GSMsim
A MATLAB Implementation of a GSM Simulation Platform

Arne Norre Ekstrøm & Jan H. Mikkelsen
Institute of Electronic Systems,
Division of Telecommunications, Aalborg University,
Fredrik Bajers Vej 7A, DK-9220 Aalborg Øst, Denmark

Telephone:	+45 96 35 86 53
Fax:	+46 98 15 67 40
E-mail:	**aneks / hmi @kom.auc.dk**

December, 1997

Preface

This technical report documents a MATLAB toolbox – *GSMsim* – developed as part of a research effort on CMOS front-end RF-IC design. *GSMsim* provides a mean for evaluating the performance of both transmitter and receiver front-ends in a GSM system. The performance evaluation is of the Monte-Carlo type, and is based on a BER measure calculated by comparing a random input bit sequence and the resulting sequence estimated by the receiver. The toolbox includes baseband functionalities from both transmitter and receiver. The modular structure of the toolbox is designed for easy addition of user defined functionalities. The individual simulation parts developed are described using a mixture of pseudo code and MATLAB notations. Test results are provided whenever appropriate.

The toolbox is implemented using MATLAB Version 5.1.0.421 on SOL2. The entire MATLAB toolbox may be retrieved from the URL:

http://www.kom.auc.dk/TELE/SW-packages/matlab/GSMsim.tar.gz

It is the hope of the authors that this toolbox will turn out useful in future research projects as well as related student projects.

Arne Norre Ekstrøm & Jan Hvolgaard Mikkelsen
Aalborg University, December 1997

Forord

Denne tekniske rapport beskriver en MATLAB toolbox – *GSMsim* – udviklet som et led i forskning indenfor CMOS front-end RF-IC design. *GSMsim* åbner mulighed for at evaluere performancen af både sender og modtager front-ends i et GSM system. Performance evalueringen er af Monte-Carlo typen, og er baseret på den BER der kan udregnes ved sammenligning af en tilfældig input bit sekvens og den resulterende sekvens estimeret af modtageren. Toolboxen inkluderer baseband funktionaliteter fra både sender og modtager. Den modulære struktur af toolboxen er designet med henblik på let tilføjelse af brugerdefinerede funktioner. I dokumentationen er de individuelle dele af toolboxen beskrevet ved hjælp af en blanding af pseudo kode og MATLAB notation. Der er anført testresultater hvor forfatterne har fundet det tjeneligt.

Toolboxen er implementeret ved hjælp af MATLAB Version 5.1.0.421 på SOL2. Toolboxen kan hentes på følgende URL:

http://www.kom.auc.dk/TELE/SW-packages/matlab/GSMsim.tar.gz

Forfatterene håber at toolboxen vil vise sig anvendelig i fremtidige forskningsprojekter såvel som relaterede studenterprojekter.

Arne Norre Ekstrøm & Jan Hvolgaard Mikkelsen
Aalborg Universitet, December 1997

Contents

1

Introduction

WITHIN the last decade the high frequency electronic design community has displayed a renewed interest in CMOS (*Complementary Metal Oxide Semiconductor*) integrated circuits. This is primarily due to the cost effectiveness of CMOS implementations when compared to for instance Bipolar, BiCMOS, or GaAs (*Gallium Arsenide*) implementations. Also, CMOS design has the potential of low voltage and low power operation. These are key words of significant – and increasing – importance, especially in the design of portable handsets for cellular radio communications.

The use of CMOS is in most wireless equipment limited to DSP (*Digital Signal Processing*) applications and low frequency analog designs [18]. The potential of CMOS for high frequency applications motivates much of todays research in integrated circuit design. As a result, designs presenting high frequency CMOS applications are starting to emerge [1, 4, 5, 7, 19]. At higher frequencies the analog signal processing limitations of CMOS are more apparent [17]. Here, moving traditional analog signal processing tasks to the digital domain may prove advantageous for CMOS. Hence, to fully evaluate the performance potential of RF-IC CMOS based front-ends it is advantageous to take a system level approach. To accomplish this it is chosen to consider the GSM (*Global System for Mobile Communication*) system, as this currently is the most wide spread of all the cellular systems [2]. Also, GSM is a system with very well defined specifications.

1.1 Approach and Conceptual Transceiver Structure

The intention is hence to develop a software platform capable of generating a series of appropriate GSM data blocks and subsequently perform correct reception of these. Complex baseband representation is chosen as this reduces the required simulation sample rate and thus also the overall simulation time and memory consumption. Moreover, it is chosen to implement the tool as an MATLAB [13] toolbox, as this provides an easy entry to implementing the simulation tool. Also, an excellent graphics tool is readily at hand when using MATLAB. This makes illustrating and verifying the product an easy task.

To analyze specific front-end architectures and designs for GSM operation one just have to insert a software description of the receiver – or transmitter – prior to running the demodulator. The front-end description must, of course, comply with some predefined interface restrictions. The toolbox described consists of baseband parts only. More specificly, the parts included are illustrated in Figure 1.1, where a conceptual block diagram of a GSM transmitter and receiver system is sketched. Only the highlighted blocks are implemented.

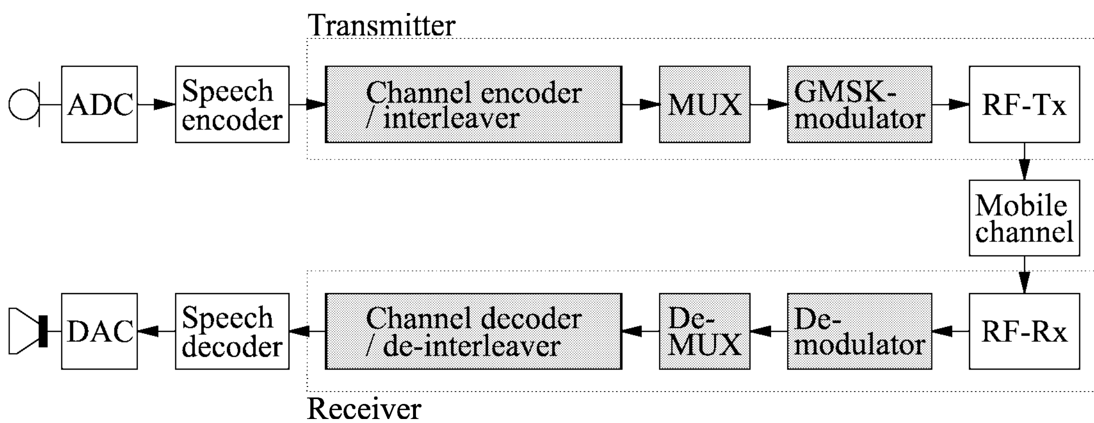


Figure 1.1: Conceptual block diagram for a GSM transmitter/receiver system. Only the six highlighted blocks are included in the toolbox.

The voice interfaces – including microphone, speech encoder/decoder, and loudspeaker – are not intended to be included in the toolbox. Instead, to supply the input signal to the channel encoder/interleaver random bits are generated, as Figure 1.2 displays. By comparing this random input sequence with the reconstructed sequence delivered by the channel decoder/de-interleaver block the BER (*Bit Error Rate*) performance of the system is estimated.

The RF-Tx, RF-Rx, and the mobile channel blocks are optional as to the closed loop structure of the toolbox. If run without any of these blocks the simulation proceeds flawless. The toolbox provides for easy inclusion of user defined RF blocks and mobile channel. The toolbox may hence be seen as a three part tool consisting of a data transmitter part, a receiver part, and an overall simulation flow control part. Each of these three separate parts consists of one or more MATLAB functions.

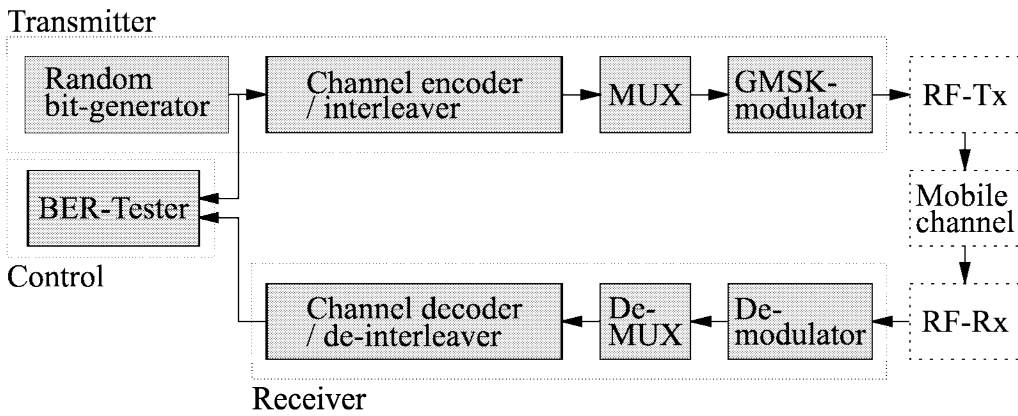


Figure 1.2: Block diagram illustrating the data structure of the implemented software. Dashed blocks are optional in the simulation runs.

1.1.1 Overall Transmitter Structure

The overall structure of the implemented transmitter is illustrated in Figure 1.3. The transmitter is, as illustrated, made up of four distinct functional blocks.

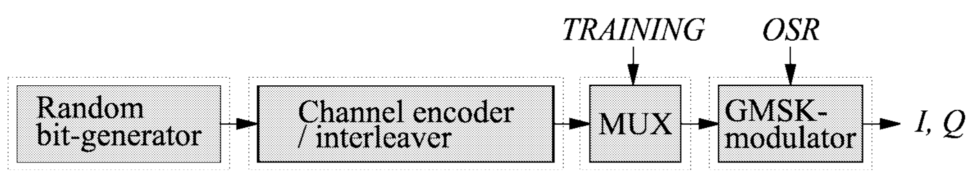


Figure 1.3: Display of the overall structure of the transmitter part of the toolbox. The input and output labels, $TRAINING, OSR, I,$ and Q all relate to actual parameters used in the implementations.

To provide an input data stream to the channel encoder/interleaver a sequence of random data bits is generated by the random bit generator. This sequence is – after processing – then accepted by the MUX which splits the incoming sequence to form a GSM normal burst. As this burst type requires that a training sequence is included this also must be supplied. This is in Figure 1.3 illustrated by the $TRAINING$ parameter. The term $TRAINING$ is also used throughout the software implementations to represent the training sequence. Upon having generated the prescribed GSM normal burst data structure the MUX returns this to the GMSK-modulator, where GMSK is short for *Gaussian Minimum Shift Keying*. The GMSK-modulator block performs a differential encoding of the incoming burst to form a NRZ (*Non Return to Zero*) sequence. This modified sequence is then subject to the actual GMSK-modulation after which, the resulting signal is represented as a complex baseband signal using the corresponding I and Q signals. The number of sample values per data bit, $OSR \cdot r_b$, is left as an user definable parameter. It is here customary to operate using four samples per bit, hence, an OSR of four is normally used [15].

The actual theory behind the transmitter blocks, the parsing of parameters, and the data flows within the transmitter implementations are described in detail in Chapter 2.

1.1.2 Overall Receiver Structure

The overall structure of the implemented data receiver is illustrated in Figure 1.4. Here three functional blocks are designed in order to implement the data receiver.

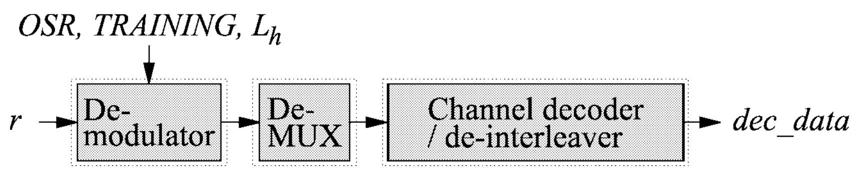


Figure 1.4: Display of the overall structure of the data receiver part of the toolbox. The input and output labels, $r, OSR, TRAINING, L_h$, and dec_data all relate to actual parameters used in the implementations.

The demodulator accepts a GSM burst, r, using a complex baseband representation. Based on this data sequence, information concerning the oversampling rate OSR, the training sequence $TRAINING$, and the desired length of the receiving filter, L_h, the demodulator determines the most probable bit sequence. This demodulated sequence is then used as input to the DeMUX where the bits are split in order to retrieve the actual data bits from the sequence. The remaining control bits and the training sequence are here discharged. As a final operation to retrieve the estimated transmitted bits channel decoding and de-interleaving is performed. It is important to note that the parameter values of OSR and $TRAINING$ used in the receiver must equal those used in the transmitter.

The parsing of parameters and data flows within the receiver functions are described in detail in Chapter 3.

In Chapter 4, the topics of installation and use of the *GSMsim* toolbox is covered.

After this very general introduction to the implemented toolbox the following two appendices provide a summary of the theory behind the functions and the actual structure of the implemented functions are also presented in detail. The blocks indicated in Figure 1.2 as being optional are not considered further in the document.

The MATLAB source code used for implementing the central parts of *GSMsim* is included in Appendix C.

2

Transmitter Background

This chapter presents the functional structure of the implemented transmitter as well as presents the individual MATLAB functions developed as part of the transmitter implementation. The overall structure of the transmitter, presented in Figure 1.3, is used to indicate where the various functions belong in the transmitter data flow. The implemented functions are described with respect to input/output parameters and the underlying theory. For a full description of the actual implementations please refer to Appendix A.

This chapter is divided into three sections. The first section describes the implemented data generator, channel encoder, interleaver and multiplexer while the second section addresses the differential encoder and the GMSK-modulator implementations. Finally, the third section describes some of the tests performed to verify the operation of the transmitter implementation.

2.1 Data Generation, Channel Encoding, Interleaving, and Multiplexing

The generation of data and the tasks of interleaving the data, performing the channel encoding, and multiplexing the resulting data segments are implemented in three separate blocks. These blocks then make sure that a correct GSM normal burst bit format structure is generated. This is done by first generating a series of random bits which, in turn, are inserted in a prescribed frame structure. To implement this combined operation four functions, `data_gen.m`, `channel_enc.m`, `interleave.m`, and `burst_g.m`, are used. These functions are shown in Figure 2.1.

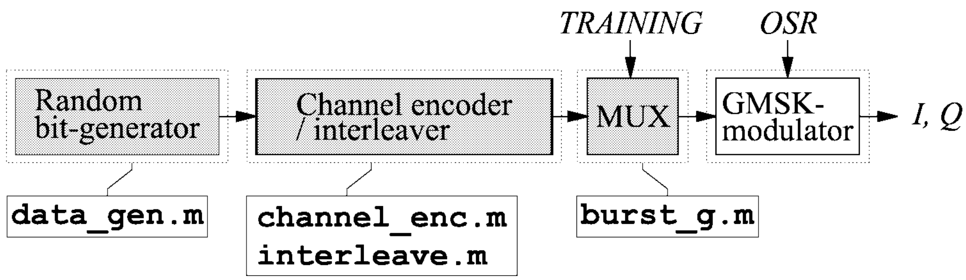


Figure 2.1: Illustration of the transmitter implementation. The relations between blocks and actual implemented functions are indicated.

2.1.1 Data Generation

The function `data_gen.m` is in fact a very simple function as it is based on the `rand` function included as a default function in MATLAB. As input `data_gen.m` accepts an integer, $INIT\_L$, representing the desired length of the random bit sequence that the function is to return as output. The variable name tx_data is used to return the random data output. Note that the data are generated using the MATLAB function, `rand`, which produces up to 2^{1492} random numbers before repeating itself. As described in section 2.1.2 a single GSM data block uses 260 random bits. The maximum number of blocks that may be simulated before `rand` starts to repeat itself can be found to

$$Blocks_{max} = \frac{2^{1492}}{260} \approx 500 \cdot 10^{444} \tag{2.1}$$

This number of burst is more than enough to secure prober statistics for the simulations.

2.1.2 Channel Encoding

The purpose of the channel encoder is to provide the GSM receiver with the ability to detect transmission errors and eventually correct some of these. This is to improve the transmission quality from a bit error point of view. Various encoding standards are used in GSM depending on the mode of transmission. The encoding implemented here goes for the burst type TCH/FS (*Traffic CHannel Full rate Speech*) which is a normal speech burst.

The channel encoding is here implemented by the function `channel_enc.m`. As its input, tx_block, the `channel_enc.m` accepts a 260 bit long vector. The content of tx_block is encoded to produce a 456 bit long vector which then is returned as output using the variable name tx_enc.

More specificly `channel_enc.m` splits the incoming 260 information bits into three different classes, i.e. class Ia, class Ib, class II, depending on the importance of the bits. For instance, any transmission errors in the class Ia bits effect the overall speech quality more severely than errors in class II bits. Due to this variation in bit importance the different classes of bits are encoded accordingly. The channel encoding scheme utilized in GSM is illustrated in Figure 2.2.

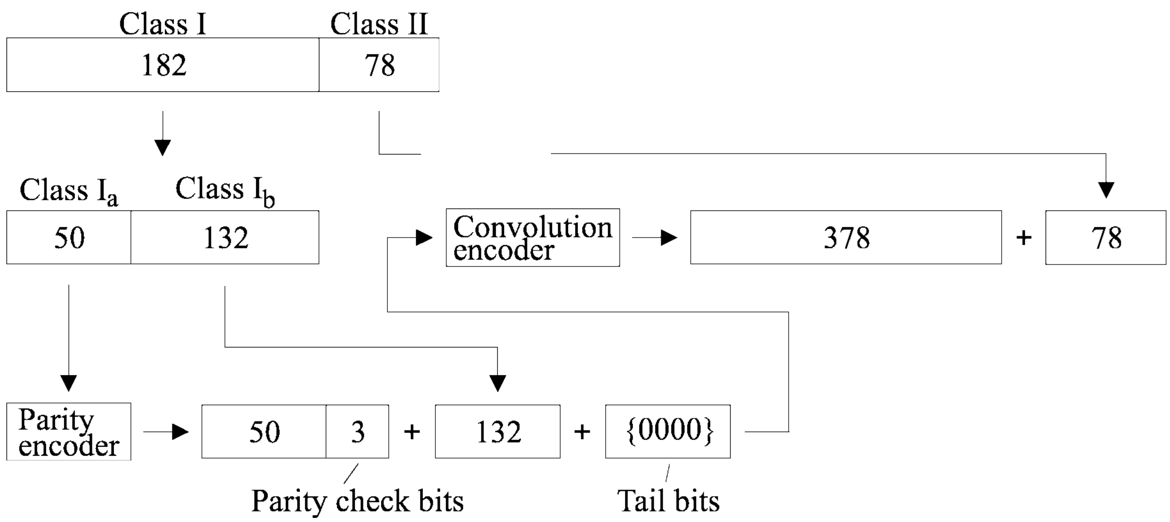


Figure 2.2: Channel encoding in GSM. A total of 196 redundant bits are added.

The channel encoding scheme is thus as follows. The 50 most significant class I bits, the class Ia bits, are extracted from the sequence and parity encoded. The parity encoder used in GSM is a systematic cyclic encoder based on three check bits. Systematic means that the parity bits are added to the original class Ia bit sequence. This way the class Ia bits are left unchanged and the resulting code word has the structure

$$\overline{V} = \{[k \text{ data bits}][r \text{ check bits}]\} \tag{2.2}$$

The generator polynomial used in the encoder has a length of 4 bits and is given as [9]

$$G(x) = x^3 + x + 1 \quad \rightarrow \quad \overline{G} = \{1011\} \tag{2.3}$$

The check bits are found as the remainder, $r(x)$, of the division

$$\frac{x^r \cdot D(x)}{G(x)} = Q(x) + \frac{r(x)}{G(x)}, \tag{2.4}$$

where the number of check bits is given by r, $D(x)$ represents the data bits intented for encoding and $Q(x)$ the division quotient. The remainder, $r(x)$, is then directly used to form the check bit sequence required in generating $\overline{V}$.

The multiplication $x^r \cdot D(x)$ is equivalent to shifting $D(x)$ r places to the left. Also, the implementation makes use of a default function, `deconv.m`, provided by MATLAB to perform the division.

After parity encoding of the class Ia bits these are recombined with the class Ib bits and a tail sequence of four zeros is finally added. The resulting class I sequence, now consisting of 189 bits, is then feed to the convolution encoder.

The convolution encoder takes a block of k bits as input and returns a block of n bits as output. The rate of the encoder, defined as the ratio k/n, is in the GSM system specified to be $1/2$. In the convolution encoding scheme each output bit, c_n, is depending not only on the input bit presently being encoded, b_k, but also on some of the previous input bits. The number of input bits required in the processing of encoded output bit is called the constraint length of the encoder. GSM specifies a constraint length of 5 in its encoding scheme defined as

$$\begin{aligned} c_{2k} &= b_k \oplus b_{k-3} \oplus b_{k-4} \\ c_{2k+1} &= b_k \oplus b_{k-1} \oplus b_{k-3} \oplus b_{k-4}, \end{aligned}$$

where $\oplus$ implies modulo 2 addition, and

$$k \in \{0, 1, 2, \ldots, 189\} \text{ and } b_k = 0 \text{ for } -\infty \leq k < 0 \tag{2.5}$$

As the convolution encoder is defined as a rate $1/2$ encoder two output bits are generated for every input bit, hence the two expressions. When operated as a shift register the convolution encoder takes on the form illustrated in Figure 2.3.

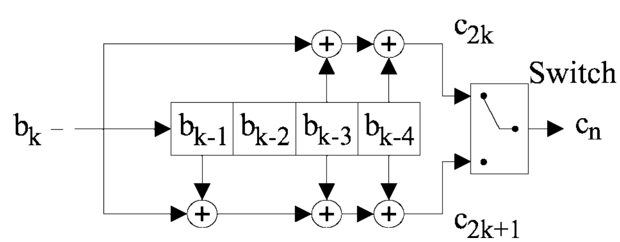


Figure 2.3: The convolution encoder scheme used in GSM for encoding of TCH/FS bursts. All additions are modulo 2 additions.

The combined channel encoder implementation is found in Enclosure A.

The two encoding schemes in the channel encoder are tested separately. Both the parity and the convolution encoder operates as expected. As an example, a typical input/output scenario for the combined encoder is illustrated in Figure 2.4. Notice that the output rate is twice the input rate.

Figure 2.4 illustrates the encoding of the 25 first bits of input data and, as such, the effects of parity encoding is not displayed.

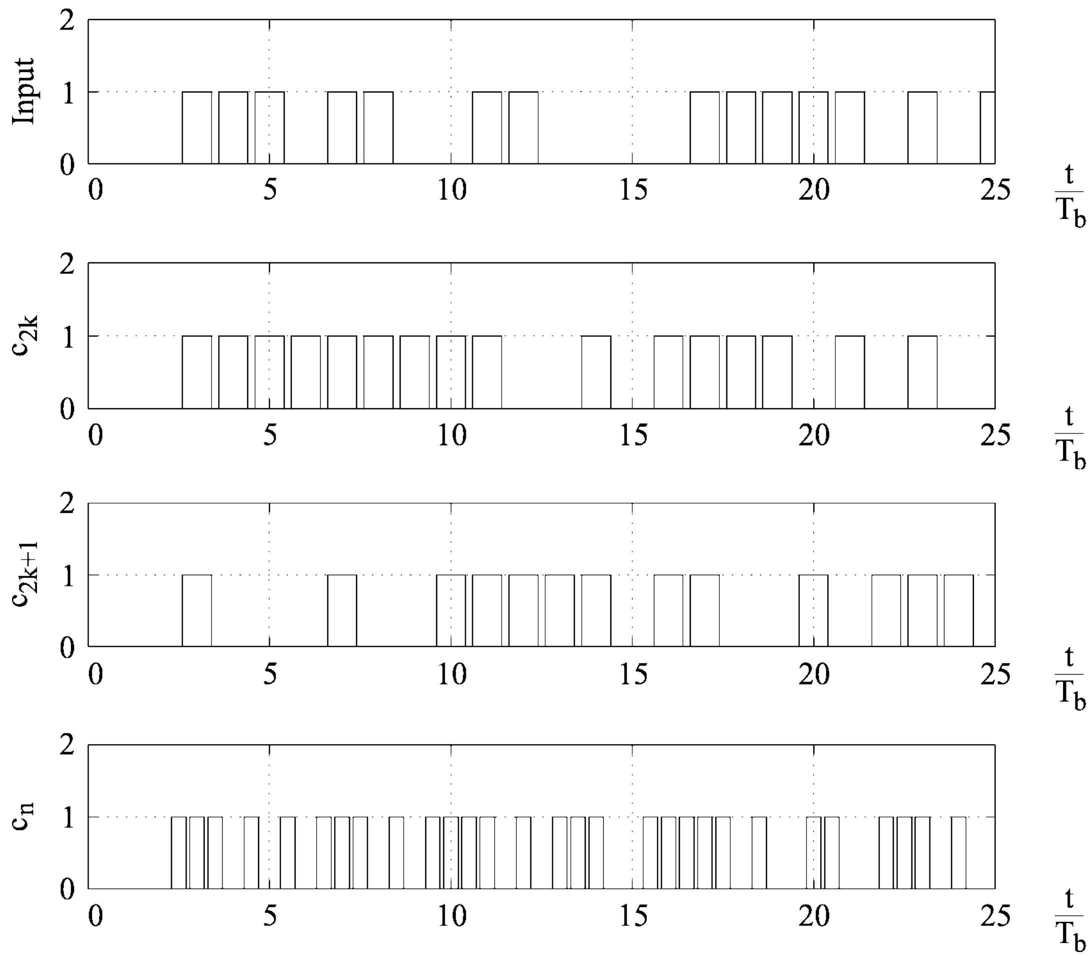


Figure 2.4: Typical input/output scenario of the combined channel encoder. The seed value used in `data_g.m` is 931316785.

2.1.3 Interleaving

The interleaver shuffles the bits contained in the data blocks output from the channel encoder, and distributes them over a number of bursts. The input variable is thus tx_enc, and the output is delivered to a number of instances of the variable tx_data. The purpose of this procedure is to ensure that the errors that appear in a received data block are uncorrelated. The motivation for reducing the correlation between bit errors is that the convolution code used to protect the class I bits has better performance when errors are not correlated [16]. Correlation between bit errors can occur in for example fading conditions.

The interleaver operates according to the following two formulas [15]

$$\begin{aligned} b &= ((57 \cdot (T \bmod 4) + t \cdot 32 + 196 \cdot (t \bmod 2)) \bmod 456) && (2.6) \\ B &= ((T - (b \bmod 8)) \operatorname{div} 4), && (2.7) \end{aligned}$$

which imply that bit number t in tx_data intended for burst number T is found in instance B of tx_enc as bit number b. In the above $(x \bmod y)$ is the remainder of the division x/y, and $(x \operatorname{div} y)$ is the corresponding quotient. The operation of (2.6) and (2.7) are illustrated by Figure 2.5.

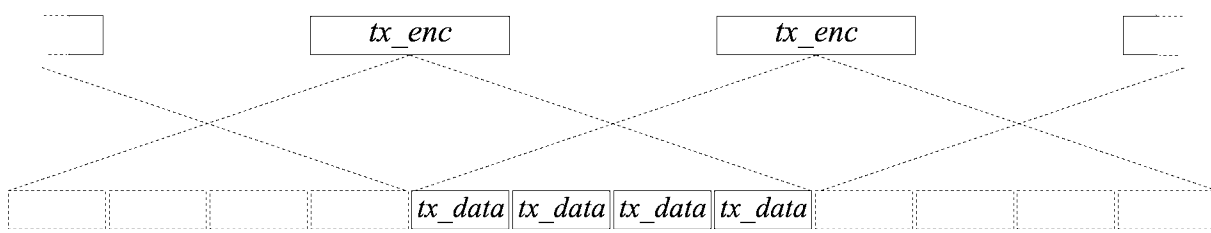


Figure 2.5: Illustration of the interleaving prosess as described by (2.6) and (2.7).

It can be realized by writing (2.6) and (2.7) out for a significant number of bursts, that the interleaver can be implemented so that it operates at two code blocks at a time. For each interleaving pass four sets of tx_data are returned. These data are further processed in the multiplexer, which is described in the next section. Since two instances of tx_enc contain two times 456 bit, and four set of tx_data contain 456 bit, it is evident that all the bits contained in the input to the interleaver are not represented in the output. This is solved by passing each code block to the interleaver two times. In practice this is done by implementing a queue of code blocks, as illustrated in Figure 2.6.

The interleaver is implemented in the MATLAB function `interleave.m`, and the four sets of tx_data, are returned in a matrix for convenience. For speed optimization the interleaving positions are precalculated in the implementation. This precalculation is implemented in the functions `make_interleave_m.m` and `make_deinterleave_m.m`

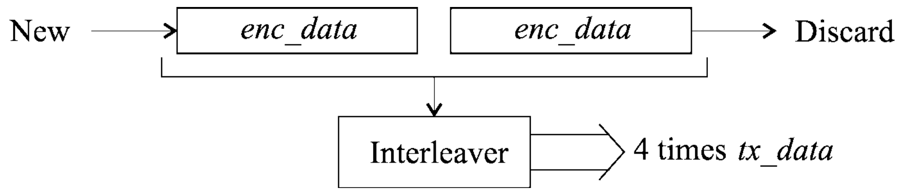


Figure 2.6: Operation of the interleaver, aided by a queue. The interleaver reads out the entire content of the queue. The queue has two slots, and in each interleaving pass a new block is pushed into the queue, and the eldest block is discarded.

2.1.4 Multiplexing

The input to the Multiplexer is tx_data, and the output is given in tx_burst. What the multiplexer does, is to take tx_data from the interleaver, and place it appropriately in a frame structure. The GSM-recommendations dictate specific burst structures for different transmission purposes. The implemented burst, referred to as a GSM normal burst, has the structure displayed in Figure 2.7.

3	57	1	26	1	57	3
Tail bits	Data bits Block 1	CTRL bit	Training sequence	CTRL bit	Data bits Block 2	Tail bits
{000}		{1/0}		{1/0}		{000}

Figure 2.7: The prescribed GSM normal burst structure. The number of bits in each part of the burst is indicated by the integers above the individual parts.

From Figure 2.7 is is clear that the GSM normal burst is made up of $2 \cdot (3 + 57 + 1) + 26 = 148$ bits in total. Of these 148 bits, $2 \cdot 57 = 114$ are pure data bits. Hence, `burst_g.m` must accept a total of 114 bits as input when a normal burst is considered. Of the 114 data bits input to the multiplexer only $114 * (260/456) = 65$ bits are true information bits, as can be seen from section 2.1.3. 65 information bits out of a total of 148 transmitted bits corresponds to a transmission efficiency of approximately 44%.

The bit patterns included below the burst sequence in the figure indicate that these parts have predefined patterns that must be complied with. For instance, the tail bits must constitute of three zeros, $\{000\}$, while the control bits can be selected at random as these are left unused in the normal burst. The training bit sequence can be any of eight prescribed ones. Thus, to form this burst structure the multiplexing involves tx_data as well as a training sequence, $TRAINING$. These are then ordered in the correct manner and supplemented by tail and

CTRL bits to form the correct output format returned using the variable name tx_burst.

2.2 GMSK-Modulation

The implemented GMSK-modulator is made up of three functions, namely `diff_enc.m`, `gmsk_mod.m`, and `ph_g.m`, as illustrated in Figure 2.8.

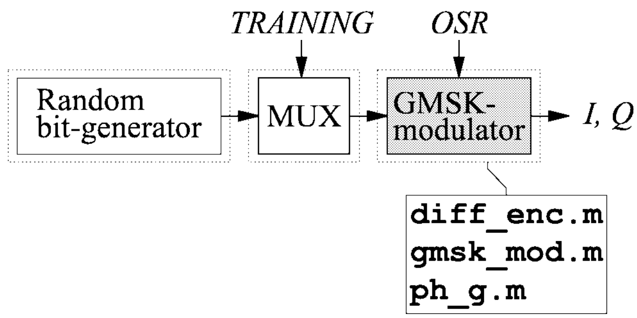


Figure 2.8: The functions `diff_enc.m`, `gmsk_mod.m`, and `ph_g.m` are all related to the GMSK-modulator implementation.

These three functions implement two separate tasks, as both a differential encoding of the burst sequence as well as the actual GMSK-modulation is performed. These two operations are described further in the following sections.

2.2.1 Differential Encoding

The output from the MUX, $burst$, is a binary {0, 1} bit sequence. This sequence is is first mapped from the RTZ (*Return To Zero*) signal representation to a NRZ representation before being input to the GMSK-modulator. This task is accomplished by the function `diff_enc.m`.

GSM makes use of the following combined differential encoding and level shifting scheme, where $d \in \{0, 1\}$ and $a \in \{-1, 1\}$ represent input and output sequences, respectively[10]

$$\begin{aligned} \hat{d}[n] &= d[n] \oplus d[n-1] \\ a[n] &= 1 - 2 \cdot \hat{d}[n], \end{aligned} \tag{2.8}$$

To avoid the start condition problem the GSM-recommendation [10] prescribes that an infinite length sequence of all ones are assumed to precede the burst to be processed. Hence, when calculating $a[0]$, and thereby also $\hat{d}[0]$, it may be assumed that $d[-1]$ is one.

The above encoding scheme is directly implemented in `diff_enc.m` where the variables $burst$ and $diff\_enc\_data$ are used to represent the input and output sequences, respectively. That is to say, that $burst$ equals d and $diff\_enc\_data$ equals a when comparing (2.8) to the actual implementation.

2.2.2 Modulation

After the differential encoding of the information burst the signal is GMSK-modulated. This is implemented by the function `gmsk_mod.m` where a complex baseband representation of the modulated signal is obtained.

GMSK is a modulation form derived from the very similar MSK (*Minimum Shift Keying*). Both are variants of the more general CPFSK (*Continuous Phase Frequency Shift Keying*) modulation forms.

Mathematically the generation of a MSK-signal may be described as

$$s(t, \overline{a}) = \sqrt{\frac{2E_c}{T_b}} \cos\left\{2\pi f_c t + \Theta(t, \overline{a})\right\}, \tag{2.9}$$

where E_c is the bit energy, f_c the carrier frequency, and $\Theta(t, \overline{a})$ the information carrying phase of the MSK signal. Through the use of a complex baseband notation f_c may be removed from the expression whereby Sine and Cosine values of $\Theta(t, \overline{a})$ is sufficient in describing the signal. This may be seen from the general complex baseband definition

$$\begin{aligned} s(t, \overline{a}) &= A \cdot \cos\left\{2\pi f_c t + \Theta(t, \overline{a})\right\} \\ &= A\left[s_c(t, \overline{a}) \cos\left\{2\pi f_c t\right\} - s_s(t, \overline{a}) \sin\left\{2\pi f_c t\right\}\right], \end{aligned} \tag{2.10}$$

where A describes the carrier amplitude and $\Theta(t, \overline{a})$ the phase modulation of the carrier. Also, in obtaining (2.10) the following definition is introduced

$$\begin{aligned} \tilde{s}(t, \overline{a}) &= s_c(t, \overline{a}) + j \cdot s_s(t, \overline{a}) = e^{j\Theta(t, \overline{a})} \\ &= \cos\left\{\Theta(t, \overline{a})\right\} + j \cdot \sin\left\{\Theta(t, \overline{a})\right\}, \end{aligned} \tag{2.11}$$

which represents the complex envelope of the modulated signal. From (2.11) it is clear that by taking the Cosine and the Sine of $\Theta(t, \overline{a})$ two low-pass baseband signals results, the in-phase, I, and the quadrature-phase, Q, signals, respectively. These two low-pass signals fully describe the original signal as described by (2.9).

Making use of the following pulse shaping function, $p(t)$, definition

$$p(t) = \begin{cases} \cos\left(\dfrac{\pi t}{2T_b}\right) & -T_b \leq t \leq T_b \\ 0 & \text{otherwise,} \end{cases} \tag{2.12}$$

the in-phase and quadrature phase components, $s_c(t,\overline{a})$ and $s_s(t,\overline{a})$, may be rewritten using the following linear form [8]

$$\begin{aligned} s_c(t,\overline{a}) &= p(t) * a_c(t) = \sum_{n\ even} a_c[n] \cdot p(t - nT_b) && (2.13)\\ s_s(t,\overline{a}) &= p(t) * a_s(t) = \sum_{n\ odd} a_s[n] \cdot p(t - nT_b), && (2.14)\end{aligned}$$

where the weighted impulse responses, $a_c(t)$ and $a_s(t)$, are given as

$$\begin{aligned} a_c(t) &= \sum_{n\ even} a_c[n]\delta(t - nT_b);\ a_c[n] = \cos(\Theta[n]) && (2.15)\\ a_s(t) &= \sum_{n\ odd} a_s[n]\delta(t - nT_b);\ a_s[n] = \sin(\Theta[n]), && (2.16)\end{aligned}$$

where δ is the Dirac pulse function and $\Theta[n]$ is given as

$$\Theta[n] = \frac{\pi}{2}\sum_{k=0}^{n-1} a[k] \tag{2.17}$$

To further simplify the MSK-baseband description a complex sequence, I, is introduced where the complex data symbols are given as

$$I[n] \equiv e^{j\Theta[n]} = \cos(\Theta[n]) + j \cdot \sin(\Theta[n]) = a_c[n] + j \cdot a_s[n] \tag{2.18}$$

Based on this complex definition the MSK-baseband representation may be described as one single convolution [3]

$$\tilde{s}(t,\overline{I}) = p(t) * a(t) = \sum_n I[n] \cdot p(t - nT_b), \tag{2.19}$$

where

$$a(t) = \sum_n I[n] \cdot \delta(t - nT_b) \tag{2.20}$$

Returning to the definition in (2.18) it is found that $I[n]$ alternating assumes real and imaginary values. This is a direct result of $\Theta[n] \in \{0, \pi/2, \pi, 3\pi/2\}$. This leads to the following recursive MSK-mapping definition [3]

$$I[n] = j \cdot I[n-1] \cdot a[n-1], \tag{2.21}$$

where

$$\begin{aligned} I[n] &\in \{1, -1, j, -j\}\\ a[n] &\in \{1, -1\}\end{aligned}$$

Further, in the GSM-recommendations [10] a differential encoding scheme is prescribed. Incorporating this encoding with the MSK-description in (2.19) an OQAM (*Offset Quadrature Amplitude Modulation*) MSK-model, including the GSM differential encoding, is obtained. This model is illustrated in Figure 2.9.

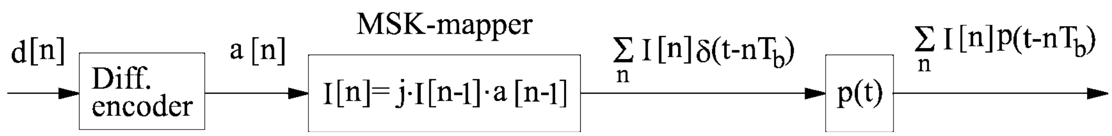


Figure 2.9: Final OQAM-model for MSK including the differential encoding prescribed in GSM.

This simplified MSK-representation comes in handy when trying to understand the structure of the data detector presented in Section 3.2.

In GMSK the phase shift are made smoother than in for instance MSK. This results in a more narrow frequency spectrum. The price paid for the desirable bandwidth reduction is ISI (*Inter Symbol Interference*) which results in an increased BER. A GMSK-signal can be generated using different approaches, e.g. the approach illustrated in Figure 2.9 with an appropriate choice of $p(t)$. The implementation used here is some what different as illustrated in Figure 2.10.

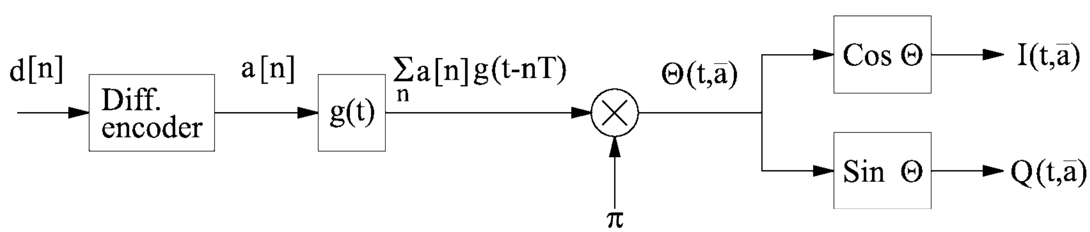


Figure 2.10: GMSK-baseband modulator implementation.

