Master's Thesis

The Impact of Channel Impairments on SNR Estimation for DVB-S2

Johannes Pribyl, BSc

Institute of Communication Networks and Satellite Communications
Graz University of Technology

Supervisor: Univ.-Prof. Dipl.-Ing. Dr. techn. Otto Koudelka

Graz, March 2012

Abstract

This thesis was compiled at the Research Group for Space and Acoustics of the Institute for Information and Communication Technologies at the Joanneum Research Forschungsgesellschaft mbH.

The goal was to examine several different methods for SNR estimation using different stochastic moments. An existing DVB-S simulator was modified to fit DVB-S2 standards, then expanded by the implementation of the estimation algorithms. These algorithms were then subjected to different channel impairments and tested for their performance under ideal and impaired circumstances.

The result of this work are several tested modules for the DVB-S simulator developed at the institute, and many simulations and insights concerning the performance and ideal working conditions of SNR estimators. Large datasets and simulation times had to be dealt with; the code was incorporated into a large simulation environment.

Kurzfassung

Diese Arbeit wurde an der Forschungsgruppe für Weltraumtechnik und Akustik des Instituts für Informations- und Kommunikationstechnologien der Joanneum Research Forschungsgesellschaft mbH erstellt.

Das Ziel war es, verschiedene SNR-Schätzer zu untersuchen. Dazu wurde ein institutseigener DVB-S-Simulator erweitert, um den neueren DVB-S2-Standard zu unterstützen. Weiters wurden die Schätzer implementiert und unter verschiedenen Bedingungen untersucht.

Das Ergebnis dieser Arbeit waren Module für den Simulator, sowie Simulationen und Einblicke in das Einsatzgebiet der SNR-Schätzer. Eine interessante Herausforderung waren die großen Datenmengen und Laufzeiten der Simulationen, sowie die Bearbeitung des bereits existierenden Quellcodes.

Acknowledgements

This thesis was compiled during my time at the department for space and acoustics of the Institute for Information and Communication Technologies at the Joanneum Research Forschungsgesellschaft mbH. In this context I would like to thank DI Michael Schmidt for his administrative support and advice; DI Johannes Ebert, and DI Harald Schlemmer supported me with their ample technical knowledge.

Furthermore, I would like to thank my supervisor from the Institute of Communications Networks and Satellite Communications at Graz University of Technology, Univ.-Prof. Dipl.-Ing. Dr. techn. Otto Koudelka, for his help and support.

Lastly and most importantly, I want to thank my family for their support during my years of study. This would not have been possible without your encouragement and support.

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Contents

Abstract / Kurzfassung 1

Acknowledgements 2

Statutory Declaration 4

Contents 6

1 Introduction 7
  1.1 DVB-S2 7
  1.2 Adaptive Coding and Modulation 7
  1.3 Physical Layer Framing 8
  1.4 Estimation theory 8

2 Outline 9

3 Fundamentals 10
  3.1 Physical Layer Framing 10
    3.1.1 Structure of the PLHEADER 11
    3.1.2 The SOF field 12
    3.1.3 The PLS code 12
  3.2 The Simulator 13
    3.2.1 Transmission Path 13
    3.2.2 Softbit Generation 14
    3.2.3 PLScode Decoding 14
  3.3 Modulation 15
    3.3.1 Phase-Shift Keying 15
    3.3.2 Amplitude and Phase-Shift Keying 16
  3.4 Channel Impairments 16
    3.4.1 Carrier Phase Error 16
    3.4.2 Frequency Error 17
    3.4.3 Signal-to-Noise Ratio 18
  3.5 Phase Estimation 19

3.6 SNR Estimation 20
3.6.1 Data-aided SNR Estimation 20
3.6.2 Non-data-aided SNR Estimation for M-PSK 20
3.6.3 Non-data-aided SNR Estimation for 16-APSK 21
3.6.4 Non-data-aided SNR Estimation for 32-APSK 22

4 Implementation 23
4.1 Setting up the Work Environment and Compiling the Simulator 23
4.2 Operating the Simulator 24
4.2.1 Simfiles 24
4.2.2 x1, x2: Sweep Support 25
4.2.3 A Typical Simfile 25
4.2.4 Batch Files for Simulation Automation 27
4.3 Program Structure 27
4.4 PLFRAME Handling 28
4.4.1 PLFRAME Insertion 28
4.4.2 PLFRAME Removal 29
4.5 Analysis 30
4.5.1 Analyze SNR 30
4.5.2 Analyze PLFRAME 31
4.6 Other Code 32
4.6.1 Defines and Debug Code 32
4.6.2 Base Class 33
4.6.3 Testdata Setter 33

5 Results 35
5.1 Carrier Phase Error Estimation 35
5.1.1 Simulation Set-up 36
5.1.2 Simulation Results 36
5.2 SNR Estimation 38
5.2.1 Estimator Length 38
5.2.2 General Behaviour of SNR Estimation Algorithms . . . 39
5.3 The Influence of Estimator Length 43
5.3.1 Simulation Set-up 43
5.3.2 Simulation Results 43
5.4 Performance for typical Estimator Lengths 49
5.4.1 Simulation Set-up 49
5.4.2 Simulation Results 49
5.5 The Impact of Frequency Error 55
5.5.1 Simulation Set-up 55
5.5.2 Simulation Results 55

6 Conclusion and Outlook 60

List of Abbreviations 61

List of Tables 62

List of Figures 63

List of Listings 64

Bibliography 66

A Program Source Code 67
A.1 Common Code 67
A.1.1 Defines and Debug Code 67
A.1.2 Base Class 70
A.2 PLFRAME Handling 84
A.2.1 PLFRAME Insertion 84
A.2.2 PLFRAME Removal 89
A.3 Analysis 93
A.3.1 Analyze PLFRAME 93
A.3.2 Analyze SNR 98
A.4 Other Code 106
A.4.1 Testdata 106

B Batch Files for Simulation Automation 109

Chapter 1

Introduction

During the course of this work, a DVB-S simulator was put into operation and subsequently modified; the goals were to support the DVB-S2's PLFRAMES and to implement several SNR estimation algorithms. Furthermore, channel impairments were applied to the simulated channel and their effect on SNR estimation and PLHEADER performance was studied.

This section will present the basic concepts used in this work; section 1.1 introduces the standard DVB-S2 which uses the concept of Adaptive Coding and Modulation (ACM), explained in section 1.2. PLFRAMING and SNR estimation are necessary in order use this feature, and will be introduced in sections 1.3 and 1.4, respectively.

1.1 DVB-S2

DVB-S2 is the second generation DVB via satellites; it is a digital television standard and a successor to DVB-S [12]. It is defined in standard EN 302 307 by the European Telecommunications Standards Institute (ETSI) [1]. It includes – but is not limited to – the following features which are of special interest to this work:

- Modulation schemes up to 32-APSK.
- Adaptive Coding and Modulation (ACM)
- Physical Layer (PL) Framing

The modulations used in this work are explained in section 3.3; different modulation and coding schemes are necessary for the ACM functionality.

1.2 Adaptive Coding and Modulation

Adaptive Coding and Modulation (ACM) means generally the "matching of the modulation, coding and other signal and protocol parameters to the conditions on the radio link" [16]. In the scope of this work, ACM is understood as usage of different code rates and modulations for the next PLFRAME, depending on the Signal-to-Noise Ratio (SNR) of one or more previous PLFRAME(s).

The advantage of this approach is that the available channel capacity is utilized as best as possible. A lower data rate would be suboptimal in good conditions, whereas bad conditions would lead to many dropped frames if only high data rates were used.

1.3 Physical Layer Framing

According to [1] – the DVB-S2 standard –, the data stream shall be cut into consecutive XFECFRAMES which then are packed into PLFRAMES. This is to enable the usage of Adaptive Coding and Modulation. Each PLFRAME can be coded and modulated independently from its predecessor or its successor. The necessary meta-information is transmitted using the highly redundant PLHEADER, which is modulated using a highly reliable modulation.

There are several different options which determine the size of each PLFRAME; furthermore, so-called *pilot blocks* can be inserted at regular intervals to enable certain error-correction measures, e.g. against frequency errors. For a detailed discussion of the standard see section 3.1.

1.4 Estimation theory

Estimation in a mathematical sense means to approximate "the values of [unknown] parameters based on measured [...] data that has a random component" [13]. According to [2], an estimator has several properties; the most important ones in the scope of this work are the MSE and estimator bias.

The *Mean Square Error* (MSE) is a measure for how much the estimates of the estimator deviate from the true value; a low MSE is therefore desirable [14]. The *Cramér-Rao Lower Bound* (CRLB) is a theoretical lower limit to an estimator's MSE [11]; cp. also section 5.2.

The *estimator bias* on the other hand occurs if the estimates cluster around a value different from the actual parameter value to be estimated. If no bias occurs, the estimator is said to be *bias-free.*

Chapter 2

Outline

During the course of this work several estimation algorithms had to be tested and analyzed using a simulator which had been developed at the Institute for Information and Communication Technologies at Joanneum Research.

To this end, two different tasks had to be fulfilled. Firstly, the existing DVB-S simulator had to be expanded by adding a physical layer framing mechanism; secondly, several SNR estimation algorithms had to be implemented and subsequently analyzed using several different channel impairments from the simulator.

The simulator itself was written in C++ and was divided into many different modules, each fulfilling a certain task within the simulation chain; the input files to the simulator, on the other hand, had to comply to a different, proprietary standard.

Due to the large data sets involved, high simulation times had to be dealt with. The simulator itself provided some mechanisms for simulation automation; additional automation possibilities were created using Microsoft Windows batch files and specialized simulator modules.

Chapter 3

Fundamentals

This section introduces several fundamental concepts which were used or implemented in this project. Furthermore, several basic algorithms and formulas are provided here in order to facilitate the general understanding.

First, the structure of the PLHEADER is discussed and explained in detail. Second, the architecture of the simulator which was modified during the course of this work is outlined. Lastly, an overview over phase and SNR estimation methods and general estimation theory is provided.

3.1 Physical Layer Framing

The process of PL framing was standardized by the ETSI in its standard EN 302 307 [1]. The main steps are:

1. The input data should be cut into “SLOTS” of equal length of 90 symbols.
2. Generation and insertion of a “PLHEADER” which contains all necessary information about the PL frame. The PLHEADER should occupy exactly one SLOT at the beginning of the frame.
3. Insertion of so-called “pilot blocks” to help receiver synchronization. The pilot blocks should be inserted every 16 SLOTS.
4. Randomization of the symbols using a PL scrambler.

Furthermore, there are two parameters which influence the PLFRAME’s structure:

- Frame length. The length of data to be packed into the PLFRAME can be 64 800 Bit or 16 200 Bit.
- Pilots. Pilot blocks can be switched on or off. The purpose of pilot blocks is to enable certain error-correction measures, e.g. against phase error or frequency errors.

3.1.1 Structure of the PLHEADER

The PLHEADER is composed of the following fields:

- Start of Frame (SOF). A unique, constant field, length 26 symbols. This constant field can be used for different purposes, e.g. frame synchronization, carrier phase correction, or DA SNR estimation.
- Physical Layer Signalling (PLS) code. A code with high redundancy, length 64 symbols. It is generated from seven bits which hold information about the PLFRAME's structure.

The following figure illustrates the correct PLFRAME composition. Red and yellow items were generated during the course of this project, yellow items are optional:

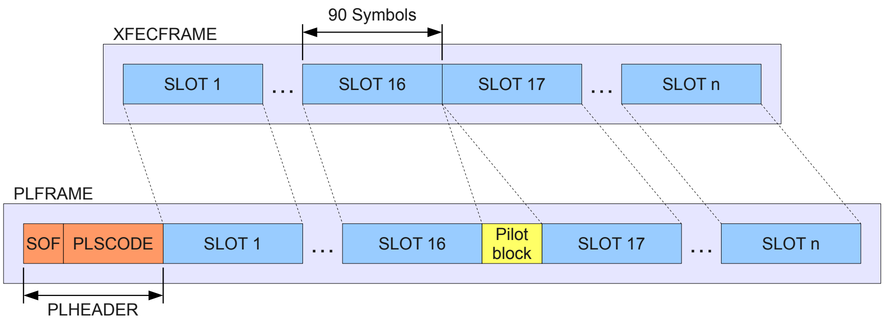


Figure 3.1: PLFRAME construction overview

The complete PLHEADER should then be modulated into symbols using the $\pi/2$-BPSK modulation. This is illustrated by the following formulas:

For all even i:

$$\operatorname{Im}(s_i) = 1 - 2 \cdot b_i \tag{3.1}$$

$$\operatorname{Re}(s_i) = -\operatorname{Im}(s_i) \tag{3.2}$$

For all odd i:

$$\operatorname{Im}(s_i) = 1 - 2 \cdot b_i \tag{3.3}$$

$$\operatorname{Re}(s_i) = \operatorname{Im}(s_i) \tag{3.4}$$

The crucial difference between even and odd symbols is the inverted real part which "flips" the symbols along the real axis. This provides a higher error tolerance and is therefore used for this important meta information.

3.1.2 The SOF field

The SOF consists of a constant, known sequence of 26 bits (given in [1, Chapter 5.5.2.1]). It can be used for phase shift detection or for correlation to detect the start of the PLFRAME.

3.1.3 The PLS code

The PLS code is generated of seven bits consisting of the following two fields:

- MODCOD, 5 bits. This field specifies the input data's **mod**ulation and **cod**e rate. There are 20 possible combinations (specified in [1, Table 12]). Furthermore, this field can be used to indicate an empty dummy PLFRAME.
- TYPE, 2 bits. This field specifies the length of the input data and the presence or absence of pilot fields. The most significant bit indicates the length (0 = normal, 1 = short), the least significant bit indicates the presence of pilots (0 = no pilots, 1 = pilots).

The first six bits – the MODCOD bits and the data bit of the TYPE field – are then multiplied with a 6-by-32 generator matrix (given in [1, Figure 13b]).

For the last step, the pilots bit of the TYPE field is used: If it is 0, each bit will simply be repeated, e.g. $(y_1 y_2 ... y_{32})$ becomes $(y_1 y_1 y_2 y_2 ... y_{32} y_{32})$. If the pilots bit is 1, each repeated bit is inverted; so $(y_1 y_2 ... y_{32})$ becomes $(y_1 \overline{y}_1 y_2 \overline{y}_2 ... y_{32} \overline{y}_{32})$.

Finally, the PLS code is scrambled with a binary sequence (defined in [1, Chapter 5.5.2.4]) using a bit-wise XOR function.

3.2 The Simulator

This section describes several fundamental concepts related to the simulator used during the course of this project.

3.2.1 Transmission Path

A typical transmission path is composed of the three main components sender, channel and receiver. The following Figure 3.2 gives a detailed overview.

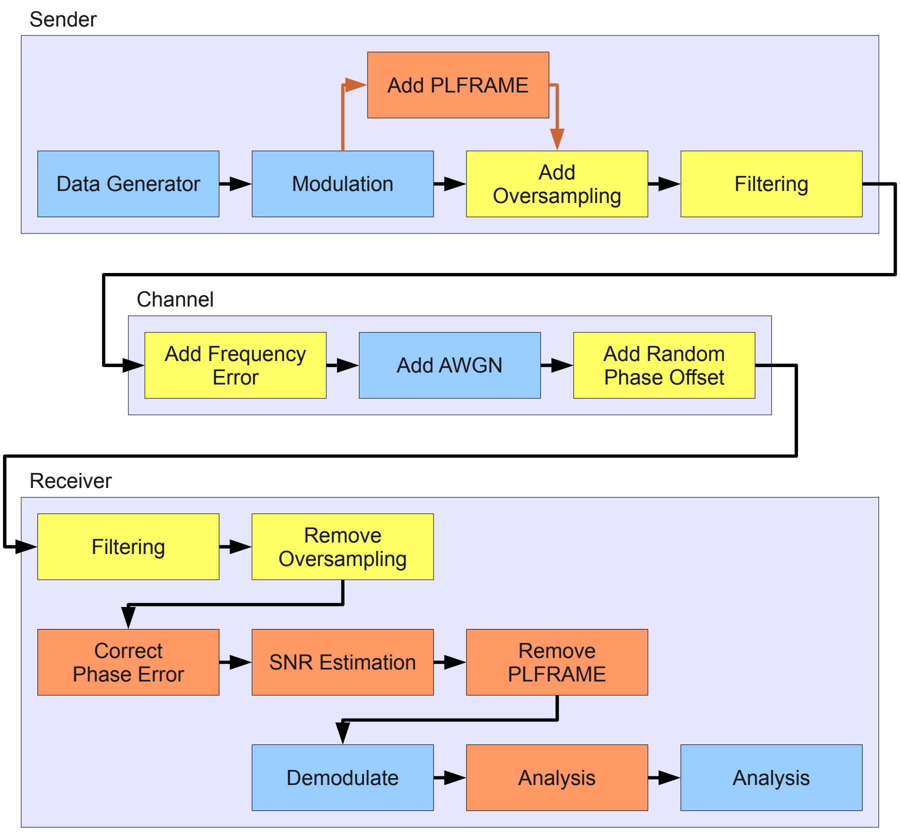


Figure 3.2: A typical transmission path

Blue items are standard components, yellow items are optional and not always used. The red items were added in the course of this project.

3.2.2 Softbit Generation

$\pi/2$-BPSK symbols are differently modulated depending on whether the symbol index is even or odd (for the official definition see subsection 3.1.1 and [1, Chapter 5.5.2]). The following table lists all possible modulation combinations:

Table 3.1: Possible symbol values for $\pi/2$-BPSK modulation

Bit Value	Even Symbol	Odd Symbol	Softbit (ideal)
0	$(-1, 1)$	$(1, 1)$	-1
1	$(1, -1)$	$(-1, -1)$	1

To generate soft bits from such symbols, Equation 3.5 is used. It was extracted from the simulator module `MapperSymbolToSoftBit`, where it was implemented in a different, slightly more complicated way using several `if`-statements:

$$b_i = \frac{\alpha_i \cdot \mathrm{Re}\,(s_i) - \mathrm{Im}\,(s_i)}{2} \tag{3.5}$$

where

$$\alpha_i = (1 - (i \bmod 2) \cdot 2)\,. \tag{3.6}$$

Here, α_i takes either the value 1 for even i or -1 for odd i. The formula as a whole takes either -1 or 1 for perfect values s_i. For noisy values s_i, b_i can take any value between -1 and 1; greater absolute values indicate greater certainty.

3.2.3 PLScode Decoding

The PLScode is decoded by comparing all ($2^7 = 128$) possible PLScode values with the received PLScode. The value with the lowest Hamming distance is chosen to be the correct value.

Simply put, the Hamming distance is a measure for the difference between two symbol sequences, e.g. strings or binary numbers [15]. It is calculated using the following formula:

$$D = \sum_{i}^{n} |r_i - s_i| \tag{3.7}$$

This formula sums up all bit-wise differences. It takes the value 0 for identical values. Due to the high redundancy of the PLScode and its generation, this method works even for very low SNR values, where many bit errors occur.

3.3 Modulation

Modulation is the process of modulating a high-frequency *carrier signal* by varying one ore more of its properties depending on a *modulation signal* which contains the information to be transmitted. In this work we use digital modulation methods which work with two properties of the carrier signal: Phase and amplitude.

3.3.1 Phase-Shift Keying

Phase-Shift Keying (PSK) modulations vary the carrier signal's phase to transmit information. A convenient way of visualizing the modulation's effect is a constellation digram of the complex plane as shown in Figure 3.3.

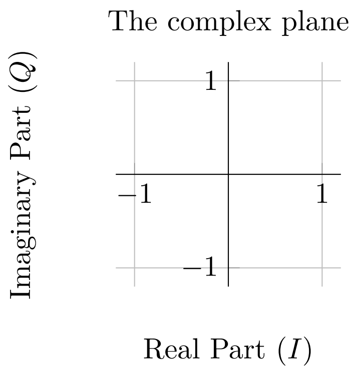


Figure 3.3: An empty constellation diagram

By changing the phase of the carrier signal, we move on a circle around the origin whose radius depends on the carrier's amplitude. This way it is possible to define different symbols at different places in the complex plane. One symbol consists of one or more bits of the modulation signal. Figure 3.4 shows the three PSK-based modulations used in this work.

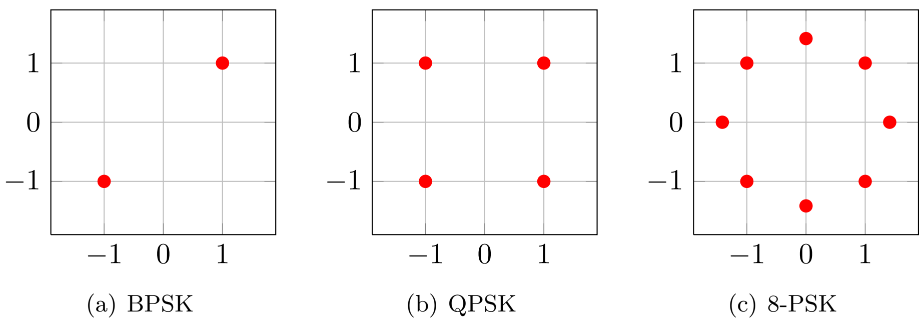


Figure 3.4: Three different PSK constellation diagrams

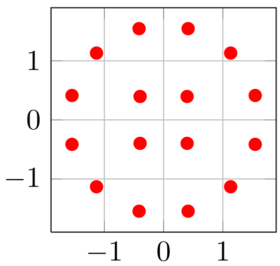
For every modulation, the amount of symbols used (n) and the amount of bits per symbol (b) are linked via the following, simple formula:

$$n = 2^b \tag{3.8}$$

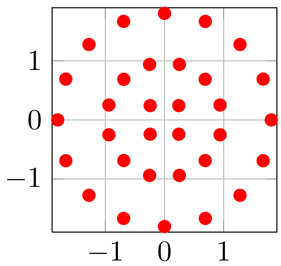
8-PSK for example uses three bits per symbol and has eight different symbols: $8 = 2^3$. The more bits are combined in one symbol, the more information can be transmitted; but with a growing number of symbols, the modulation grows more susceptible to channel impairments.

3.3.2 Amplitude and Phase-Shift Keying

A slightly more complex kind of modulation is Amplitude and Phase-Shift Keying (APSK). Here, not only the carrier's phase is changed, but also its amplitude. Therefore, in the constellation diagrams, the symbols form not only one, but two ore more circles around the origin. Figure 3.5 shows the two APSK-based modulations used in this work.



(a) 16-APSK



(b) 32-APSK

Figure 3.5: Two different APSK constellation diagrams

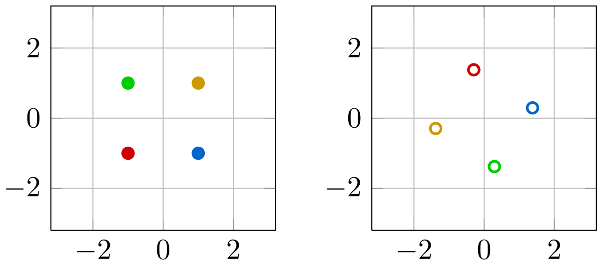
3.4 Channel Impairments

Different channel impairments were used and studied during the course of this work. These are introduced and explained briefly in the following section.

3.4.1 Carrier Phase Error

The simplest error is the carrier phase error. It occurs because the distance between sender and receiver is not a multiple of the carrier wavelength. Therefore, the received signal will always have an offset compared to the sent signal. Figure 3.6 illustrates the effect of this phenomenon.

Since this error affects all symbols in the same way, it is sufficient to estimate the error once; this could be done by using the algorithm presented in section 3.5. After this, all symbols are corrected by the same angle.

