Unequal Error Protection
Turbo Codes

Diploma Thesis

Neele von Deetzen

Issuance date: 30.08.2004
Submission date: 28.02.2005
Supervisor: Prof. Dr.-Ing. W. Henkel
Responsible professor: Prof. Dr.-Ing. K.D. Kammeyer

I affirm that I wrote the Diploma Thesis by my self and that I did not use other than the indicated sources and resources.

Bremen, February 28th, 2005
## Contents

1 Introduction .................................................. 1

2 Convolutional Codes .......................................... 3
   2.1 Structure .................................................. 3
   2.2 Graphical Representation of Convolutional Codes .......... 6
   2.3 Blocking of Convolutional Codes .......................... 7
   2.4 Properties .................................................. 8
   2.5 Decoding ................................................... 10
      2.5.1 Viterbi Algorithm ..................................... 11
      2.5.2 BCJR Algorithm ........................................ 13
      2.5.3 Log-MAP and Max-Log-MAP ............................ 16

3 Parallel Concatenated Convolutional Codes / Turbo Codes ... 18
   3.1 Structure .................................................. 18
   3.2 Properties .................................................. 19
   3.3 Decoding Structure ......................................... 21
   3.4 Decoding Algorithms ....................................... 22

4 An Analysis Tool: The EXIT-Chart ............................. 24

5 UEP: Punctured Turbo Codes ................................... 28
   5.1 Principles of Puncturing .................................... 28
   5.2 Rate-Compatible Punctured Turbo Codes .................... 31
   5.3 Results ..................................................... 32
6 UEP: Pruned Turbo Codes

6.1 Principles of Pruning ........................................... 37
6.2 Single-Trellis Decoding ........................................ 41
6.3 Path-Compatible Pruned Turbo Codes ....................... 42
6.4 Results .......................................................... 43

7 Conclusions ....................................................... 49

Bibliography ......................................................... 53
Chapter 1

Introduction

Video and audio codecs deliver data of different importance and thus, there is a need for different protection levels. Furthermore, user equipment may require different quality, e.g., different video resolution. A PDA requires less resolution than a laptop screen. In order to meet the different requirements and to ensure some kind of graceful degradation for adverse channel conditions, an unequal error protection (UEP) has to be implemented.

Intending to use state of the art capacity-achieving codes like LDPC, Repeat-Accumulate, or Turbo Codes, we need to realise them with UEP properties. In this work, we will only concentrate on Turbo Codes, i.e., parallel concatenated codes with iterative decoding. So-called irregular LDPC codes would allow UEP properties as well, but they are outside of our scope.

Turbo Codes consist of two or more encoders processing data that have partly been interleaved in a more or less random fashion. UEP properties could in principle be achieved by

- applying puncturing to adapt the rate of the component convolutional codes. This is a well-known procedure and many protection levels can be achieved by the so-called RCPC (rate-compatible punctured convolutional) codes. Puncturing schemes are readily available. RCPC codes have been proposed for Turbo Codes, as well, with a possible adaptation of the interleavers. [1, 2, 3, 4, 5, 6, 7, 8]

- pruning certain paths of the trellis of a code such that the distance properties improve. Pruning has not been investigated much until now. Methods for finding such codes are given in [9] and [10]. Such codes are sometimes called path-compatible pruned convolutional (PCPC) codes, which have to be adapted to Turbo Codes.

- multi-fold coding. Hereby, data blocks are encoded several times. Blocks of more important data are encoded more often than others.
concatenating Turbo Codes with other codes like Reed-Muller oder Reed-Solomon codes while devoting the UEP properties to those additional codes.

In this work, we will focus on the first two methods. The first one seems to have been the most obvious way without adding more complexity. The second one seems not to be studied in detail, yet.

This thesis is structured as follows. In Chapter 2 we focus on the structure, properties, and decoding methods of convolutional codes. Chapter 3 deals with the structure, properties and decoding of parallel concatenated convolutional codes, i.e., Turbo Codes. In Chapter 4, an analysis tool for iteratively decoded concatenated codes, the EXIT chart, is introduced. Chapter 5 explains the principles of puncturing, how to embed it into Turbo Codes, how to find good RCPC codes and shows results with those codes. In Chapter 6, pruned codes are discussed and we show how to find good PCPC codes and present results. Finally, conclusions are given in Chapter 7.
Chapter 2

Convolutional Codes

2.1 Structure

In this chapter, we explain the structure of convolutional codes. Their name is due to the fact that the input is convolved with the generator to generate the output. The input and output data streams are theoretically infinitely long, but in praxis blocked. Generally, the rate of the code is $R_C = k/n$, where $k$ and $n$ are the number of input and output bits at one time instance, respectively. Since convolutional codes have memory, they can be represented by shift registers. Figure 2.1 shows such a shift register.

![Shift register with $k = 1$ and $n = 2$](image)

**Figure 2.1**: Shift register with $k = 1$ and $n = 2$

In this example, $k = 1$, $n = 2$, and therefore $R_C = 1/2$. The encoder has $m = 3$ memory elements and thus a *constraint length* of $L = m + 1 = 4$. The generator is described as a matrix determined by the taps of the shift register. For the example in Fig. 2.1 the generators of the two outputs would be

\[
\begin{align*}
g_0 & = \begin{bmatrix} g_{00} & g_{01} & g_{02} & g_{03} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix} \equiv 15_8 \quad (2.1) \\
g_1 & = \begin{bmatrix} g_{10} & g_{11} & g_{12} & g_{13} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix} \equiv 13_8 \quad (2.2)
\end{align*}
\]

or as a matrix...
\[ G = \begin{bmatrix} g_{00} & g_{01} & g_{02} & g_{03} \\ g_{10} & g_{11} & g_{12} & g_{13} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} = [15_8; 13_8]. \] (2.3)

Alternatively, one can also describe the generators by polynomials. Equations (2.1) and (2.2) then become

\[ g_0(D) = g_{00} + g_{01}D + g_{02}D^2 + g_{03}D^3 = 1 + D + D^3 \] (2.4)
\[ g_1(D) = g_{10} + g_{11}D + g_{12}D^2 + g_{13}D^3 = 1 + D^2 + D^3 \] (2.5)

or generally

\[ g_r(D) = \sum_{\nu=0}^{L-1} g_{r\nu} \cdot D^\nu \] (2.6)

The input-output relation of a convolutional encoder of rate \(1/n\) can be expressed by the following equations using the generator matrix.

\[ \begin{bmatrix} x_{00} & x_{10} & x_{01} & x_{11} & \ldots \end{bmatrix} = \begin{bmatrix} g_{00}g_{10} & g_{01}g_{11} & \cdots & g_{0m}g_{1m} \\ g_{00}g_{10} & g_{01}g_{11} & \cdots & g_{0m}g_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots \end{bmatrix} \begin{bmatrix} u_0 & u_1 & \ldots \end{bmatrix}. \] (2.7)

For a convolutional code of rate \(k/n\), the generator is three-dimensional with dimensions \([k \times n \times L]\). A single generator from input \(k_i\) to output \(n_i\) is described by a vector \(g_{k_i,n_i,l_i}\), where \(1 \leq k_i \leq k\), \(1 \leq n_i \leq n\), \(1 \leq l_i \leq L - 1 = m\). The generators of a convolutional code of rate \(2/3\) could for example be as follows.

\[ g_{00} = [11] \quad g_{01} = [01] \quad g_{02} = [11] \]
\[ g_{10} = [11] \quad g_{11} = [01] \quad g_{12} = [11] \]
In this case, Eqn. (2.7) would become

\[
\begin{bmatrix}
x_00x_{10}x_{20} & x_01x_{11}x_{21} & \cdots \\
g_{01,0} & \cdots & g_{01,m} \\
g_{02,0} & \cdots & g_{02,m} \\
\vdots & \ddots & \vdots \\
g_{0k,0} & \cdots & g_{0k,m} \\
g_{10,0} & \cdots & g_{11,0} \\
g_{12,0} & \cdots & g_{12,0} \\
\vdots & \ddots & \vdots \\
g_{1k,0} & \cdots & g_{1k,0} \\
\cdots & & \cdots \\
g_{n1,0} & \cdots & g_{n1,0} \\
\end{bmatrix} \begin{bmatrix}
u_1,0 & u_2,0 & u_3,0 & \cdots \\
u_1,1 & u_2,1 & u_3,1 & \cdots \\
\end{bmatrix} = \begin{bmatrix}
x_{00} & x_{01} & x_{02} & \cdots \\
u_1,0 & u_2,0 & u_3,0 & \cdots \\
u_1,1 & u_2,1 & u_3,1 & \cdots \\
\end{bmatrix} .
\]  

(2.8)

The above explained convolutional encoders are non-recursive. Figure 2.2 shows the structure of a recursive systematic code (RSC).

![Shift register of a RSC with \( k = 1 \) and \( n = 2 \)](image)

Figure 2.2: Shift register of a RSC with \( k = 1 \) and \( n = 2 \)

Such a recursive systematic code (RSC) can be constructed from a non-recursive one. Generally, one has to choose, which \( k \) of the \( n \) outputs should be systematic. Then, the columns of the \([k \times n]\) generator matrix \( G(D) \) are reordered such that the first \( k \) columns correspond to the systematic outputs. Subsequently, one has to build a \([k \times k]\) sub-matrix \( M(D) \) from the first \( k \) columns of the reordered generator matrix.

