Lab Exercises

Modules:

Communications Engineering
and
Computer Engineering

Experiment at Ti :

Implementing an OFDM Transceiver by Software Defined Radio

Location: Seminar room 24 A 407, Walter-Schottky-Haus, Sommerfeldstraße 24, 52074 Aachen, at TI

Contact: `laboratory@ti.rwth-aachen.de`

Name of the student: ______________________________

Important remark:

Before coming to lab, students should read the script and solve the preparatory exercises in Chapter 5.

Institute for Theoretical
Information Technology
Univ.-Prof. Dr. rer. nat. Rudolf Mathar

RWTHAACHEN
UNIVERSITY

Contents

1 Introduction

Current broadband wireless standards are based on Orthogonal Frequency Division Multiplexing (OFDM), a multi-carrier modulation scheme which provides strong robustness against intersymbol interference (ISI) by dividing the broadband channel into many orthogonal narrowband subchannels in such a way that attenuation across each subchannel stays flat. Orthogonalization of subchannels is performed with low complexity by using the fast Fourier transform (FFT), an efficient implementation of discrete Fourier transform (DFT), such that the serial high-rate data stream is converted into multiple parallel low-rate streams, each modulated on a different subcarrier.

There is a variety of systems using OFDM already as WLAN (Wireless LAN, IEEE 802.11), WiMAX (Worldwide Interoperability for Microwave Access, IEEE 802.16), DAB (Digital Audio Broadcasting), DVB (Digital Video Broadcasting), DSL (Digital Subscriber Line), etc. Beside the existing systems there is active research on future systems, e.g. LTE (Long Term Evolution), enhancing the existing standards to improve system performance. The investigation and assessment of information theoretic concepts for wireless resource management of those new systems in real-world scenarios requires flexible testbeds with a wide range of reconfigurable parameters. This functionality is currently offered in Software Defined Radio (SDR) technology based on general purpose hardware only.

We designed a modular, SDR based and reconfigurable framework which treats the OFDM transmission link as a black box. The given framework contains transmitter and receiver nodes that are composed of a host commodity computer and a general purpose radio frequency (RF) hardware, namely Universal Software Radio Peripheral (USRP). Baseband OFDM signal processing at host computers is implemented in the GNU Radio framework, an open source, free software toolkit for building SDRs [GNUac].

The control and feedback mechanisms provided by the given framework allow for reconfigurable assignments of predefined transmission parameters at the input and estimation of link quality at the output. High flexibility, provided by a large set of reconfigurable parameters, which are normally static in real systems, enables implementation and assessment of different signal processing and resource allocation algorithms for various classes of system requirements.

During this lab exercises the SDR concept will be studied. Insight into the high flexibility in system design offered by SDR or comparable systems and corresponding architectural constraints will be gained. Within the framework, high reconfigurability of transmission parameters allows for easy assessment and evaluation of OFDM system performance in real wireless channel conditions and for comparison with theoretically derived results.

This script is organized as follows. An introduction to basic OFDM system's characteristics is given in Chapter 2. In Section 2.1, a corresponding discrete-model is introduced and applied for analytical assessment of the influence of system impairments which are discussed in Section 2.2. A short survey of coherent modulation techniques commonly used in OFDM systems and their performance evaluation in additive white Gaussian noise (AWGN) channels are presented in Section 2.3. Basic principles, architectural concepts of SDR and an introduction to GNU Radio framework are pictured in Chapter 3. In Section 3.1, system benefits and practical limitations of SDR are addressed. Deeper insight into GNU Radio architecture and an example of wireless channel simulation within a given framework can be gained in Section 3.2. In Chapter 4, a detailed system description of the SDR framework, which will be used for lab exercises, is given. Finally, the preparatory and lab exercises are described and corresponding tasks are depicted in Chapters 5 and 6.

2 OFDM Basics

In this chapter the basic principles of OFDM baseband signal processing are given and an appropriate discrete-time OFDM system model is introduced. In the following section impact and prevention of synchronization errors and equalization are explained. Finally, digital modulations commonly used in wireless transmission standards are described in Section 2.3.

OFDM is a multi-carrier modulation scheme that is widely adopted in many recently standardized broadband communication systems due to its ability to cope with frequency selective fading [PMK07]. The block diagram of a typical OFDM system is shown in Fig. 2.1. The main idea behind OFDM is to divide a high-rate encoded data stream (with symbol rate T_S) into N parallel substreams (with symbol rate $T = NT_S$) that are modulated onto N orthogonal carriers (referred to as subcarriers). This operation is easily implemented in the discrete time domain through an N-point inverse discrete Fourier transform (IDFT) unit and the result is transmitted serially. At the receiver, the information is recovered by performing a DFT on the received block of signal samples. The data transmission in OFDM systems

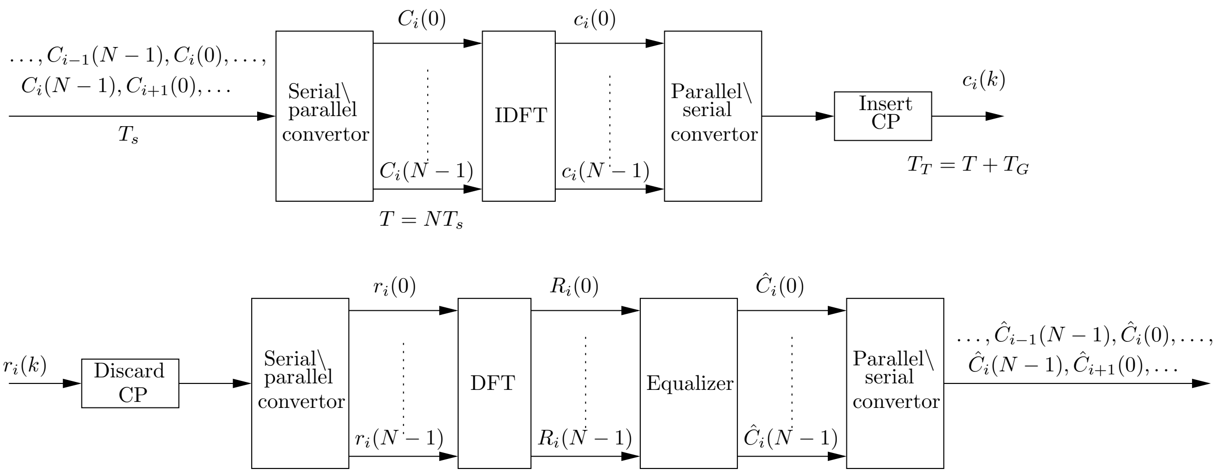

$\ldots, C_{i-1}(N-1), C_i(0), \ldots,$
$C_i(N-1), C_{i+1}(0), \ldots$
T_s
Serial\ parallel convertor
$C_i(0)$
$C_i(N-1)$
$T = NT_s$
IDFT
$c_i(0)$
$c_i(N-1)$
Parallel\ serial convertor
Insert CP
$c_i(k)$
$T_T = T + T_G$
$r_i(k)$
Discard CP
Serial\ parallel convertor
$r_i(0)$
$r_i(N-1)$
DFT
$R_i(0)$
$R_i(N-1)$
Equalizer
$\hat{C}_i(0)$
$\hat{C}_i(N-1)$
Parallel\ serial convertor
$\ldots, \hat{C}_{i-1}(N-1), \hat{C}_i(0), \ldots,$
$\hat{C}_i(N-1), \hat{C}_{i+1}(0), \ldots$

Figure 2.1: Block diagram of a typical OFDM system

is accomplished in a symbolwise fashion, where each OFDM symbol conveys a number N of samples of (possibly coded) complex data symbols. As a consequence of the time dispersion associated with the frequency-selective channel, contiguous OFDM symbols may partially overlap in the time-domain. This phenomenon results into inter symbol interference ISI, with ensuing limitations of the system performance. The common approach to mitigate ISI is to introduce a guard interval of appropriate length among adjacent symbols. In practice, the guard interval is obtained by duplicating the last N_G samples of each IDFT output and, for this reason, is commonly referred to as cyclic prefix (CP). As illustrated in Fig. 2.2, the CP is appended in front of the corresponding IDFT output. This results into an extended OFDM symbol consisted of $N_T = N + N_G$ samples which can totally remove the ISI as long as N_G is properly designed according to the channel delay spread.

Referring to Fig. 2.1, it can be seen that the received samples are divided into adjacent segments of length N_T, each corresponding to a different transmitted OFDM symbol. Without loss of generality, lets concentrate on the ith OFDM symbol at the receiver. The first operation is the CP removal, which is simply accomplished by discarding the first N_G samples of the considered segment. The remaining N samples are fed to a DFT and the corresponding output is subsequently passed to the channel equalizer. Assuming that synchronization has already been established and the CP is sufficiently long to eliminate the ISI, only a one-tap complex-valued multiplier is required to compensate for the channel distortion over each subcarrier, which will be further described in Subsection 2.2.3. To better understand this fundamental property of OFDM, however, we need to introduce the mathematical model of the communication scheme depicted in Fig. 2.1.

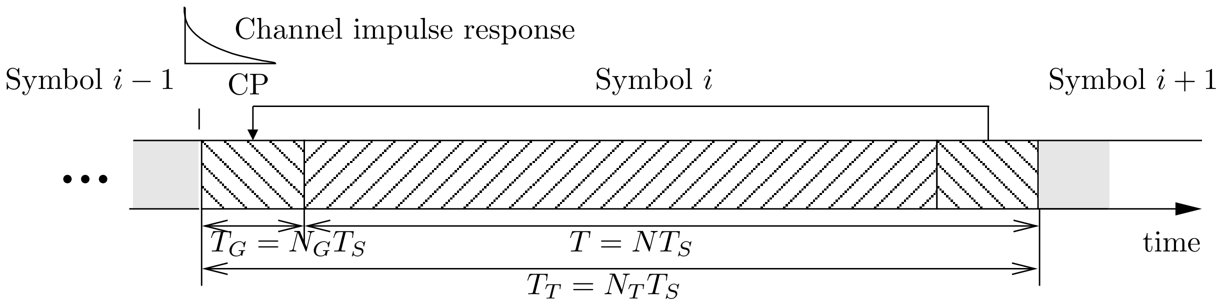


Figure 2.2: Structure of an OFDM symbol

2.1 Discrete-time OFDM System Model

Since OFDM is a block based communication model, a serial data stream is converted into parallel blocks of size N and IDFT is applied to obtain time-domain OFDM symbols. Complex data symbols $C_i(n)$, for $n = 0, \ldots, N-1$, within the ith OFDM symbol are taken from either a Phase Shift Keying (PSK) or Quadrature Amplitude Modulation (QAM) constellation. Then, time domain representation of the ith OFDM symbol after IDFT and CP insertion is given by

$$c_i(k) = \begin{cases} \sum_{n=0}^{N-1} C_i(n) e^{j2\pi kn/N}, & -N_G \leq k \leq N-1 \\ 0, & \text{else} \end{cases}, \tag{2.1}$$

where N_G is the length of the CP which is an important design parameter of the OFDM system that defines the maximum acceptable length of channel impulse response. Furthermore, the transmitted signal can be obtained by concatenating OFDM symbols in time domain as

$$c(k) = \sum_i c_i(k - iN_T). \tag{2.2}$$

In wireless communication systems transmitted signals are typically reflected, diffracted, and scattered, arriving at the receiver along multiple paths with different delays, amplitudes, and phases as illustrated in Fig 2.3. This leads to an overlapping of different copies of the same signal on the receiver side differing in their amplitude, time of arrival and phase. A common model to describe the wireless channel makes use of the channel impulse response, written as $h(l) = \alpha(l)e^{j\theta(l)}$, for $l = 0, \ldots, L-1$, where L presents the total number of received signal paths, while $\alpha(l)$ and $\theta(l)$ are attenuation and phase shift of the lth path, respectively. The differences in the time of arrival are eliminated by the cyclic prefix which is described in the next section. In addition to multipath effects, additive noise is introduced to the transmitted signal. The main sources of additive noise are thermal background noise, electrical noise in the receiver amplifiers, and interference [LS06]. The noise decreases the signal-to-noise ratio (SNR) of the received signal, resulting in a decreased performance. The total effective noise at the receiver of an OFDM system can be modeled as AWGN with a uniform spectral density and zero-mean Gaussian probability distribution. The time domain noise samples are represented by $w(k) \sim SCN(0, \sigma_w^2)$, where σ_w^2 denotes the noise variance. Therefore, the discrete-time model of received OFDM signals can be written as

$$y(k) = \sum_{l=0}^{L-1} h(l)c(k-l) + w(k). \tag{2.3}$$

Multipath propagation and additive noise affect the signal significantly, corrupting the signal and often placing limitations on the performance of the system.