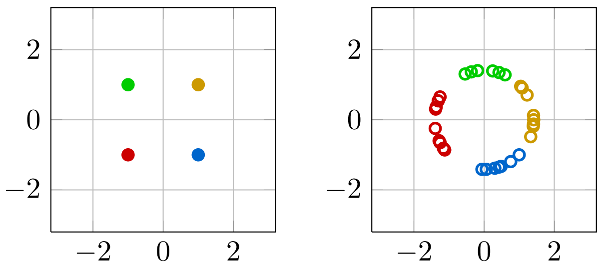


(a) Clean Symbols (b) Rotated Symbols

Figure 3.6: A QPSK-Signal with a Carrier Phase Error of $\theta = 147°$

3.4.2 Frequency Error

Frequency errors can be compared to phase errors; but instead of affecting all symbols in the same way, a frequency error rotates the whole constellation diagram continuously. As a consequence, the symbols slowly drift away from their ideal place and eventually overlap with other symbols. Figure 3.7 shows this effect.



(a) Clean Symbols (b) Symbols w/ Frq. Error

Figure 3.7: A QPSK-Signal with a Frequency Error of $\Delta f_T \approx -7 \cdot 10^{-3}$

A common source of frequency errors, especially in satellite-based communication, is a Doppler frequency shift. A different reason for frequency errors could be slightly different clocks at sender and receiver stations. Frequency errors are usually specified using the symbol Δf_T, which gives the rotation per symbol normalized to one full circle ($1 \mathrel{\widehat{=}} 2\pi$). Normal values for Δf_T are usually below 10^{-3}.

3.4.3 Signal-to-Noise Ratio

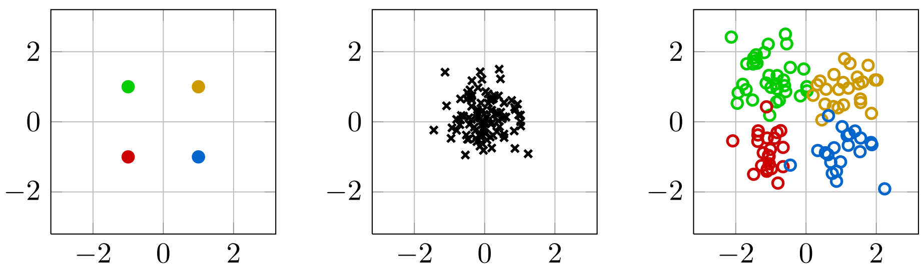
Signal-to-Noise Ratio (SNR) (ρ) is a measure for comparing the power level of a signal to the power level of the noise interfering with this signal. It is defined as the ratio of signal power to noise power:

$$\rho = \frac{P_S}{P_N} \tag{3.9}$$

Most of the time it is used in the logarithmic unit *decibel* (dB) [17]:

$$\text{SNR} = 10 \cdot log_{10}(\rho)\,[\text{dB}] \tag{3.10}$$

Figure 3.8 shows the effect of noise on a QPSK signal at an SNR of 9 dB. Most symbols can be related to their original position; for good performance, however, this SNR value seems to be the lower limit [8]. QPSK can still be used at lower SNR levels, but it needs to use redundancy in the form of better code rates.



(a) Clean Symbols (b) Additive Noise (c) Resulting Combination

Figure 3.8: The Effect of AWGN on a QPSK-Signal with SNR = 6 dB

For comparison, Figure 3.9 shows the more complicated constellation diagram of a 32-APSK signal; at 21 dB, the SNR is relatively low.

AWGN is a form of "wideband or *white* noise [which shows a] Gaussian distribution of amplitude" [10]. This noise $n(t)$ is then added to the signal $s(t)$ to create the received signal $r(t)$:

$$r(t) = s(t) + n(t) \tag{3.11}$$

Depending on the noise and signal power, different modulations and code rates may have to be used in order to achieve a low error rate. The SNR in Figure 3.9 is only 21 dB, much higher than the 6 dB in Figure 3.8; however, it shouldn't be much lower for this kind of modulation [8].

The Effect of AWGN

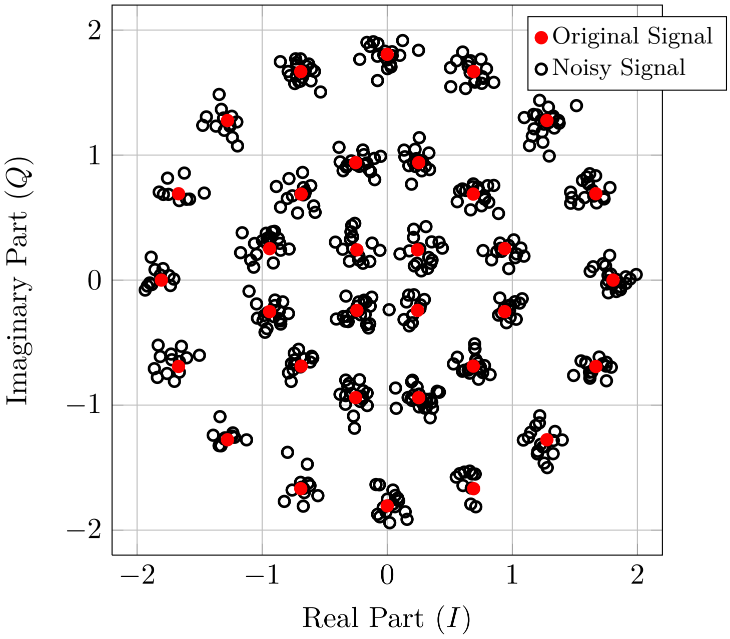


Figure 3.9: The effect of AWGN on a 32-APSK-signal with SNR = 21 dB

3.5 Phase Estimation

The following formula is used for phase estimation [6, Section 4.1]. p_k and r_k denote correct and received values, respectively; L denotes the estimator length, i.e. the number of symbols used for the estimation.

$$\hat{\theta}_{DA} = \arg\left(\sum_{k=0}^{P-1} p_k^* \cdot r_k\right) \tag{3.12}$$

Every received symbol is compared to its known counterpart and the angle differences are summed up. If no symbol was distorted, all imaginary parts – and with them the angles – would become 0. A slightly similar formula – expanded by a phase correction term – is used in subsection 3.6.1 for DA SNR estimation.

3.6 SNR Estimation

In SNR estimation, different stochastic moments – mostly of second and fourth order – are used [5, pp. 50–51].

3.6.1 Data-aided SNR Estimation

In data-aided SNR estimation, the correct values (p_k) of the received symbols (r_k) are known and used. If the expected symbol energy of the received and the reference symbols is 1, the following formula can be used for SNR estimation [6, Section 4.2]:

$$\hat{\rho}_{DA} = \frac{M_1^2}{M_2 - M_1^2} \tag{3.13}$$

where

$$M_1 = \frac{1}{L} \cdot \sum_{k=0}^{L-1} \mathrm{Re}\left(p_k^* \cdot r_k \cdot e^{-j \cdot \hat{\theta}_{DA}}\right) \tag{3.14}$$

$$M_2 = \frac{1}{L} \cdot \sum_{k=0}^{L-1} |r_k|^2 \tag{3.15}$$

In Equation 3.14, p_k^* denotes the complex conjugate of p_k and $e^{-j \cdot \hat{\theta}_{DA}}$ is a correction term for carrier phase error (see section 3.5).

3.6.2 Non-data-aided SNR Estimation for M-PSK

Like all estimation algorithms used in this paper, this estimator is moment-based; it is taken from [9, Equation 39].

$$\hat{\rho}_{NDA} = \frac{\sqrt{2 \cdot M_2^2 - M_4}}{M_2 - \sqrt{2 \cdot M_2^2 - M_4}} \tag{3.16}$$

with

$$M_2 = \frac{1}{L} \cdot \sum_{k=0}^{L-1} |r_k|^2 \tag{3.17}$$

$$M_4 = \frac{1}{L} \cdot \sum_{k=0}^{L-1} |r_k|^4 \tag{3.18}$$

M_2 and M_4 are the second- and fourth-order moments, r_k denotes the received symbols.

3.6.3 Non-data-aided SNR Estimation for 16-APSK

The calculation for 16-APSK is more difficult since the symbols can lay on one of two rings; therefore, not all symbols have the same magnitude. So, we have to calculate the boundary between the two rings and use only symbols on the outer ring for SNR estimation. The outer ring is used because it contains the majority ($\frac{3}{4}$) of all symbols and thus allows for a larger estimator length L [4, 5]:

First, the signal power has to be estimated using the formula

$$\hat{S} = \sqrt{\frac{2 \cdot M_2^2 - M_4}{2 - K_c}} \tag{3.19}$$

with the the symbol kurtosis K_c given by

$$K_c = \frac{1}{4} \cdot R_1^4 + \frac{3}{4} \cdot R_2^4. \tag{3.20}$$

M_2 and M_4 are defined above, R_1 and R_2 are defined in [1, Table 9].

Using the signal power, we can calculate the estimated partition radius between inner and outer ring:

$$\hat{R}_{12} = \frac{1}{2} \cdot \sqrt{\hat{S}} \cdot (R_1 + R_2) \tag{3.21}$$

After determining this partition radius, we discard all symbols with too low magnitude:

$$|r_k| > \hat{R}_{12} : z_k = r_k \tag{3.22}$$

Using only the remaining symbols z_k, we recalculate the second- and fourth-order moments M_2' and M_4' and use them to estimate the SNR:

$$M_2' = \frac{1}{L'} \cdot \sum_{k=0}^{L'-1} |z_k|^2 \tag{3.23}$$

$$M_4' = \frac{1}{L'} \cdot \sum_{k=0}^{L'-1} |z_k|^4 \tag{3.24}$$

$$\hat{\rho}_{NDA} = \frac{1}{R_2^2} \cdot \frac{\sqrt{2 \cdot M'^2_2 - M'_4}}{M'_2 - \sqrt{2 \cdot M'^2_2 - M'_4}} \tag{3.25}$$

The division factor $\frac{1}{R_2^2}$ is to scale the value to all symbols, not only those from the outer circle.

3.6.4 Non-data-aided SNR Estimation for 32-APSK

The method for 32-APSK resembles very much the method for 16-APSK [5]. The symbol kurtosis is now defined as

$$K_c = \frac{1}{8} \cdot R_1^4 + \frac{3}{8} \cdot R_2^4 + \frac{4}{8} \cdot R_3^4. \tag{3.26}$$

Furthermore, the partition radius lays now between middle and outermost ring and is now given by

$$\hat{R}_{12} = \frac{1}{2} \cdot \sqrt{\hat{S}} \cdot (R_2 + R_3) \,. \tag{3.27}$$

Finally, the multiplicative factor $\frac{1}{R_2^2}$ changes to $\frac{1}{R_3^2}$ and the formula for the SNR estimation is now

$$\hat{\rho}_{NDA} = \frac{1}{R_3^2} \cdot \frac{\sqrt{2 \cdot M'^2_2 - M'_4}}{M'_2 - \sqrt{2 \cdot M'^2_2 - M'_4}}. \tag{3.28}$$

All other formulas remain as described in subsection 3.6.3.

Chapter 4

Implementation

This chapter will discuss the general program structure and how the general principles described in the previous chapter were implemented. For selected parts an overview will be given how the correct solution was reached.

4.1 Setting up the Work Environment and Compiling the Simulator

The initial task of setting up the work environment and making the provided code compile proved to be quite intricate. Several different components were necessary, among them libraries (*Intel Math Kernel Library*, *Intel Integrated Performance Primitives*), IDE (*Eclipse for C++*) and compiler (*MinGW*). Additionally, some changes had to be made to the provided source code before being able to compile it.

Using Linux

The first problem was encountered though the attempt to use a Linux-based operating system. Although the code was originally planned to be platform-independent, several proprietary libraries had been included. These libraries could not trivially be replaced by open libraries, so the project was finally switched to a different machine using Windows XP.

Prerequisites and Additional Software

After several failed attempts of setting up the work environment, a software installation and maintenance guide was provided [7]. The guide specified the exact software versions and installation sequence to be used and made an easier set-up possible.

Project Set-up

Since a slightly outdated copy of the simulator code was provided, the project did not compile initially. Also, it was not possible to obtain a copy of an working Eclipse workspace from one of the other productive systems, so several project settings had to be tried out. Amongst other things, the MinGW compiler was replaced by the Eclipse "internal compiler" due to compilation problems. This internal compiler was used on other machines as well, as was discovered later.

Necessary Source Code Modifications

Finally, some changes were made to the source code under the supervision of two authors of the simulator source code:

- In the file `utils.h` the line `using namespace std;` was added.
- In the file `utils.h` the line `#include <complex.h>` was changed to `#include <complex>`.
- In the file `utils.h` the line `#include <stdio.h>` was added.
- In the file `acquisition.cpp` (line 1267) the `if` statement is removed because the constant `SHRT_MAX` is not defined (file `limits.h` is missing).
- In the file `acquisition.cpp` (line 1634) the constant `SHRT_MAX` is replaced by the numeric value `32767` for the same reason.

4.2 Operating the Simulator

This section describes how the simulator used in this work is operated and what additional measures were taken in order to facilitate automatic multiple-sweep simulation.

4.2.1 Simfiles

The simulator is operated using so-called *simfiles* (**sim**ulation configuration **files**) which contain all necessary data and parameters for a given simulation run. A typical command line invocation could be:

```
C:\modemsim\simulator\Release> simulator.exe simfile.sim
```

A typical simfile is explained in subsection 4.2.3.

4.2.2 x1, x2: Sweep Support

A practical feature supported by the simulator is sweep support. Using the keywords **x1** and **x2**, one can specify up to two parameters to be varied during the simulation in order to obtain a set of curves. A typical usage would be the following:

```
MODULE FrequencyError
x1 delta_fT             -1.5 1 0.5
```

It can be used with any parameter of any module by simply writing it in front of the parameter. Additionally, the normal, single value is replaced by the three values for `min_val`, `max_val`, and `step_size`. `min_val` always has to be smaller than `max_value`. The example given above would yield the values $\{-1.5, -1, -.5, 0, .5, 1\}$. On a side note: **x2** can be used before **x1**, it only seems to affect the execution order and with it the grouping of the results in the output files.

Though the support for simulation sweeps is very usable, the current implementation still has some limitations. Firstly, it is not possible to skip certain values; this was bypassed by writing a new simulator module (see subsection 4.6.3) and by using batch files (see subsection 4.2.4). Secondly, only linear sweeps are supported; this, too, was handled with the new simulator module. Lastly, the implementation is limited to two sweep variables, which was answered by using batch files for simulation.

4.2.3 A Typical Simfile

All simulation files have to obey certain guidelines in order to be correct. Listing 4.1 shows a typical sim-file used to configure the simulator. This file was used for the simulations concerning the influence of phase error correction, presented in section 5.1.

Listing 4.1: A typical example for a simulation file

```
MODULE SimulationControl //must be first entry!!!
   // Bytes (normal=8100; short=2025)
   burstLen                    10
   loopsPerPoint               1e7
   loopsPerIntermediateResult 1e7
   // mapping 1=1;12=1/2;23=2/3;34=3/4;45=4/5;78=7/8;89=8/9
   codeRate                    12
   // 1=BPSK; 2=QPSK; 3=8-PSK; 4=16-APSK;
   modulation                  2

// Sender --------------------------------------------------

MODULE DataGeneratorBytes
```

```
   // 0=RANDOM; 1=ZERO_BURST; 2=DIRAC_BURST
   mode                    0

MODULE ByteToBit

TESTPOINT                  1

MODULE MapperBitToSymbol

MODULE InsertPLFrame
   active                  1
   insert_pilots           0 // 0=no pilots
   frame_size              1 // 0=normal; 1=short

TESTPOINT                  2

// Channel ————————————————————————————————————————————

MODULE AWGN
x1 snr                    −5 1 1

MODULE ConverterCartesianToPolar
MODULE RandomPhaseOffset
MODULE ConverterPolarToCartesian

// Receiver ———————————————————————————————————————————

TESTPOINT                  3

MODULE RemovePLFrame
   active                  1
   phase_correction        1

MODULE MapperSymbolToSoftBit

MODULE MapperSoftBitToHardBit

TESTPOINT                  4

MODULE AnalyzePLFrame
   tp1                     2
   tp2                     3
   phase_correction        1

MODULE Analyzer_SER
   tp1                     1
   tp2                     4

//End of Simulation file ***
```

Lines 1–9 show the simulation's basic parameters, like the amount of data to be processed, and the code rate and modulation to be used. Furthermore, it is possible to run a simulation several times to achieve statistically correct results. The amount of loops is specified in the parameter `loopsPerPoint`.

Lines 13–28 show the sender-side preparations for sending the data: Random data creation, modulation, PLFRAME insertion, and test points.

Lines 32–37 show the channel with the two impairments AWGN and phase error. Lines 35 and 37 are for data conversion.

Finally, lines 41–60 show the receiver side of the simulation chain: Test points, PLFRAME removal, demodulation and analyses.

4.2.4 Batch Files for Simulation Automation

In order to facilitate simulation sweep with more than two variables and for allowing automatic simulation runs during the night and weekend, two batch files were created. the first file, `all_sims.bat`, executes all simfiles in the current directory, saves the simulator output to a file and renames all created files depending on the simfile's file name. The second file, `go.bat`, calls the first file using a low priority in order to allow normal working while the simulations are running. Both files can be seen in Appendix B.

4.3 Program Structure

During this work, much functionality was outsourced into a base class called `ETSI_Baseclass` from which nearly all modules created are derived. Additionally, modules have to be derived from the class `Module` in order to be handled correctly by the simulator main structures. Figure 4.1 shows the dependencies between the different classes discussed in this section:

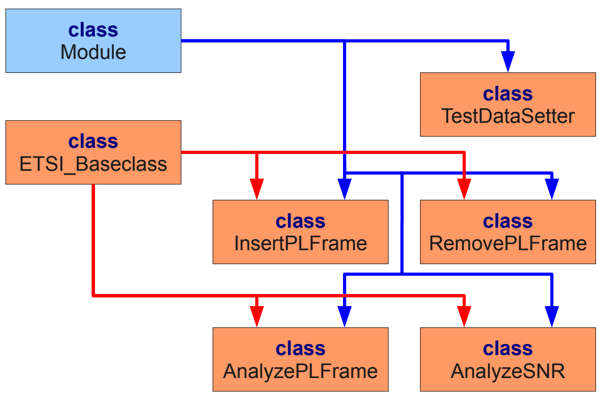


Figure 4.1: Class dependency diagram. Arrows symbolize inheritance.

Red arrows symbolize **`protected`** inheritance relationships with the class `ETSI_Baseclass`, while blue arrows symbolize **`public`** relationships with the class `Module`. Blue items are old parts of the simulator, red items were created during the course of this project.

4.4 PLFRAME Handling

This section describes the classes used for PLFRAME insertion and removal. Some functionality used by these classes is also found in section 4.6.

4.4.1 PLFRAME Insertion

```
class InsertPLFrame: public Module, protected
    ETSI_Baseclass
```

The main program sequence is found in this classes `execute()` method. In the beginning, the PLS code is generated from the meta data given in the current burst's header. Subsequently, the entire PLHEADER is generated, modulated, and inserted in the current burst container. Finally, the pilot blocks are inserted at the appropriate places if the corresponding option is set.

A higher-level discussion of the underlying ETSI-standardized process can be found in section 3.1. The corresponding source code is given in Listings A.5 and A.6.

Module Usage

This module has the following parameters:

- `active`: If set to 0, this module doesn't change the current burst, just like if it was commented out. Mandatory parameter.
- `insert_pilots`: Specifies whether pilot blocks should be included. 0 means no pilots are included. Mandatory parameter.
- `frame_size`: Specifies the PLFRAME's frame size. 0 means normal frame size (64 800 Bit), 1 means short frame size (16 200 Bit). Mandatory parameter.

All other parameters necessary for PLFRAME generation are taken from the current burst's header data. A correct usage of this module could look like this:

Listing 4.2: Sample usage for InsertPLFrame

```
MODULE InsertPLFrame
   active             1
   frame_size         0 // 0=normal; 1=short
   insert_pilots      1 // 0=no pilots; 1=insert pilots
```

Finally, this module's interface mode is `IM_SYMBOL_CARTESIAN_FLOAT`.

4.4.2 PLFRAME Removal

```
class RemovePLFrame: public Module, protected
    ETSI_Baseclass
```

This class removes an existing PLHEADER. Additionally, it decodes the header's PLS code field and removes the pilot blocks if necessary. No analyses or other checks are made, this class is just for simple removal of PLFRAME overhead parts. Again, the main program sequence can be found in the `execute()` method.

The corresponding source code is given in Listings A.7 and A.8. Since many program parts are used by the `AnalyzePLFrame` class, much source code was outsourced into the `ETSI_Baseclass` class which is discussed in subsection 4.6.2.

Module Usage

This module has the following parameters:

- `[active]`: If set to 0, this module doesn't change the current burst, just like if it was commented out. Optional parameter, default value is 0.
- `[phase_correction]`: Specifies whether phase error correction is to be used. 0 means no correction is used. Optional parameter, default value is 0.

A correct usage of this module could look like this:

Listing 4.3: Sample usage for RemovePLFrame

```
MODULE RemovePLFrame
   active             1
   phase_correction   0 // 0=no correction
```

This module's interface mode is `IM_SYMBOL_CARTESIAN_FLOAT`.

4.5 Analysis

This section presents the modules used for different kinds of analyses. One goal of theses analyses was to ensure the correct functioning of the other modules; the other goal was to conduct the investigations described in chapter 5.

4.5.1 Analyze SNR

```
class AnalyzeSNR: public Module, protected ETSI_Baseclass
```

This module contains the different SNR estimation methods implemented during the course of this work. Firstly, it determines the amount of usable data based on the estimation method to be used. Secondly, it copies the usable data into a local array and calls the method corresponding to the chosen estimation method; those estimation methods are implemented in the `ETSI_Baseclass` class. If `plframe_used=0`, all data will be used; otherwise only overhead or non-overhead parts will be used, depending on the method chosen.

Module Usage

This module has the following parameters:

- `data_aided`: Specifies whether DA or NDA SNR estimation algorithms should be used. Mandatory parameter.
- `plframe_used`: Specifies whether the data contains PLFRAME-related overhead data (PLHEADER and pilot blocks). 0 means no plframe is used. Mandatory parameter.
- `[reference_tp]`: Test point for reference data used for DA estimation. Mandatory if `data_aided=true`, not used otherwise. Should be immediately in front of the channel.
- `[phase_correction]`: Specifies whether phase error correction is to be used. 0 means no correction is used. Mandatory if `data_aided=true`, not used otherwise.

A correct usage of this module could look like Listing 4.4:

Listing 4.4: Sample usage for `AnalyzeSNR`

```
[...]

TESTPOINT                1

// Channel ------------------------------------------------

[...]

// Receiver -----------------------------------------------

MODULE AnalyzeSNR
   data_aided            1
   reference_tp          1
   plframe_used          0
   phase_correction      0

[...]
```

This module's interface mode is `IM_ANY`.

4.5.2 Analyze PLFRAME

```
class AnalyzePLFrame: public Module, protected
    ETSI_Baseclass
```

This module compares the PLFRAME header data from two different test points in the transmission chain. It decodes both PLS code fields and compares the four parameters modulation, code rate, frame size, and whether pilots were used.

If one or more fields differ, the frame is counted as erroneous. It, too, uses code from the `ETSI_Baseclass` class which is discussed in subsection 4.6.2.

Module Usage

This module has the following parameters:

- `tp1`: First test point. Should be after a PLFRAME was included, and immediately in front of the channel. Mandatory parameter.
- `tp2`: Second test point. Should be immediately after the channel. Mandatory parameter.
- `[phase_correction]`: Specifies whether phase error correction is to be used. 0 means no correction is used. Optional parameter, default value is 0.