For example, we have the following generator matrix

\[
G(D) = \begin{bmatrix}
1 + D & D & 1 + D \\
D & 1 & D \\
\end{bmatrix} .
\]  

(2.9)

If we choose the first two outputs to be systematic, the generator matrix does not have to be reordered. Then, we write a \([k \times k]\) sub-matrix

\[
M(D) = \begin{bmatrix}
1 + D & D \\
D & 1 \\
\end{bmatrix} .
\]  

(2.10)
The systematic generator matrix $G'(D)$ is now calculated by multiplying the reordered, or like in this case, original generator matrix $G(D)$ by the inverse of $M(D)$.

$$G'(D) = M^{-1}(D)G(D)$$

$$= \frac{1}{1+D+D^2} \begin{bmatrix} 1 & D \\ D & 1 + D \end{bmatrix} \begin{bmatrix} 1+D & D & 1 + D \\ D & 1 & D \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & \frac{1}{1+D+D^2} \\ 0 & 1 & \frac{1}{1+D+D^2} \end{bmatrix}$$

### 2.2 Graphical Representation of Convolutional Codes

**State diagram**

One possibility of describing convolutional codes is the state diagram. It represents the possible states of the memory elements and the transitions between them. Figure 2.3 shows the state diagram of a $[7; 5]_8$ non-recursive encoder.

![State diagram of a $[7; 5]_8$ non-recursive encoder](image)

**Figure 2.3:** State diagram of a $[7; 5]_8$ non-recursive encoder

The branches between the states represent possible transitions and they are labelled with the corresponding input and output bits. For example, when being in state 01, one would obtain state 10 with an input 1 yielding an output 00.
2.3 Blocking of Convolutional Codes

The trellis diagram is similar to the state diagram but it contains time information. Figure 2.4 shows the trellis of a $[7; 5]_8$ non-recursive encoder. The trellis starts from the all-zero state and is terminated to end in the all-zero state.

![Trellis Diagram](image)

**Figure 2.4:** Trellis diagram of a $[7; 5]_8$ non-recursive encoder

The trellis consists of nodes and branches. The nodes represent the possible states of the memory cells while the branches stand for transitions between them. The horizontal direction describes the time axis while the different states are shown in the vertical direction. The branches are labelled with the input and output bits corresponding to the transition of state $S_i$ to state $S_{i+1}$.

2.3 Blocking of Convolutional Codes

In practice, the length of the information sequence and thus of the trellis is of course finite. There are three ways how to obtain a finite trellis, which are explained now.

**Truncation**

In the first method, we encode only the information bits of length $k \cdot K$ and obtain a code sequence of length $n \cdot K$. The ending state is not defined and thus every state is assumed equally probable. The disadvantage of this method is that the error probability increases at the end of each block.
2.4. PROPERTIES

Termination

When considering terminated trellises, we append $k \cdot m$ additional bits to the information sequence, which results in a defined ending state. In case of a non-recursive convolutional code, this appendage simply consists of $k \cdot m$ zeros, leading to the all-zero state. The disadvantage of this method is the reduction of the code rate due to transmitting more redundancy.

Tail Biting

In the case of tail biting, we first encode the last $k \cdot m$ information bits for achieving a state which is the same as the ending state. From that state, the encoding of the real information sequence starts. The additionally pretransmitted information bits are encoded but the corresponding code bits will not be transmitted, so that there is no overhead. A drawback of this procedure is that the starting and ending states are not known. We only know that they are the same. One can for example apply iterative decoding on a torus, which results from joining the two trellis ends [11].

2.4 Properties

The most important property of a convolutional code apart from $k$, $n$, and $m$ is the free distance $d_{\text{free}}$. The free distance is the minimal Hamming distance between two arbitrary code sequences. For linear codes, the Hamming distance can be replaced by the Hamming weight. We thus have

$$d_{\text{free}} = \min(w_H(x)|x \neq 0) \quad (2.14)$$

with the Hamming weight $w_H$ and the code sequence $x$. The free distance can also be determined from the trellis diagram by searching the trellis for the code sequence with the minimal weight, starting from and ending in the all-zero state. The free distance of a convolutional code determines the asymptotic behaviour of a code for large signal-to-noise ratios.

Not only the free distance but also the whole distance spectrum is an important property of convolutional codes. If the distance spectrum is known, one can determine the Union Bound for estimating the bit error probability for high signal-to-noise ratios. For determining the distance spectrum, the all-zero state of the state diagram has to be cut. The all-zero state is defined to be the starting state $S_s$ and the ending state $S_e$. Now the branches are labelled with powers of the operators $W$, $D$, and $L$, where $W$ stands for the input weight, $D$ for the output weight, and $L$ for the length of the sequence. $WD^2L$ would for example describe a path of input weight one, output weight
2 and length 1. Figure 2.5 shows the corresponding state diagram for the same $[7; 5]_8$ code as above.

![State diagram](image)

**Figure 2.5:** State diagram of a $[7; 5]_8$ non-recursive encoder for determining the distance spectrum

First, linear equations have to be determined for all states apart from the starting state $S_s$. These are:

\[
\begin{align*}
S_{10} &= WD^2L \cdot S_s + WL \cdot S_{01}, \\
S_{01} &= DL \cdot S_{10} + DL \cdot S_{11}, \\
S_{11} &= WDL \cdot S_{11} + WDL \cdot S_{10}, \\
S_e &= D^2L \cdot S_{01}.
\end{align*}
\] (2.15, 2.16, 2.17, 2.18)

Solving these equations for $S_e/S_s$ yields the distance spectrum $T(W, D, L)$

\[
\frac{S_e}{S_s} = \frac{WD^5L^3}{1 - WDL - WDL^2} =: T(W, D, L) = \sum_w \sum_d \sum_l T_{w,d,l} \cdot W^w \cdot D^d \cdot L^l,
\] (2.19)

where the coefficient $T_{w,d,l}$ represents the number of sequences with input weight $w$, output weight $d$, and length $l$. The distance spectrum is also called IOWEF (input-output weight enumerating function). In the case of systematic convolutional encoders, the input-redundancy weight enumerating function (IRWEF) is sometimes more interesting than the IOWEF. Its structure is similar to Eqn. (2.19), but instead of regarding the weight of the whole code sequence, we only consider the weight of the redundancy. Equation (2.19) then becomes

\[
\tilde{T}(W, Z, L) = \sum_z \sum_d \sum_l T_{w,z,l} \cdot W^w \cdot Z^z \cdot L^l,
\] (2.20)

where $w$ and $l$ again denote the input weight and the sequence length, respectively. Instead of the output weight $d$, we now use the redundancy weight $z = d - w$. 
Furthermore, we can define \( a_d \) to be the number of sequences with a certain output weight \( d \).

\[
a_d = \sum_w \sum_l T_{w,d,l}
\]  

For determining the **Union Bound**, we define an upper bound for the probability of decoding a received sequence incorrectly [12].

\[
P_w \leq \sum_{i=1}^{n} a_i Q \left( \sqrt{2iR_c E_b / N_0} \right)
\]  

where the \( Q \)-function is defined as

\[
Q(\alpha) = \frac{1}{\sqrt{2\pi}} \int_{\alpha}^{\infty} e^{-x^2/2} dx.
\]  

We now define the coefficient \( b_i \) to be the number of nonzero information bits corresponding to a weight \( i \) code sequence. Then we obtain an upper bound for the bit error probability.

\[
P_b \leq \frac{1}{k} \sum_{i=1}^{n} a_i b_i Q \left( \sqrt{2iR_c E_b / N_0} \right)
\]

Due to the properties of the \( Q \)-function and \( a_1 = \ldots = a_{d_{\text{free}} - 1} = 0 \), the first term of the sum in Eqn. (2.24), i.e., \( a_{d_{\text{free}}} b_{d_{\text{free}}} Q \left( \sqrt{2d_{\text{free}}R_c E_b / N_0} \right) \) dominates the upper bound and is a good approximation for the bit-error probability at high signal-to-noise ratios.

\[
P_b \approx \frac{1}{k} a_{d_{\text{free}}} b_{d_{\text{free}}} Q \left( \sqrt{2d_{\text{free}}R_c E_b / N_0} \right)
\]

If the encoder is systematic, \( b_i / k \approx i / n \) holds and yields

\[
P_b \approx \frac{d_{\text{free}}}{n} a_{d_{\text{free}}} Q \left( \sqrt{2d_{\text{free}}R_c E_b / N_0} \right).
\]  

Equation (2.26) is the **Union Bound** for systematic encoders.

### 2.5 Decoding

In this section, two widely used algorithms for the decoding of convolutional codes are presented. The first one is the Viterbi algorithm, which is a maximum likelihood
2.5. DECODING

sequence estimator (MLSE) and yields hard-decision output data. The second decoder, we will describe, is the BCJR algorithm, which is a symbol-by-symbol maximum a-posteriori (MAP) algorithm.

2.5.1 Viterbi Algorithm

The Viterbi algorithm was first introduced by A. J. Viterbi in 1967 [13]. It performs a maximum likelihood sequence estimation (MLSE) and is used for decoding and equalization. When having equally likely code sequences, the MLSE criterion is the optimum decoding strategy. For the derivation, the information sequence, the code sequence, and the received sequence are denoted by $u$, $x$, and $r$, respectively. Furthermore, $a$ describes any possible code sequence.