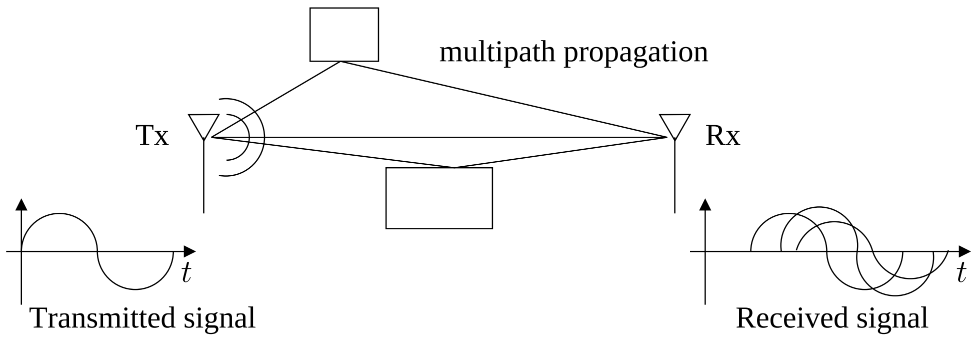


Figure 2.3: The basic principle of multipath propagation

2.2 OFDM System Impairments

Since timing and frequency errors in multi-carrier systems destroy orthogonality among subcarriers which results in large performance degradations, synchronization of time and frequency plays a major role in the design of a digital communication system. Essentially, this function aims at retrieving some reference parameters from the received signal that are necessary for reliable data detection. In an OFDM system, the following synchronization tasks can be identified [PMK07]:

- *sampling clock synchronization*: in practical systems the sampling clock frequency at the receiver is slightly different from the corresponding frequency at the transmitter. This produces intercarrier interference (ICI) at the output of the receiver's DFT with a corresponding degradation of the system performance. The purpose of a sampling clock synchronization is to limit this impairment to a tolerable level.
- *timing synchronization*: the goal of this operation is to identify the starting point of each received OFDM symbol in order to find the correct position of the DFT window. In burst-mode transmissions timing synchronization is also used to locate the start of the frame (frame synchronization) which is a collection of OFDM symbols.
- *frequency synchronization*: a frequency error between the local oscillators at the transmitter and receiver results in a loss of orthogonality among subcarriers with ensuing limitations of the system performance. Frequency synchronization aims at restoring orthogonality by compensating for any frequency offset caused by oscillator inaccuracies.

The block diagram of the receiver is depicted in Fig. 2.4. In the analog frontend, the incoming waveform $r_{RF}(t)$ is filtered and down-converted to baseband using two quadrature sinusoids generated by a local oscillator (LO). The baseband signal is then passed to the analog-to-digital converter (ADC), where it is sampled with frequency $f_s = 1/T_s$. Due to Doppler shifts and/or oscillator instabilities, the frequency f_{LO} of the DFT is not exactly equal to the received carrier frequency f_c. The difference $f_d = f_c - f_{LO}$ is referred to as carrier frequency offset (CFO), or shorter frequency offset, causing a phase shift of $2\pi k f_d$. Therefore, the received baseband signal can be expressed as

$$r(k) = y(k)e^{j2\pi\varepsilon k/N} \tag{2.4}$$

where

$$\varepsilon = N f_d T_s \tag{2.5}$$

is the frequency offset normalized to subcarrier spacing $\Delta f = 1/(NT_s)$. In addition, since the time scales at the transmitter and the receiver are not perfectly aligned, at the start-up the receiver does not know where the OFDM symbols start and, accordingly, the DFT window will be placed in a wrong position. As it will be shown later, since small (fractional) timing errors do not produce any degradation of the system performance, it suffices to estimate the beginning of each received OFDM symbol within one sampling period. Let Δk denotes the number of samples by which the receive time scale is shifted from its ideal

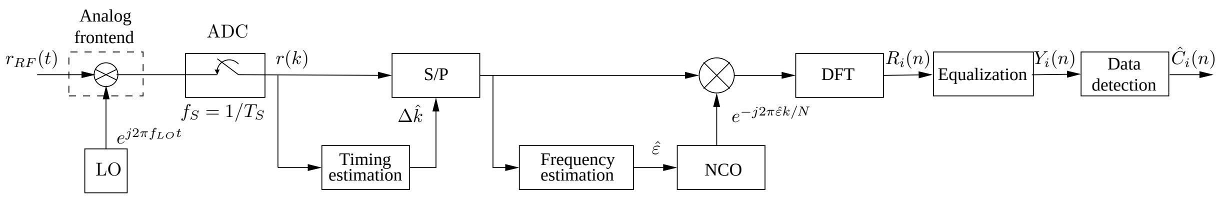


Figure 2.4: Block diagram of a basic OFDM receiver

setting. The samples from ADC are thus expressed by

$$r(k) = e^{j2\pi\varepsilon k/N} y(k - \Delta k) + w(k). \tag{2.6}$$

Replacing (2.2) and (2.3) in (2.6), samples are given as

$$r(k) = e^{j2\pi\varepsilon k/N} \sum_{i} \sum_{l=0}^{L-1} h(l) c_i(k - l - \Delta k - iN_T) + w(k). \tag{2.7}$$

The frequency and timing synchronization units shown in Fig. 2.4 employ the received samples $r(k)$ to compute estimates of ε and Δk, noted as $\hat{\varepsilon}$ and $\hat{\Delta k}$. The former is used to counter-rotate $r(k)$ at an angular speed $2\pi\hat{\varepsilon}k/N$ (frequency correction) using numerically controlled oscillator (NCO), while the timing estimate is exploited to achieve the correct position of the received signal within the DFT window (timing correction). Specifically, the samples $r(k)$ with indices $iN_T + \Delta k \leq k \leq iN_T + \Delta k + N - 1$ are fed to the DFT device and the corresponding output is used to detect the data symbols conveyed by the ith OFDM block.

2.2.1 Effects of Frequency Offset

In order to assess the impact of a frequency error on the system performance, we assume ideal timing synchronization and let $\Delta k = 0$ and $N_g \geq L - 1$. At the receiver, the DFT output for the ith OFDM symbol is computed as

$$R_i(n) = \frac{1}{N} \sum_{k=0}^{N-1} r(k + iN_T) e^{-j2\pi kn/N}, \; 0 \leq n \leq N - 1 \tag{2.8}$$

Substituting (2.7) into (2.8) we get

$$\begin{aligned} R_i(n) &= \frac{1}{N} \sum_{k=0}^{N-1} \left[e^{j2\pi\varepsilon(k+iN_T)/N} \sum_{l=0}^{L-1} h(l) c_i(k - l) + w(k) \right] e^{-j2\pi kn/N} \\ &= \frac{1}{N} e^{j\varphi_i} \sum_{k=0}^{N-1} e^{j2\pi k(\varepsilon - n)/N} \sum_{l=0}^{L-1} h(l) \sum_{m=0}^{N-1} C_i(m) e^{j2\pi(k-l)m/N} + W_i(n) \\ &= \frac{1}{N} e^{j\varphi_i} \sum_{m=0}^{N-1} \left\{ \sum_{l=0}^{L-1} h(l) e^{j2\pi lm/N} \right\} C_i(m) \sum_{k=0}^{N-1} e^{-j2\pi k(m+\varepsilon-n)/N} + W_i(n) \\ &= \frac{1}{N} e^{j\varphi_i} \sum_{m=0}^{N-1} H(m) C_i(m) \sum_{k=0}^{N-1} e^{j2\pi k(m+\varepsilon-n)/N} + W_i(n) \end{aligned} \tag{2.9}$$

where $\varphi_i = 2\pi i\varepsilon N_T/N$, $W_i(n)$ is Gaussian distributed thermal noise with variance σ_w^2 derived as

$$W_i(n) = \frac{1}{N} \sum_{k=0}^{N-1} w(k) e^{-j2\pi kn/N}, \; 0 \leq n \leq N - 1 \tag{2.10}$$

and $H(m)$ is channel frequency response defined as DFT of channel impulse response given as

$$H(m) = \sum_{l=0}^{L-1} h(l) e^{-j2\pi lm/N}, \ 0 \leq m \leq N-1. \tag{2.11}$$

Performing standard mathematical manipulations (2.9) is derived to

$$R_i(n) = e^{j\varphi_i} \sum_{m=0}^{N-1} H(m) C_i(m) f_N(\varepsilon + m - n) e^{j\pi(N-1)(\varepsilon+m-n)/N} + W_i(n), \tag{2.12}$$

where

$$\begin{aligned} f_N(x) &= \frac{\sin(\pi x)}{N \sin(\pi x/N)} \\ &\approx \frac{\sin(\pi x)}{\pi x} \end{aligned} \tag{2.13}$$

can be derived using the standard approximation $sin(t) \approx t$ for small values of argument t.

In the case when the frequency offset is a multiple of subcarrier spacing Δf, i.e., ε is integer-valued, (2.12) reduces to

$$R_i(n) = e^{j\varphi_i} H(|n-\varepsilon|_N) C_i(|n-\varepsilon|_N) + W_i(n), \tag{2.14}$$

where $|n-\varepsilon|_N$ is the value of $n-\varepsilon$ reduced to interval $[0, N-1)$. This equation indicates that **an integer frequency offset does not destroy orthogonality among subcarriers and only results into a shift of the subcarrier indices by a quantity** ε. In this case the nth DFT output is an attenuated and phase-rotated version of $C_i(|n-\varepsilon|_N)$ rather than of $C_i(n)$. Otherwise, when ε is not integer-valued the subcarriers are no longer orthogonal and ICI does occur. In this case it is convenient to rewrite (2.12) like

$$R_i(n) = e^{j[\varphi_i + \pi\varepsilon(N-1)/N]} H(n) C_i(n) f_N(\varepsilon) + I_i(n, \varepsilon) + W_i(n), \tag{2.15}$$

where $I_i(n, \varepsilon)$ accounts for ICI and is given as

$$I_i(n, \varepsilon) = e^{j\varphi_i} \sum_{m=0, m\neq n}^{N-1} H(m) C_i(m) f_N(\varepsilon + m - n) e^{j\pi(N-1)(\varepsilon+m-n)/N}. \tag{2.16}$$

From (2.15) it follows that non-integer normalized frequency offset ε influences the received signal on nth subcarrier twofold. Firstly, received signals on all subcarriers are **equally** attenuated by $f_n^2(\varepsilon)$ and phase shifted by $(\varphi_i + \pi\varepsilon(N-1)/N)$, while the second addend in (2.15) presents interference from other subcarriers (ICI).

Letting $E\left\{|H(n)|^2\right\} = 1$ and assuming independent and identically distributed data symbols with zero mean and power $S = E\left\{|C_i(n)|^2\right\} = 1$, the interference term $I_i(n, \varepsilon)$ can reasonably be modeled as a Gaussian zero-mean random variable with variance (power) defined as

$$\sigma_i^2(\varepsilon) = E\left\{|I_i(n)|^2\right\} = S \sum_{\substack{m=0 \\ m\neq n}}^{N-1} f_N^2(\varepsilon + m - n). \tag{2.17}$$

Under assumption that all subcarriers are used and by means of the identity

$$\sum_{m=0}^{N-1} f_N^2(\varepsilon + m - n) = 1, \tag{2.18}$$

which holds true independently of ε, interference power (2.17) can be written as

$$\sigma_i^2(\varepsilon) = S\left[1 - f_N^2(\varepsilon)\right]. \tag{2.19}$$

A useful indicator to evaluate the effect of frequency offset on the system performance is the loss in SNR, which is defined as

$$\gamma(\varepsilon) = \frac{SNR^{(ideal)}}{SNR^{(real)}}, \tag{2.20}$$

where $SNR^{(ideal)}$ is the SNR of a perfectly synchronized system given as

$$SNR^{(ideal)} = S/\sigma_w^2 = E_S/N_0, \tag{2.21}$$

where E_S is the average received energy over each subcarrier while $N_0/2$ is the two-sided power spectral density of the ambient noise, while

$$SNR^{(real)} = Sf_N^2(\varepsilon)/\left[\sigma_w^2 + \sigma_i^2(\varepsilon)\right], \tag{2.22}$$

is the SNR in the presence of a frequency offset ε. Substituting (2.21) and (2.22) into (2.20), it becomes

$$\gamma(\varepsilon) = \frac{1}{f_N^2(\varepsilon)}\left[1 + \frac{E_S}{N_0}(1 - f_N^2(\varepsilon))\right]. \tag{2.23}$$

For small values of ε (2.23) can be simplified using the Taylor series expansion of $f_N^2(\varepsilon)$ around $\varepsilon = 0$, resulting in

$$\gamma(\varepsilon) = 1 + \frac{1}{3}\frac{E_S}{N_0}(\pi\varepsilon)^2. \tag{2.24}$$

It can be seen that the SNR loss is approximately proportional to the square of the normalized frequency offset ε.

2.2.2 Effects of Timing Offset

In order to assess the performance of the OFDM system in the presence of small time offset let assume perfect frequency synchronization, i.e., $\varepsilon = 0$ and consider only the effect of time offset when it is smaller than the uncorrupted part of the cyclic prefix, i.e., $\Delta k \leq N_g - (L+1)$.