From Figure 2.10 it is seen that the symbol sequence, α, is convolved with $g(t)$, which is a frequency pulse function, and then multiplied with π, resulting in the generation of the phase function $\Theta(t, \alpha)$.

The phase function, $\Theta(t, \overline{a})$, may be written as [10]

$$\Theta(t, \overline{a}) = \sum_i a[i]\pi h \int_{-\infty}^{t-i\tau} g(\tau) d\tau, \tag{2.22}$$

where h the modulation index which for GSM equals 1/2. The frequency pulse function, $g(t)$, is mathematically defined as a convolution in time of a rectangular pulse, $v(t)$, and a Gaussian function, $h_g(t)$

$$g'(t) = v(t) * h_g(t), \tag{2.23}$$

where [10]

$$v(t) = \begin{cases} 1/(2T_b) & \text{for } 0 \leq |t| \leq T_b/2 \\ 0 & \text{otherwise} \end{cases} \tag{2.24}$$

$$h_g(t) = \frac{1}{\sqrt{2\pi}\sigma T_b} \exp\left[\frac{-t^2}{2\sigma^2 T_b^2}\right] \text{ where } \sigma = \frac{\sqrt{\ln\ 2}}{2\pi B T_b} \tag{2.25}$$

The 3 dB bandwidth, B, of the Gaussian function is specified by the normalized bandwidth, BT_b, which is specified to 0.3 for GSM [10]. The ideal Gaussian function has an infinite time duration, $t \in [-\infty, \infty]$. For reasons of signal processing this signal is truncated to a specific length, L where OSR and L then determine the number of samples used to represent the bell shaped Gaussian pulse. Typically, a value higher than 3 is chosen for L [12]. To make the frequency pulse function causal it is time shifted by an amount of $LT_b/2$. This results in the following truncated frequency pulse

$$g(t) = g'\left(t - \frac{LT_b}{2}\right) \cdot w_L(t) \text{ where } w_L(t) = \begin{cases} 1 \text{ for } 0 \leq t \leq LT_b \\ 0 \text{ otherwise} \end{cases} \tag{2.26}$$

To provide this information the function `ph_g.m` is used. Based on two input parameters, BT_b, and OSR the function calculates the required values of the frequency and phase shaping functions $g(t)$ and $q(t)$, respectively. These values are returned using the output parameters g_fun and q_fun for the frequency and phase functions, respectively.

The different stages in generating the truncated frequency pulse function, $g(t)$, and eventual the phase smoothing response function, $q(t)$, are shown in Figure 2.11. The first two plots, Figures 2.11a and 2.11b, illustrate functions that are made use of internally to the function `ph_g.m` while Figures 2.11c and 2.11d illustrate the output functions g_fun and q_fun, respectively.

Please note that the function is implemented assuming a truncation length, L, of 3. Hence, `ph_g.m` returns $3 \cdot OSR$ samples of $g(t)$ and $q(t)$ within the interval 1 to 4 t/T_b.

Having generated the required shaping information the function `gmsk_mod.m` performs the actual calculation of the phase value $\Theta(t, \overline{a})$. This is done by a sliding window approach where the $g(t)$ function is slid across the input sequence while accumulating the previous phase information.

The structure of the implemented GMSK-modulator is, however, based on `gmsk_mod.m` calling the `ph_g.m` function and as a result a few extra input parameters are required. To perform correctly `gmsk_mod.m` requires four input parameters need to be specified. These are the differential encoded information sequence, $burst$, the bit duration, T_b, the normalized bandwidth, BT_b, and the simulation oversample ratio, OSR. Of these four input parameters the two, BT_b, and OSR, are passed on to `ph_g.m`.

The resulting phase function is evaluated through Sine and Cosine to obtain the in-phase, I, and quadrature phase, Q, values returned by the function. The variables i and q are used to return the in-phase and the quadrature signals, respectively.

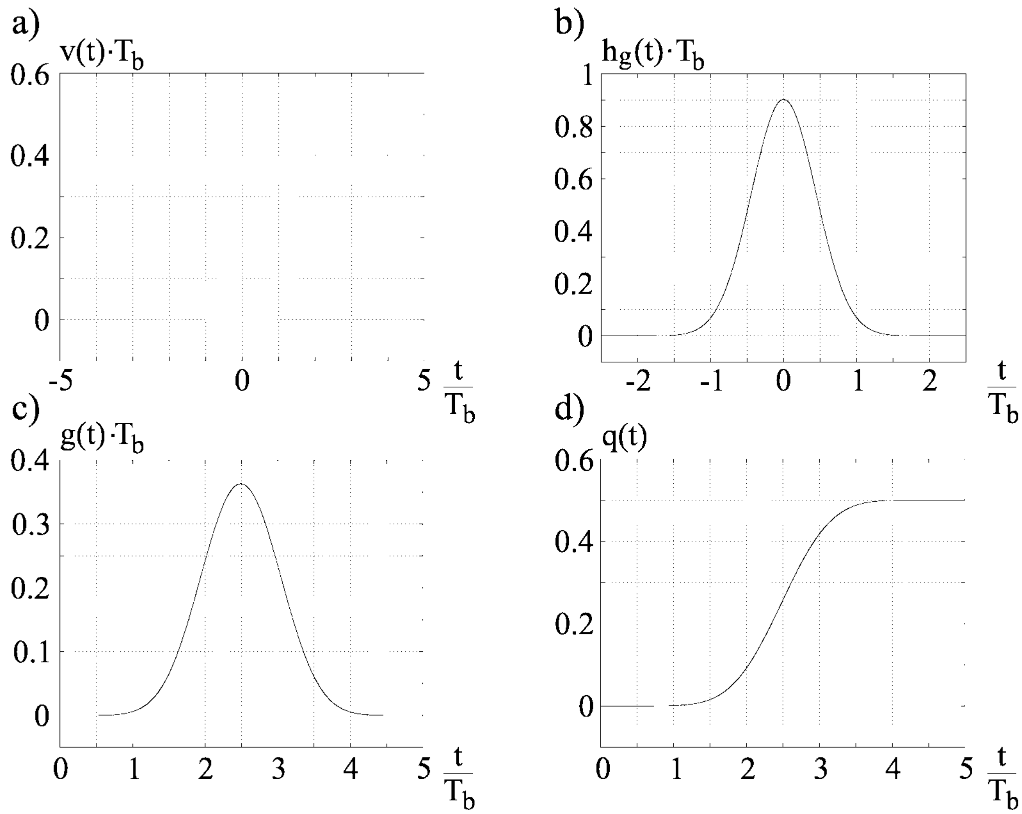


Figure 2.11: Step-by-step illustration of the generation of the phase smoothing function. a) The rectangular pulse, b) the Gaussian bell shape. c) The resulting frequency pulse function, and d) the equivalent phase shaping function.

2.3 Transmitter Test

To test the operation of the implemented transmitter two time-domain tests and a single frequency domain test are carried out.

First the result of a time domain test of the relationship between the I and Q signals is illustrated in Figure 2.12.

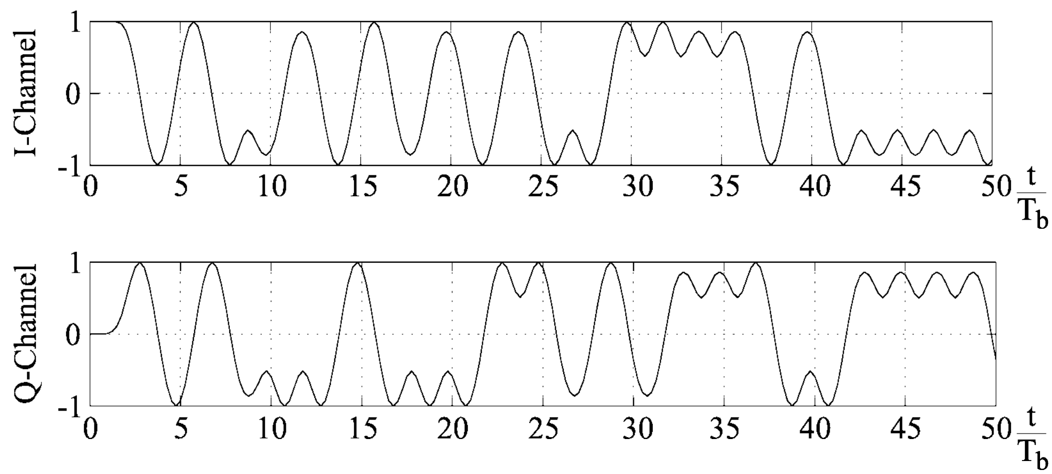


Figure 2.12: I and Q baseband outputs from the implemented modulator when given a random input sequence.

From this the I and Q signals are seen to display the expected behavior. When compared with other visualizations found in relevant literature the in-phase and quadrature phase signals are found to resemble these. Furthermore, as the I and Q signals are given as $\cos(\Theta)$ and $\sin(\Theta)$, respectively, the following relation must be fulfilled at all times

$$I_n^2 + Q_n^2 = \cos^2(\Theta) + \sin^2(\Theta) = 1 \tag{2.27}$$

This relation has been tested in MATLAB and the result shows that the I and Q signals are correctly related as a result of 1 is obtained for every sample value tested.

A second time domain test is performed by feeding the modulator a sequence consisting of all ones. This is equivalent to transmitting a GSM frequency correction burst. As every one of the transmitted ones eventually adds $\pi/2$ to the phase of the signal four bits are required before the signal returns to its initial phase state. The rate of the input sequence then determines the speed of this phase rotation. Hence, when delivered such a sequence the modulator should return a sinusoidal signal of frequency $r_b/4$ for both I and Q channels. Due to the Sine/Cosine relation the Q channel should trail the I channel by an amount of one T_b. This is illustrated in Figure 2.13.

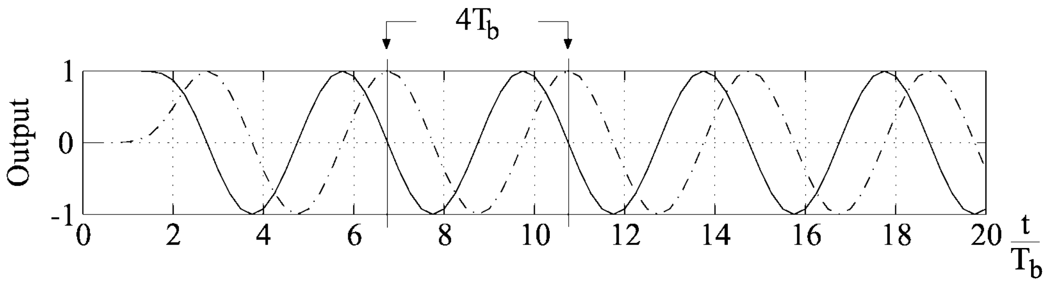


Figure 2.13: I and Q (dashed) signals when the modulator is given a sequence of all ones.

Figure 2.13 illustrates the I and Q channels over a time period of 20 T_b's. From this the modulation is seen to operate as expected as the signals display periods having a time durations of 4 T_b's, which of course equals $r_b/4$. Thus the time domain test indicate that the performance of the system is acceptable.

Also, a frequency test is performed to analyze the spectral properties of the baseband signal, as produced by the implemented modulator. The resulting spectrum is compared to the GSM requirements and to other reported spectrums [14].

As the resulting power spectrum, shown in Figure 2.14, reveal some filtering need to be implemented in order to fully comply with the GSM 05.05 requirements [11]. This tx-filtering is not implemented in *GSMsim*.

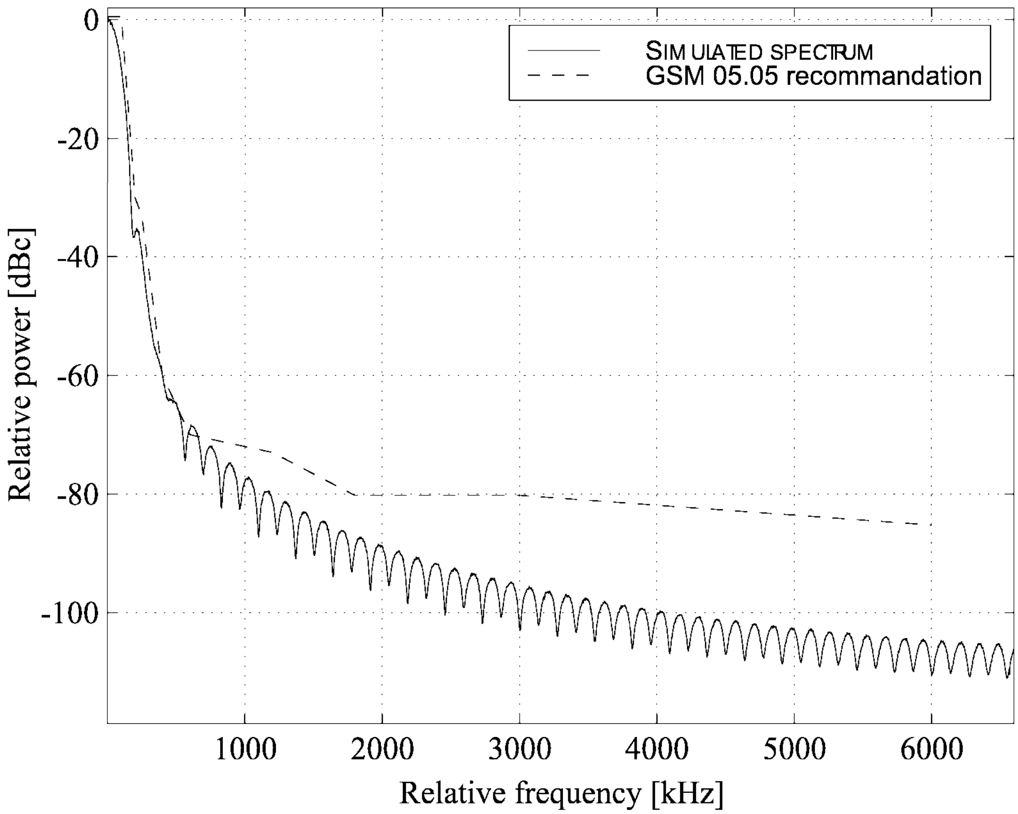


Figure 2.14: The power spectrum generated by the modulator. The spectrum is generated by averaging over 10000 spectras produced by GMSK modulated sequences each 1024 bits long. In the simulation a sample rate of $f_s = 64 \cdot r_b$ is used. The dashed line represents the GSM 05.05 requirement [11].

3

Receiver Background

The receiver implementation used in the *GSMsim* toolbox is shown in Figure 3.1. In the diagram presented in the Figure 3.1 the demodulator block part of the data receiver is expanded compared to the diagram illustrated in Figure 1.4. Hence, the demodulator part of Figure 1.4 is expanded into three separate blocks.

In this chapter the theory underlying the function of the implementation is given a short introduction.

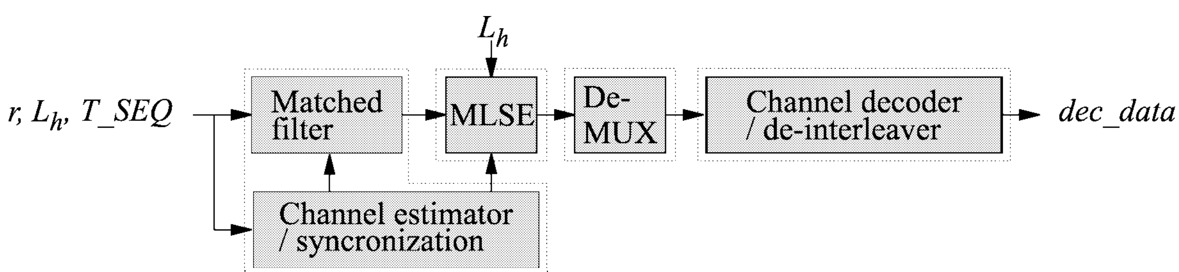


Figure 3.1: Block diagram of the receiver implementation used in the *GSMsim* toolbox.

As described in Chapter 1, the receiver implementation does not include a front-end, since the original intention with the toolbox is to provide for easy simulation of user defined front-ends. This has the effect that no channel selection, or filtering, is done, since the implementation of such vary with the front-end implementation.

The contents of this chapter is divided into four sections. The first section describes, synchronization, channel estimation, and matched filtering. The second section introduces the theory underlying the MLSE (*Minimum Least Square Error*) implementation. The third section contains a short introduction of the de-multiplexer, de-interleaver, and channel decoder. None of the three sections provide an in depth treatment of the subjects, but rather provide for a summary of the used techniques. The last section contains a description of tests performed on the receiver implementations.

3.1 Synchronization, Channel Estimation, and Matched Filtering

Synchronization, channel estimation, and matched filtering is done in a two step process. This is illustrated in Figure 3.2. For the matched filter to operate correctly the synchronization and channel estimation must be done first.

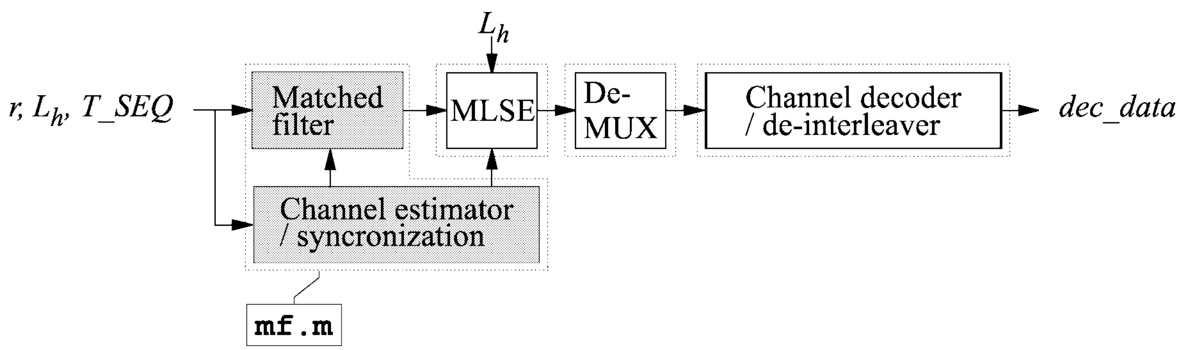


Figure 3.2: Illustration of how the synchronization, channel estimation, and matched filtering is divided into two parts.

As can be seen from Figure 3.2 both the channel estimator and the matched filter have the sampled received signal, r, as input. r is a sampled sequence which is expected to contain the received GSM burst. Also, the oversampling factor, OSR, described as f_s/r_b, with f_s being the sample frequency, and r_b the symbol rate, is input to both of these two blocks. Finally, these two blocks have L_h as input, where L_h is the desired length of the channel impulse response measured in bit time durations. The channel estimator passes an estimate of the channel impulse response, h, to the matched filter. Also, the channel estimator passes the sample number corresponding to the estimated beginning of the burst in r.

To interface correctly with the MLSE implementation `mf.m` must return a down-sampled – one sample per symbol – version of the now matched filtered burst. Also, the MLSE requires information concerning the matched filter. This information is supplied by also returning the impulse response autocorrelation, i.e. R_{hh}.

To understand the operation of `mf.m`, recall from earlier that a training sequence is inserted in each burst. The method used for obtaining synchronization is based on the mathematical properties of this training sequence.

The training sequence, $TRAINING$, used in *GSMsim* is as follows [15]

$$TRAINING = [0,0,1,0,0,1,0,1,1,1,0,0,0,0,1,0,0,0,1,0,0,1,0,1,1,1], \tag{3.1}$$

for which the following MSK-mapped equivalent, T_{SEQ}, is used

$$T_{SEQ} = \tag{3.2}$$
$$[1,j,1,-j,1,-j,-1,j,-1,-j,-1,-j,1,j,1,-j,1,j,1,-j,1,-j,-1,j,-1,-j].$$

This sequence is one of eight predefined training sequences when a normal burst is considered. Now, from T_{SEQ} the central sixteen MSK-symbols are picked and referred to as T_{SEQ_C}. If T_{SEQ_C} is extended by placing five zeros in both ends, a sequence, T_{SEQ_E}, is obtained. This is done in order to obtain equal length vectors that, when evaluated using the following MATLAB command

```
stem(abs(xcorr(T_SEQ_E,  T_SEQ))),
```

produces a result similar to that presented in Figure 3.3.

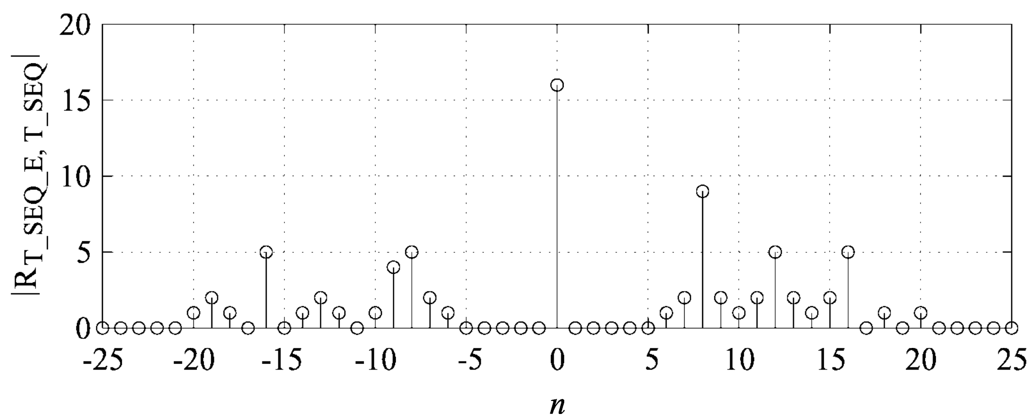


Figure 3.3: Cross correlation between T_{SEQ} and the extended version of T_{SEQ_C}. n represents the number of samples that the two sequences have been displaced in calculating the correlation value.

What Figure 3.3 illustrates is that

$$R_{T_{SEQ_C},T_{SEQ}} = \begin{cases} 16 & \text{for } n = 0 \\ 0 & \text{for } n \in \{\pm1, \pm2, \pm3, \pm4, \pm5\} \\ ? & \text{otherwise,} \end{cases} \tag{3.3}$$

where the question mark represents the undefined correlation noise that is found outside of the interval $n \leq \pm 5$.

The result presented in Figure 3.3 may be verified through manual calculations, using the following

$$R_{T_{SEQ_C},T_{SEQ}}[n] = T_{SEQ_C}[-]^* * T_{SEQ}, \tag{3.4}$$

where $*$ denotes convolution, and $T_{SEQ_C}[-]^*$ is $T^*_{SEQ_C}$ with its elements reversed. This property is useful since the received signal corresponding to the transmission of the training sequence, here called $r_{T_{SEQ}}$, may be written as

$$r_{T_{SEQ}} = T_{SEQ} * h + w, \tag{3.5}$$

where h is the channel impulse response, and w is unknown additive noise. If convoluting this with $T_{SEQ_C}[-]^*$ then the following is obtained

$$\begin{aligned} r_{T_{SEQ}} * T_{SEQ_C}[-]^* &= T_{SEQ} * T_{SEQ_C}[-]^* * h + w * T_{SEQ_C}[-]^* && (3.6)\\ &= \begin{cases} 16h + w * T_{SEQ_C}[-]^* & \text{for } n = 0 \\ w * T_{SEQ_C}[-]^* & \text{for } n \in \{\pm 1, \pm 2, \pm 3, \pm 4, \pm 5\} \end{cases} && (3.7)\\ &\approx \begin{cases} 16h & \text{for } n = 0 \\ 0 & \text{for } n \in \{\pm 1, \pm 2, \pm 3, \pm 4, \pm 5\} \end{cases} . && (3.8) \end{aligned}$$

The approximation leading from (3.7) to (3.8), is based on the assumption that the noise, w is white and the knowledge that T_{SEQ} has white noise like properties, as illustrated by Figure 3.3. It is indicated by the above, that if an entire burst containing T_{SEQ} is considered, then similar calculations can be done. Thus, if an entire burst is convoluted by $T_{SEQ_C}[-]^*$ it is seen that an estimate of the channel impulse response is present in the result, called v. Also, it is observed that the estimate of the impulse response that is contained in v is likely to be more powerful than the neighboring contents of v. This is due to the factor sixteen and the zero samples. This knowledge leads to the sliding window technique, which allows for both channel estimation and synchronization at the same time.

The sliding window technique uses the fact that in the GSM system coarse synchronization is present on the basis of dedicated synchronization bursts. This coarse synchronization is used for sampling a time interval of the received signal, in which the desired burst is likely to be found. This, possibly oversampled, sample sequence is referred to as r.

The first step in the sliding window technique is to convolute r with $T_{SEQ_C}[-]^*$, to obtain a signal v

$$v = r * T_{SEQ_C}[-]^* \tag{3.9}$$

Here, v is an intermediate result, and all samples in v are immediately squared to yield an energy estimate e

$$e[n] = v[n]^2 \tag{3.10}$$

Now the window energy, we, is found using

$$we[m] = \sum_{k=m}^{m+L} e[k], \tag{3.11}$$

for all but the last L samples in e, where $L = (L_h * OSR) - 1$. The sample m_{max} in we containing the highest energy value is estimated as corresponding directly to the first sample of the channel impulse response in v. From m_{max}, and the known oversampling ratio, it is now possible to extract an estimate of the channel impulse response, and also calculate the beginning of the burst.

Note from the above that the obtained channel impulse response estimate, h, cannot be any longer than five T_b's. This is due to the number of zero samples surrounding the peak in (3.3). In the present implementation the length of h measured in bit time durations has been limited, as is expressed by L_h

$$L_h \in \{2,\ 3,\ 4\}. \tag{3.12}$$

In this context, it is worth noting that the number of samples in h is given as $OSR \cdot (L_h + 1)$, and not L_h.

In the described procedure the entire r sequence is processed. In the actual implementation, however, only a sub-sequence is processed. This is possible since the location of the training sequence within a GSM burst is known. Refer to Section B.1 for details on this.

Having obtained sample synchronization, and an estimate of the channel impulse response, the matched filtering can be done as

$$Y = r * h^*[-]. \tag{3.13}$$

Along with the filtering of r down sampling is done as well. This is needed since r contains at least f_s/r_b as many samples as desired in Y. Recall here that Y must contain one sample per MSK symbol in the received burst. In this work a special technique is used so that the obtained synchronization is not lost during the matched filtering described by (3.13).

All the functions described in this chapter are implemented in a single MATLAB function, called `mf.m`. The actual implementation of `mf.m` is described in Section B.1.

3.2 Minimum Least Square Error (MLSE) Detection

The part of the receiver that handles the actual detection of the received sequence is the MLSE (*Minimum Least Square Error*) Detector. Here, the MLSE is implemented as a Viterbi equalizer based on the modified Ungerboek algorithm [3]. The placement of the MLSE in the receiver is shown in Figure 3.4.

As shown in Figure 3.4, the MLSE input is interfaced by two blocks internally in the receiver. These two blocks are the matched filter, and the channel estimator. The input to the MLSE is

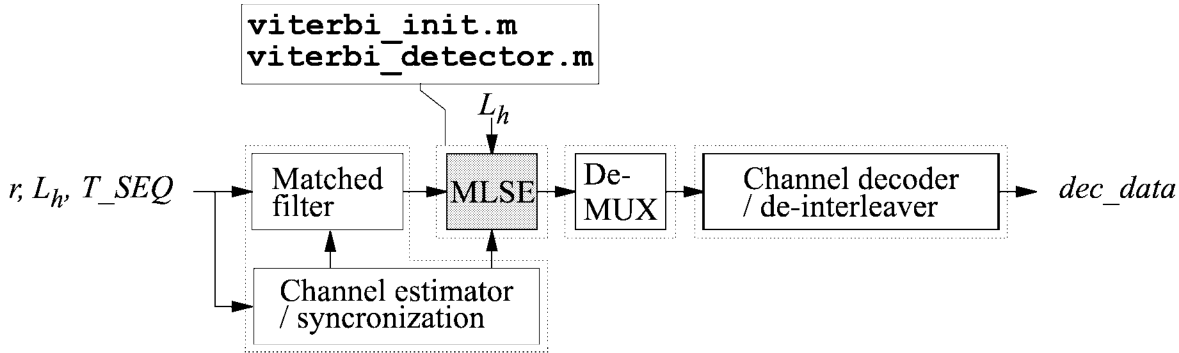


Figure 3.4: The placement of the MLSE in the overall receiver structure.

the matched filtered and down sampled signal, referred to as Y, along with R_{hh} which is the autocorrelation of the estimated channel impulse response. Y is a sequence of samples, and contains one sample for each transmitted symbol. The output of the MLSE, rx_burst, which is an estimate of the most probable sequence of transmitted binary symbols.

The MLSE, as it is implemented here, operates on basis of the system shown in Figure 3.5c. To understand the figure, recall the alternative OQAM transmitter model described earlier on page 13.

Figure 3.5a, included for comparison, represents the implemented system. The implemented modulator structure is merely one of a number of possible solutions, in fact, the structure shown in Figure 3.5b can be used with the same result. This is exploited in Figure 3.5c where the MLSE is shown as an Viterbi detector which assumes a system where a stream of MSK-symbols are transmitted trough an extended mobile channel. This extended channel is purely fictive and covers the full signal path from the output of the MSK-mapper to the input of the matched filter. The MSK-symbols may be obtained from the binary sequence to be transmitted, and vice versa. It is thus sufficient to find the transmitted sequence of MSK-symbols, and then map these symbols to binary information. Therefore, the Viterbi detector estimates the sequence of MSK-symbols input to the extended mobile channel.

In order for the implemented algorithm to work the system bounded by the label I and the matched filter output in Figure 3.5c is required to have a causal impulse response, h, of finite duration L_h. This requirement seems reasonable when considering the real life scenario. Furthermore, it is required that this impulse response does not change significantly during the reception of a GSM burst.

With these requirements the bounded system may be considered as a finite state machine with each state, to the discrete time n, only depending on the previous L_h MSK-symbols in I. That is, the MSK-symbols trigger the state shifts of the machine and thus, the next state is uniquely determined by the present MSK-symbol in I. The state of the machine to the time n is referred

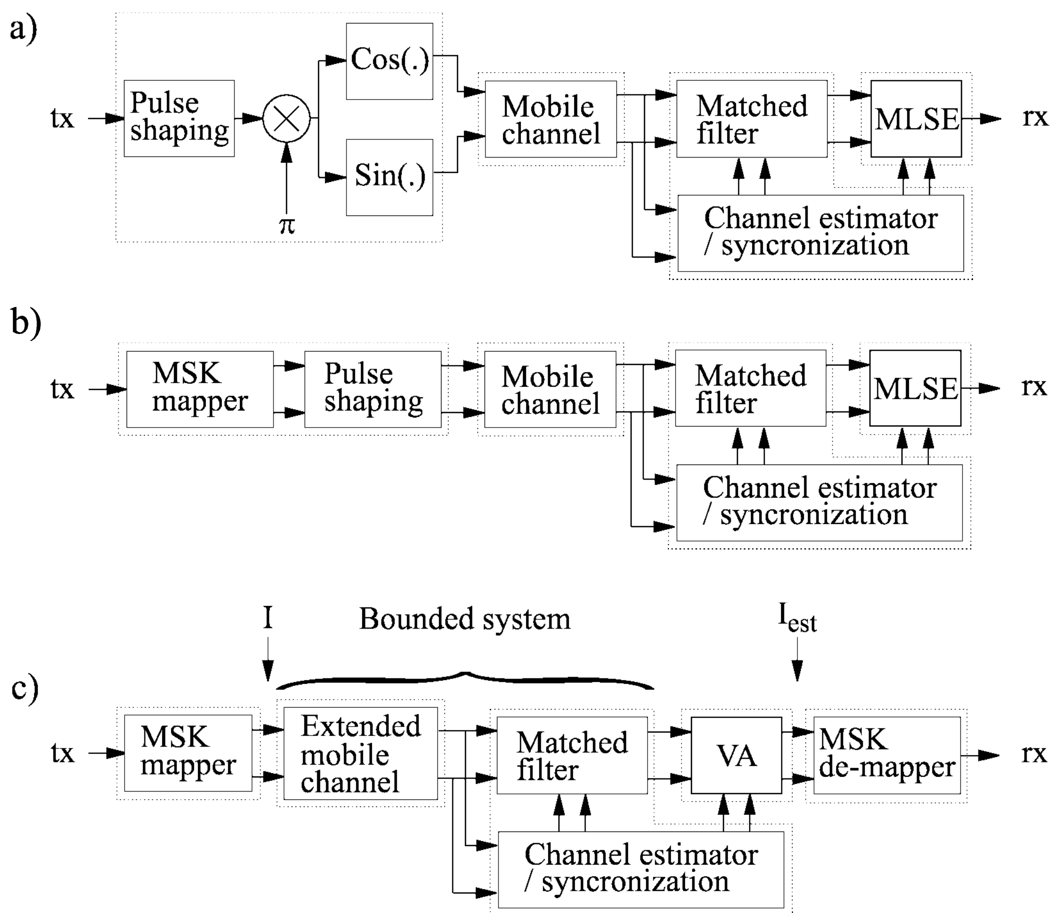


Figure 3.5: Various representations of a general baseband transmission system. a) The transmission system showing the implemented system. b) The system as it is described using the OQAM model. c) The system on which the MLSE is based.

to as $\sigma[n]$ and is represented as

$$\sigma[n] = [I[n], I[n-1], \ldots, I[n-(L_h-1)]], \tag{3.14}$$

in which the right hand side is the sequence of the last L_h MSK-symbols. In general states, for which $I[n]$ assumes one of the values $-j$ or j, are referred to as complex. Likewise states where $I[n]$ assumes one of the values -1 or 1, are referred to as real. This is to prove useful later in this section.

In order to find the number of legal states, recall from the description of the OQAM receiver on page 13, that four MSK-symbols exist, namely $1, -1, j$ and $-j$. Also, recall from the above, that a state is described by the last L_h symbols. Additionally, recall from the description of the OQAM receiver on page 13 that if the symbol $I[n]$ to the time n is real, then $I[n+1]$ is complex, and vice versa. From this it is evident, that the number of states, M, is given by

$$M = 2^{L_h+1}, \tag{3.15}$$

which is the number of possible states at any time. In the above, $\sigma[n]$ is thus contained in a set of states consisting of M states. If referring to the individual states as s_m, then this set can be expressed as

$$\sigma[n] \in \{s_1, s_2, \ldots, s_M\}. \tag{3.16}$$

The concept that $\sigma[n]$ belongs to a set of states, which may be numbered from 1 to M, is used directly in the implementation done in the present work. Internally in the program a state is uniquely identified by an integer, called the state number, and not by MSK-symbols. Referring to (3.15), and observing that L_h is limited to four – or less – it is seen that the number of states in the state machine is thirty two or less. In the implementation there is no consciously predefined mapping between the MSK-symbols and the state numbers. Alternatively to a predefined mapping, a mapping table is constructed at runtime. This mapping table can be referred to for retrieval of the MSK-symbols whenever they are needed. The lookup is done using the state number as an index. Using the state number representation a mutation of the present state to obtain the legal previous and next states requires a call to the integer to symbols mapping table, followed by the actual mutation. In order to avoid the undesirable overhead associated with this, a set of transition tables are constructed. These transition tables can be used to obtain the legal next states or previous states by using the present state number as an index. Apart from limiting the legal previous and next states relative to a single state, it is possible to reduce the number of legal states to any discrete time. This is due to the fact that it is possible to determine a unique start state of the algorithm [15]. Since the MSK-symbols are shifting between complex and real values, knowledge of the start state effectively limits the number of legal states at any time to $M/2$. To see this refer to the formal state representation given by (3.14). In consequence of this it can for example be stated that if the start state, to $n = 0$ is complex then the state to $n = 200$ is also complex.

Having established the state concept, the problem of finding the most probable sequence of MSK-symbols now changes to locating the most probable path trough a state trellis. The concept of a state trellis is illustrated in Figure 3.6 for $L_h = 1$. Note that actual state trellises have

$M = 2^{1+1} = 4$ different states and just as many transitions as there are samples in Y. This is a result of Y containing one sample per transmitted symbol.

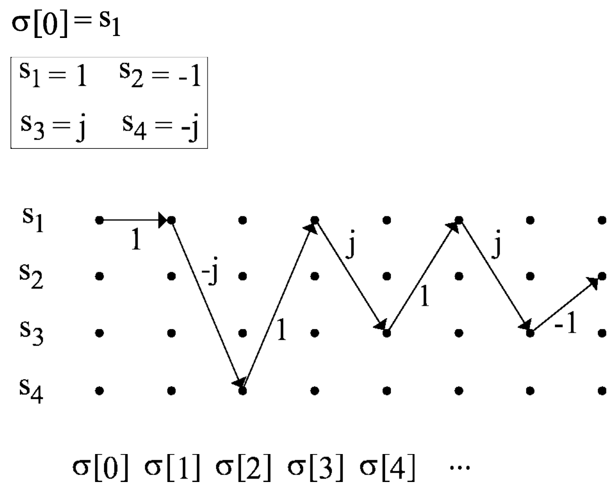


Figure 3.6: The transmission of a sequence of MSK-symbols using $L_h = 1$ and a transmitted sequence equal to $I = 1, -j, 1, j, 1, j, -1$. The described state machine assumes a new state to each discrete time n.

When operating with the trellis concept, it should be noted that all states have two legal next states. This can be realized by recalling that

$$I[n] \in \{1, -1\} \vee I[n] \in \{j, -j\}, \tag{3.17}$$

which in turn implies that all states have only two legal previous states.

Turning the attention to the method for finding the most probable path trough the trellis, the concept of an metric is introduced. To all discrete times n, all states m have an associated survivor metric. In the present implementation of the metric calculation, the rule is, that the higher the metric value the better. The term survivor stems from the fact that two paths lead to every state. Each path results in a metric for the state. The survivor metric is the highest valued of the two metrics. The metric of a path to a state is found by taking the metric of the previous state in the path, and then adding a contribution generated by the transition from the previous state to this state. The concept of survivor metrics is illustrated in Figure 3.7. The actual computation of the metric increment related to a state transition – referred to as a gain, $Gain$, in the metric – is is done on the basis of the following formula [3]

$$\begin{aligned} Gain(Y[n], s_a, s_b) &= 2\Re\{I^*[n]Y[n]\} \\ &\quad -2\Re\left\{I^*[n]\sum_{m=n-L_h}^{n-1} I[m]R_{hh}[n-m]\right\} - |I[n]|^2 R_{hh}[0], \end{aligned} \tag{3.18}$$

where s_a and s_b is previous and present state, respectively, described by their MSK-symbols. Y_n is the n'th sample in Y. Note from (3.18) that only the values of R_{hh} ranging from index 0 to L_h are used. For speed optimization (3.18) is reduced to

$$Gain(Y[n], s_a, s_b) = \Re\{I^*[n]Y[n]\} - \Re\left\{I^*[n] \sum_{m=n-L_h}^{n-1} I[m]R_{hh}[n-m]\right\}, \tag{3.19}$$

which results in the same decisions being made.

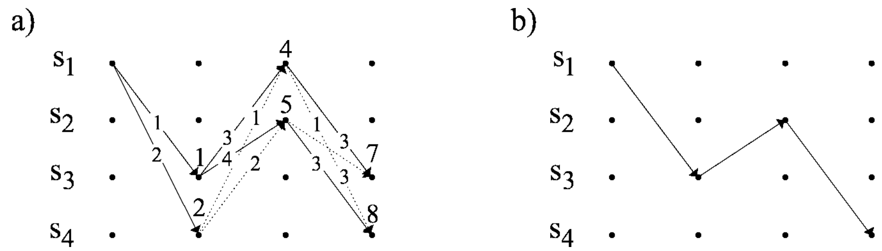


Figure 3.7: Illustration of survivor metrics. a) The survivor metric is found by taking the metric of the two legal previous states, and adding the contributions from the transitions. b) The survivor is the path with the highest valued metric.