The maximum likelihood criterion requires

$$p(r|\hat{x}) \geq p(r|a).$$

(2.27)

The decoding is carried out by determining $\hat{x} = \arg \max_a p(r|a)$. For a memoryless channel, the right side of Eqn. (2.27) can be factorised into the following:

$$p(r|a) = \prod_{l=0}^{N-1} p(r(l)|a(l)) = \prod_{l=0}^{N-1} \prod_{i=1}^{n} p(r_i(l)|a_i(l)),$$

(2.28)

where $N$ is the number of the encoded data symbols and $n$ is the number of bits per code symbol. As the natural logarithm is a monotonically increasing function, Eqn. (2.27) can also be expressed by its logarithm. The right side then changes to

$$\ln (p(r|a)) = \ln \left( \prod_{l=0}^{N-1} \prod_{i=1}^{n} p(r_i(l)|a_i(l)) \right)$$

(2.29)

$$= \sum_{l=0}^{N-1} \sum_{i=1}^{n} \ln (p(r_i(l)|a_i(l)))$$

(2.30)

$$= \sum_{l=0}^{N-1} \sum_{i=1}^{n} \gamma'(r_i(l)|a_i(l)).$$

(2.31)

The expression $\gamma'(r_i(l)|a_i(l))$ is called Viterbi metric and represents logarithm of the transition probabilities of the channel. For AWGN channels, the probability densities $p(r|a)$ are exponential functions:

$$p \left( r_i(l)|a_i(l) = \pm \sqrt{E_s} \right) = \frac{1}{\sqrt{\pi N_0}} e^{-\frac{(r_i(l)|a_i(l))^2}{N_0}},$$

(2.32)
where \( E_s/N_0 \) is the signal-to-noise ratio. The Viterbi metric becomes

\[
\gamma'(r_i(l)|a_i(l)) = \ln (p(r_i(l)|a_i(l))) = C - \frac{(r_i(l) - a_i(l))^2}{N_0} \quad (2.33)
\]

\[
= C - \frac{r_i(l)^2}{N_0} + 2 \cdot \frac{a_i(l)r_i(l)}{N_0} - \frac{a_i(l)^2}{N_0} \quad (2.34)
\]

\[
= C - \frac{r_i(l)^2}{N_0} + \frac{a_i(l)r_i(l)}{N_0/2} - \frac{E_s}{N_0} . \quad (2.35)
\]

Since the first, second, and fourth term do not depend on \( a_i(l) \), they can be omitted when maximizing the probability density. As a metric, we therefore define

\[
\gamma(r_i(l)|a_i(l)) = \frac{2}{N_0} \cdot a_i(l) \cdot r_i(l) . \quad (2.36)
\]

The procedure of the algorithm can best be illustrated by means of the trellis diagram. The decoding result represents that path in the trellis which has the largest metric or alternatively the smallest Euclidean distance to the received code sequence. Searching for the best possible trellis path would in most cases be much too complex. Due to the Markov property of convolutional codes, each state only depends on the previous state and the input. This property leads to the possibility of calculating the metrics recursively.

The Viterbi algorithm will now be described in more detail. Let the trellis begin in the all-zero state at time instant \( l = 0 \). Then, we calculate the path metrics of all possible transitions to the next state. There are \( 2^k \) paths merging in as well as starting from each state. At each state, only one of the merging transitions will survive, namely the one with the largest path metric. All other merging paths will not be considered any more. The accumulated path metric of a transition and also the state, where the transition is emerging from, is stored. For the next time instant, the calculated path metrics starting from that state are added to the previously stored metric before being compared to another path. This procedure is performed for each time instant, step by step. For terminated trellises, the path ending in the all-zero state with the best metric is automatically determined. If the trellis is not terminated, the path with the best metric among all paths at the \( N \)-th time instant may be chosen as a decoding result, in order to reduce edge effects (higher error rates at the block ends) as much as possible.

Figure 2.6 shows an example of the Viterbi algorithm for a convolutional code of rate \( R_c = 1/2 \), with constraint length \( L_c = 3 \) and generators \( g_1 = 7_8 \) and \( g_2 = 5_8 \). The number of information bits is \( K = 4 \) plus two tailbits.

Let the information sequence be \( u = 1001(00) \), then the coded sequence is \( x = (+1 + 1 + 1 - 1 + 1 + 1 + 1 + 1 (+1 - 1 + 1 + 1)) \). Let the hard quantized received sequence be \( r = (-1 - 1 + 1 - 1 + 1 + 1 + 1 + 1 (+1 - 1 + 1 + 1)) \). Then, the Viterbi
algorithm follows the procedure outlined in Fig. 2.6.

![Figure 2.6: Viterbi algorithm](image)

The solid and dashed lines represent information bits 0 and 1, respectively. The actual sequence is marked in bold. The numbers above the trellis denote the received sequence. The single path metrics are positioned right beside each transition and the numbers above each state are the sums of the metrics of the surviving paths leading to those states. Although the number of errors (3) is larger than the number of correctible errors of this code ($d_{\text{free}} = 2$), when referring to the free distance, the Viterbi algorithm is able to correct all of them. This is due to the distribution of the errors. The Viterbi algorithm is predestined to correcting single, random error events but fails to correct burst errors.

### 2.5.2 BCJR Algorithm

The BCJR was proposed by Bahl, Cocke, Jelinek, and Raviv [14] in 1974. Like the Viterbi algorithm, it was not only intended for decoding but also as an equalizing algorithm. The BCJR algorithm is based on the maximum a-posteriori probability (MAP) criterion and also exploits the Markov property of convolutional codes. The output of the BCJR algorithm is a soft value, i.e., it calculates log-likelihood ratios (LLR’s). The definition of a conditional LLR is as follows:

$$L(\hat{u}_i(l)) = \ln \frac{P(u_i(l) = 0|r)}{P(u_i(l) = 1|r)}.$$  \hspace{1cm} (2.37)
This can also be expressed as

\[ L(\hat{u}_i(l)) = \ln \frac{\sum_{(S_\nu, S_\mu) \atop u_i(l)=0} p(S_\nu, S_\mu, r)}{\sum_{(S_\nu, S_\mu) \atop u_i(l)=1} p(S_\nu, S_\mu, r)}. \]  

(2.38)

We now divide the received sequence \( r \) into three parts: The first part \( r(k < l) \) corresponds to the received symbols before the actual time instant, the second part \( r(l) \) corresponds to the actual time instant, and the third part \( r(k > l) \) represents the received symbols after the actual time instant \( l \). Then, Eqn. (2.38) can be rewritten as

\[ L(\hat{u}_i(l)) = \ln \frac{\sum_{(S_\nu, S_\mu) \atop u_i(l)=0} p(S_\nu, S_\mu, r(k < l), r(l), r(k > l))}{\sum_{(S_\nu, S_\mu) \atop u_i(l)=1} p(S_\nu, S_\mu, r(k < l), r(l), r(k > l))}. \]  

(2.39)

The probability densities in the numerator and denominator of Eqn. (2.39) can be reformulated as

\[ p(S_\nu, S_\mu, r(k < l), r(l), r(k > l)) = p(S_\nu, r(k < l)) \cdot p(S_\mu, r(l)|S_\nu) \cdot p(r(k > l)|S_\mu). \]  

(2.40)

The three parts of this product are now denoted by \( \alpha_{l-1}(S_\nu) \), \( \gamma_l(S_\nu, S_\mu) \), and \( \beta_l(S_\mu) \):

\[ \alpha_{l-1}(S_\nu) = p(S_\nu, r(k < l)) \]  

(2.41)

\[ \beta_l(S_\mu) = p(r(k > l)|S_\mu) \]  

(2.42)

\[ \gamma_l(S_\nu, S_\mu) = p(S_\mu, r(l)|S_\nu) = p(r(l)|S_\nu, S_\mu) \cdot p(S_\mu|S_\nu) \text{ with } \]  

(2.43)

\[ p(r(l)|S_\nu, S_\mu) = \frac{1}{\sqrt{2\pi\sigma_N^2}} \cdot \exp \left( -\frac{||r(l) - z_{\nu\mu}||^2}{2\sigma_N^2} \right) \]  

(2.44)

for AWGN channels, where \( z_{\nu\mu} \) is the code sequence corresponding to the state transition. Equation (2.39) can be rewritten using \( \alpha \), \( \beta \), and \( \gamma \).

\[ L(\hat{u}_i(l)) = \ln \frac{\sum_{(S_\nu, S_\mu) \atop u_i(l)=0} \alpha_{l-1}(S_\nu) \cdot \gamma_l(S_\nu, S_\mu) \cdot \beta_l(S_\mu)}{\sum_{(S_\nu, S_\mu) \atop u_i(l)=1} \alpha_{l-1}(S_\nu) \cdot \gamma_l(S_\nu, S_\mu) \cdot \beta_l(S_\mu)}. \]  

(2.45)

A big advantage of this representation is that \( \alpha \) and \( \beta \) can be calculated recursively:
\[ \alpha_l(S_\mu) = p(S_\mu, r(k < l + 1)) = \sum_{S_\nu} p(S_\nu, S_\mu, r(k < l), r(l)) \]  
(2.46)

\[ = \sum_{S_\nu} \gamma_l(S_\nu, S_\mu) \cdot \alpha_{l-1}(S_\nu) \]  
(2.47)

\[ \beta_{l-1}(S_\nu) = p(r(k > l - 1)|S_\nu) = \frac{1}{P(S_\nu)} \sum_{S_\mu} p(S_\nu, S_\mu, r(l), r(k > l)) \]  
(2.48)

\[ = \sum_{S_\mu} \gamma_l(S_\nu, S_\mu) \cdot \beta_l(S_\mu). \]  
(2.49)

From these equations, one can see that \( \alpha \) can be calculated through a forward recursion, since each \( \alpha \) is calculated from the \( \alpha \)'s of the previous states. Whereas, \( \beta \) can be calculated through a backward recursion. For calculating the decoding result by means of the BCJR algorithm, one has thus to calculate one forward recursion through the trellis for determining \( \alpha \) and one backward recursion for determining \( \beta \).

