups of two in an independent fashion with minimal help from tutors. You are expected to solve problems on your own and organize your work as you see fit.

Due to the online format of this years ICT lab II questions must be posted on the discussion boards available in Stud.IP. The tutors will provide feedback after the intermediate deadline detailed in the timetable in section 1.3.

CIT masters are expected to implement a basic OFDM point-to-point transmission chain. In the following, Section 1.2 discusses the specific goals and requirements of this lab in more detail. Then, Section 1.3 introduces the lab dates and the general timing of the lab over the whole summer term. Section 2 explains the evaluation guidelines that will be used to evaluate if the lab has been passed successfully or not. Finally, the task description, programming and simulation guidelines as well as requirements for each communication blocks are given in Section 3.

## 1.2 Goals and Requirements

### 1.2.1 Requirements

This laboratory is mandatory for CIT master students. Besides different Bachelor backgrounds we expect you to have certain knowledge and skills at the beginning of the laboratory. To some degree, it is expected that you will have to research topics less well known to you, but nonetheless the following is expected:

- Self-motivated working style (researching unknown topics with minimal tutor help)
- Communications technology knowledge
- Basic knowledge of MATLAB or PYTHON

### 1.2.2 Goals

The following goals are targeted with this laboratory:

- Self-motivated problem solution including research and collaboration with other lab attendees
- Deepening knowledge about all the basic processing steps in multi-carrier communications from system dimensioning to equalization
- Developing programming skills in MATLAB or PYTHON

### 1.3 Time line

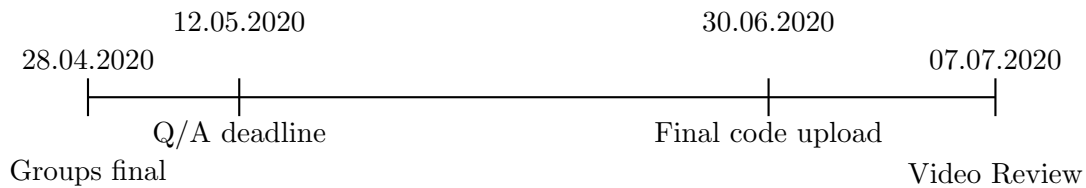


Figure 1: Time line for ICT Lab 2 summer semester 2020.

This laboratory is planned to be running alongside other courses during the whole summer term with only a few predetermined deadlines for an overall workload of 1CP or 30h for CIT masters. Figure 1 shows the specific dates for summer semester 2020 and their individual purpose.

The first date is the mandatory deadline to finalize the groups required for this lab. Then up to the second date questions will be collection on the ICT lab discussion board. The tutors will provide answers at or shortly after the deadline. The third date is the deadline to upload your final code to the Stud.IP "Code Drop (CIT)" folder. The tutor will test each groups implementation according to the API requirements described in Sections 3.4.2 and 3.4.4 and check the code for proper comments. If the code follows the API and provides the required figures and results, your group proceeds to the video review. If your code does not follow the API or has major problems not easily analysed by the tutor, your group will fail this lab. Minor problems can be fixed within a one week window up to the 07.07. Finally, a tutor will invite you to a video conference to check the fulfillment of the overall tasks. Each group has to answers three questions regarding their code. The purpose of this is to check your ownership of the code and your understanding of the underlying theory. If the tutors deems your answers sufficient, you pass.

**Groups who do not pass this test will have to repeat ICT lab 2 in the following year.**

## 2 Evaluation Guidelines

### 2.1 General Rules

Besides the solution of the task that is detailed below, we expect you to adhere to some general rules:

- Solve the tasks by yourselves.
- Write your own code and do not copy!

- Design your own slides and do not copy (pictures, too)!

Group efforts in solving the sub tasks are encouraged and expected, but we will collect the solutions of all groups at the end of the lab and test your personal knowledge about your solution. The goal of this lab is to enhance your ability to break down bigger tasks into smaller steps, organize your work and research for yourself. If you just copy the solution of other groups, you will simply limit your own benefit.