A correct usage of this module could look like this:

Listing 4.5: Sample usage for `AnalyzePLFrame`

```
[...]

TESTPOINT                1

// Channel ------------------------------------------------

[...]

// Receiver -----------------------------------------------

TESTPOINT                2

[...]

MODULE AnalyzePLFrame
   tp1                   1
   tp2                   2
   phase_correction      1 // 0=no correction

[...]
```

This module's interface mode is `IM_ANY`.

4.6 Other Code

This section discusses the code parts which is not related to a specific task or which is used in several modules.

4.6.1 Defines and Debug Code

```
ETSI_common.h
ETSI_common.cpp
```

These files contain several constants and definitions used throughout the entire code. Examples are the table used for PLS code demodulation (see subsection 3.2.3) or constants defined in [1] like the SOF field (see subsection 3.1.2) and the constant value used for scrambling the PLHEADER (see subsection 3.1.3).

4.6.2 Base Class

```
class ETSI_Baseclass
```

This base class is not to be instantiated directly; it contains code that the other module classes have in common. Methods include the following application fields:

- SNR estimation
- Phase error estimation and correction
- PLS code encoding
- PLS code decoding
- PLHEADER data access

4.6.3 Testdata Setter

```
class TestDataSetter: public Module
```

The simulator supports automated sweeps over one or two simulation parameters using the **x1** and **x2** keywords. This method, however, only allows linear sweeps of constant step size. The module `TestDataSetter` was written to allow a logarithmic sweep over the `burstLen` parameter. It replaces the parameter value according to the following formula:

$$\text{burstLen} = 10^{\frac{\text{burstLen}}{2}} \tag{4.1}$$

Additionally, `burstLen` is scaled depending on the modulation used, then converted to bytes. Thus, there will always be the same amount of symbols after modulation, independent of the modulation method used:

$$\text{burstLen} = \frac{\text{burstLen} \cdot \text{scalingFactor}}{8} \tag{4.2}$$

In order to facilitate easy sweeps over all five modulations used in this work, the `modulation` parameter value 5 is interpreted as 32-APSK. Normally, 32-APSK would be represented by 6, while 5 stands for 16-PSK. To guarantee compatibility with other modules, this redefinition is corrected after the `burstLen` calculations.

Table 4.1 shows the parameter values before and after `TestDataSetter`.

Table 4.1: Parameter values before and after `TestDataSetter`

`modulation`		`burstLen`	Symbols after modulation
Before	After		
1	1	1	3
2	2	2	10
3	3	3	32
4	4	4	100
5	**6**	5	316
		6	1 000
		7	3 162
		8	10 000

Module Usage

This module has no parameters to set. A correct usage could look like this:

Listing 4.6: Sample usage for `TestDataSetter`

```
MODULE SimulationControl //must be first entry!!!
x1 burstLen                1 8 1
x2 modulation              2 5 1

[...]

// Sender -------------------------------------------------

MODULE TestDataSetter

[...]
```

This module's interface mode is `IM_ANY`.

Chapter 5

Results

After implementing the functionality described in chapter 4, several experiments were conducted; their goal was to test the implemented components and learn more about the applicability of the underlying mathematical concepts.

One step was to study the components for testing and correcting channel impairments, such as carrier phase offset or channel noise. The other step consisted in testing the robustness of the PLFRAME structure under the same channel impairments.

In the following sections, the simulation results will be explained. Among other things, the detailed simulation configurations used for each simulation will be given; for easier understanding we will use figures similar to Figure 3.2. Like in this figure, red components were developed during this work, and blue components were already part of the simulator. Additionally, the components whose parameters were varied during the simulation are encircled with a red circle. Figure 5.1 demonstrates these symbols:

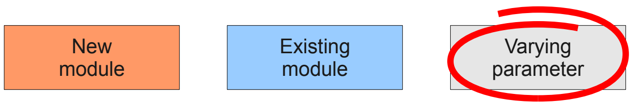


Figure 5.1: Transmission path symbol explanation

5.1 Carrier Phase Error Estimation

Since phase errors are a very basic channel impairment, a phase error estimation and correction module was tested first.

5.1.1 Simulation Set-up

Figure 5.2 shows the simulation set-up used. The coderate used was $R = \frac{1}{2}$, the modulation was QPSK. Only the PLHEADER was used in this simulation, so a very small payload of only 10 Byte could be used to increase simulation speed.

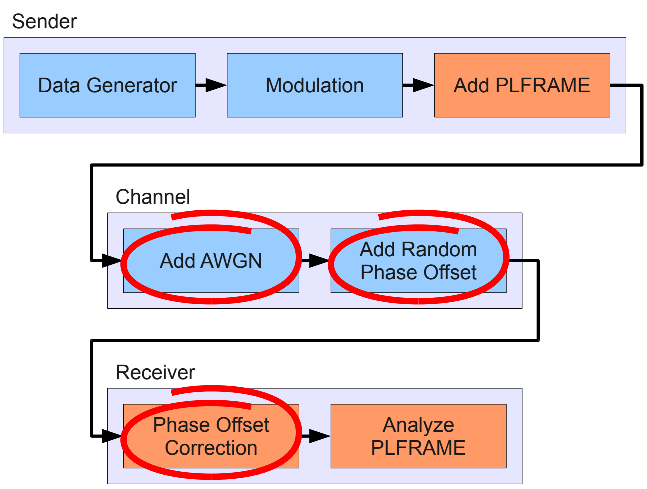


Figure 5.2: Transmission path for the carrier phase error correction

After phase correction, the frame's PLHEADER was decoded. If the recovered values for the MODCOD and TYPE fields weren't correct, the frame was dropped and counted as frame error.

5.1.2 Simulation Results

Figure 5.3 shows the performance of the carrier phase noise correction module: Only a small increase of the PLFRAME error rate ϵ can be observed.

During the simulation, $\epsilon = 0$ was reached at higher SNR levels; the lines are drawn up to this point. Table 5.1 shows the approximate minimum SNR values to reach a PLFRAME error rate of 10^{-6} or less:

Table 5.1: Min. SNR values for phase correction

Mode	SNR value
Without phase noise	-2.5
With corrected phase noise	-2

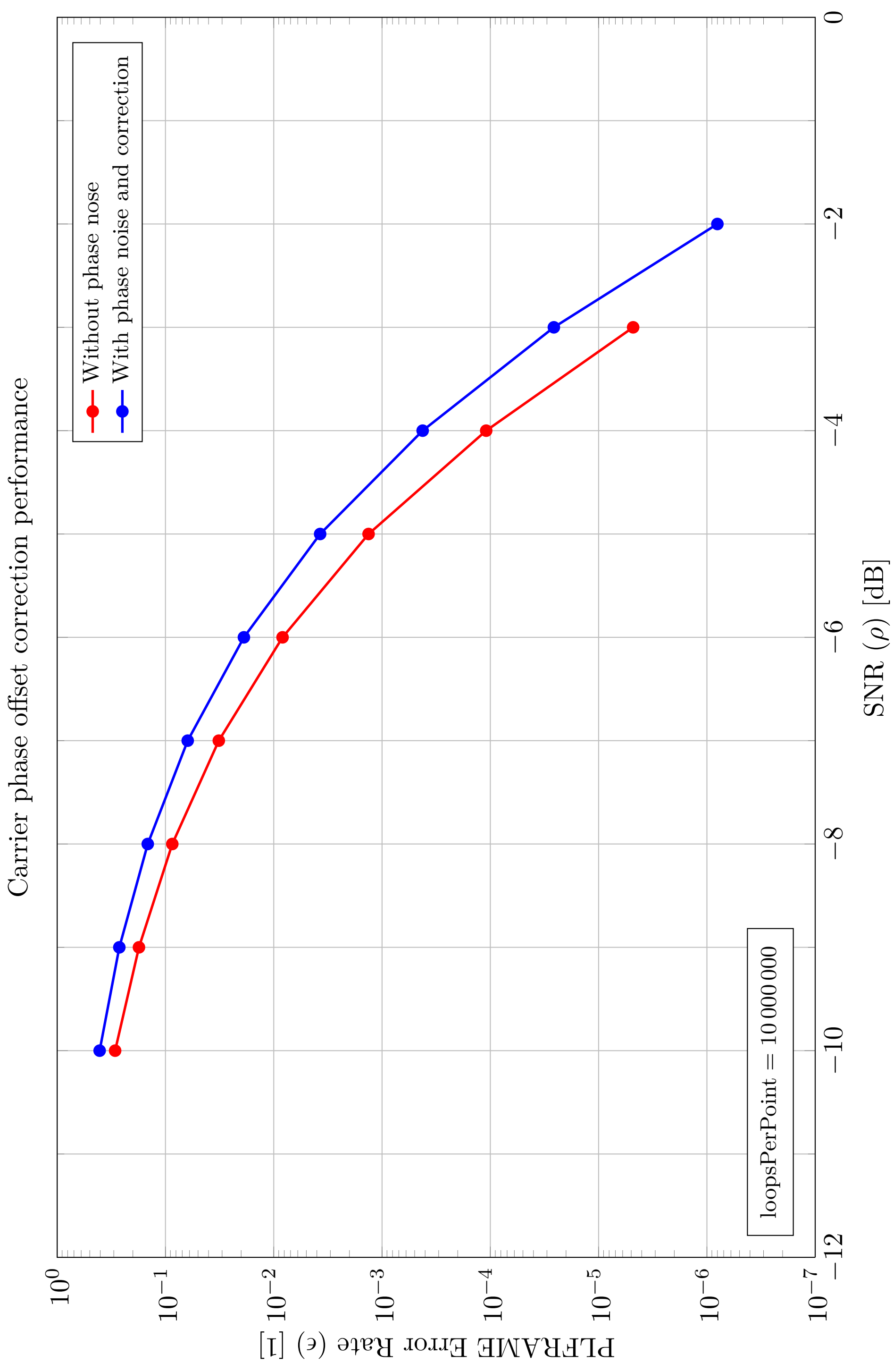


Figure 5.3: Carrier phase offset correction performance

5.2 SNR Estimation

The main part of this work revolves around SNR estimation. Several different algorithms for SNR estimation have been implemented [5, 6, 9]. These algorithms are tested in this section by applying several different channel impairments; the impairments considered were carrier phase offset, AWGN, and frequency errors.

The simulation configurations varied depending on the different analyses made and will be presented in each one of the following sections. The configuration used for Figures 5.5 to 5.7 is depicted in Figure 5.4:

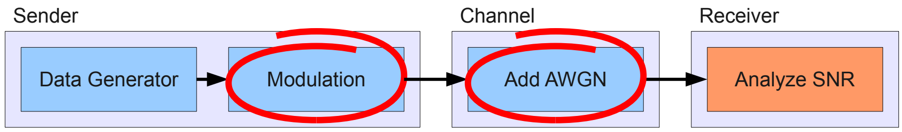


Figure 5.4: Transmission path for the basic SNR estimator analyses

In Figures 5.5 to 5.17 the Cramér-Rao Lower Bound (CRLB) is included which is the lower estimation bound for the given SNR ρ and estimator length L [11]. The Normalized CRLB (NCRLB) is calculated as follows:

$$\mathrm{NCRLB} = \frac{\mathrm{CRLB}}{\rho^2} = \frac{1}{L} \cdot \left(1 + \frac{2}{\rho}\right) \tag{5.1}$$

5.2.1 Estimator Length

The performance of SNR estimators depends strongly on the amount of symbols used for estimation; this value is called *estimator length*. For further use, all possible estimator lengths for PLFRAMES are provided in Table 5.2. If no pilot blocks are used, the estimator length for data-aided SNR estimation is always the length of the PLHEADER, i.e. 90 Symbols.

A quick and easy way to determine the amount of pilot blocks for a given amount of payload data is to divide the data length by $(90 \cdot 16) = 1440$ and round down the result:

$$\#\text{ of Pilots} = \left\lfloor \frac{\#\text{ of Payload Symbols}}{90 \cdot 16} \right\rfloor \tag{5.2}$$

If this yields a whole number, we have to subtract 1, since pilot blocks at the very end of the frame are not transmitted (see [1, Chapter 5.5.3]).

Section 5.3 elaborates further on the influence of estimator length on the performance of SNR estimation.

Table 5.2: Possible estimator lengths and numbers of pilot blocks

Modulation	Frame Size	# Pilots	Estimator Length [Symbols]	
			L_{DA}	L_{NDA}
QPSK	short	5	270	8 100
	normal	22	882	32 400
8-PSK	short	3	198	5 400
	normal	14	594	21 600
16-APSK	short	2	162	4 050
	normal	11	486	16 200
32-APSK	short	2	162	3 240
	normal	8	378	12 960

5.2.2 General Behaviour of SNR Estimation Algorithms

Figures 5.5 and 5.6 compare the estimation performance for different modulations – and therefore different estimation methods. Figure 5.5 compares the estimated SNR to the true value. 16- and 32-APSK show a deviation from the ideal line for values below 15 dB; the other estimation methods do very well. Figure 5.6 compares the MSE of the different methods to the theoretical limit, the CRLB.

Evidently, SNR estimation works better for modulations using less possible symbol values, like QPSK. Clearly, the data-aided (DA) method works best, yielding estimates very close to the theoretical limit.

On the other hand, the DA method is not a bias-free estimation method; even though its MSE may be very low, the estimation result ($\hat{\rho}$) will always show a slight offset. This behaviour is illustrated in Figure 5.7.

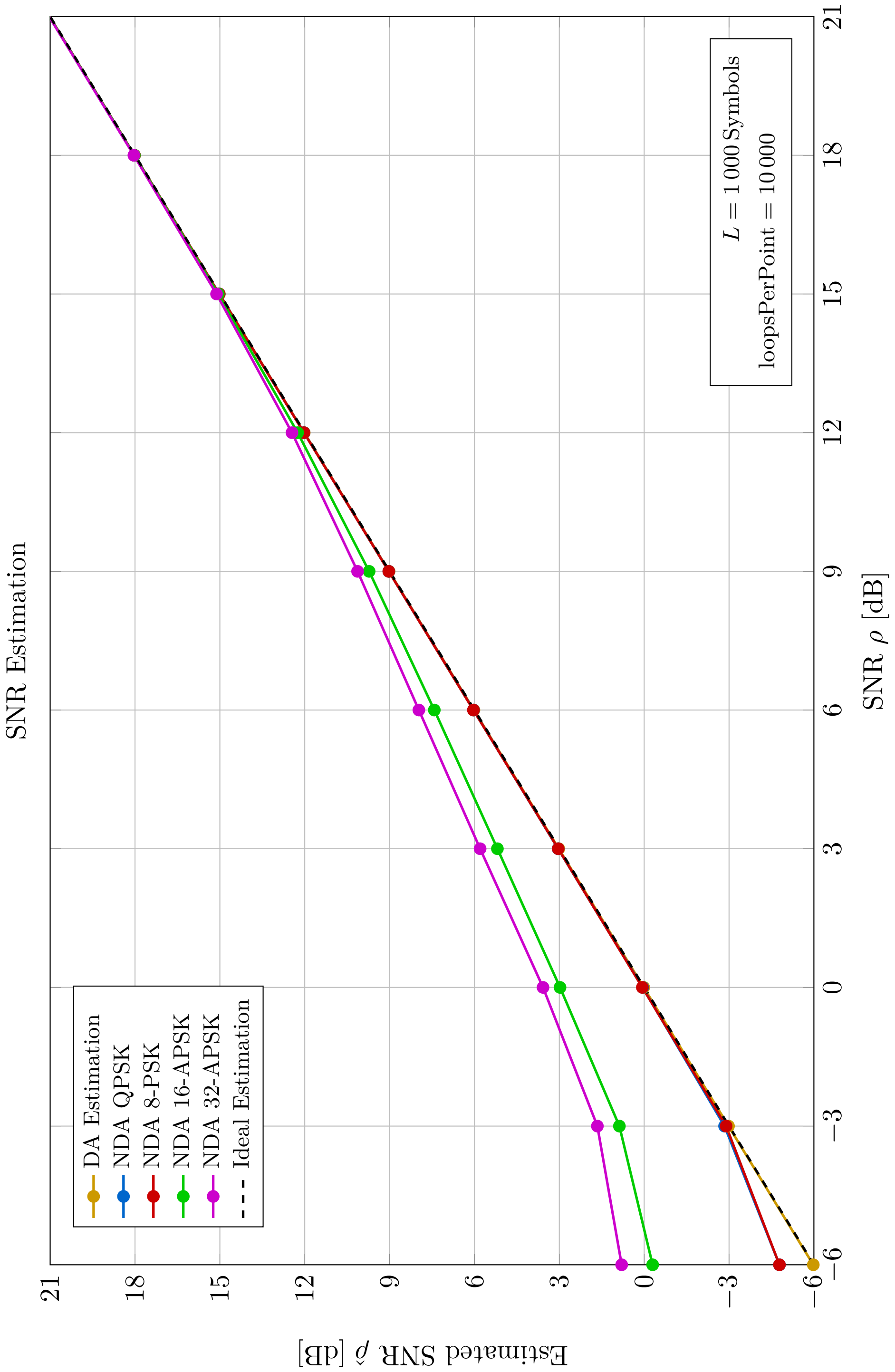


Figure 5.5: Estimated SNR $\hat{\rho}$ vs. SNR ρ

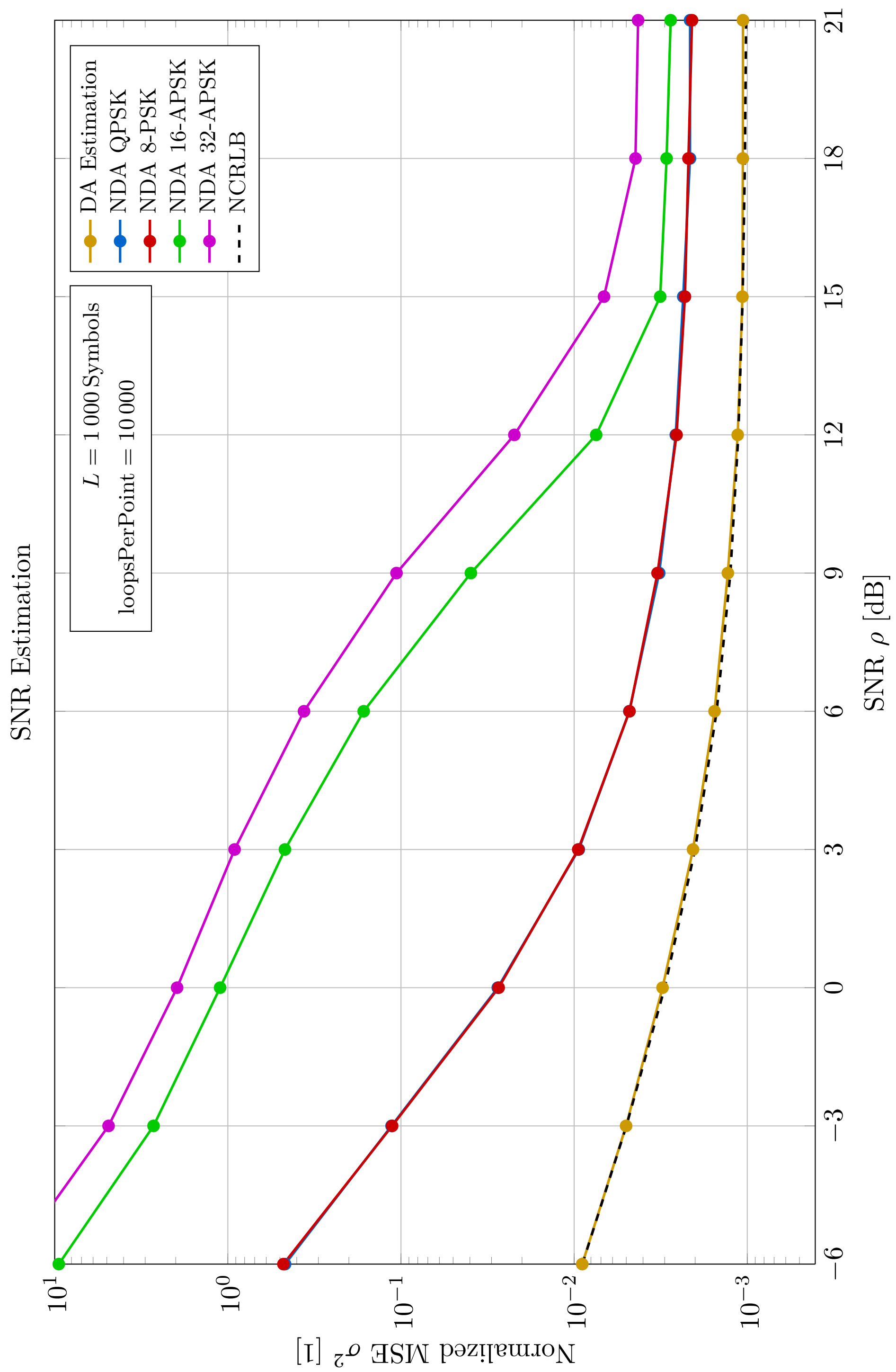


Figure 5.6: Normalized Mean Square Error σ^2 vs. SNR ρ

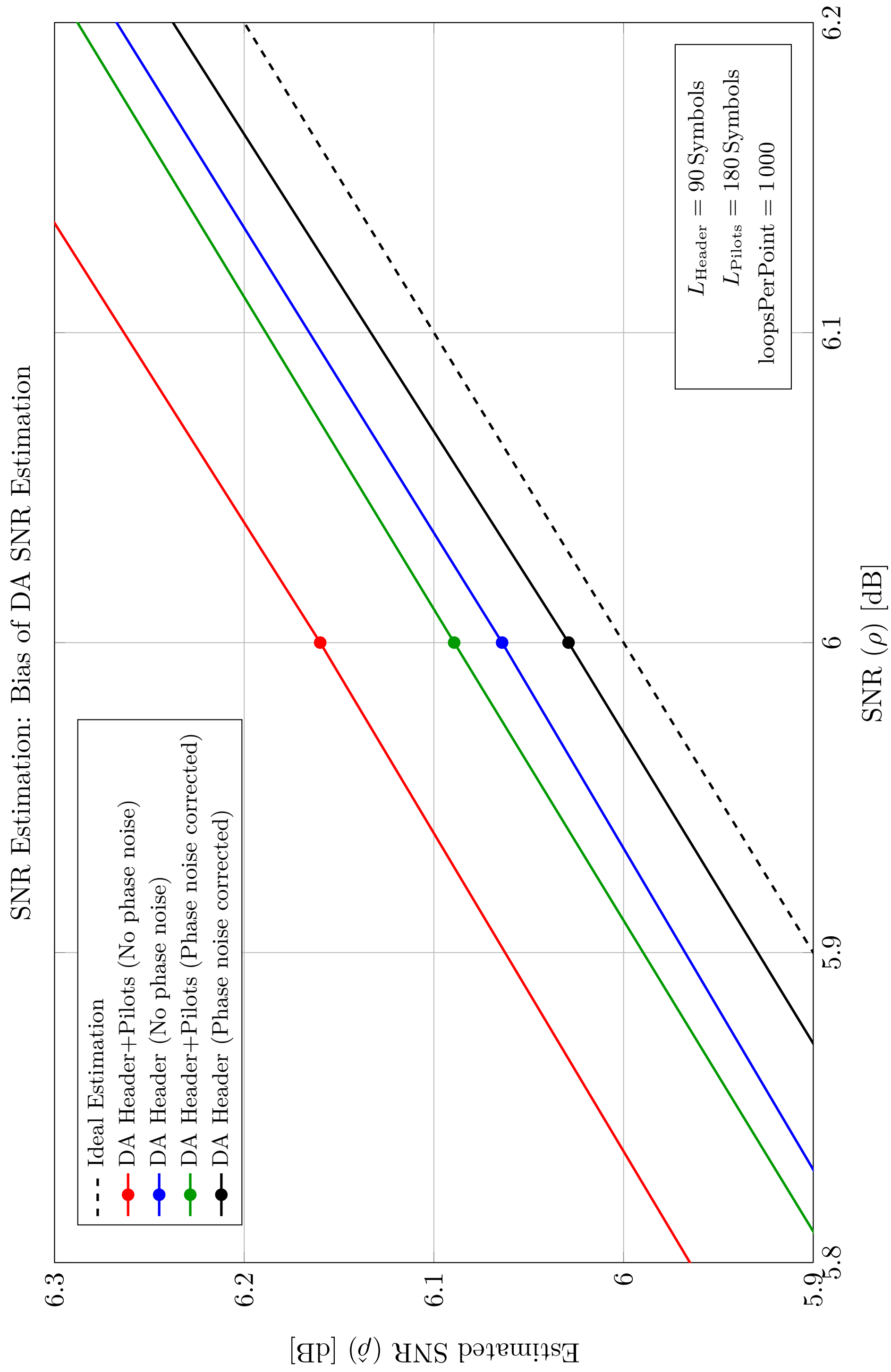


Figure 5.7: SNR Estimation: Bias of DA SNR Estimation

5.3 The Influence of Estimator Length

The estimator performance depends on the amount of data used for the estimation, i.e. the *estimator length*. Therefore, the influence of estimation length is of interest for this application. For some general observations about estimator length please refer to subsection 5.2.1.