Figure 2.7 shows a trellis segment of a recursive systematic convolutional code of rate \( R_c = 1/2 \) with generators \( \mathbf{g} = [7; 5]_8 \). The encoder has \( m = 2 \) memory cells and therefore \( 2^m = 4 \) possible states that are denoted by 0, ..., 3 which correspond to the decimal counterpart of the binary states. As we only have one input stream, there are two transitions starting from as well as merging into one state. The trellis segment shows \( \alpha, \beta, \) and \( \gamma \) of the states and transitions. The dashed and solid transitions correspond to information bit \( u_i = 0 \) and \( u_i = 1 \), respectively. The arrows above and underneath the trellis segment illustrate the recursion directions of \( \alpha \) and \( \beta \), respectively.

![Trellis segment of a rate 1/2 convolutional code](image_url)
For starting these recursions, \(\alpha\) and \(\beta\) have to be initialized. Since the trellis is always started from the all-zero state, \(\alpha_0(S = 0) = 1\) and \(\alpha_0(S \neq 0) = 0\). This means that the probability of the state being the all-zero state is one and the probabilities for the other states is zero. The initialization of \(\beta\) depends on if the trellis is terminated or not. In the case of trellis termination, we also have \(\beta_{N-1}(S = 0) = 1\) and \(\beta_{N-1}(S \neq 0) = 0\). If the trellis is not terminated, the \(\beta\)'s of the last state are set to the values of the \(\alpha\)'s at those states or are, alternatively, set to equal probabilities \(2^{-m}\).

### 2.5.3 Log-MAP and Max-Log-MAP

Since the calculation of the BCJR algorithm is still quite complex, it is advantageous to do the calculations in the logarithmic domain. This is done in the Log-MAP and Max-Log-MAP algorithms. Since we assume Gaussian noise, we choose the natural logarithm. Instead of \(\alpha_l(S_\mu)\), \(\beta_l(S_\nu)\), and \(\gamma_l(S_\nu, S_\mu)\), we are now using \(\bar{\alpha}_l(S_\mu) = \ln(\alpha_l(S_\mu))\), \(\bar{\beta}_l(S_\nu) = \ln(\beta_l(S_\nu))\), and \(\bar{\gamma}_l(S_\nu, S_\mu) = \ln(\gamma_l(S_\nu, S_\mu))\). Within the following calculations, we make use of the so-called Jacobi logarithm

\[
\ln(e^a + e^b) = \max(a, b) + \ln(1 + e^{-|a-b|}) \quad (2.50)
\]

for reducing computational complexity. For more than two terms, the equation can be solved iteratively. We obtain

\[
\bar{\alpha}_l(S_\mu) = \ln(\alpha_l(S_\mu)) = \ln\left(\sum_{S_\nu} \gamma_l(S_\nu, S_\mu) \cdot \alpha_{l-1}(S_\nu)\right) \quad (2.51)
\]

\[
= \ln\left(\sum_{S_\nu} \exp(\ln(\gamma_l(S_\nu, S_\mu) \cdot \alpha_{l-1}(S_\nu)))\right) \quad (2.52)
\]

\[
= \ln\left(\sum_{S_\nu} \exp(\bar{\gamma}_l(S_\nu, S_\mu) + \bar{\alpha}_{l-1}(S_\nu))\right) \quad (2.53)
\]

\[
\overset{(2.50)}{=} \max_{S_\nu} (\bar{\gamma}_l(S_\nu, S_\mu) + \bar{\alpha}_{l-1}(S_\mu)) + \ln \left(1 + e^{-|\Delta_l|}\right) \quad \text{and} \quad (2.54)
\]

\[
\bar{\beta}_{l-1}(S_\nu) = \ln(\beta_{l-1}(S_\nu)) = \ln\left(\sum_{S_\mu} \exp(\ln(\gamma_l(S_\nu, S_\mu) \cdot \beta_l(S_\mu)))\right) \quad (2.55)
\]

\[
\overset{(2.50)}{=} \max_{S_\mu} (\bar{\gamma}_l(S_\nu, S_\mu) + \bar{\beta}_l(S_\nu)) + \ln \left(1 + e^{-|\Delta_l|}\right), \quad (2.56)
\]

where \(\Delta_l\) represents the difference of the terms of the maximization. For \(\bar{\gamma}_l(S_\nu, S_\mu)\), we can write
\[
\gamma_l(S_\nu, S_\mu) = \ln \left( \frac{\gamma_l(S_\nu, S_\mu) - \left| z_{\nu\mu} \right|^2}{2\sigma_N^2} \right) + \ln \left( P(S_\mu | S_\nu) \right)
\]
\[
= C - \frac{\left| r(l) - z_{\nu\mu} \right|^2}{2\sigma_N^2} + \ln \left( 1 + e^{\mp L_a(u)} \right)
\]
\[
= C - \frac{\left| r(l) - z_{\nu\mu} \right|^2}{2\sigma_N^2} - \ln \left( 1 + e^{\mp L_a(u)} \right)
\]
\[
\xrightarrow{(2.50)} C - \frac{\left| r(l) - z_{\nu\mu} \right|^2}{2\sigma_N^2} - \max(0, \mp L_a(u)) + \ln \left( 1 + e^{\mp L_a(u)} \right)
\]

using (2.43) and (2.44) and expressing \( P(S_\mu | S_\nu) \) by the a-priori information \( L_a(u) \).

\[
P(S_\mu | S_\nu) = \frac{1}{1 + e^{\mp L_a}}
\]

In the last equations, the plus and minus signs hold for those transitions, where \( u_i(l) = 1 \) and \( u_i(l) = 0 \), respectively, and \( C \) is a constant which can be omitted. For further reducing complexity, the logarithmic correction terms in the above equations can be stored in a look-up table instead of calculating the logarithm each time.

These equations lead to the Log-MAP algorithm. This algorithm has the same performance as the BCJR algorithm but less complexity. For the Max-Log-MAP algorithm, we omit the logarithmic correction terms in (2.54), (2.56), and (2.60). This is a further reduction in complexity but leads to small performance degradations.
Chapter 3

Parallel Concatenated Convolutional Codes / Turbo Codes

The motivation for concatenating codes is the desire to design powerful codes with limited decoding complexity. One can construct powerful codes, e.g., by increasing the constraint length, but this leads to very complex decoders. Another possibility is to concatenate small codes in serial or in parallel to obtain a better overall code. The decoding can be done by iteratively using soft-output decoding of the component codes. In the following sections, we present the structure, properties, and decoding methods of concatenated codes. Since we will use ”Turbo” Codes in the next chapters, we will only concentrate on parallel concatenated codes.

3.1 Structure

In the case of parallel concatenated convolutional codes, there are two or more single convolutional encoders that are both supplied with the same information bits, only in different order. From now on, we restrict the treatment to ”Turbo” Codes with two parallel encoders. The first encoder is supplied with the original information data while the input of the second is an interleaved version of the information sequence. We assume systematic convolutional encoders. The output of the ”Turbo” encoder consists of the systematic output, i.e., the information sequence itself, and the outputs of the two convolutional encoders. The overall rate of the encoder thus is

$$R_c = \frac{k}{n_1 + n_2 - k},$$

if the constituent encoders are of rate $k/n_1$ and $k/n_2$, respectively. We transmit both redundancy parts but only one systematic part. Figure 3.1 shows a block diagram for such a Turbo encoder with two rate-1/2 constituent convolutional codes. Thus, the overall rate is $R_c = 1/3$. The encoder output consists of $c_1$, the systematic bits, $c_2$,
the parity bits of the first encoder, and $c_3$, the parity bits of the second encoder.

![Block diagram of a Turbo encoder](image)

**Figure 3.1**: Block diagram of a Turbo encoder

The combination of two single codes, however, does not always lead to a code with better properties than those of the single codes. Finding good concatenated codes requires exhaustive computer searches. Furthermore does the performance of a concatenated code not only depend on the constituent codes but also on the interleaver.

### 3.2 Properties

One obvious issue for a Turbo Code is the choice of its constituent encoders. We will not give special examples but present a few facts that generally lead to satisfactory codes. First of all, many Turbo Codes consist of two equal encoders. However, good asymmetric Turbo Codes have also been found. Furthermore, the aim is to construct an overall encoder from small constituent encoders, therefore the single encoders should have small constraint lengths. In practice, the constraint length is mostly smaller than 5. Despite of the single component encoders, the block length of Turbo Codes should be chosen large, typically several thousand bits. Choosing smaller block lengths would lead to statistical bindings and thus to performance degradations. Larger constituent encoders, i.e., with higher constraint length would also lead to early statistical bindings in the iterative decoding process.

As for convolutional codes, the distance spectrum is again one of the most important properties of Turbo Codes. It has a strong influence on the power of the code, i.e., on its correction capability. Since Turbo Codes mostly consist of recursive systematic convolutional encoders, we will now concentrate on the input-redundancy weight enumerating function (IRWEF), which does not consider code sequences of certain input
and output weights, respectively, but of certain input and redundancy weight. The IRWEF of a Turbo Code contains the number of code sequences with input weight \( w \), redundancy weight \( z \), and length \( l \). The redundancy weight can be expressed by the sum of the redundancy weight of the constituent encoders. Due to the interleaver, which is in most cases a random or pseudo-random interleaver, the input sequence of the second encoder is unknown. This means that the input weight and thus, the redundancy weight of the second encoder is unknown, as well.