## 2.2 What to expect from the tutors?

The tutors will help you understand the tasks, may give you help finding the right information and evaluate your work to judge if you have passed or not.

Most importantly:

- Tutors will **not** write MATLAB or PYTHON code for you!
- Tutors will give you hints and tips to help you to *find the solution yourself*!
- Tutors will only help you if you *follow the guidelines* and API descriptions given in this document!

## 2.3 Required Performance

To pass ICT lab 2, your code will be firstly checked by the tutor in a **code pretest** after the code upload deadline, then your performance will be finally rated on the date of the video review.

### 2.3.1 Code Pretest

According to the time line in Fig. 1, you must upload your ready-to-run code to the correct StudIP folder in time for a code pretest, where your code will be checked according to the following expectations:

- Each block must be implemented according to the API definition and requirements.
- We expect you to write clean **and** well documented MATLAB or PYTHON code that is easily readable by the tutor. Consider this lab to be part of a job that will be carried on by another team after you finish.
- You should write your code strictly following the programming and simulation guidelines in 3.2 and 3.3.

Only the students who pass the code pretest have the chance to attend the final video conference for evaluation.

### 2.3.2 Video Review

Additionally to the code pretest to measure the API functionality, on the date of the video review, the tutor will ask you 3 main questions about your implementation to test your individual grasp of the solution.

## 3 Task Description

### 3.1 General Description

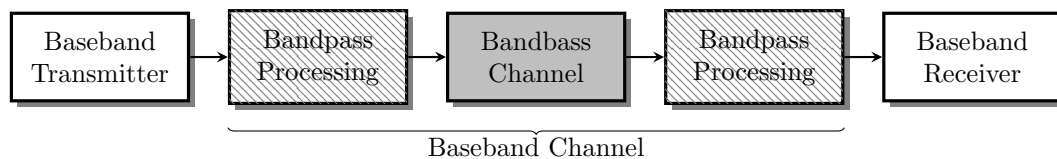


Figure 2: Overview of a point-to-point communication setup. Shaded and gray marked blocks will be provided.

The general idea of this lab is the implementation of a complete point-to-point OFDM communication chain as illustrated in Fig. 2, including transmitter, channel and receiver. To restrict the breadth of this task, only the baseband processing at transmitter and receiver indicated by white blocks has to be implemented by each group. An equivalent baseband channel model will be provided to test the overall communication chain. This model summarizes all channel and hardware effects that are attributed to bandpass processing, including but not limited to up/down conversion, amplification, antenna patterns, and so on. However, some of the bandpass effects will be included into the lab by equivalent baseband descriptions as “non-linear hardware” (see the following sections for more details).

The baseline OFDM system has to be implemented of the lab according to the specifications in Section 3.4.

### 3.2 Programming Guidelines

You can choose either MATLAB or Python to program.

- Use a main program like provided in the file `main`, where the main parameters are defined and in which all functions are called
- Create one function per API block, e.g. `digital_source` is one function
- Follow the API definitions exactly, i.e. all functions and parameters must be named as specified!

- Avoid available implementations of communication functions, e.g. *do not* use a function like `modulate` or `quantize`

### 3.3 Simulation of BER and PAPR

In addition to the transmitter and receiver implementation a simulation environment has to be created that uses the transmitter and receiver implementations to numerically analyze the performance of the whole multi-point-to-point communication chain. The following requirements have to be fulfilled:

- Allow simulation of different SNRs, e.g., using an outer loop.
- Calculation and collection of PAPR for OFDM
- Save the results in terms of uncoded/coded bit error rate (BER) for different SNR choices in a vector.
- Save the results in terms of PAPR of  $\mathbf{x}$  at the non-linear hardware output
- Plot the uncoded/coded BER per user vs. the SNR.
- Figure showing “live” PAPR statistics
- Observe and explain changes with frequency offsets switched on

Note that the number of simulated packets, i.e. the total number of bits simulated, defines the quality of the simulation results. For low BER values a high number of bits must be simulated, e.g. at least  $10^4$  bits for BERs of  $10^{-3}$ .