The task of finding the most probable sequence, as illustrated in Figure 3.7b, may hence be formulated as follows.

To find the most probable path trough the trellis, start at the predetermined start state, and continue to the end of the state trellis. While traversing the trellis constantly compute survivor metrics for all legal states to all discrete times. Furthermore, record, for all of these states to each discrete time, the previous state that was chosen as contributor to the survivor metric for the individual state. Having processed the entire state trellis, the state with the highest metric, at the final discrete time, is chosen to be the last state in the most likely sequence of states. Having found the final state, lookup what state was the previous state, and continue in this manner until the entire sequence of states in the most likely path is established.

From the sequence of states the sequence of symbols is readily found from the first element of the MSK-symbols which make up each state. The MSK-symbols may readily be MSK de-mapped to obtain a NRZ representation. This de-mapped sequence then needs to be differential decoded and subsequently transformed into the binary RTZ representation. However, by using the following relation the MSK-symbols may be transformed directly into a differential decoded NRZ representation [15]

$$rx\_burst[n] = I_{est}[n]/(j \cdot rx\_burst[n-1] \cdot I_{est}[n-1]) \tag{3.20}$$

Hence, (3.20) implements both the MSK de-mapping as well as the differential decoding.

It is, in (3.20), necessary to identify a start value for $rx\_burst[0]$ and $I_{est}[0]$. From earlier work [15] these are both known to equal unity. In (3.20) the variable rx_burst is in NRZ format. The transformation to RTZ is done by adding unity to all elements and then divide by two.

The Viterbi detector is implemented in MATLAB. The implementation is split into two functions. The splitting is motivated by the fact that the data structures used by the algorithm do not depend on Y. Thus initialization of state translation tables, need not be done more than once for all the bursts in a simulation. Details about the implementation, including a pseudo code description of the algorithms, are given in Appendix B.2.

3.3 De-Multiplexing, De-Interleaving and Channel Decoding

The tasks of de-multiplexing, de-interleaving and decoding the data, are implemented in three separate blocks. The overall task of these three blocks is to regenerate the transmitted coded data blocks. This functionality is implemented via three MATLAB functions `channel_dec.m`, `deinterleave.m`, and `DeMUX.m`. These functions, and their relation to the block diagram, are shown in Figure 3.8.

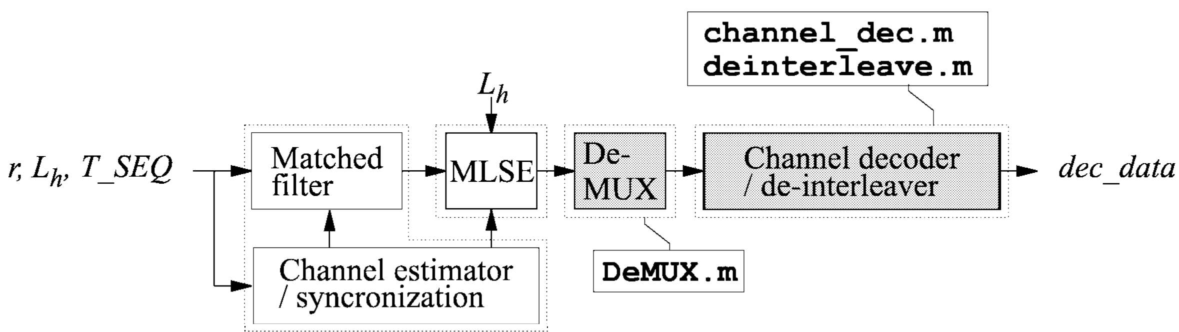


Figure 3.8: Illustration of the receiver implementation. The relations between blocks and actual implemented functions are indicated.

3.3.1 De-Multiplexing

The de-multiplexer is the first functional block to follow the Viterbi equalizer block in the implemented receiver. The placement of the de-multiplexer in the receiver structure is illustrated in Figure 3.8.

The input to the de-multiplexer is rx_burst, which is output from the MLSE, as described in the previous section. The output from the de-multiplexer is the contents of the two data fields in a standard GSM burst. These data are returned in a variable called rx_data. Refer to Section 2.1, and Figure 2.7 on page 11, for an description of a GSM burst as used in the *GSMsim* toolbox. The de-multiplexer is simple in its function, since all that needs to be done is to locate the data fields in rx_burst, and then copy these to rx_data. In this implementation the data fields are located by using the sample numbers, e.g. the first data field is found in $rx\_burst(4 : 60)$.

3.3.2 De-Interleaving

The de-interleaver reconstructs the received encoded data, rx_enc, from the received data, rx_data. The operation is the inverse of the interleaver, and may thus be considdered as an reordering of the shuffeled bits.

The de-interleaver operates according to the following two formulas [15]

$$R = 4 \cdot B + (b \bmod 8) \tag{3.21}$$
$$r = 2 \cdot ((49 \cdot b) \bmod 57) + ((b \bmod 8) \operatorname{div} 4), \tag{3.22}$$

which provide the information that bit number b for rx_enc instance number B, may be retrieved from rx_data corresponding to burst number R at position r.

It can be realized by writing (3.21) and (3.22) for a significant number of code blocks, that the de-interleaver can be implemented so that it operates on eight sets of rx_data at a time. For each de-interleaving pass one instance of rx_enc is returned. Since rx_enc contains 456 bit, and eight sets of rx_data contain two times 456 bit, it is evident that all the bits contained in the input to the de-interleaver are not represented in the output. This is solved by passing each set of rx_data to the interleaver two times. In practice this is done by implementing a queue of rx_data sets, as illustrated in Figure 2.6.

The interleaver is implemented in the MATLAB function `interleave.m`. The two times four sets of rx_data, are passed to the function in matrix form for convenience.

3.3.3 Channel Decoding

The coding scheme utilized in the GSM system may be viewed as a two level coding where an inner and an outer coding is made use of. This is illustrated in Figure 3.10

The inner coding scheme is here made up of the GMSK-modulation and demodulation while the outer code is a regular convolution encoding scheme.

The outer code used in GSM, and described in Section 2.1.2, is based on a rate 1/2 convolution encoder using a constraint length of 5. This implements a finite state machine on which is it possible to predetermine legal state transitions as was described in Section 3.2. This way it is

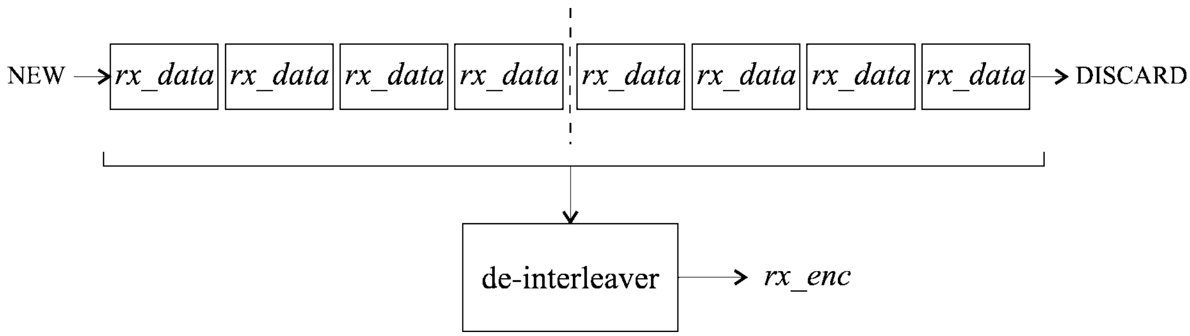


Figure 3.9: Operation of the de-interleaver, aided by a queue. The de-interleaver reads out the entire content of the queue. The queue has two times four slots, and in each interleaving pass four new sets of rx_data are pushed into the queue, and the eldest four instances are discarded. One instance of rx_enc is returned in each pass.

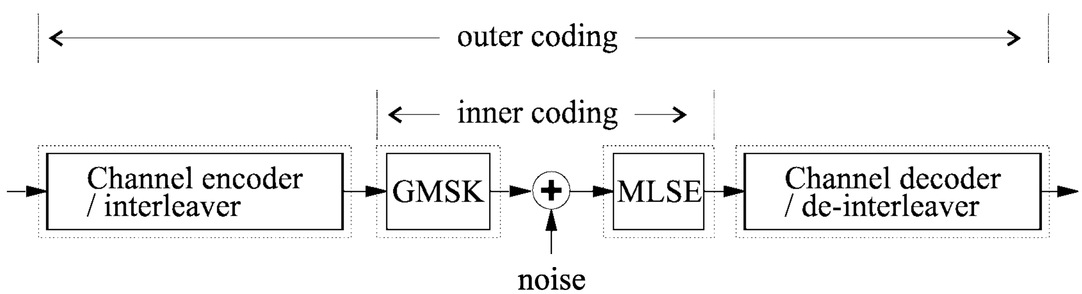


Figure 3.10: Illustration of the two level coding scheme utilized in the GSM system.

possible to build a state transition diagram that may be used in the decoding of the received sequence. Such a state transition diagram is illustrated in Figure 3.11.

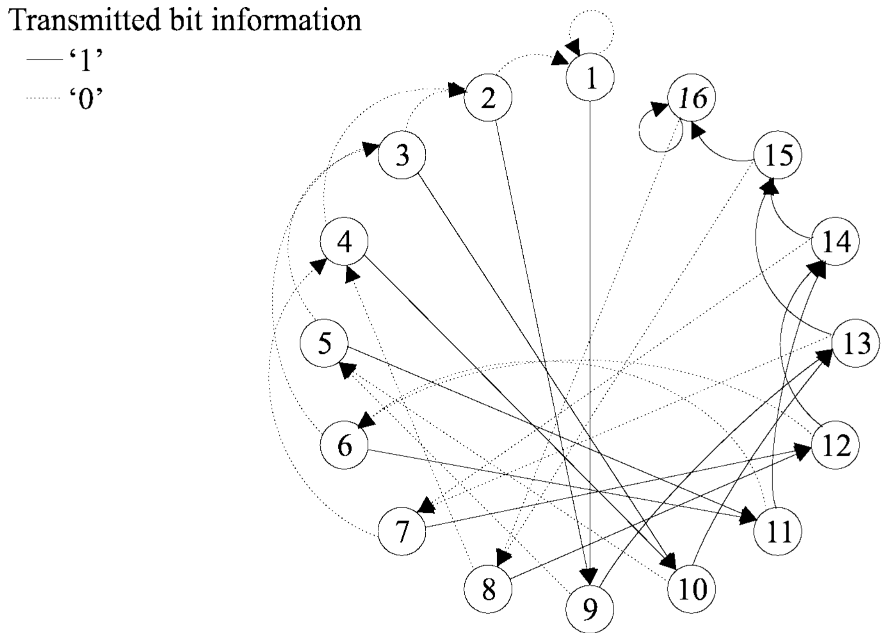


Figure 3.11: State transition diagram for the GSM system.

The state transition diagram of Figure 3.11 is deduced from the transmitter encoder with states ordered in a binary manner. That is, state 1 represents the situation where the encoder has all zeros in its registers, i.e. $s_1 = \{0\ 0\ 0\ 0\}$, while state 2 is given as $s_2 = \{0\ 0\ 0\ 1\}$. This way it is possible to characterize the encoder completely.

The optimum decoder for a convolution encoded signal is the Viterbi decoder [6] as it represents a recursive optimal solution to the problem of estimating a state transition sequence of a finite-state Markov process. As the principle of the Viterbi decoder has been described in Section 3.2, where the GMSK demodulation is described, this is not addresses further here.

Only the metric used in determining the most probable sequence is of interest as this differ from the one used in the mentioned GMSK demodulator. As the convolution decoder operates on binary information a much simpler metric definition is used. At any particular discrete time instance, k, a set of metrics is given by [6]

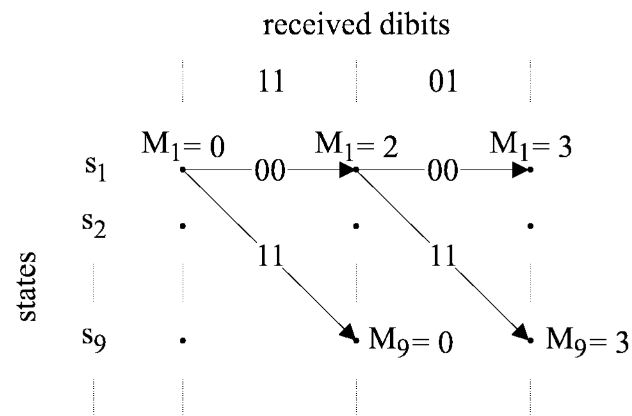
$$\lambda(s_0^k) = \sum_{i=0}^{k-1} \lambda(\xi_i), \tag{3.23}$$

where $s_0^k = \{s_0, s_1, s_2, \ldots, s_k\}$ represents a given sequence of states and ξ_i represents the i'th state transition. The metric increase definition does, in principle, not differ much from the definition in (3.19), page 30. Only, here the simple Euclidean distance measure is used to determine the metric increase for a given state transition. Hence, the following definition stands

$$\begin{aligned} \lambda(\xi_i) &= (y_k - x_k)^2 \\ &= \mathrm{xor}(y_k^{bin}, x_k^{bin}), \end{aligned} \tag{3.24}$$

where y_k^{bin} is the received binary symbol and x_k^{bin} the binary symbols expected to cause a given state transition.

The simplification in (3.24) is possible as the decoder operates on binary information signal. The calculation of this increase in the metric value for a given state transition is illustrated in Figure 3.12.



$\lambda_1 = \mathrm{XOR}\{11, 00\} = 2$ $\quad$ $\lambda_1 = \mathrm{XOR}\{01, 00\} = 1$
$\lambda_9 = \mathrm{XOR}\{11, 11\} = 0$ $\quad$ $\lambda_9 = \mathrm{XOR}\{01, 11\} = 1$
$M_1 = 0 + 2 = 2$ $\quad$ $M_1 = 2 + 1 = 3$
$M_9 = 0 + 0 = 0$ $\quad$ $M_9 = 2 + 1 = 3$

Figure 3.12: Illustration of the metric calculations that compromises part of the channel decoder implementation.

Based on the principle illustrated in Figure 3.12 and the principle of a survivor metric the entire trellis structure is formed as was the case in the GMSK demodulator.

Having determined the end state with the smallest metric value the trellis is backtracked to determine the most probable state sequence. This sequence may then be decoded to retrieve the decoded estimated transmitted bit sequence.

Note, that as the channel decoder operates on input dibits and outputs bits the data rate is reduced by a factor of two.

3.4 Receiver Test

To test the operation of the implemented receiver various tests have been carried out. During the implementation of the individual blocks testing has been performed on all levels. Here only the high level tests are presented.

Also, due to the complexity of the implemented data receiver the test results presented here are separated into two sections. The first test considers the matched filter implementation and the second considers the implemented Viterbi detector.

3.4.1 Test of `mf.m`

Testing the `mf.m` function implies at least two tests. A test of the actual channel impulse response estimation and a test of the synchronization included in the function.

To test the calculation of the impulse response a burst is generated, differentially encoded and mapped to a MSK-representation. This signal is the applied to an artificial channel and the fed to the matched filter. By comparing the artificial impulse response with the estimated one this part of `mf.m` may be verified. The test has been carried out using numerous impulse responses of varying lengths. The result of two of these tests are shown in Figure 3.13.

What Figures 3.13a and 3.13b show is that for channel impulse responses of lengths equal to 4 T_b's a correct estimation is achieved. This result goes for impulse responses of lengths equal to $1 \leq T_b \leq 5$ in fact. As the length of the responses exceed 5 T_b's errors are introduced as Figures 3.13c and 3.13d reveal. This result is not surprising when Figure 3.3 is recalled. From this figure it is clear that correct estimation can only be expected for impulse responses of lengths less that, or equal to, 5 T_b's. When this value is exceeded the correlation no longer produces only zero values besides $R_{hh}[0]$, as Figure 3.3 shows. As a result the correlation limitations of the training sequence start to affect the channel estimate.

The synchronization implemented in `mf.m` is tested using two approaches. First, various lengths of random samples are inserted in front and after the generated burst. Using the same channel impulse response for all received bursts the matched filter function is able to establish correct synchronization. This is verified manually by comparing the $burst\_start$ parameter of the function with the known correct value. Using the same verification approach the same burst is evaluated using different channel impulse responses. The response types are here chosen as [1 0 0 0 0], [0 1 0 0 0], etc. This way the synchronization may also be tested.

For both test approaches the synchronization is found to operate correctly. As the estimated impulse response also is calculated correctly the complete `mf.m` implementation is found to operate correctly.

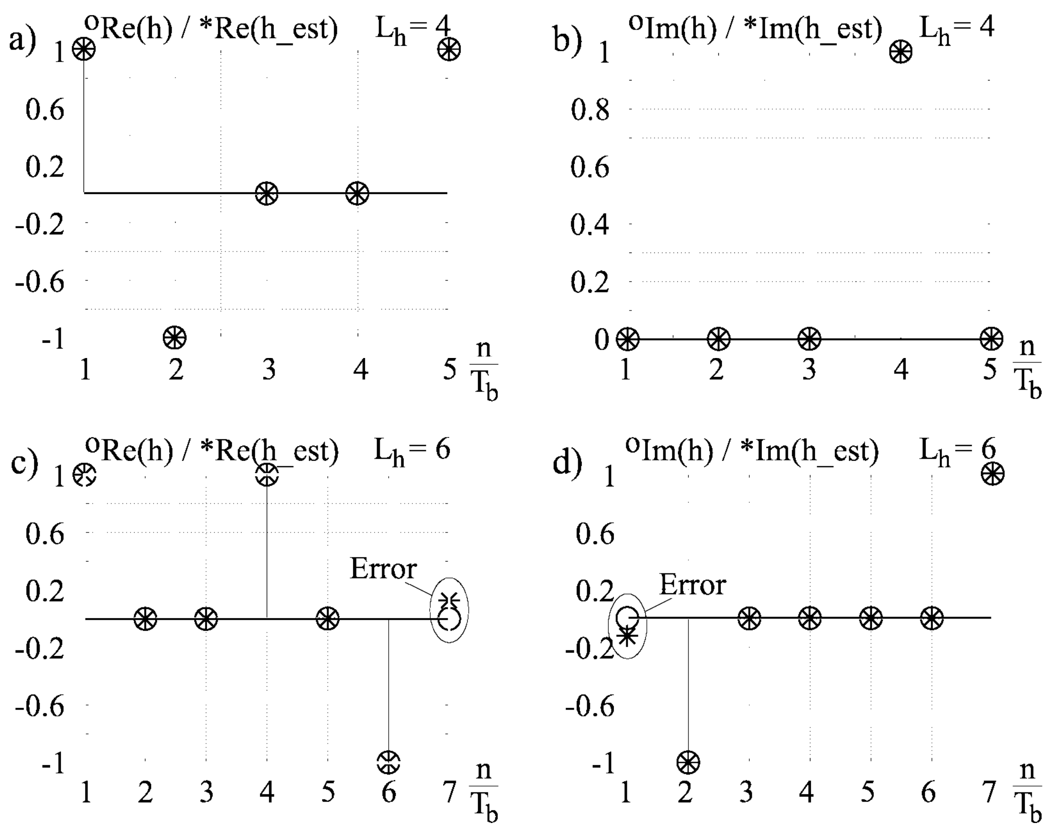


Figure 3.13: The results from two `mf.m` tests. Real parts and imaginary parts of the actual response, h, and the estimated response, h_est, are compared. Actual values are indicated using 'o' while the estimated values are indicated using '*'. a) Real parts using a L_h of 4. b) Imaginary parts using a L_h of 4. c) Real parts using a L_h of 6. d) Imaginary parts using a L_h of 6.

3.4.2 Test of `viterbi_detector.m`

The Viterbi detector has been tested in two major tests. In the first test the detector is fed a sequence of non-distorted MSK-symbols. This is done using an OSR of 1 and the following impulse response

$$h = [1,\ 0,\ 0,\ 0,\ 0], \tag{3.25}$$

and L_h is set to 5 and the corresponding value of

$$R_{hh} = [1,\ 0,\ 0,\ 0,\ 0], \tag{3.26}$$

is used. With these settings the metric of the final survivor path should be 148. To realize this, observe that 148 transitions exist. Also, the metric gain for a single transition should equal

$$\begin{aligned} Gain(Y[n], s_a, s_b) &= \Re\{I^*[n]Y[n]\} - \Re\left\{ I^*[n] \sum_{m=n-L_h}^{n-1} I[m]R_{hh}[n-m] \right\} \\ &= 1 - 0 = 1. \end{aligned} \tag{3.27}$$

The test is done by slight rewrites of the code, and using the auxiliary MATLAB test-script `viterbi_ill.m`. The result of the test is that the best path has the total metric 148, indicating correct operation. Also, the algorithm identifies this correctly, and does flawless mapping from the survivor path and back to the transmitted binary symbols. From this test, it is concluded that the metrics of a given previous survivor state is transferred correctly to the corresponding present state, and that the mapping from a path to binary information is correct. Thus, what may be referred to as the basic functionality and control flow within the algorithm is working as specified.

Having verified the control and data flow of the algorithm via the first test, a second test, intended to verify the values calculated inside the algorithm, is done. During this test, it has been verified, by comparing results found by hand calculations against those calculated by the program, that the gain values are calculated correctly. In this test, $R_{hh} = [1,\ 2,\ 3]$ was used to keep the complexity low. The test showed that values calculated by hand yield the same results as those found in internal data structures.

4

Use of the *GSMsim* Toolbox

This chapter describes the installation and use of the *GSMsim* toolbox. As is described in the foregoing part of this work, the *GSMsim* toolbox consists of 11 major functions – plus some minor sub-functions – intended to be directly interfaced by the user. To summarize, these functions are:

`data_gen.m:`	Generates random data for transmission.
`channel_enc.m:`	Performs the parity and convolutional encoding of the data bits.
`interleave.m:`	Interleavs the encoded data sequences.
`gsm_mod.m:`	Modulates the bursts and does multiplexing as well.
`mf.m:`	Performs channel estimation, synchronization, matched filtering and down sampling.
`channel_simulator.m:`	Performs simulation of transmitter front-end, channel and receiver front-end.
`viterbi_init.m:`	Sets up data structures for the Viterbi detector to use.
`viterbi_detector.m:`	Implements a hard decision only MLSE based on Ungerboeck's modified Viterbi algorithm.
`DeMUX.m:`	Does simple demultiplexing of the received data sequence.
`deinterleave.m:`	Takes care of de-interleaving the received data sequences.
`channel_dec.m:`	Performs the channel decoding.

Together these functions constitute a minimal GSM simulation base. To obtain a full GSM simulation platform RF-parts and channel models need to be added. Addition of components is easily done due to the modular implementation of the existing functions.

> **NOTE:** The function `channel_simulator` is intended for replacement by user implemented functions, and should **NOT**, under any circumstances, be used for scientific purposes.

The first part of this chapter describes some details about installing the *GSMsim* toolbox on a user level. If you desire to do a system level installation, the procedure you need to follow is likely to be similar. In the second part of the chapter a brief description of the syntax of the major functions listed above is presented. After this the chapter contain a description of two demos `GSMsim_demo.m` and `GSMsim_demo_2.m` which are included in the toolbox. The last two sections in this chapter contain profiling and convergence information.

4.1 Installation of *GSMsim*

The *GSMsim* toolbox is distributed as a GNU zipped tape archive. The procedure of extracting and installing *GSMsim* is illustrated here by a step by step example. The example assumes that

- The *GSMsim* distribution file is available as: `~/tmp/GSMsim.tar.gz`
- The directory, where *GSMsim* is intended to be installed is: `~/matlab/`
- Standard UNIX `tar` is available
- Standard GNU `gunzip` is available
- Emacs is available
- A UNIX shell, like BASH, is available
- The file `~/matlab/startup.m` is automaticly executed at MATLAB startup

The first step is to change to the desired directory location

```
cd ~/matlab
```

Then unpack the distribution

```
gunzip -c ~/tmp/GSMsim.tar.gz | tar xvf -
```

In the next step, edit the file `~/matlab/startup.m`, using your favorite text editor. For example issue the command

```
emacs ~/matlab/startup.m &
```

Then insert the lines

```
startdir=pwd ;
cd ~/matlab/GSMsim/config ;
GSMsim_config ;
eval( [ 'cd ' startdir ] ) ;
```

at the top of `~/matlab/startup.m`. Having inserted these lines save the file and exit your editor. In emacs, this is done by issuing the key sequence

```
C-x C-s C-x C-c
```

This concludes the installation of *GSMsim*. In order to test the installation, start MATLAB in a shell. This is done by simply typing `matlab`, at the prompt, and pressing `Enter`. When MATLAB has started, you may test the installation by issuing the command

```
GSMsim_demo(10,2)
```

at the MATLAB prompt. If a result similar to the one shown in Figure 4.1 appear, then the installation is a success.

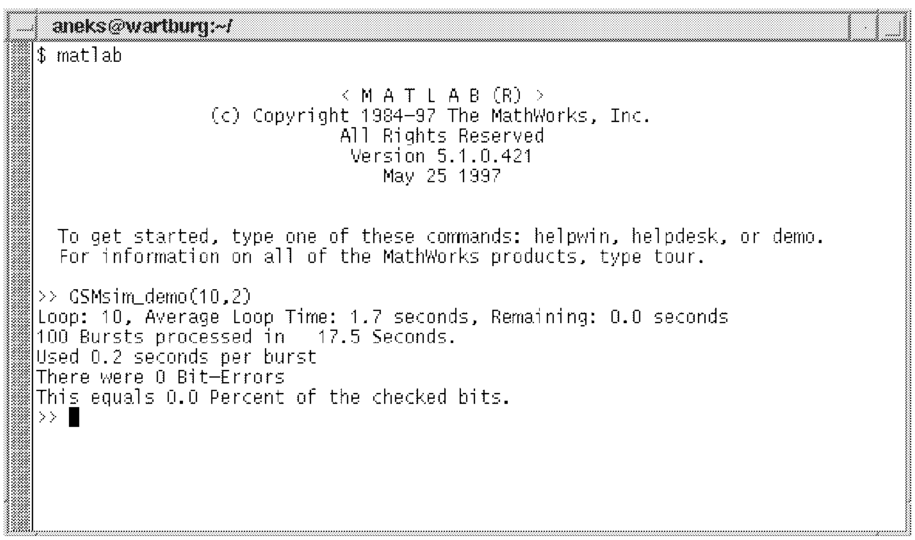


Figure 4.1: Illustration of how to test a *GSMsim* installation.

4.2 Syntax of the Major Functions

This section serves as a reference guide for the implemented functions. The aim is to supply sufficient information for the toolbox to be useful.

4.2.1 Syntax of `data_gen.m`

Matlab Call Syntax:

```
[ tx_data ] = data_gen(INIT_L)
```

Input Parameters:

`INIT_L`: An integer indicating the number of bits to be generated.

Output Parameters:

`tx_data`: The generated data,

4.2.2 Syntax of `channel_enc.m`

Matlab Call Syntax:

```
[ tx_enc ] = channel_enc(tx_block)
```

Input Parameters:

`tx_block`: A 260 bits long vector containing the raw and non processed data sequence intended for transmission.

Output Parameters:

`tx_enc`: A 456 bits long vector containing the now encoded data sequence. This includes parity encoding, addition of check bits, and convolution encoding.

4.2.3 Syntax of `interleave.m`

Matlab Call Syntax:

```
[ tx_data_matrix ] = interleave(tx_enc0,tx_enc1)
```

Input Parameters:

`tx_enc0`: The previous instance of `tx_enc` returned from the channel coder.

`tx_enc1`: The latest instance of `tx_enc` returned from the channel coder.

Output Parameters:

`tx_data_matrix`:
The four sets of `tx_data` produced in a single interleaver pass. Each set is placed in a row of the matrix. The first row contain the instance of `tx_data` which is to be transmitted first, and row four contains the instance of `tx_data` which is to be transmitted last.

4.2.4 Syntax of `gsm_mod.m`

Matlab Call Syntax:

```
[ tx_burst , I , Q ] =
            gsm_mod(Tb,OSR,BT,tx_data,TRAINING)
```

Input Parameters:

`Tb:`	Bit time in seconds.
`OSR:`	Oversampling ratio. Here the oversampling ratio is defined as: f_s/r_b, where f_s is the sample frequency, and r_b is the bit rate.
`BT:`	Bandwidth bit time product. Usually 0.3.
`tx_data:`	The contents of the data fields in the burst to be transmitted, represented as a binary row vector, using ones and zeros.
`TRAINING:`	Training sequence which is to be inserted in the burst. Represented as a row vector. Binary format is used in the form of ones and zeros.

Output Parameters:

`tx_burst:`	The entire transmitted burst, represented as an binary row vector, using ones and zeros.
`I:`	In-phase part of modulated burst. The format is a row vector of real floating point numbers.
`Q:`	Quadrature part of modulated burst. The format is a row vector of real floating point numbers.

4.2.5 Syntax of `channel_simulator.m`

Matlab Call Syntax:

```
[ r ] = channel_simulator(I,Q,OSR)
```

Input Parameters:

`I:`	In-phase part of modulated burst. The format is a row vector of real floating point numbers.
`Q:`	Quadrature part of modulated burst. The format is a row vector of real floating point numbers.
`OSR:`	Oversampling ratio.

Output Parameters:

`r:`	Complex baseband representation of the received GMSK-modulated signal. The format is a row vector consisting of complex floating point numbers.

4.2.6 Syntax of `mf.m`

Matlab Call Syntax:

```
[ Y, Rhh ] = mf(r,Lh,T_SEQ,OSR)
```

Input Parameters:

`r:`	Complex baseband representation of the received GMSK-modulated signal as it is returned from the channel simulator. The format is a row vector consisting of complex floating point numbers.
`Lh:`	The desired length of the matched filter impulse response measured in bit time durations.
`T_SEQ:`	A MSK-mapped representation of the 26 bits long training sequence used in the transmitted burst, i.e. the training sequence used in the generation of `r`.
`OSR:`	Oversampling ratio.

Output Parameters:

`Y:`	Complex baseband representation of the matched filtered and down converted received signal. Represented as a complex valued row vector.
`Rhh:`	Autocorrelation of the estimated channel impulse response. Represented as a Lh+1 elements long complex valued column vector starting with Rhh[0], and ending with Rhh[Lh].

4.2.7 Syntax of `viterbi_init.m`

Matlab Call Syntax:

```
[ SYMBOLS , PREVIOUS , NEXT , START , STOPS ] =
            viterbi_init(Lh)
```

Input Parameters:

`Lh:`	The length of the matched filter impulse response measured in bit time durations.

Output Parameters:

`SYMBOLS:`	State number to MSK symbols translation table.
`NEXT:`	Present state to next state transition table.
`PREVIOUS:`	Present state to previous state transition table.
`START:`	Start state number.
`STOPS:`	Set of legal stop states.

4.2.8 Syntax of `viterbi_detector.m`

Matlab Call Syntax:

```
[ rx_burst ] = viterbi_detector(SYMBOLS,
               NEXT,PREVIOUS,START,STOPS,Y,Rhh)
```

Input Parameters:

`SYMBOLS:`	State number to MSK symbols translation table.
`NEXT:`	Present state to next state transition table.
`PREVIOUS:`	Present state to previous state transition table.
`START:`	Start state number.
`STOPS:`	Set of legal stop states.
`Y:`	Complex baseband representation of the matched filtered and down converted received signal. Represented as a complex valued row vector.
`Rhh:`	Autocorrelation of the estimated channel impulse response. Represented as a Lh+1 elements long complex valued column vector starting with Rhh[0], and ending with Rhh[Lh].

Output Parameters:

`rx_burst:`	The most likely sequence of symbols. Representation is a row vector consistent of binary symbols, represented as zeros and ones.

4.2.9 Syntax of `DeMUX.m`

Matlab Call Syntax:

```
[ rx_data ] = DeMUX(rx_burst)
```

Input Parameters:

`rx_burst`: The received GSM burst as estimated by the Viterbi detector.

Output Parameters:

`rx_data`: The contents of the data fields in the received burst.

4.2.10 Syntax of `deinterleave.m`

Matlab Call Syntax:

```
[ rx_enc ] = deinterleave(rx_data_matrix)
```

Input Parameters:

`rx_data_matrix`:
A matrix containing eight instances of `rx_data`. Each instance is aligned in a row. The data are arranged so that the eldest instance of `rx_data` is kept in row number one, and the latest arrived instance is kept in row number eight.

Output Parameters:

`rx_enc` The received code block, as reconstructed from the eight instances of `rx_data`.

4.2.11 Syntax of `channel_dec.m`

Matlab Call Syntax:

```
[ rx_block, FLAG_SS, PARITY_CHK ] = channel_dec(rx_enc)
```

Input Parameters:

`rx_enc:`	A 456 bits long vector containing the encoded data sequence as estimated by the Viterbi equalizer. The format of the sequence must be according to the GSM 05.03 encoding scheme.

Output Parameters:

`rx_block:`	A 260 bits long vector containing the final estimated information data sequence.
`FLAG_SS:`	Indication of whether the correct stop state was reached. Flag is set to '1' if an error has occurred here.
`PARITY_CHK:`	The 3 parity check bit inserted into the transmitted bit sequence.

4.3 The GSMsim_demo.m Function

GSMsim_demo.m is a implementation of an example of a GSM simulation platform based on the major functions described in the beginning of this chapter, but leaving out the channel coding and interleaving. This reduced simulation is useful, for example, in the case where type II performance is of primary interest. The call syntax of the function is

```
GSMsim_demo(LOOPS,Lh)
```

where LOOPS indicate how many times the function is to process ten GSM bursts. The algorithm which form the basis for GSMsim_demo.m is

```
viterbi_init
for n=1:LOOPS do
  for n=1:10 do
    data_gen
    gsm_mod
    channel_simulator
    mf
    viterbi_detector
    DeMUX
    Count errors.
  end for
  Update display.
end for
Present simulation result on screen.
```

Note, that the function processes ten bursts between each screen update. This is motivated by the fact that in networked environments, as Aalborg University, screen updates may take up unreasonable much time. This is worth remembering when implementing custom simulation scripts. In general the GSMsim_demo.m may serve as a starting point for creating such scripts.

4.4 The `GSMsim_demo_2.m` Function

`GSMsim_demo_2.m` is an example of a GSM simulation which includes all the functions available in the *GSMsim* toolbox. That is to say that channel coding and interleaving is also included, in contrast to what is the case in `GSMsim_demo`. Also `GSMsim_demo_2`, includes an example of how to create a simulation log. The call syntax of `GSMsim_demo_2` is

```
GSMsim_demo_2(NumberOfBlocks,Lh,LogName)
```

where `NumberOfBlocks` regulates how many instances of `tx_block` that is processed in a simulation, and `LogName` indicates a basename which is to be used for the name of the simulation log. The simulation script will output a log to a file called

```
"LogName"_"NumberOfBlocks"_Lh.sim
```

The basic algorithm of `GSMsim_demo_2` is

```
viterbi_init
for n=1:NumberOfBlocks do
  data_gen
  channel_enc
  interleave
  for generated bursts do
    gsm_mod
    channel_simulator
    mf
    viterbi_detector
    DeMUX
  end for
  deinterleave
  channel_dec
  Count errors.
  Update logfile.
  Update display.
end for
Present simulation result on screen.
```

During the simulation a status report is continuously updated at the screen, showing the progress along with the remaining simulation time. Note that the simulation log is saved during the simulation, and not at the end of the simulation. This provide for recovery of the simulation results in the case of a system crash or other failures.

Using the `GSMsim_demo_2.m` four Bit Error Rates are produced

Type Ia BER: The Bit Error Rate within the decoded type Ia bits.

Type Ib BER: The Bit Error Rate within the decoded type Ib bits.

Type II BER: The Bit Error Rate within the unprotected type II bits.

Type II BER-CHEAT: This Bit Error Ratio is constructed by considering all the bits in the received blocks as unprotected type II bits, and is thus the same as the Type II BER but with a much more substantial statistical basis.

All the results are measured in percent.

4.5 Performance

The `GSMsim_demo_2.m` is useful for evaluating which part of an simulation takes the major part of the time. This is done by using the `profile` command available in MATLAB version 5. The profiling is done for $L_h \in \{2, 3, 4\}$. The results are shown in Figures 4.2, 4.3 and 4.4.

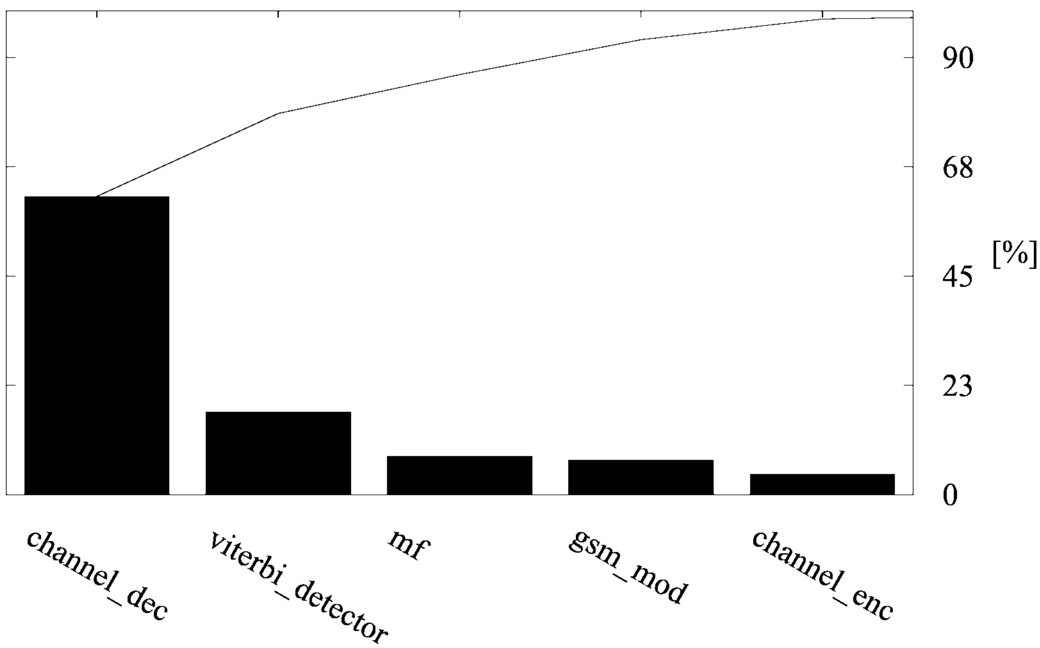


Figure 4.2: Profile for `GSMsim_demo_2.m` using $L_h = 2$, the simulation is done for 100 blocks.

As it can be seen from Figures 4.2, 4.3 and 4.4 it is not advisable to include channel encoding in the simulations if type I bit error rates are not of specific interest. Note that tables describing coding gain for various coding techniques do exist [16].