5.3.1 Simulation Set-up

The simulation set-up involved few components; nevertheless, it was quite complex as it uses several different indexed variables: The data length, modulation, and SNR all were varied. The coderate used was $R = \frac{3}{4}$; all other simulation parameters can be seen in Figures 5.9 to 5.12. Figure 5.8 shows the simulation configuration used:

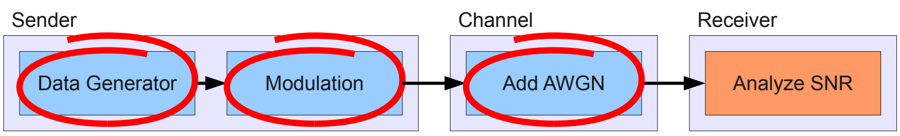


Figure 5.8: Transmission Path for the Influence of Estimator Length

5.3.2 Simulation Results

In Figures 5.9 to 5.12, the MSE of different estimators is compared to the CRLB for varying estimator lengths and for different SNR values.

The most important observation is that estimator length has a strong influence on the accuracy of the estimate. DA estimation always exhibits a MSE close to the theoretical limit, even for the lowest tested estimator length, three symbols. For NDA estimation using modulations with constant envelope like QPSK or 8-PSK the minimum estimator length seems to lay around 10 symbols. The more complex modulations examined – 16- and 32-APSK – reach their operation range at around 100 symbols. The following Table 5.3 shows the approximate minimum estimator lengths L to reach an MSE of 10^{-1} or less:

However, one has to bear in mind that even estimates close to the theoretical minimum are quite bad for shorter estimator lengths. For the SNR values regarded, the NCRLB line falls below 10^{-2} only around $L = 300$. Another effect to consider when evaluating estimation performance is estimator bias which is not visible in diagrams showing the MSE oder standard deviation.

An interesting observation is the bad performance of the APSK estimators in Figure 5.11, compared to the rather good performance in Fig-

Table 5.3: Min. Estimator Lengths for $MSE \leq 10^{-1}$

Modulation	$-3\,\text{dB}$	$3\,\text{dB}$	$7\,\text{dB}$	$15\,\text{dB}$
BPSK	50	20	13	10
QPSK	1 000	100	50	33
8-PSK	-	100	50	33
16-APSK	-	-	2 000	50
32-APSK	-	-	-	80

ure 5.12. By taking a look at Figures 5.16 and 5.17 from the next section we can confirm this observation. Evidently, the best range of application for APSK-based estimators is 15 dB and above; for lower values, estimator performance quickly decreases. Note that in Figure 5.12 the saturation effect starts much later and takes place on a much lower level.

For PLFRAME applications, we can draw the main conclusion that data length is not an issue. According to Table 5.2, the shortest estimator length for NDA SNR estimation is 3 240 Symbols, which is quite high compared to the data lengths tested here. At lengths like these, the main concern lies on factors like SNR, estimator bias, or noise.

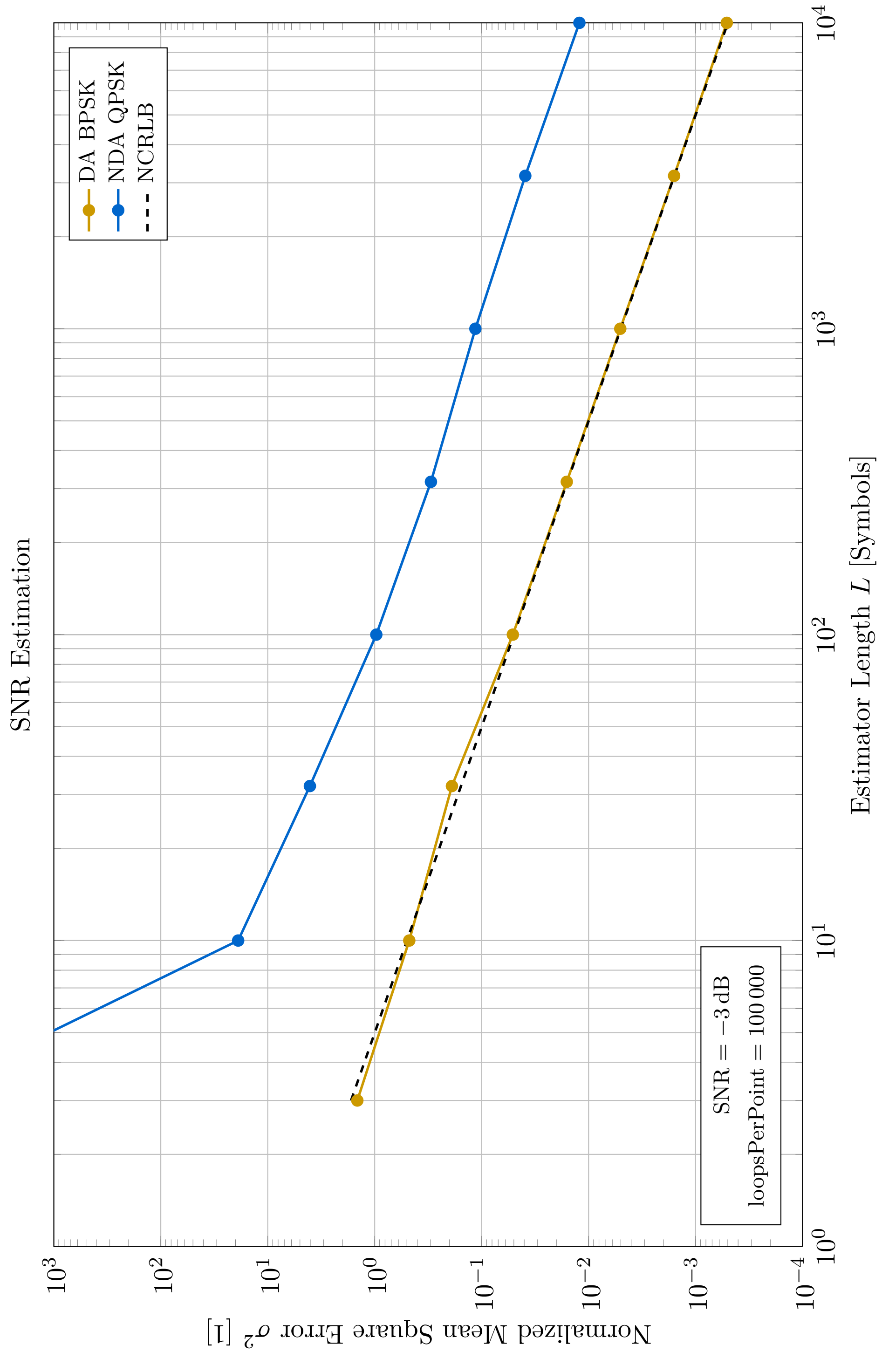


Figure 5.9: SNR $\rho = -3\,\mathrm{dB}$

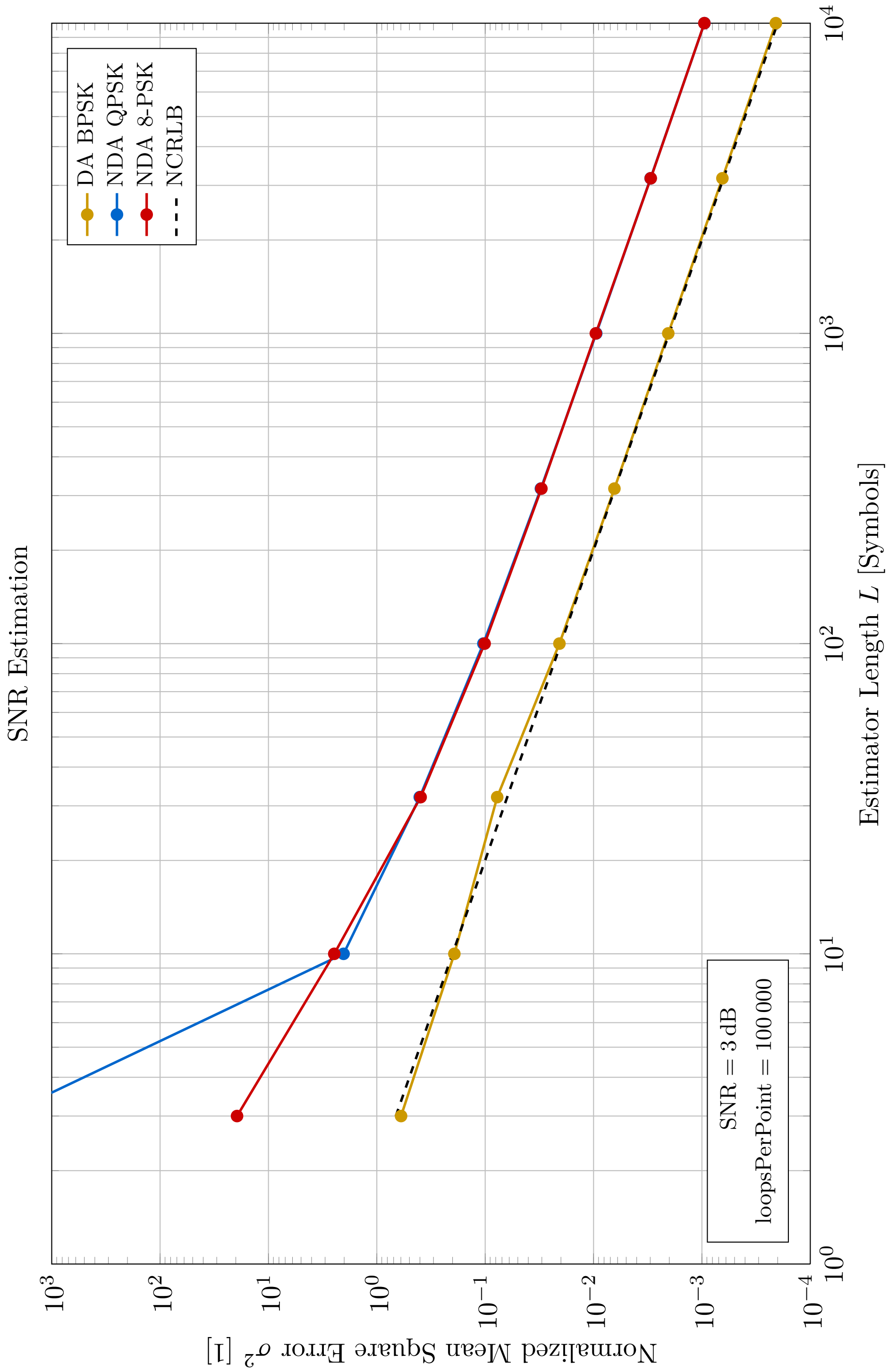


Figure 5.10: SNR $\rho = 3\,\text{dB}$

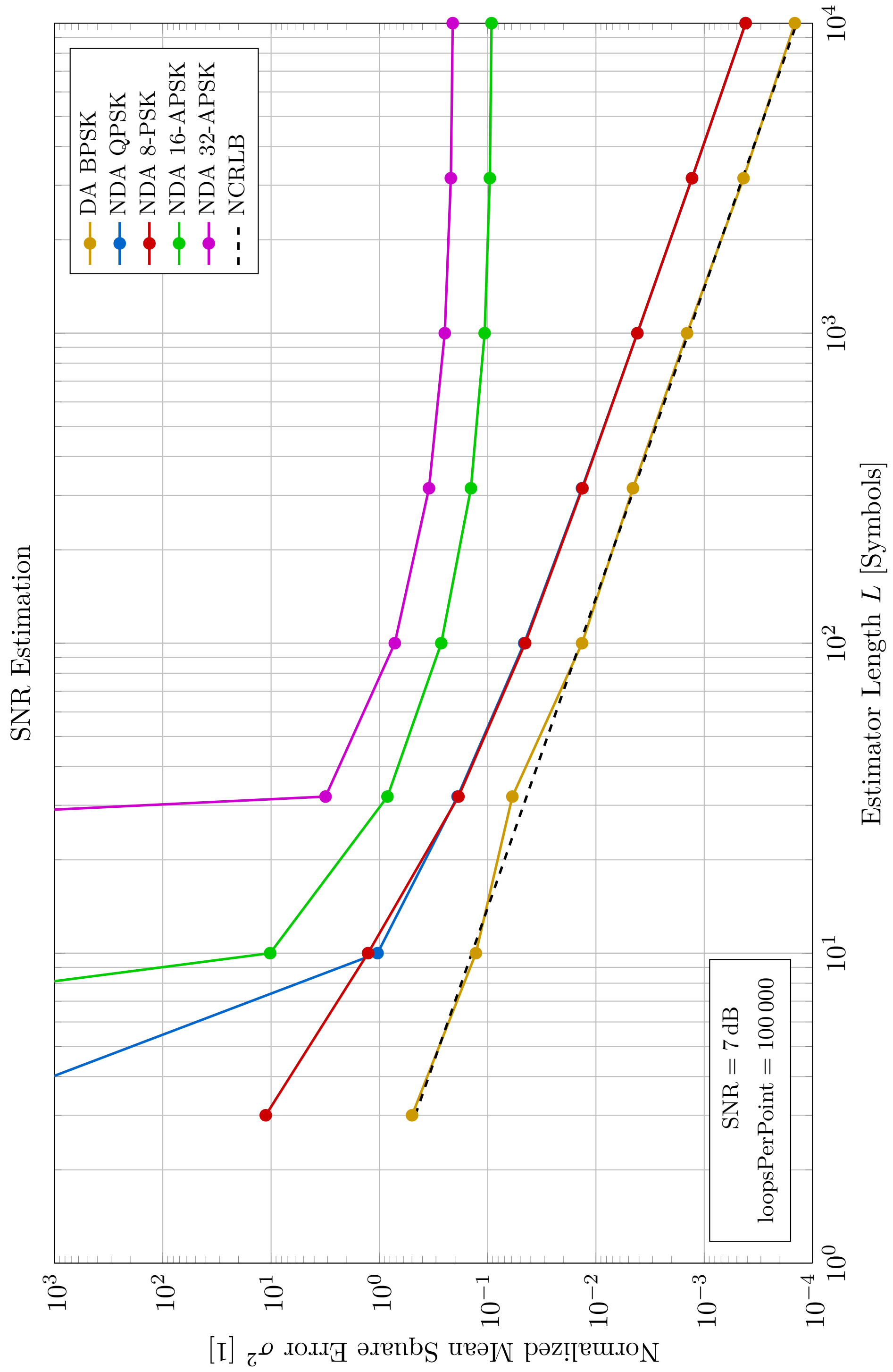


Figure 5.11: SNR $\rho = 7\,\mathrm{dB}$

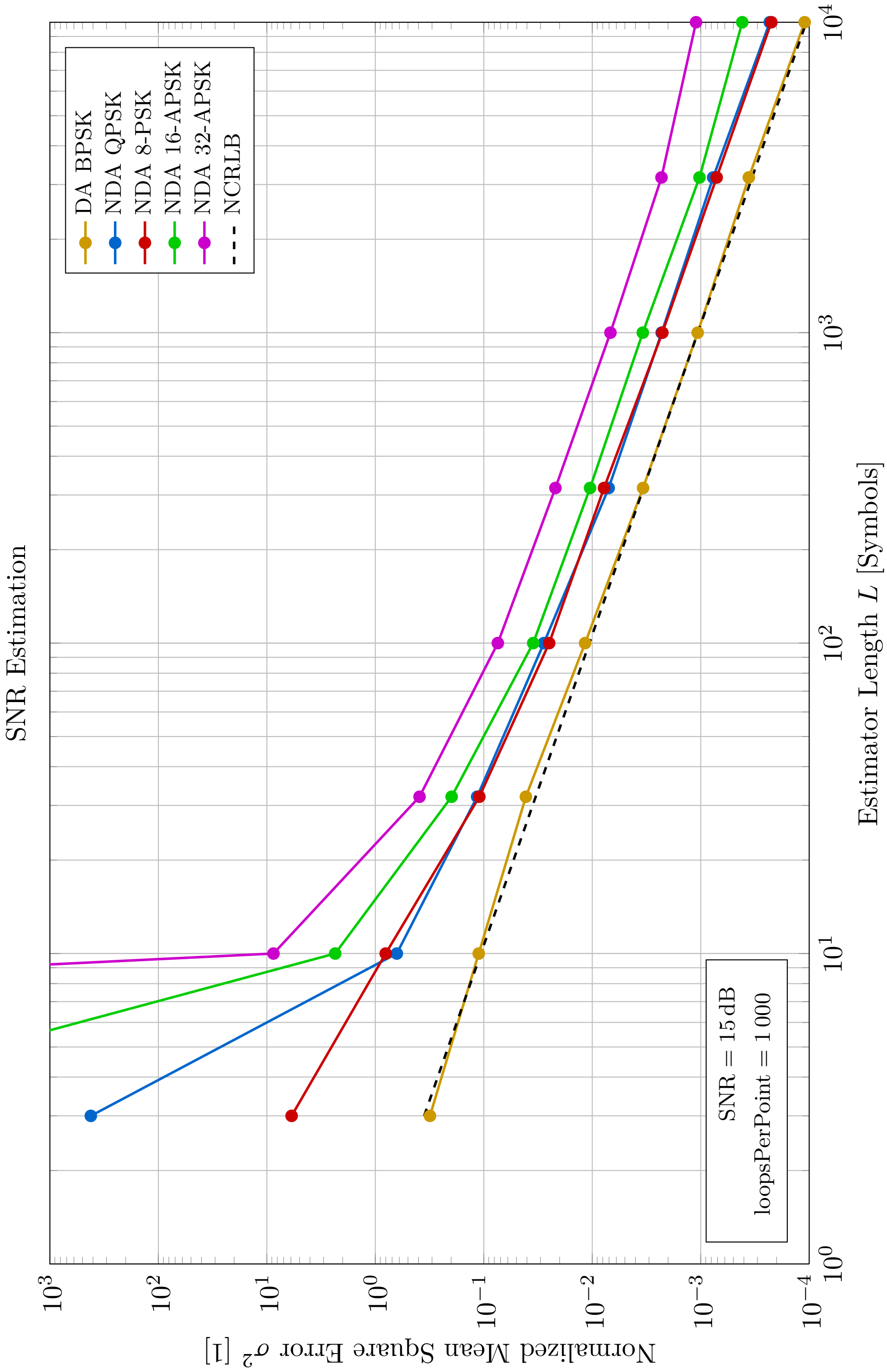


Figure 5.12: SNR $\rho = 15\,\mathrm{dB}$

5.4 Performance for typical Estimator Lengths

For use in real-world applications we are interested in the SNR estimation algorithms' performance when it comes to typical estimator lengths used in PLFRAMES. The following Figures 5.14 to 5.17 repeat the measuring principle introduced in Figure 5.6; this time however, there are different estimator lengths involved according to the different modulation types found in a PLFRAME. Additionally, the different NCRLBs are included.

5.4.1 Simulation Set-up

All simulation parameters can be found in Table 5.2 and in the Figures 5.14 to 5.17 themselves. Furthermore, the coderate used was $R = \frac{3}{4}$. Additionally, a carrier phase offset is applied to the signal and corrected before analysis. Figure 5.13 shows the simulation configuration used:

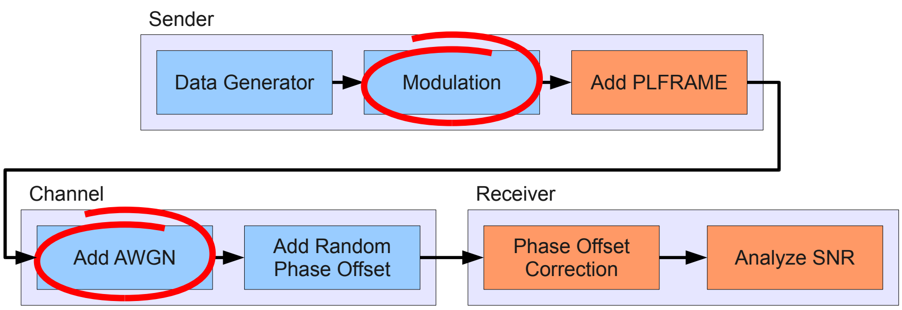


Figure 5.13: Transmission Path for Typical Estimator Lengths

For the DA SNR estimation, the overhead parts of the PLFRAME – i.e. the PLHEADER and the pilot blocks – are used; for the NDA estimation, the payload data is used. Therefore, the modulation of the DA parts never changes.

5.4.2 Simulation Results

Figures 5.14 to 5.17 show that for QPSK and 8-PSK, NDA estimation methods nearly always show a smaller MSE compared to DA methods; this behaviour can be observed due to the big difference in estimator lengths. For SNR values of -3 dB and higher, it is always advisable to use NDA estimation.

16- and 32-APSK modulations exhibit a different behavior: NDA estimation stays at a quite high level until around 12 dB, but then quickly plummets below MSE values of DA estimation.

Therefore, the conclusion is that NDA estimation should be used for SNR values of 15 dB and higher.