For still being able to determine the distance spectrum of a Turbo Code, we introduce the concept of a uniform interleaver. This interleaver of size \( K \) maps the input of weight \( w \) to all possible \( \binom{K}{w} \) permutations with equal probability \( 1/\binom{K}{w} \), where \( K \) is the length of the interleaver. This concept allows us to determine an average distance spectrum of the desired Turbo Code.

We first introduce the conditional weight enumerating function (CWEF) of a convolutional code

\[
A_w(Z) = \sum_z A_{w,z} Z^z ,
\]

with \( A_{w,z} \) being the coefficients for codewords with input weight \( w \) and redundancy weight \( z \). In other words,

\[
A_{w,z} = \sum_l T_{w,z,l}
\]

with \( T_{w,z,l} \) being the coefficients given in Eqn. (2.20). The CWEF of a Turbo Code can then be expressed as

\[
A_{TC}^w(Z) = \frac{A_{C1}^w(Z)A_{C2}^w(Z)}{\binom{K}{w}} ,
\]

where \( A_{C1}^w(Z) \) and \( A_{C2}^w(Z) \) describe the CWEF’s of the two constituent encoders. The IRWEF of a Turbo Code is then given by

\[
A^{TC}(W,Z) = \sum_{w,z} A_{w,z} W^w Z^z = \sum_{1 \leq w \leq K} W^w A_w^{TC}(Z). \]

It should be noted that this is just an average distance spectrum and can thus contain non-integer coefficients.

Another important property of Turbo Codes is the free distance \( d_{\text{free}} \). With \( d_{\text{free},1} \) and \( d_{\text{free},2} \) as free distances of the two constituent encoders, the free distance of the Turbo Code can be given by

\[
d_{\text{free,TC}} = d_{\text{free},1} + d_{\text{free},2} - 1 ,
\]

where we assume a rate-1/n convolutional code.
3.3 Decoding Structure

The optimal way of decoding a Turbo Code would be to decode the overall parallel concatenated code. Since this would be very complex, the common procedure is iterative decoding, which employs the single decoders of the two constituent codes. These two decoders are activated one after another for several iterations, passing extrinsic information to each other. Figure 3.2 shows the block diagram of a Turbo decoder.

![Figure 3.2: Block diagram of a Turbo decoder](image)

The decoding procedure will now be explained in detail. We will use the log-likelihood ratio or $L$-values introduced in Eqn. (2.37). The log-likelihood ratio of an estimated bit is given by

$$ L(\hat{u}_i) = \ln \left( \frac{P(u_i = 0 | r)}{P(u_i = 1 | r)} \right). $$

(3.7)

The sign of a log-likelihood ratio corresponds to the sign of the estimated bit, while the absolute value provides reliability information about the estimated bit. Large absolute values correspond to reliable information, whereas small values indicate uncertain decisions. Equation (3.7) can be reformulated as

$$ L(\hat{u}_i) = \ln \left( \frac{\sum_{x \in X_i^{(0)}} p(x, r) p(r | x) \cdot P(x)}{\sum_{x \in X_i^{(1)}} p(x, r) p(r | x) \cdot P(x)} \right) = \ln \left( \frac{\sum_{x \in X_i^{(0)}} p(r | x) \cdot P(x)}{\sum_{x \in X_i^{(1)}} p(r | x) \cdot P(x)} \right) $$

(3.8)

$$ = \ln \left( \frac{\sum_{x \in X_i^{(0)}} \prod_{j=0}^{n-1} p(r_j | x_j) \prod_{j=0}^{k-1} P(x_j)}{\sum_{x \in X_i^{(1)}} \prod_{j=0}^{n-1} p(r_j | x_j) \prod_{j=0}^{k-1} P(x_j)} \right), $$

(3.9)
where $\mathcal{X}_i^{(0)}$ and $\mathcal{X}_i^{(1)}$ describe the sets of all code sequences containing a 0 and a 1 at position $i$, respectively. Furthermore, Eqn. (3.9) can be rewritten for systematic codes as

$$L(\hat{u}_i) = L(r_i|x_i) + L_a(\hat{u}_i) + \ln \frac{\sum_{x \in \mathcal{X}_i^{(0)}} \prod_{j=0}^{n-1} p(r_j|x_j) \cdot \prod_{j=0, j \neq i}^{k-1} P(x_j)}{\sum_{x \in \mathcal{X}_i^{(1)}} \prod_{j=0}^{n-1} p(r_j|x_j) \cdot \prod_{j=0, j \neq i}^{k-1} P(x_j)} \cdot L_e(\hat{u}_i).$$  \hspace{1cm} (3.10)

We observe that the log-likelihood ratio $L(\hat{u}_i)$ consists of three parts: The log-likelihood-ratio of the systematic component, the intrinsic information $L(r_i|x_j)$, the a-priori information $L_a(\hat{u}_i)$, and a third term $L_e(\hat{u}_i)$ which is called extrinsic information.

We now explain the iterative decoding. Both decoders receive the channel output corresponding to the respective encoder. The first decoder is supplied with the systematic channel output bits and the redundancy of the first encoder, while the second decoder receives the interleaved systematic bits and the redundancy of the second encoder. Decoder 1 produces an estimate of the received data by computing $L$-values, which contain the sign and some reliability information of the estimated bits. Before passing this information to the second decoder, the a-priori and intrinsic information is subtracted from the estimated $L$-values in order to extract the extrinsic information. The extrinsic values enter an interleaver and become a-priori information for the second decoder. Decoder 2 receives its a-priori information and the channel output, which contains an interleaved version of the systematic bits. It computes log-likelihood ratios as well, extracts the extrinsic information, and deinterleaves in order to deliver the a-priori input to the first decoder. This procedure is applied over several iterations until no more information gain can be achieved at the outputs of the decoders.

### 3.4 Decoding Algorithms

As decoding algorithm, we obviously have to choose an algorithm that delivers soft-decision outputs to the next decoder. Algorithms expecting and delivering soft information are called soft-in/soft-out (SISO) algorithms. In Section 2.5 we explained the Viterbi and the BCJR algorithms with its approximations. The latter deliver log-likelihood ratios as output and are thus suitable for iterative decoding. The original Viterbi algorithm only computes the maximum likelihood path of the trellis and thus a hard-decision output. The Viterbi algorithm has to be modified...
slightly for achieving reliability information on its decisions. This is realised by the Soft-Output Viterbi Algorithm (SOVA). We will not give the complete derivation but just the main idea and the modifications of the conventional Viterbi algorithm.

Let us assume a rate-1/n code which leads to two transitions starting from and ending in one state. The two paths merging into state $S_i$ at time instant $k$ are denoted by $A$ and $B$. Let the accumulated path metrics at time instant $k$ be $\Gamma_A^{(k)}$ and $\Gamma_B^{(k)}$. The probability of a received sequence $r$ under the condition that path $A$ and path $B$ were chosen, is

$$p(r|A) = \text{const.} \cdot e^{-\Gamma_A} \quad \text{and}$$

$$p(r|B) = \text{const.} \cdot e^{-\Gamma_B}.$$  

Without loss of generality, we assume that we decided for $A$ to be the correct path. With Bayes' theorem, we can express the probability that path $A$ is correct under the condition that $r$ has been received.

$$p_c = p(A|r) = \frac{P(A, r)}{p(r)} = \frac{P(r|A)P(A)}{P(r|A)P(A) + P(r|B)P(B)}$$

$$= \frac{1}{1 + e^{-|\Gamma_A - \Gamma_B|}}$$

The Soft-Output Viterbi algorithm performs the add-compare-select computations for determining the path metric already described in Section 2.5.1. Since the Soft-Output Viterbi algorithm needs to calculate soft-decision information, we additionally have to determine the reliability of the path decisions at each time instant. These reliability measure is updated after each time instant for each path segment of path $A$ within the interval $t - \tau \leq t' \leq t$, where $\tau$ describes the finite path memory.

$$L_A(t')^{(l+1)} = \begin{cases} L_A(t')^{(l)} \cdot p_c & \text{for } x_A(t') \neq x_B(t') \\ L_A(t')^{(l)} & \text{for } x_A(t') = x_B(t') \end{cases}$$

These updates are only calculated if the estimated information symbols of the paths $A$ and $B$ are different. Otherwise, a path selection would not have any significance for the output bit.
Chapter 4

An Analysis Tool: The EXIT-Chart

Typically, we use bit-error rates for analyzing the performance of a system. In this chapter, we introduce another analysis tool, the EXtrinsic Information Transfer chart also known as EXIT chart, which is especially useful for concatenated codes and iterative decoding. The EXIT chart was first developed by Stephan ten Brink in 2001 [15]. The starting point for the derivation of the EXIT chart is the observation that the a-priori input $L_a$ of one of the decoders can be modelled by an independent Gaussian random variable $n_a$ with variance $\sigma_a$ and zero mean together with the transmitted systematic bits $x_s$.

$$L_a = \frac{\sigma_a}{2} x_s + n_a$$

(4.1)

The conditional probability density function of the $L$-values $L_a$ is

$$p_a(L_a|X = x_s) = \frac{e^{-(L_a-(\sigma_a^2/2)x_s)^2/2\sigma_a^2}}{\sqrt{2\pi}\sigma_a}$$

(4.2)

This leads to the mutual information $I_a = I(X; L)$.

$$I_a = \sum_{x_s=-1,1} \int_{-\infty}^{+\infty} p_a(\xi|X = x_s) \cdot \log_2 \frac{2 \cdot p_a(\xi|X = x_s)}{p_a(\xi|X = -1) + p_a(\xi|X = +1)} d\xi$$

(4.3)

With Eqn. (4.2), Eqn. (4.3) becomes

$$I_a(\sigma_a) = 1 - \int_{-\infty}^{+\infty} p_a(\xi|X = x_s) \cdot \log_2(1 + e^{-\xi}) d\xi$$

(4.4)

$$= 1 - E \left\{ \log_2(1 + e^{-L_a}) \right\} \approx 1 - \frac{1}{N} \sum_{i=1}^{N} \log_2(1 + e^{-x_{s,i} L_{a,i}}).$$

(4.5)
The mutual information for the extrinsic output $L_e$ can be formulated in the same way, i.e.,

$$I_e(\sigma_a) = 1 - E \left\{ \log_2(1 + e^{-L_e}) \right\} \approx 1 - \frac{1}{N} \sum_{i=1}^{N} \log_2(1 + e^{-x_{s,i} L_e;i}) . \quad (4.6)$$

It should be noted that the mutual information always lies between zero and one.