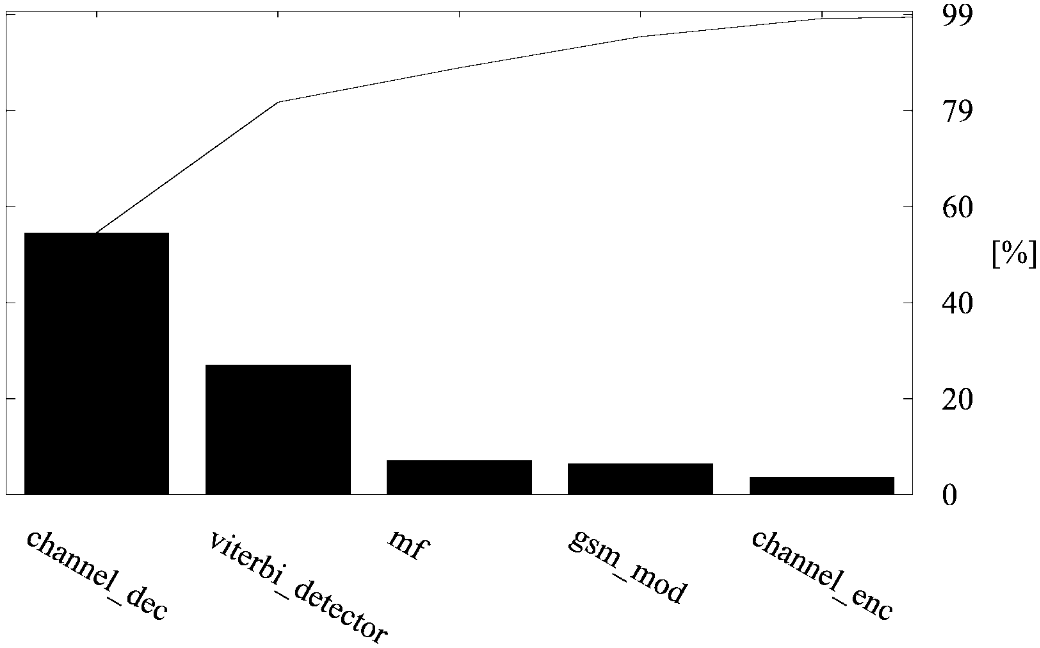


Figure 4.3: Profile for `GSMsim_demo_2.m` using $L_h = 3$, the simulation is done for 100 blocks.

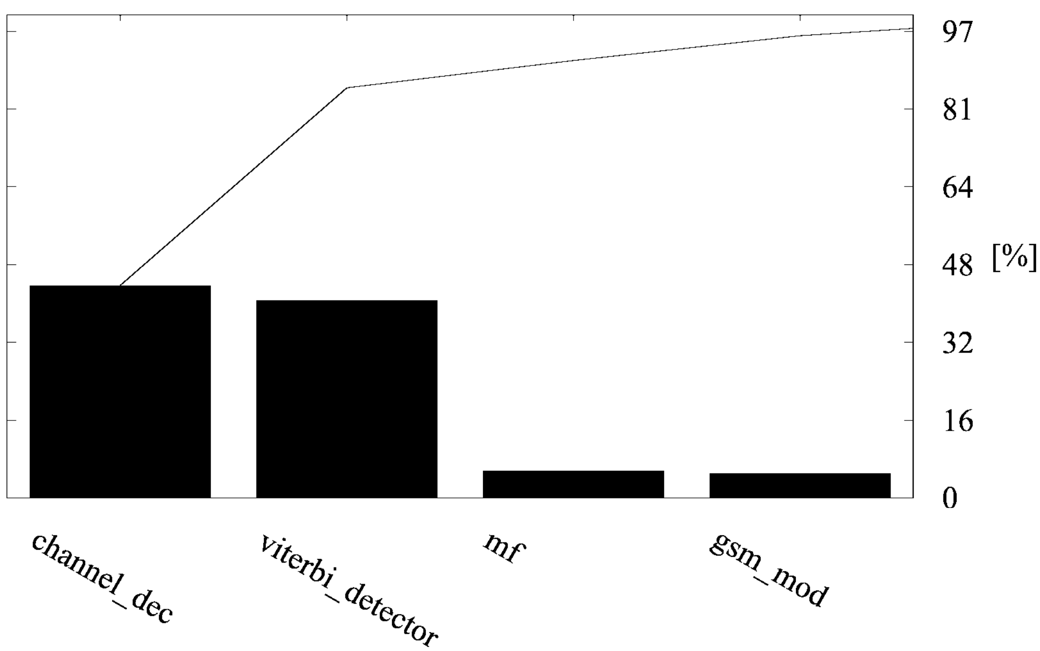


Figure 4.4: Profile for `GSMsim_demo_2.m` using $L_h = 4$, the simulation is done for 100 blocks.

4.6 Convergence

This section aims to illustrate the simulation length required for the resulting BER estimates to converge. In order to get estimates of the convergence for all types of Bit Error Rates produced by the *GSMsim* toolbox the `GSMsim_demo_2` script is used.

To illustrate the convergence of the results three simulations are run for $10,000$ blocks equaling $40,000$ bursts. In the three simulations L_h is set to $2, 3$ and 4, respectively. In order to get an impression of the convergence, the four Bit Error Rates described in Section 4.4 are plotted in Figures 4.5, 4.6 and 4.7.

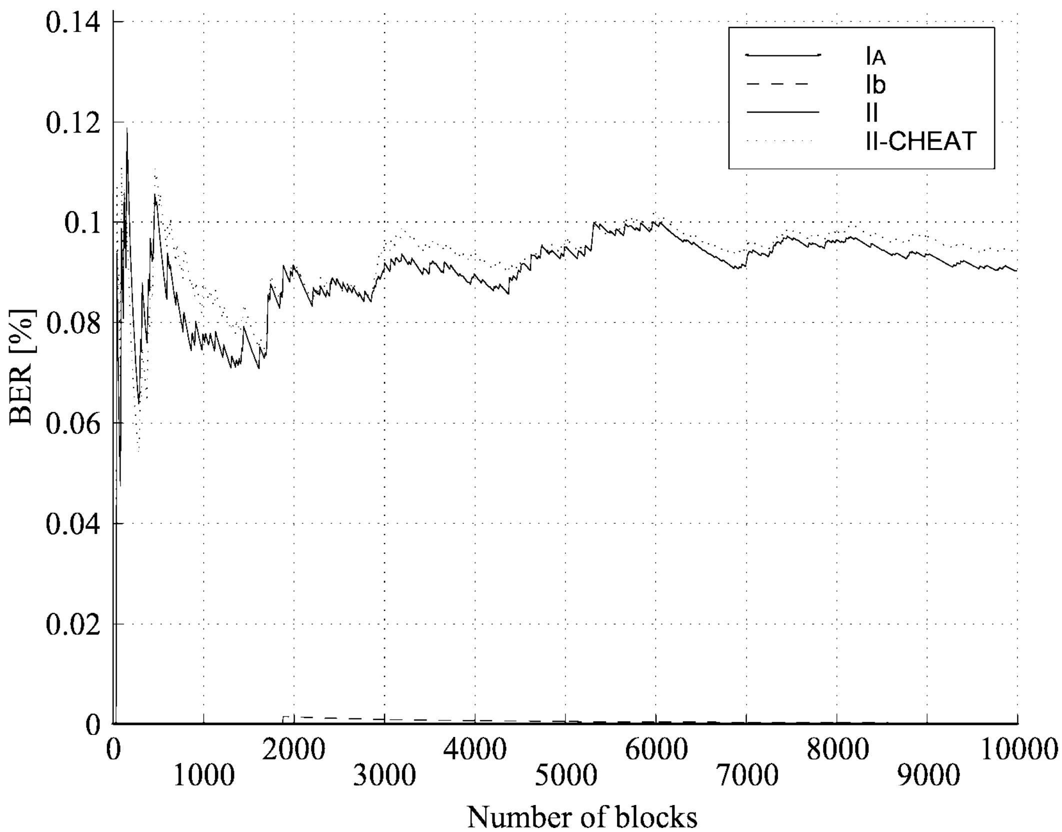


Figure 4.5: The convergence properties, illustrated by using $L_h = 2$. The top line is the type II-cheat BER curve. Immediately bellow this line is the actual type II BER curve. Both type I curves are almost at the zero line. The simulation result is plotted for each tenth simulated burst.

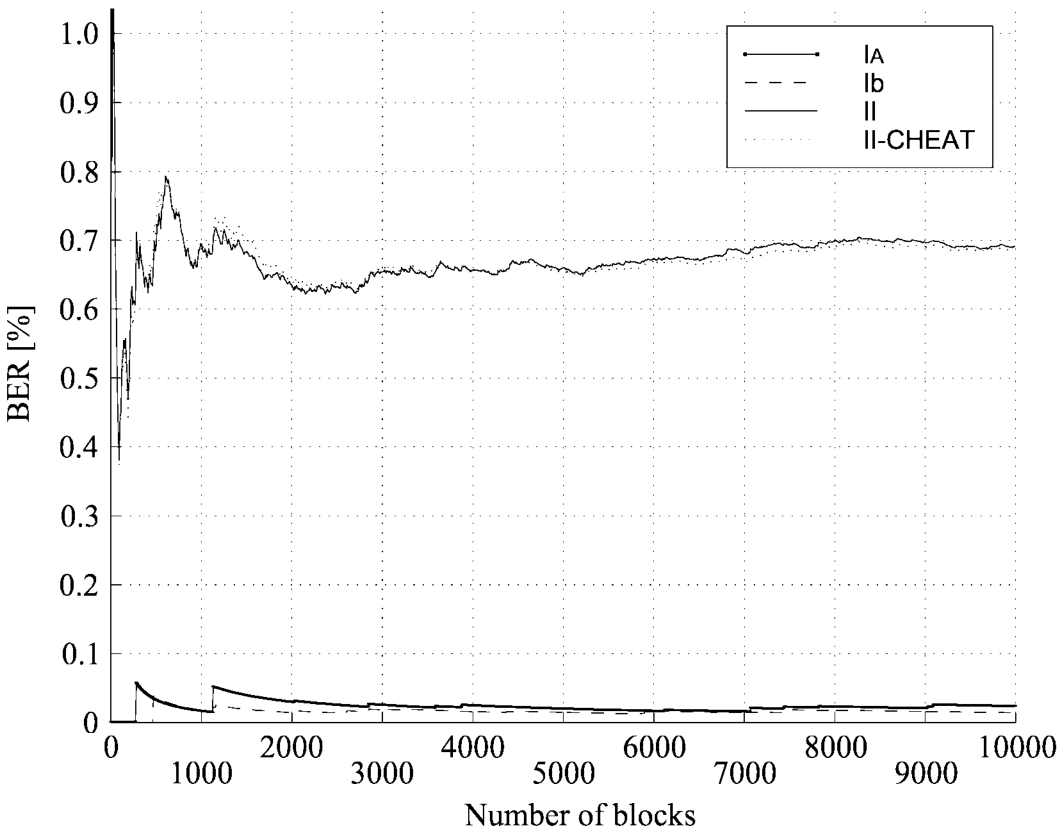


Figure 4.6: The convergence properties, illustrated by using $L_h = 3$. The two type II BER curves are almost identical, and are located at the top of the graph. Likewise, the two type I curves are almost identical. The simulation result is plotted for each tenth simulated burst.

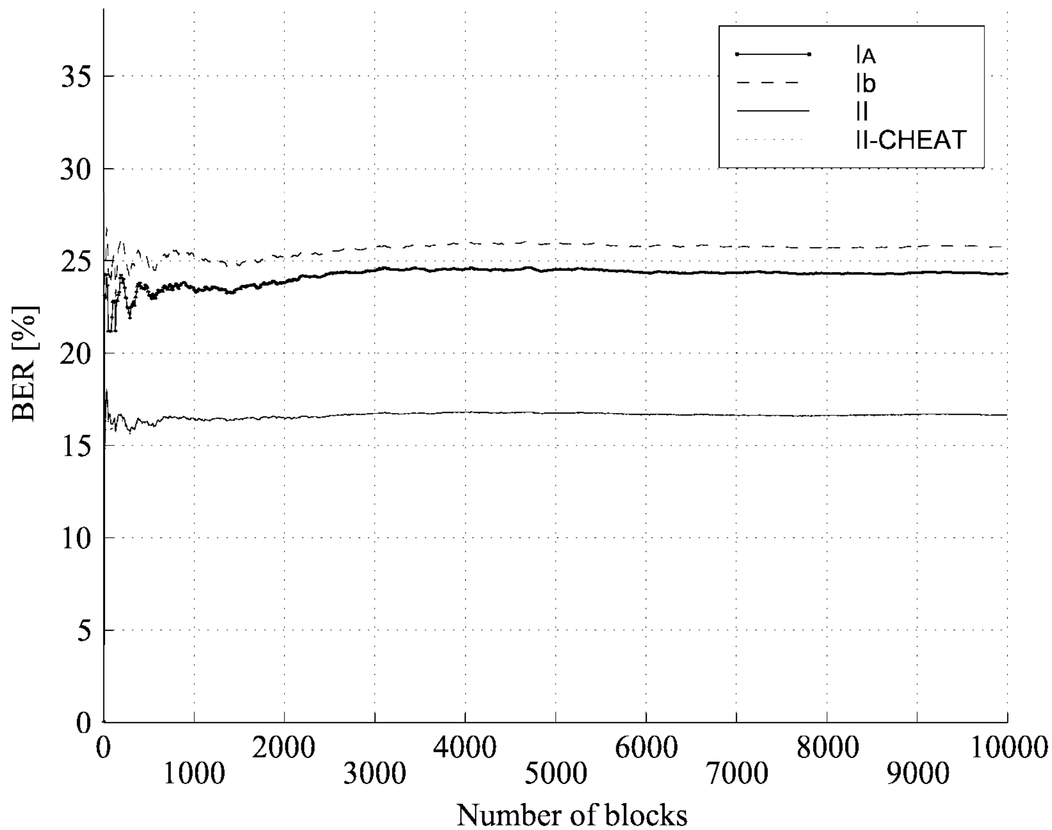


Figure 4.7: The convergence properties, illustrated by using $L_h = 4$. The type II error curves are nearly equal, and tend to converge to about 17%. The top curve represents type Ib errors. The type Ia curve is second from the top. The simulation result is plotted for each tenth simulated burst.

References

[1] Asad A. Abidi. CMOS-only RF and Baseband Circuits for a Monolithic 900 MHz Wireless Transceiver. In *Bipolar Circuits & Tech. Mtg.*, pages 35–42, October 1996.

[2] Julian Bright; Stephen McClelland; Bhawani Shankar. Microwave in Europe: Trends Shaping the Future. *Microwave Journal*, September 1996.

[3] Johan Brøndum. GSM data receiver structures. Technical report, Institute for Electronic Systems Department of Communication Technology, 1993.

[4] Jan Crols & Michel S. J. Steyaert. A Single-Chip 900 MHz CMOS Receiver Front-End with a High Performance Low-IF Topology. *IEEE Jour. of Solid-State Circuits*, 1995.

[5] Jan Crols & Michiel Steyaert. A Full CMOS 1.5 GHz Highly Linear Broadband Downconversion Mixer. *IEEE Jour. of Solid-State Circuits*, 30(7):736–742, jul 1995.

[6] G. David Forney, Jr. The Viterbi Algorithm. In *Proceedings of the IEEE*, volume 61, pages 268–273, march 1973.

[7] Armond Hairapetian. An 81 MHz IF Receiver in CMOS. In *1996 IEEE Int. Solid-State Circuits Conference*, pages 56–57, 1996.

[8] Simon Haykin. *Communication Systems*. John Wiley & Sons, 2 edition, 1983.

[9] European Telecommunications Standards Institute. GSM 05.03: Channel Coding. Version: 4.6.0, July 1993.

[10] European Telecommunications Standards Institute. GSM 05.04: Modulation. Version: 4.6.0, July 1993.

[11] European Telecommunications Standards Institute. Digital cellular telecommunications system (Phase 2); Radio transmission and reception (GSM 05.05). Ninth Edition, November 1996.

[12] Benny Madsen. Data-Receiver For the Pan-European Mobile Telephone System (GSM). EF 226, Industrial Research Project, The Danish Academy of Technical Sciences, September 1990.

[13] The MathWorks, Inc., Cochituate Place, 24 Prime Park Way, Natick, Mass. 01760. *MATLAB Reference Guide*.

[14] Michel Mouly and Marie-Bernardette Pautet. *The GSM System for Mobile Communications*. M. Mouly et Marie-B. Pautet, 1992.

[15] Anders Østergaard Nielsen, Henrik Refsgaard Hede, Klaus Ingemann Pedersen, Michael Hedelund Andersen, Thomas Almholdt, and Troels Emil Kolding. DSP realtime implementation of GSM testbed system. Student Report by group 870 at Department of Communication Technology, Institute for Electronic Systems, Aalborg University, June 1995.

[16] John G. Proakis. *Digital Communications*. McGraw-Hill Book Company, 2 edition, 1989.

[17] Behzad Razavi; Kwing F. Lee; Ran H. Yan. Design of High-Speed, Low-Power Frequency Dividers and Phase-Locked Loops in Deep Submicron CMOS. *IEEE Jour. of Solid-State Circuits*, 30(2):101–109, feb 1995.

[18] Jans Sevenhans & Dirk Rabaey. The Challenges for Analogue Circuit Design in Mobile Radio VLSI Chips. *Microwave Engineering Europe*, pages 53–59, may 1993.

[19] Chung-Yo Wu & Shuo-Yuan Hsiao. The Design of a 3-V 900-MHz CMOS Bandpass Amplifier. *IEEE Jour. of Solid-State Circuits*, 32(2):159–168, February 1997.

A

Transmitter Implementations

AS described previously the implemented transmitter consists of three blocks made up of in all five MATLAB functions. The functions of these and their placement in the transmitter structure is illustrated in Figure A.1.

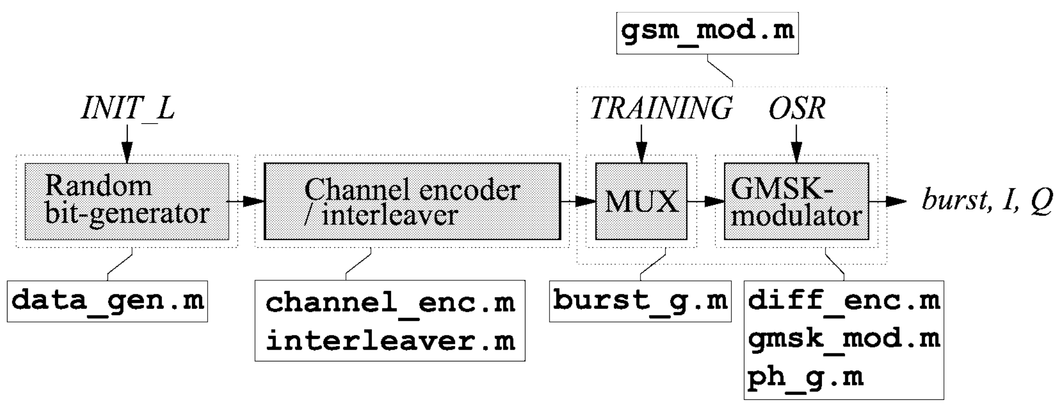


Figure A.1: Illustration of the five implemented functions constituting the transmitter. Also included is a sixth function that combines the other five functions into a single transmitter function call.

As shown in Figure A.1 a sixth function is added to the transmitter implementation. This is done to allow for a single function call to access the entire transmitter. It should be noted here that while the GMSK-modulator returns only I and Q as outputs the variable tx_burst is a result of the `gsm_mod.m` function. The variable tx_burst is in fact the output sequence returned by the MUX block. Hence, tx_burst contains the burst sequence as it is found prior to both differential encoding and modulation. The syntax and the required input and output parameters of gsm_mod.m are describen in Section 4.2.4.

Due to the simplicity of the transmitter implementation this is described as a whole here. This is opposed to describing the five functions in individual appendices. The following two sections thus describe the data generator and multiplexer and the GMSK-modulator implementations, respectively.

A.1 Data Generator

The data generator is implemented by the MATLAB function data_gen.m. This function serves to produce random data to the channel encoder. This is to emulate the speech encoder.

A.1.1 Input, Output, and Processing

The data generator block has the following inputs:

$INIT\_L$: An integer determining the number of random data bits that the data_gen.m routine is to return.

The corresponding output is:

tx_block: The random bits generated by the function.

A.2 Channel Encoder

The channel encoder operation is implemented by the MATLAB function channel_enc.m. The task of this function is to implement the outer encoding required for use in the GSM system.

A.2.1 Input and Output

The channel encoder makes use of the following input parameter

tx_block: A 260 bits long vector contaning the data sequence intended for transmission.

The corresponding output from `channel_enc.m` is

tx_enc: The resulting 456 bits long vector contaning the encoded data sequence.

A.2.2 Internal Data Flow

Besides from the above mentioned information carrying parameters the channel encoder also operates some internal information.

The GSM encoding scheme operates using two levels of bits where the more important are those that affects the speach quality the most. These bits, termed class I bits, are furthermore split into class Ia and classe Ib bits. This separation is also made use of in `channel_enc.m` where the variables $c1$, $c1a$, $c1b$, and $c2$ are used to represent the class I, the class Ia, the class Ib, and the class II bits, respectively.

A.2.3 Processing

First the input, tx_block, is split into the different classes

$c1a = tx\_block(1 : 50)$
$c1b = tx\_block(51 : 182)$
$c2 = tx\_block(183 : 260)$

Having split the data the $c1a$ bits are parity encoded using three check bits. Due to the syntax of the MATLAB function $deconv.m$ some post processing is required to have the parity bit result in binary format.

$g = [1\ 0\ 1\ 1]$
$d = [c1a\ 0\ 0\ 0]$
$[q, r] = deconv(d, g)$
$L = length(r)$
$out = abs(r(L - 2 : L))$
for $n = 1 : length(out)$ **do**
  if $ceil(out(n)/2)\ = floor(out(n)/2)$ **then**
    $out(n) = 1$
  else
    $out(n) = 0$
  end if
end for

$c1a = [c1a\ out]$

The next step is to recombine the class I bits and then perform the convolutional encoding of these.

$c1 = [c1a\ c1b\ 0\ 0\ 0\ 0]$
$register = zeros(1, 4)$
$data\_seq = [register c1]$
$enc\_a = zeros(1, 189)$
$enc\_b = zeros(1, 189)$
$encoded = zeros(1, 378)$
for $n = 1 : 189$ **do**
  $enc\_a(n) = xor(xor(data\_seq(n + 4), data\_seq(n + 1)), data\_seq(n))$
  $enc\_temp = xor(data\_seq(n + 4), data\_seq(n + 3))$
  $enc\_b(n) = xor(xor(enc\_temp, data\_seq(n + 1)), data\_seq(n))$
  $encoded(2 * n - 1) = enc\_a(n)$
  $encoded(2 * n) = enc\_b(n)$
end for

Finally the now encoded class I bits are recombined with the class II bits to form the final output.

$tx\_enc = [encoded\ c2]$

A.3 Interleaver

As described in Section 2.1.3 the interleaver is implemented in the function `interleave.m`. The purpose of the interleaver is to ensure that the bit errors that occur in the received encoded data blocks are uncorrelated.

A.3.1 Input, Output, and Processing

The interleaver has two input variables

tx_enc0:	The previous code block returned from the channel coder.
tx_enc1:	The latest GSM code block returned from the channel coder.

The output from the interleaver is

tx_data_matrix: The four sets of tx_data produced in a single interleaver pass. Each set is placed in a row of the matrix. The first row contain the instance of tx_data which is to be transmitted first, and row four contains the instance of tx_data which is to be transmitted last.

The interleaver is externally aided by a queue, which administrates the propper alignment of the tx_block variables. Internally the interleaver simply perform a number of copy operations as described by the formulas in (2.6) and (2.7) in Section 2.1.3. The file `interleave.m` is constructed by the aid of the following lines

$Blocks = 1$
$BitsInBurst = 113$
$out = fopen('interleave.tmp','w')$
for $T = 0 : 3$ **do**
  for $t = 0 : BitsInBurst$ **do**
    $b = mod((57 * mod(T, 4) + t * 32 + 196 * mod(t, 2)), 456)$
    $B = floor((T - mod(b, 8))/4)$
    $fprintf(out,'tx\_data\_matrix(\%d, \%d) = tx\_enc\%d(\%d);$
      $\backslash n', T + 1, t + 1, B + 1, b + 1)$
  end for
end for
$fclose(out)$

A.4 Multiplexer

The multiplexer operation is implemented by the MATLAB function `burst_g.m`. The operation of this function serve to produce GSM burst frames according to the prescribed formats dictated in the GSM recommendations [10].

A.4.1 Input, Output, and Processing

The multiplexer block has the following input

tx_data: A 114 bit long data sequence.
$TRAINING$: A 26 bit long MSK representation of the desired training sequence to be included in the GSM burst.

The corresponding output from `burst_g.m` is

tx_burst:	The required GSM burst bit sequence including tail, control, and training sequence bits.

The implementation is done in a simple way as

$$TAIL = [000]$$
$$CTRL = [1]$$
$$tx\_burst = [TAIL\ tx\_data(1:57)\ CTRL\ TRAINING\ CTRL\ tx\_data(58:114)\ TAIL]$$

after which poin the variable tx_burst contains a valid GSM normal burst, which is then modulated as described in the next section.

A.5 GMSK-Modulator

The GMSK-modulator operation is implemented by three MATLAB functions, which are named `diff_enc.m`, `ph_g.m`, and `gmsk_gen.m` respectively. The combined operation of these functions serves to differential encode the GSM burst, as received from `burst_g.m`, and perform the GMSK-modulation according to the prescriptions dictated in the GSM recommendations [10].

A.5.1 Input and Output

The combined GMSK-modulator has the following inputs

tx_burst:	The required GSM burst bit sequence including tail, control, and training sequence bits as returned by the `burst_g.m` routine.
T_b:	Bit time duration in seconds.
OSR:	Oversampling ratio. Here the oversampling ratio is defined as: f_s/r_b, where f_s is the sample frequency, and r_b is the bit rate.
BT:	Bandwidth bit time product. Usually 0.3.

The corresponding output from `gmsk_mod.m` is:

i / q:	In-phase and quadrature-phase parts of modulated burst, respectively.

A.5.2 Internal Data Flow

Besides from the above mentioned information carrying parameters the GMSK-modulator block also exchanges some internal information. More specifically, the following parameters are used to parse internal information

- $diff\_enc\_data$: The differential encoded version of the GSM normal burst. This variable is returned as output from `diff_enc.m` and serves as input to `gmsk_mod.m`.
- L: The truncation length used to limit the time duration of the otherwise infinite length Gaussian pulse. This value is in `ph_g.m` defined to 3.
- g_fun: The resulting L times OSR values of the resulting frequency pulse function as returned by `pg_g.m`.

A.5.3 Processing

The bursts are differentially encoded before the actual modulation. This come into expression in the following

$$burst = diff\_enc(tx\_burst)$$
$$[I,\ Q] = gmsk\_mod(burst,\ Tb,\ OSR,\ BT)$$

As `gmsk_mod.m` makes use of the sub-function `ph_g.m` presenting the code for this might be appropriate in illustrating how these two functions are interlinked. Thus, the detailed implementation is as follows

$[g,\ q] = ph\_g(Tb,\ OSR,\ BT)$
$bits = length(burst)$
for $n = 1 : bits$ **do**
$f\_res((n-1) \cdot OSR + 1 : (n+2) \cdot OSR)$
$\quad = f\_res((n-1) \cdot OSR + 1 : (n+2) \cdot OSR) + burst(n) \cdot g$
end for
$theta = \pi \cdot cumsum(f\_res)$
$I = \cos(theta)$
$Q = \sin(theta)$

At this point the variables I and Q contain the in-phase and the quadrature-phase outputs from the GMSK-modulation, respectively.

B

Receiver Implementations

As described previously the implemented receiver is separated into two main blocks. One that handles the task of synchronization, channel estimation, as well as matched filtering and another block handling the decoding of the received and matched filtered signal. To implement this a total of eight MATLAB functions are generated. These functions and their placement in the receiver structure is illustrated in Figure B.1.

Note in Figure B.1, that the MLSE block which was shown in the, otherwise similar, Figure 3.1 has been explicitly divided into two blocks as discussed in Section 3.2, page 31. Furthermore, Figure B.1 shows that the VA block consists of several functions, in fact seven in all. The matched filter, channel estimation and synchronization are, however, implemented in a single function. Hence, the following sections describe the matched filter and channel estimation/synchronization as a combined block while the VA is described in a separate section.

B.1 Synchronization, Channel Estimation, and Matched Filtering

The combined task of synchronizing the received burst, performing the channel estimation and matched filtering is here implemented in a single function `mf.m`. The combined operation of these tasks aims to remove the transmitter pulse shaping and channel effects through equalizing as well as to find the optimum sample points in the received burst.

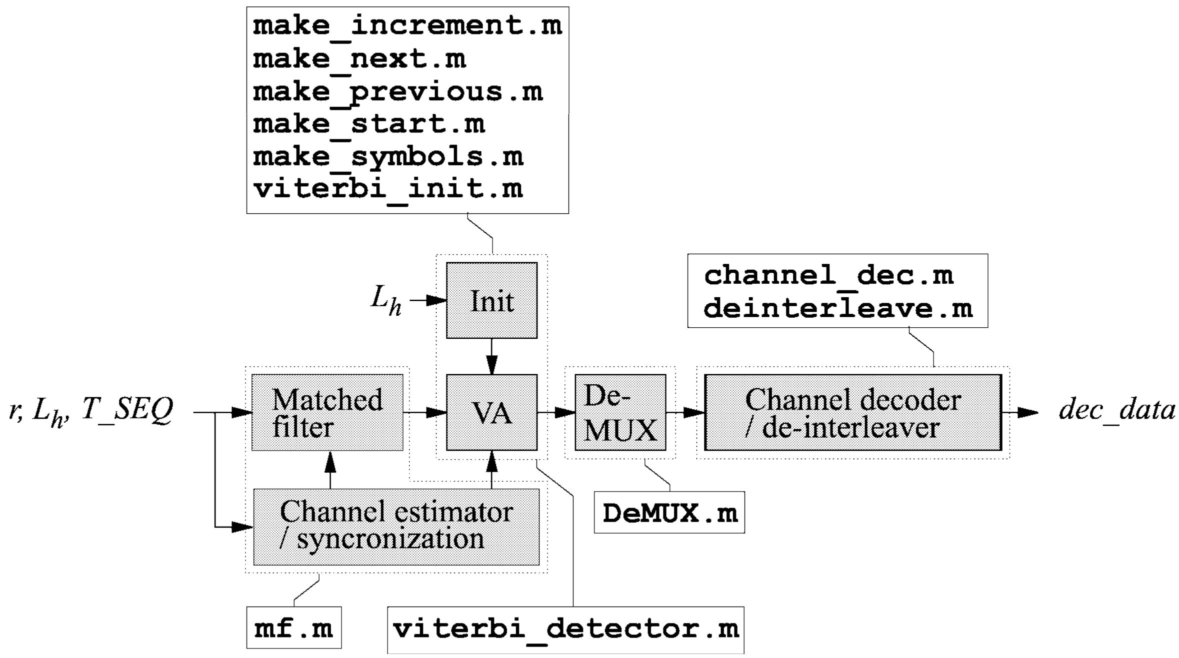


Figure B.1: Illustration of the ten implemented functions constituting the receiver data detector.

B.1.1 Input and Output

The operations combined in `mf.m` result in the following inputs:

r:	Complex baseband representation of the received GMSK modulated signal.
L_h:	The desired length of the matched filter impulse response measured in bit time durations.
T_SEQ:	A MSK-modulated representation of the 26 bits long training sequence used in the transmitted burst, i.e. the training sequence used in the generation of r.
OSR:	The oversample ratio defined as f_s/r_b.

The corresponding outputs from `mf.m` is:

Y:	A complex baseband representation of the matched filtered and down converted received signal.
R_{hh}:	The autocorrelation of the estimated channel impulse response. The format is a L_h+1 element column vector starting with $R_{hh}[0]$ and ending with $R_{hh}[L_h]$

B.1.2 Internal Data Flow

To link the three different tasks included in `mf.m` a number of internal variables are made use of. Two of these are $T16$ and r_sub, where the former contains the 16 most central bits of the otherwise 26 bits long training sequence, T_{SEQ}. The latter, r_sub, contains a sub-set of the received burst r. This sub-set is chosen in a manner that ensures that the training sequence part of the received burst is present in r_sub. This is done by not only extracting the sixteen most central bit time durations of r but rather extract extra samples preceding – and also succeeded – the central sixteen bit time durations, as illustrated in Figure B.2.

The two extra sequences each correspond to a time period of approximately 10 T_b. If the 16 most central training sequence bits are not located within the resulting sub-set the GSM network guard time is exceeded and the received burst would be corrupted by burst collision anyways. Hence, instead of searching for the training sequence through out the entire received burst, only r_sub needs to be evaluated.

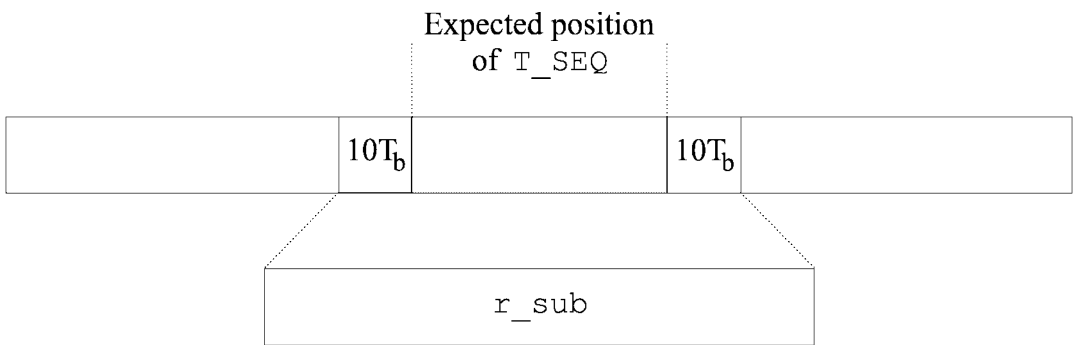


Figure B.2: Extraction of r_sub from r.

On basis of these two sub-sequences, $T16$ and r_sub, the function `mf.m` forms a channel estimate, $chan\_est$, by calculating the cross correlation between the sub-sequences. From $chan\_est$ a power estimate sequence, $power\_est$, is calculated to determine the most likely channel impulse response estimate which is stored in the variable h_est. As a final internal parameter the variable $burst\_start$ is used. This value, representing the sample number in r corresponding to the first bit of the transmitted burst, is used in performing the actual matched filtering.

B.1.3 Processing

The first part of `mf.m`, where the two sub-sequences are formed, takes the form

$$T16 = conj(T\_SEQ(6:21))$$
$$r\_sub = r(start\_sub : end\_sub),$$

where the parameters $start\_sub$ and end_sub are calculated according to the guard time requirements indicated above.

From these two sub-sequences a $chan\_est$ sequence is calculated. Within this sequence the channel impulse response estimate is to be found

$chan\_est = zeros(1,\ length(r\_sub) - OSR \cdot 16)$
for $n = 1 : length(chan\_est)$ **do**
  $chan\_est(n) = r\_sub(n : OSR : n + OSR \cdot 16) \cdot T16$
end for

The location of the channel impulse response estimate – within the sequence $chan\_est$ – is found by forming a power sequence based on $chan\_est$. This new sequence, $power\_est$, is evaluated using a sliding window approach using a window length of WL. Searching through this power sequence the maximum power window is located and the impulse response estimate is extracted

$WL = OSR \cdot (L + 1)$
$search = (abs(chan\_est))^2$
for $n = 1 : (length(search) - (WL - 1))$ **do**
  $power\_est(n) = sum(search(n : n + WL - 1))$
end for
$[peak,\ sync\_w] = max(power\_est)$
$h\_est = chan\_est(sync\_w : sync\_w + WL + 1)$

The next task is to synchronize the received burst, that is to say to find the first sample in r that corresponds to bit one in the transmitted burst. Recall, that the channel impulse response is found by cross correlating a received sequence with a known sequence, the training sequence. This implies that the sample number corresponding to the maximum value of h_est directly serves as an indication of the location of the first bit in $T16$ as this is located within r. Taking into account that only a sub-sequence of r has been used the sample corresponding to the first bit in r, $burst\_start$, may be derived

$[peak,\ sync\_h] = max(abs(h\_est))$
$sync\_T16 = sync\_w + sync\_h - 1$
$burst\_start = start\_sub + sync\_T16 - 1 - (OSR \cdot 16 + 1) + 1$
$burst\_start = burst\_start - 2 \cdot OSR + 1$

The first calculation of $burst\_start$ may seem unclear at first. This is mostly due to the MATLAB notation where zero cannot be used to index vectors. Hence, the plus and minus ones serve to compensate for this index problem.

The last $burst\_start$ calculation compensates for a delay inherently introduced in the transmitter as a result of the shaping operation of GMSK. As each bit is stretched over a period of 3 T_b – with it's maximum phase contribution in the last bit period – a delay of 2 T_b is expected. This corresponds to $burst\_start$ being misplaced by $2 \cdot OSR$ which then is corrected in the above code.

Having determined the channel impulse response estimate h_est and having established burst synchronization through $burst\_start$ the received burst may be matched filtered. The code responsible for the output generation is as follows

$R\_temp = xcorr(h\_est)$
$pos = (length(R\_temp) + 1)/2$
$R_{hh} = R\_temp(pos : OSR : pos + L \cdot OSR)$
$m = length(h\_est) - 1$
$r\_extended = [zeros(1, L)\ r\ zeros(1, m)]$
for $n = 1 : 148$ **do**
  $Y(n) = r\_extended(L + burst\_start + (n-1) \cdot OSR$
      $: L + burst\_start + (n-1) \cdot OSR + m) \cdot conj(h\_est)$
end for

Finally, the function returns R_{hh} and Y as outputs for the subsequent data detector, i.e. the Viterbi detector.

B.2 Viterbi Detector (MLSE)

As described previously, the Viterbi detector is implemented in two blocks. This two function concept is motivated by the fact that the setup of internal tables does not need to be done for each burst, as they can be reused. The split up is illustrated in Figure B.1.

The MLSE is interfaced by the matched filter and the channel estimator. The input to the MLSE is the matched filtered and down sampled signal, Y, along with R_{hh} and L_h which are the autocorrelation of the estimated channel impulse response and its duration measured in bit time durations, respectively. The format of R_{hh} is special, as described in the following Section B.2.1. Y is a sequence of samples with one sample for each transmitted symbol. The output of the MLSE, rx_burst, is an estimate of the most likely sequence of transmitted binary bits.

B.2.1 Input and Output

The total Viterbi detector, as constructed by the two blocks illustrated in Figure B.1, has the following inputs:

Y: A sequence of samples as they are returned from the matched filter. It is expected to be a complex valued vector, with one sample corresponding to each MSK-symbol.

R_{hh}: The autocorrelation of the channel impulse response as estimated by the `mf.m` routine. It is expected that R_{hh} is a complex valued sequence of samples represented as a column vector. Also it is expected that R_{hh} contains $L_h + 1$ samples. The layout of R_{hh} is: $R_{hh} = [R_{hh}[0],\ R_{hh}[1],\ \ldots,\ R_{hh}[L_h]]$.

L_h: The number of elements in R_{hh} minus one. This is needed due to the splitting of the function, as will emerge later.

The corresponding output is:

rx_burst: The estimated bit sequence.

The splitting of the algorithm is done so that elements independent of the received burst are only processed once. Specifically, L_h determines the state structures used by the algorithm. These states and related variables are thus setup only once.

B.2.2 Internal Data flow

In this section the interface between the two blocks constituting the Viterbi detector is described. In the following the two blocks are referred to by their MATLAB function names as `viterbi_init` and `viterbi_detector` respectively. The interface between these two functions is illustrated in Figure B.3.