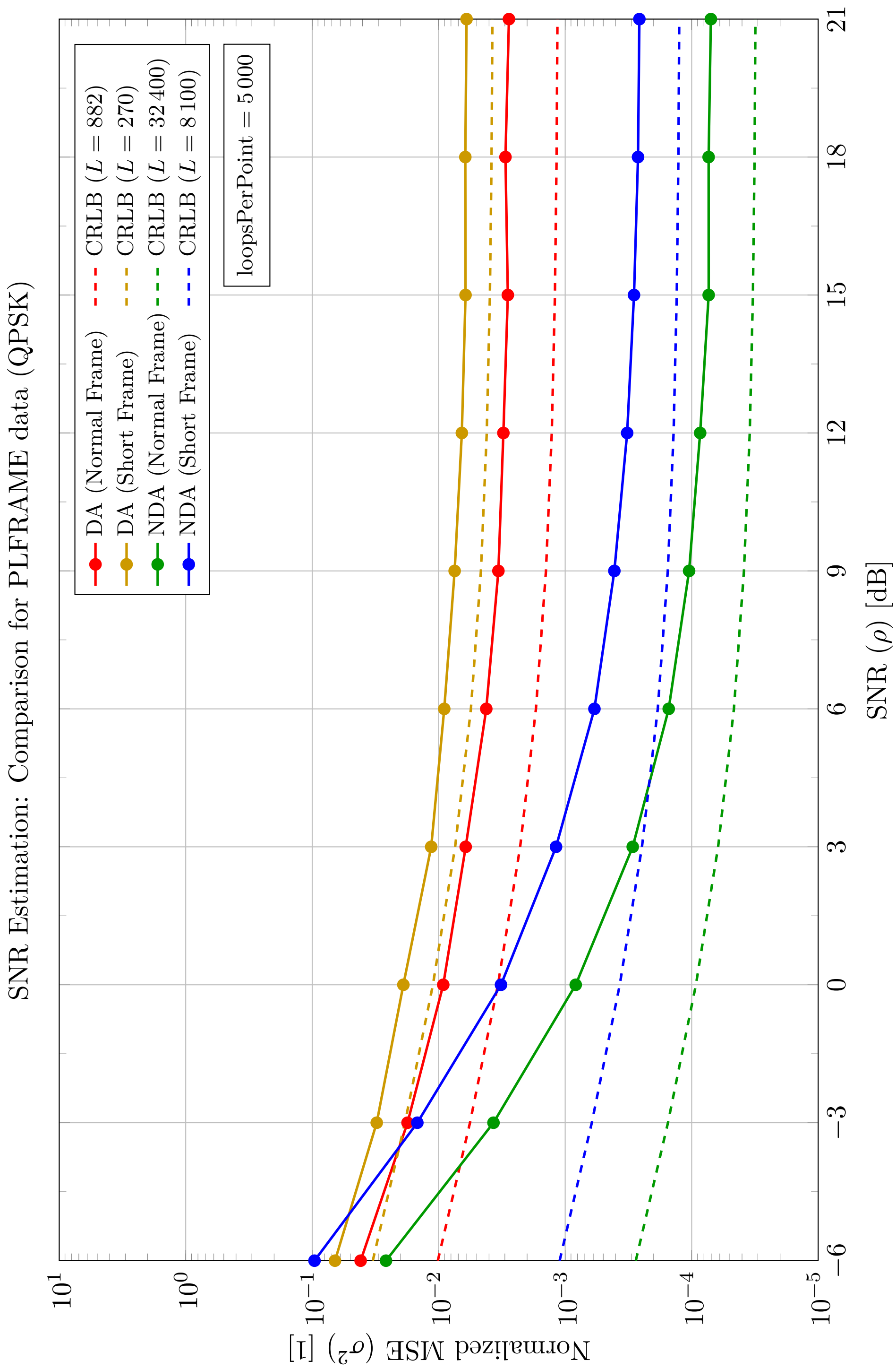


Figure 5.14: SNR Estimation: Comparison for PLFRAME data (QPSK)

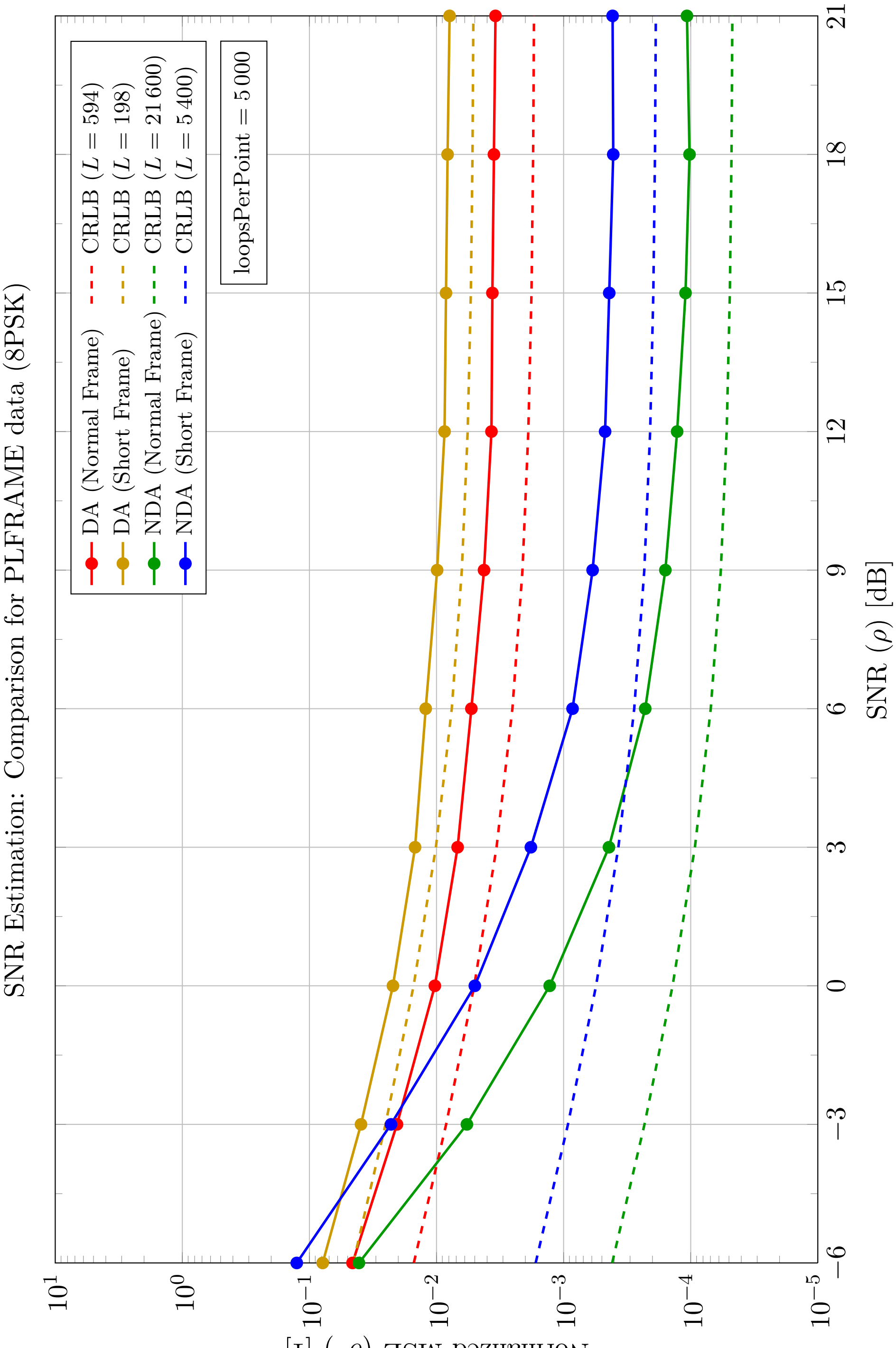


Figure 5.15: SNR Estimation: Comparison for PLFRAME data (8-PSK)

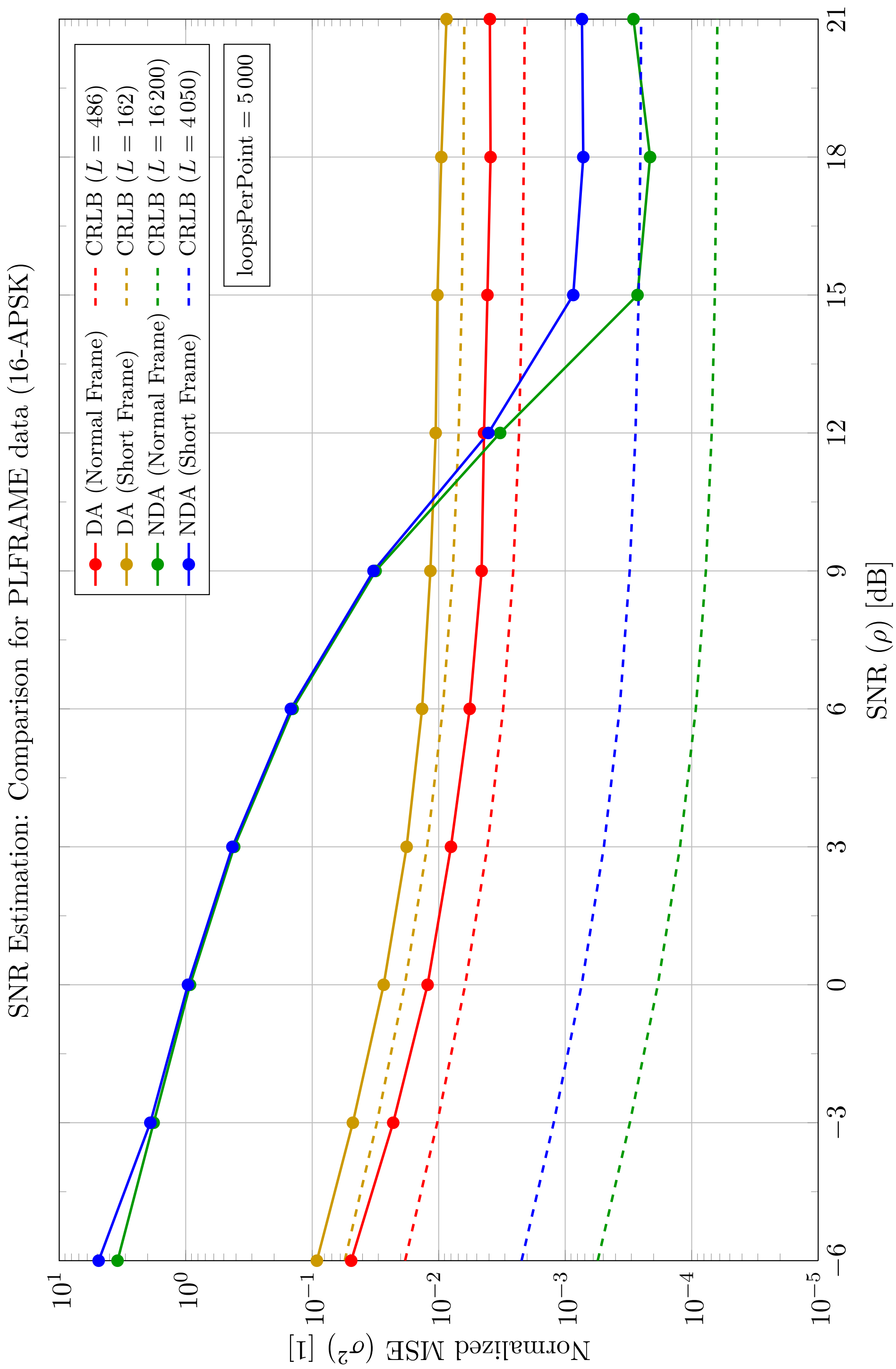


Figure 5.16: SNR Estimation: Comparison for PLFRAME data (16-APSK)

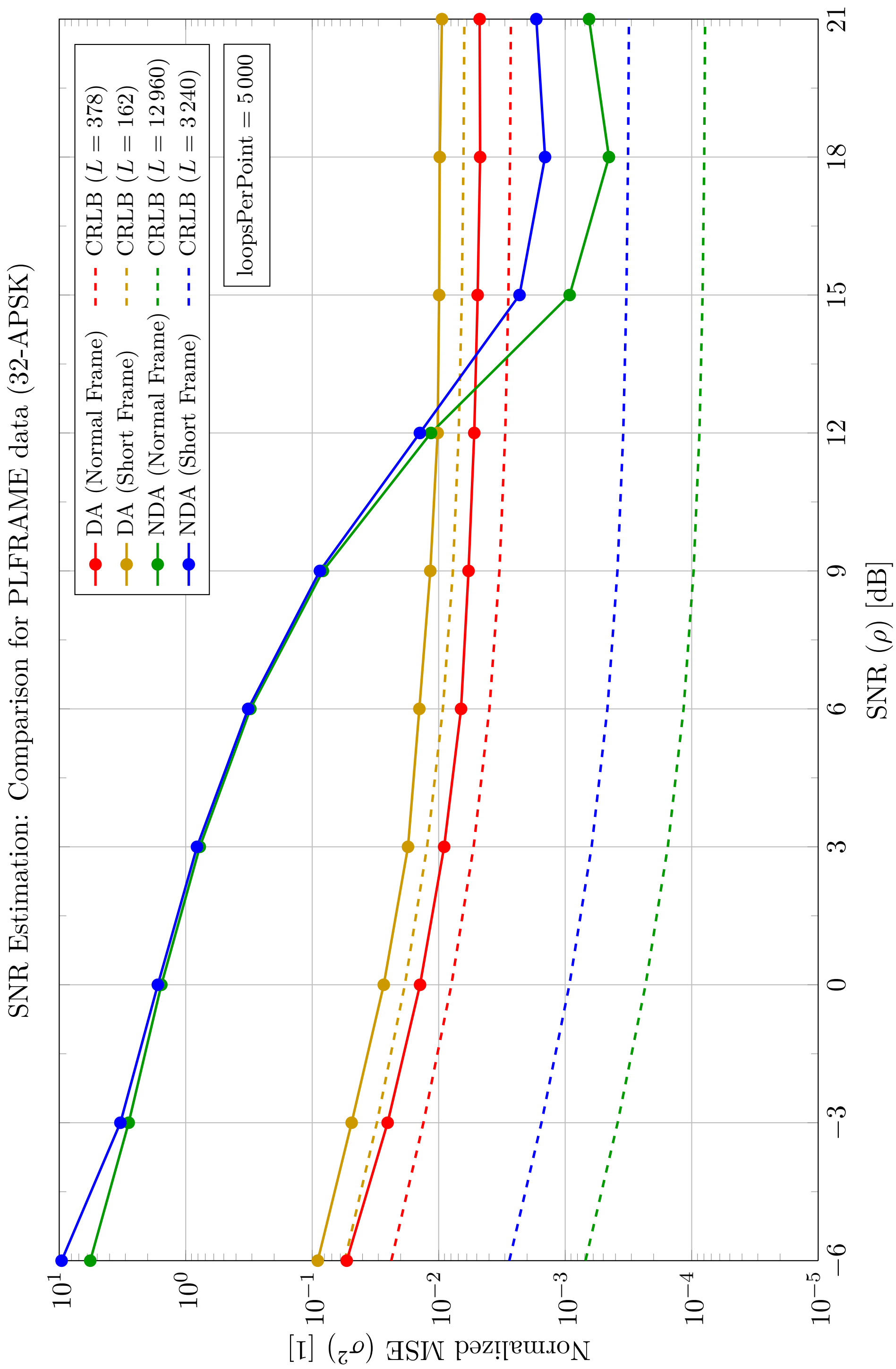


Figure 5.17: SNR Estimation: Comparison for PLFRAME data (32-APSK)

5.5 The Impact of Frequency Error

Another important channel impairment is a frequency error; frequency errors can be introduced by Doppler effects between the receiving ground station and the satellite. Frequency errors can be seen as a per-symbol phase shift; i.e. each symbol is rotated away from its preceding symbol by a constant amount Δf_T.

5.5.1 Simulation Set-up

In the channel, a frequency error and AWGN are applied to the signal; furthermore, a carrier phase offset is introduced and corrected. As before, a code rate of $R = \frac{3}{4}$ was used. The estimation lengths are 90 and 270 Symbols for pilot-less and pilot-assisted DA estimation, respectively, and 8 100 Symbols for NDA estimation. Figure 5.18 shows the simulation configuration used:

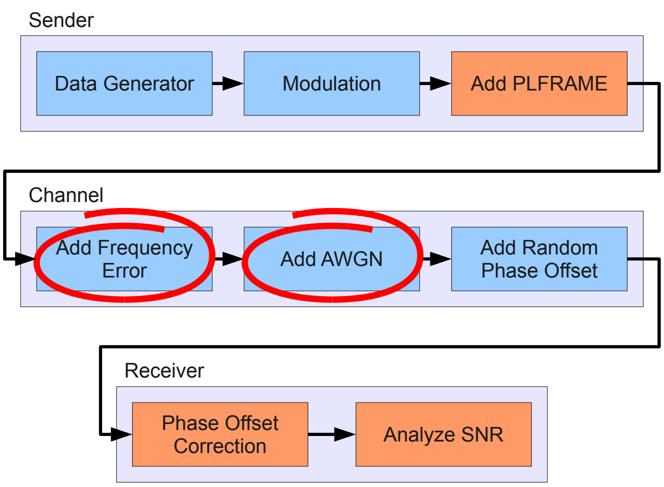


Figure 5.18: Transmission Path for Frequency Error Analysis

5.5.2 Simulation Results

Most importantly, the simulations show that NDA estimation depends only on the SNR value, not on the frequency error. This is little surprising, since NDA estimation works with second- and fourth-order moments, which aren't influenced by symbol rotation.

The second observation concerns the usage of pilot blocks for SNR estimation. While header-only estimation works well up to a certain point, pilot-aided estimation works nearly one decade longer. For this to work, however, it is important to estimate and correct the carrier phase for each segment independently. This way, each different phase offset accumulated due to the frequency error is corrected.

An interesting side note: In Figures 5.19 to 5.21, the DA estimations perform worse than NDA-based estimation. This is is not a contradiction to the results from section 5.3, but is due to the big difference in estimator lengths; section 5.4 showed the same phenomenon. Typical DA estimator lengths are in the magnitude of 100, whereas typical NDA estimator lengths are about two orders of magnitude larger, around 10 000 symbols.

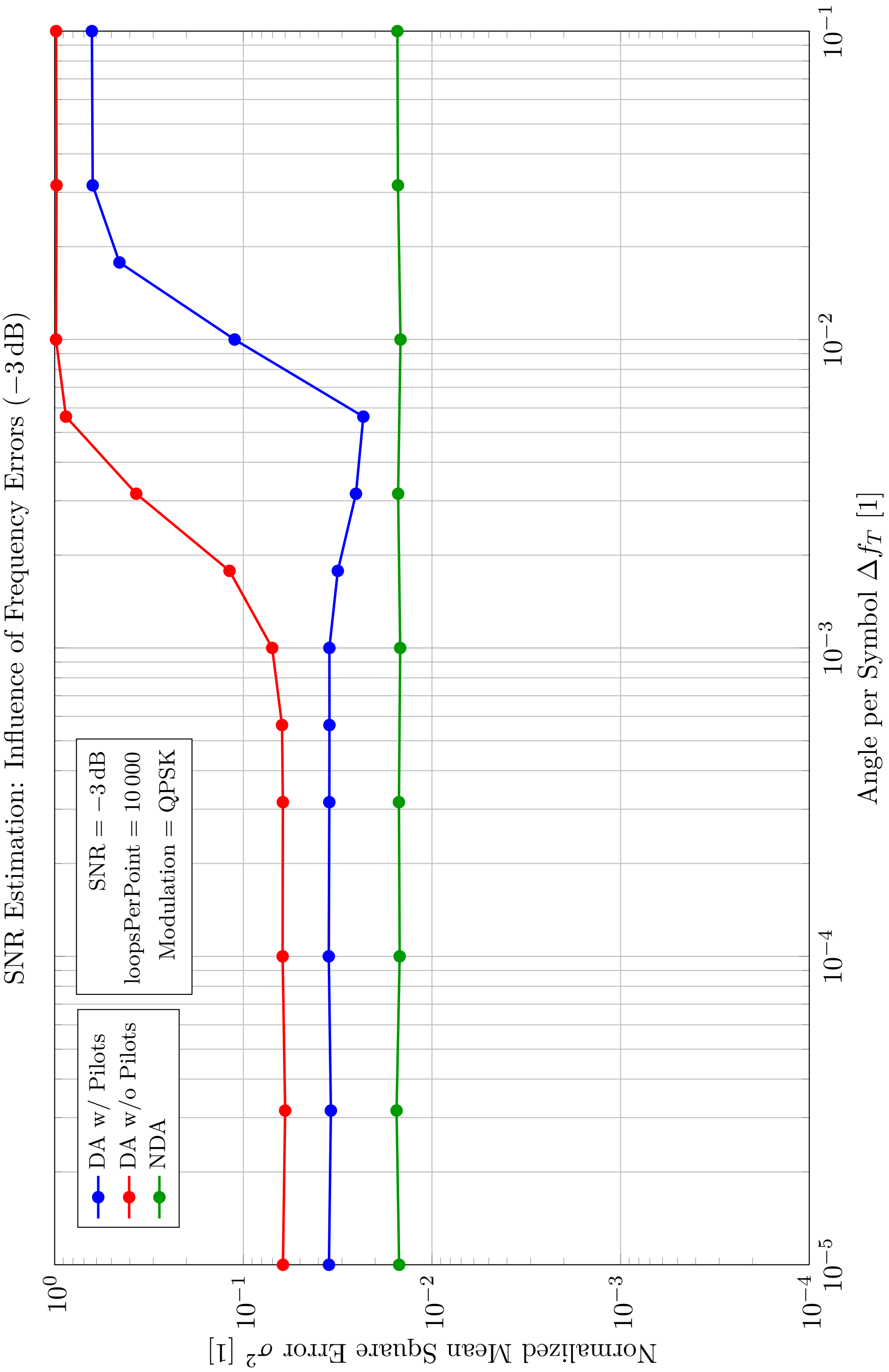


Figure 5.19: SNR Estimation: Influence of Frequency Errors (−3 dB)

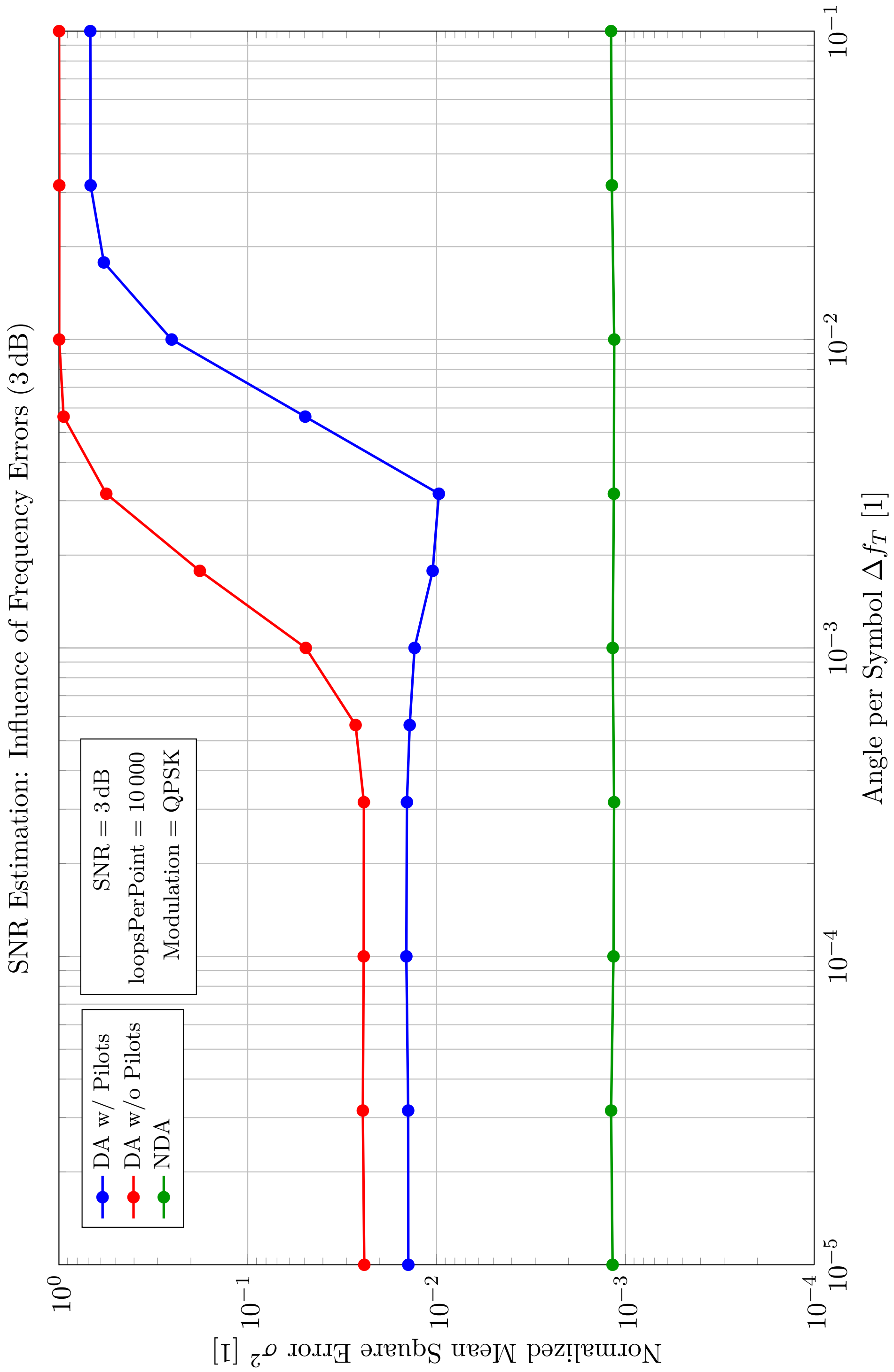


Figure 5.20: SNR Estimation: Influence of Frequency Errors (3 dB)