The EXIT chart now plots the transfer function between the a-priori information and the extrinsic information.

$$I(L_e; X) = T(I(L_a; X)) \quad (4.7)$$

Figure 4.1 shows this transfer function of a Log-MAP decoder for a recursive systematic convolutional code of rate $R_c = 2/3$ and memory $m = 2$. The generator is $g = [17, 06, 15]$.

One can see that the extrinsic information is quite small where the a-priori information is zero. With increasing a-priori information, the extrinsic information also increases until it reaches the point $(1, 1)$.

When using EXIT charts for iteratively decoded concatenated codes, the transfer curves for both decoders are plotted into one single transfer chart. Since the extrinsic information of one decoder becomes the a-priori information of the other decoder, the transfer curve of the second decoder is flipped, thereby representing the inverse transfer function.

$$I(L_a; X) = T^{-1}(I(L_e; X)) \quad (4.8)$$
This is possible because the original transfer function is monotonically increasing and therefore, its inverse exists. One can additionally add the trajectories between the two transfer functions which represent the information transferred between the two decoders. Figure 4.2 shows the complete EXIT chart of an iterative decoding scheme with two identical rate $R_c = 2/3$ recursive systematic convolutional encoders with generator $g = [10, 13, 15]$ and memory $m = 2$ at $E_b/N_0 = 1.4$ dB.

![EXIT Chart](image)

**Figure 4.2:** Transfer chart of a rate $R_c = 2/3$ convolutional encoder

From the EXIT chart, we can conclude several properties. If the two curves intersect before reaching the point $(1, 1)$, the encoding does not converge and we are not able to achieve error-free transmission. The trajectories between the curves represent the decoder activations and therefore indicate the number of decoding iterations that are necessary for virtually error-free transmission.

The quality of the results depends on the signal-to-noise ratio. For lower signal-to-noise ratios, the two curves intersect and do not permit error-free transmission. With increasing signal-to-noise ratios, the two curves widen and let the intersection point move towards the point $(1, 1)$. For even larger signal-to-noise ratios, the already open curves further widen which leads to faster convergence.

Hence, the segments of bit-error rate curves are related to the properties of an EXIT chart. These can be divided into three parts. The first one, the pinch-off region, is where the bit-error rate hardly decreases with increasing signal-to-noise ratio. The second part is called the waterfall region and describes the region, where the bit error rate dramatically decreases. The third region represents the "error floor" of the bit error rate.

These three regions can also be determined by the EXIT chart. The first region represents those cases, where the transfer curves intersect below the point $(1, 1)$. The waterfall region describes the case, when the transfer curves open and allow the
decoding to converge and the error floor represents the case, when the transfer curves further widen. However, not much can be deduced for this error floor region. Evaluating the EXIT chart in addition to the union bound provides an estimate for the bit-error curve, since the EXIT chart provides information on the waterfall region, whereas the union bound approximates the asymptotic behaviour for high signal-to-noise ratios and low bit-error rates.

Consider the independent random variable $n_a$ as the noise of an independent channel. In the case of this independent channel being a binary erasure channel, it has been proven in [16] that the area under the first (upper) EXIT curve represents the capacity, whereas area under the second (lower) represents the code rate. In order to reach the capacity, the area between the two curves has to be minimized without intersection of the curves.
Chapter 5

UEP: Punctured Turbo Codes

The next two chapters describe two possibilities for applying unequal error protection (UEP) to Turbo Codes. The necessity of unequal error protection can have several reasons. For example in video, audio, and image coding often exist data with different relevance. When transmitting over noisy channels, these unequally important data have to be protected by codes with different correction capabilities. Another reason might be the representation of images or videos on different screens requiring different resolutions.

For providing different correction capabilities, we obviously have to choose different channel codes. The drawback of totally different codes is that the system would become more complex, because all encoders and decoders would have to be implemented, separately. To avoid this, one may, e.g., apply for example puncturing. In the next sections, we describe the principles of punctured convolutional codes and apply puncturing to Turbo Codes.

5.1 Principles of Puncturing

Puncturing means transmitting not all code bits produced by the encoder. This leads to a smaller amount of redundancy and thus, a higher code rate \( R_c \). Puncturing can, for example, be used for adjusting the length of the code sequence to a given frame length, for adapting the code rate to transmission conditions or for unequal error protection. As mentioned above, some of the code bits are not transmitted in a punctured code. This is carried out with a certain puncturing period \( L_p \). The procedure can be
represented by a puncturing matrix $P$ of dimensions $[n \times L_p]$. 

$$P = \begin{bmatrix} p_{00} & p_{01} & \cdots & p_{0(L_p-1)} \\ p_{10} & p_{11} & \cdots & p_{1(L_p-1)} \\ \vdots & \vdots & \ddots & \vdots \\ p_{(n-1)0} & p_{(n-1)1} & \cdots & p_{(n-1)(L_p-1)} \end{bmatrix} = [p_0 p_1 \cdots p_{L_p-1}]$$ (5.1)

The columns of the puncturing matrix are assigned periodically to the code symbols, where a zero in the puncturing column implicates that the corresponding bit of the code symbol will not be transmitted. Generally, the puncturing matrix consists of $k \cdot L_p + l$ ones instead of $n \cdot L_p$. This means that the code rate changes to 

$$R'_c = \frac{k \cdot L_p}{k \cdot L_p + l}$$ (5.2)

where the parameter $l$ is restricted to $1 \leq l \leq (n-k) \cdot L_p$. This restriction leads to the following interval of possible code rates. 

$$\frac{k \cdot L_p}{n \cdot L_p} = \frac{k}{n} \leq R'_c \leq \frac{k \cdot L_p}{k \cdot L_p + 1}$$ (5.3)

For an original convolutional code of rate $R_c = 1/2$ and puncturing period $L_p = 4$, the code rate can be adjusted between $1/2 \leq R'_c \leq 4/5$ with 4 different rates within this interval. For a higher resolution of the possible code rates of this mother code, one has to choose a larger puncturing period. If we choose $L_p = 8$, the code rate can be varied within the interval $1/2 \leq R'_c \leq 8/9$ in 8 steps.

The following figure shows the block diagram of a convolutional encoder with puncturing. The puncturing matrix, sometimes also called puncturing table, is applied to the encoder output bits according to a modulo rule. This means that the columns of the puncturing matrix $p_i$ are cyclically assigned to the consecutive code words $x(l) = [x_0(l) \ x_1(l) \ldots x_{n-1}(l)]$, i.e., $i = l \mod L_p$.

![Figure 5.1: $R_c = 1/2$ convolutional encoder with puncturing](image)

For systematic encoders, the puncturing matrix is usually only applied to the redundancy bits, leaving the systematic bits unaffected. The puncturing matrix $P$ is then of
dimensions \([(n-k) \times L_p]\) and contains \(l\) ones, where \(1 \leq l \leq (n-k)\). The interval of possible code rates becomes the same as in Eqn. (5.3).

For decoding punctured convolutional codes, both the transmitter and the receiver have to know the puncturing period and the pattern, i.e., the puncturing matrix. The decoder then inserts zeros in place of the punctured bits and just applies the decoder of the mother code.

When applying puncturing to Turbo Codes, we again only puncture the redundancy bits of the two encoders. Figure 5.2 shows the block diagram of a Turbo encoder of original rate \(R_c = 1/3\). The dimensions of the puncturing matrix are now \([(n_1 - k + n_2 - k) \times L_p]\) = \([(n_1 + n_2 - 2k) \times L_p]\) with \(k/n_1\) and \(k/n_2\) denoting the code rates of the two constituent encoders. The code rate is now

\[
R'_{c,TC} = \frac{k \cdot L_p}{k \cdot L_p + l},
\]

where \(l\) lies within the interval \(1 \leq l \leq (n_1 + n_2 - 2k) \cdot L_p\) and the interval of possible code rates becomes

\[
\frac{k \cdot L_p}{k \cdot L_p + (n_1 + n_2 - 2k) \cdot L_p} = \frac{k}{n_1 + n_2 - k} \leq R'_{c,TC} \leq \frac{k \cdot L_p}{k \cdot L_p + 1}.
\]

For the example in Fig. 5.2 and a puncturing period of \(L_p = 4\), we obtain possible code rates in the interval \(1/3 \leq R'_{c,TC} \leq 4/5\) with 8 steps and for \(L_p = 8\), \(1/3 \leq R'_{c,TC} \leq 8/9\).