### 3.4 Basic OFDM transmission

#### 3.4.1 Transmitter Model

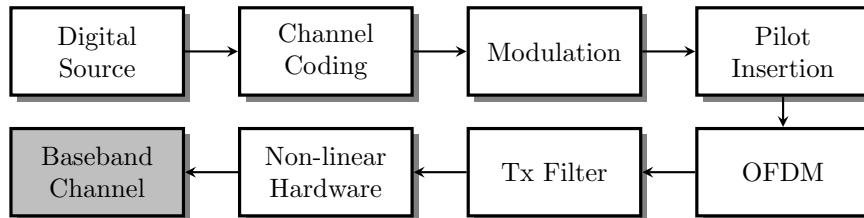


Figure 3: General structure of the Baseband Transmitter as introduced in Fig. 2 with interface to Baseband Channel. Gray blocks will be provided, white ones are to be implemented according to the specifications.

In this phase of this lab, the transmitter and receiver (see, Section 3.4.3) of a basic point-to-point communication chain has to be implemented according to the specifications below. Fig. 3 shows the building blocks of such a transmitter of which all white blocks need to be implemented, whereas the gray blocks will be provided. Each block is defined by its inputs and outputs and a short requirements list that describes the functionality

in Section 3.4.2. Your task is the fulfillment of these requirements for each block while adhering to the specified inputs, outputs and function names. Please note, that some blocks are marked as “switchable” by a parameter `switch_off`, which means that such a block should not change the input data in any way if switched “off” by `switch_off=1`, i.e., `output=input`.



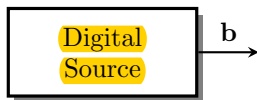
Additionally to functional requirements, e.g., a specific Tx filtering, also optional graphical output may be required. For example, the Tx filter input and output may be plotted in a figure to show changes in the shape of the spectrum. Graphical output should always be optional, i.e., controlled by a switching variable `switch_graph`, to analyze your implementation and the results as needed.

### 3.4.2 Transmitter API Definitions

---

```
b = generate_frame(frame_size, switch_graph)
```

---




This block as `digital source generate` frames of random binary data.

**Parameters:**

`frame_size` indicates the frame length

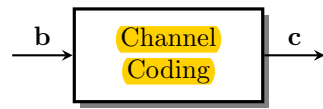
**Requirements:**

1. Choose the frame length `frame_size` according to the OFDM parameters.
  2. Generate a mean free bin  sequence of equally probable zeros and ones.
  3. Show a figure of the binary pattern of one frame.
-

---

```
c = encode_hamming(b, parity_check_matrix, n_zero_padded_bits, switch_off)
```

---



This block facilitates Channel Coding by a [7,4] Hamming code.

**Parameters:**

<code>parity_check_matrix</code>	code parity check matrix
<code>n_zero_padded_bits</code>	number of zeros should be added after encoding

**Requirements:**

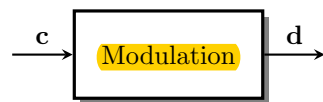
1. Correct channel encoding with generator matrix calculated by `parity_check_matrix`.
2. Restructure the binary signal **b** into blocks of channel coding block length and encode blocks via the [7,4] Hamming block code.

Note: the required number of bits may not be exactly achievable due to the coding. Add `n_zero_padded_bits` zeros after encoding to alleviate this effect.

---

```
d = map2symbols(c, constellation_order, switch_graph)
```

---



This block facilitates Bit to Symbol mapping (sometimes also called Modulation) of the encoded bit sequence to either 4-, 16-, or 64-QAM.

**Parameters:**

<code>constellation_order</code>	adjust modulation;
	<code>constellation_order=2</code> 4-QAM,
	<code>constellation_order=4</code> 16-QAM,
	<code>constellation_order=6</code> 64-QAM.

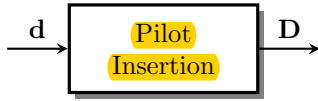
**Requirements:**

1. Correct modulation with Grey mapping.
2. Show a figure of the modulated symbols of one block.
3. Normalize the average symbol power to 1.