Under these assumptions, (2.8) is derived to

$$\begin{aligned}
R_i(n) &= \frac{1}{N}\sum_{k=0}^{N-1}\left[\sum_{l=0}^{L-1} h(l)c_i(k-l-\Delta k) + w_i(k)\right]e^{-j2\pi kn/N} \\
&= \frac{1}{N}\sum_{k=0}^{N-1}\sum_{l=0}^{L-1} h(l)\sum_{m=0}^{N-1} C_i(m)e^{j2\pi(k-l-\Delta k)m/N}e^{-j2\pi kn/N} + W_i(n) \\
&= \frac{1}{N}\sum_{m=0}^{N-1}\left[\sum_{l=0}^{L-1} h(l)e^{-j2\pi lm/N}\right]C_i(m)\left[\sum_{k=0}^{N-1} e^{-j2\pi k(m-n)/N}\right]e^{-j2\pi\Delta km/N} + W_i(n) \\
&= \sum_{m=0}^{N-1} H(m)C_i(m)\delta(m-n)e^{-j2\pi\Delta km/N} + W_i(n) \\
&= H(n)C_i(n)e^{-j2\pi\Delta km/N} + W_i(n),
\end{aligned} \tag{2.25}$$

where $W_i(n)$ and $H(m)$ are defined in (2.10) and (2.11), respectively, while $\delta(m-n)$ presents the Kronecker delta defined as

$$\delta(m-n) = \begin{cases} 1, & m = n \\ 0, & m \neq n. \end{cases}. \tag{2.26}$$

From expression above it can be seen that small time offsets causes only a linear phase rotation across the DFT outputs and can be compensated by the channel equalizer, which can not distinguish between phase shifts introduced by the channel and those derived from the time offset. Timing offset does not destroy the orthogonality of the carriers and the effect of timing error is a phase rotation which linearly changes with subcarrier order.

2.2.3 Equalization

Channel equalization is the process through which a coherent receiver compensates for any distortion induced by frequency-selective fading. For the sake of simplicity, ideal timing and frequency synchronization is considered throughout this subsection. The channel is assumed static over each OFDM block, but can vary from block to block. Under these assumptions, and assuming that the receiver is perfectly synchronized, i.e, $\varepsilon = 0$ and $\Delta k = 0$, the output of the receiver's DFT unit during the ith symbol is given by

$$R_i(n) = H_i(n)C_i(n) + W_i(n),\ 0 \leq n \leq N-1 \tag{2.27}$$

where $C_i(n)$ is the complex data symbol and $W_i(n)$ as well as $H(m)$ are defined in (2.10) and (2.11), respectively. An important feature of OFDM is that channel equalization can independently be performed over each subcarrier by means of a bank of one-tap multipliers. As shown in Fig. 2.5, the nth DFT output $R_i(n)$ is weighted by a complex-valued coefficient $P_i(n)$ in order to compensate for the channel-induced attenuation and phase rotation. The equalized sample $Y_i(n) = P_i(n)R_i(n)$ is then subsequently passed to the detection unit, which delivers final decisions $\hat{C}_i(n)$ on the transmitted data. Intuitively, the simplest

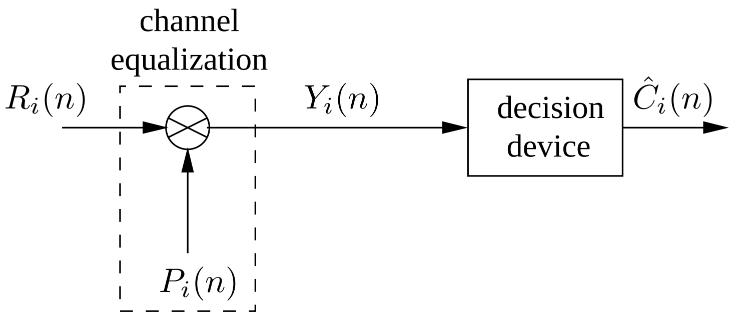


Figure 2.5: Block diagram of an OFDM receiver

method for the design of the equalizer coefficients, is to perform a pure channel inversion, know as Zero-Forcing (ZF) criterion. The equalizer coefficients are then given by

$$P_i(n) = \frac{1}{H_i(n)}, \tag{2.28}$$

while the DFT output takes the form

$$Y_i(n) = \frac{R_i(n)}{H_i(n)} = C_i(n) + \frac{W_i(n)}{H_i(n)},\ 0 \leq n \leq N-1. \tag{2.29}$$

From (2.29) it can be noticed that ZF equalization is capable of totally compensating for any distortion induced by the wireless channel. However, the noise power at the equalizer output is given by $\sigma_w^2/|H_i(n)|^2$ and may be excessively large over deeply faded subcarriers characterized by low channel gains.

Inherent system requirement for ZF equalizer is the knowledge of the channel transfer function $H_i(n)$. Therefore, in many wireless OFDM systems, sequence of data symbols is preceded by several reference OFDM symbols (preambles) known to the receiver, forming the **OFDM frame**. Typical frame structure is shown in Fig. 2.6 where preambles are typically used for **synchronization** and/or **channel estimation** purposes. In typical fixed wireless standards as WLAN, it can be assumed that the channel remains static over frame duration, i.e., $H_i(n) = H(n)$ for $i = 1, \dots, I$, where I is the total number of OFDM symbols within one frame. Then, channel estimates obtained from the preambles can be used to coherently detect the entire payload.

Assuming that the OFDM frame has one preamble with index $i = p = 1$, the output of the DFT block (2.27) can be written as

$$R_p(n) = H(n)C_p(n) + W_p(n),\ 0 \leq n \leq N-1 \tag{2.30}$$

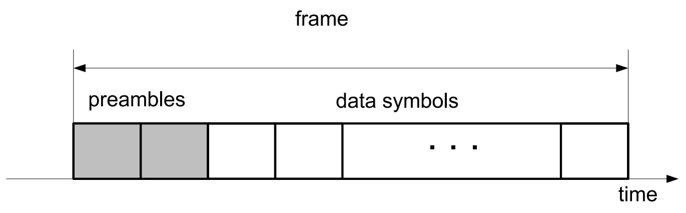


Figure 2.6: Frame structure

where $C_p(n)$ are complex data symbols **known** to the receiver. Then, estimates of the channel frequency response $\hat{H}(n)$ can be obtained as

$$\hat{H}(n) = \frac{R_p(n)}{C_p(n)} = H(n) + \frac{W_p(n)}{C_p(n)},\ 0 \leq n \leq N-1. \tag{2.31}$$

On the other hand, in applications characterized by relatively high mobility as those envisioned by the Long Term Evaluation (LTE) standard, the channel response undergoes significant variations over one frame and must continuously be tracked to maintain reliable data detection. In this case, in addition to initial reference blocks, known symbols called pilots are normally inserted into the payload section of the frame at some convenient positions. These pilots are scattered in both time and frequency directions (i.e., they are positioned over different blocks and different subcarriers), and are used as reference values for channel estimation and tracking.

In order to assess and compare the influence of system impairments on different data rates supported in OFDM systems, a short survey of commonly used coherent modulation techniques and their performance evaluation in AWGN channel are given is the next section.

2.3 Digital Modulations Used in OFDM Systems

Consider some digital information that is given by a finite bit sequence. To transmit this information over a physical, analog channel by a passband signal we need a mapping rule between the set of bit sequences and the set of possible signals or constellation points on the complex plane, as shown in Fig. 2.7. Such a mapping rule is called a digital modulation scheme. A linear digital modulation scheme is characterized by the complex baseband signal [Rou08]

$$C(t) = \sum_i C_i g(t-kT), \tag{2.32}$$

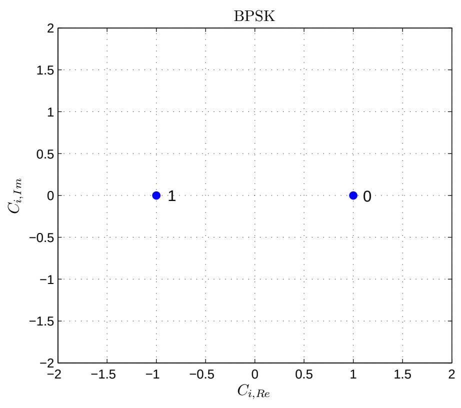
where C_i is a given constellation point and $g(t)$ is a pulse shape used for transmission. Since mapping is usually performed in digital domain we will keep discrete domain representation of modulated complex symbols for further simplification. In the following we will resume some of the coherent modulation schemes typically used in OFDM systems.

2.3.1 Phase Shift Keying (PSK)

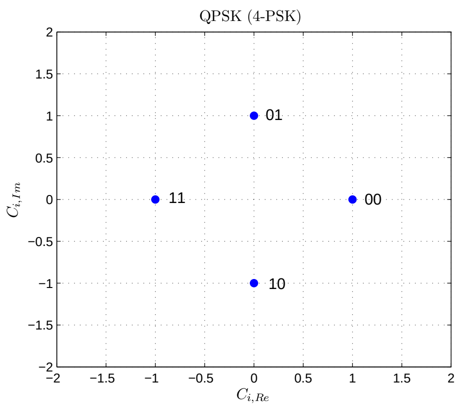
PSK or Multiple PSK (M-PSK) modulation, where M is the number of constellation points, is characterized that all signal information is put into the phase of the transmitted signal, preserving constant envelope property. The M-PSK complex symbol C_i can be written as

$$C_i = \sqrt{S} e^{j(\frac{2\pi m}{M}+\theta_0)},\ m = 0, 1, \ldots, M-1, \tag{2.33}$$

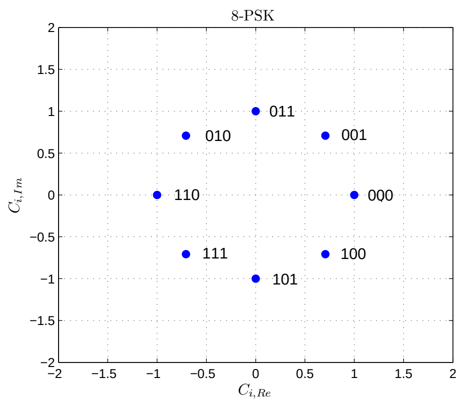
where S is the average signal power and θ_0 is an arbitrary constant phase. Constellation diagrams for $M = 2, 4, 8$, i.e., Binary Phase Shift Keying (BPSK), Quadrature Phase Shift Keying (QPSK) or 4-PSK and 8-PSK, respectively, when $\theta_0 = 0$, are shown in Fig. 2.7. The simplest PSK modulation format is



(a) BPSK constellation diagram



(b) QPSK(4-PSK) constellation diagram



(c) 8-PSK constellation diagram

Figure 2.7: M-PSK constellation diagrams

BPSK, where a logical „1“ is encoded as 0 phase, and a logical „0“ is coded as a phase of π. Then, the modulated symbol, defined in (2.33), can be written as

$$C_i = \pm\sqrt{S} \tag{2.34}$$

with constellation diagram shown in Fig. 2.7(a). MPSK constellation diagram for 4-PSK (2 bits mapped into $4 = 2^2$ phases) and 8-PSK (3 bits mapped into $8 = 2^3$ phases), are shown in Fig. 2.7(b) and Fig. 2.7(c), respectively.

The bit error rate (BER) is defined as the ratio between the number of successfully received to the number of total transmitted information bits and is usually taken as a measure of modulation quality. For BPSK in AWGN it is given as [Gol05]

$$p_{b,BPSK} = Q\left(\sqrt{\frac{2E_b}{N_0}}\right), \tag{2.35}$$

where $\frac{E_b}{N_0}$ is the SNR per bit and $Q(x)$ is defined as

$$Q(x) = \frac{1}{2}\,\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right), \tag{2.36}$$

where

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-y^2}dy \tag{2.37}$$

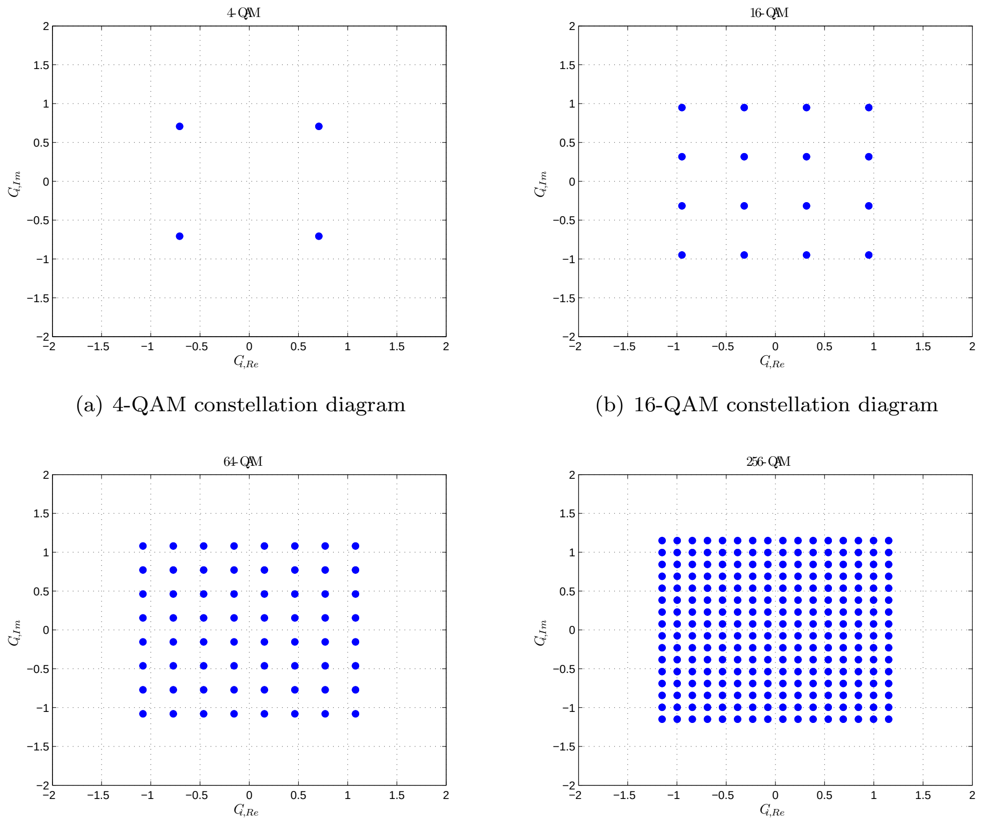
is the complementary error function (erfc). For higher order M-PSK, where $M > 4$, the symbol error rate (SER) can be expressed as

$$p_{s,M-PSK} = 2Q\left(\sqrt{\frac{2E_b \log_2 M}{N_0}} \sin\frac{\pi}{M}\right), \tag{2.38}$$

where $\frac{E_s}{N_0} = \frac{E_b \log_2 M}{N_0}$ is the SNR per symbol. For Gray-coded modulations, i.e., when adjacent constellation points differ in one bit as in Fig. 2.7, the BER in the high SNR regime for each modulation is approximately

$$p_{b,M-PSK} \approx \frac{p_{s,M-PSK}}{\log_2 M}.$$

2.3.2 Quadrature Amplitude Modulation (QAM)



(a) 4-QAM constellation diagram (b) 16-QAM constellation diagram

(c) 64-QAM constellation diagram (d) 256-QAM constellation diagram

Figure 2.8: QAM constellation diagrams

QAM is a bandwidth efficient signaling scheme that, unlike M-PSK does not possess a constant envelope property, thus offering higher bandwidth efficiency, i.e., more bits per second (bps) can be transmitted in a given frequency bandwidth. QAM modulated signals for M constellation points can be written as

$$C_i = \sqrt{S}K(X_i + jY_i),$$

where $X_i, Y_i \in \left\{\pm 1, \pm 3, \ldots, \sqrt{M} - 1\right\}$ and K is a scaling factor for normalizing the average power for all constellations to S. The K value for various constellations is shown in Table 2.1. Corresponding QAM constellation diagrams for 4-QAM (2 bits mapped into $4 = 2^2$ points), 16-QAM (4 bits mapped into $16 = 2^4$ points), 64-QAM (6 bits mapped into $64 = 2^6$ points), and 256-QAM (8 bits mapped into