As can be seen in Figure B.3 `viterbi_init` has only L_h as input. The output is used only by `viterbi_detector`. The output consists of the following variables:

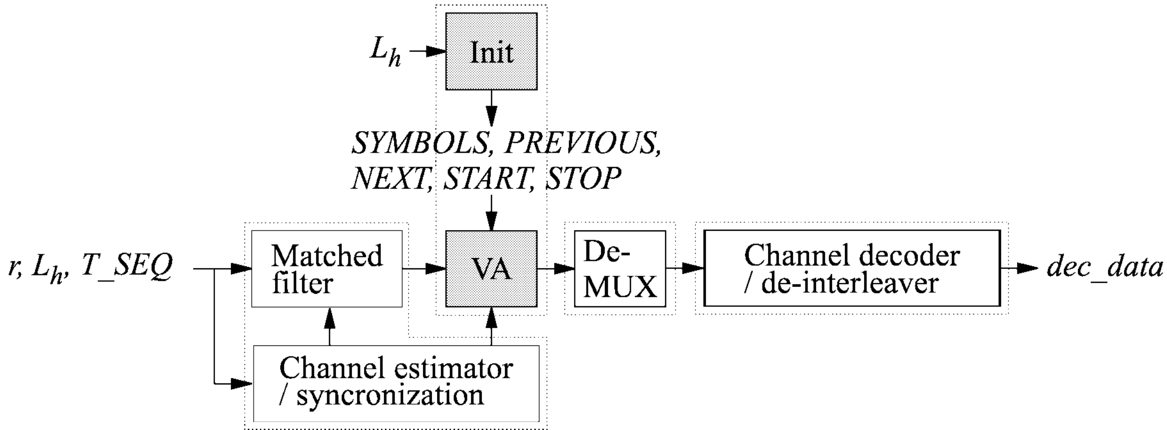


Figure B.3: Illustration of the interface between the two functions `viterbi_init` and `viterbi_detector`. The figure also illustrates the block names of the MATLAB functions. Note that the block labeled Init is equal to `viterbi_init`, and the block labeled VA is equal to `viterbi_detector`.

$SYMBOLS$: State number to MSK-symbols mapping table. Row s contains the MSK-symbols corresponding to a state. Taking n as the time reference the state is referred to as $\sigma[n]$, i.e. the state to discrete time n. Say that $\sigma[n] = 7$, then the MSK-symbols $I_{\sigma[n]=7}$ corresponding to state $\sigma[n]$ is related to $SYMBOLS$ so that $I_{\sigma[n]=7} = [I[n], I[n-1], \ldots, I[n-(L_h-1)]] =$ *SYMBOLS(7,:)*.

$PREVIOUS$: This is a state to legal previous state mapping table. The legal states, here called $LEGAL$, that may proceed state number s are obtained from $PREVIOUS$ as $LEGAL =$ *PREVIOUS(s,:)*.

$NEXT$: This is a state to legal next state mapping table. The legal states, here called $LEGAL$, that may succeed state number s are obtained from $NEXT$ as $LEGAL = NEXT(s,:)$.

$START$: The start state of the Viterbi algorithm. This is a single integer, since the start state of the Viterbi detector is uniquely determined.

$STOPS$: The set of legal stop states for the Viterbi detector. This is an array of integers, since the stop states are limited, but not always unique.

The resulting list of input variables feed to the VA block is thus

$$SYMBOLS, NEXT, PREVIOUS, START, STOPS, Y, R_{hh}$$

all of which are described above. The resulting output is rx_burst, which also is described above.

B.2.3 Processing

The setup of the pre-calculable tables and values, related to the Init block, is done by the MATLAB function `viterbi_init`. `viterbi_init` is implemented as a sequence of calls to a number of sub-functions, as described by the following piece of code

$SYMBOLS = make\_symbols(L_h)$
$PREVIOUS = make\_previous(SYMBOLS)$
$NEXT = make\_next(SYMBOLS)$
$START = make\_start(L_h, SYMBOLS)$
$STOPS = make\_stops(L_h, SYMBOLS)$

The sub-functions are described individually in the following. As described the state number to MSK-symbols translation table is to be setup. This is done using the following algorithm, which is implemented in the MATLAB function `make_symbols`

$SYMBOLS = [1; j; -1; -j]$
for $n = 1 : L_h - 1$ **do**
  $SYMBOLS =$
  $[[SYMBOLS(:, 1) \cdot j, SYMBOLS]; [SYMBOLS(:, 1) \cdot (-j), SYMBOLS]]$
end for
if $isreal(SYMBOLS(1, 1))$ **then**
  $SYMBOLS = flipud(SYMBOLS)$
end if

The if-structure ensures that state number one begins with a complex MSK-symbol. This feature is used to cut in half the number of calculations required by the Viterbi detector. From the $SYMBOLS$-table the $NEXT$-table is created by direct search using the following approach

$[states, elements] = size(SYMBOLS)$
for $this\_state = 1 : states$ **do**
  $search\_vector = SYMBOLS(this\_state, 1 : elements - 1)$
  $k = 0$
  for $search = 1 : states$ **do**
    if $search\_matrix(search, :) == search\_vector$ **then**
      $k = k + 1$
      $NEXT(this\_state, k) = search$
    end if

end for
end for

which is implemented as the MATLAB function `make_next`. Likewise the $PREVIOUS$-table is constructed as

$[states, elements] = size(SYMBOLS)$
for $this\_state = 1 : states$ **do**
$search\_vector = SYMBOLS(this\_state, 2 : elements)$
$k = 0$
for $search = 1 : states$ **do**
if $search\_matrix(search, :) == search\_vector$ **then**
$k = k + 1$
$PREVIOUS(this\_state, k) = search$
end if
end for
end for

The above is implemented as the MATLAB function `make_previous`. As previously noted the state number corresponding to the start state is determined at runtime. This is done using the MSK-representation of the start state which is shown in Table B.1 [15].

L_h	$\sigma[0]$
2	$[1,\ -j]$
3	$[1,\ -j,\ -1]$
4	$[1,\ -j,\ -1,\ j]$

Table B.1: The MSK-representation of the legal start states of the Viterbi detector.

The returned value is stored in the variable $START$, and is determined using the following strategy

if $L_h == 2$ **then**
$start\_symbols = [1,\ -j]$
else if $L_h == 3$ **then**
$start\_symbols = [1,\ -j,\ -1]$
else if $L_h == 4$ **then**
$start\_symbols = [1,\ -j,\ -1,\ j]$
end if
$START = 0$
while $START\_NOT\_FOUND$ **do**
$START = START + 1$

if $SYMBOLS(START,:) == start\_symbols)$ **then**
$START\_NOT\_FOUND = 0$
end if
end while

The location of the integer corresponding to the start state is handled by the MATLAB function `make_start`. The stop state of the Viterbi detector is not always uniquely defined but is always contained within a limited set. The legal stop states are listed in Table B.2 for the considered values of L_h [15].

L_h	$\sigma[148]$
2	$\{[-1,\ j]\}$
3	$\{[-1,\ j,\ 1]\}$
4	$\{[-1,\ j,\ 1,\ j], [-1,\ j,\ 1,\ -j]\}$

Table B.2: The MSK-representation of the legal stop states of the Viterbi detector.

From Table B.2 the state numbers corresponding to the stop states are stored in $STOPS$ using the following method

if $L_h == 2$ **then**
$stop\_symbols = [-1,\ j]$
$count = 1$
else if $L_h == 3$ **then**
$stop\_symbols = [-1,\ j,\ 1]$
$count = 1$
else if $L_h == 4$ **then**
$stop\_symbols = [[-1,\ j,\ 1,\ j];][-1,\ j,\ 1,\ -j]$
$count = 2$
end if
$index = 0$
$stops\_found = 0$
while $stops\_found < count$ **do**
$index = index + 1$
if $SYMBOLS(index,:) == stop\_symbols(stops\_found + 1,:)$ **then**
$stops\_found = stops\_found + 1$
$STOPS(stops\_found) = index$
end if
end while

This is implemented in MATLAB by the function called `make_stops`. This concludes the description of the retrieval of the variables returned from `viterbi_init`. In the following viterbi_detector, which contain the actual implementation of the Viterbi detector, is described. Unlike viterbi_init, viterbi_detector is run for all bursts and, thus, it is implemented as a single function. This is done in order to avoid the overhead associated with a function call.

It should be clear that the Viterbi detector identifies the most probable path trough a state trellis. The trellis involves as many state shifts as the number of MSK-symbols in a GSM-burst, which equals 148. This is also the number of elements in Y. The assumption which form the basis for the algorithm is that the metric of a state to the time n can be calculated from

- The $n-1$'th state and its associated metric.
- The n'th present state.
- The metric of the n'th element in Y.

Referring to the previous state as p and to the present state as s then the metric of state s is expressed as

$$METRIC(s,n) = \max_p \{Value(n-1,p) + Gain(Y_n, s, p)\}, \tag{B.1}$$

which implies that the metric of state number s, to the time n, is found by choosing the previous state number p so that the metric is maximized. It is here chosen to assign the initial state the metric value 0. The initial state is referred to as state number 0. Based on this all that needs to be done is to calculate the gain from state to state. The gain is calculated using

$$Gain(Y[n], s_s, s_p) = \Re\{I^*[n]y[n]\} - \Re\left\{I^*[n]\sum_{m=n-L_h}^{n-1} I[m]R_{hh}[n-m]\right\}, \tag{B.2}$$

as has been presented earlier. With the variable definitions above, and introducing MATLAB notation, this becomes

$$\begin{aligned} Gain(Y(n), s, p) &= \Re\{SYMBOLS(s,1)^* \cdot Y(n)\} \\ &\quad -\Re\{SYMBOLS(s,1)^* \cdot SYMBOLS(p,:) \cdot R_{hh}\}, \end{aligned} \tag{B.3}$$

Note that the last part of (B.3) is independent of Y, but rather depends on the previous symbols. Thus the same calculations have to be done each time the algorithm considers a shift from state a to state b. Since this is done approximately seventy times per burst, a considerable speedup can be expected from pre-calculating that last part of (B.3). Thus, before starting the actual VA algorithm viterbi_detector internally does pre-calculation of a table called $INCREMENT$ which represents these values. The layout of increment is so that $INCREMENT(a,b) = \Re\{SYMBOLS(b,1)^* \cdot SYMBOLS(a,:) \cdot R_{hh}\}$ represents the precalculable increment when moving from state a to state b. The pseudo code for setting up this table is

$[M, L_h] = size(SYMBOLS)$
for $n = 1 : M$ **do**
  $m = NEXT(n, 1)$
  $INCREMENT(n, m) = \Re(SYMBOLS(m, 1)^* \cdot SYMBOLS(n, :) \cdot R_{hh}(2 : L_h + 1))$
  $m = NEXT(n, 2)$
  $INCREMENT(n, m) = \Re(SYMBOLS(m, 1)^* \cdot SYMBOLS(n, :) \cdot R_{hh}(2 : L_h + 1))$
end for

Having established a method for calculating the metric of the states it is possible to find the metrics of the final states. This is done by starting at the predefined start state, and then calculate the gains associated with all 148 state shifts, while summing up the gain values of each path. However, as described in connection with (B.1), arriving at a state requires a choice between two candidates. Thus, for all states it is important to save information of which state is the chosen previous state. The chosen previous state is also referred to as the survivor. In the present implementation this done in a table. Since 148 state shifts, and M states exist the table is M times 148 elements big. The state that leads to state s at the time n is stored in $SURVIVOR(s, n)$. However, since the start state of the algorithm is bounded, initialization is required. As can be seen from Table B.1 it takes L_h symbols to remove the effect of the constraint introduced by the start state. This initialization is described by the following piece of pseudo code

$PS = START$
$S = NEXT(START, 1)$
$METRIC(S, 1) = Gain(Y(n), S, PS)$
$SURVIVOR(S, 1) = START$
$S = NEXT(START, 2)$
$METRIC(S, 1) = Gain(Y(n), S, PS)$
$SURVIVOR(S, 1) = START$
$COMPLEX = 0$
for $N = 2 : L_h$ **do**
  if $COMPLEX$ **then**
    $COMPLEX = 0$
  else
    $COMPLEX = 1$
  end if
  $STATE\_CNTR = 0$
  for $PS = PREVIOUS\_STATES$ **do**
    $STATE\_CNTR = STATE\_CNTR + 1$
    $S = NEXT(PS, 1)$
    $METRIC(S, N) = METRIC(PS, N - 1) + Gain(Y(n), S, PS)$
    $SURVIVOR(S, N) = PS$
    $USED(STATE\_CNTR) = S$
    $STATE\_CNTR = STATE\_CNTR + 1$
    $S = NEXT(PS, 2)$

$METRIC(S,N) = METRIC(PS,N-1) + Gain(Y(n),S,PS)$
$SURVIVOR(S,N) = PS$
$USED(STATE\_CNTR) = S$
end for
$PREVIOUS\_STATES = USED$
end for

Having initialized the algorithm the remainder of the states are processed using the following technique

$PROCESSED = L_h$
if not $COMPLEX$ **then**
$COMPLEX = 0$
$PROCESSED = PROCESSED + 1$
$N = PROCESSED$
for $S = 2:2:M$ **do**
$PS = PREVIOUS(S,1)$
$M1 = METRIC(PS,N-1) + Gain(Y(n),S,PS)$
$PS = PREVIOUS(S,2)$
$M2 = METRIC(PS,N-1) + Gain(Y(n),S,PS)$
if $M1 > M2$ **then**
$METRIC(S,N) = M1$
$SURVIVOR(S,N) = PREVIOUS(S,1)$
else
$METRIC(S,N) = M2$
$SURVIVOR(S,N) = PREVIOUS(S,2)$
end if
end for
end if
$N = PROCESSED + 1$
while $N < length(Y)$ **do**
for $S = 1:2:M-1$ **do**
$PS = PREVIOUS(S,1)$
$M1 = METRIC(PS,N-1) + Gain(Y(n),S,PS)$
$PS = PREVIOUS(S,2)$
$M2 = METRIC(PS,N-1) + Gain(Y(n),S,PS)$
if $M1 > M2$ **then**
$METRIC(S,N) = M1$
$SURVIVOR(S,N) = PREVIOUS(S,1)$
else
$METRIC(S,N) = M2$
$SURVIVOR(S,N) = PREVIOUS(S,2)$
end if

end for
$N = N + 1$
for $S = 2 : 2 : M$ **do**
$PS = PREVIOUS(S, 1)$
$M1 = METRIC(PS, N - 1) + Gain(Y(n), S, PS)$
$PS = PREVIOUS(S, 2)$
$M2 = METRIC(PS, N - 1) + Gain(Y(n), S, PS)$
if $M1 > M2$ **then**
$METRIC(S, N) = M1$
$SURVIVOR(S, N) = PREVIOUS(S, 1)$
else
$METRIC(S, N) = M2$
$SURVIVOR(S, N) = PREVIOUS(S, 2)$
end if
end for
$N = N + 1$
end while

Note in the above, that the first **if** structure, which ensures that an equal number of states remains for the while loop to process.

Now the remaining task is to identify the received symbols. This involves determining the received sequence of MSK-symbols. The algorithm used for that task is

$BEST\_LEGAL = 0$
for $FINAL = STOPS$ **do**
if $METRIC(FINAL, STEPS) > BEST\_LEGAL$ **then**
$BEST\_LEGAL = METRIC(FINAL, 148)$
$S = FINAL$
end if
end for
$IEST(STEPS) = SYMBOLS(S, 1)$
$N = STEPS - 1$
while $N > 0$ **do**
$S = SURVIVOR(S, N + 1)$
$IEST(N) = SYMBOLS(S, 1)$
$N = N - 1$
end while

Finally, the MSK-symbols are to be translated to a sequence of binary data bits and returned in rx_burst. To do this the following is employed

$rx\_burst(1) = IEST(1)/(j \cdot 1 \cdot 1)$

for $n = 2 : STEPS$ **do**
  $rx\_burst(n) = IEST(n)/(j \cdot rx\_burst(n-1) \cdot IEST(n-1))$
end for
$rx\_burst = (rx\_burst + 1)./2$

This concludes the description of the `viterbi_detector`, and thus of the implementation of the Viterbi detector.

In summary it is repeated that the present implementation is made up by two functions, namely `viterbi_init` and `viterbi_detector`. The job of `viterbi_init` is to setup translation and transition tables, along with other information, for use by `viterbi_detector`. `viterbi_detector` handles all the processing of the received data.

B.3 De-multiplexer

The de-multiplexer is simple in its implementation, since all that needs to be done is to extract to sub-vectors directly from a burst and then return these two vectors as a single continued vector.

B.3.1 Input, Output and Processing

The input to the function is:

rx_burst:	The estimated bit sequence, in the same format as it is returned by the function `viterbi_detector.m`.

and the corresponding output is:

rx_data:	The de-multiplexed data bits.

The de-multiplexing is implemented by a single line of MATLAB code:

$rx\_data = [rx\_burst(4 : 60), rx\_burst(89 : 145)]$

B.4 De-Interleaver

As described in Section 3.3.2 the de-interleaver is implemented in the MATLAB function called `deinterleave.m`. The purpose of the de-interleaver is to reorder the bits which was initially shuffeled by the interleaver. As is the case with the interleaver, it is possible to implement the de-interleaver in a simple manner, resulting in a low computational burden at runtime.

B.4.1 Input, Output, and Processing

The de-interleaver takes eight instances of rx_data as its input:

rx_data_matrix: A matrix containg eight instances of rx_data. Each instance is aligned in a row. The data are arranged so that the eldest instance of rx_data is kept in row number one, and the latest arrived instance is kept in row number eight.

The output from the de-interleaver is:

rx_enc: The received code block, as reconstructed from the eight instances of rx_data.

The de-interleaver is externally aided by a queue, which administrates the propper alignment of the rx_data instances. Internally the de-interleaver simply perform a number of copy operations as described by the formulas in (3.21) and (3.22) in Section 3.3.2. The file `deinterleave.m` is constructed using the following lines

$BitsInBlock = 455$
$out = fopen('deinterleave.tmp', 'w')$
$B = 0$
for $b = 0 : BitsInBlock$ **do**
  $R = 4 * B + mod(b, 8)$
  $r = 2 * mod((49 * b), 57) + floor(mod(b, 8)/4)$
  $fprintf(out, 'rx\_enc(\%d) = rx\_data\_matrix(\%d, \%d); \backslash n', b + 1, R + 1, r + 1)$
end for
$fclose(out)$

Implementing the de-interleaver in this way, with pre calculated indexes, proves to be much faster than when the indexes are calculated at runtime.

B.5 Channel Decoder

The channel decoder operation is implemented by the MATLAB function `channel_dec.m`. The task of this function is to implement the outer decoding required for use in the GSM system.

B.5.1 Input and Output

The channel decoder makes use of the following input parameter

rx_enc:	A 456 bits long vector contaning the encoded data sequence as estimated by the SOVA. The format of the sequency must be according to the GSM 05.03 encoding scheme

The corresponding output from `channel_enc.m` is

rx_block:	The resulting 260 bits long vector decoded data sequence.
$FLAG\_SS$:	Error flag. Due to the structure of the encoding scheme the decoder should end in the same state as it starts of in. If this is not the case the decoded output contains errors. If an error has occured $FLAG\_SS$ is set to '1'.
$PARITY\_CHK$:	Estimate of the 3 parity check bit inserted in the transmitter.

B.5.2 Internal Data Flow

Besides from the above mentioned information carrying parameters the channel decoder also operates using some internal information.

As is the case in the channel encoder two levels of bits are dealt with. The separation into class I, $c1$, and class II bits, $c2$, is nessesary as only the class I bits are encoded.

Furthermore a number of matrices and vectors are generated to help keep track of the different paths in the state trellis and the corresponding metrics. These variables are termed $STATE$ and $METRIC$, respectively. Also, to destinguish between legal and illegal state transitions two matrices, $NEXT$ and $PREVIOUS$, are set up to determine wich two states a given state may switch to next and what states that are allowed to lead to a given state, respectively.

In order to enable the calculation of the metric a matrix, $DIBIT$, is set up. When, in the channel encoder, a transition from one state to another state occurs two bits, here referred to as dibits are output. Which one of the four possible dibit combinations that are output for a given transition is stored in the $DIBIT$ matrix. In close relation to this matrix a BIT matrix is also required. The structure of BIT is just as that of the $DIBIT$ matrix only here the content is the one bit binary input that is required for a given state transition. Hence, the BIT matrix is used in mapping state transition information to actual binary – and decoded – information.

B.5.3 Processing

First the input, rx_enc, is split into the different classes and the various internal variables are initialized

$c1 = rx\_enc(1:378)$
$c2 = rx\_enc(379:456)$

$START\_STATE = 1$
$END\_STATE = 1$

$STATE = zeros(16, 189)$
$METRIC = zeros(16, 2)$

$NEXT = zeros(16, 2)$
$zeroin = 1$
$onein = 9$
for $n = 1:2:15$ **do**
  $NEXT(n, :) = [zeroin\ onein]$
  $NEXT(n+1, :) = NEXT(n, :)$
  $zeroin = zeroin + 1$
  $onein = onein + 1$
end for

$PREVIOUS = zeros(16, 2)$
$offset = 0$
for $n = 1:8$ **do**
  $PREVIOUS(n, :) = [n + offset\ n + offset + 1]$
  $offset = offset + 1$
end for
$PREVIOUS = [PREVIOUS(1:8, :); PREVIOUS(1:8, :)]$

Having split the data the $c1$ bits are decoded using the Viterbi algorithm. check bits. To reduce the number of calculations the run of the Viterbi is split into to parts. The first part is a run in where only the known legal next states are used in the metric calculations. This is run for 4 strate transitions. From that point on all states in the state trellis are in use and the previous legal states are used in stead.

$VISITED\_STATES = START\_STATE$
for $n = 0:3$ **do**
  $rx\_DIBITXy = c1(2*n+1)$
  $rx\_DIBITxY = c1(2*n+1+1)$
  for $k = 1 : length(VISITED\_STATES)$ **do**

$PRESENT\_STATE = VISITED\_STATES(k)$
$next\_state_0 = NEXT(PRESENT\_STATE, 1)$
$next\_state_1 = NEXT(PRESENT\_STATE, 2)$
$symbol\_0 = DIBIT(PRESENT\_STATE, next\_state\_0)$
$symbol\_1 = DIBIT(PRESENT\_STATE, next\_state\_1)$

if $symbol\_0 == 0$ **then**
$LAMBDA = xor(rx\_DIBITXy, 0) + xor(rx\_DIBITxY, 0)$
end if
if $symbol\_0 == 1$ **then**
$LAMBDA = xor(rx\_DIBITXy, 0) + xor(rx\_DIBITxY, 1)$
end if
if $symbol\_0 == 2$ **then**
$LAMBDA = xor(rx\_DIBITXy, 1) + xor(rx\_DIBITxY, 0)$
end if
if $symbol\_0 == 3$ **then**
$LAMBDA = xor(rx\_DIBITXy, 1) + xor(rx\_DIBITxY, 1)$
end if

$METRIC(next\_state\_0, 2) = METRIC(PRESENT\_STATE, 1) + LAMBDA$

if $symbol\_1 == 0$ **then**
$LAMBDA = xor(rx\_DIBITXy, 0) + xor(rx\_DIBITxY, 0)$
end if
if $symbol\_1 == 1$ **then**
$LAMBDA = xor(rx\_DIBITXy, 0) + xor(rx\_DIBITxY, 1)$
end if
if $symbol\_1 == 2$ **then**
$LAMBDA = xor(rx\_DIBITXy, 1) + xor(rx\_DIBITxY, 0)$
end if
if $symbol\_1 == 3$ **then**
$LAMBDA = xor(rx\_DIBITXy, 1) + xor(rx\_DIBITxY, 1)$
end if

$METRIC(next\_state\_1, 2) = METRIC(PRESENT\_STATE, 1) + LAMBDA$
$STATE([next\_state\_0, next\_state\_1], n + 1) = PRESENT\_STATE$

if $k == 1$ **then**
$PROCESSED = [next\_state\_0\; next\_state\_1]$
else
$PROCESSED = [PROCESSED\; next\_state\_0\; next\_state\_1]$
end if
end for

$VISITED\_STATES = PROCESSED$
$METRIC(:,1) = METRIC(:,2)$
$METRIC(:,2) = 0$
end for

Having completed the run in process of the Viterbi algorithm all 16 states are now considered using the $PREVIOUS$ table

for $n = 4 : 188$ **do**
$rx\_DIBITXy = c1(2 * n + 1)$
$rx\_DIBITxY = c1(2 * n + 1 + 1)$
for $k = 1 : 16$ **do**
$prev\_state\_1 = PREVIOUS(k, 1)$
$prev\_state\_2 = PREVIOUS(k, 2)$
$symbol\_1 = DIBIT(prev\_state\_1, k)$
$symbol\_2 = DIBIT(prev\_state\_2, k)$

if $symbol\_0 == 0$ **then**
$LAMBDA = xor(rx\_DIBITXy, 0) + xor(rx\_DIBITxY, 0)$
end if
if $symbol\_0 == 1$ **then**
$LAMBDA = xor(rx\_DIBITXy, 0) + xor(rx\_DIBITxY, 1)$
end if
if $symbol\_0 == 2$ **then**
$LAMBDA = xor(rx\_DIBITXy, 1) + xor(rx\_DIBITxY, 0)$
end if
if $symbol\_0 == 3$ **then**
$LAMBDA = xor(rx\_DIBITXy, 1) + xor(rx\_DIBITxY, 1)$
end if

if $symbol\_1 == 0$ **then**
$LAMBDA = xor(rx\_DIBITXy, 0) + xor(rx\_DIBITxY, 0)$
end if
if $symbol\_1 == 1$ **then**
$LAMBDA = xor(rx\_DIBITXy, 0) + xor(rx\_DIBITxY, 1)$
end if
if $symbol\_1 == 2$ **then**
$LAMBDA = xor(rx\_DIBITXy, 1) + xor(rx\_DIBITxY, 0)$
end if
if $symbol\_1 == 3$ **then**
$LAMBDA = xor(rx\_DIBITXy, 1) + xor(rx\_DIBITxY, 1)$
end if

$METRIC\_1 = METRIC(prev\_state\_1, 1) + LAMBDA\_1$
$METRIC\_2 = METRIC(prev\_state\_2, 1) + LAMBDA\_2$

if $METRIC\_1 < METRIC\_2$ **then**
$METRIC(k, 2) = METRIC\_1$
$STATE(k, n + 1) = prev\_state\_1$
else
$METRIC(k, 2) = METRIC\_2$
$STATE(k, n + 1) = prev\_state\_2$
end if
end for

$METRIC(:, 1) = METRIC(:, 2)$
$METRIC(:, 2) = 0$
end for

Having build the state transition trellis finding the most probable sequence of states is now a matter of backtracking through the trellis. This gives the state transition sequence that then is mapped to binary information which when combined with the class II bits gives the final decoded information signal.

$STATE\_SEQ = zeros(1, 189)$
$[STOP\_METRIC, STOP\_STATE] = min(METRIC(:, 1))$
$STATE\_SEQ(189) = STOP\_STATE$
for $n = 188 : -1 : 1$ **do**
$STATE\_SEQ(n) = STATE(STATE\_SEQ(n + 1), n + 1)$
end for
$STATE\_SEQ = [START\_STATE\ STATE\_SEQ]$

for $n = 1 : length(STATE\_SEQ) - 1$ **do**
$DECONV\_DATA(n) = BIT(STATE\_SEQ(n), STATE\_SEQ(n + 1))$
end for

$DATA\_Ia = DECONV\_DATA(1 : 50)$
$PARITY\_CHK = DECONV\_DATA(51 : 53)$
$DATA\_Ib = DECONV\_DATA(54 : 185)$
$TAIL\_BITS = DECONV\_DATA(186 : 189)$

$rx\_block = [DATA\_Ia\ DATA\_Ib\ c2]$

C

Source code

This Chapter contains the source code for *GSMsim*. The more simple code, such as the one used for generating the illustrations in the present work, is not included.

Source code C.1: `burst_g.m`

```
function tx_burst = burst_g(tx_data, TRAINING)
%
% burst_g:  This function generates a bit sequence representing
%           a general GSM information burst. Included are tail
%           and ctrl bits, data bits and a training sequence.
%
%           The GSM burst contains a total of 148 bits accounted
%           for in the following burststructure (GSM 05.05)
%
%           | TAIL   DATA   CTRL | TRAINING   CTRL | DATA   TAIL |
%           |  3      57     1   |    26       1   |  57     3   |
%
%           [TAIL]    = [000]
%           [CTRL]    = [0] or [1] here [1]
%           [TRAINING] is passed to the function
%
% SYNTAX:   tx_burst = burst_g(tx_data, TRAINING)
%
% INPUT:    tx_data:  The burst data.
%           TRAINING: The training sequence which is to be used.
%
% OUTPUT:   tx_burst: A complete 148 bits long GSM normal burst binary
%                     sequence
%
% GLOBAL:
%
% SUB FUNC: None
%
% WARNINGS: None
%
% TEST(S):  Function tested
%
% AUTHOR:   Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: burst_g.m,v 1.6 1997/12/17 15:32:23 aneks Exp $

TAIL = [0 0 0];
CTRL = [1];

% COMBINE THE BURST BIT SEQUENCE
%
tx_burst = [TAIL tx_data(1:57) CTRL TRAINING CTRL tx_data(58:114) TAIL];
```

Source code C.2: `data_gen.m`

```
function [tx_data] = data_gen(INIT_L);
%
% data_gen:   This function generates a block of random data bits.
%
% SYNTAX:     [tx_data] = data_gen(INIT_L)
%
% INPUT:      INIT_L:  The length of the generated data vector.
%
% OUTPUT:     tx_data: An element vector contaning the random data
%                      sequence of length INIT_L. INIT_L is a variable
%                      set by gsm_set.
%
% SUB FUNC:   None
%
% WARNINGS:   None
%
% TEST(S):    Function tested.
%
% AUTOR:      Arne Norre Ekstrøm / Jan H. Mikkelsen
% EMAIL:      aneks@kom.auc.dk / hmi@kom.auc.dk
%
% $Id: data_gen.m,v 1.6 1997/09/22 11:46:29 aneks Exp $
%

% GENERATE init_l RANDOM BITS. FUNCTION IS BASED ON THAT RAND RETURNS
% UNIFORMLY DISTRIBUTED DATA IN THE INTERVAL ] 0.0 ; 1.0 [.
%
tx_data = round(rand(1,INIT_L));
```

Source code C.3: channel_enc.m

```
function [tx_enc] = chan_enc(tx_block)
%
% chan_enc:   This function accepts a 260 bits long vector contaning the
%             data sequence intended for transmission. The length of
%             the vector is expanded by channel encoding to form a data
%             block with a length of 456 bits as required in a normal
%             GSM burst.
%
%             | Class I    Class II |
%             [   182        78    ]
%
%             [ Class Ia   Class Ib | Class II ]
%             [    50         132   |    78    ]
%
%             The Class Ia bits are separatly parity encoded whereby 3 error
%             control bits are added. Subsequently, the Class Ia bits are
%             combined with the Class Ib bits for convolutional encoding
%             according to GSM 05.03. The Class II bits are left unprotected
%
% SYNTAX:     [tx_enc] = channel_enc(tx_block)
%
% INPUT:      tx_block   A 260 bits long vector contaning the data sequence
%                        intended for transmission.
%
% OUTPUT:     tx_enc   A 456 bits long vector contaning the encoded data
%                      sequence
%
% SUB_FUNC:   None
%
% WARNINGS:   None
%
% TEST(S):    Parity encoding - tested to operate correctly.
%             Convolution encoding - tested to operate correctly.
%
% AUTHOR:   Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: channel_enc.m,v 1.9 1998/02/12 10:48:31 aneks Exp $

L = length(tx_block);

% INPUT CHECK
%
if L ~= 260
  disp(' ')
  disp('Input data sequence size violation. Program terminated.')
  disp(' ')
  break;
end

% SEPARATE INPUT IN RESPECTIVE CLASSES
%
c1a = tx_block(1:50);
c1b = tx_block(51:182);
c2 = tx_block(183:260);

% PARITY ENCODING. THREE CHECK BITS ARE ADDED
%
g = [1 0 1 1];
d = [c1a 0 0 0];
[q,r] = deconv(d,g);

% ADJUST RESULT TO BINARY REPRESENTATION
%
L = length(r);
out = abs(r(L-2:L));
for n = 1:length(out),
  if ceil(out(n)/2) ~= floor(out(n)/2)
    out(n) = 1;
  else
    out(n) = 0;
  end
end

c1a = [c1a out];

% CLASS I BITS ARE COMBINED AND 4 TAIL BITS, {0000}, ARE ADDED AS
% PRESCRIBED IN THE GSM RECOMMENDATION 05.03
%
c1 = [c1a c1b 0 0 0 0];

% CONVOLUTIONAL ENCODING OF THE RANDOM DATA BITS. THE ENCODING IS
% ACCORDING TO GSM 05.05
%
register = zeros(1,4);
data_seq = [register c1];
enc_a = zeros(1,189);
enc_b = zeros(1,189);
encoded = zeros(1,378);

for n=1:189,
  enc_a(n) = xor( xor( data_seq(n+4),data_seq(n+1) ), data_seq(n) );
  enc_temp = xor( data_seq(n+4),data_seq(n+3) );
  enc_b(n) = xor ( xor( enc_temp, data_seq(n+1) ), data_seq(n));
  encoded(2*n-1) = enc_a(n);
  encoded(2*n) = enc_b(n);
  clear enc_temp
end

% PREPARE DATA FOR OUTPUT
%
tx_enc = [encoded c2];
```