---

```
D = insert_pilots(d, fft_size, N_blocks, pilot_symbols)
```

---



This block facilitates the framing of data into `N_blocks` blocks of `fft_size` symbols and prepends pilot data for one OFDM pilot symbol.

**Parameters:**

`fft_size`            FFT length /OFDM symbol length  
`N_blocks`            number of blocks  
`pilot_symbols`      generated pilot symbols

**Requirements:**

1. Restructure the serial data stream `d` into blocks of FFT length symbols as a matrix `D`.
2. Insert one block of known pilots for channel estimation.



Use a standard FFT length `fft_size=1024`.

---

```
z = modulate_ofdm(D, fft_size, cp_size, switch_graph)
```

---



This block facilitates OFDM modulation and CP insertion. The parameters should be chosen to achieve Inter-Symbol Interference (ISI) free transmission given the channel statistics.

**Parameters:**

`cp_size`            CP length

**Requirements:**

1. Choose correct parameters to alleviate the effects of the effective channel impulse response in the over-sampled domain.
2. Correct OFDM modulation and CP insertion.
3. Serialize the `N_blocks+1` OFDM symbols to a single time sequence `z`.
4. Show a figure of one OFDM symbol in time and frequency domain.





---

```
s = filter_tx(z, oversampling_factor, switch_graph, switch_off)
```

---



This block facilitates filtering of the OFDM symbols with a digital low-pass filter to suppress out of band radiation from OFDM modulation. This is relevant to decrease interference on neighboring channels (e.g. in WLAN).

**Parameters:**

`oversampling_factor`      oversampling factor

**Requirements:**

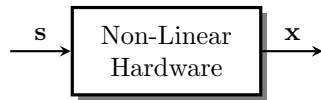
1. Filter sequence **z** with an appropriate filter to reduce the out of band radiation 40dB below the signal level using an oversampling factor `oversampling_factor=20`.
2. Normalize the filter output signal appropriately to ensure that the power of the signal is 1.
3. Show a figure of the designed low-pass filter with bandwidth and side lobe suppression.
4. Show a figure of the filter output.

Note: Filter design tools can be used to design an appropriate filter (see, e.g., DSP exercises, SciPy `scipy.signal.firwin`).

---

```
x = impair_tx_hardware(s, clipping_threshold, switch_graph)
```

---



This block models the influence of an amplifier on the baseband signal by hard thresholding.

**Parameters:**

`clipping_threshold`    tx clipping threshold

**Requirements:**

1. Implement a simple hard thresholding function that limits the *absolute value* of the baseband signal **s** such that it is linearly scaled to be smaller than 1 up to values of `clipping_threshold` and clipped to 1 if greater than `clipping_threshold`.
2. Ensure that the phase of **s** is not changed by this block.
3. Analyze distortions by this block with different threshold levels (e.g., weak clipping, no clipping, etc.).
4. Show a figure of (non-)clipped signal / Show that (no) clipping is in effect.

---

```
y = simulate_channel(x, snr_db, channel_type)
```

---



This block models a frequency selective baseband channel that distorts the wideband OFDM signal and adds white Gaussian noise to the signal.

**Parameters:**

**snr\_db** will be used to check the performance of the transceiver chain at different SNRs (in dB).  
**channel\_type** type of channel. 'AWGN': Additive White Gaussian Noise, 'FSBF': Frequency Selective Block Fading.

---

### 3.4.3 Receiver Model

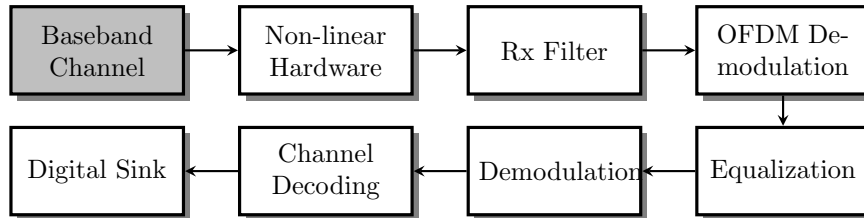


Figure 4: General structure of the Baseband Receiver with interface to Baseband Channel. Gray blocks will be provided, white ones are to be implemented according to the specifications.