$16 = 2^8$ points), are shown in Fig. 2.8. It can be noticed that 4-QAM corresponds to QPSK with constant phase shift $\theta_0 = \pi/4$. The symbol error rate (SER) for QAM modulations can be expressed as

Modulation	Number of bits m	K
4-QAM	2	$1/\sqrt{2}$
16-QAM	4	$1/\sqrt{10}$
64-QAM	6	$1/\sqrt{42}$
256-QAM	8	$1/\sqrt{170}$

Table 2.1: Modulation dependent parameters

$$p_{s,M-QAM} = 1 - \left(1 - 2\left(1 - \frac{1}{\sqrt{M}}\right) Q\left(\sqrt{3\frac{E_b \log_2 M}{(M-1)N_0}}\right)\right)^2 . \tag{2.39}$$

where $Q(x)$ is defined in (2.36). For Gray-coded modulations the BER in the high SNR regime for each modulation is, as for M-PSK case, approximately

$$p_{b,M-QAM} \approx \frac{p_{s,M-QAM}}{\log_2 M}.$$

Quadrature Amplitude Modulation (QAM) schemes like 4-QAM (QPSK), 16-QAM and 64-QAM are used in typical wireless digital communications specifications like Wireless Local Access Network (WLAN) and Worldwide Interoperability for Microwave Access (WiMAX) [Erg09].

3 Software Defined Radio and GNU Radio Framework

In this chapter a general introduction to the SDR concept is given. Additionally, advantages of SDRs and given hardware limitations are addressed. Here, the GNU Radio SDR framework is presented giving insight into the basic architectural features.

A SDR is a radio that is built entirely or in large parts in software, which runs on a general purpose computer. A more extensive definition is given by Joseph Mitola, who established the term Software Radio [Mit06]:

„A software radio is a radio whose channel modulation waveforms are defined in software. That is, waveforms are generated as sampled digital signals, converted from digital to analog via a wideband digital-to-analog converter (DAC) and then possibly upconverted from intermediate frequency (IF) to RF. The receiver, similarly, employs a wideband ADC that captures all of the channels of the software radio node. The receiver then extracts, downconverts and demodulates the channel waveform using software on a general purpose processor. Software radios employ a combination of techniques that include multi-band antennas and RF conversion; wideband ADC and DAC; and the implementation of IF, baseband and bitstream processing functions in general purpose programmable processors. The resulting software defined radio (or „software radio“) in part extends the evolution of programmable hardware, increasing flexibility via increased programmability.“

This means, that instead of using analog circuits or a specialized Digital Signal Processor (DSP) to process radio signals, the digitized signals are processed by architecture independent, and high level software running on general purpose processors. The term radio designates any device, that transmits and/or receives radio waves. While most modern radios contain firmware that is written in some kind of programming language, the important distinction in a software radio is that it is not tailored to a specific chip or platform, and it is therefore possible to reuse its code across different underlying architectures [Mue08].

3.1 Ideal Software Defined Radio and Practical Limitations

In the ideal case, the only hardware that is needed besides a computer is an antenna and an ADC for the receiver, as well as a digital-to-analog converter (DAC) for the transmitter. A SDR would thus look as depicted in Fig. 3.1. In the receiver, a transmitted radio signal is picked up by an antenna, and then fed into an ADC to sample it. Once digitized, the signal is sent to some general purpose computer (e.g. an embedded PC) for processing. The transmitter looks very similar, except that the signal is sent in the reverse direction, and a DAC is used instead of an ADC. In a complete transceiver, the processing unit and the antenna may be shared between receiver and transceiver.

While the approach presented in the previous section is very simple and (in the ideal case) extremely versatile, it is not practical, due to limitations in real hardware. However, various solutions have been suggested to overcome these problems. A quick look at the different hardware limitations is given below. For better readability, only the receiving side is discussed. The transmitting side is symmetrical.

- *Analog-Digital Converters*: According to sampling theorem, the sampling rate of ADC must be at least twice as high as the bandwidth of received signal which limits the maximum bandwidth of the received signal. Current ADCs are capable of sampling rates in the area of 100 Mega Samples Per Second (MSPS), which translates to a bandwidth of 50 MHz. While this bandwidth is enough for most current applications, the carrier frequency is usually higher than 50 MHz. In practice, a RF

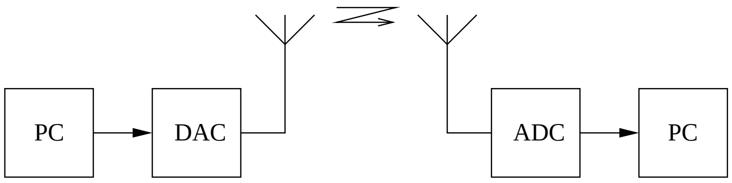


Figure 3.1: Ideal SDR transmission

frontend is therefore usually required, to convert the received signal to an intermediate frequency (IF).

The second parameter, the ADC resolution influences the dynamic range of the receiver. As each additional bit doubles the resolution of the sampled input voltage, the dynamic range can be roughly estimated as $R = 6dB \times n$ where R is the dynamic range and n the number of bits in the ADC. As ADCs used for SDR usually have a resolution of less than 16 bits, it is important to filter out strong interfering signals, such as signals from mobile phones, before the wideband ADC. This is usually done in the RF frontend.

- *Bus Speed*: Another problem lies in getting the data from the ADC to the computer. For any practical bus, there is a maximum for the possible data rate, limiting the product of sample rate and resolution of the samples. The speed of common buses in commodity PCs ranges from a few Mbps to several Gbps as an example, the Peripheral Component Interconnect (PCI) 2.2 bus has a theoretical maximum speed of 4256 Mbps.
- *Performance of the Processing Unit*: For real-time processing, the performance of the Central Processing Unit (CPU)and the sample rate limit the number of mathematical operations that can be performed per sample, as samples must be processed as fast as they arrive. In practice, this means that fast CPUs, clever programming and possibly parallelization is needed. If this does not suffice, a compromise must be found, to use a less optimal but faster signal processing algorithm.
- *Latency*: Since general purpose computers are not designed for real-time applications, a rather high latency can occur in practical SDRs. While latency is not much of an issue in transmit-only or receive-only applications, many wireless standards, such as Global System for Mobile communications (GSM) or Wireless Metropolitan Access Network (DECT) require precise timing, and are therefore very difficult to implement in an SDR.

Because of the use of general purpose processing units, an implementation of a given wireless application as an SDR is likely to use more power and occupy more space than a hardware radio with analog filtering and possibly a dedicated signal processor. Because an SDR contains more complex components than a hardware radio, it will likely be more expensive, given a large enough production volume.

Nevertheless, SDR concepts carry the flexibility of software over to the radio world and introduces a number of interesting possibilities. For example, very much the same way as someone may load a word processor or an Internet browser on a PC, depending on the task at hand, a SDR could allow its user to load a different configuration, depending on whether the user wants to listen to a broadcast radio transmission, place a phone call or determine the position via Global Positioning System (GPS). A new application may even be added after the device is finished. Since the same hardware can be used for any application, a great reuse of resources is possible. Another interesting possibility enabled by SDR is the creation of a cognitive radio, which is aware of its RF environment and adapts itself to changes in the environment. By doing this, a cognitive radio can use both the RF spectrum and its own energy resources more efficiently. As a cognitive radio requires a very high degree of flexibility, the concept of SDR is very convenient for its practical realization.

3.2 GNU Radio Architecture

GNU Radio is an open source, free software toolkit for building SDRs [GNUac]. It is designed to run on personal computers and (PC) combined with minimal hardware allowing the construction of simple software radios [Mue08]. The project was started in early 2000 by Eric Blossom and has evolved into a mature software infrastructure that is used by a large community of developers. It is licensed under the GNU General Public License (GPL), thus anyone is allowed to use, copy and modify GNU Radio without limits, provided that extensions are made available under the same license. While GNU Radio was initially started on a Linux platform, it now supports various Windows, MAC and various Unix platforms.

GNU Radio architecture consists of two components. The first component is the set of numerous building blocks which represents C++ implementations of digital signal processing routines such as (de)modulation, filtering, (de)coding and I/O operations such as file access. For further information about C++ programming see for example [Mey98, Mey99, Mey01, Str00, SA07]. The second component is a framework to control the data flow among blocks, implemented as Python scripts enabling easy reconfiguration and control of various system functionalities and parameters. For further studies on Python see e.g. [Mar03]. By „wiring“ together such building blocks, a user can create a software defined radio, similar to connecting physical RF building blocks to create a hardware radio. An RF interface for GNU Radio architecture is realized by USRP boards, a general purpose RF hardware, which performs computationally intensive operations as filtering, up- and down-conversion. The USRP and its recent version USRP2 are connected to PC over a USB 2.0 and Ethernet cable, respectively, and are controlled through a robust application programming interface (API) provided by GNU Radio.

3.2.1 Gnu Radio Framework

A data flow among different blocks is abstracted by **flowgraph**, a directed acyclic graph in which the vertices are the GNU Radio blocks and the edges corresponds to data **streams**, as shown in Fig. 3.2. Generally, GNU Radio blocks, shown in Fig. 3.3 operate on continuous streams of data. Most blocks

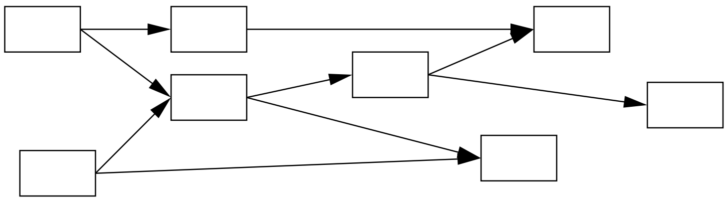


Figure 3.2: An example of **flowgraph**

have a set of input and/or output ports, therefore, they consume data from input streams and generate data for their output streams. Special blocks called sources and sinks only consume or produce data, respectively. Examples of sources and sinks are blocks that read and write, respectively, from USRP receive ports, sockets and file descriptors. Each block has an input and output signature (IO signatures) that defines the minimum and maximum number of input and output streams it can have, as well as size of the data type on corresponding stream. The supported types are

- `c` - complex interleaved floats (8 byte each),
- `f` - floats (4 byte),
- `s` - short integers (2 byte) and
- `b` - byte integers (1 byte).