Source code C.4: `interleave.m`

```
function [ tx_data_matrix ] = interleave(tx_enc0,tx_enc1)
%
% interleave:
%           This function performs interleaving of two information
%           blocks, each containing 456 bits of information. Output
%           is an matrix with 4 rows, each containing 114 bits of
%           information for inclusion in an GSM burst.
%
% SYNTAX:   [ tx_data_matrix ] = interleave(tx_enc0,tx_enc1)
%
% INPUT:    tx_enc0:
%                 The first block in an interleaving pass.
%           tx_enc1:
%                 The second block in an interleaving pass.
%
% OUTPUT:   tx_data_matrix:
%                 A matrix containing 114 bits of data in each row,
%                 ready to be split into two and passed to burst_g.m
%
% WARNINGS: Not all 2 x 456 bits are represented in the output, this is
%           exactely as specified in the recommadations.
%
% TEST(S):  interleave -> deinterleave = 0 Errors.
%
% AUTHOR:   Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: interleave.m,v 1.4 1997/11/20 11:10:42 aneks Exp $
tx_data_matrix(1,1)=tx_enc1(1);
tx_data_matrix(1,2)=tx_enc0(229);
tx_data_matrix(1,3)=tx_enc1(65);
tx_data_matrix(1,4)=tx_enc0(293);
tx_data_matrix(1,5)=tx_enc1(129);
tx_data_matrix(1,6)=tx_enc0(357);
tx_data_matrix(1,7)=tx_enc1(193);
tx_data_matrix(1,8)=tx_enc0(421);
tx_data_matrix(1,9)=tx_enc1(257);
tx_data_matrix(1,10)=tx_enc0(29);
tx_data_matrix(1,11)=tx_enc1(321);
tx_data_matrix(1,12)=tx_enc0(93);
tx_data_matrix(1,13)=tx_enc1(385);
tx_data_matrix(1,14)=tx_enc0(157);
tx_data_matrix(1,15)=tx_enc1(449);
tx_data_matrix(1,16)=tx_enc0(221);
tx_data_matrix(1,17)=tx_enc1(57);
tx_data_matrix(1,18)=tx_enc0(285);
tx_data_matrix(1,19)=tx_enc1(121);
tx_data_matrix(1,20)=tx_enc0(349);
tx_data_matrix(1,21)=tx_enc1(185);
tx_data_matrix(1,22)=tx_enc0(413);
tx_data_matrix(1,23)=tx_enc1(249);
tx_data_matrix(1,24)=tx_enc0(21);
tx_data_matrix(1,25)=tx_enc1(313);
tx_data_matrix(1,26)=tx_enc0(85);
tx_data_matrix(1,27)=tx_enc1(377);
tx_data_matrix(1,28)=tx_enc0(149);
tx_data_matrix(1,29)=tx_enc1(441);
tx_data_matrix(1,30)=tx_enc0(213);
tx_data_matrix(1,31)=tx_enc1(49);
tx_data_matrix(1,32)=tx_enc0(277);
tx_data_matrix(1,33)=tx_enc1(113);
tx_data_matrix(1,34)=tx_enc0(341);
tx_data_matrix(1,35)=tx_enc1(177);
tx_data_matrix(1,36)=tx_enc0(405);
tx_data_matrix(1,37)=tx_enc1(241);
tx_data_matrix(1,38)=tx_enc0(13);
tx_data_matrix(1,39)=tx_enc1(305);
tx_data_matrix(1,40)=tx_enc0(77);
tx_data_matrix(1,41)=tx_enc1(369);
tx_data_matrix(1,42)=tx_enc0(141);
tx_data_matrix(1,43)=tx_enc1(433);
tx_data_matrix(1,44)=tx_enc0(205);
tx_data_matrix(1,45)=tx_enc1(41);
tx_data_matrix(1,46)=tx_enc0(269);
tx_data_matrix(1,47)=tx_enc1(105);
tx_data_matrix(1,48)=tx_enc0(333);
tx_data_matrix(1,49)=tx_enc1(169);
tx_data_matrix(1,50)=tx_enc0(397);
tx_data_matrix(1,51)=tx_enc1(233);
tx_data_matrix(1,52)=tx_enc0(5);
tx_data_matrix(1,53)=tx_enc1(297);
tx_data_matrix(1,54)=tx_enc0(69);
tx_data_matrix(1,55)=tx_enc1(361);
tx_data_matrix(1,56)=tx_enc0(133);
tx_data_matrix(1,57)=tx_enc1(425);
tx_data_matrix(1,58)=tx_enc0(197);
tx_data_matrix(1,59)=tx_enc1(33);
tx_data_matrix(1,60)=tx_enc0(261);
tx_data_matrix(1,61)=tx_enc1(97);
tx_data_matrix(1,62)=tx_enc0(325);
tx_data_matrix(1,63)=tx_enc1(161);
tx_data_matrix(1,64)=tx_enc0(389);
tx_data_matrix(1,65)=tx_enc1(225);
tx_data_matrix(1,66)=tx_enc0(453);
tx_data_matrix(1,67)=tx_enc1(289);
tx_data_matrix(1,68)=tx_enc0(61);
tx_data_matrix(1,69)=tx_enc1(353);
tx_data_matrix(1,70)=tx_enc0(125);
tx_data_matrix(1,71)=tx_enc1(417);
tx_data_matrix(1,72)=tx_enc0(189);
tx_data_matrix(1,73)=tx_enc1(25);
tx_data_matrix(1,74)=tx_enc0(253);
tx_data_matrix(1,75)=tx_enc1(89);
tx_data_matrix(1,76)=tx_enc0(317);
tx_data_matrix(1,77)=tx_enc1(153);
tx_data_matrix(1,78)=tx_enc0(381);
tx_data_matrix(1,79)=tx_enc1(217);
tx_data_matrix(1,80)=tx_enc0(445);
tx_data_matrix(1,81)=tx_enc1(281);
tx_data_matrix(1,82)=tx_enc0(53);
tx_data_matrix(1,83)=tx_enc1(345);
tx_data_matrix(1,84)=tx_enc0(117);
tx_data_matrix(1,85)=tx_enc1(409);
tx_data_matrix(1,86)=tx_enc0(181);
tx_data_matrix(1,87)=tx_enc1(17);
tx_data_matrix(1,88)=tx_enc0(245);
tx_data_matrix(1,89)=tx_enc1(81);
tx_data_matrix(1,90)=tx_enc0(309);
tx_data_matrix(1,91)=tx_enc1(145);
tx_data_matrix(1,92)=tx_enc0(373);
tx_data_matrix(1,93)=tx_enc1(209);
tx_data_matrix(1,94)=tx_enc0(437);
tx_data_matrix(1,95)=tx_enc1(273);
tx_data_matrix(1,96)=tx_enc0(45);
```

```
tx_data_matrix(1,97)=tx_enc1(337);
tx_data_matrix(1,98)=tx_enc0(109);
tx_data_matrix(1,99)=tx_enc1(401);
tx_data_matrix(1,100)=tx_enc0(173);
tx_data_matrix(1,101)=tx_enc1(9);
tx_data_matrix(1,102)=tx_enc0(237);
tx_data_matrix(1,103)=tx_enc1(73);
tx_data_matrix(1,104)=tx_enc0(301);
tx_data_matrix(1,105)=tx_enc1(137);
tx_data_matrix(1,106)=tx_enc0(365);
tx_data_matrix(1,107)=tx_enc1(201);
tx_data_matrix(1,108)=tx_enc0(429);
tx_data_matrix(1,109)=tx_enc1(265);
tx_data_matrix(1,110)=tx_enc0(37);
tx_data_matrix(1,111)=tx_enc1(329);
tx_data_matrix(1,112)=tx_enc0(101);
tx_data_matrix(1,113)=tx_enc1(393);
tx_data_matrix(1,114)=tx_enc0(165);
tx_data_matrix(2,1)=tx_enc1(58);
tx_data_matrix(2,2)=tx_enc0(286);
tx_data_matrix(2,3)=tx_enc1(122);
tx_data_matrix(2,4)=tx_enc0(350);
tx_data_matrix(2,5)=tx_enc1(186);
tx_data_matrix(2,6)=tx_enc0(414);
tx_data_matrix(2,7)=tx_enc1(250);
tx_data_matrix(2,8)=tx_enc0(22);
tx_data_matrix(2,9)=tx_enc1(314);
tx_data_matrix(2,10)=tx_enc0(86);
tx_data_matrix(2,11)=tx_enc1(378);
tx_data_matrix(2,12)=tx_enc0(150);
tx_data_matrix(2,13)=tx_enc1(442);
tx_data_matrix(2,14)=tx_enc0(214);
tx_data_matrix(2,15)=tx_enc1(50);
tx_data_matrix(2,16)=tx_enc0(278);
tx_data_matrix(2,17)=tx_enc1(114);
tx_data_matrix(2,18)=tx_enc0(342);
tx_data_matrix(2,19)=tx_enc1(178);
tx_data_matrix(2,20)=tx_enc0(406);
tx_data_matrix(2,21)=tx_enc1(242);
tx_data_matrix(2,22)=tx_enc0(14);
tx_data_matrix(2,23)=tx_enc1(306);
tx_data_matrix(2,24)=tx_enc0(78);
tx_data_matrix(2,25)=tx_enc1(370);
tx_data_matrix(2,26)=tx_enc0(142);
tx_data_matrix(2,27)=tx_enc1(434);
tx_data_matrix(2,28)=tx_enc0(206);
tx_data_matrix(2,29)=tx_enc1(42);
tx_data_matrix(2,30)=tx_enc0(270);
tx_data_matrix(2,31)=tx_enc1(106);
tx_data_matrix(2,32)=tx_enc0(334);
tx_data_matrix(2,33)=tx_enc1(170);
tx_data_matrix(2,34)=tx_enc0(398);
tx_data_matrix(2,35)=tx_enc1(234);
tx_data_matrix(2,36)=tx_enc0(6);
tx_data_matrix(2,37)=tx_enc1(298);
tx_data_matrix(2,38)=tx_enc0(70);
tx_data_matrix(2,39)=tx_enc1(362);
tx_data_matrix(2,40)=tx_enc0(134);
tx_data_matrix(2,41)=tx_enc1(426);
tx_data_matrix(2,42)=tx_enc0(198);
tx_data_matrix(2,43)=tx_enc1(34);
tx_data_matrix(2,44)=tx_enc0(262);
tx_data_matrix(2,45)=tx_enc1(98);
tx_data_matrix(2,46)=tx_enc0(326);
tx_data_matrix(2,47)=tx_enc1(162);
tx_data_matrix(2,48)=tx_enc0(390);
tx_data_matrix(2,49)=tx_enc1(226);
tx_data_matrix(2,50)=tx_enc0(454);
tx_data_matrix(2,51)=tx_enc1(290);
tx_data_matrix(2,52)=tx_enc0(62);
tx_data_matrix(2,53)=tx_enc1(354);
tx_data_matrix(2,54)=tx_enc0(126);
tx_data_matrix(2,55)=tx_enc1(418);
tx_data_matrix(2,56)=tx_enc0(190);
tx_data_matrix(2,57)=tx_enc1(26);
tx_data_matrix(2,58)=tx_enc0(254);
tx_data_matrix(2,59)=tx_enc1(90);
tx_data_matrix(2,60)=tx_enc0(318);
tx_data_matrix(2,61)=tx_enc1(154);
tx_data_matrix(2,62)=tx_enc0(382);
tx_data_matrix(2,63)=tx_enc1(218);
tx_data_matrix(2,64)=tx_enc0(446);
tx_data_matrix(2,65)=tx_enc1(282);
tx_data_matrix(2,66)=tx_enc0(54);
tx_data_matrix(2,67)=tx_enc1(346);
tx_data_matrix(2,68)=tx_enc0(118);
tx_data_matrix(2,69)=tx_enc1(410);
tx_data_matrix(2,70)=tx_enc0(182);
tx_data_matrix(2,71)=tx_enc1(18);
tx_data_matrix(2,72)=tx_enc0(246);
tx_data_matrix(2,73)=tx_enc1(82);
tx_data_matrix(2,74)=tx_enc0(310);
tx_data_matrix(2,75)=tx_enc1(146);
tx_data_matrix(2,76)=tx_enc0(374);
tx_data_matrix(2,77)=tx_enc1(210);
tx_data_matrix(2,78)=tx_enc0(438);
tx_data_matrix(2,79)=tx_enc1(274);
tx_data_matrix(2,80)=tx_enc0(46);
tx_data_matrix(2,81)=tx_enc1(338);
tx_data_matrix(2,82)=tx_enc0(110);
tx_data_matrix(2,83)=tx_enc1(402);
tx_data_matrix(2,84)=tx_enc0(174);
tx_data_matrix(2,85)=tx_enc1(10);
tx_data_matrix(2,86)=tx_enc0(238);
tx_data_matrix(2,87)=tx_enc1(74);
tx_data_matrix(2,88)=tx_enc0(302);
tx_data_matrix(2,89)=tx_enc1(138);
tx_data_matrix(2,90)=tx_enc0(366);
tx_data_matrix(2,91)=tx_enc1(202);
tx_data_matrix(2,92)=tx_enc0(430);
tx_data_matrix(2,93)=tx_enc1(266);
tx_data_matrix(2,94)=tx_enc0(38);
tx_data_matrix(2,95)=tx_enc1(330);
tx_data_matrix(2,96)=tx_enc0(102);
tx_data_matrix(2,97)=tx_enc1(394);
tx_data_matrix(2,98)=tx_enc0(166);
tx_data_matrix(2,99)=tx_enc1(2);
tx_data_matrix(2,100)=tx_enc0(230);
tx_data_matrix(2,101)=tx_enc1(66);
tx_data_matrix(2,102)=tx_enc0(294);
tx_data_matrix(2,103)=tx_enc1(130);
tx_data_matrix(2,104)=tx_enc0(358);
tx_data_matrix(2,105)=tx_enc1(194);
tx_data_matrix(2,106)=tx_enc0(422);
tx_data_matrix(2,107)=tx_enc1(258);
tx_data_matrix(2,108)=tx_enc0(30);
tx_data_matrix(2,109)=tx_enc1(322);
tx_data_matrix(2,110)=tx_enc0(94);
```

```
tx_data_matrix(2,111)=tx_enc1(386);
tx_data_matrix(2,112)=tx_enc0(158);
tx_data_matrix(2,113)=tx_enc1(450);
tx_data_matrix(2,114)=tx_enc0(222);
tx_data_matrix(3,1)=tx_enc1(115);
tx_data_matrix(3,2)=tx_enc0(343);
tx_data_matrix(3,3)=tx_enc1(179);
tx_data_matrix(3,4)=tx_enc0(407);
tx_data_matrix(3,5)=tx_enc1(243);
tx_data_matrix(3,6)=tx_enc0(15);
tx_data_matrix(3,7)=tx_enc1(307);
tx_data_matrix(3,8)=tx_enc0(79);
tx_data_matrix(3,9)=tx_enc1(371);
tx_data_matrix(3,10)=tx_enc0(143);
tx_data_matrix(3,11)=tx_enc1(435);
tx_data_matrix(3,12)=tx_enc0(207);
tx_data_matrix(3,13)=tx_enc1(43);
tx_data_matrix(3,14)=tx_enc0(271);
tx_data_matrix(3,15)=tx_enc1(107);
tx_data_matrix(3,16)=tx_enc0(335);
tx_data_matrix(3,17)=tx_enc1(171);
tx_data_matrix(3,18)=tx_enc0(399);
tx_data_matrix(3,19)=tx_enc1(235);
tx_data_matrix(3,20)=tx_enc0(7);
tx_data_matrix(3,21)=tx_enc1(299);
tx_data_matrix(3,22)=tx_enc0(71);
tx_data_matrix(3,23)=tx_enc1(363);
tx_data_matrix(3,24)=tx_enc0(135);
tx_data_matrix(3,25)=tx_enc1(427);
tx_data_matrix(3,26)=tx_enc0(199);
tx_data_matrix(3,27)=tx_enc1(35);
tx_data_matrix(3,28)=tx_enc0(263);
tx_data_matrix(3,29)=tx_enc1(99);
tx_data_matrix(3,30)=tx_enc0(327);
tx_data_matrix(3,31)=tx_enc1(163);
tx_data_matrix(3,32)=tx_enc0(391);
tx_data_matrix(3,33)=tx_enc1(227);
tx_data_matrix(3,34)=tx_enc0(455);
tx_data_matrix(3,35)=tx_enc1(291);
tx_data_matrix(3,36)=tx_enc0(63);
tx_data_matrix(3,37)=tx_enc1(355);
tx_data_matrix(3,38)=tx_enc0(127);
tx_data_matrix(3,39)=tx_enc1(419);
tx_data_matrix(3,40)=tx_enc0(191);
tx_data_matrix(3,41)=tx_enc1(27);
tx_data_matrix(3,42)=tx_enc0(255);
tx_data_matrix(3,43)=tx_enc1(91);
tx_data_matrix(3,44)=tx_enc0(319);
tx_data_matrix(3,45)=tx_enc1(155);
tx_data_matrix(3,46)=tx_enc0(383);
tx_data_matrix(3,47)=tx_enc1(219);
tx_data_matrix(3,48)=tx_enc0(447);
tx_data_matrix(3,49)=tx_enc1(283);
tx_data_matrix(3,50)=tx_enc0(55);
tx_data_matrix(3,51)=tx_enc1(347);
tx_data_matrix(3,52)=tx_enc0(119);
tx_data_matrix(3,53)=tx_enc1(411);
tx_data_matrix(3,54)=tx_enc0(183);
tx_data_matrix(3,55)=tx_enc1(19);
tx_data_matrix(3,56)=tx_enc0(247);
tx_data_matrix(3,57)=tx_enc1(83);
tx_data_matrix(3,58)=tx_enc0(311);
tx_data_matrix(3,59)=tx_enc1(147);
tx_data_matrix(3,60)=tx_enc0(375);
tx_data_matrix(3,61)=tx_enc1(211);
tx_data_matrix(3,62)=tx_enc0(439);
tx_data_matrix(3,63)=tx_enc1(275);
tx_data_matrix(3,64)=tx_enc0(47);
tx_data_matrix(3,65)=tx_enc1(339);
tx_data_matrix(3,66)=tx_enc0(111);
tx_data_matrix(3,67)=tx_enc1(403);
tx_data_matrix(3,68)=tx_enc0(175);
tx_data_matrix(3,69)=tx_enc1(11);
tx_data_matrix(3,70)=tx_enc0(239);
tx_data_matrix(3,71)=tx_enc1(75);
tx_data_matrix(3,72)=tx_enc0(303);
tx_data_matrix(3,73)=tx_enc1(139);
tx_data_matrix(3,74)=tx_enc0(367);
tx_data_matrix(3,75)=tx_enc1(203);
tx_data_matrix(3,76)=tx_enc0(431);
tx_data_matrix(3,77)=tx_enc1(267);
tx_data_matrix(3,78)=tx_enc0(39);
tx_data_matrix(3,79)=tx_enc1(331);
tx_data_matrix(3,80)=tx_enc0(103);
tx_data_matrix(3,81)=tx_enc1(395);
tx_data_matrix(3,82)=tx_enc0(167);
tx_data_matrix(3,83)=tx_enc1(3);
tx_data_matrix(3,84)=tx_enc0(231);
tx_data_matrix(3,85)=tx_enc1(67);
tx_data_matrix(3,86)=tx_enc0(295);
tx_data_matrix(3,87)=tx_enc1(131);
tx_data_matrix(3,88)=tx_enc0(359);
tx_data_matrix(3,89)=tx_enc1(195);
tx_data_matrix(3,90)=tx_enc0(423);
tx_data_matrix(3,91)=tx_enc1(259);
tx_data_matrix(3,92)=tx_enc0(31);
tx_data_matrix(3,93)=tx_enc1(323);
tx_data_matrix(3,94)=tx_enc0(95);
tx_data_matrix(3,95)=tx_enc1(387);
tx_data_matrix(3,96)=tx_enc0(159);
tx_data_matrix(3,97)=tx_enc1(451);
tx_data_matrix(3,98)=tx_enc0(223);
tx_data_matrix(3,99)=tx_enc1(59);
tx_data_matrix(3,100)=tx_enc0(287);
tx_data_matrix(3,101)=tx_enc1(123);
tx_data_matrix(3,102)=tx_enc0(351);
tx_data_matrix(3,103)=tx_enc1(187);
tx_data_matrix(3,104)=tx_enc0(415);
tx_data_matrix(3,105)=tx_enc1(251);
tx_data_matrix(3,106)=tx_enc0(23);
tx_data_matrix(3,107)=tx_enc1(315);
tx_data_matrix(3,108)=tx_enc0(87);
tx_data_matrix(3,109)=tx_enc1(379);
tx_data_matrix(3,110)=tx_enc0(151);
tx_data_matrix(3,111)=tx_enc1(443);
tx_data_matrix(3,112)=tx_enc0(215);
tx_data_matrix(3,113)=tx_enc1(51);
tx_data_matrix(3,114)=tx_enc0(279);
tx_data_matrix(4,1)=tx_enc1(172);
tx_data_matrix(4,2)=tx_enc0(400);
tx_data_matrix(4,3)=tx_enc1(236);
tx_data_matrix(4,4)=tx_enc0(8);
tx_data_matrix(4,5)=tx_enc1(300);
tx_data_matrix(4,6)=tx_enc0(72);
tx_data_matrix(4,7)=tx_enc1(364);
tx_data_matrix(4,8)=tx_enc0(136);
tx_data_matrix(4,9)=tx_enc1(428);
tx_data_matrix(4,10)=tx_enc0(200);
```

```
tx_data_matrix(4,11)=tx_enc1(36);
tx_data_matrix(4,12)=tx_enc0(264);
tx_data_matrix(4,13)=tx_enc1(100);
tx_data_matrix(4,14)=tx_enc0(328);
tx_data_matrix(4,15)=tx_enc1(164);
tx_data_matrix(4,16)=tx_enc0(392);
tx_data_matrix(4,17)=tx_enc1(228);
tx_data_matrix(4,18)=tx_enc0(456);
tx_data_matrix(4,19)=tx_enc1(292);
tx_data_matrix(4,20)=tx_enc0(64);
tx_data_matrix(4,21)=tx_enc1(356);
tx_data_matrix(4,22)=tx_enc0(128);
tx_data_matrix(4,23)=tx_enc1(420);
tx_data_matrix(4,24)=tx_enc0(192);
tx_data_matrix(4,25)=tx_enc1(28);
tx_data_matrix(4,26)=tx_enc0(256);
tx_data_matrix(4,27)=tx_enc1(92);
tx_data_matrix(4,28)=tx_enc0(320);
tx_data_matrix(4,29)=tx_enc1(156);
tx_data_matrix(4,30)=tx_enc0(384);
tx_data_matrix(4,31)=tx_enc1(220);
tx_data_matrix(4,32)=tx_enc0(448);
tx_data_matrix(4,33)=tx_enc1(284);
tx_data_matrix(4,34)=tx_enc0(56);
tx_data_matrix(4,35)=tx_enc1(348);
tx_data_matrix(4,36)=tx_enc0(120);
tx_data_matrix(4,37)=tx_enc1(412);
tx_data_matrix(4,38)=tx_enc0(184);
tx_data_matrix(4,39)=tx_enc1(20);
tx_data_matrix(4,40)=tx_enc0(248);
tx_data_matrix(4,41)=tx_enc1(84);
tx_data_matrix(4,42)=tx_enc0(312);
tx_data_matrix(4,43)=tx_enc1(148);
tx_data_matrix(4,44)=tx_enc0(376);
tx_data_matrix(4,45)=tx_enc1(212);
tx_data_matrix(4,46)=tx_enc0(440);
tx_data_matrix(4,47)=tx_enc1(276);
tx_data_matrix(4,48)=tx_enc0(48);
tx_data_matrix(4,49)=tx_enc1(340);
tx_data_matrix(4,50)=tx_enc0(112);
tx_data_matrix(4,51)=tx_enc1(404);
tx_data_matrix(4,52)=tx_enc0(176);
tx_data_matrix(4,53)=tx_enc1(12);
tx_data_matrix(4,54)=tx_enc0(240);
tx_data_matrix(4,55)=tx_enc1(76);
tx_data_matrix(4,56)=tx_enc0(304);
tx_data_matrix(4,57)=tx_enc1(140);
tx_data_matrix(4,58)=tx_enc0(368);
tx_data_matrix(4,59)=tx_enc1(204);
tx_data_matrix(4,60)=tx_enc0(432);
tx_data_matrix(4,61)=tx_enc1(268);
tx_data_matrix(4,62)=tx_enc0(40);
tx_data_matrix(4,63)=tx_enc1(332);
tx_data_matrix(4,64)=tx_enc0(104);
tx_data_matrix(4,65)=tx_enc1(396);
tx_data_matrix(4,66)=tx_enc0(168);
tx_data_matrix(4,67)=tx_enc1(4);
tx_data_matrix(4,68)=tx_enc0(232);
tx_data_matrix(4,69)=tx_enc1(68);
tx_data_matrix(4,70)=tx_enc0(296);
tx_data_matrix(4,71)=tx_enc1(132);
tx_data_matrix(4,72)=tx_enc0(360);
tx_data_matrix(4,73)=tx_enc1(196);
tx_data_matrix(4,74)=tx_enc0(424);
tx_data_matrix(4,75)=tx_enc1(260);
tx_data_matrix(4,76)=tx_enc0(32);
tx_data_matrix(4,77)=tx_enc1(324);
tx_data_matrix(4,78)=tx_enc0(96);
tx_data_matrix(4,79)=tx_enc1(388);
tx_data_matrix(4,80)=tx_enc0(160);
tx_data_matrix(4,81)=tx_enc1(452);
tx_data_matrix(4,82)=tx_enc0(224);
tx_data_matrix(4,83)=tx_enc1(60);
tx_data_matrix(4,84)=tx_enc0(288);
tx_data_matrix(4,85)=tx_enc1(124);
tx_data_matrix(4,86)=tx_enc0(352);
tx_data_matrix(4,87)=tx_enc1(188);
tx_data_matrix(4,88)=tx_enc0(416);
tx_data_matrix(4,89)=tx_enc1(252);
tx_data_matrix(4,90)=tx_enc0(24);
tx_data_matrix(4,91)=tx_enc1(316);
tx_data_matrix(4,92)=tx_enc0(88);
tx_data_matrix(4,93)=tx_enc1(380);
tx_data_matrix(4,94)=tx_enc0(152);
tx_data_matrix(4,95)=tx_enc1(444);
tx_data_matrix(4,96)=tx_enc0(216);
tx_data_matrix(4,97)=tx_enc1(52);
tx_data_matrix(4,98)=tx_enc0(280);
tx_data_matrix(4,99)=tx_enc1(116);
tx_data_matrix(4,100)=tx_enc0(344);
tx_data_matrix(4,101)=tx_enc1(180);
tx_data_matrix(4,102)=tx_enc0(408);
tx_data_matrix(4,103)=tx_enc1(244);
tx_data_matrix(4,104)=tx_enc0(16);
tx_data_matrix(4,105)=tx_enc1(308);
tx_data_matrix(4,106)=tx_enc0(80);
tx_data_matrix(4,107)=tx_enc1(372);
tx_data_matrix(4,108)=tx_enc0(144);
tx_data_matrix(4,109)=tx_enc1(436);
tx_data_matrix(4,110)=tx_enc0(208);
tx_data_matrix(4,111)=tx_enc1(44);
tx_data_matrix(4,112)=tx_enc0(272);
tx_data_matrix(4,113)=tx_enc1(108);
tx_data_matrix(4,114)=tx_enc0(336);
```

Source code C.5: `diff_enc.m`

```
function DIFF_ENC_DATA = diff_enc(BURST)
%
% diff_enc:   This function accepts a GSM burst bit sequence and
%             performs a differential encoding of the sequence. The
%             encoding is according to the GSM 05.05 recommendations
%
%             Input: D(i)
%             Output: A(i)
%
%             D'(i) = D(i) (+) D(i-1)  ; (+) = XOR
%                D(i), D'(i) = {0,1}
%             A(i) = 1 - 2*D'(i)
%                A(i) = {-1,1}
%
% SYNTAX:     [diff_enc_data] = diff_enc(burst)
%
% INPUT:      burst   A binary, (0,1), bit sequence
%
% OUTPUT:     diff_enc_data   A differential encoded, (+1,-1), version
%                             of the input burst sequence
%
% SUB_FUNC:   None
%
% WARNINGS:   None
%
% TEST(S):    Function tested
%
% AUTHOR:   Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: diff_enc.m,v 1.5 1998/02/12 10:49:36 aneks Exp $

L = length(BURST);

% INTERMEDIATE VECTORS FOR DATA PROCESSING
%
d_hat = zeros(1,L);
alpha = zeros(1,L);

% DIFFERENTIAL ENCODING ACCORDING TO GSM 05.05
% AN INFINITE SEQUENCE OF 1'ENS ARE ASSUMED TO
% PRECEED THE ACTUAL BURST
%
data = [1 BURST];
for n = 1+1:(L+1),
  d_hat(n-1) = xor( data(n),data(n-1) );
end
alpha = 1 - 2.*d_hat;

% PREPARING DATA FOR OUTPUT
%
DIFF_ENC_DATA = alpha;
```

Source code C.6: `gmsk_mod.m`

```
function [I,Q] = gmsk_mod(BURST,Tb,OSR,BT)
%
% gmsk_mod:   This function accepts a GSM burst bit sequence and
%             performs a GMSK modulation of the sequence. The
%             modulation is according to the GSM 05.05 recommendations
%
% SYNTAX:     [i,q] = gmsk_mod(burst,Tb,osr,BT)
%
% INPUT:      burst   A differential encoded bit sequence (-1,+1)
%             Tb      Bit duration (GSM: Tb = 3.692e-6 Sec.)
%             osr     Simulation oversample ratio. osr determines the
%                     number of simulation steps per information bit
%             BT      The bandwidth/bit duration product (GSM: BT = 0.3)
%
% OUTPUT:     i,q     In-phase (i) and quadrature-phase (q) baseband
%                     representation of the GMSK modulated input burst
%                     sequence
%
% SUB_FUNC:   ph_g.m  This sub function is required in generating the
%                     frequency and phase pulse functions.
%
% WARNINGS:   Sub function ph_g.m assumes a 3xTb frequency pulse
%             truncation time
%
% TEST(S):    Function tested using the following relations
%
%             i)
%
%             I^2 + Q^2 = Cos(a)^2 + Sin(a)^2 = 1
%
%             ii)
%
%             When the input consists of all 1's the resulting baseband
%             outputs the function should return a sinusoidal signal of
%             frequency rb/4, i.e. a signal having a periode time of
%             approximately 4*Tb = 4*3.692e-6 s = 1.48e-5 s for GSM.
%
% AUTHOR:   Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: gmsk_mod.m,v 1.5 1998/02/12 10:50:10 aneks Exp $

% ACCUIRE GMSK FREQUENCY PULSE AND PHASE FUNCTION
%
[g,q] = ph_g(Tb,OSR,BT);

% PREPARE VECTOR FOR DATA PROCESSING
%
bits = length(BURST);
f_res = zeros(1,(bits+2)*OSR);

% GENERATE RESULTING FREQUENCY PULSE SEQUENCE
%
for n = 1:bits,
  f_res((n-1)*OSR+1:(n+2)*OSR) = f_res((n-1)*OSR+1:(n+2)*OSR) + BURST(n).*g;
end

% CALCULATE RESULTING PHASE FUNCTION
%
theta = pi*cumsum(f_res);
```

Source code C.7: `gsm_mod.m`

```
function [ tx_burst , I , Q ] = gsm_mod(Tb,OSR,BT,tx_data,TRAINING)

% GSM_MOD:  This Matlab code generates a GSM normal burst by
%           combining tail, ctrl, and training sequence bits with
%           two bloks of random data bits.
%           The data bits are convolutional encoded according
%           to the GSM recommendations
%           The burst sequence is differential encoded and then
%           subsequently GMSK modulated to provide oversampled
%           I and Q baseband representations.
%
% SYNTAX:   [ tx_burst , I , Q ] = gsm_mod(Tb,OSR,BT,tx_data,TRAINING)
%
% INPUT:    Tb: Bit time, set by gsm_set.m
%          OSR: Oversampling ratio (fs/rb), set by gsm_set.m
%           BT: Bandwidth Bittime product, set by gsm_set.m
%      tx_data: The contents of the datafields in the burst to be
%               transmitted. Datafield one comes first.
%     TRAINING: The Training sequence which is to be inserted in the
%               burst.
%
% OUTPUT:
%     tx_burst: The entire transmitted burst before differential
%               precoding.
%            I: Inphase part of modulated burst.
%            Q: Quadrature part of modulated burst.
%
% WARNINGS: No interleaving or channel coding is done, and thus the
%           GSM recommadations are violated. Data simulations done using
%           this format can only be used for predicting Class II performance.
%
% AUTHOR:   Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: gsm_mod.m,v 1.10 1997/12/17 15:29:27 aneks Exp $

% GENERATE BURST SEQUENCE (THIS IS ACTUALLY THE MUX).
%
tx_burst = burst_g(tx_data,TRAINING);

% DIFFERENTIAL ENCODING.
%
burst = diff_enc(tx_burst);

% GMSK MODULATION OF BURST SEQUENCE
%
[I,Q] = gmsk_mod(burst,Tb,OSR,BT);

% PREPARE DATA FOR OUTPUT
%
I = cos(theta);
Q = sin(theta);
```

Source code C.8: `ph_g.m`

```
function [G_FUN, Q_FUN] = ph_g(Tb,OSR,BT)
%
% PH_G:      This function calculates the frequency and phase functions
%            required for the GMSK modulation. The functions are
%            generated according to the GSM 05.05 recommendations
%
% SYNTAX:    [g_fun, q_fun] = ph_g(Tb,osr,BT)
%
% INPUT:     Tb       Bit duration (GSM: Tb = 3.692e-6 Sec.)
%            osr      Simulation oversample ratio. osr determines the
%                     number of simulation steps per information bit
%            BT       The bandwidth/bit duration product (GSM: BT = 0.3)
%
% OUTPUT:    g_fun, q_fun   Vectors containing frequency and phase
%                           function outputs when evaluated at osr*tb
%
% SUB_FUNC: None
%
% WARNINGS: Modulation length of 3 is assumed !
%
% TEST(S):   Tested through function gsmk_mod.m
%
% AUTHOR:    Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:     hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: ph_g.m,v 1.6 1998/02/12 10:50:54 aneks Exp $

% SIMULATION SAMPLE FREQUENCY
%
Ts = Tb/OSR;

% PREPARING VECTORS FOR DATA PROCESSING
%
PTV = -2*Tb:Ts:2*Tb;
RTV = -Tb/2:Ts:Tb/2-Ts;

% GENERATE GAUSSIAN SHAPED PULSE
%
sigma = sqrt(log(2))/(2*pi*BT);
gauss = (1/(sqrt(2*pi)*sigma*Tb))*exp(-PTV.^2/(2*sigma^2*Tb^2));

% GENERATE RECTANGULAR PULSE
%
rect = 1/(2*Tb)*ones(size(RTV));

% CALCULATE RESULTING FREQUENCY PULSE
%
G_TEMP = conv(gauss,rect);

% TRUNCATING THE FUNCTION TO 3xTb
%
G = G_TEMP(OSR+1:4*OSR);

% TRUNCATION IMPLIES THAT INTEGRATING THE FREQUENCY PULSE
% FUNCTION WILL NOT EQUAL 0.5, HENCE THE RE-NORMALIZATION
%
G_FUN = (G-G(1))./(2*sum(G-G(1)));

% CALCULATE RESULTING PHASE PULSE
%
Q_FUN = cumsum(G_FUN);
```

Source code C.9: `make_increment.m`

```
function [ INCREMENT ] = make_increment(SYMBOLS,NEXT,Rhh)
%
% MAKE INCREMENT:
%           This function returns a lookuptable containing the
%           metric increments related to moving from state n to m.
%           The data is arranged so that the increment accosiated
%           with a move from state n to m is located in
%           INCREMENT(n,m). To minimize computations only legal
%           transitions are considdered.
%
% SYNTAX:   [ INCREMENT ] = make_increment(SYMBOLS,NEXT,Rhh)
%
% INPUT:    SYMBOLS: The table of symbols corresponding the the state-
%                    numbers.
%           NEXT:    A transition table containing the next legal
%                    states, as it is generated by the code make_next.
%           Rhh:     The autocorrelation as estimated by mf.m.
%
% OUTPUT:   INCREMENT:
%                    The increment table as described above.
%
% SUB FUNC: None
%
% WARNINGS: There is no syntax checking on input or output.
%
% TEST(S):  By hand, against expected values.
%
% AUTOR:    Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: make_increment.m,v 1.6 1997/09/22 11:39:34 aneks Exp $

% IN THIS PEACE OF CODE THE SYNTAX CHECKING IS MINIMAL
% THIS HAS BEEN CHOSEN TO AVOID THE OVERHEAD. RECALL THAT
% THIS CODE IS EXECUTED EACH TIME A BURST IS RECEIVED.

% FIND THE NUMBER OF SYMBOLS THAT WE HAVE
%
[M,Lh]=size(SYMBOLS);

% INITIALIZE THE INCREMENT MATRIX
%
INCREMENT=zeros(M);

% RECALL THAT THE I SEQUENCE AS IT IS STORED IN STORED AS:
% [ I(n-1) I(n-2) I(n-3) ... I(n-Lh) ]
%
% ALSO RECALL THAT Rhh IS STORED AS:
% [ Rhh(1) Rhh(2) Rhh(3) ... Rhh(Lh) ]
%
% THE FORMULA TO USE IS:
% INCREMENT(n,m)
% =
% real(conj(I(n))*(I(n-Lh)*Rhh(Lh)+I(n-Lh+1)*Rhh(Lh-1)+...+I(n-1)*Rhh(1))
%
% THEY CAN THUS BE MULTIPLIED DIRECTLY WITH EACH OTHER

% LOOP OVER THE STATES, AS FOUND IN THE ROWS IN SYMBOLS.
%
for n=1:M,
  % ONLY TWO LEGAL NEXT STATES EXIST, SO THE LOOP IS UNROLLED
  %
  m=NEXT(n,1);
  INCREMENT(n,m)=real(conj(SYMBOLS(m,1))*SYMBOLS(n,:)*Rhh(2:Lh+1).');
  m=NEXT(n,2);
  INCREMENT(n,m)=real(conj(SYMBOLS(m,1))*SYMBOLS(n,:)*Rhh(2:Lh+1).');
end
```

Source code C.10: `make_next.m`

```
function [ NEXT ] = make_next(SYMBOLS)
%
% MAKE_NEXT:
%           This function returns a lookuptable containing a mapping
%           between the present state and the legal next states.
%           Each row correspond to a state, and the two legal states
%           related to state n is located in NEXT(n,1) and in
%           NEXT(n,2). States are represented by their related
%           numbers.
%
% SYNTAX:   [ NEXT ] = make_next(SYMBOLS)
%
% INPUT:    SYMBOLS: The table of symbols corresponding the the state
%                    numbers.
%
% OUTPUT:   NEXT:    The transition table describing the legal next
%                    states asdescribed above.
%
% SUB_FUNC: None
%
% WARNINGS: None
%
% TEST(S):  The function has been verified to return the expected
%           results.
%
% AUTOR:    Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: make_next.m,v 1.3 1997/09/22 08:13:29 aneks Exp $

% FIRST WE NEED TO FIND THE NUMBER OF LOOPS WE SHOULD RUN.
% THIS EQUALS THE NUMBER OF SYMBOLS. ALSO MAXSUM IS NEEDED FOR
% LATER OPERATIONS.
%
[ states , maxsum ]=size(SYMBOLS);

search_matrix=SYMBOLS(:,2:maxsum);
maxsum=maxsum-1;

% LOOP OVER THE SYMBOLS.
%
for this_state=1:states,
  search_vector=SYMBOLS(this_state,1:maxsum);
  k=0;
  for search=1:states,
    if (sum(search_matrix(search,:)==search_vector)==maxsum)
      k=k+1;
      NEXT(this_state,k)=search;
      if k > 2,
        error('Error: Identified too many next states');
      end
    end
  end
end
```

Source code C.11: `make_previous.m`

```
function [ PREVIOUS ] = make_previous(SYMBOLS)
%
% MAKE_PREVIOUS:
%           This function returns a lookuptable containing a mapping
%           between the present state and the legal previous states.
%           Each row correspond to a state, and the two legal states
%           related to state n is located in PREVIOUS(n,1) and in
%           previous(n,2). States are represented by their related
%           numbers.
%
% SYNTAX:   [ PREVIOUS ] = make_previous(SYMBOLS)
%
% INPUT:    SYMBOLS: The table of symbols corresponding the the state
%                    numbers.
%
% OUTPUT:   PREVIOUS:
%                    The transition table describing the legal previous
%                    states asdescribed above.
%
% SUB_FUNC: None
%
% WARNINGS: None
%
% TEST(S):  Verified against expected result.
%
% AUTOR:    Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: make_previous.m,v 1.3 1997/09/22 08:14:27 aneks Exp $

% FIRST WE NEED TO FIND THE NUMBER OF LOOPS WE SHOULD RUN.
% THIS EQUALS THE NUMBER OF SYMBOLS. ALSO MAXSUM IS NEEDED FOR
% LATER OPERATIONS.
%
[ states , maxsum ]=size(SYMBOLS);

maxsum=maxsum-1;
search_matrix=SYMBOLS(:,1:maxsum);

% LOOP OVER THE SYMBOLS.
%
for this_state=1:states,
  search_vector=SYMBOLS(this_state,2:maxsum+1);
  k=0;
  for search=1:states,
    if (sum(search_matrix(search,:)==search_vector)==maxsum)
      k=k+1;
      PREVIOUS(this_state,k)=search;
      if k > 2,
        error('Error: Identified too many previous states');
      end
    end
  end
end
```

Source code C.12: `make_start.m`

```
function [ START ] = make_start(Lh,SYMBOLS)
%
% MAKE START:
%           This code returns a statenumber corresponding to the start
%           state as it is found from Lh. The method is to use the table
%           of symbolic start states as it is listed in the report made
%           by 95gr870T. For the table lookups are made in SYMBOLS. in
%           order to map from the symbol representation to the state number
%           representation.
%
% SYNTAX:   [ START ] = make_start(Lh,SYMBOLS)
%
% INPUT:    SYMBOLS: The table of symbols corresponding the the state-
%                    numbers.
%           Lh:      Length of the estimated impulseresponse.
%
% OUTPUT:   START:   The number representation of the legal start state.
%
% SUB FUNC: None
%
% WARNINGS: The table of symbolic representations has not been verified
%           but is used directly as it is listed in the report made
%           by 95gr870T.
%
% TEST(S):  The function has been verified to return a state number
%           which matches the symbolic representation.
%
% AUTOR:    Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: make_start.m,v 1.2 1997/09/22 11:40:17 aneks Exp $

% WE HAVEN'T FOUND IT YET.
%
START_NOT_FOUND = 1;

% OBTAIN THE SYMBOLS FROM Lh. THIS IS THE TABLE LISTED IN THE REPORT MADE
% BY 95gr870T. (SATEREPRESENTATION IS SLIGHTLY CHANGED).
%
if Lh==1,
  start_symbols = [ 1 ];
elseif Lh==2,
  start_symbols = [ 1 -1 ];
elseif Lh==3,
  start_symbols = [ 1 -1 -1 ];
elseif Lh==4,
  start_symbols = [ 1 -1 -1 1 ];
else
  fprintf('\n\nError: Illegal value of Lh, terminating...');
end

% NOW MAP FROM THE SYMBOLS TO A STATE NUMBER BY SEARCHING
% SYMBOLS.
%
START=0;
while START_NOT_FOUND,
  START=START+1;
  if sum(SYMBOLS(START,:)==start_symbols)==Lh,
    START_NOT_FOUND=0;
  end
end
```

Source code C.13: `make_stops.m`

```
function [ STOPS ] = make_stops(Lh,SYMBOLS)
%
% MAKE STOPS:
%          This code returns a statenumber corresponding to the set of
%          legal stop states as found from Lh. The method is to use the table
%          of symbolic stop states as it is listed in the report made
%          by 95gr870T. For the table lookups are made in SYMBOLS. in
%          order to map from the symbol representation to the state number
%          representation.
%
% SYNTAX:   [ STOPS ] = make_stops(Lh,SYMBOLS)
%
% INPUT:    SYMBOLS: The table of symbols corresponding the the state-
%                    numbers.
%          Lh:      Length of the estimated impulseresponse.
%
% OUTPUT:   STOPS:   The number representation of the set of legal stop
%                    states.
%
% SUB FUNC: None
%
% WARNINGS: The table of symbolic representations has not been verified
%          but is used directly as it is listed in the report made
%          by 95gr870T.
%
% TEST(S):  The function has been verified to return a state number
%          which matches the symbolic representation.
%
% AUTOR:    Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: make_stops.m,v 1.2 1997/09/22 11:44:21 aneks Exp $

% OBTAIN THE SYMBOLS FROM Lh. THIS IS THE TABLE LISTED IN THE REPORT MADE
% BY 95gr870T. (SATEREPRESENTATION IS SLIGHTLY CHANGED).
%
if Lh==1,
  stop_symbols = [ 1 ];
  count=1;
elseif Lh==2,
  stop_symbols = [ 1 j ];
  count=1;
elseif Lh==3,
  stop_symbols = [ 1 j 1 ];
  count=1;
elseif Lh==4,
  stop_symbols = [ [ -1 j 1 -j ] ; [ -1 j 1 j ] ];
  count=2;
else
  fprintf('\n\nError: Illegal value of Lh, terminating...');
end

% NOW THAT WE HAVE THE SYMBOL REPRESENTATION THE REMAINING JOB IS
% TO MAP THE MSK SYMBOLS TO STATE NUMBERS
%
index = 0;
stops_found=0;
while stops_found < count,
  index=index+1;
  if sum(SYMBOLS(index,:)==stop_symbols(stops_found+1,:))==Lh,
    stops_found=stops_found+1;
    STOPS(stops_found)=index;
  end
end
```