In order to find good punctured Turbo Codes, one has to search through all possible puncturing patterns.
5.2 Rate-Compatible Punctured Turbo Codes

If punctured Turbo Codes are used for unequal error protection, they have to satisfy a rate-compatibility requirement. Let us assume, the encoder input contains information on the susceptibility to decoding errors. This means, there are blocks of data that have to be protected by a more powerful code than others. This information is called source significance information (SSI). These blocks are encoded by the same mother code but with different puncturing patterns $P_l$. Suppose, the information blocks are arranged by their importance, where the first block is the most important. The rate-compatibility restriction has already been given in [17] for rate-compatible punctured convolutional (RCPC) codes and is the same for Turbo Codes.

$$\text{if } p_{l_1,ij} = 1 \text{ then } p_{l_2,ij} = 1 \text{ for all } l_2 \geq l_1 \geq 1$$

(5.6)

or equivalently

$$\text{if } p_{l_1,ij} = 0 \text{ then } p_{l_2,ij} = 0 \text{ for all } l_2 \leq l_1 \leq (n_1 + n_2 - 2k) \cdot L_p .$$

(5.7)

This restriction means that the additional ones in the puncturing matrix of a stronger code can only be placed, where the puncturing matrix of the lower level code contains zeros. An example could be

$$P_1 = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \text{ and }$$

(5.8)

$$P_2 = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} .$$

(5.9)

One can see that the puncturing matrix $P_2$ of the stronger code only contains an additional one where $P_1$ has a zero. All other elements stay the same.

The rate-compatibility condition guarantees that in a transition between two puncturing matrices $P_1$ and $P_2$ no distance loss occurs. Let path 1 and path 2 be paths corresponding to the same information sequence, where path 1 lies completely within one block and path 2 crosses the transition from block $l_1$ to $l_2$ (see Fig. 5.3).

If the puncturing matrix $P_2$ would contain a zero, where the puncturing matrix $P_1$ contains a one, this one would be punctured by the code of block $l_2$ and result in a distance loss of the path. Condition (5.6) guarantees that this does not happen. On the other hand, it might happen that, due to less puncturing in block $l_2$, additional ones occur on part B of path 2, which would lead to an increase in distance $d_2$. 
5.3. Results

In this section, we present some results for punctured Turbo Codes. As constituent encoders, we used a rate $R_c = 1/2$ recursive systematic convolutional code with generators $g = [7, 5]_8$ and thus memory $m = 2$ and constraint length $L_c = 3$. We tried out puncturing periods $L_{p,1} = 3$, $L_{p,2} = 4$, and $L_{p,3} = 5$. Table 5.3 shows the possible code rates corresponding to the puncturing periods.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
\hline
$R'_c$ & 0.333 & 0.357 & 0.364 & 0.375 & 0.385 & 0.400 & 0.417 & 0.429 & 0.444 & 0.455 \\
\hline
$L_{p,1} = 3$ & 1/3 & & 3/8 & & & 3/7 & & & & \\
$L_{p,2} = 4$ & 1/3 & 4/11 & & 4/10 & & 4/9 & & & & \\
$L_{p,3} = 5$ & 1/3 & 5/14 & 5/13 & 5/12 & 5/11 & & & & & \\
\hline
\end{tabular}
\caption{Code rates corresponding to a rate $1/3$ code with $L_p = 3, \ldots, 5$}
\end{table}

One can see that we can construct a family of, in this example, 20 different rate codes from one mother code. We can, of course, achieve even higher resolutions with larger puncturing periods $L_p$.

We now show the EXIT charts of the mother Turbo Code with two equal constituent recursive systematic encoders with generators $g = [7, 5]_8$, code rate $R_c = 1/2$ and constraint length $L_c = 3$. The overall code rate of the Turbo Code is $R_{c,TC} = 1/3$. We apply three different puncturing schemes, i.e.,

\begin{equation}
\begin{align*}
P_1 &= \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}, \quad P_2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{and} \quad P_3 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix},
\end{align*}
\end{equation}
which lead to the codes \( C_1, C_2, \) and \( C_3 \). Let the unpunctured mother code be denoted by \( C \). The code rates of the punctured Turbo Code are

\[
R_{c,TC;1} = \frac{k \cdot L_{p,1}}{k \cdot L_{p,1} + l_1} = \frac{3}{8} = 0.375 ,
\]

(5.11)

\[
R_{c,TC;2} = \frac{k \cdot L_{p,2}}{k \cdot L_{p,2} + l_2} = \frac{3}{7} \approx 0.429 ,
\]

(5.12)

and

\[
R_{c,TC;3} = \frac{k \cdot L_{p,3}}{k \cdot L_{p,3} + l_3} = \frac{3}{6} = 0.5 .
\]

(5.13)

We used the Log-MAP algorithm for decoding and a frame length of 8192 bits. Figure 5.4 shows the EXIT chart of the mother code of rate \( R_{c,TC} \approx 0.333 \) for \( E_b/N_0 = 0.5 \) dB.

![EXIT chart of the mother Turbo Code C](image)

**Figure 5.4:** EXIT chart of the mother Turbo Code \( C \)

The trajectories of the decoding iterations do not completely match the transfer curves due to finite interleaver size and termination of only one of the codes. The deviation decreases for larger interleaver sizes. The decoder converges after 10 iterations.

If we now apply the puncturing scheme \( P_1 \), we increase the code rate to \( R_{c,TC;1} = 0.375 \) and suffer a performance degradation. This can be seen in Fig. 5.5, which shows the EXIT chart of the punctured code \( C_1 \) again for \( E_b/N_0 = 0.5 \) dB. This code, however, still converges but needs more iterations (15).
Consider Fig. 5.6. We now use puncturing scheme $P_2$ with $R_{c,TC,2} \approx 0.429$. We observe that the transfer curves intersect before reaching the point $(1,1)$, which means that the decoder gets “stuck” at this $E_b/N_0$. We have to increase the $E_b/N_0$ in order to let the decoder converge.

Figure 5.6: EXIT chart of punctured Turbo Code $C_2$

Figure 5.7 shows the EXIT chart of a code with even more punctured bits and thus even higher code rate, i.e., $R_{c,TC,3} = 0.5$. The intersection point moves even closer to
the point (0,0) and this code would require even higher $E_b/N_0$ than code $C_2$ in Fig. 5.6.

Figure 5.7: EXIT chart of punctured Turbo Code $C_3$

Figure 5.8 depicts the bit-error rate curves of the four codes with code rates $R_{c,TC}$, $R_{c,TC1}$, $R_{c,TC2}$, and $R_{c,TC3}$. As expected, the mother code has the lowest, and the punctured code with $P_3$ the highest bit-error rate.

From figures 5.4 and 5.5, we conclude that the waterfall regions of $C$ and $C_1$ must be located at approximately $E_b/N_0 = 0.5$ dB, where the one of code $C$ is at little lower $E_b/N_0$, since the transfer curves are more open than those of code $C_1$. The waterfall region of code $C_2$ in Fig. 5.6 is located at a little higher $E_b/N_0$, since it almost converges at $E_b/N_0 = 0.5$ dB. From the EXIT chart in Fig. 5.7, we can only predict that the waterfall region will lie at higher $E_b/N_0$ than the others. These observations are well confirmed in Fig. 5.8.

Figure 5.8 additionally contains the so-called Shannon limits, represented by vertical lines in the style of the corresponding curves. The channel capacity of an AWGN channel is given by

$$C = \log_2 \left( 1 + \frac{E_s}{N_0} \right).$$

Since the code rate is always smaller or equal to the channel capacity, we have

$$R_c \leq \log_2 \left( 1 + R_c \cdot \frac{E_b}{N_0} \right),$$
5.3. RESULTS

which leads to

\[ \frac{E_b}{N_0} \geq \frac{2^{R_c} - 1}{R_c}. \] (5.16)

This \( \frac{E_b}{N_0} \) gives us the limit, below which we cannot reach error-free transmission with a code of rate \( R_c \), even with unlimited coding effort. For the codes in Fig. 5.8 with code rates \( R_{c,TC} \approx 0.333, R_{c,TC,1} = 0.375, R_{c,TC,2} \approx 0.429, \) and \( R_{c,TC,3} = 0.5 \), the Shannon limits are approximately \(-1.08\) dB, \(-1.01\) dB, \(-0.93\) dB, and \(-0.82\) dB. We only inserted the limits for the mother code and for the worst code into the figure. Thus, the distance of the bit-error curves to the Shannon limit lies between approximately 2 dB and 5 dB. Since this distance is quite large, a future task will be to find better codes with smaller distances to the Shannon limit. Until now, we have just investigated the principles and effects of puncturing but did not optimise the code itself.

Figure 5.8: Bit-error curves of differently punctured codes

![Diagram showing bit-error curves for different punctured codes.](image-url)
Chapter 6

UEP: Pruned Turbo Codes

In Chapter 5, we introduced the principles of puncturing in unequal error protection schemes. The main idea was to change the code rate $R_c = k/n$ by puncturing some of the output bits, i.e., modifying the denominator of the code rate. Again, the aim is to find a family of codes instead of a set of completely different codes. When applying pruning, we, like in Chapter 5, construct sub-codes of different code rates from a single mother code. We change the code rate by modifying the numerator instead of the denominator. In the following section, we explain, how this modification is carried out and what consequences it has for the code properties.