Each block defines a **general_work()** function that operates on its input to produce output streams. In order to help the scheduler decide when to call the work function, blocks also provide a **forecast()** function that returns the system runtime, the number of input items it requires to produce a number of output items and how many output items it can produce given a number of input items. At runtime, blocks tell the system how many input (output) items they consumed (produced). Blocks may consume data on each input stream at a different rate, but all output streams must produce data at the same rate. The input and output streams of a block have buffers associated with them. Each input stream has a read buffer, from which the block reads data for processing. Similarly, after processing, blocks write data to the appropriate write buffers of its output streams. The data buffers are used to implement the

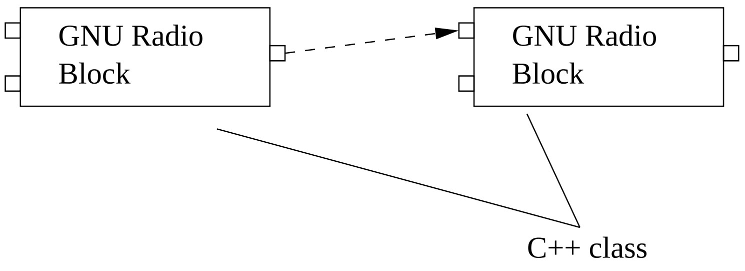


Figure 3.3: GNU Radio blocks

edges in the flowgraph: the input buffers for a block are the output buffers of the upstream block in the flowgraph. GNU Radio buffers are single writer, multiple reader FIFO (First in First Out) buffers. Several blocks are connected in Python forming a flowgraph using the **connect** function which specifies

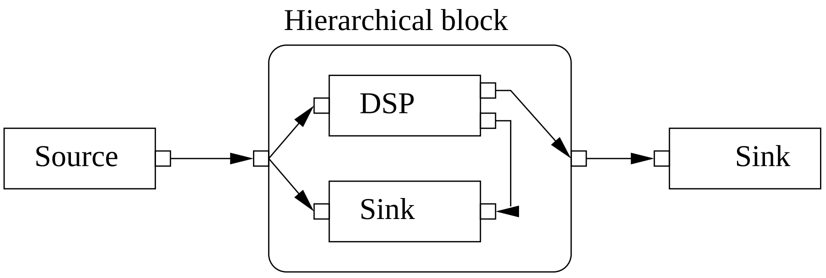


Figure 3.4: An example of a flowgraph with a hierarchical block

how the output stream(s) of a processing block connects to the input stream of one or more downstream blocks. The flowgraph mechanism then automatically builds the flowgraph; the details of this process are hidden from the user. A key function during flowgraph construction is the allocation of data buffers to connect neighboring blocks. The buffer allocation algorithm considers the input and output block sizes used by blocks and the relative rate at which blocks consume and produce items on their input and output streams. Once buffers have been allocated, they are connected with the input and output streams of the appropriate block.

Several blocks can also be combined in a new block, named **hierarchical** block, as shown in Fig. 3.4. **Hierarchical** blocks are implemented in Python and together with other blocks can be combined into new **hierarchical** blocks. Input and output ports of hierarchical blocks have same constraints as those of terminal blocks.

The GNU Radio scheduler executes the graph that was built by the flowgraph mechanism. During the execution, the scheduler queries each block for its input requirements and it uses the above-mentioned forecast functions to determine how much data the block can consume from its available input. If sufficient data is available in the input buffers, the schedule calls the block's work function. If a block does not have sufficient input, the scheduler simply moves on to the next block in the graph. Skipped blocks will be executed later, when more input data is available. The scheduler is designed to operate on continuous data streams.

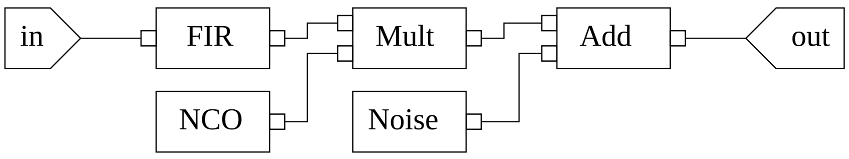


Figure 3.5: Wireless communication channel simulation model

3.2.2 An Example: Wireless Channel Simulation

It will be shown how a model for a static wireless channel can be implemented as a GNU Radio **hierarchical** block. The channel is affected by multipath propagation, frequency offset and additive noise. Fig. 3.5 shows a model with internal blocks and corresponding ports [Aur09].

Multipath effects are modeled using a FIR-filter where complex filter coefficients are taken from an arbitrary channel model, e.g. Rayleigh channel model. The signal from an input port is derived to the corresponding GNU Radio block **gr.fir_filter_ccc**. The suffix ccc denotes that the input stream, output stream and filter coefficients are of complex data types.

According to (2.4), the frequency offset is modeled as a sinus wave with fixed frequency and is multiplied with the incoming signal. The corresponding GNU radio blocks are the complex sine signal source **gr.sig_source_c** and the multiplicator with complex inputs and outputs **gr.multiply_cc**, respectively.

Finally, complex additive Gaussian noise generated by **gr.noise_source_c** is added to the incoming signal in the **gr.add_cc** block and the result is directed to the output port.

The initial parameters of a given hierarchical block, named **simple_channel**, are additive noise standard variance, frequency offset normalized to sampling frequency and complex FIR-filter coefficients. IO signatures of input and output ports are identical and in the framework there is minimum one port and maximum one port for both input and output.

During runtime, internal blocks are initialized and connected to the flowgraph. The corresponding python script is shown below.

Program 1 Python script for simulation of a wireless communication channel

```
class simple_channel(gr.hier_block2):
  def __init__(self, noise_rms, frequency_offset, channel_coefficients):
    gr.hier_block2.__init__(self, "simple_channel", # Blocktype Identifier
        gr.io_signature(1,1,gr.sizeof_gr_complex),  # incoming
        gr.io_signature(1,1,gr.sizeof_gr_complex))  # outgoing

    # for example channel_coefficients = [0.5+0.1j, 0.2-0.01j]
    multipath_sim = gr.fir_filter_ccc(1, channel_coefficients)

    # frequency_offset normalized to sampling frequency
    # amplitude = 1.0, DC offset = 0.0
    offset_src = gr.sig_source_c(1, gr.GR_SIN_WAVE, frequency_offset, 1.0, 0.0)
    mix = gr.multiply_cc()

    # noise_rms -> var(noise) = noise_rms**2
    noise_src = gr.noise_source_c(gr.GR_GAUSSIAN, noise_rms/sqrt(2))
    add_noise = gr.add_cc()

    # describe signal paths
    self.connect(self, multipath_sim)       # incoming port
    self.connect(multipath_sim, (mix,0))
    self.connect(offset_src, (mix,1))
    self.connect(mix, (noise_add,0))
    self.connect(noise_src, (noise_add,1))
    self.connect(noise_add, self)           # outgoing port
```

4 GNU Radio OFDM Transceiver Implementation

This chapter gives an introduction to an SDR based GNU Radio OFDM transceiver implemented in our institute [ZAM09]. A system diagram of our OFDM framework is shown in Fig. 4.1. Transmitter and

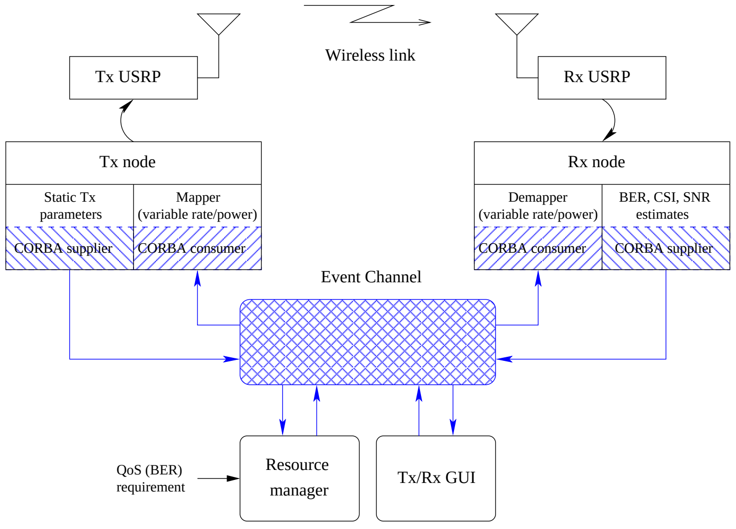


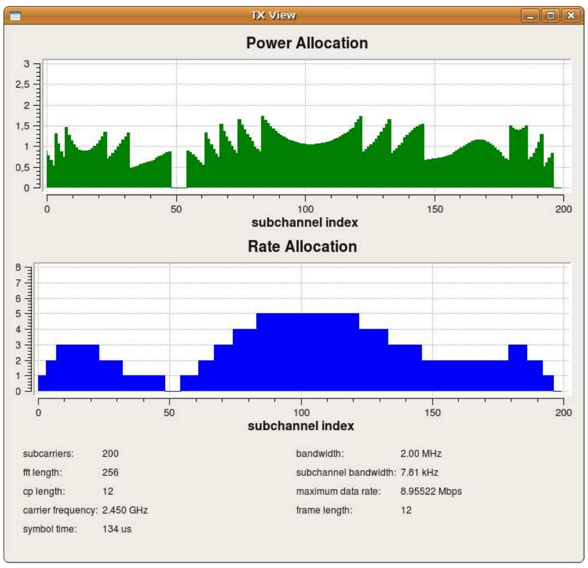
Figure 4.1: The system overview

receiver node are composed of a host commodity computer and general purpose RF hardware, USRP. The baseband signal processing at the host computers is implemented in the GNU Radio framework. Within the framework, additional OFDM specific GNU Radio blocks are implemented, particularly at receiver's synchronization stage, since OFDM systems are highly sensitive to time offsets and oscillators' mismatch between transmitter and receiver due to necessity for subchannel orthogonality, see Section 2.2. The main adaptive and capacity achieving functionality is performed in blocks for adaptive mapping and demapping of various rates (taken from the set of available modulations given in Table 4.1) and power levels across subchannels.

Algorithms for estimation of link quality, expressed through average SNR and channel state information (CSI) over subchannels, are extensively studied and implemented at the receiver. Furthermore, in order to assess system performance, the receiver implementation contains blocks for BER measurements.

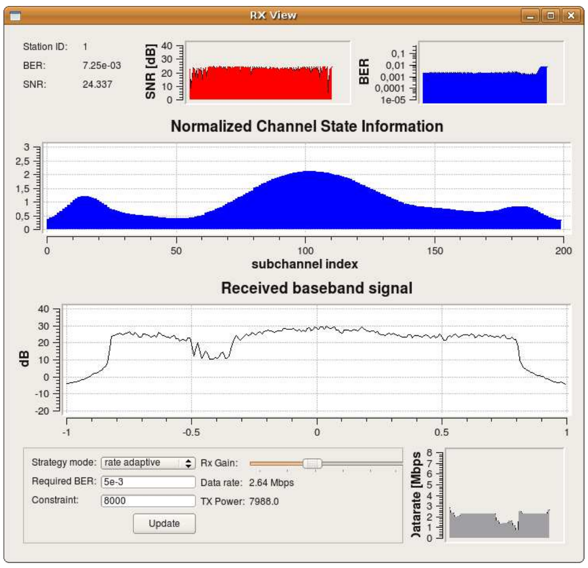
The communication between transmitter and receiver node is organized as a reconfigurable continuous one-way transmission of OFDM symbol frames. As shown in Table 4.1, the set of input configuration parameters can be divided into two classes. The set of **static parameters** containing FFT, number of subchannels, frame size, etc., is initialized at transmission start and is known to both nodes. The set of **dynamic parameters** which are reconfigurable at runtime includes total transmit power and allocated rate and power over subchannels.

The backbone of the system is realized over a local Ethernet network by a Common Object Request Broker Architecture (CORBA) **event service** [GNUrg], a distributed communication model that allows an application to send an event that will be received by any number of objects located in different logical and/or physical entities. For more information on CORBA see [MR97, OHE97]. Estimated parameters that indicate link quality (average SNR, CSI, and BER) and current static transmitter's parameters are



subcarriers: 200
fft length: 256
cp length: 12
carrier frequency: 2.450 GHz
symbol time: 134 us
bandwidth: 2.00 MHz
subchannel bandwidth: 7.81 kHz
maximum data rate: 8.95522 Mbps
frame length: 12

(a) The transmitter's GUI



Station ID: 1
BER: 7.25e-03
SNR: 24.337
Strategy mode: rate adaptive
Rx Gain:
Required BER: 5e-3
Data rate: 2.64 Mbps
Constraint: 8000
TX Power: 7988.0

(b) The receiver's GUI with interactive control interface

Figure 4.2: The framework's GUI with interactive control interface

supplied as CORBA events to the **event channel** which allows other components (consumers) within the system to register their interests in events. The central control unit that determines optimal input transmission parameters for given requirements is called **resource manager**. Controlled by an interactive graphical user interface (GUI) it consumes supplied events forwarded from the **event channel**, performs allocation in an optimal manner, and supplies new transmission parameters, i.e., total transmit power and power/rate per subchannel, which are finally consumed by other components in the system. The

Carrier frequency (**static**)	2400 – 2483 MHz
Bandwidth (**static**)	Variable, up to 2 MHz
FFT length (**static**)	64 – 1024
Frame length (**static**)	Variable
Modulations (**dynamic**)	BPSK, QPSK, 8-PSK, 16-QAM, 32-QAM, 64-QAM, 128-QAM, 256-QAM
Power (**dynamic**)	Up to 20 mW

Table 4.1: OFDM symbol parameters

GUI, facilitating the demonstration, is developed in Qt/C++ framework. The transmitter's GUI contains static transmission parameters and current allocation of rate and power over subchannels, as shown in Fig. 4.2(a). Furthermore, the receiver's GUI, given in Fig. 4.2(b), dynamically shows estimated channel parameters (average SNR, CSI, BER) and contains an interactive interface for controlling allocation strategies and transmission parameters in the **resource manager**.

The system can also run in simulation mode on a single PC, without the RF interface (USRP boards), where transmitter and receiver „communicate" over an artificial channel. A set of channel models available through IT++ libraries [GNUpp] is ported to GNU Radio framework in order to evaluate the system performance excluding hardware impairments.

Due to the high modularity and distributed nature of the system supported by a generalized interface, the dedicated **resource manager** can be easily reconfigured for different classes of given requirements and various sets of controllable parameters.