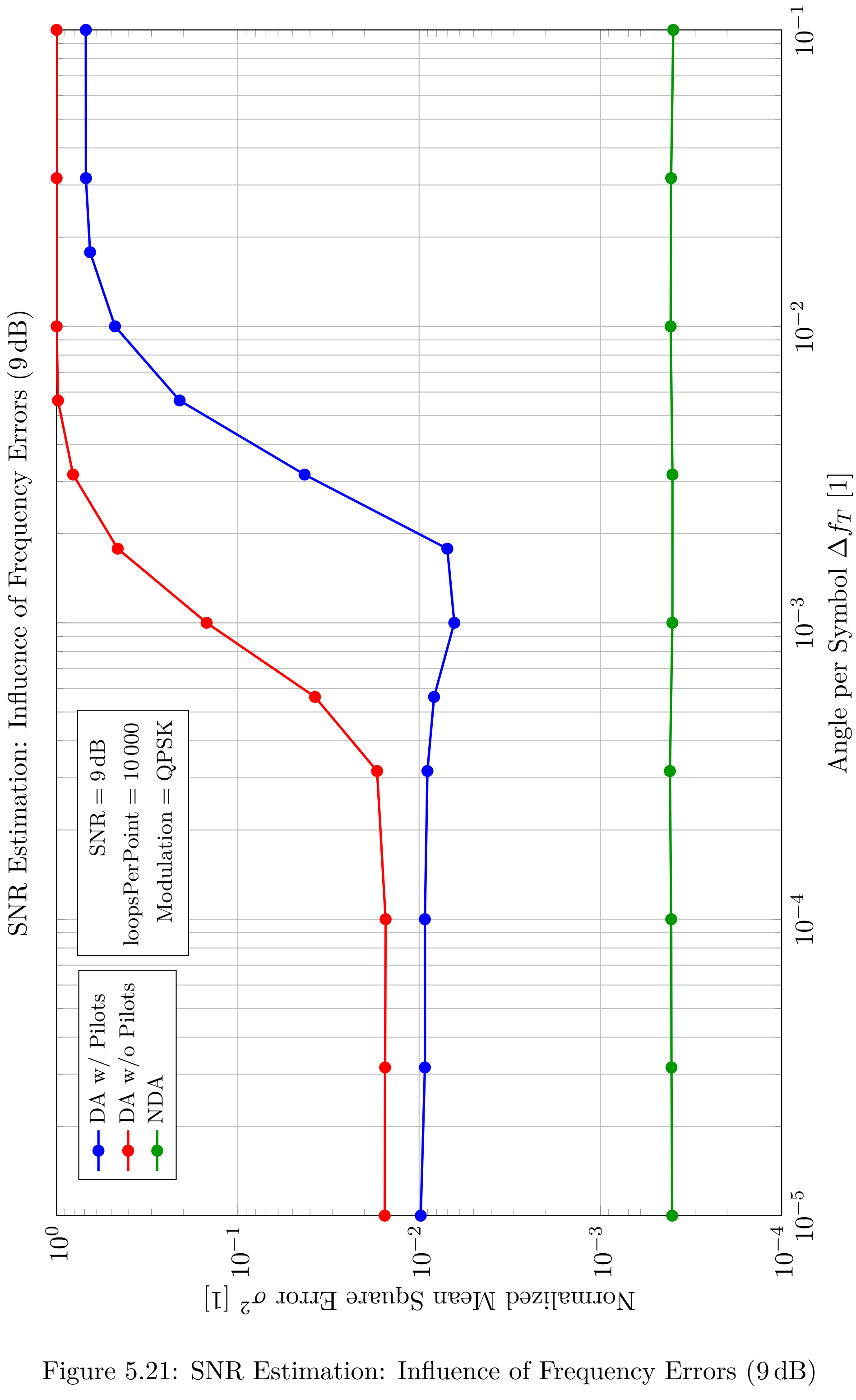


Figure 5.21: SNR Estimation: Influence of Frequency Errors (9 dB)

Chapter 6

Conclusion and Outlook

The goal of this work was to expand the existing DVB-S simulator by adding a physical layer framing mechanism; furthermore, several SNR estimation algorithms had to be implemented and subsequently analyzed. For these analyses, different channel impairments had to be used.

The result are several new modules for the simulator which can be seen in Appendix A, and which are explained in detail in chapter 4. Furthermore, the results of the conducted simulations concerning SNR estimation and channel impairments are presented in chapter 5. Interesting challenges were the set-up and operation of the simulator, a large already existing code base, and large datasets and simulation times.

Although several channel impairments were investigated during the course of this work, there are still some impairments left for future work, which may play a role in applied satellite communication. Examples are non-linear distortions due to non-ideal amplifiers [3]. Another interesting field of research could be frame synchronization, which was taken for granted during this work. A possible approach to this problem could be to use the SOF as input to a correlation function in order to determine the start of the PLFRAME.

List of Abbreviations

[...]	Ellipsis
ACM	Adaptive Coding and Modulation
APSK	Amplitude and Phase-Shift Keying
AWGN	Additive White Gaussian Noise
BPSK	Binary PSK
CRLB	Cramér-Rao Lower Bound
DA	Data-aided
DVB	Digital Video Broadcasting
ETSI	European Telecommunications Standards Institute
IDE	Integrated Development Environment
MSE	Mean Square Error
NCRLB	Normalized CRLB
NDA	Non-data-aided
PL	Physical Layer
PLS	Physical Layer Signalling
PSK	Phase-Shift Keying
QPSK	Quadrature PSK
SNR	Signal-to-Noise Ratio
SOF	Start of Frame

List of Tables

3.1 Possible symbol values for $\pi/2$-BPSK modulation 14

4.1 Parameter values before and after `TestDataSetter` 34

5.1 Min. SNR values for phase correction 36
5.2 Possible estimator lengths and numbers of pilot blocks 39
5.3 Min. Estimator Lengths for $MSE \leq 10^{-1}$ 44

List of Figures

3.1 PLFRAME construction overview 11
3.2 A typical transmission path 13
3.3 An empty constellation diagram 15
3.4 Three different PSK constellation diagrams 15
3.5 Two different APSK constellation diagrams 16
3.6 A QPSK-Signal with a Carrier Phase Error of $\theta = 147°$ . . . 17
3.7 A QPSK-Signal with a Frequency Error of $\Delta f_T \approx -7 \cdot 10^{-3}$. 17
3.8 The Effect of AWGN on a QPSK-Signal with SNR = 6 dB . . 18
3.9 The effect of AWGN on a 32-APSK-signal with SNR = 21 dB 19

4.1 Class dependency diagram. Arrows symbolize inheritance. . . 27

5.1 Transmission path symbol explanation 35
5.2 Transmission path for the carrier phase error correction . . . 36
5.3 Carrier phase offset correction performance 37
5.4 Transmission path for the basic SNR estimator analyses . . . 38
5.5 Estimated SNR $\hat{\rho}$ vs. SNR ρ 40
5.6 Normalized Mean Square Error σ^2 vs. SNR ρ 41
5.7 SNR Estimation: Bias of DA SNR Estimation 42
5.8 Transmission Path for the Influence of Estimator Length . . . 43
5.9 SNR $\rho = -3$ dB 45
5.10 SNR $\rho = 3$ dB 46
5.11 SNR $\rho = 7$ dB 47
5.12 SNR $\rho = 15$ dB 48
5.13 Transmission Path for Typical Estimator Lengths 49
5.14 SNR Estimation: Comparison for PLFRAME data (QPSK) . 51
5.15 SNR Estimation: Comparison for PLFRAME data (8-PSK) . 52
5.16 SNR Estimation: Comparison for PLFRAME data (16-APSK) 53
5.17 SNR Estimation: Comparison for PLFRAME data (32-APSK) 54
5.18 Transmission Path for Frequency Error Analysis 55
5.19 SNR Estimation: Influence of Frequency Errors (-3 dB) . . . 57
5.20 SNR Estimation: Influence of Frequency Errors (3 dB) 58
5.21 SNR Estimation: Influence of Frequency Errors (9 dB) 59

List of Listings

4.1 A typical example for a simulation file 25
4.2 Sample usage for `InsertPLFrame` 29
4.3 Sample usage for `RemovePLFrame` 29
4.4 Sample usage for `AnalyzeSNR` 31
4.5 Sample usage for `AnalyzePLFrame` 32
4.6 Sample usage for `TestDataSetter` 34
A.1 Defines and Debug Code (Header file) 67
A.2 Defines and Debug Code (Main file) 69
A.3 Base Class (Header file) 70
A.4 Base Class (Main file) 72
A.5 Insert PLHEADER (Header file) 84
A.6 Insert PLHEADER (Main file) 85
A.7 Remove PLHEADER (Header file) 89
A.8 Remove PLHEADER (Main file) 90
A.9 Analyze PLFRAME (Header file) 93
A.10 Analyze PLFRAME (Main file) 94
A.11 Analyze SNR (Header file) 98
A.12 Analyze SNR (Main file) 99
A.13 Testdata Setter (Header file) 106
A.14 Testdata Setter (Main file) 107
B.1 `go.bat`: Run `all_sims.bat` with low priority 109
B.2 `all_sims.bat`: Execute all simfiles in the current directory 109

Bibliography

[1] European Telecommunications Standards Institute. *Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2)*, Aug 2009. EN 302 307 V1.2.1 http://www.etsi.org/deliver/etsi_en/302300_302399/302307/01.02.01_60/en_302307v010201p.pdf.

[2] H.-J. Bartsch. *Taschenbuch mathematischer Formeln*. Fachbuchverlag Leipzig im Carl Hanser Verlag, 20th edition, 2004.

[3] E. Casini, R. De Gaudenzi, and A. Ginesi. DVB-S2 modem algorithms design and performance over typical satellite channels. *International Journal of Satellite Communications and Networking*, 22:281–318, Jun 2004.

[4] W. Gappmair and O. Koudelka. Moment-Based SNR Estimation of Signals with Non-Constant Envelope. In *Proc. 3rd Advanced Satellite Mobile Systems Conf., Herrsching, Germany*, pages 301–304, May 2006.

[5] Joanneum Research. SNR Estimation for 16- and 32-APSK. Unpublished (Filename: 4267_001.pdf).

[6] Joanneum Research. Code-Aware Joint Estimation of Carrier Phase and SNR for Linear Modulation Schemes. Unpublished (Filename: AnalysisSNRonly.pdf), 2010.

[7] Joanneum Research. Software Installation & Maintenance Document. Unpublished (Filename: SW_Installation&Maintenance_Rev4.doc), Jun 2010.

[8] Newtec. *EL470 IP Satellite Modem Product Leaflet, Rev.9*, Mar 2012. http://www.newtec.eu/fileadmin/user_upload/product_leaflets/Elevation/IP_Satellite_Modem_EL470_R9.pdf.

[9] D. R. Pauluzzi and N. C. Beaulieu. A Comparison of SNR Estimation Techniques for the AWGN Channel. *IEEE Transactions on Communications*, 48:1681–1691, Oct 2000.

[10] Wikimedia Foundation. Additive white gaussian noise. Internet, Mar 2012.
http://en.wikipedia.org/wiki/Additive_white_Gaussian_noise.

[11] Wikimedia Foundation. Cramér–Rao bound. Internet, Mar 2012.
http://en.wikipedia.org/wiki/CRLB.

[12] Wikimedia Foundation. DVB-S2. Internet, Mar 2012.
http://en.wikipedia.org/wiki/DVB-S2.

[13] Wikimedia Foundation. Estimation theory. Internet, Mar 2012.
http://en.wikipedia.org/wiki/Estimation_theory.

[14] Wikimedia Foundation. Estimator. Internet, Mar 2012.
http://en.wikipedia.org/wiki/Estimator.

[15] Wikimedia Foundation. Hamming distance. Internet, Mar 2012.
http://en.wikipedia.org/wiki/Hamming_distance.

[16] Wikimedia Foundation. Link adaptation. Internet, Mar 2012.
http://en.wikipedia.org/wiki/Link_adaptation.

[17] Wikimedia Foundation. Signal-to-noise ratio. Internet, Mar 2012.
http://en.wikipedia.org/wiki/Signal-to-noise_ratio.

Appendix A

Program Source Code

For this project, several additions had to be made to the DVB-S simulator. The final source code comprises more than 1 700 lines of code and is listed in this section.

A.1 Common Code

A.1.1 Defines and Debug Code

Listing A.1: Defines and Debug Code (Header file)

```
/*
 * ETSI_Common.h
 *
 *  Created on: 22.09.2011
 *      Author: Johannes Pribyl
 */

#ifndef ETSI_COMMON_H_
#define ETSI_COMMON_H_

#include "modemSim.h"
#include "utils.h"

// Code rates required by ETSI standard but not included in
    modemDefines.h
const int FEC_1_4 = 14;
const int FEC_3_5 = 35;

// Constant values defined by the ETSI standard
const long SOF_ = 0x18D2E82;
const long long SCRAMBLE_CODE_ = 0x719D83C953422DFA;

// Table with all possible values of the PLS field
const unsigned long long DECODE_PLS_TABLE[128] = { 0ull,
```

```
        6148914691236517205ull, 18446744073709551615ull,
        12297829382473034410ull, 4294967295ull,
           6148914692668172970ull,
        18446744069414584320ull, 12297829381041378645ull,
           281470681808895ull,
        6149008514797120170ull, 18446462603027742720ull,
        12297735558912431445ull, 281474976645120ull,
           6149008516228732245ull,
        18446462598732906495ull, 12297735557480819370ull,
           71777214294589695ull,
        6172840429334713770ull, 18374966859414961920ull,
        12273903644374837845ull, 71777218556133120ull,
           6172840430755228245ull,
        18374966855153418495ull, 12273903642954323370ull,
           72056494543077120ull,
        6172933522750876245ull, 18374687579166474495ull,
        12273810550958675370ull, 72056498804490495ull,
           6172933524171347370ull,
        18374687574905061120ull, 12273810549538204245ull,
        1085102592571150095ull, 6510615555426900570ull,
        17361641481138401520ull, 11936128518282651045ull,
        1085102596360827120ull, 6510615556690126245ull,
        17361641477348724495ull, 11936128517019425370ull,
        1085350949055099120ull, 6510698340921550245ull,
        17361393124654452495ull, 11936045732788001370ull,
        1085350952844660495ull, 6510698342184737370ull,
        17361393120864891120ull, 11936045731524814245ull,
        1148435428713435120ull, 6531726500807662245ull,
        17298308644996116495ull, 11915017572901889370ull,
        1148435432473620495ull, 6531726502061057370ull,
        17298308641235931120ull, 11915017571648494245ull,
        1148681852462100495ull, 6531808642057217370ull,
        17298062221247451120ull, 11914935431652334245ull,
        1148681856222171120ull, 6531808643310574245ull,
        17298062217487380495ull, 11914935430398977370ull,
        3689348814741910323ull, 7378697629483820646ull,
        14757395258967641292ull, 11068046444225730969ull,
        3689348817318890700ull, 7378697630342814105ull,
        14757395256390660915ull, 11068046443366737510ull,
        3689517697150995660ull, 7378753923620182425ull,
        14757226376558555955ull, 11067990150089369190ull,
        3689517699727897395ull, 7378753924479149670ull,
        14757226373981654220ull, 11067990149230401945ull,
        3732415143318664140ull, 7393053072342738585ull,
        14714328930390887475ull, 11053691001366813030ull,
        3732415145875590195ull, 7393053073195047270ull,
        14714328927833961420ull, 11053691000514504345ull,
        3732582711467756595ull, 7393108928392436070ull,
        14714161362241795020ull, 11053635145317115545ull,
```

```
      3732582714024604620ull, 7393108929244718745ull,
      14714161359684946995ull, 11053635144464832870ull,
      4340410370284600380ull, 7595718147998050665ull,
      14106333703424951235ull, 10851025925711500950ull,
      4340410372558406595ull, 7595718148755986070ull,
      14106333701151145020ull, 10851025924953565545ull,
      4340559384174969795ull, 7595767819294840470ull,
      14106184689534581820ull, 10850976254414711145ull,
      4340559386448706620ull, 7595767820052752745ull,
      14106184687260844995ull, 10850976253656798870ull,
      4378410071969971395ull, 7608384715226507670ull,
      14068334001739580220ull, 10838359358483043945ull,
      4378410074226082620ull, 7608384715978544745ull,
      14068333999483468995ull, 10838359357731006870ull,
      4378557926219170620ull, 7608433999976240745ull,
      14068186147490380995ull, 10838310073733310870ull,
      4378557928475212995ull, 7608434000728254870ull,
      14068186145234338620ull, 10838310072981296745ull };

struct PLS_ {
   char MODCOD;
   char TYPE;
};

#define PRINT_BITS(var) PrintBits(var, #var, sizeof(var));
void PrintBits(unsigned long long SOF_field, const string name, int max_val);
void debugBlockOut(BurstContainer *burst, unsigned int length = 0);

#endif /* ETSI_COMMON_H_ */
```

Listing A.2: Defines and Debug Code (Main file)

```
/*
 * ETSI_Common.cpp
 *
 *  Created on: 22.09.2011
 *      Author: Johannes Pribyl
 */

#include "ETSI_Common.h"
#include "utils.h"

using namespace std;

// Usage via macro: PRINT_BITS(var);
void PrintBits(unsigned long long value, const string name, int max_val) {

```

```
   bool bit;
   unsigned long long mask = 1;
   mask = mask << ((max_val * 8) - 1);
   cout << name << "\t";
   if (name.length() < 2)
      cout << "\t";
   for (unsigned int i = 0; i < (max_val * 8); i++) {
      bit = (mask >> i) & value;
      cout << bit;
      if (i % 4 == 3)
         cout << "␣";
   }
   cout << "\t" << value << "\n";
}

void debugBlockOut(BurstContainer *burst, unsigned int length) {

   if (length == 0)
      length = burst->header.containerBodyLength;

   for (unsigned int i = 0; i < length; i++) {
      cout << i << ":\t";
      if (burst->bodySymbolCartFloat[i].re > 0)
         cout << "␣";
      cout << burst->bodySymbolCartFloat[i].re << "␣\t";
      if (burst->bodySymbolCartFloat[i].im > 0)
         cout << "␣";
      cout << burst->bodySymbolCartFloat[i].im << "\n";
   }
   cout << "\n";

   cout << "burst->header.containerBodyLength:␣"
         << burst->header.containerBodyLength << "\n";
}
```

A.1.2 Base Class

Listing A.3: Base Class (Header file)

```
/*
 * ETSI_Baseclass.h
 *
 *  Created on: 24.10.2011
 *      Author: Johannes Pribyl
 */

#ifndef ETSI_BASECLASS_H_
#define ETSI_BASECLASS_H_
```

```
#include "ETSI_Common.h"

class ETSI_Baseclass {
protected:
    void removeHeaderAndPilots();
    void getHeaderAndPilots();
    double sumAbsolutePowers(symb_cart_float* values, unsigned int length, int power, double threshold = 0);
    int codeRate2IndexMapping(int codeRateId);

    double estimateSNRmb32APSK(symb_cart_float* body_cart, unsigned int length, unsigned char code_rate, float symbEner);
    double estimateSNRmb16APSK(symb_cart_float* body_cart, unsigned int length, unsigned char code_rate);
    double estimateSNRmbMPSK(symb_cart_float* body_cart, unsigned int length);
    double estimateSNRda(symb_cart_float* ref_body_cart, symb_cart_float* body_cart, unsigned int length);
    double estimateSNRda_plframe(symb_cart_float* ref_body_cart, symb_cart_float* body_cart, unsigned int length);

    void correctPhaseError(symb_cart_float* body_cart,
          symb_polar_float* body_polar, unsigned int length);
    float estimatePhaseError(symb_cart_float* PLHeader, bool header);

    void demodulatePLHeader(symb_cart_float* PLHeader, float* PLScode);
    void descramblePLScode(float* PLScode);
    struct PLS_ decodePLScode(float* PLScode);

    signed char getModulation(char MODCOD);
    signed char getCodeRate(char MODCOD);

    bool getFrameSize(char TYPE);
    bool getPilotFields(char TYPE);

    float getSimilarity(unsigned long long reference, float* data);
    void getModulatedSOF(symb_cart_float* modSOF);

    unsigned long long encodePLS(struct PLS_ PLS);
    unsigned long long scramblePLScode(unsigned long long PLScode);
```

```
    void modulatePLHeader(unsigned long long PLScode,
        symb_cart_float* PLHeader);
};

#endif /* ETSI_BASECLASS_H_ */
```

Listing A.4: Base Class (Main file)

```
/*
 * ETSI_Baseclass.cpp
 *
 *  Created on: 24.10.2011
 *      Author: Johannes Pribyl
 */

#include "ETSI_Baseclass.h"

using namespace std;

double ETSI_Baseclass::estimateSNRmb32APSK(symb_cart_float*
     receivedBody, unsigned int length, unsigned char
    code_rate, float symbEner) {

    //Array with all radius ratios according to EN 302 307
    double Y1[5] = { 2.84, 2.72, 2.64, 2.54, 2.53 };
    double Y2[5] = { 5.27, 4.87, 4.64, 4.33, 4.30 };

    //Map code rate to array index
    int CRindex = codeRate2IndexMapping(code_rate);

    //Calculate R1, R2, R3 (Code adapted from utils.cpp,
        lines 1662ff and 1525f)
    double R1 = 32 / (4 + 12 * pow(Y1[CRindex], 2) + 16 *
        pow(Y2[CRindex], 2));
    R1 = sqrt(R1) * sqrt(symbEner);
    double R2 = R1 * Y1[CRindex];
    double R3 = R1 * Y2[CRindex];

    //Scale symbols to energy 1
    symb_polar_float receivedBodyPol[length];

    converter->ippConvertingCartesianToPolar(receivedBody,
        receivedBodyPol,
           length);

    for (unsigned int i = 0; i < length; i++) {
       receivedBodyPol[i].amp /= SRT_2;
    }