The first phase of this lab also comprises the implementation of the OFDM receiver for a frequency selective block-fading channel and the overall simulation. Fig. 4 shows the building blocks of such a receiver and Section 3.4.4 details the individual blocks in terms of inputs, outputs and requirements. To simplify the task some parameters can be assumed as known at the receiver side, i.e., the scaling of the transmit signal is also known. This also applies to modulation, channel code and frame length.

### 3.4.4 Receiver API Definitions

---

```
s_tilde = impair_rx_hardware(y, clipping_threshold, switch_graph)
```

---



This block models the influence of an amplifier on the baseband signal by hard thresholding.

**Parameters:**

`clipping_threshold`                      rx clipping threshold

**Requirements:**

1. Implement a simple hard thresholding function block that is transparent and does not change the signal in any way.
2. Show a figure of received signal and signal after hardware indicating that no clipping is in effect

---

```
z_tilde = filter_rx(s_tilde, downsampling_factor, switch_graph, switch_off)
```

---



This block facilitates filtering of the received signal with a digital low-pass filter.

**Parameters:**

`downsampling_factor`                      downsampling factor

**Requirements:**

1. Filter the received signal with a matched low-pass filter using an downsampling factor of `downsampling_factor=20`, i.e., identical to the transmitter side.
2. Normalize the filter output signal appropriately to ensure that the power of the signal is not changed.
3. Show a figure of the filter output
4. Show a figure of eye pattern




---

```
D_tilde = demodulate_ofdm(z_tilde, fft_size, cp_size, switch_graph)
```

---



This block facilitates OFDM demodulation, i.e. FFT and cyclic prefix removal.

**Requirements:**

1. Use the same parameters as for OFDM modulation
2. Correct OFDM demodulation and CP removal
3. Show a figure in the symbol space

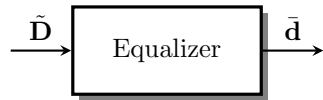




---

```
d_bar = equalize(D_tilde, pilot_symbols, switch_graph)
```

---



This block facilitates channel estimation using the inserted pilots and equalizes the received symbols accordingly.

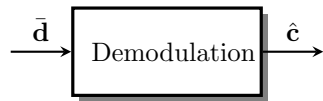
**Requirements:**

1. Extract pilots and estimate channel using inserted pilots
2. Equalize data for all OFDM channels
3. Show a figure of equalized symbols

---

```
c_hat = detect_symbols(d_bar, constellation_order, switch_graph)
```

---



This block facilitates hard estimation of the code bits for 4-, 16- or 64-QAM.

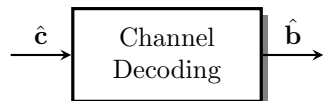
**Requirements:**


1. Decide the received signal to 4-, 16- or 64-QAM symbols with Gray mapping to estimate the code bits.
2. Ensure proper processing in terms of the channel encoded blocks afterwards.
3. Show a figure of the estimated symbols with decision thresholds

---

```
b_hat = decode_hamming(c_hat, parity_check_matrix, n_zero_padded_bits, switch_off, switch_graph)
```

---



This block facilitates Channel Decoding of a [7,4] Hamming code. 

**Requirements:**

1. Correct errors in the estimated code words of the [7,4] Hamming block code by syndrome decoding.
2. Show a figure of exemplary code word indicating corrected errors

Note: Do not forget to remove the additional zeros potentially inserted at the encoder.

---

---

```
BER = digital_sink(b, b_hat,...)
```

---



Processing of the reconstructed and original signal to analyze the errors due to transmission. Here, the digital sink represents the analysis of the received and reconstructed signals. Knowledge of all other signals in the system is implicitly assumed.

**Requirements:**

1. Calculate the error in terms of the coded and uncoded bit error rate (BER).
  2. Show a figure of binary estimate and the original signal indicating erroneous positions.
  3. Show a figure of BER curve w.r.t. different SNRs
-