5 Preparatory Exercises

5.1 Exercise: System Performance in an AWGN Channel

Note: For this exercise SNR values are given in decibel (dB). The relation to the linear value is given by

$$SNR[dB] = 10 \cdot log_{10}(SNR). \tag{5.1}$$

1. Determine the missing BER entries p_b in Table 5.1 for given values of received SNR and various modulation schemes using (2.35) and (2.39).
   Hints*: For calculating the missing entries of Table 5.1 use the* ***erfc*** *function which is a member of the standard Matlab library. Calculated values should be rounded to two digits, e.g., if the calculated value is* 0.001234 *write it as* $1.23 \cdot 10^{-3}$.

Table 5.1: BER p_b to SNR [dB] dependencies for various modulation schemes

E_S/N_0 [dB]	0	5	10	15	20	25	30
BER_{BPSK}							
BER_{4-QAM}							
BER_{16-QAM}							
BER_{64-QAM}							
$BER_{256-QAM}$							

2. Draw graphs for the BER p_b depending on the SNR E_S/N_0 into Fig. 5.1. Sketch the graphs for all modulation schemes given in Table 5.1 using all entries from this table. Give an appropriate legend.

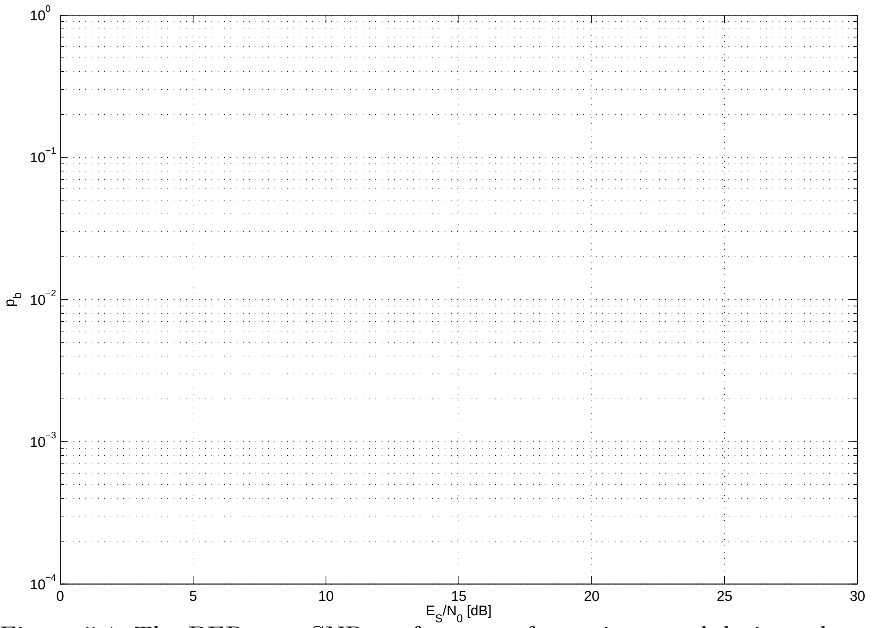


Figure 5.1: The BER over SNR performance for various modulation schemes

3. Using the results from Fig. 5.1, determine the required SNR E_S/N_0 values for achieving a BER of 10^{-3}. What are the differences of the determined SNRs for BPSK, 16-QAM, and 256-QAM?

5.2 Exercise: SNR Loss due to Frequency Offset

1. Determine the missing SNR loss entries and fill in the Table 5.2 for different frequency offsets and SNRs using (2.23).
   Notes: *Calculate SNR losses in decibel (dB). The relation to the linear value is given by*
   $$\gamma(\varepsilon)[dB] = 10 \cdot log_{10}(\gamma(\varepsilon)). \qquad (5.2)$$
   Assume that $\frac{\pi\varepsilon}{N}$ *is small.*

Table 5.2: SNR losses for different values of frequency offsets and SNRs

ε	10^{-2}	$2\cdot10^{-2}$	$5\cdot10^{-2}$	10^{-1}	$2\cdot10^{-1}$	$4\cdot10^{-1}$
$\gamma(\varepsilon)\|_{E_S/N_0=5[dB]}[dB]$						
$\gamma(\varepsilon)\|_{E_S/N_0=10[dB]}[dB]$						
$\gamma(\varepsilon)\|_{E_S/N_0=15[dB]}[dB]$						
$\gamma(\varepsilon)\|_{E_S/N_0=20[dB]}[dB]$						

2. Draw curves for the SNR loss $\gamma(\varepsilon)$ in dependency of the frequency offset ε into Fig. 5.2. Draw the graphs to all SNRs given in Table 5.2 taking all entries from this table. Give an appropriate legend.

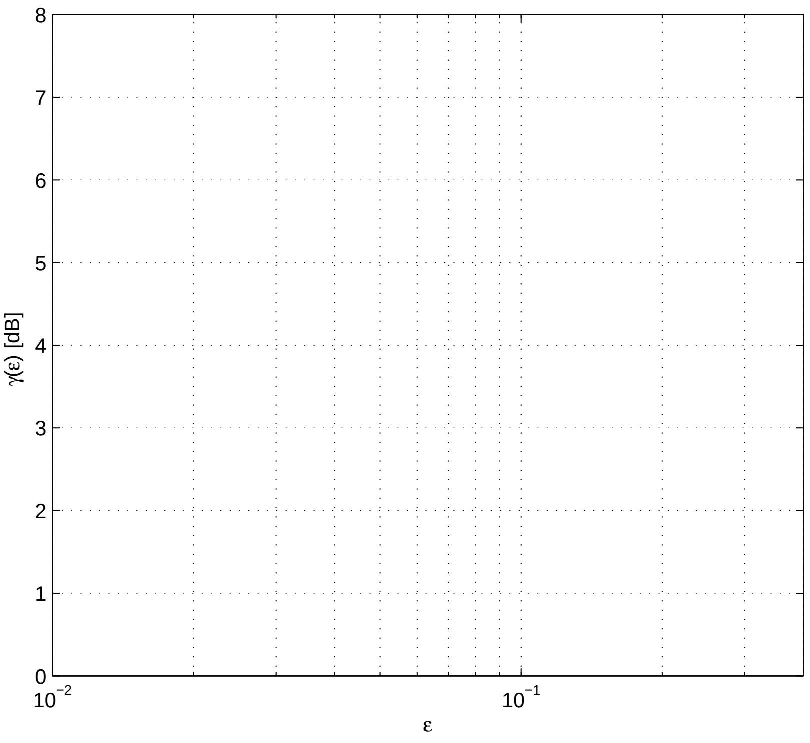


Figure 5.2: SNR loss over frequency offsets for different SNRs

3. Using the results from Fig. 5.2, determine the maximum tolerable frequency offsets ε for preserving a SNR loss below 1 [dB].
4. Using (2.5), for system bandwidth of $\frac{1}{T_S} = 2$ MHz and DFT length of $N = 256$, calculate the maximum tolerable frequency offset f_d in Hz for preserving the same SNR losses as in 3.

5. For the same value of frequency offset f_d in Hz as in 4., what is the SNR loss for $\frac{1}{T_S} = 5$ MHz and $N = 256$?
6. For the same value of frequency offset f_d in Hz as in 4., what is the SNR loss for $\frac{1}{T_S} = 2$ MHz and $N = 64$?
7. Describe the observed influence of signal bandwidth as well as the DFT length on the SNR loss.

6 Laboratory Exercises

6.1 Description

The goal of these lab exercises is to evaluate the system performance of a given SDR framework in real-time transmission conditions and compare them with previously obtained analytical results. The high flexibility of the developed framework allows for high reconfigurability of transmission signal parameters and easy monitoring of the desired performance.

During the exercises there will be up to 5 groups of 2-3 students working with 2 PCs equipped with USRPs. Exercises 6.2.1, 6.3 and 6.4.1 should be done on a single PC and each group should be appropriately divided. For exercises 6.2.2 and 6.4.2 the whole group should use 2 PCs in order to perform real-time transmission.

Before beginning to work on the following exercises you need to start CORBA's name and event service and set appropriate configurations as it is described in tasks 1-8 in Appendix A.2.

6.2 Lab 1: AWGN Channel

Firstly, the analytical results obtained in the preparatory exercises of Section 5.1 will be verified in **local environment** by means of a simulated AWGN channel. Afterwards, the system performance in a real-time transmission with USRP boards will be evaluated and compared to the analytical results.

6.2.1 Local Environment

This task needs to be performed on a single PC. Given an AWGN simulation model as shown in Fig. 6.1 transmitter and receiver communicate over the **channel** which is realized using a FIFO pipe file.

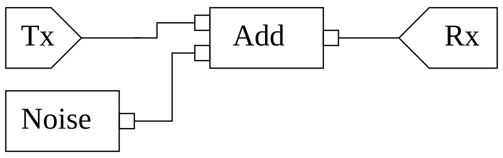


Figure 6.1: AWGN channel model

1. Open six terminal windows which will be used for system's manipulation.
2. In the first and the second terminal start the transmitter and the receiver, respectively, working in **local environment** for particular SNR values given in Table 6.1 by executing following scripts in separate terminal windows:
   `./usrp_tx --to-file ~/omnilog/channel --bandwidth=1M --fft-length=256 --subcarriers=200 --station-id=100 --nameservice-ip=`**your_comp** `--awgn --snr=`**snr_value**
   and
   `./usrp_rx --from-file ~/omnilog/channel --bandwidth=1M --fft-length=256 --subcarriers=200 --station-id=1 --nameservice-ip=`**your_comp** `--scatterplot`
   where **your_comp** denotes the operating PC hostname and **snr_value** is the simulated SNR value.

3.
   - To set the transmit amplitude for obtaining the desired SNR, in the third terminal start the calibration resource manager by executing:
     `./run_script ../python/rm_calibrate.py`
   - In order to monitor the system performance start the transmitter's and the receiver's GUIs in the fourth and the fifth terminal, respectively, as it is described in Appendix A.3.
   - At the receiver's GUI, fit the values of estimated SNR to the values given in Table 6.1 by changing transmission amplitude (changing the values in textfield **Constraint**). The selected transmission amplitude will be used further on for actual simulations.
   - Stop the calibration script in the third terminal.
4. Start the appropriate resource manager in the third terminal by executing:
   `./run_script ../python/rm_exercise_1_1.py -a` **tx_amplitude** `--snr=`**snr_value**
   where **snr_value** denotes the nominal SNR value given in Table 6.1 and **tx_amplitude** is the previously calibrated transmission amplitude.
5. In the sixth terminal start the constellation diagrams as it is described in Appendix A.3.
6. During simulation, read measured SNR and BER from the resource manager terminal or from the log file located in **~/omnilog/** and fill in the Table 6.1. When measurement is over stop the resource manager in the third terminal.
7. Repeat the steps 2, 4-6 for all SNR values given in Table 6.1.

Table 6.1: The BER for various measured SNRs and modulation schemes in **local environment**

Nominal E_s/N_0 [dB]	5	10	15	20	25	30
Measured E_s/N_0 [dB]						
BER_{BPSK}						
Measured E_s/N_0 [dB]						
BER_{4-QAM}						
Measured E_s/N_0 [dB]						
BER_{16-QAM}						
Measured E_s/N_0 [dB]						
BER_{64-QAM}						
Measured E_s/N_0 [dB]						
$BER_{256-QAM}$						

8. Add the BER curves from Table 6.1 to Fig. 5.1 and complete the legend.
9. Compare the obtained curves.

6.2.2 Real-time Transmission

This task needs to be performed on two PCs. For real-time transmission in realistic channel conditions the **channel** and AWGN blocks shown in Fig. 6.1 are replaced by the real channel between two USRP boards.

1. On the transmitter's PC in the first terminal start the transmitter by executing:
   `./usrp_tx -f` **frequency** `--bandwidth=1M --fft-length=256`
   `--subcarriers=200 --station-id=100 --nameservice-ip=`**comp_1**
   where **comp_1** is the name of the transmitting PC while transmission **frequency** depends on the daughterboard type of your USRP. It will be given to you by the supervisors.
2. In order to use the same name service which runs on a transmitter's PC, on a receiver's PC make a omniorb configuration file which will be stored in **~/omnilog/** by executing
   `./mk_configuration` **comp_1**
3. On the receiver's PC set the omniorb configuration file
   export `OMNIORB_CONFIG=~/omnilog/omniORB-`**comp_1**`.cfg`

Note: this action should be performed in all active terminal windows on the receiver's PC.

4. On the receiver's PC start the receiver in the first terminal by executing:
   `./usrp_rx -f` **frequency** `--bandwidth=1M --fft-length=256 --subcarriers=200 --station-id=1 --nameservice-ip=`**comp_1** `--scatterplot`
5. To set the transmit amplitude for obtaining the desired SNR value, on transmitter's PC in the second terminal start the calibration resource manager by executing:
   `./run_script ../python/rm_calibrate.py`
6. In order to monitor the system performance start the transmitter's GUI in the third terminal on transmitter's PC and the receiver's GUIs in the second terminal receiver's PC, as it is described in Appendix A.3, using the transmitter's PC hostname for **nameservice** argument.
7. At the receiver's GUI, fit the values of estimated SNR to the values given in Table 6.2 by changing transmission amplitude (changing the values in textfield **Constraint** and **rx_gain**). The selected transmission amplitude will be used further on for actual simulations.
8. Stop the calibration script.
9. Start the appropriate resource manager in the second terminal on transmitter's PC by executing:
   `./run_script ../python/rm_exercise_1_2.py -a` **tx_amplitude** `--snr=`**snr_value**
   where **snr_value** denotes the nominal SNR value given in Table 6.2 and **tx_amplitude** is the previously calibrated transmission amplitude.
10. In the third terminal on receiver's PC start the constellation diagrams as it is described in Appendix A.3.
11. During simulation, read measured SNR and BER from the resource manager terminal or from the log file located in **~/omnilog/** and fill in the Table 6.2. When measurement is over stop the resource manager.
12. Repeat the steps 5-9, 11 for all SNR values given in Table 6.2.