Source code C.14: `make_symbols.m`

```
function [ SYMBOLS ] = make_symbols(Lh)
%
% MAKE SYMBOLS:
%           This function returns a table containing the mapping
%           from state numbers to symbols. The table is contained
%           in a matrix, and the layout is:
%
%
%           | Symbols for state 1 |
%           | Symbols for state 2 |
%                   :
%                   :
%           | Symbols for state M |
%           -                     -
%
%           Where M is the total number of states, and can be calculated
%           as: 2^(Lh+1). Lh is the length of the estimated impulse
%           response, as found in the mf-routine. In the symbols for
%           a statenumber the order is as:
%
%           I(n-1) I(n-2) I(n-3) ... I(n-Lh)
%
%           Each of the symbols belong to { 1 , -1 , j , -j }.
%
% SYNTAX:   [SYMBOLS] = make_symbols(Lh)
%
% INPUT:    Lh:      Length of the estimated impulse resonse.
%
% OUTPUT:   SYMBOLS: The table of symbols corresponding the the state-
%                    numbers, as described above.
%
% SUB_FUNC: None
%
% WARNINGS: None
%
% TEST(S):  Compared result against expected values.
%
% AUTOR:    Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: make_symbols.m,v 1.6 1997/09/22 11:38:57 aneks Exp $

% THIS CODE CANNOT HANDLE Lh=1 or Lh>4.
%
if Lh==1,
  error('GSMsim-Error: Lh is constrained to be in the interval [1:4].');
elseif Lh > 4,
  error('GSMsim-Error: Lh is constrained to be in the interval [1:4].')
end

% make initiating symbols
%
SYMBOLS=[ 1; j; -1 ; -j];
%
for n=1:Lh-1
  SYMBOLS=[[ SYMBOLS(:,1)*j , SYMBOLS ] ; [ SYMBOLS(:,1)*(-j) , SYMBOLS ]];
end
%
% NOW WE NEED TO ASSURE THAT THE STATE RELATED TO THE NUMBER ONE
% IS COMPLEX. THIS IS REQUIRED BY THE IMPLEMENTATION OF THE VITERBI
% ALGORITHM.
%
if isreal(SYMBOLS(1,1)),
  SYMBOLS=flipud(SYMBOLS);
end
```

Source code C.15: mf.m (Renamed to mafi.m)

```
function [Y, Rhh] = mafi(r,L,T_SEQ,OSR)
%
% MAFI:      This function performes the tasks of channel impulse
%            respons estimation, bit syncronization, matched
%            filtering and signal sample rate downconversion.
%
% SYNTAX:    [y, Rhh] = mafi(r,Lh,T_SEQ,OSR)
%
% INPUT:     r      Complex baseband representation of the received
%                   GMSK modulated signal
%            Lh     The desired length of the matched filter impulse
%                   response measured in bit time durations
%            T_SEQ  A MSK-modulated representation of the 26 bits long
%                   training sequence used in the transmitted burst,
%                   i.e. the training sequence used in the generation of r
%            OSR    Oversampling ratio, defined as f_s/r_b.
%
% OUTPUT:    Y      Complex baseband representation of the matched
%                   filtered and down converted received signal
%
%            Rhh    Autocorrelation of the estimated channel impulse
%                   response. The format is a Lh+1 unit long column vector
%                   starting with Rhh[0], and ending with Rhh[Lh].
%                   Complex valued.
%
% SUB_FUNC:  None
%
% WARNINGS:  The channel estimation is based on the 16 most central
%            bits of the training sequence only
%
% TEST(S):   Tested manually through test script mf_ill.m
%
% AUTOR:     Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:     hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: mafi.m,v 1.1 1998/10/01 10:20:21 hmi Exp $

DEBUG=0;

% PICK CENTRAL 16 BITS [ B   C | A | AS COMPROMISE
% AND PERFORM COMPLEX CONJUGATION
%
T16 = conj(T_SEQ(6:21));

% EXTRACT RELEVANT PART OF THE RECEIVED SIGNAL. USING
% GUARD TIMES AS GUIDELINES IMPLIES EXTRACTING THE PART
% STARTING APPROXIMATELY AT 10 Tb's BEFORE THE 16 MOST
% CENTRAL TRAINING SEQUENCE BITS AND ENDING APPROXIMATELY
% 10 Tb's AFTER. ASSUME THAT BURSTS TEND TO BE CENTERED IN
% A SAMPLE STREAM.
%
GUARD = 10;
center_r=round(length(r)/2);
start_sub=center_r-(GUARD+8)*OSR;
end_sub=center_r+(GUARD+8)*OSR;

% YOU MAY WANT TO ENABLE THIS FOR SPECIAL DEBUGGING
%
%start_sub=1;
%end_sub=length(r);

r_sub = r(start_sub:end_sub);

if DEBUG,
  % DEBUGGING, VERIFIES THAT WE PICK THE RIGHT PART OUT
  %
  count=1:length(r);
  figure
  plot(count,real(r));
  plug=start_sub:end_sub;
  hold on;
  plot(plug,real(r_sub),'r')
  hold off;
  title('Real part of r and r_sub (red)');
  %pause;
end

% PREPARE VECTOR FOR DATA PROCESSING
%
chan_est = zeros(1,length(r_sub)-OSR*16);

% ESTIMATE CHANNEL IMPULSE RESPONSE USING ONLY EVERY
% OSR'th SAMPLE IN THE RECEIVED SIGNAL
%
for n = 1:length(chan_est),
  chan_est(n)=r_sub(n:OSR:n+15*OSR)*T16.';
end

if DEBUG,
  % DEBUGGING, PROVIDES A PLOT OF THE ESTIMATED IMPULSE
  % RESPONSE FOR THE USER TO GAZE AT
  figure;
  plot(abs(chan_est));
  title('The absolute value of the correlation');
  %pause;
end

chan_est = chan_est./16;

% EXTRACTING ESTIMATED IMPULS RESPONS BY SEARCHING FOR MAXIMUM
% POWER USING A WINDOW OF LENGTH OSR*(L+1)
%
WL = OSR*(L+1);

search = abs(chan_est).^2;
for n = 1:(length(search)-(WL-1)),
  power_est(n) = sum(search(n:n+WL-1));
end

if DEBUG,
  % DEBUGGING, SHOWS THE POWER ESTIMATE
  figure;
  plot(power_est);
  title('The window powers');
  %pause;
end

% SEARCHING FOR MAXIMUM VALUE POWER WINDOW AND SELECTING THE
% CORRESPONDING ESTIMATED MATCHED FILTER TAP COEFFICIENS. ALSO,
% THE SYNCRONIZATION SAMPLE CORRESPONDING TO THE FIRST SAMPLE
% IN THE T16 TRAINING SEQUENCE IS ESTIMATED
%
[peak, sync_w] = max(power_est);
h_est = chan_est(sync_w:sync_w+WL-1);
```

```
[peak, sync_h] = max(abs(h_est));
sync_T16 = sync_w + sync_h - 1;

if DEBUG,
  % DEBUGGING, SHOWS THE POWER ESTIMATE
  figure;
  plot(abs(h_est));
  title('Absolute value of extracted impulse response');
  %pause;
end

% WE WANT TO USE THE FIRST SAMPLE OF THE IMPULSERESPONSE, AND THE
% CORRESPONDING SAMPLES OF THE RECEIVED SIGNAL.
% THE VARIABLE sync_w SHOULD CONTAIN THE BEGINNING OF THE USED PART OF
% TRAINING SEQUENCE, WHICH IS 3+57+1+6=67 BITS INTO THE BURST. THAT IS
% WE HAVE THAT sync_T16 EQUALS FIRST SAMPLE IN BIT NUMBER 67.
%
burst_start = ( start_sub + sync_T16 - 1 ) - ( OSR * 66 + 1 ) + 1;

% COMPENSATING FOR THE 2 Tb DELAY INTRODUCED IN THE GMSK MODULATOR.
% EACH BIT IS STRECHED OVER A PERIOD OF 3 Tb WITH ITS MAXIMUM VALUE
% IN THE LAST BIT PERIOD. HENCE, burst_start IS 2 * OSR MISPLACED.
%
burst_start = burst_start - 2*OSR + 1;

% CALCULATE AUTOCORRELATION OF CHANNEL IMPULSE
% RESPONS. DOWN CONVERSION IS CARRIED OUT AT THE SAME
% TIME
%
R_temp = xcorr(h_est);

pos = (length(R_temp)+1)/2;

Rhh=R_temp(pos:OSR:pos+L*OSR);

% PERFORM THE ACTUAL MATCHED FILTERING
%

m = length(h_est)-1;

% A SINGLE ZERO IS INSERTED IN FRONT OF r SINCE THERE IS AN EQUAL
% NUMBER OF SAMPLES IN r_sub WE CANNOT BE TOTALLY CERTAIN WHICH
% SIDE OF THE MIDDLE THAT IS CHOSEN THUS AN EXTRA SAMPLE IS
% NEEDED TO AVOID CROSSING ARRAY BOUNDS.
%
GUARDmf = (GUARD+1)*OSR;
r_extended = [ zeros(1,GUARDmf) r zeros(1,m) zeros(1,GUARDmf) ];

% RECALL THAT THE ' OPERATOR IN MATLAB DOES CONJUGATION
%

for n=1:148,
  aa=GUARDmf+burst_start+(n-1)*OSR;
  bb=GUARDmf+burst_start+(n-1)*OSR+m;
  Y(n) = r_extended(aa:bb)*h_est';
end
```

Source code C.16: viterbi_detector.m

```
function [ rx_burst ] = viterbi_detector(SYMBOLS,NEXT,PREVIOUS,START,STOPS,Y,Rhh)
%
% VITERBI DETECTOR:
%           This matlab code does the actual detection of the
%           received sequence. As indicated by the name the algorithm
%           is the viterbi algorithm, which is a MLSE. At this time
%           the approach is to use Ungerboecks modified algorithm, and
%           to return hard output only.
%
% SYNTAX:   [ rx_burst ]
%           =
%           viterbi_detector(SYMBOLS,NEXT,PREVIOUS,START,STOPS,Y,Rhh)
%
% INPUT:    SYMBOLS:  The table of symbols corresponding the the state-
%                     numbers. Format as made by make_symbols.m
%           NEXT:     A transition table containing the next legal
%                     states, as it is generated by the code make_next.m
%           PREVIOUS: The transition table describing the legal previous
%                     states as generated by make_previous.m
%           START:    The start state of the algorithm.
%           STOPS:    The legal stop states.
%           Y:        Complex baseband representation of the matched
%                     filtered and down converted received signal, as it
%                     is returned by mf.m
%           Rhh:      The autocorrelation as estimated by mf.m
%
% OUTPUT:   rx_burst: The most likely sequence of symbols.
%
% SUB_FUNC: make_increment
%
% WARNINGS: None.
%
% TEST(S):  Tested with no noise, perfect syncronization, and channel
%           estimation/filtering. (Refer to viterbi_ill.m)
%
% AUTOR:    Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: viterbi_detector.m,v 1.7 1997/11/18 12:41:26 aneks Exp $

% KNOWLEDGE OF Lh AND M IS NEEDED FOR THE ALGORITHM TO OPERATE
%
[ M , Lh ] = size(SYMBOLS);

% THE NUMBER OF STEPS IN THE VITERBI
%
STEPS=length(Y);

% INITIALIZE TABLES (THIS YIELDS A SLIGHT SPEEDUP).
%
METRIC = zeros(M,STEPS);
SURVIVOR = zeros(M,STEPS);

% DETERMINE PRECALCULATABLE PART OF METRIC
%
INCREMENT=make_increment(SYMBOLS,NEXT,Rhh);

% THE FIRST THING TO DO IS TO ROLL INTO THE ALGORITHM BY SPREADING OUT
% FROM  THE START STATE TO ALL THE LEGAL STATES.
%
```

```
PS-START;

% NOTE THAT THE START STATE IS REFERRED TO AS STATE TO TIME 0
% AND THAT IT HAS NO METRIC.
%
S-NEXT(START,1);
METRIC(S,1)-real(conj(SYMBOLS(S,1))*Y(1))-INCREMENT(PS,S);
SURVIVOR(S,1)-START;

S-NEXT(START,2);
METRIC(S,1)-real(conj(SYMBOLS(S,1))*Y(1)) INCREMENT(PS,S);
SURVIVOR(S,1)-START;

PREVIOUS STATES-NEXT(START,:);

% MARK THE NEXT STATES AS REAL. N.B: COMPLEX INDICATES THE POLARITY
% OF THE NEXT STATE, E.G. STATE 2 IS REAL.
%
COMPLEX-0;

for N - 2:Lh,
  if COMPLEX,
    COMPLEX-0;
  else
    COMPLEX-1;
  end
  STATE_CNTR-0;
  for PS - PREVIOUS STATES,
    STATE CNTR-STATE CNTR+1;
    S-NEXT(PS,1);
    METRIC(S,N)-METRIC(PS,N 1)+real(conj(SYMBOLS(S,1))*Y(N)) INCREMENT(PS,S);
    SURVIVOR(S,N)-PS;
    USED(STATE CNTR)-S;
    STATE CNTR-STATE CNTR+1;
    S-NEXT(PS,2);
    METRIC(S,N)-METRIC(PS,N 1)+real(conj(SYMBOLS(S,1))*Y(N)) INCREMENT(PS,S);
    SURVIVOR(S,N)-PS;
    USED(STATE_CNTR)-S;
  end
  PREVIOUS_STATES-USED;
end
% AT ANY RATE WE WILL HAVE PROCESSED Lh STATES AT THIS TIME
%
PROCESSED-Lh;

% WE WANT AN EQUAL NUMBER OF STATES TO BE REMAINING. THE NEXT LINES ENSURE
% THIS.
%

if ~COMPLEX,
  COMPLEX-1;
  PROCESSED-PROCESSED+1;
  N-PROCESSED;
  for S - 2:2:M,
    PS-PREVIOUS(S,1);
    M1-METRIC(PS,N-1)-real(conj(SYMBOLS(S,1))*Y(N)-INCREMENT(PS,S));
    PS-PREVIOUS(S,2);
    M2-METRIC(PS,N-1)-real(conj(SYMBOLS(S,1))*Y(N)-INCREMENT(PS,S));
    if M1 > M2,
      METRIC(S,N)-M1;
      SURVIVOR(S,N)-PREVIOUS(S,1);
    else
      METRIC(S,N)-M2;
      SURVIVOR(S,N)-PREVIOUS(S,2);
    end
  end
end

% NOW THAT WE HAVE MADE THE RUN-IN THE REST OF THE METRICS ARE
% CALCULATED IN THE STRAIGHT FORWARD MANNER. OBSERVE THAT ONLY
% THE RELEVANT STATES ARE CALCULATED, THAT IS REAL FOLLOWS COMPLEX
% AND VICE VERSA.
%
N-PROCESSED-1;
while N <- STEPS,
  for S - 1:2:M 1,
    PS-PREVIOUS(S,1);
    M1-METRIC(PS,N 1)+real(conj(SYMBOLS(S,1))*Y(N) INCREMENT(PS,S));
    PS-PREVIOUS(S,2);
    M2-METRIC(PS,N 1)+real(conj(SYMBOLS(S,1))*Y(N) INCREMENT(PS,S));
    if M1 > M2,
      METRIC(S,N)-M1;
      SURVIVOR(S,N)-PREVIOUS(S,1);
    else
      METRIC(S,N)-M2;
      SURVIVOR(S,N)-PREVIOUS(S,2);
    end
  end
  N-N-1;
  for S - 2:2:M,
    PS-PREVIOUS(S,1);
    M1-METRIC(PS,N 1)+real(conj(SYMBOLS(S,1))*Y(N) INCREMENT(PS,S));
    PS-PREVIOUS(S,2);
    M2-METRIC(PS,N 1)+real(conj(SYMBOLS(S,1))*Y(N) INCREMENT(PS,S));
    if M1 > M2,
      METRIC(S,N)-M1;
      SURVIVOR(S,N)-PREVIOUS(S,1);
    else
      METRIC(S,N)-M2;
      SURVIVOR(S,N)-PREVIOUS(S,2);
    end
  end
  N-N-1;
end


% HAVING CALCULATED THE METRICS, THE MOST PROBABLE STATESEQUENCE IS
% INITIALIZED BY CHOOSING THE HIGHEST METRIC AMONG THE LEGAL STOP
% STATES.
%
BEST LEGAL-0;
for FINAL - STOPS,
  if METRIC(FINAL,STEPS) > BEST LEGAL,
    S-FINAL;
    BEST LEGAL-METRIC(FINAL,STEPS);
  end
end

% UNCOMMENT FOR TEST OF METRIC
%
% METRIC
% BEST_LEGAL
% S
% pause

% HAVING FOUND THE FINAL STATE, THE MSK SYMBOL SEQUENCE IS ESTABLISHED
%
TEST(STEPS)-SYMBOLS(S,1);
```

```
N-STEPS 1;
while N > 0,
  S-SURVIVOR(S,N+1);
  IEST(N)-SYMBOLS(S,1);
  N-N-1;
end

% THE ESTIMATE IS NOW FOUND FROM THE FORMULA:
% IEST(n)-j*rx_burst(n)*rx_burst(n-1)*IEST(n-1)
% THE FORMULA IS REWRITTEN AS:
% rx burst(n)-IEST(n)/(j*rx burst(n-1)*IEST(n 1))
% FOR INITIALIZATION THE FOLLOWING IS USED:
% IEST(0)-1 og rx burst(0)-1
%
rx burst(1)-IEST(1)/(j*1*1);
for n - 2:STEPS,
  rx burst(n)-IEST(n)/(j*rx burst(n-1)*IEST(n 1));
end

% rx_burst IS POLAR (-1 AND 1), THIS TRANSFORMS IT TO
% BINARY FORM (0 AND 1).
%
rx_burst-(rx_burst-1)./2;
```

Source code C.17: `viterbi_init.m`

```
function [ SYMBOLS , PREVIOUS , NEXT , START , STOPS ] - viterbi_init(Lh)
% VITERBI_INIT:
%           This function returns the tables which are used by the
%           viterbi demodulator which is implemented in the GSMsim
%           package.
%
% SYNTAX:   [ SYMBOLS , PREVIOUS , NEXT , START , STOPS ]
%           -
%           viterbi init(Lh)
%
% INPUT:    Lh:       The length of the channel impulse response
%                     minus one.
%
% OUTPUT:   SYMBOLS:  Statenumber to MSK symbols mapping table.
%           PREVIOUS: This state to legal previous state mapping table.
%           NEXT:     This state to legal next state mapping table.
%           START:    The start state of the viterbi algorithm.
%           STOPS:    The set of legal stop states for the viterbi
%                     algorithm.
%
% GLOBAL:   None
%
% SUB_FUNC: make_symbols,make_previous,make_next,make_start,make_stops
%
% WARNINGS: None
%
% TEST:     Verified that the function actually runs the subfunctions.
%
% AUTHOR:   Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: viterbi init.m,v 1.4 1998/02/12 10:52:15 aneks Exp $

SYMBOLS - make_symbols(Lh);
PREVIOUS - make_previous(SYMBOLS);
NEXT - make_next(SYMBOLS);
START - make_start(Lh,SYMBOLS);
STOPS - make_stops(Lh,SYMBOLS);
```

Source code C.18: DeMUX.m

```
function [ rx_data ] = DeMUX(rx_burst)

% DeMUX:   This pice of code does the demultiplexing of the received
%          GSM burst.
%
% SYNTAX:   [ rx_data ] = DeMUX(rx_burst)
%
% INPUT: ESTIMAB: An entire 148 bit GSM burst. The format is expected
%                 to be:
%
%           | TAIL   DATA   CTRL | TRAINING   CTRL | DATA   TAIL |
%           |  3      57     1   |    26       1   |  57     3   |
%
% OUTPUT:
%      rx_data: The contents of the datafields in the received burst.
%
% WARNINGS: None.
%
% AUTHOR:   Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: DeMUX.m,v 1.3 1997/11/18 12:46:18 aneks Exp $

rx_data=[ rx_burst(4:60) , rx_burst(89:145) ];
```

Source code C.19: deinterleave.m

```
function [ rx_enc ] = deinterleave(rx_data_matrix)
%
% deinterleave:
%           This function does deinterleaving of de multiplexed GSM
%           information bursts, eg. 114 sequential bits as extracted
%           from a GSM burst. The input is 8 x 114 bit, and the output
%           is a single 456 bit information block, as deinterleaved from the
%           input.
%
% SYNTAX:   [ rx_enc ] = deinterleave(rx_data_matrix)
%
% INPUT:    rx_data_matrix:
%                 The 'latest' 8 instances of rx_data, which are 114 bit
%                 long, and must be stored in the rows of rx_data_matrix. If
%                 the bursts in the matrix are numbered as they sre
%                 received, the burst in row one has number one, etc.
%
% OUTPUT:   rx_enc:
%                 A 456 bit datablock, as demultiplexed from the 8 input
%                 bursts.
%
% WARNINGS: Observe that not all 8 x 114 bits are contained in the output.
%
% TEST(S):  interleave  > deinterleave = 0 Errors.
%
% AUTHOR:   Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: deinterleave.m,v 1.4 1997/11/20 11:12:27 aneks Exp $
rx_enc(1)=rx_data_matrix(1,1);
rx_enc(2)=rx_data_matrix(2,99);
rx_enc(3)=rx_data_matrix(3,83);
rx_enc(4)=rx_data_matrix(4,67);
rx_enc(5)=rx_data_matrix(5,52);
rx_enc(6)=rx_data_matrix(6,36);
rx_enc(7)=rx_data_matrix(7,20);
rx_enc(8)=rx_data_matrix(8,4);
rx_enc(9)=rx_data_matrix(1,101);
rx_enc(10)=rx_data_matrix(2,85);
rx_enc(11)=rx_data_matrix(3,69);
rx_enc(12)=rx_data_matrix(4,53);
rx_enc(13)=rx_data_matrix(5,38);
rx_enc(14)=rx_data_matrix(6,22);
rx_enc(15)=rx_data_matrix(7,6);
rx_enc(16)=rx_data_matrix(8,104);
rx_enc(17)=rx_data_matrix(1,87);
rx_enc(18)=rx_data_matrix(2,71);
rx_enc(19)=rx_data_matrix(3,55);
rx_enc(20)=rx_data_matrix(4,39);
rx_enc(21)=rx_data_matrix(5,24);
rx_enc(22)=rx_data_matrix(6,8);
rx_enc(23)=rx_data_matrix(7,106);
rx_enc(24)=rx_data_matrix(8,90);
rx_enc(25)=rx_data_matrix(1,73);
rx_enc(26)=rx_data_matrix(2,57);
rx_enc(27)=rx_data_matrix(3,41);
rx_enc(28)=rx_data_matrix(4,25);
rx_enc(29)=rx_data_matrix(5,10);
rx_enc(30)=rx_data_matrix(6,108);
rx_enc(31)=rx_data_matrix(7,92);
```

```
rx_enc(32)=rx_data_matrix(8,76);
rx_enc(33)=rx_data_matrix(1,59);
rx_enc(34)=rx_data_matrix(2,43);
rx_enc(35)=rx_data_matrix(3,27);
rx_enc(36)=rx_data_matrix(4,11);
rx_enc(37)=rx_data_matrix(5,110);
rx_enc(38)=rx_data_matrix(6,94);
rx_enc(39)=rx_data_matrix(7,78);
rx_enc(40)=rx_data_matrix(8,62);
rx_enc(41)=rx_data_matrix(1,45);
rx_enc(42)=rx_data_matrix(2,29);
rx_enc(43)=rx_data_matrix(3,13);
rx_enc(44)=rx_data_matrix(4,111);
rx_enc(45)=rx_data_matrix(5,96);
rx_enc(46)=rx_data_matrix(6,80);
rx_enc(47)=rx_data_matrix(7,64);
rx_enc(48)=rx_data_matrix(8,48);
rx_enc(49)=rx_data_matrix(1,31);
rx_enc(50)=rx_data_matrix(2,15);
rx_enc(51)=rx_data_matrix(3,113);
rx_enc(52)=rx_data_matrix(4,97);
rx_enc(53)=rx_data_matrix(5,82);
rx_enc(54)=rx_data_matrix(6,66);
rx_enc(55)=rx_data_matrix(7,50);
rx_enc(56)=rx_data_matrix(8,34);
rx_enc(57)=rx_data_matrix(1,17);
rx_enc(58)=rx_data_matrix(2,1);
rx_enc(59)=rx_data_matrix(3,99);
rx_enc(60)=rx_data_matrix(4,83);
rx_enc(61)=rx_data_matrix(5,68);
rx_enc(62)=rx_data_matrix(6,52);
rx_enc(63)=rx_data_matrix(7,36);
rx_enc(64)=rx_data_matrix(8,20);
rx_enc(65)=rx_data_matrix(1,3);
rx_enc(66)=rx_data_matrix(2,101);
rx_enc(67)=rx_data_matrix(3,85);
rx_enc(68)=rx_data_matrix(4,69);
rx_enc(69)=rx_data_matrix(5,54);
rx_enc(70)=rx_data_matrix(6,38);
rx_enc(71)=rx_data_matrix(7,22);
rx_enc(72)=rx_data_matrix(8,6);
rx_enc(73)=rx_data_matrix(1,103);
rx_enc(74)=rx_data_matrix(2,87);
rx_enc(75)=rx_data_matrix(3,71);
rx_enc(76)=rx_data_matrix(4,55);
rx_enc(77)=rx_data_matrix(5,40);
rx_enc(78)=rx_data_matrix(6,24);
rx_enc(79)=rx_data_matrix(7,8);
rx_enc(80)=rx_data_matrix(8,106);
rx_enc(81)=rx_data_matrix(1,89);
rx_enc(82)=rx_data_matrix(2,73);
rx_enc(83)=rx_data_matrix(3,57);
rx_enc(84)=rx_data_matrix(4,41);
rx_enc(85)=rx_data_matrix(5,26);
rx_enc(86)=rx_data_matrix(6,10);
rx_enc(87)=rx_data_matrix(7,108);
rx_enc(88)=rx_data_matrix(8,92);
rx_enc(89)=rx_data_matrix(1,75);
rx_enc(90)=rx_data_matrix(2,59);
rx_enc(91)=rx_data_matrix(3,43);
rx_enc(92)=rx_data_matrix(4,27);
rx_enc(93)=rx_data_matrix(5,12);
rx_enc(94)=rx_data_matrix(6,110);
rx_enc(95)=rx_data_matrix(7,94);
```

```
rx_enc(96)=rx_data_matrix(8,78);
rx_enc(97)=rx_data_matrix(1,61);
rx_enc(98)=rx_data_matrix(2,45);
rx_enc(99)=rx_data_matrix(3,29);
rx_enc(100)=rx_data_matrix(4,13);
rx_enc(101)=rx_data_matrix(5,112);
rx_enc(102)=rx_data_matrix(6,96);
rx_enc(103)=rx_data_matrix(7,80);
rx_enc(104)=rx_data_matrix(8,64);
rx_enc(105)=rx_data_matrix(1,47);
rx_enc(106)=rx_data_matrix(2,31);
rx_enc(107)=rx_data_matrix(3,15);
rx_enc(108)=rx_data_matrix(4,113);
rx_enc(109)=rx_data_matrix(5,98);
rx_enc(110)=rx_data_matrix(6,82);
rx_enc(111)=rx_data_matrix(7,66);
rx_enc(112)=rx_data_matrix(8,50);
rx_enc(113)=rx_data_matrix(1,33);
rx_enc(114)=rx_data_matrix(2,17);
rx_enc(115)=rx_data_matrix(3,1);
rx_enc(116)=rx_data_matrix(4,99);
rx_enc(117)=rx_data_matrix(5,84);
rx_enc(118)=rx_data_matrix(6,68);
rx_enc(119)=rx_data_matrix(7,52);
rx_enc(120)=rx_data_matrix(8,36);
rx_enc(121)=rx_data_matrix(1,19);
rx_enc(122)=rx_data_matrix(2,3);
rx_enc(123)=rx_data_matrix(3,101);
rx_enc(124)=rx_data_matrix(4,85);
rx_enc(125)=rx_data_matrix(5,70);
rx_enc(126)=rx_data_matrix(6,54);
rx_enc(127)=rx_data_matrix(7,38);
rx_enc(128)=rx_data_matrix(8,22);
rx_enc(129)=rx_data_matrix(1,5);
rx_enc(130)=rx_data_matrix(2,103);
rx_enc(131)=rx_data_matrix(3,87);
rx_enc(132)=rx_data_matrix(4,71);
rx_enc(133)=rx_data_matrix(5,56);
rx_enc(134)=rx_data_matrix(6,40);
rx_enc(135)=rx_data_matrix(7,24);
rx_enc(136)=rx_data_matrix(8,8);
rx_enc(137)=rx_data_matrix(1,105);
rx_enc(138)=rx_data_matrix(2,89);
rx_enc(139)=rx_data_matrix(3,73);
rx_enc(140)=rx_data_matrix(4,57);
rx_enc(141)=rx_data_matrix(5,42);
rx_enc(142)=rx_data_matrix(6,26);
rx_enc(143)=rx_data_matrix(7,10);
rx_enc(144)=rx_data_matrix(8,108);
rx_enc(145)=rx_data_matrix(1,91);
rx_enc(146)=rx_data_matrix(2,75);
rx_enc(147)=rx_data_matrix(3,59);
rx_enc(148)=rx_data_matrix(4,43);
rx_enc(149)=rx_data_matrix(5,28);
rx_enc(150)=rx_data_matrix(6,12);
rx_enc(151)=rx_data_matrix(7,110);
rx_enc(152)=rx_data_matrix(8,94);
rx_enc(153)=rx_data_matrix(1,77);
rx_enc(154)=rx_data_matrix(2,61);
rx_enc(155)=rx_data_matrix(3,45);
rx_enc(156)=rx_data_matrix(4,29);
rx_enc(157)=rx_data_matrix(5,14);
rx_enc(158)=rx_data_matrix(6,112);
rx_enc(159)=rx_data_matrix(7,96);
```

```
rx_enc(160)=rx_data_matrix(8,80);
rx_enc(161)=rx_data_matrix(1,63);
rx_enc(162)=rx_data_matrix(2,47);
rx_enc(163)=rx_data_matrix(3,31);
rx_enc(164)=rx_data_matrix(4,15);
rx_enc(165)=rx_data_matrix(5,114);
rx_enc(166)=rx_data_matrix(6,98);
rx_enc(167)=rx_data_matrix(7,82);
rx_enc(168)=rx_data_matrix(8,66);
rx_enc(169)=rx_data_matrix(1,49);
rx_enc(170)=rx_data_matrix(2,33);
rx_enc(171)=rx_data_matrix(3,17);
rx_enc(172)=rx_data_matrix(4,1);
rx_enc(173)=rx_data_matrix(5,100);
rx_enc(174)=rx_data_matrix(6,84);
rx_enc(175)=rx_data_matrix(7,68);
rx_enc(176)=rx_data_matrix(8,52);
rx_enc(177)=rx_data_matrix(1,35);
rx_enc(178)=rx_data_matrix(2,19);
rx_enc(179)=rx_data_matrix(3,3);
rx_enc(180)=rx_data_matrix(4,101);
rx_enc(181)=rx_data_matrix(5,86);
rx_enc(182)=rx_data_matrix(6,70);
rx_enc(183)=rx_data_matrix(7,54);
rx_enc(184)=rx_data_matrix(8,38);
rx_enc(185)=rx_data_matrix(1,21);
rx_enc(186)=rx_data_matrix(2,5);
rx_enc(187)=rx_data_matrix(3,103);
rx_enc(188)=rx_data_matrix(4,87);
rx_enc(189)=rx_data_matrix(5,72);
rx_enc(190)=rx_data_matrix(6,56);
rx_enc(191)=rx_data_matrix(7,40);
rx_enc(192)=rx_data_matrix(8,24);
rx_enc(193)=rx_data_matrix(1,7);
rx_enc(194)=rx_data_matrix(2,105);
rx_enc(195)=rx_data_matrix(3,89);
rx_enc(196)=rx_data_matrix(4,73);
rx_enc(197)=rx_data_matrix(5,58);
rx_enc(198)=rx_data_matrix(6,42);
rx_enc(199)=rx_data_matrix(7,26);
rx_enc(200)=rx_data_matrix(8,10);
rx_enc(201)=rx_data_matrix(1,107);
rx_enc(202)=rx_data_matrix(2,91);
rx_enc(203)=rx_data_matrix(3,75);
rx_enc(204)=rx_data_matrix(4,59);
rx_enc(205)=rx_data_matrix(5,44);
rx_enc(206)=rx_data_matrix(6,28);
rx_enc(207)=rx_data_matrix(7,12);
rx_enc(208)=rx_data_matrix(8,110);
rx_enc(209)=rx_data_matrix(1,93);
rx_enc(210)=rx_data_matrix(2,77);
rx_enc(211)=rx_data_matrix(3,61);
rx_enc(212)=rx_data_matrix(4,45);
rx_enc(213)=rx_data_matrix(5,30);
rx_enc(214)=rx_data_matrix(6,14);
rx_enc(215)=rx_data_matrix(7,112);
rx_enc(216)=rx_data_matrix(8,96);
rx_enc(217)=rx_data_matrix(1,79);
rx_enc(218)=rx_data_matrix(2,63);
rx_enc(219)=rx_data_matrix(3,47);
rx_enc(220)=rx_data_matrix(4,31);
rx_enc(221)=rx_data_matrix(5,16);
rx_enc(222)=rx_data_matrix(6,114);
rx_enc(223)=rx_data_matrix(7,98);
rx_enc(224)=rx_data_matrix(8,82);
rx_enc(225)=rx_data_matrix(1,65);
rx_enc(226)=rx_data_matrix(2,49);
rx_enc(227)=rx_data_matrix(3,33);
rx_enc(228)=rx_data_matrix(4,17);
rx_enc(229)=rx_data_matrix(5,2);
rx_enc(230)=rx_data_matrix(6,100);
rx_enc(231)=rx_data_matrix(7,84);
rx_enc(232)=rx_data_matrix(8,68);
rx_enc(233)=rx_data_matrix(1,51);
rx_enc(234)=rx_data_matrix(2,35);
rx_enc(235)=rx_data_matrix(3,19);
rx_enc(236)=rx_data_matrix(4,3);
rx_enc(237)=rx_data_matrix(5,102);
rx_enc(238)=rx_data_matrix(6,86);
rx_enc(239)=rx_data_matrix(7,70);
rx_enc(240)=rx_data_matrix(8,54);
rx_enc(241)=rx_data_matrix(1,37);
rx_enc(242)=rx_data_matrix(2,21);
rx_enc(243)=rx_data_matrix(3,5);
rx_enc(244)=rx_data_matrix(4,103);
rx_enc(245)=rx_data_matrix(5,88);
rx_enc(246)=rx_data_matrix(6,72);
rx_enc(247)=rx_data_matrix(7,56);
rx_enc(248)=rx_data_matrix(8,40);
rx_enc(249)=rx_data_matrix(1,23);
rx_enc(250)=rx_data_matrix(2,7);
rx_enc(251)=rx_data_matrix(3,105);
rx_enc(252)=rx_data_matrix(4,89);
rx_enc(253)=rx_data_matrix(5,74);
rx_enc(254)=rx_data_matrix(6,58);
rx_enc(255)=rx_data_matrix(7,42);
rx_enc(256)=rx_data_matrix(8,26);
rx_enc(257)=rx_data_matrix(1,9);
rx_enc(258)=rx_data_matrix(2,107);
rx_enc(259)=rx_data_matrix(3,91);
rx_enc(260)=rx_data_matrix(4,75);
rx_enc(261)=rx_data_matrix(5,60);
rx_enc(262)=rx_data_matrix(6,44);
rx_enc(263)=rx_data_matrix(7,28);
rx_enc(264)=rx_data_matrix(8,12);
rx_enc(265)=rx_data_matrix(1,109);
rx_enc(266)=rx_data_matrix(2,93);
rx_enc(267)=rx_data_matrix(3,77);
rx_enc(268)=rx_data_matrix(4,61);
rx_enc(269)=rx_data_matrix(5,46);
rx_enc(270)=rx_data_matrix(6,30);
rx_enc(271)=rx_data_matrix(7,14);
rx_enc(272)=rx_data_matrix(8,112);
rx_enc(273)=rx_data_matrix(1,95);
rx_enc(274)=rx_data_matrix(2,79);
rx_enc(275)=rx_data_matrix(3,63);
rx_enc(276)=rx_data_matrix(4,47);
rx_enc(277)=rx_data_matrix(5,32);
rx_enc(278)=rx_data_matrix(6,16);
rx_enc(279)=rx_data_matrix(7,114);
rx_enc(280)=rx_data_matrix(8,98);
rx_enc(281)=rx_data_matrix(1,81);
rx_enc(282)=rx_data_matrix(2,65);
rx_enc(283)=rx_data_matrix(3,49);
rx_enc(284)=rx_data_matrix(4,33);
rx_enc(285)=rx_data_matrix(5,18);
rx_enc(286)=rx_data_matrix(6,2);
rx_enc(287)=rx_data_matrix(7,100);
```

```
rx_enc(288)=rx_data_matrix(8,84);
rx_enc(289)=rx_data_matrix(1,67);
rx_enc(290)=rx_data_matrix(2,51);
rx_enc(291)=rx_data_matrix(3,35);
rx_enc(292)=rx_data_matrix(4,19);
rx_enc(293)=rx_data_matrix(5,4);
rx_enc(294)=rx_data_matrix(6,102);
rx_enc(295)=rx_data_matrix(7,86);
rx_enc(296)=rx_data_matrix(8,70);
rx_enc(297)=rx_data_matrix(1,53);
rx_enc(298)=rx_data_matrix(2,37);
rx_enc(299)=rx_data_matrix(3,21);
rx_enc(300)=rx_data_matrix(4,5);
rx_enc(301)=rx_data_matrix(5,104);
rx_enc(302)=rx_data_matrix(6,88);
rx_enc(303)=rx_data_matrix(7,72);
rx_enc(304)=rx_data_matrix(8,56);
rx_enc(305)=rx_data_matrix(1,39);
rx_enc(306)=rx_data_matrix(2,23);
rx_enc(307)=rx_data_matrix(3,7);
rx_enc(308)=rx_data_matrix(4,105);
rx_enc(309)=rx_data_matrix(5,90);
rx_enc(310)=rx_data_matrix(6,74);
rx_enc(311)=rx_data_matrix(7,58);
rx_enc(312)=rx_data_matrix(8,42);
rx_enc(313)=rx_data_matrix(1,25);
rx_enc(314)=rx_data_matrix(2,9);
rx_enc(315)=rx_data_matrix(3,107);
rx_enc(316)=rx_data_matrix(4,91);
rx_enc(317)=rx_data_matrix(5,76);
rx_enc(318)=rx_data_matrix(6,60);
rx_enc(319)=rx_data_matrix(7,44);
rx_enc(320)=rx_data_matrix(8,28);
rx_enc(321)=rx_data_matrix(1,11);
rx_enc(322)=rx_data_matrix(2,109);
rx_enc(323)=rx_data_matrix(3,93);
rx_enc(324)=rx_data_matrix(4,77);
rx_enc(325)=rx_data_matrix(5,62);
rx_enc(326)=rx_data_matrix(6,46);
rx_enc(327)=rx_data_matrix(7,30);
rx_enc(328)=rx_data_matrix(8,14);
rx_enc(329)=rx_data_matrix(1,111);
rx_enc(330)=rx_data_matrix(2,95);
rx_enc(331)=rx_data_matrix(3,79);
rx_enc(332)=rx_data_matrix(4,63);
rx_enc(333)=rx_data_matrix(5,48);
rx_enc(334)=rx_data_matrix(6,32);
rx_enc(335)=rx_data_matrix(7,16);
rx_enc(336)=rx_data_matrix(8,114);
rx_enc(337)=rx_data_matrix(1,97);
rx_enc(338)=rx_data_matrix(2,81);
rx_enc(339)=rx_data_matrix(3,65);
rx_enc(340)=rx_data_matrix(4,49);
rx_enc(341)=rx_data_matrix(5,34);
rx_enc(342)=rx_data_matrix(6,18);
rx_enc(343)=rx_data_matrix(7,2);
rx_enc(344)=rx_data_matrix(8,100);
rx_enc(345)=rx_data_matrix(1,83);
rx_enc(346)=rx_data_matrix(2,67);
rx_enc(347)=rx_data_matrix(3,51);
rx_enc(348)=rx_data_matrix(4,35);
rx_enc(349)=rx_data_matrix(5,20);
rx_enc(350)=rx_data_matrix(6,4);
rx_enc(351)=rx_data_matrix(7,102);
rx_enc(352)=rx_data_matrix(8,86);
rx_enc(353)=rx_data_matrix(1,69);
rx_enc(354)=rx_data_matrix(2,53);
rx_enc(355)=rx_data_matrix(3,37);
rx_enc(356)=rx_data_matrix(4,21);
rx_enc(357)=rx_data_matrix(5,6);
rx_enc(358)=rx_data_matrix(6,104);
rx_enc(359)=rx_data_matrix(7,88);
rx_enc(360)=rx_data_matrix(8,72);
rx_enc(361)=rx_data_matrix(1,55);
rx_enc(362)=rx_data_matrix(2,39);
rx_enc(363)=rx_data_matrix(3,23);
rx_enc(364)=rx_data_matrix(4,7);
rx_enc(365)=rx_data_matrix(5,106);
rx_enc(366)=rx_data_matrix(6,90);
rx_enc(367)=rx_data_matrix(7,74);
rx_enc(368)=rx_data_matrix(8,58);
rx_enc(369)=rx_data_matrix(1,41);
rx_enc(370)=rx_data_matrix(2,25);
rx_enc(371)=rx_data_matrix(3,9);
rx_enc(372)=rx_data_matrix(4,107);
rx_enc(373)=rx_data_matrix(5,92);
rx_enc(374)=rx_data_matrix(6,76);
rx_enc(375)=rx_data_matrix(7,60);
rx_enc(376)=rx_data_matrix(8,44);
rx_enc(377)=rx_data_matrix(1,27);
rx_enc(378)=rx_data_matrix(2,11);
rx_enc(379)=rx_data_matrix(3,109);
rx_enc(380)=rx_data_matrix(4,93);
rx_enc(381)=rx_data_matrix(5,78);
rx_enc(382)=rx_data_matrix(6,62);
rx_enc(383)=rx_data_matrix(7,46);
rx_enc(384)=rx_data_matrix(8,30);
rx_enc(385)=rx_data_matrix(1,13);
rx_enc(386)=rx_data_matrix(2,111);
rx_enc(387)=rx_data_matrix(3,95);
rx_enc(388)=rx_data_matrix(4,79);
rx_enc(389)=rx_data_matrix(5,64);
rx_enc(390)=rx_data_matrix(6,48);
rx_enc(391)=rx_data_matrix(7,32);
rx_enc(392)=rx_data_matrix(8,16);
rx_enc(393)=rx_data_matrix(1,113);
rx_enc(394)=rx_data_matrix(2,97);
rx_enc(395)=rx_data_matrix(3,81);
rx_enc(396)=rx_data_matrix(4,65);
rx_enc(397)=rx_data_matrix(5,50);
rx_enc(398)=rx_data_matrix(6,34);
rx_enc(399)=rx_data_matrix(7,18);
rx_enc(400)=rx_data_matrix(8,2);
rx_enc(401)=rx_data_matrix(1,99);
rx_enc(402)=rx_data_matrix(2,83);
rx_enc(403)=rx_data_matrix(3,67);
rx_enc(404)=rx_data_matrix(4,51);
rx_enc(405)=rx_data_matrix(5,36);
rx_enc(406)=rx_data_matrix(6,20);
rx_enc(407)=rx_data_matrix(7,4);
rx_enc(408)=rx_data_matrix(8,102);
rx_enc(409)=rx_data_matrix(1,85);
rx_enc(410)=rx_data_matrix(2,69);
rx_enc(411)=rx_data_matrix(3,53);
rx_enc(412)=rx_data_matrix(4,37);
rx_enc(413)=rx_data_matrix(5,22);
rx_enc(414)=rx_data_matrix(6,6);
rx_enc(415)=rx_data_matrix(7,104);
```

```
rx_enc(416)=rx_data_matrix(8,88);
rx_enc(417)=rx_data_matrix(1,71);
rx_enc(418)=rx_data_matrix(2,55);
rx_enc(419)=rx_data_matrix(3,39);
rx_enc(420)=rx_data_matrix(4,23);
rx_enc(421)=rx_data_matrix(5,8);
rx_enc(422)=rx_data_matrix(6,106);
rx_enc(423)=rx_data_matrix(7,90);
rx_enc(424)=rx_data_matrix(8,74);
rx_enc(425)=rx_data_matrix(1,57);
rx_enc(426)=rx_data_matrix(2,41);
rx_enc(427)=rx_data_matrix(3,25);
rx_enc(428)=rx_data_matrix(4,9);
rx_enc(429)=rx_data_matrix(5,108);
rx_enc(430)=rx_data_matrix(6,92);
rx_enc(431)=rx_data_matrix(7,76);
rx_enc(432)=rx_data_matrix(8,60);
rx_enc(433)=rx_data_matrix(1,43);
rx_enc(434)=rx_data_matrix(2,27);
rx_enc(435)=rx_data_matrix(3,11);
rx_enc(436)=rx_data_matrix(4,109);
rx_enc(437)=rx_data_matrix(5,94);
rx_enc(438)=rx_data_matrix(6,78);
rx_enc(439)=rx_data_matrix(7,62);
rx_enc(440)=rx_data_matrix(8,46);
rx_enc(441)=rx_data_matrix(1,29);
rx_enc(442)=rx_data_matrix(2,13);
rx_enc(443)=rx_data_matrix(3,111);
rx_enc(444)=rx_data_matrix(4,95);
rx_enc(445)=rx_data_matrix(5,80);
rx_enc(446)=rx_data_matrix(6,64);
rx_enc(447)=rx_data_matrix(7,48);
rx_enc(448)=rx_data_matrix(8,32);
rx_enc(449)=rx_data_matrix(1,15);
rx_enc(450)=rx_data_matrix(2,113);
rx_enc(451)=rx_data_matrix(3,97);
rx_enc(452)=rx_data_matrix(4,81);
rx_enc(453)=rx_data_matrix(5,66);
rx_enc(454)=rx_data_matrix(6,50);
rx_enc(455)=rx_data_matrix(7,34);
rx_enc(456)=rx_data_matrix(8,18);
```