```

```
    converter->ippConvertingPolarToCartesian(receivedBodyPol, receivedBody,
        length);

    R1 /= SRT_2;
    R2 /= SRT_2;
    R3 /= SRT_2;

    //Calculate M2 and M4
    double M2 = sumAbsolutePowers(receivedBody, length, 2);
    double M4 = sumAbsolutePowers(receivedBody, length, 4);

    //Calculate estimated signal power
    double K = 0.125 * pow(R1, 4) + 0.375 * pow(R2, 4) + 0.5 * pow(R3, 4);
    double S = sqrt((2.0 * pow(M2, 2) - M4) / (2 - K));

    //Calculate partition radius R12 between outer and middle ring
    double R23 = sqrt(S) * (R2 + R3) / 2.0;

    //Calculate M2 and M4 using only symbols with sufficient magnitude
    M2 = sumAbsolutePowers(receivedBody, length, 2, R23);
    M4 = sumAbsolutePowers(receivedBody, length, 4, R23);

    //Calculate estimated signal and noise powers
    S = sqrt(2 * pow(M2, 2) - M4);
    double N = M2 - S;

    return (S / N) / pow(R3, 2);
}

double ETSI_Baseclass::estimateSNRmb16APSK(symb_cart_float* receivedBody, unsigned int length, unsigned char code_rate) {

    //Calculate R1 and R2 (Code taken from class MapperBitToSymbol)
    double beta = Mapper::getBetaFromCoderateId(code_rate);
    double R1 = sqrt(4.0 / (1 + 3 * beta * beta));
    double R2 = sqrt(4.0 * beta * beta / (1 + 3 * beta * beta));

    //Calculate M2 and M4
    double M2 = sumAbsolutePowers(receivedBody, length, 2);
    double M4 = sumAbsolutePowers(receivedBody, length, 4);

    //Calculate estimated signal power
```

```
    double K = 0.25 * pow(R1, 4) + 0.75 * pow(R2, 4);
    double S = sqrt((2.0 * pow(M2, 2) - M4) / (2 - K));

    //Calculate partition radius R12 between the two rings
    double R12 = sqrt(S) * (R1 + R2) / 2.0;

    //Calculate M2 and M4 using only symbols with sufficient
        magnitude
    M2 = sumAbsolutePowers(receivedBody, length, 2, R12);
    M4 = sumAbsolutePowers(receivedBody, length, 4, R12);

    //Calculate estimated signal and noise powers
    S = sqrt(2 * pow(M2, 2) - M4);
    double N = M2 - S;

    return (S / N) / pow(R2, 2);
}

double ETSI_Baseclass::estimateSNRmbMPSK(symb_cart_float*
    receivedBody, unsigned int length) {

    //Calculate M2 and M4
    double M2 = sumAbsolutePowers(receivedBody, length, 2);
    double M4 = sumAbsolutePowers(receivedBody, length, 4);

    //Calculate estimated signal and noise powers
    double S = sqrt(2 * pow(M2, 2) - M4);
    double N = M2 - S;

    return S / N;
}

double ETSI_Baseclass::estimateSNRda(symb_cart_float*
    ref_body_cart, symb_cart_float* body_cart, unsigned int
    length) {

    //Normalise symbols to length 1
    symb_polar_float body_cart_pol[length];
    symb_polar_float ref_body_cart_pol[length];

    converter->ippConvertingCartesianToPolar(body_cart,
        body_cart_pol, length);
    converter->ippConvertingCartesianToPolar(ref_body_cart,
        ref_body_cart_pol,
            length);

    for (unsigned int i = 0; i < length; i++) {
        body_cart_pol[i].amp /= SRT_2;
        ref_body_cart_pol[i].amp /= SRT_2;
```

```
    }

    converter->ippConvertingPolarToCartesian(body_cart_pol,
        body_cart, length);
    converter->ippConvertingPolarToCartesian(
        ref_body_cart_pol, ref_body_cart,
            length);

    //Calculate M1
    double M1 = 0;
    double a, b, c, d;
    for (unsigned int i = 0; i < length; i++) {
        // Complex multiplication: (a+bi)*(c+di)=(ac-bd)+(ad+
            bc)i
        a = ref_body_cart[i].re;
        b = -ref_body_cart[i].im;
        c = body_cart[i].re;
        d = body_cart[i].im;

        // Sum up real parts
        M1 += (a * c - b * d);
    }
    M1 = M1 / length;

    //Calculate M2
    double M2 = sumAbsolutePowers(body_cart, length, 2);

    //Calculate M0
    double M0 = sumAbsolutePowers(ref_body_cart, length, 2);

    return pow(M1, 2) / (M2 * pow(M0, 2) - pow(M1, 2) * M0);
}

double ETSI_Baseclass::estimateSNRda_plframe(
    symb_cart_float* ref_body_cart, symb_cart_float*
    body_cart, unsigned int length) {

    //Estimate SNR using the PLHEADER
    double snr_sum = estimateSNRda(ref_body_cart, body_cart,
        90);
    unsigned int count_estimates = 1 + floor((length - 90) /
        36);

    //Estimate SNR using the pilot blocks
    symb_cart_float ref_part_cart[36];
    symb_cart_float part_cart[36];
    for (unsigned int pilot_index = 0; pilot_index < ((
        length - 90) / 36); pilot_index++) {
```

```
        for (unsigned int symbol_index = 0; symbol_index <
            36; symbol_index++) {
            ref_part_cart[symbol_index] = ref_body_cart[90 +
                pilot_index * 36
                    + symbol_index];
            part_cart[symbol_index] = body_cart[90 +
                pilot_index * 36
                    + symbol_index];
        }
        snr_sum += estimateSNRda(ref_part_cart, part_cart,
            36);
    }

    return snr_sum / (double) count_estimates;
}

//This method taken from utils.cpp
int ETSI_Baseclass::codeRate2IndexMapping(int codeRateId) {
    switch (codeRateId) {
    case FEC_3_4:
        return 0;
    case FEC_4_5:
        return 1;
    case FEC_5_6:
        return 2;
    case FEC_8_9:
        return 3;
    case FEC_9_10:
        return 4;
    default:
        error_handler.throwException(
                "ETSI_Baseclass::codeRate2IndexMapping():␣
                    unsupported␣or␣invalid␣code␣rate!\nSupported
                    ␣CR:␣3/4;␣4/5;␣5/6;␣8/9;␣9/10");
        return 0;
    }
}

double ETSI_Baseclass::sumAbsolutePowers(symb_cart_float*
    values, unsigned int length, int power, double threshold
    ) {

    double sum = 0;
    int items_count = 0;
    for (unsigned int i = 0; i < length; i++) {
        if (values[i].absolute() > threshold) {
            sum += pow(values[i].absolute(), power);
            items_count++;
        }
```

```
    }

    return sum / (double) items_count;
}

void ETSI_Baseclass::correctPhaseError(symb_cart_float*
    body_cart, symb_polar_float* body_polar, unsigned int
    length) {

    bool header = false;
    if (length == 90)
        header = true;

    // Estimate phase error
    float phase_error = estimatePhaseError(body_cart, header
        );

    // Convert body to polar coordinates
    converter->ippConvertingCartesianToPolar(body_cart,
        body_polar, length);

    // Substract estimated phase from each symbol
    for (unsigned int i = 0; i < length; i++) {
        body_polar[i].phase -= phase_error;
    }

    // Convert body back to cartesian coordinates
    converter->ippConvertingPolarToCartesian(body_polar,
        body_cart, length);
}

float ETSI_Baseclass::estimatePhaseError(symb_cart_float*
    PLHeader, bool plheader) {

    symb_cart_float sum;
    unsigned int length = 36;

    if (plheader)
        length = 26;

    symb_cart_float referenceData[length];
    float a, b, c, d;

    sum.re = 0;
    sum.im = 0;

    // Get reference data
    if (plheader) {
        getModulatedSOF(referenceData);
```

```
    } else {
        for(unsigned int i=0;i<length;i++) {
            referenceData[i].re = 1;
            referenceData[i].im = 1;
        }
    }

    // Calculate phase error
    for (unsigned int i = 0; i < length; i++) {
        // Complex multiplication:
        //  (a+bi)*(c+di)=(ac-bd)+(ad+bc)i
        a = PLHeader[i].re;
        b = -PLHeader[i].im;
        c = referenceData[i].re;
        d = referenceData[i].im;
        sum.re += (a * c - b * d);
        sum.im += (a * d + b * c);
    }

    float phase_error = -sum.argument();
    if (phase_error < 0)
        phase_error += 2 * PI;

    return phase_error;
}

// Inserts the modulated SOF into the first 26 elements of
    the modSOF array
void ETSI_Baseclass::getModulatedSOF(symb_cart_float*
    modSOF) {

    unsigned long long mask = 1;
    bool bit;

    // The ETSI standard requires an additional factor of 1/
        sqrt(2) for I and Q, but this factor was changed to 1
         due to compatibility issues with older parts of this
         simulator

    mask = mask << 25;
    for (int i = 0; i < 26; i++) {
        // Modulate SOF
        bit = (mask >> i) & SOF_;
        if (i % 2 == 0) {
            modSOF[i].im = 1 - 2 * bit;
            modSOF[i].re = -modSOF[i].im;
        } else {
            modSOF[i].im = 1 - 2 * bit;
            modSOF[i].re = modSOF[i].im;
```

```
      }
   }
}

// PLScode should be 90 symbols long
void ETSI_Baseclass::demodulatePLHeader(symb_cart_float*
    PLHeader, float* PLScode) {

   // Demodulate Pi/2 BPSK to softbits
   for (int i = 0; i < 90; i++) {
      PLScode[i] = ((1 - (i % 2) * 2) * PLHeader[i].re -
          PLHeader[i].im) / 2;
   }
}

// PLScode should be 90 symbols long
void ETSI_Baseclass::descramblePLScode(float* PLScode) {

   unsigned long long mask = 1ull << 63;
   // First 26 symbols are SOF and can be skipped over
   for (int i = 26; i < 90; i++) {
      if ((SCRAMBLE_CODE_ & mask) != 0) {
         PLScode[i] *= -1;
      }
      mask = mask >> 1;
   }
}

// data should be 90 symbols long
// higher return value means lower similarity
float ETSI_Baseclass::getSimilarity(unsigned long long
    reference, float* data) {

   int cur_ref_symbol = 0;
   unsigned long long mask = 1ull << 63;
   float similarity = 0;

   // First 26 symbols are SOF and can be skipped over
   for (int i = 26; i < 90; i++) {
      if ((reference & mask) == 0) {
         cur_ref_symbol = -1;
      } else {
         cur_ref_symbol = 1;
      }

      similarity += fabs(cur_ref_symbol - data[i]);
      mask = mask >> 1;
   }

```

```
   return similarity;
}

// PLScode should be 90 symbols long
struct PLS_ ETSI_Baseclass::decodePLScode(float* PLScode) {

   struct PLS_ PLS;

   float current_sim = 0;
   float best_match_sim = FLT_MAX;
   int best_match_index = -1;

   for (int i = 0; i < 128; i++) {
      current_sim = getSimilarity(DECODE_PLS_TABLE[i],
          PLScode);
      if (current_sim < best_match_sim) {
         best_match_index = i;
         best_match_sim = current_sim;
      }
   }

   PLS.MODCOD = best_match_index >> 2;
   PLS.TYPE = best_match_index & 3;

   return PLS;
}

signed char ETSI_Baseclass::getModulation(char MODCOD) {

   switch (MODCOD) {
   case 0: // Dummy PLFRAME
      return 0;
   case 1:
   case 2:
   case 3:
   case 4:
   case 5:
   case 6:
   case 7:
   case 8:
   case 9:
   case 10:
   case 11:
      return QPSK;
   case 12:
   case 13:
   case 14:
   case 15:
   case 16:
```

```
    case 17:
        return PSK_8;
    case 18:
    case 19:
    case 20:
    case 21:
    case 22:
    case 23:
        return APSK_16;
    case 24:
    case 25:
    case 26:
    case 27:
    case 28:
        return APSK_32;
    default: //Unknown MODCOD value
        return -1;
    }
}

signed char ETSI_Baseclass::getCodeRate(char MODCOD) {

    switch (MODCOD) {
    case 0:
        return 0;
    case 1:
        return FEC_1_4;
    case 2:
        return FEC_1_3;
    case 3:
        return FEC_2_5;
    case 4:
        return FEC_1_2;
    case 5:
    case 12:
        return FEC_3_5;
    case 6:
    case 13:
    case 18:
        return FEC_2_3;
    case 7:
    case 14:
    case 19:
    case 24:
        return FEC_3_4;
    case 8:
    case 20:
    case 25:
        return FEC_4_5;
```

```
   case 9:
   case 15:
   case 21:
   case 26:
      return FEC_5_6;
   case 10:
   case 16:
   case 22:
   case 27:
      return FEC_8_9;
   case 11:
   case 17:
   case 23:
   case 28:
      return FEC_9_10;
   default: //Unknown MODCOD value
      return -1;
   }
}

bool ETSI_Baseclass::getFrameSize(char TYPE) {

   return TYPE >> 1;
}

bool ETSI_Baseclass::getPilotFields(char TYPE) {

   return TYPE % 2;
}

unsigned long long ETSI_Baseclass::encodePLS(struct PLS_
    PLS) {

   bool row1[32] = {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,
         1,0,1,0,1,0,1,0,1,0,1,0,1};
   bool row2[32] = {0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,
         1,0,0,1,1,0,0,1,1,0,0,1,1};
   bool row3[32] = {0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,
         0,1,1,1,1,0,0,0,0,1,1,1,1};
   bool row4[32] = {0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,
         0,0,0,0,0,1,1,1,1,1,1,1,1};
   bool row5[32] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,
         1,1,1,1,1,1,1,1,1,1,1,1,1};
   bool row6[32] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
         1,1,1,1,1,1,1,1,1,1,1,1,1};

   int MODCOD = PLS.MODCOD;
   int TYPE = PLS.TYPE;

```

```
    bool input[7] = { 16 & MODCOD, 8 & MODCOD, 4 & MODCOD,
           2 & MODCOD, 1 & MODCOD, 2 & TYPE, 1 & TYPE };

    bool result[64];

    for (int i = 0; i < 32; i++) {
       result[i * 2] = row1[i] * input[0] ^ row2[i]
             * input[1] ^ row3[i] * input[2] ^ row4[i]
             * input[3] ^ row5[i] * input[4] ^ row6[i]
             * input[5];
       result[i * 2 + 1] = result[i * 2] ^ input[6];
    }

    long long result_long = 0;
    for (int i = 0; i < 64; i++) {
       result_long = result_long << 1;
       result_long += result[i];
    }

    return result_long;
}

unsigned long long ETSI_Baseclass::scramblePLScode(unsigned
     long long PLScode) {

    return PLScode ^ SCRAMBLE_CODE_;
}

void ETSI_Baseclass::modulatePLHeader(unsigned long long
    PLScode, symb_cart_float* PLHeader) {

    unsigned long long mask = 1;
    mask = mask << 63;
    bool bit;

    // Get modulated SOF
    getModulatedSOF(PLHeader);

    for (int i = 0; i < 64; i++) {
       // Modulate PLScode
       bit = (mask >> i) & PLScode;
       if (i % 2 == 0) {
          PLHeader[i + 26].im = 1 - 2 * bit;
          PLHeader[i + 26].re = -PLHeader[i + 26].im;
       } else {
          PLHeader[i + 26].im = 1 - 2 * bit;
          PLHeader[i + 26].re = PLHeader[i + 26].im;
       }
    }
```

```
}
```

A.2 PLFRAME Handling

A.2.1 PLFRAME Insertion

Listing A.5: Insert PLHEADER (Header file)

```
/*
 * ETSI_InsertPLFrame.h
 *
 *  Created on: 17.08.2011
 *      Author: Johannes Pribyl
 */

#ifndef ETSI_INSERT_H_
#define ETSI_INSERT_H_

#include "ETSI_Baseclass.h"

class InsertPLFrame: public Module, protected ETSI_Baseclass {
public:
   void setParam(Module *module, std::string str, float val);
   void init(Module *module, int loopCnt);
   void getInfo(std::string* info);
   void execute(Module *module, BurstContainer* burst, int loopCnt);
   void getResult(Module *module, std::string* head, std::string* result,
         bool finalResult);
   unsigned char getInterfaceMode() {
      return IM_SYMBOL_CARTESIAN_FLOAT;
   }
   void cleanup() {
   }

private:
   bool active_;
   bool frame_size_;
   bool insert_pilots_;
   char getMODCOD(unsigned char modulation, unsigned char codeRate);
   char getTYPE(bool frame_size, bool pilots);
   void insertPilotAt(unsigned long index, BurstContainer* burst);
};
```

```
#endif /* ETSI_INSERT_H_ */
```

Listing A.6: Insert PLHEADER (Main file)

```
/*
 * ETSI_InsertPLFrame.cpp
 *
 *  Created on: 17.08.2011
 *      Author: Johannes Pribyl
 */

#include "ETSI_InsertPLFrame.h"

using namespace std;

char InsertPLFrame::getMODCOD(unsigned char modulation,
    unsigned char codeRate) {

   if (modulation == QPSK) {
      if (codeRate == FEC_1_4) {
         return 1;
      } else if (codeRate == FEC_1_3) {
         return 2;
      } else if (codeRate == FEC_2_5) {
         return 3;
      } else if (codeRate == FEC_1_2) {
         return 4;
      } else if (codeRate == FEC_3_5) {
         return 5;
      } else if (codeRate == FEC_2_3) {
         return 6;
      } else if (codeRate == FEC_3_4) {
         return 7;
      } else if (codeRate == FEC_4_5) {
         return 8;
      } else if (codeRate == FEC_5_6) {
         return 9;
      } else if (codeRate == FEC_8_9) {
         return 10;
      } else if (codeRate == FEC_9_10) {
         return 11;
      } else {
         cerr << "ERROR:␣Unsupported␣code␣rate␣for␣this␣
             modulation!\n";
         exit(EXIT_FAILURE);
      }
   } else if (modulation == PSK_8) {
      if (codeRate == FEC_3_5) {
         return 12;
```

```
      } else if (codeRate == FEC_2_3) {
         return 13;
      } else if (codeRate == FEC_3_4) {
         return 14;
      } else if (codeRate == FEC_5_6) {
         return 15;
      } else if (codeRate == FEC_8_9) {
         return 16;
      } else if (codeRate == FEC_9_10) {
         return 17;
      } else {
         cerr << "ERROR: Unsupported code rate for this modulation!\n";
         exit(EXIT_FAILURE);
      }
   } else if (modulation == APSK_16) {
      if (codeRate == FEC_2_3) {
         return 18;
      } else if (codeRate == FEC_3_4) {
         return 19;
      } else if (codeRate == FEC_4_5) {
         return 20;
      } else if (codeRate == FEC_5_6) {
         return 21;
      } else if (codeRate == FEC_8_9) {
         return 22;
      } else if (codeRate == FEC_9_10) {
         return 23;
      } else {
         cerr << "ERROR: Unsupported code rate for this modulation!\n";
         exit(EXIT_FAILURE);
      }
   } else if (modulation == APSK_32) {
      if (codeRate == FEC_3_4) {
         return 24;
      } else if (codeRate == FEC_4_5) {
         return 25;
      } else if (codeRate == FEC_5_6) {
         return 26;
      } else if (codeRate == FEC_8_9) {
         return 27;
      } else if (codeRate == FEC_9_10) {
         return 28;
      } else {
         cerr << "ERROR: Unsupported code rate for this modulation!\n";
         exit(EXIT_FAILURE);
      }
```

```
    } else {
        cerr << "ERROR: Unsupported modulation!\n";
        exit(EXIT_FAILURE);
    }
}

char InsertPLFrame::getTYPE(bool frame_size, bool pilots) {

    return frame_size * 2 + pilots;
}

void InsertPLFrame::insertPilotAt(unsigned long index, BurstContainer* burst) {

    // Move contents of burst->bodySymbolCartFloat 36 places
    for (unsigned long i = burst->header.containerBodyLength; i >= index; i--) {
        burst->bodySymbolCartFloat[i + 36].re
                = burst->bodySymbolCartFloat[i].re;
        burst->bodySymbolCartFloat[i + 36].im
                = burst->bodySymbolCartFloat[i].im;
    }

    // Insert symbols into burst->bodySymbolCartFloat
    for (unsigned long i = index; i < index + 36; i++) {
        // ETSI standard requires I=Q=1/sqrt(2), but this was changed to I=Q=1 due to compatibility issues with older parts of this simulator
        burst->bodySymbolCartFloat[i].re = 1;
        burst->bodySymbolCartFloat[i].im = 1;
    }
    burst->header.containerBodyLength += 36;
}

void InsertPLFrame::setParam(Module *module, std::string str, float val) {

    if (str == "active") {
        if (val == 0) {
            active_ = false;
        } else {
            active_ = true;
        }
    }
    if (str == "insert_pilots") {
        if (val == 0) {
            insert_pilots_ = false;
        } else {
            insert_pilots_ = true;
```

```
        }
    }
    if (str == "frame_size") {
        if (val == 0) {
            frame_size_ = false;
        } else {
            frame_size_ = true;
        }
    }
}

void InsertPLFrame::init(Module *module, int loopCnt) {
}

void InsertPLFrame::getInfo(std::string *info) {
}

void InsertPLFrame::execute(Module *module, BurstContainer *burst, int loopCnt) {

    if (not active_)
        return;

    struct PLS_ PLS;

    PLS.MODCOD = getMODCOD(burst->header.modulation, burst->header.codeRate);
    PLS.TYPE = getTYPE(frame_size_, insert_pilots_);
    long long PLScode = encodePLS(PLS);
    PLScode = scramblePLScode(PLScode);
    symb_cart_float PLHeader[90];
    modulatePLHeader(PLScode, PLHeader);

    // Move contents of burst->bodySymbolCartFloat 90 places
    for (long i = burst->header.containerBodyLength; i >= 0; i--) {
        burst->bodySymbolCartFloat[i + 90].re
                = burst->bodySymbolCartFloat[i].re;
        burst->bodySymbolCartFloat[i + 90].im
                = burst->bodySymbolCartFloat[i].im;
    }

    // Insert symbols into burst->bodySymbolCartFloat
    for (int i = 0; i < 90; i++) {
        burst->bodySymbolCartFloat[i].re = PLHeader[i].re;
        burst->bodySymbolCartFloat[i].im = PLHeader[i].im;
    }
    burst->header.containerBodyLength += 90;
    burst->header.burstLen += 90;
```

```
    // Insert pilot blocks
    if (insert_pilots_) {

        // The first pilot should be inserted 16 SLOTS (1
            SLOT = 90 symbols) after the PLHEADER (length of
            PLHEADER = 90 symbols)
        const unsigned int START_POINT = 90 + 90 * 16;

        // The next pilot should be inserted 16 SLOTS after
            the current pilot (length of one pilot = 36
            symbols)
        const unsigned int INCREMENT = 90 * 16 + 36;

        for (unsigned long i = START_POINT; i
                < burst->header.containerBodyLength; i +=
                    INCREMENT) {
            insertPilotAt(i, burst);
        }
    }
}

void InsertPLFrame::getResult(Module *module, std::string *
    head, std::string *result, bool finalResult) {
}
```

A.2.2 PLFRAME Removal

Listing A.7: Remove PLHEADER (Header file)

```
/*
 * ETSI_RemovePLFrame.h
 *
 *  Created on: 31.08.2011
 *      Author: Johannes Pribyl
 */

#ifndef ETSI_REMOVE_H_
#define ETSI_REMOVE_H_

#include "ETSI_Baseclass.h"

class RemovePLFrame: public Module, protected
    ETSI_Baseclass {
public:
    RemovePLFrame() {
        active_ = false;
        phase_correction_ = false;
    }
```

```
    ~RemovePLFrame() {
    }

    void setParam(Module *module, std::string str, float val);
    void init(Module *module, int loopCnt);
    void getInfo(std::string* info);
    void execute(Module *module, BurstContainer* burst, int loopCnt);
    void getResult(Module *module, std::string* head, std::string* result,
            bool finalResult);
    unsigned char getInterfaceMode() {
        return IM_SYMBOL_CARTESIAN_FLOAT;
    }
    void cleanup() {
    }

private:
    bool active_;
    bool phase_correction_;
    void removePilotAt(unsigned long index, BurstContainer* burst);
};

#endif /* ETSI_REMOVE_H_ */
```