Table 6.2: The BER for various measured SNRs and modulation schemes during real-time transmission

Nominal E_s/N_0 [dB]	5	10	15	20	25	30
Measured E_s/N_0 [dB]						
BER_{BPSK}						
Measured E_s/N_0 [dB]						
BER_{4-QAM}						
Measured E_s/N_0 [dB]						
BER_{16-QAM}						
Measured E_s/N_0 [dB]						
BER_{64-QAM}						
Measured E_s/N_0 [dB]						
$BER_{256-QAM}$						

13. Add the BER curves from Table 6.2 to Fig. 5.1 and complete the legend.
14. Compare the curves of Fig. 5.1.

6.3 Lab 2: Multipath Channel

This task needs to be performed on a single PC. The goal of this lab exercise is to observe the influence of a frequency selective channel on the system performance. The experiment will be conducted in **local environment** since the real channel environment for the given transmission bandwidth does not experience multipath propagation. A corresponding simulation model is shown in Fig. 6.2.

1. Open six terminal windows which you will need for system's manipulation.

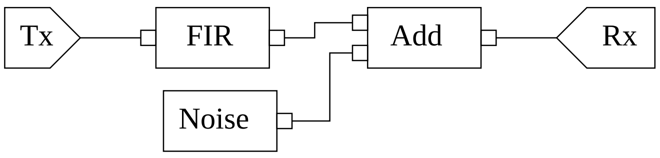


Figure 6.2: Multipath channel model

2. In the first and the second terminal start the transmitter and the receiver, respectively, working in **local environment** for particular SNR values given in Table 6.3 by executing following scripts in separate terminal windows:
   `./usrp_tx --to-file ~/omnilog/channel --bandwidth=1M --fft-length=256 --subcarriers=200 --station-id=100 --nameservice-ip=`**your_comp** `--awgn --freq-sel --snr=`**snr_value**
   and
   `./usrp_rx --from-file ~/omnilog/channel --bandwidth=1M --fft-length=256 --subcarriers=200 --station-id=1 --nameservice-ip=`**your_comp** `--scatterplot`
   where **your_comp** denotes the operating PC hostname and **snr_value** is the simulated SNR value.
3. - To set the transmit amplitude for obtaining the desired SNR, in the third terminal start the calibration resource manager by executing:
     `./run_script ../python/rm_calibrate.py`
   - In order to monitor the system performance start the transmitter's and the receiver's GUIs in the fourth and the fifth terminal, respectively, as it is described in Appendix A.3.
   - At the receiver's GUI, fit the values of estimated SNR to the values given in Table 6.3 by changing transmission amplitude (changing the values in textfield **Constraint**). The selected transmission amplitude will be used further on for actual simulations.
   - Stop the calibration script in the third terminal.
4. Start the appropriate resource manager in the third terminal by executing:
   `./run_script ../python/rm_exercise_2_1.py -a` **tx_amplitude** `--snr=`**snr_value**
   where **snr_value** denotes the nominal SNR value given in Table 6.3 and **tx_amplitude** is the previously calibrated transmission amplitude.
5. In the sixth terminal start the constellation diagrams as it is described in Appendix A.3.
6. During simulation, read measured SNR and BER from the resource manager terminal or from the log file located in **~/omnilog/** and fill in the Table 6.3. When measurement is over stop the resource manager in the third terminal.
7. Repeat the steps 2, 4-6 for all SNR values given in Table 6.3.

Table 6.3: The BER for various SNRs and modulation schemes in a frequency selective channel with equalization

Nominal E_s/N_0 [dB]	5	10	15	20	25	30
Measured E_s/N_0 [dB]						
BER_{BPSK}						
Measured E_s/N_0 [dB]						
BER_{4-QAM}						
Measured E_s/N_0 [dB]						
BER_{16-QAM}						
Measured E_s/N_0 [dB]						
BER_{64-QAM}						
Measured E_s/N_0 [dB]						
$BER_{256-QAM}$						

8. Repeat the previous experiment, now disabling the equalization block at the receiver. Write the results into the Table 6.4.

Table 6.4: The BER for various SNRs and modulation schemes in a frequency selective channel without equalization

Nominal E_s/N_0 [dB]	5	10	15	20	25	30
Measured E_s/N_0 [dB]						
BER_{BPSK}						
Measured E_s/N_0 [dB]						
BER_{4-QAM}						
Measured E_s/N_0 [dB]						
BER_{16-QAM}						
Measured E_s/N_0 [dB]						
BER_{64-QAM}						
Measured E_s/N_0 [dB]						
$BER_{256-QAM}$						

9. Draw graphs for the BER p_b in dependency of the SNR E_S/N_0 into Fig. 6.3. Sketch the curves to all modulations given in Table 6.3 for all entries of this table. Give an appropriate legend.

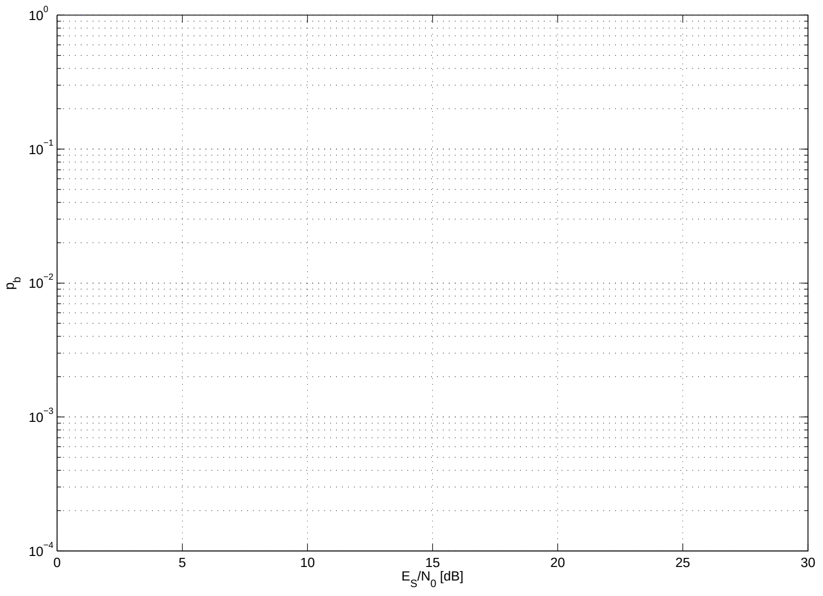


Figure 6.3: The BER over SNR performance for various modulation schemes

10. Compare the obtained curves for the case with and without equalization.

6.4 Lab 3: Frequency Offset Analysis

The goal of this lab exercise is to examine the influence of frequency offset on system performance. Without loss of generality, the experiment will be conducted only for QPSK modulation scheme.

6.4.1 Local Environment

This task needs to be performed on a single PC. The corresponding simulation model of wireless channel in the presence of AWGN and CFO is shown in Fig. 6.4.

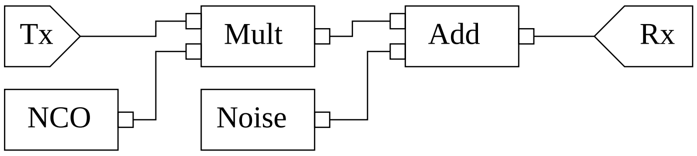


Figure 6.4: Multipath channel model

1. Open six terminal windows which you will need for system's manipulation.
2. In the first and the second terminal start the transmitter and the receiver, respectively, working in **local environment** for particular SNR values given in Table 6.5 by executing following scripts in separate terminal windows:
   `./usrp_tx --to-file ~/omnilog/channel --bandwidth=1M --fft-length=256 --subcarriers=200 --station-id=100 --nameservice-ip=`**your_comp**`--awgn --snr=`**snr_value** `--freqoff=`**freq_off**
   and
   `./usrp_rx --from-file ~/omnilog/channel --bandwidth=1M --fft-length=256 --subcarriers=200 --station-id=1 --nameservice-ip=`**your_comp** `--scatterplot --disable-freq-sync`
   where **your_comp** denotes the operating PC hostname, **snr_value** is the simulated SNR value and where **freq_off** is the simulated frequency offset.
3. - To set the transmit amplitude for obtaining the desired SNR, start the calibration resource manager in the third terminal by executing:
     `./run_script ../python/rm_calibrate.py`
     Note: during calibration freq_off should be set to 0.
   - In order to monitor the system performance start the transmitter's and the receiver's GUIs in the fourth and the fifth terminal, respectively, as it is described in Appendix A.3 in steps 1-2.
   - At the receiver's GUI, fit the values of estimated SNR to the values given in Table 6.5 by changing transmission amplitude (changing the values in textfield **Constraint**). The selected transmission amplitude will be used further on for actual simulations.
   - Stop the calibration script in the third terminal.
4. Start the appropriate resource manager in the third terminal by executing:
   `./run_script ../python/rm_exercise_3_1.py -a` **tx_amplitude** `--snr=`**snr_value**
   where **snr_value** denotes the nominal SNR value given in Table 6.5 and **tx_amplitude** is the previously calibrated transmission amplitude.
5. In the sixth terminal start the constellation diagrams as it is described in Appendix A.3.
6. During simulation, read measured SNR from the resource manager terminal or from the log file located in **~/omnilog/** and fill in the Table 6.5. When measurement is over stop the resource manager in the third terminal.
7. Repeat the steps 2, 4-6 for all values of frequency offsets and SNRs given in Table 6.5.
8. Use entries from Table 6.5 in order to fill in the Table 6.6.
9. Add the SNR loss curves from Table 6.6 to Fig. 5.2 and complete the legend.

Table 6.5: Measured SNR for different values of frequency offsets and nominal SNRs

ε	0	10^{-2}	$2 \cdot 10^{-2}$	$5 \cdot 10^{-2}$	10^{-1}	$2 \cdot 10^{-1}$
Measured $E_S/N_0\|_{E_S/N_0=5[dB]}[dB]$						
Measured $E_S/N_0\|_{E_S/N_0=10[dB]}[dB]$						
Measured $E_S/N_0\|_{E_S/N_0=15[dB]}[dB]$						
Measured $E_S/N_0\|_{E_S/N_0=20[dB]}[dB]$						

Table 6.6: SNR losses for different values of frequency offsets and nominal SNRs

ε	10^{-2}	$2 \cdot 10^{-2}$	$5 \cdot 10^{-2}$	10^{-1}	$2 \cdot 10^{-1}$
$\gamma(\varepsilon)\|_{E_S/N_0=5[dB]}[dB]$					
$\gamma(\varepsilon)\|_{E_S/N_0=10[dB]}[dB]$					
$\gamma(\varepsilon)\|_{E_S/N_0=15[dB]}[dB]$					
$\gamma(\varepsilon)\|_{E_S/N_0=20[dB]}[dB]$					

10. Compare the obtained curves to those derived from the analytical model. How much do they differ and why?

6.4.2 Real-time Transmission

This task needs to be performed on two PCs.

1. On transmitter's PC in the first terminal start the transmitter by executing:
   `./usrp_tx -f` **frequency** `--bandwidth=1M --fft-length=256 --subcarriers=200 --station-id=100 --nameservice-ip=`**comp_1**
   where **comp_1** is the name of the transmitting PC while transmission **frequency** depends on the daughterboard type of your USRP. It will be given to you by the supervisors.
2. In order to use a name service which runs on a transmitter's PC, on a receiver's PC make a omniorb configuration file which will be stored in `~/`**omnilog/** by executing
   `./mk_configuration` **comp_1**
3. On receiver's PC set the omniorb configuration file
   export `OMNIORB_CONFIG=~/omnilog/omniORB-`**comp_1**`.cfg`
   Note: this action needs to be repeated in all active terminal windows on the receiver's PC.
4. On the receiver's PC start the receiver in the first terminal by executing:
   `./usrp_rx -f` **frequency** `--bandwidth=1M --fft-length=256 --subcarriers=200 --station-id=1 --nameservice-ip=`**comp_1** `--scatterplot --log-freq-off`
5. To set the transmit amplitude for obtaining the desired SNR value, on transmitter's PC in the second terminal start the calibration resource manager by executing:
   `./run_script ../python/rm_calibrate.py`
6. In order to monitor the system performance start the transmitter's GUI in the third terminal on transmitter's PC and the receiver's GUIs in the second terminal receiver's PC, as it is described in Appendix A.3.
7. At the receiver's GUI, fit the values of estimated SNR to 10 dB by changing transmission amplitude (changing the values in textfield **Constraint** and **rx_gain**). The selected transmission amplitude will be used further on for actual simulations.
8. Stop the calibration script.
9. Start the appropriate resource manager in the second terminal on transmitter's PC by executing:
   `./run_script ../python/rm_exercise_3_2.py -a` **tx_amplitude** `--snr=10`
   where **tx_amplitude** is the previously calibrated transmission amplitude.
10. In the third terminal on receiver's PC start the constellation diagrams as it is described in Appendix A.3.
11. Stop the receiver after about ten seconds and open the log file by executing:
    `gr_plot_float.py ~/omnilog/`**freq_off**`.float`
12. Read the estimated frequency offset $\hat{\varepsilon}$ from the figure and calculate frequency offset $\hat{f}_d$ in Hz using (2.5). Note that the **bandwidth** you use as transmission parameter corresponds to $1/T_s$.
13. Add measured frequency offset $\hat{f}_d$ to frequency previously used for real-time transmission and start the receiver with the new corrected frequency.