Source code C.20: `channel_dec.m`

```
function [rx_block,FLAG_SS,PARITY_CHK] = channel_dec(rx_enc)
%
% channel_dec:
%
% SYNTAX:      [rx_block, FLAG_SS, PARITY_CHK] = channel_dec(rx_enc)
%
% INPUT:       rx_enc  A 456 bits long vector contaning the encoded
%                      data sequence as estimated by the SOVA. The
%                      format of the sequency must be according to
%                      the GSM 05.03 encoding scheme
%
% OUTPUT:      rx_block  A 260 bits long vector contaning the final
%                        estimated information data sequence.
%
%              FLAG_SS Indication of correct stop state. Flag is set
%                      to '1' if an error has occured here.
%
%              PARITY_CHK The 3 parity check bit inserted in the
%                         transmitter.
%
% SUB_FUNC:    None
%
% WARNINGS:    None
%
% TEST(S):     Operation tested in conjunction with the channel_enc.m
%              module. Operation proved to be correct.
%
% AUTHOR:   Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: channel_dec.m,v 1.8 1998/02/12 10:53:13 aneks Exp $

L = length(rx_enc);

% TEST INPUT DATA
%
if L ~= 456
  disp(' ')
  disp('Input data sequence size violation. Program terminated.')
  disp(' ')
  break;
end

% SEPARATE DATA IN CLASS I, c1, AND CLASSE II, c2, BITS
% CLASS I BITS ARE DECODED WHILE CLASS II ARE LEFT
% UNCHANGED
%

c1 = rx_enc(1:378);
c2 = rx_enc(379:L);

% INITIALIZE VARIOUS MATRIXES
%
% REMEMBER THE VA DECODER OPERATES ON DI BITS
% HENCE ONLY 378/2 = 189 STATE TRANSITIONS OCCURE
%

START_STATE = 1;
END_STATE = 1;
```

```
STATE = zeros(16,189);
METRIC = zeros(16,2);

NEXT = zeros(16,2);
zeroin = 1;
onein = 9;
for n = 1:2:15,
  NEXT(n,:) = [zeroin onein];
  NEXT(n+1,:) = NEXT(n,:);
  zeroin = zeroin + 1;
  onein = onein + 1;
end

PREVIOUS = zeros(16,2);
offset = 0;
for n = 1:8,
  PREVIOUS(n,:) = [n+offset n+offset+1];
  offset = offset + 1;
end
PREVIOUS = [ PREVIOUS(1:8,:) ; PREVIOUS(1:8,:)];

% SETUP OF DIBIT DECODER TABLE. THE BINARY DIBITS ARE
% HERE REPRESENTED USING DECIMAL NUMBER, I.E. THE DIBIT
% 00 IS REPRESENTED AS 0 AND THE DIBIT 11 AS 3
%
% THE TABEL IS SETUP SO THAT THE CALL DIBIT(X,Y) RETURNS
% THE DIBIT SYMBOL RESULTING FROM A STATE TRANSITION FROM
% STATE X TO STATE Y
%

DIBIT = [  0 NaN NaN NaN NaN NaN NaN NaN   3 NaN NaN NaN NaN NaN NaN NaN;
           3 NaN NaN NaN NaN NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN;
         NaN   3 NaN NaN NaN NaN NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN;
         NaN   0 NaN NaN NaN NaN NaN NaN NaN   3 NaN NaN NaN NaN NaN NaN;
         NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   3 NaN NaN NaN NaN NaN;
         NaN NaN   3 NaN NaN NaN NaN NaN NaN NaN   0 NaN NaN NaN NaN NaN;
         NaN NaN NaN   3 NaN NaN NaN NaN NaN NaN NaN   0 NaN NaN NaN NaN;
         NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   3 NaN NaN NaN NaN;
         NaN NaN NaN NaN   1 NaN NaN NaN NaN NaN NaN NaN   2 NaN NaN NaN;
         NaN NaN NaN NaN   2 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN;
         NaN NaN NaN NaN NaN   2 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN;
         NaN NaN NaN NaN NaN   1 NaN NaN NaN NaN NaN NaN NaN   2 NaN NaN;
         NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN NaN NaN NaN NaN   2 NaN;
         NaN NaN NaN NaN NaN NaN   2 NaN NaN NaN NaN NaN NaN NaN   1 NaN;
         NaN NaN NaN NaN NaN NaN NaN   2 NaN NaN NaN NaN NaN NaN NaN   1;
         NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN NaN NaN NaN NaN   2];

% SETUP OF BIT DECODER TABLE.
% THE TABEL IS SETUP SO THAT THE CALL BIT(X,Y) RETURNS
% THE DECODED BIT RESULTING FROM A STATE TRANSITION FROM
% STATE X TO STATE Y
%

BIT = [  0 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN NaN NaN NaN NaN;
         0 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN NaN NaN NaN NaN;
       NaN   0 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN NaN NaN NaN;
       NaN   0 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN NaN NaN NaN;
       NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN NaN NaN;
       NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN NaN NaN;
       NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN NaN;
       NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN NaN;
       NaN NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN;
       NaN NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN NaN;
       NaN NaN NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN;
       NaN NaN NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   1 NaN NaN;
       NaN NaN NaN NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   1 NaN;
       NaN NaN NaN NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   1 NaN;
       NaN NaN NaN NaN NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   1;
       NaN NaN NaN NaN NaN NaN NaN   0 NaN NaN NaN NaN NaN NaN NaN   1];

% STARTUP METRIK CALCULATIONS.
%
% THIS IS TO REDUCE THE NUMBER OF CALCULATIONS REQUIRED
% AND IT IT RUN ONLY FOR THE FIRST 4 DIBIT PAIRS
%

VISITED_STATES = START_STATE;
for n = 0:3,

  rx_DIBITXy = cl(2*n + 1);
  rx_DIBITxY = cl(2*n + 1 + 1);

  for k = 1:length(VISITED_STATES),

    PRESENT_STATE = VISITED_STATES(k);

    next_state_0 = NEXT(PRESENT_STATE,1);
    next_state_1 = NEXT(PRESENT_STATE,2);

    symbol_0 = DIBIT(PRESENT_STATE,next_state_0);
    symbol_1 = DIBIT(PRESENT_STATE,next_state_1);

    if symbol_0 == 0
      LAMBDA = xor(rx_DIBITXy,0) + xor(rx_DIBITxY,0);
    end
    if symbol_0 == 1
      LAMBDA = xor(rx_DIBITXy,0) + xor(rx_DIBITxY,1);
    end
    if symbol_0 == 2
      LAMBDA = xor(rx_DIBITXy,1) + xor(rx_DIBITxY,0);
    end
    if symbol_0 == 3
      LAMBDA = xor(rx_DIBITXy,1) + xor(rx_DIBITxY,1);
    end

    METRIC(next_state_0,2) = METRIC(PRESENT_STATE,1) + LAMBDA;

    if symbol_1 == 0
      LAMBDA = xor(rx_DIBITXy,0) + xor(rx_DIBITxY,0);
    end
    if symbol_1 == 1
      LAMBDA = xor(rx_DIBITXy,0) + xor(rx_DIBITxY,1);
    end
    if symbol_1 == 2
      LAMBDA = xor(rx_DIBITXy,1) + xor(rx_DIBITxY,0);
    end
    if symbol_1 == 3
      LAMBDA = xor(rx_DIBITXy,1) + xor(rx_DIBITxY,1);
    end

    METRIC(next_state_1,2) = METRIC(PRESENT_STATE,1) + LAMBDA;
    STATE([next_state_0, next_state_1],n + 1) = PRESENT_STATE;

    if k == 1
      PROCESSED = [next_state_0 next_state_1];
    else
      PROCESSED = [PROCESSED next_state_0 next_state_1];
    end
```

```
  end

  VISITED_STATES = PROCESSED;
  METRIC(:,1) = METRIC(:,2);
  METRIC(:,2) = 0;
end


% STARTING THE SECTION WHERE ALL STATES ARE RUN THROUGH
% IN THE METRIC CALCULATIONS. THIS GOES ON FOR THE
% REMAINING DIBITS RECEIVED
%

for n = 4:188,

  rx_DIBITXy = c1(2*n - 1);
  rx_DIBITxY = c1(2*n - 1 + 1);

  for k = 1:16,

    prev_state_1 = PREVIOUS(k,1);
    prev_state_2 = PREVIOUS(k,2);

    symbol_1 = DIBIT(prev_state_1,k);
    symbol_2 = DIBIT(prev_state_2,k);

    if symbol_1 == 0
      LAMBDA_1 = xor(rx_DIBITXy,0) + xor(rx_DIBITxY,0);
    end
    if symbol_1 == 1
      LAMBDA_1 = xor(rx_DIBITXy,0) + xor(rx_DIBITxY,1);
    end
    if symbol_1 == 2
      LAMBDA_1 = xor(rx_DIBITXy,1) + xor(rx_DIBITxY,0);
    end
    if symbol_1 == 3
      LAMBDA_1 = xor(rx_DIBITXy,1) + xor(rx_DIBITxY,1);
    end

    if symbol_2 == 0
      LAMBDA_2 = xor(rx_DIBITXy,0) + xor(rx_DIBITxY,0);
    end
    if symbol_2 == 1
      LAMBDA_2 = xor(rx_DIBITXy,0) + xor(rx_DIBITxY,1);
    end
    if symbol_2 == 2
      LAMBDA_2 = xor(rx_DIBITXy,1) + xor(rx_DIBITxY,0);
    end
    if symbol_2 == 3
      LAMBDA_2 = xor(rx_DIBITXy,1) + xor(rx_DIBITxY,1);
    end

    METRIC_1 = METRIC(prev_state_1,1) + LAMBDA_1;
    METRIC_2 = METRIC(prev_state_2,1) + LAMBDA_2;

    if METRIC_1 < METRIC_2
      METRIC(k,2) = METRIC_1;
      STATE(k,n-1) = prev_state_1;
    else
      METRIC(k,2) = METRIC_2;
      STATE(k,n-1) = prev_state_2;
    end
  end

  METRIC(:,1) = METRIC(:,2);
  METRIC(:,2) = 0;

end

% STARTING BACKTRACKING TO DETERMINE THE MOST
% PROBABLE STATE TRANSITION SEQUENCE
%

STATE_SEQ = zeros(1,189);

[STOP_METRIC, STOP_STATE] = min(METRIC(:,1));

if STOP_STATE == END_STATE
  FLAG_SS = 0;
else
  FLAG_SS = 1;
end

STATE_SEQ(189) = STOP_STATE;

for n = 188:-1:1,
  STATE_SEQ(n) = STATE(STATE_SEQ(n+1), n-1);
end

STATE_SEQ = [START_STATE STATE_SEQ];

% RESOLVING THE CORRESPONDING BIT SEQUENCE
%

for n = 1:length(STATE_SEQ)-1,
  DECONV_DATA(n) = BIT(STATE_SEQ(n), STATE_SEQ(n+1));
end

% SEPARATING THE DATA ACCORDING TO THE ENCODING
% RESULTING FROM THE TRANSMITTER.
%

DATA_Ia = DECONV_DATA(1:50);
PARITY_CHK = DECONV_DATA(51:53);
DATA_Ib = DECONV_DATA(54:185);
TAIL_BITS = DECONV_DATA(186:189);

rx_block = [DATA_Ia DATA_Ib c2];
```

Source code C.21: `channel_simulator.m`

```
function [ r ] = channel_simulator(I,Q,OSR)
% CHANNEL_SIMULATOR:
%           This function is intended as an skeleton for channel
%           simulator implementations, and is not to be considderd
%           as an actual channel simulator. It does however provide
%           a mean for making the GSMsim package produce detection
%           errors. Substitute this function with userdefined
%           functions.
%
% SYNTAX:   channel_simulator(I,Q,OSR)
%
% INPUT:    I:      The inphase signal as produced by the modulator.
%           Q:      The quadrature signal as it is produced by the
%                   modulator .
%           OSR:    The over sampling ratio, defined as f_s/r_b. This
%                   parameter is not used in the incuded channel_simulator
%                   function, but is passed to the function for future
%                   use.
%
% OUTPUT:   r:      The received signal, as predicted by the channel
%                   simulator.
%
% WARNINGS: Do not use this function for scientific purposes.
%
% AUTHOR:   Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: channel_simulator.m,v 1.6 1998/02/12 10:56:13 aneks Exp $

% THE CHANNEL SIMULATOR INCLUDED IN THE GSMsim PACKAGE ONLY ADD
% NOISE, AND SHOULD  NOT  BE USED FOR SCIENTIFIC PURPOSES.
%
% SUBSTITUTE THE NEXT LINES WITH A LINE CONTAINING A CHANNEL SIMULATION.
% HERE WE SIMPLY ADD NOISE OF VARIANCE Var TO THE MODULATED SIGNAL. NOTE
% THAT THIS NOISE IS ONLY BANDLIMITED BY THE SAMPLERATE.
%
Var=0.02;
FACTOR=sqrt(Var);
samples=length(I);
random=randn(1,2*samples);
r= I + j.*Q + random(1:samples).*FACTOR;
r= r + random(samples+1:2*samples).*FACTOR.*j;
```

Source code C.22: `GSMsim_demo.m`

```
function [ ] = GSMsim_demo(LOOPS,Lh)
% GSMSIM_DEMO:
%           This function demonstrates the function of the GSMsim
%           package. Use this file as a starting point for building
%           your own simulations.
%
% SYNTAX:   GSMsim_demo(LOOPS,Lh)
%
% INPUT:    LOOPS:  The number of loops that the demo is to run,
%                   each loop contain 10 burst.
%           Lh:     The length of the channel impulse response
%                   minus one.
%
% OUTPUT:   None, but on screen.
%
% WARNINGS: Do not expect this example to be more than exactly that,
%           an example. This example is NOT scientifically correct.
%
% AUTHOR:   Jan H. Mikkelsen / Arne Norre Ekstrøm
% EMAIL:    hmi@kom.auc.dk / aneks@kom.auc.dk
%
% $Id: GSMsim_demo.m,v 1.15 1998/10/01 10:19:04 hmi Exp $

tTotal=clock;

% THERE HAS NOT BEEN ANY ERRORS, YET...
%
B_ERRS=0;

% gsm_set MUST BE RUN PRIOR TO ANY SIMULATIONS, SINCE IT DOES SETUP
% OF VALUES NEEDED FOR OPERATION OF THE PACKAGE.
%
gsm_set;

% PREPARE THE TABLES NEEDED BY THE VITERBI ALGORITHM.
%
[ SYMBOLS , PREVIOUS , NEXT , START , STOPS ] = viterbi_init(Lh);

% THIS IS THE SIMULATION LOOP, OVER THE BURSTS
%
A_Loop=0;
for Loop=1:LOOPS

  % GET THE TIME
  %
  t0=clock;

  for n=1:10;
    % GET DATA FOR A BURST
    %
    tx_data = data_gen(INIT_L);

    % THIS IS ALL THAT IS NEEDED FOR MODULATING A GSM BUST, IN THE FORMAT
    % USED IN GSMsim. THE CALL INCLUDES GENERATION AND MODULATTION OF DATA.
    %
    [ burst , I , Q ] = gsm_mod(Tb,OSR,BT,tx_data,TRAINING);

    % AT THIS POINT WE RUN THE CHANNEL SIMULATION. NOTE, THAT THE CHANNEL
    % INCLUDES TRANSMITTER FORNT END, AND RECEIVER FRONT END. THE CHANNEL
    % SELECTION IS BY NATURE INCLUDED IN THE RECEIVER FRONT END.
```

Source code C.23: GSMsim_demo_2.m

```
function [] = GSMsim_demo_2(NumberOfBlocks,Lh,LogName)
% CSMSIM DEMO:
%           This demonstrates the function of the GSMsim
%           package. Use this file as a starting point for building
%           your own simulations.
%
% SYNTAX:   GSMsim demo 2(NumberOfBlocks,Lh,logname)
%
% INPUT:    NumberOfBlocks:
%                   The number of GSM code blocks to process, a block
%                   corresponds to four GSM bursts.
%
%           Lh:     The length of the channel impulse response
%                   minus one.
%           LogName:
%                   The basename of the file to which the simulation log is
%                   to be written. The simulation log is handy for
%                   evaluating the convergence og a simulation.
%
% OUTPUT:   To a file called: logname.NumberOfBlocks.Lh
%           Simulation statistics are constantly echoed to the screen for
%           easy reference.
%
% WARNINGS: Do not expect this example to be more than exactly that,
%           an example. This example is NOT scientifically correct.
%
% AUTHOR:   Arne Norre Ekstrøm / Jan H. Mikkelsen
% EMAIL:    aneks@kom.auc.dk / hmi@kom.auc.dk
%
% $Id: GSMsim_demo_2.m,v 1.6 1998/10/01 10:18:47 hmi Exp $

% This is an aid for the final screen report.
%
tTotal=clock;

% Create the name of the log file for future reference.
%
LogFile=[LogName ' ' num2str(NumberOfBlocks,'%9d') ' ' ];
LogFile=[LogFile num2str(Lh,'%3d') '.sim'];

% Print header to the log file, abort if file allready exist.
%
fid=fopen(LogFile,'r');
if fid== 1
  LogFID=fopen(LogFile,'w');
  fprintf(LogFID,'%% GSMsim_demo_2 - simulation log.\n');
  fprintf(LogFID,'%%\n');
  fprintf(LogFID,'%% Settings:       NumberOfBlocks=%d\n',NumberOfBlocks);
  fprintf(LogFID,'%%                                Lh=%d\n',Lh);
  fprintf(LogFID,'%%\n');
  fprintf(LogFID,'%% BlockNumber');
  fprintf(LogFID,'%% B_ERRS_Ia B_ERRS_Ib B_ERRS_II B_ERRS_II_CHEAT\n');
  fprintf(LogFID,'%%\n');
  fclose(LogFID);
else
  error('The logfile allready exists, aborting simulation...');
end

% There has, not yet, been observed any errors.
%
      % THE CHANNEL SIMULATOR INCLUDED IN THE GSMsim PACKAGE ONLY ADD
      % NOISE, AND SHOULD _NOT_ BE USED FOR SCIENTIFIC PURPOSES.
      %
      %r=channel_simulator(I,Q,OSR);
      r = I + j*Q;

      % RUN THE MATCHED FILTER, IT IS RESPONSIBLE FOR FILTERING SYNCRONIZATION
      % AND RETRIEVAL OF THE CHANNEL CHARACTERISTICS.
      %
      [Y, Rhh] = mafi(r,Lh,T_SEQ,OSR);

      % HAVING PREPARED THE PRECALCULATABLE PART OF THE VITERBI
      % ALGORITHM, IT IS CALLED PASSING THE OBTAINED INFORMATION ALONG WITH
      % THE RECEIVED SIGNAL, AND THE ESTIMATED AUTOCORRELATION FUNCTION.
      %
      rx burst = viterbi detector(SYMBOLS,NEXT,PREVIOUS,START,STOPS,Y,Rhh);

      % RUN THE DeMUX
      %
      rx_data=DeMUX(rx_burst);

      % COUNT THE ERRORS
      %
      B_ERRS=B_ERRS+sum(xor(tx_data,rx_data));

    end

    % FIND THE LOOPTIME
    %
    elapsed=etime(clock,t0);

    % FIND AVERAGE LOOP TIME
    %
    A Loop=(A Loop+elapsed)/2;

    % FIND REMAINING TIME
    %
    Remain = (LOOPS-Loop)*A_Loop;

    % UPDATE THE DISPLAY
    %
    fprintf(1,'\r');
    fprintf(1,'Loop: %d, Average Loop Time: %2.1f seconds',Loop,A_Loop);
    fprintf(1,', Remaining: %2.1f seconds ',Remain);

end

Ttime=etime(clock,tTotal);
BURSTS=LOOPS*10;
fprintf(1,'\n%d Bursts processed in %6.1f Seconds.\n',BURSTS,Ttime);
fprintf(1,'Used %2.1f seconds per burst\n',Ttime/BURSTS);
fprintf(1,'There were %d Bit Errors\n',B ERRS);
BER=(B ERRS*100)/(BURSTS*148);
fprintf(1,'This equals %2.1f Percent of the checked bits.\n',BER);
```

```
B_ERRS_Ia=0;
B_ERRS_Ib=0;
B_ERRS_II=0;
B_ERRS_II_CHEAT=0;

% gsm_set MUST BE RUN PRIOR TO ANY SIMULATIONS, SINCE IT DOES SETUP
% OF VALUES NEEDED FOR OPERATION OF THE PACKAGE.
%
gsm_set;

% PREPARE THE TABLES NEEDED BY THE VITERBI ALGORITHM.
%
[ SYMBOLS , PREVIOUS , NEXT , START , STOPS ] = viterbi_init(Lh);

% We need to initialize the interleaving routines, for that we need an so
% called first burst for the interleaver, this burst will not be fully
% received. Nor will bit errors be checked, hence there is no reason for
% encoding it...
%
tx_enc1=round(rand(1,456));

% Now we need a tx_data_matrix to start the deinterleaver thus get data
% for a burst. Bit errors will be checked for in this block.
%
tx_block2=data_gen(260);

% Do channel coding of data
%
tx_enc2=channel_enc(tx_block2);

% Interleave data
%
tx_data_matrix=interleave(tx_enc1,tx_enc2);

% Time goes by, and new become old, thus swap before entry of loop.
%
tx_enc1=tx_enc2;
tx_block1=tx_block2;

% Transmit and receive burst
%
rx_data_matrix1=tx_data_matrix;

% Sliding average time report aid.
%
A_Loop=0;

for N=2:NumberOfBlocks+1,

  % Time report aid.
  %
  t0=clock;

  % Get data for a new datablock, number two is the latest.
  %
  tx_block2=data_gen(260);

  % Do channel coding of data
  %
  tx_enc2=channel_enc(tx_block2);

  % tx_data_matrix contains data for four burst, generated from two blocks.
  %
  tx_data_matrix=interleave(tx_enc1,tx_enc2);

  for n=1:4,

    % THIS IS ALL THAT IS NEEDED FOR MODULATING A GSM BUST, IN THE FORMAT
    % USED IN GSMsim. THE CALL INCLUDES GENERATION AND MODULATTION OF DATA.
    %
    [ tx_burst , I , Q ] = gsm_mod(Tb,OSR,BT,tx_data_matrix(n,:),TRAINING);

    % AT THIS POINT WE RUN THE CHANNEL SIMULATION. NOTE, THAT THE CHANNEL
    % INCLUDES TRANSMITTER FORNT-END, AND RECEIVER FRONT-END. THE CHANNEL
    % SELECTION IS BY NATURE INCLUDED IN THE RECEIVER FRONT END.
    % THE CHANNEL SIMULATOR INCLUDED IN THE GSMsim PACKAGE ONLY ADDS
    % NOISE, AND SHOULD  NOT  BE USED FOR SCIENTIFIC PURPOSES.
    %
    r=channel_simulator(I,Q,OSR);

    % RUN THE MATCHED FILTER, IT IS RESPONSIBLE FOR FILTERING SYNCRONIZATION
    % AND RETRIEVAL OF THE CHANNEL CHARACTERISTICS.
    %
    [Y, Rhh] = mafi(r,Lh,T_SEQ,OSR);

    % HAVING PREPARED THE PRECALCULATABLE PART OF THE VITERBI
    % ALGORITHM, IT IS CALLED PASSING THE OBTAINED INFORMATION ALONG WITH
    % THE RECEIVED SIGNAL, AND THE ESTIMATED AUTOCORRELATION FUNCTION.
    %
    rx_burst = viterbi_detector(SYMBOLS,NEXT,PREVIOUS,START,STOPS,Y,Rhh);

    % RUN THE DeMUX
    %
    rx_data_matrix2(n,:)=DeMUX(rx_burst);

  end

  % This is for bypassing the channel, uncomment to use
  %
  % rx_data_matrix2=tx_data_matrix;

  % A block is regenerated using eight bursts.
  %
  rx_enc=deinterleave( [ rx_data_matrix1 ; rx_data_matrix2 ] );

  % A good cheat/trick is to use all the encoded bits for estimating a type
  % II BER. This estimate is 100% statistically correct!!!
  %
  B_ERRS_II_CHEAT_NEW=sum(xor(tx_enc1,rx_enc));

  % Do channel decoding
  %
  rx_block=channel_dec(rx_enc);

  % Count errors
  %
  B_ERRS_ALL=xor(rx_block,tx_block1);
  B_ERRS_Ia_NEW=sum(B_ERRS_ALL(1:50));
  B_ERRS_Ib_NEW=sum(B_ERRS_ALL(51:182));
  B_ERRS_II_NEW=sum(B_ERRS_ALL(183:260));

  % Update the log file.
  %
  LogFID=fopen(LogFile,'a');
  fprintf(LogFID,'%d %d ',N-1,B_ERRS_Ia_NEW);
  fprintf(LogFID,'%d %d ',B_ERRS_Ib_NEW,B_ERRS_II_NEW);
  fprintf(LogFID,'%d\n',B_ERRS_II_CHEAT_NEW);
  fclose(LogFID);
```

```
% Sum the errors
%
B_ERRS_Ia=B_ERRS_Ia+B_ERRS_Ia_NEW;
B_ERRS_Ib=B_ERRS_Ib+B_ERRS_Ib_NEW;
B_ERRS_II=B_ERRS_II+B_ERRS_II_NEW;
B_ERRS_II_CHEAT=B_ERRS_II_CHEAT+B_ERRS_II_CHEAT_NEW;

% Time goes by, and new become old, thus swap for next loop.
%
rx_data_matrix1=rx_data_matrix2;
tx_enc1=tx_enc2;
tx_block1=tx_block2;

% Find the loop time
%
elapsed=etime(clock,t0);

% Find average loop time
%
A_Loop=(A_Loop+elapsed)/2;

% FIND REMAINING TIME
%
Remain = (NumberOfBlocks-1-N)*A_Loop;

% UPDATE THE DISPLAY
%
fprintf(1,'\r');
fprintf(1,'Block: %d, Average Block Time: %2.1f seconds',N-1,A_Loop);
fprintf(1,', Remaining: %2.1f seconds ',Remain);

end

Ttime=etime(clock,tTotal);
BURSTS=NumberOfBlocks*4;

% Find BER.
%
TypeIaBits=NumberOfBlocks*50;
TypeIaBER=100*B_ERRS_Ia/TypeIaBits;
TypeIbBits=NumberOfBlocks*132;
TypeIbBER=100*B_ERRS_Ib/TypeIbBits;
TypeIIBits=NumberOfBlocks*78;
TypeIIBER=100*B_ERRS_II/TypeIIBits;
TypeIIBitsCHEAT=NumberOfBlocks*456;
TypeIIBER_CHEAT=100*B_ERRS_II_CHEAT/TypeIIBitsCHEAT;

fprintf(1,'\n%d Bursts processed in %6.1f Seconds.\n',BURSTS,Ttime);
fprintf(1,'Used %2.1f seconds per burst\n',Ttime/BURSTS);
fprintf(1,'\n');
fprintf(1,'Type Ia BER: %3.2f\n',TypeIaBER);
fprintf(1,'Type Ib BER: %3.2f\n',TypeIbBER);
fprintf(1,'Type II BER: %3.2f\n',TypeIIBER);
fprintf(1,'Type II BER-CHEAT: %3.2f\n',TypeIIBER_CHEAT);
```

Source code C.24: `gsm_set.m`

```
% GSM_SET:  This script initializes the values needed by the
%           GSMsim package, and must run be for the package to work.
%
% SYNTAX:   gsm_set
%
% INPUT:    None
% OUTPUT:   Configuration variables created in memory, these are:
%           Tb(= 3.692e-6)
%           BT(= 0.3)
%           OSR(= 4)
%           SEED(= 931316785)
%           INIT_L(= 260)
%
% SUB_FUN:  None
%
% WARNINGS: Values can be cleared by other functions, and thus this script
%           should be rerun in each simulation.
%           The random number generator is set to a standard seed value
%           within this script. This causes the random numbers generated
%           matlab to follow a standard pattern.
%
% AUTHOR:   Arne Norre Ekstrøm / Jan H. Mikkelsen
% EMAIL:    aneks@kom.auc.dk / hmi@kom.auc.dk
%
% $Id: gsm_set.m,v 1.11 1997/09/22 11:38:19 aneks Exp $

% GSM 05.05 PARAMETERS
%
Tb = 3.692e-6;
BT = 0.3;
OSR = 4;

% INITIALIZE THE RANDOM NUMBER GENERATOR.
% BY USING THE SAME SEED VALUE IN EVERY SIMULATION, WE GET THE SAME
% SIMULATION DATA, AND THUS SIMULATION RESULTS MAY BE REPRODUCED.
%
SEED = 931316785;
rand('seed',SEED);

% THE NUMBER OF BITS GENERATED BY THE DATA GENERATOR. (data_gen)
%
INIT_L = 114;

% SETUP THE TRAINING SEQUENCE USED FOR BUILDING BURSTS
%
TRAINING = [0 0 1 0 0 1 0 1 1 1 0 0 0 0 1 0 0 0 1 0 0 1 0 1 1 1];

% CONSTRUCT THE MSK MAPPED TRAINING SEQUENCE USING TRAINING.
%
T_SEQ = T_SEQ_gen(TRAINING);
```

Source code C.25: `T_SEQ_gen.m`

```
function T_SEQ = T_SEQ_gen(TRAINING)
%
% T_SEQ_GEN:
%           This function generates the MSK-mapped version of the
%           training sequence used in the GSMsim package.
%
% SYNTAX:   T_SEQ = T_SEQ_gen(TRAINING)
%
% INPUT:    TRAINING: The training sequence represented as bits. (0's and 1's)
%
% OUTPUT:     T_SEQ: A MSK mapped representation of the 26 bits long
%                    training sequence.
%
% SUB_FUNC: None
%
% WARNINGS: First MSK symbol is set to 1. This may be a problem!!!
%
% TEST(S):  Result is verified against those reported by 95gr870T
%
% AUTOR:    Arne Norre Ekstrøm / Jan H. Mikkelsen
% EMAIL:    aneks@kom.auc.dk / hmi@kom.auc.dk
%
% $Id: T_SEQ_gen.m,v 1.5 1998/02/12 10:59:07 aneks Exp $

% TEST TO SEE WETHER THE LENGTH OF tc IS CORRECT.
% IF NOT, THEN ABORT...
%
if length(TRAINING) ~= 26
  error('TRAINING is not of length 26, terminating.');
end

% MAKE A POLAR VERSION OF TRAINING
TRAININGPol=(2.*TRAINING)-1;

% DO DIFFERENTIAL ENCODING OF THE BITS
%
for n=2:length(TRAININGPol)
  a(n)=TRAININGPol(n)*TRAININGPol(n-1);
end

% DO GMSK MAPPING OF POLAR a(n)
%
% THIS IS A CHOICE, AND IT MAY BE WRONG!!!!!!
T_SEQ(1)=1;
for n=2:length(TRAININGPol)
  T_SEQ(n)=j*a(n)*T_SEQ(n-1);
end
```

Source code C.26: `GSMsim_config.m`

```
% This script adds the paths needed for the GSMsim package to
% run correctly. If you change the structure of the directories
% within the GSMsim package then you need to edit this script.
% This script should be executed while standing in the directory
% GSMtop/config.
%
% AUTOR:    Arne Norre Ekstrøm / Jan H. Mikkelsen
% EMAIL:    aneks@kom.auc.dk / hmi@kom.auc.dk
%
% $Id: GSMsim_config.m,v 1.3 1998/02/12 11:00:32 aneks Exp $

% We should have this script in GSMtop/config, now go to GSMtop
cd .. ;

% Find out where GSMtop is located on the disk...
GSMtop=pwd ;

% Now that we have got the information we need for setting up the path,
% then lets set it up.
path(path,[ GSMtop '/config' ]);
path(path,[ GSMtop '/examples' ]);
path(path,[ GSMtop '/utils' ]);
path(path,[ GSMtop '/src/modulator' ]);
path(path,[ GSMtop '/src/demodulator' ]);

% Just to make this fool proof, re enter the config directory
cd config ;
```

Source code C.27: `make_interleave_m.m`

```
% MAKE-INTERLEAVE-M:
%           As the name indicates, this tiny matlab script does construction
%           of the interleave.m-function.
%
% SYNTAX:   make_interleave_m
%
% INPUT:    None.
%
% OUTPUT:   To interleave.tmp
%
% WARNINGS: An existing file will be overwritten.
%
% AUTHOR:   Arne Norre Ekstrøm / Jan H. Mikkelsen
% EMAIL:    aneks@kom.auc.dk / hmi@kom.auc.dk
%
% $Id: make_interleave_m.m,v 1.4 1997/12/18 13:27:27 aneks Exp $

Blocks=1;
BitsInBurst=113;

out=fopen('interleave.tmp','w');

for T=0:3,
  for t=0:BitsInBurst,
    b=mod((57*mod(T,4)+t*32+196*mod(t,2)),456);
    B=floor((T+mod(b,8))/4);
    fprintf(out,'tx data matrix(%d,%d)=tx enc%d(%d);\n',T+1,t+1,B+1,b+1);
  end
end
fclose(out);
```

Source code C.28: `make_deinterleave_m.m`

```
% MAKE-DEINTERLEAVE-M:
%           As the name indicates, this tiny matlab script does construction
%           of the deinterleave.m-function.
%
% SYNTAX:   make_deinterleave_m
%
% INPUT:    None.
%
% OUTPUT:   To deinterleave.tmp
%
% WARNINGS: An existing file will be overwritten.
%
% AUTHOR:   Arne Norre Ekstrøm / Jan H. Mikkelsen
% EMAIL:    aneks@kom.auc.dk / hmi@kom.auc.dk
%
% $Id: make_deinterleave_m.m,v 1.4 1997/12/18 13:26:54 aneks Exp $

BitsInBlock=455;

out=fopen('deinterleave.tmp','w');

B=0;

for b=0:BitsInBlock,
  R=4*B+mod(b,8);
  r=2*mod((49*b),57)+floor(mod(b,8)/4);
  fprintf(out,'rx enc(%d)=rx data matrix(%d,%d);\n',b+1,R+1,r+1);
end

fclose(out);
```