Listing A.8: Remove PLHEADER (Main file)

```
/*
 * ETSI_RemovePLFrame.cpp
 *
 *  Created on: 21.09.2011
 *      Author: Johannes Pribyl
 */

#include "ETSI_RemovePLFrame.h"

using namespace std;

void RemovePLFrame::removePilotAt(unsigned long index, BurstContainer* burst) {

    for (unsigned long i = index; i < (burst->header.containerBodyLength - 36); i++) {
        burst->bodySymbolCartFloat[i].re
                = burst->bodySymbolCartFloat[i + 36].re;
        burst->bodySymbolCartFloat[i].im
                = burst->bodySymbolCartFloat[i + 36].im;
    }
```

```
    burst->header.containerBodyLength -= 36;
}

void RemovePLFrame::setParam(Module *module, std::string str, float val) {

    if (str == "active") {
        if (val == 0) {
            active_ = false;
        } else {
            active_ = true;
        }
    }
    if (str == "phase_correction") {
        if (val == 0) {
            phase_correction_ = false;
        } else {
            phase_correction_ = true;
        }
    }
}

void RemovePLFrame::init(Module *module, int loopCnt) {
}

void RemovePLFrame::getInfo(std::string *info) {
}

void RemovePLFrame::execute(Module *module, BurstContainer *burst, int loopCnt) {

    if (not active_)
        return;

    // Estimate and correct the phase error using the SOF data
    if (phase_correction_) {
        correctPhaseError(burst->bodySymbolCartFloat,
                burst->bodySymbolPolarFloat, burst->header.containerBodyLength);
    }

    // Read PLHeader from burst->bodySymbolCartFloat
    symb_cart_float PLHeader[90];
    for (int i = 0; i < 90; i++) {
        PLHeader[i].re = burst->bodySymbolCartFloat[i].re;
        PLHeader[i].im = burst->bodySymbolCartFloat[i].im;
    }

```

```
  // Remove PLHEADER from burst->bodySymbolCartFloat
  for (unsigned long i = 0; i < (burst->header.containerBodyLength - 90); i++) {
    burst->bodySymbolCartFloat[i].re
        = burst->bodySymbolCartFloat[i + 90].re;
    burst->bodySymbolCartFloat[i].im
        = burst->bodySymbolCartFloat[i + 90].im;
  }
  burst->header.containerBodyLength -= 90;
  burst->header.burstLen -= 90;

  // Demodulate PLHeader
  float PLScode[90];
  demodulatePLHeader(PLHeader, PLScode);

  // Descramble PLScode
  descramblePLScode(PLScode);

  // Decode PLScode
  struct PLS_ PLS = decodePLScode(PLScode);

  // Remove pilot blocks from burst->bodySymbolCartFloat
  if (getPilotFields(PLS.TYPE)) {

    // The first pilot should be inserted after 16 SLOTS (1 SLOT = 90 symbols, PLHEADER was already removed!)
    const unsigned int START_POINT = 90 * 16;

    // The next pilot should be inserted 16 SLOTS after the current pilot
    const unsigned int INCREMENT = 90 * 16;

    for (unsigned long i = START_POINT; i
        < burst->header.containerBodyLength; i += INCREMENT) {
      removePilotAt(i, burst);
    }
  }
}

void RemovePLFrame::getResult(Module *module, std::string *head, std::string *result, bool finalResult) {
}
```

A.3 Analysis

A.3.1 Analyze PLFRAME

Listing A.9: Analyze PLFRAME (Header file)

```
/*
 * ETSI_AnalyzePLFrame.h
 *
 *  Created on: 21.10.2011
 *      Author: Johannes Pribyl
 */

#ifndef ETSI_ANALYZE_H_
#define ETSI_ANALYZE_H_

#include "ETSI_Baseclass.h"

class AnalyzePLFrame: public Module, protected
    ETSI_Baseclass {
public:
   AnalyzePLFrame() {
      tp1_ = 0;
      tp1_set_ = false;
      tp2_ = 0;
      tp2_set_ = false;
      total_loops_ = 0;
      failed_loops_ = 0;
      phase_correction_ = false;
   }
   ~AnalyzePLFrame() {
   }

   void setParam(Module *module, std::string str, float val
       );
   void init(Module *module, int loopCnt);
   void getInfo(std::string* info);
   void execute(Module *module, BurstContainer* burst, int
       loopCnt);
   void getResult(Module *module, std::string* head, std::
       string* result,
          bool finalResult);
   unsigned char getInterfaceMode() {
      return IM_ANY;
   }
   void cleanup() {
   }

private:
   int tp1_;
```

```
    bool tp1_set_;
    int tp2_;
    bool tp2_set_;
    bool phase_correction_;

    unsigned long long total_loops_;
    unsigned long long failed_loops_;
};

#endif /* ETSI_ANALYZE_H_ */
```

Listing A.10: Analyze PLFRAME (Main file)

```
/*
 * ETSI_AnalyzePLFrame.cpp
 *
 *  Created on: 21.10.2011
 *      Author: Johannes Pribyl
 */

#include "ETSI_AnalyzePLFrame.h"

using namespace std;

void AnalyzePLFrame::setParam(Module *module, std::string
    str, float val) {

    ostringstream s;
    SimulationControl* simCtrl;
    simCtrl = dynamic_cast<SimulationControl*> (module);
    error_handler.checkCast(simCtrl, "AnalyzePLFrame");

    if (str == "tp1") {
        tp1_ = (int) val;
        if (simCtrl->getTestPointSet(simCtrl->
            getTestpointModuleIndex(tp1_))) {
            tp1_set_ = true;
            return;
        }
        s << tp1_;
        error_handler.throwException(
                "ETSI_AnalyzePLFrame::setParam:␣TESTPOINT␣#" +
                    s.str()
                        + "␣does␣not␣exist!");
    }
    if (str == "tp2") {
        tp2_ = (int) val;
        if (simCtrl->getTestPointSet(simCtrl->
            getTestpointModuleIndex(tp2_))) {
            tp2_set_ = true;
```

```
            return;
        }
        s << tp2_;
        error_handler.throwException(
                "ETSI_AnalyzePLFrame::setParam:␣TESTPOINT␣#" +
                    s.str()
                    + "␣does␣not␣exist!");
    }
    if (str == "phase_correction") {
        if (val == 0) {
            phase_correction_ = false;
        } else {
            phase_correction_ = true;
        }
        return;
    }

    error_handler.throwException("ETSI_AnalyzePLFrame::
        setParam();␣" + str
            + "␣is␣not␣a␣parameter␣of␣ETSI_AnalyzePLFrame␣
                module!");
}

void AnalyzePLFrame::init(Module *module, int loopCnt) {

    total_loops_ = 0;
    failed_loops_ = 0;

    if (tp1_set_ == false)
        error_handler.throwException("AnalyzePLFrame::init();
            ␣tp1␣is␣not␣set!");
    if (tp2_set_ == false)
        error_handler.throwException("AnalyzePLFrame::init();
            ␣tp2␣is␣not␣set!");
}

void AnalyzePLFrame::getInfo(std::string *info) {
}

void AnalyzePLFrame::execute(Module *module, BurstContainer
     *burst, int loopCnt) {

    BurstContainer* b1;
    BurstContainer* b2;
    SimulationControl* simCtrl;

    simCtrl = dynamic_cast<SimulationControl*> (module);
    error_handler.checkCast(simCtrl, "AnalyzePLFrame");

```

```
b1 = simCtrl->getStoredBurst(simCtrl->
    getTestpointModuleIndex(tp1_));
b2 = simCtrl->getStoredBurst(simCtrl->
    getTestpointModuleIndex(tp2_));

total_loops_++;

//------ Burst1 ------------------------------------------
//Read PLHeader from burst->bodySymbolCartFloat
symb_cart_float PLHeader[90];
for (int i = 0; i < 90; i++) {
   PLHeader[i].re = b1->bodySymbolCartFloat[i].re;
   PLHeader[i].im = b1->bodySymbolCartFloat[i].im;
}

//Demodulate PLHeader
float PLScode[90];
demodulatePLHeader(PLHeader, PLScode);

//Descramble PLScode
descramblePLScode(PLScode);

//Decode PLScode
struct PLS_ PLS1 = decodePLScode(PLScode);

//------ Burst2 ------------------------------------------

//Estimate and correct the phase error using the SOF
    data
if (phase_correction_) {
   correctPhaseError(b2->bodySymbolCartFloat, b2->
       bodySymbolPolarFloat,
          b2->header.containerBodyLength);
}

//Read PLHeader from burst->bodySymbolCartFloat
for (int i = 0; i < 90; i++) {
   PLHeader[i].re = b2->bodySymbolCartFloat[i].re;
   PLHeader[i].im = b2->bodySymbolCartFloat[i].im;
}

//Demodulate PLHeader
demodulatePLHeader(PLHeader, PLScode);

//Descramble PLScode
descramblePLScode(PLScode);

//Decode PLScode
struct PLS_ PLS2 = decodePLScode(PLScode);
```

```
    if (getModulation(PLS1.MODCOD) != getModulation(PLS2.MODCOD)) {
        failed_loops_++;
        return;
    }

    if (getCodeRate(PLS1.MODCOD) != getCodeRate(PLS2.MODCOD)) {
        failed_loops_++;
        return;
    }

    if (getFrameSize(PLS1.TYPE) != getFrameSize(PLS2.TYPE)) {
        failed_loops_++;
        return;
    }

    if (getPilotFields(PLS1.TYPE) != getPilotFields(PLS2.TYPE)) {
        failed_loops_++;
        return;
    }
}

void AnalyzePLFrame::getResult(Module *module, std::string *head, std::string *result, bool finalResult) {

    *head = "AnalyzePLFrame\n";
    ostringstream s;

    if (total_loops_ > 0) {
        s << "PLFrame " << (float) failed_loops_ / (float) total_loops_ * 100
            << " %\n";
    } else {
        s << "-)\n";
    }

    if (finalResult == true) {
        total_loops_ = 0;
        failed_loops_ = 0;
    }
    *result = s.str();
}
```

A.3.2 Analyze SNR

Listing A.11: Analyze SNR (Header file)

```
/*
 * ETSI_AnalyzeSNR.h
 *
 *  Created on: 31.1.2012
 *      Author: Johannes Pribyl
 */

#ifndef ETSI_ANALYZE_SNR_H_
#define ETSI_ANALYZE_SNR_H_

#include "ETSI_Baseclass.h"

class AnalyzeSNR: public Module, protected ETSI_Baseclass {
public:
   AnalyzeSNR() {
      ref_tp_ = 0;
      ref_tp_set_ = false;

      data_aided_ = true;
      data_aided_set_ = false;

      plframe_used_ = true;
      plframe_used_set_ = false;

      phase_correction_ = false;
      phase_correction_set_ = false;

      pilots_used_ = false;

      mse_sum_ = 0;
      snr_sum_ = 0;
      snr_ = 0;

      total_loops_ = 0;
      burst_len_ = 0;
      len_ = 0;
   }
   ~AnalyzeSNR() {}

   void setParam(Module *module, std::string str, float val
       );
   void init(Module *module, int loopCnt);
   void getInfo(std::string* info);
   void execute(Module *module, BurstContainer* burst, int
       loopCnt);
```

```
    void getResult(Module *module, std::string* head, std::string* result, bool finalResult);
    unsigned char getInterfaceMode() {return IM_ANY;}
    void cleanup() {}

    unsigned int getUsableDataAmount(unsigned int container_length);
    bool isUsableDataIndex(unsigned int index);
    bool isOverheadDataIndex(unsigned int index);


private:
    int ref_tp_;
    bool ref_tp_set_;

    bool data_aided_;
    bool data_aided_set_;

    bool plframe_used_;
    bool plframe_used_set_;

    bool phase_correction_;
    bool phase_correction_set_;

    bool pilots_used_;

    double mse_sum_;
    double snr_sum_;
    double snr_;

    unsigned int total_loops_;
    unsigned int burst_len_;
    unsigned int len_;
};

#endif /* ETSI_ANALYZE_SNR_H_ */
```

Listing A.12: Analyze SNR (Main file)

```
/*
 * ETSI_AnalyzeSNR.cpp
 *
 *  Created on: 31.1.2012
 *      Author: Johannes Pribyl
 */

#include "ETSI_AnalyzeSNR.h"

using namespace std;

```

```
void AnalyzeSNR::setParam(Module *module, std::string str, float val) {

   ostringstream s;
   SimulationControl* simCtrl;
   simCtrl = dynamic_cast<SimulationControl*> (module);
   error_handler.checkCast(simCtrl, "AnalyzeSNR");

   if (str == "reference_tp") {
      ref_tp_ = (int) val;
      if (simCtrl->getTestPointSet(simCtrl->getTestpointModuleIndex(ref_tp_))) {
         ref_tp_set_ = true;
         return;
      }
      s << ref_tp_;
      error_handler.throwException("AnalyzeSNR::setParam: TESTPOINT #"
            + s.str() + " does not exist!");
   }
   if (str == "data_aided") {
      if (val == 0) {
         data_aided_ = false;
      } else {
         data_aided_ = true;
      }
      data_aided_set_ = true;
      return;
   }
   if (str == "plframe_used") {
      if (val == 0) {
         plframe_used_ = false;
      } else {
         plframe_used_ = true;
      }
      plframe_used_set_ = true;
      return;
   }
   if (str == "phase_correction") {
      if (val == 0) {
         phase_correction_ = false;
      } else {
         phase_correction_ = true;
      }
      phase_correction_set_ = true;
      return;
   }

   error_handler.throwException("v::setParam(); " + str
```

```
        + "␣is␣not␣a␣parameter␣of␣AnalyzeSNR␣module!");
}

void AnalyzeSNR::init(Module *module, int loopCnt) {

   total_loops_ = 0;
   if (data_aided_set_ == false)
      error_handler.throwException(
            "AnalyzeSNR::init();␣Parameter␣data␣aided␣is␣not␣set!");

   if (data_aided_ == true) {

      if ((ref_tp_set_ == false))
         error_handler.throwException(
               "AnalyzeSNR::init();␣reference␣tp␣is␣not␣set!");
      if ((phase_correction_set_ == false))
         error_handler.throwException(
               "AnalyzeSNR::init();␣Parameter␣phase_correction␣is␣not␣set!");
   }

   if (plframe_used_set_ == false)
      error_handler.throwException(
            "AnalyzeSNR::init();␣Parameter␣plframe_used␣is␣not␣set!");
}

void AnalyzeSNR::getInfo(std::string *info) {
}

void AnalyzeSNR::execute(Module *module, BurstContainer *burst, int loopCnt) {

   SimulationControl* simCtrl;
   simCtrl = dynamic_cast<SimulationControl*> (module);
   error_handler.checkCast(simCtrl, "AnalyzeSNR");

   double estimated_snr = 0;
   burst_len_ = burst->header.containerBodyLength;

   if (plframe_used_) {
      //Decode PLHEADER to see if pilots were used
      symb_cart_float* source_array;
      symb_cart_float PLHeader[90];
      float PLScode[90];

      if (data_aided_) {
```

```
      source_array
          = simCtrl->getStoredBurst(simCtrl->getTestpointModuleIndex(ref_tp_))->bodySymbolCartFloat;
    } else {
      source_array = burst->bodySymbolCartFloat;
    }

    for (int i = 0; i < 90; i++) {
      PLHeader[i].re = source_array[i].re;
      PLHeader[i].im = source_array[i].im;
    }
    demodulatePLHeader(PLHeader, PLScode);
    descramblePLScode(PLScode);
    struct PLS_ PLS = decodePLScode(PLScode);
    pilots_used_ = getPilotFields(PLS.TYPE);
  }

  if (data_aided_) {
    //Get reference data
    BurstContainer* ref_b;
    ref_b = simCtrl->getStoredBurst(simCtrl->getTestpointModuleIndex(
        ref_tp_));

    //Check if both bursts have the same body length
    if (ref_b->header.containerBodyLength
          != burst->header.containerBodyLength) {
      error_handler.throwException(
          "AnalyzeSNR::execute: Burst and reference burst have different body lengths!");
    }

    unsigned int length = getUsableDataAmount(
          burst->header.containerBodyLength);

    //Copy usable data into local arrays
    symb_cart_float body_cart[length];
    symb_cart_float ref_body_cart[length];
    unsigned int array_index = 0;
    for (unsigned int i = 0; i < burst->header.containerBodyLength; i++) {
      if (isUsableDataIndex(i)) {
        body_cart[array_index] = burst->bodySymbolCartFloat[i];
        ref_body_cart[array_index] = ref_b->bodySymbolCartFloat[i];
        array_index++;
      }
```

```
        }
        len_ = length;

        //Estimate and correct the phase error using the SOF data
        if (phase_correction_) {
            symb_polar_float body_cart_pol[length];
            if (pilots_used_) {
                correctPhaseError(body_cart, body_cart_pol, 90);
                unsigned int start_pos = 90;
                for (unsigned int i = 0; i < ((length - 90) / 36); i++) {
                    correctPhaseError(body_cart + start_pos, body_cart_pol, 36);
                    start_pos += 36;
                }
            } else {
                correctPhaseError(body_cart, body_cart_pol, length);
            }
        }

        //Estimate SNR
        if (plframe_used_) {
            estimated_snr = estimateSNRda_plframe(ref_body_cart, body_cart,
                    length);
        } else {
            estimated_snr = estimateSNRda(ref_body_cart, body_cart, length);
        }

    } else { //data_aided_ == false
        unsigned int length = getUsableDataAmount(
                burst->header.containerBodyLength);

        symb_cart_float body_cart[length];

        unsigned int array_index = 0;

        for (unsigned int i = 0; i < burst->header.containerBodyLength; i++) {
            if (isUsableDataIndex(i)) {
                body_cart[array_index] = burst->bodySymbolCartFloat[i];
                array_index++;
            }
        }
```

```
        len_ = length;

        switch (burst->header.modulation) {
        case QPSK:
        case PSK_8:
            estimated_snr = estimateSNRmbMPSK(body_cart, length);
            break;
        case APSK_16:
            estimated_snr = estimateSNRmb16APSK(body_cart, length,
                    burst->header.codeRate);
            break;
        case APSK_32:
            estimated_snr = estimateSNRmb32APSK(body_cart, length,
                    burst->header.codeRate, 2.0);
            break;
        default:
            stringstream sstm;
            sstm << "AnalyzeSNR::execute(); Unknown modulation ("
                    << (int) burst->header.modulation << ")!";
            error_handler.throwException(sstm.str());
        }
    }

    if (!isnan(estimated_snr) && !isinf(estimated_snr)) {
        total_loops_++;
        snr_ = pow(10, burst->snr / 10);
        mse_sum_ += pow(estimated_snr - snr_, 2);
        snr_sum_ += estimated_snr;
    }
}

//Calculate the amount of symbols we will be able to use for SNR estimation
unsigned int AnalyzeSNR::getUsableDataAmount(unsigned int container_length) {

    if (plframe_used_ == false)
        return container_length;

    //1) One PLHEADER (90 Symbols)
    //2) Every 16 SLOTS (1 SLOT = 90 symbols) there is one pilot block of 36 symbols
    unsigned int pilots_count = 0;

    if (pilots_used_ == false) {
```

```
        pilots_count = 0;
    } else {
        pilots_count = floor((container_length - 90) / (90 *
            16 + 36));
    }

    unsigned int overhead_data = 90 + pilots_count * 36;

    if (data_aided_ == true) {
        return overhead_data;
    } else {
        return container_length - overhead_data;
    }
}

bool AnalyzeSNR::isUsableDataIndex(unsigned int index) {

    bool is_overhead_data_index = isOverheadDataIndex(index)
        ;

    if (data_aided_ == true) {
        if (plframe_used_ == false)
            return true;
        return is_overhead_data_index;
    } else {
        return !is_overhead_data_index;
    }
}

bool AnalyzeSNR::isOverheadDataIndex(unsigned int index) {

    if (plframe_used_ == false)
        return false;

    if (index < 90)
        return true;

    if (pilots_used_ == false)
        return false;

    //One "block" is 16 SLOTS + one pilot block
    const unsigned int block_length = 90 * 16 + 36;
    index -= 90;
    index = index % block_length;

    if (index < (90 * 16)) {
        return false;
    } else {
        return true;
```

```
    }
}

void AnalyzeSNR::getResult(Module *module, std::string *
    head, std::string *result, bool finalResult) {

    *head = "AnalyzeSNR\n";
    ostringstream s;

    if (total_loops_ > 0) {
        double snr = snr_sum_ / total_loops_;
        double mse = mse_sum_ / total_loops_;
        double norm_mse = mse / pow(snr_, 2);
        double stddev = sqrt(mse);

        s << "Std.Dev./Norm.MSE/Est.SNR:␣" << stddev << "␣"
            << norm_mse << "␣"
                << snr;
        cout << "total_loops_:␣" << total_loops_ << "\test.
            len.:␣" << len_
                << "\tsnr_:␣" << snr_ << "\test_snr_:␣" << snr
                    << "\n";
    } else {
        s << "-)\n";
        cout << "no␣loops\n";
    }

    if (finalResult == true) {
        total_loops_ = 0;
        mse_sum_ = 0;
        snr_sum_ = 0;
    }
    *result = s.str();
}
```

A.4 Other Code

A.4.1 Testdata

Listing A.13: Testdata Setter (Header file)

```
/*
 * ETSI_TestDataSetter.h
 *
 *  Created on: 10.02.2012
 *      Author: Johannes Pribyl
 */

```

```
#ifndef ETSI_TESTDATASETTER_H_
#define ETSI_TESTDATASETTER_H_

#include "modemSim.h"
#include "utils.h"

class TestDataSetter: public Module {
public:
   void setParam(Module *module, std::string str, float val);
   void init(Module *module, int loopCnt);
   void getInfo(std::string* info);
   void execute(Module *module, BurstContainer* burst, int loopCnt);
   void getResult(Module *module, std::string* head, std::string* result,
         bool finalResult);
   unsigned char getInterfaceMode() {
      return IM_ANY;
   }
   void cleanup() {
   }
};

#endif /* ETSI_TESTDATASETTER_H_ */
```

Listing A.14: Testdata Setter (Main file)

```
/*
 * ETSI_TestDataSetter.cpp
 *
 *  Created on: 10.02.2012
 *      Author: Johannes Pribyl
 */

#include "ETSI_TestDataSetter.h"

using namespace std;

void TestDataSetter::setParam(Module *module, std::string str, float val) {
}

void TestDataSetter::init(Module *module, int loopCnt) {
}

void TestDataSetter::getInfo(std::string *info) {
}

```

```cpp
void TestDataSetter::execute(Module *module, BurstContainer *burst, int loopCnt) {

    if (burst->header.modulation > 5 || burst->header.modulation < 1) {
        error_handler.throwException(
                "TestDataSetter::execute(); Please use only modulations 1-5. 5 is used for 32-APSK!");
    }

    int length = floor(pow(10, burst->header.burstLen * 0.5) + 0.5);
    burst->header.burstLen = ceil(length * burst->header.modulation / 8.0);

    if (burst->header.modulation == 5) {
        burst->header.modulation = 6;
    }

    burst->header.containerBodyLength = burst->header.burstLen;
}

void TestDataSetter::getResult(Module *module, std::string *head, std::string *result, bool finalResult) {
}
```

Appendix B

Batch Files for Simulation Automation

Due to some limitations in the simulator's sweep support, Microsoft Windows XP batch files were used, which are discussed in subsection 4.2.4.

Listing B.1: `go.bat`: Run `all_sims.bat` with low priority

```
@echo off
Start /MIN /BelowNormal all_sims.bat
```

Listing B.2: `all_sims.bat`: Execute all simfiles in the current directory

```
@echo off
cls

for %%x in (*.sim) do (

   simulator %%x > %%~nx.out
)

del output\* /q
move *.out output
move *.inf output
move *.dat output
move simulator.exe simulator_old.exe

exit
```