14. Stop the receiver after about ten seconds and open the log file by executing:
    `gr_plot_float.py ~/omnilog/`**freq_off**`.float`
15. What is the new value of the estimated frequency offset?
16. Conclude the results.

A Scripts Manipulation

In this lab exercises, you will use the GNU Radio framework to test and evaluate the performance of an OFDM system developed in an SDR framework. Some useful Linux commands for easy file manipulation within the terminal window will be given. Afterwards, procedures for starting an OFDM transceiver and given GUIs will be explained.

A.1 Useful Linux Commands and Hints

- `man command` = This opens the help file for the specified command. For example, type man pwd.
- `pwd` = Print working directory.
- `ls` = List files.
- `cd` = Change directory.
- `mkdir` = Create a new directory.
- `cp` = Copy a file.
- `cp -r` = Copy a directory recursively.
- `ps -e` = Lists all current running processes
- `grep` = Searches and prints lines matching a given pattern.
- `vim` = A shell based text editor.
- `killall` = Kills all processes matching a given name or ID number.
- `rm` = Removes a file.
- `chmod` = Change read, write and execute permissions for the owner, group, and guest users.
- `<CTRL>C` = Abort process (application).
- `./` = Current directory.
- `../` = Parent directory of the current directory.
- `<TAB>` = Command line completion (automatically fills in partially typed commands).
- middle mouse button click - paste the previously selected text

A.2 OFDM System Scripts Manipulation

1. In home directory create **omnilog** directory that will be used for storing the log files required for system configuration
2. Go to **omnilog** directory and create the FIFO pipe **channel** which will be used for communication between transmitter and receiver in **local environment** mode by executing
   `mkfifo channel`
3. Within the terminal window go to your work directory
   `cd /opt/sources/ofdm_ti_praktikum/ofdm_ti_praktikum/src/apps`
4. Make a omniorb configuration file which will be stored in `~`**/omnilog/** by executing
   `./mk_configuration` **your_comp**
   where **your_comp** is your PC hostname.

5. Start the CORBA name and event service by executing the script
   `./Setup_Local_Environment`
   Note: ignore the warning messages.
6. Export environment variables
   `. /opt/gnuradio/bin/environment`
7. In order to set OMNIORB configuration file type
   export `OMNIORB_CONFIG=~/omnilog/omniORB-`**your_comp**`.cfg`
   Note: last two actions need to be repeated in all active terminals.
8. To check if name service is correctly running type
   `nameclt list`
   and the list of initiated name clients should be shown in the terminal window.
9. Transmitter (Tx) is started by typing
   `./usrp_tx`
   Prior to actually execute the script, take a look at options that can be configured from command line by typing
   `./usrp_tx --help`
   Here is the list of options:
   - `h, --help` show this help message and exit
   - `-f FREQ, --freq=FREQ` set Tx and/or Rx frequency to FREQ [default=none]
   - `-which-usrp=WHICH_USRP` select USRP box to use
   - `--from-file=FROM_FILE` Sent recorded stream with USRP
   - `--to-file=TO_FILE` Record transmitter to disk, not being sent to USRP
   - `-e INTERFACE, --interface=INTERFACE` select Ethernet interface, default is eth0
   - `-m MAC_ADDR, --mac-addr=MAC_ADDR` select USRP by MAC address, default is auto-select
   - `--usrp2` Use USRP2 Interface
   - `--fft-length=FFT_LENGTH` set the number of FFT bins [default=512]
   - `-a AMPL, --rms-amplitude=AMPL` set total bandwidth. [default=500k]
   - `--tx-freq=FREQ` set transmit frequency to FREQ [default=none]
   - `--snr=SNR` Simulate AWGN channel
   - `--freqoff=FREQOFF` Simulate frequency offset [default=none]
   - `--samplingoffset=SAMPLINGOFFSET` Simulate sampling frequency offset [default=none]
   - `--awgn` Enable static AWGN power for BER measurement
   - `--data-blocks=DATA_BLOCKS` Set the number of data blocks per OFDM frame [default=9]
   - `--subcarriers=SUBCARRIERS` Set the number of occupied FFT bins. Default: FFT window size - Pilot Subcarriers
   - `--cp-length=CP_LENGTH` Set the unique ID number for each node within the system
   - `--station-id=STATION_ID` Set IP address or hostname that hosts the CORBA NameService
   - `--nameservice-ip=NAMESERVICE_IP`
   - `--nameservice-port=NAMESERVICE_PORT` Set access port to NameService
   - `--log` enable file logs [default=False]
   - `--debug` Enable debugging mode [default=False]
10. For example, in order to start transmission over USRP, execute the script with following parameters
    `./usrp_tx -f 2.45G --bandwidth=2M --fft-length=256 --subcarriers=200 --station-id=100 --nameservice-ip=`**your_comp**
11. Similarly, in order to start transmission in local environment in AWGN channel with SNR = 20 dB, execute the script with following parameters
    `./usrp_tx --to-file channel --bandwidth=2M --fft-length=256 --subcarriers=200 --station-id=100 --nameservice-ip=`**your_comp** `--awgn --berm --snr=20`.

12. Receiver (Rx) is started by typing
    `./usrp_rx`
    Prior to actually execute the script, take a look at options that can be configured from command line by typing
    `./usrp_rx --help`
    Here is the list of options:
    - `h, --help` show this help message and exit
    - `-f FREQ, --freq=FREQ` set Tx and/or Rx frequency to FREQ [default=none]
    - `--which-usrp=WHICH_USRP` select USRP box to use
    - `--rx-gain=GAIN` set receiver gain in dB [default=midpoint]. See also --show-rx-gain-range
    - `--show-rx-gain-range` print min and max Rx gain available on selected daughterboard
    - `--from-file=FROM_FILE` Run Receiver on recorded stream
    - `--to-file=TO_FILE` Record stream from USRP to disk, no processing
    - `-e INTERFACE, --interface=INTERFACE` select Ethernet interface, default is eth0
    - `-m MAC_ADDR, --mac-addr=MAC_ADDR` select USRP by MAC address, default is auto-select
    - `--usrp2` Use USRP2 Interface
    - `--event-rxbaseband` Enable RX baseband via event channel alps
    - `--fft-length=FFT_LENGTH` set the number of FFT bins [default=512]
    - `--disable-ctf-enhancer` Disable Enhancing the MSE of CTF
    - `--scatterplot` Enable the GUI interface for showing constellation diagrams of the received signal
    - `--bandwidth=BANDWIDTH` set total bandwidth. [default=500k]
    - `--rx-freq=FREQ` set receive frequency to FREQ [default=none]
    - `--online-work` Force the OFDM receiver to work during file record [default=False]
    - `--data-blocks=DATA_BLOCKS` set the number of data blocks per OFDM frame [default=9]
    - `--subcarriers=SUBCARRIERS` set the number of occupied FFT bins. Default: FFT window size - Pilot Subcarriers
    - `--cp-length=CP_LENGTH` set the number of bits in the cyclic prefix. Default: 12.5
    - `--station-id=STATION_ID` Set the unique ID number for each node within the system
    - `--nameservice-ip=NAMESERVICE_IP` Set IP address or hostname that hosts the CORBA NameService
    - `--nameservice-port=NAMESERVICE_PORT` Set access port to NameService
    - `--log` Enable file logs [default=False]
    - `--log-freq-off` Enable log of frequency offset estimation
    - `--debug` Enable debugging mode [default=False]
    - `--disable-freq-sync` Disabling frequency synchronization stage
    - `--disable-equalization` Disabling equalization stage
    - `--disable-phase-tracking` Disabling phase tracking stage
    - `--disable-time-sync` Disabling time synchronization stage
13. For example, in order to start reception over USRP, execute the script with following parameters
    `./usrp_rx -f 2.45G --bandwidth=2M --fft-length=256 --subcarriers=200 --station-id=1 --nameservice-ip=`**your_comp** `--scatterplot`
14. Similarly, in order to start transmission in local environment in AWGN channel, execute the script with following parameters
    `./usrp_rx --from-file channel --bandwidth=2M --fft-length=256 --subcarriers=200 --station-id=1 --nameservice-ip=`**your_comp** `--scatterplot`
15. Resource manager (that will be implements in different scripts for different exercises) is started by executing the script
    `./run_script ../python/exercise_x_x.py`

A.3 GUI Scripts Manipulation

1. In order to start system's GUIs do
   `cd /opt/sources/ofdm_ti_praktikum/ofdm_ti_praktikum/src/qt/gui`
2. Tx's GUI is started by executing
   `./qtgui --tx --nameservice=`**your_comp**
3. Rx's GUI is started by executing
   `./qtgui --rx --nameservice=`**your_comp**
4. Constellation plots can be observed by going into directory
   `cd /opt/sources/ofdm_ti_praktikum/ofdm_ti_praktikum/src/qt/scatterplot`
   and executing the script
   `./scatterplot-gui`

Glossary

ADC	Analog-to-digital converter
API	Application Programming Interface
AWGN	Additive White Gaussian Noise
BER	Bit error rate
BPSK	Binary Phase Shift Keying
CFO	Carrier frequency offset
CORBA	Common Object Request Broker Architecture
CP	Cyclic prefix
CPU	Central Processing Unit
CSI	Channel state information
DAC	Digital-to-analog converter
DECT	Digital Enhanced Cordless Telecommunications
DFT	Discrete Fourier Transfom
DSP	Digital Signal Processor
FFT	Fast Fourier Transfom
GPS	Global Positioning System
GSM	Global System for Mobile communications
GUI	Graphical User Interface
ICI	Intercarrier interference
IDFT	Inverse Discrete Fourier Transfom
IF	Intermediate frequency
ISI	Intersymbol interference
LO	Local oscillator
LTE	Long Term Evaluation
NCO	Numerically controlled oscillator
OFDM	Orthogonal Frequency Division Multiplexing
PCI	Peripheral Component Interconnect
PSK	Phase Shift Keying
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying

RF Radio frequency

SDR Software Defined Radio

SER Symbol error rate

SNR Signal-to-noise ratio

USRP Universal Software Radio Peripheral

WiMAX Worldwide Interoperability for Microwave Access

WLAN Wireless Local Access Network

Index

Bibliography

[Aur09] Dominik Auras. *GNU Radio Implementierungen zur Synchronisierung von OFDM-Systemen.* Studienarbeit, RWTH Aachen, 2009.

[Erg09] Mustafa Ergen. *Mobile Broadband - Including WiMAX and LTE.* Springer Publishing Company, Incorporated, 2009.

[GNUrg] "Object Management Group", `http://www.omg.org/`.

[GNUpp] IT++, `http://sourceforge.net/apps/wordpress/itpp`.

[GNUac] "GNU Radio", `http://gnuradio.org/trac`.

[Gol05] Andrea Goldsmith. *Wireless Communications.* Cambridge University Press, New York, NY, USA, 2005.

[LS06] Ye Li and Gordon L. Stuber. *Orthogonal Frequency Division Multiplexing for Wireless Communications (Signals and Communication Technology).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Mar03] Alex Martelli. *Python in a Nutshell. A Desktop Quick Reference.* O'Reilly & Associates, 1 edition, 2003.

[Mey98] Scott Meyers. *Effektiv C++ programmieren.* Addison Wesley, 1998.

[Mey99] Scott Meyers. *Mehr Effektiv C++ programmieren.* Addison Wesley, 1999.

[Mey01] Scott Meyers. *Effective STL.* Addison Wesley, 2001.

[Mit06] Joseph III Mitola. *Cognitive Radio Architecture: The Engineering Foundations of Radio Xml.* Wiley-Interscience, New York, NY USA, 2006.

[MR97] Thomas J. Mowbray and William A. Ruh. *Inside CORBA.* Addison Wesley, 1997.

[Mue08] Andreas Mueller. *DAB, Software Receiver Implementation.* Semester Thesis, ETH Zurich, 2008.

[OHE97] Robert Orfali, Dan Harkey, and Jeri Edwards. *Instant CORBA.* John Wiley & Sons, 1997.

[PMK07] Man-on Pun, Michele Morelli, and C. C. Jay Kuo. *Multi-Carrier Techniques For Broadband Wireless Communications: A Signal Processing Perspectives.* Imperial College Press, London, UK, UK, 2007.

[Rou08] Tony J. Rouphael. *RF and Digital Signal Processing for Software-Defined Radio: A Multi-Standard Multi-Mode Approach.* Newnes, Newton, MA, USA, 2008.

[SA07] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards.* Addison Wesley, Boston, Massachusetts, USA, 5 edition, July 2007.

[Str00] Bjarne Stroustrup. *Die C++ Programmiersprache.* Addison Wesley, 2000.

[ZAM09] Milan Zivkovic, Dominik Auras, and Rudolf Mathar. Reconfigurable framework for adaptive ofdm transmission. In *WINTECH '09: Proceedings of the 4th ACM international workshop on Experimental evaluation and characterization*, pages 91–92, New York, NY, USA, 2009. ACM.