6.1 Principles of Pruning

We first explain a procedure given in [9]. The aim is to construct a sub-code of rate $R_c^{(1)} < R_c^{(0)}$, where $R_c^{(0)}$ is the code rate of the mother code. This means that

$$R_c^{(1)} = \frac{k^{(1)}}{n} < R_c^{(0)} = \frac{k^{(0)}}{n}$$

This implies that the $k^{(0)} \geq 2$ in order to be able to construct at least one class of sub-codes with rate $R_c^{(1)} = 1/n$. Let the mother code and the sub-code and their generators be denoted by $C^{(0)}$, $G^{(0)}(D)$, and $C^{(1)}$, $G^{(1)}(D)$, respectively. $k^{(0)}$ and $k^{(1)}$ are the respective numbers of input bits. The two codes shall possess the following property: Each valid code sequence of code $C^{(1)}$ also belongs to the set of code sequences of code $C^{(0)}$, i.e.,

$$\forall I^{(1)} \exists I^{(0)} : I^{(1)} \cdot G^{(1)} = I^{(0)} \cdot G^{(0)} ,$$

(6.2)
where $I^{(0)}$ and $I^{(1)}$ are arbitrary information sequences of code $C^{(0)}$ and $C^{(1)}$, respectively. One possibility of determining $G^{(1)}$ from given $G^{(0)}$, is to define the components of

$$I^{(0)} = \left( I^{(0)}_1, I^{(0)}_2, \ldots, I^{(0)}_{k^{(0)}} \right)$$

(6.3)

as shifted versions of $I^{(1)}$. For $k^{(1)} = 1$, we have

$$I^{(0)}_h = I^{(1)} \cdot D^{j_h} \quad \text{where}$$

$$k^{(1)} = 1, \ h = 1 \ldots k^{(0)}, \ j_h \in \{0, \ldots, L^{(1)} - 1\}$$

(6.4)

(6.5)

with $L^{(0)}_c$ denoting the constraint length of code $C^{(0)}$. There has to exist at least one

$$I^{(0)}_h = I^{(1)}, \ i.e., \ j_h = 0$$

(6.6)

in order to express the term ”1” appearing in code $G^{(0)}$. In order to achieve the same number of states for both codes, $k^{(1)} \cdot L^{(1)}_c = k^{(0)} \cdot L^{(0)}_c$ must be fulfilled. This leads to the condition that at least one

$$I^{(0)}_h = D^{L^{(1)}_c - L^{(0)}_c} \cdot I^{(1)}$$

(6.7)

has to exist in order to generate the same number of required memory cells. We will now present an example, where we calculate $G^{(0)}$ from $G^{(1)}$ and vice versa.

Let $G^{(1)}(D)$ be

$$G^{(1)}(D) = \begin{bmatrix} 1 & 1 + D + D^2 & 1 + D + D^2 \end{bmatrix}.$$  

(6.8)

We define the input $I^{(0)}$ to code $C^{(0)}$ as

$$I^{(0)}_1 = I^{(1)} \text{ and } I^{(0)}_2 = D \cdot I^{(0)}$$

(6.9)

and specify $k^{(1)} = 2$. From Eqn. (6.2) we obtain the following equations.

$$G^{(0)}_{11} + D \cdot G^{(0)}_{21} = G^{(1)}_1 = 1$$

(6.10)

$$G^{(0)}_{12} + D \cdot G^{(0)}_{22} = G^{(1)}_2 = 1 + D + D^2$$

(6.11)

$$G^{(0)}_{13} + D \cdot G^{(0)}_{23} = G^{(1)}_3 = 1 + D + D^2$$

(6.12)

This system of equations allows several possible solutions. One of them is
\[ \mathbf{G}^{(0)} = \begin{bmatrix} 1 & 1 + D & 1 \\ 0 & D & 1 + D \end{bmatrix}. \quad (6.13) \]

Assume \( \mathbf{G}^{(0)} \) now to be known and \( \mathbf{G}^{(1)} \) shall be computed. The generators on the left side of equations (6.10)-(6.12) are known and used for computing the generators on the right side, which yields

\[ \begin{align*}
G_1^{(1)} &= G_{11}^{(0)} + D \cdot G_{21}^{(0)} = 1 + D \cdot 0 = 1 \quad (6.14) \\
G_2^{(1)} &= G_{12}^{(0)} + D \cdot G_{22}^{(0)} = 1 + D + D \cdot D = 1 + D + D^2 \quad (6.15) \\
G_3^{(1)} &= G_{13}^{(0)} + D \cdot G_{23}^{(0)} = 1 + D \cdot (1 + D) = 1 + D + D^2 \quad (6.16)
\end{align*} \]

and results in

\[ \mathbf{G}^{(1)}(D) = \begin{bmatrix} 1 & 1 + D + D^2 & 1 + D + D^2 \end{bmatrix}. \quad (6.17) \]

For \( k^{(1)} \geq 1 \), the construction code \( C^{(1)} \) or \( C^{(0)} \), respectively, is carried out by specifying

\[ I_i^{(0)} = \sum_{h=1}^{k^{(1)}} f_{hi} \cdot I_h^{(1)} \cdot D^{j_{hi}}, \quad (6.18) \]

where \( i = 1 \ldots k^{(0)} \), \( h = 1 \ldots k^{(1)} \), and \( j_{hi} \in \{0, \ldots, L^{(1)} - L^{(0)}\} \). \( f_{hi} \) is either 0 or 1 for input sequence \( I_h^{(1)} \) in the \( i \)-th equation. At least one \( j_{hi} \) has to be zero and one has to equal \( L^{(1)} - L^{(0)} \), for the above explained reasons.

In [10], the method from [9] is also explained, but in a more general setting. Equation (6.4) is now rewritten, expressing \( I^{(0)} \) by \( I^{(1)} \) and a so-called path locating matrix \( \Theta(D) \).

\[ I^{(0)} = I^{(1)} \cdot \Theta(D) \quad (6.19) \]

Following the approach from [9], the path locating matrix contains the elements \( D^{j_{hi}} \). In [10], \( \Theta(D) \) is more general. The elements \( \theta_{u,v}(D) \) are chosen to be polynomials of maximal degree \( L^{(1)} - L^{(0)} \).

In our example, the path locating matrix would be \( \Theta(D) = [1 \ D] \).

Inserting Eqn. (6.19) into Eqn. (6.2), yields

\[ \begin{align*}
I^{(1)} \cdot \mathbf{G}^{(1)} &= I^{(1)} \cdot \Theta \cdot \mathbf{G}^{(0)} \quad (6.20) \\
\mathbf{G}^{(1)} &= \Theta \cdot \mathbf{G}^{(0)} \quad (6.21)
\end{align*} \]

In general, the generator matrices of the mother code and the sub-code are of dimensions \([k^{(0)} \times n]\) and \([k^{(1)} \times n]\), respectively. Therefore, the path locating matrix \( \Theta(D) \) has to be of dimension \([k^{(1)} \times k^{(0)}]\).
Thus, we can calculate a sub-code of rate $R^{(1)}_c = k^{(1)}/n$ from a mother code of rate $R^{(0)}_c = k^{(0)}/n$ by pre-multiplying the generator matrix of code $C^{(0)}$ by $\Theta(D)$.

In the case of pruned convolutional codes, one can clearly realise the meaning of mother code and sub-code by investigating a trellis segment. The effect of the path locating matrix is that fewer paths in the trellis are available due to a smaller number of input bits. This leads to pruning away trellis paths. The next figure shows two trellis segments of an unpruned code and of the corresponding pruned sub-code with generators $G^{(0)}(D)$ and $G^{(1)}(D)$, respectively. The generators are taken from the above example, i.e.,

$$G^{(0)}(D) = \begin{bmatrix} 1 & 1+D & 1 \\ 0 & D & 1+D \end{bmatrix} \quad \text{and} \quad G^{(1)}(D) = \begin{bmatrix} 1 & 1+D+D^2 & 1+D+D^2 \end{bmatrix}. \quad \text{(6.22)}$$

One can see that the trellis segment of the pruned code on the right is contained within the trellis segment of the unpruned mother code on the left. The "surviving" paths are illustrated in the trellis of the mother code by bold transitions where the pruned paths are represented by dashed lines. The inputs and corresponding outputs are listed in front of the nodes in the order of paths leaving the state. The larger figures represent the inputs and outputs of the paths that will be left after pruning.

**Figure 6.1:** Trellis segments of an unpruned (left) and a pruned (right) code

Furthermore, we can see that the inputs of the pruned code correspond to the first input bit of the mother code. The path locating matrix thus determines which paths will survive and which are pruned away and therefore is called *path locating matrix*. 
Apart from changing the code rate $R_c^{(0)}$, the additional aim of pruned codes is to prune away those paths in the trellis, which lead to low distances. The corresponding path locating matrices have to be determined by exhaustive computer search.

For pruned Turbo Codes, we apply pruning to the constituent convolutional encoders. We choose equal codes for both single encoders. Generally, from a convolutional mother code of rate $R_c^{(0)} = k^{(0)}/n$, we are able to construct $k^{(0)} - 1$ sub-codes of rates $R_c^{(i)}$, where $1 \leq k^{(i)} \leq k^{(0)} - 1$. The overall code rate of the Turbo Code with the mother codes as constituent encoders is

$$R_{c,TC}^{(0)} = \frac{k^{(0)}}{2 \cdot n - k^{(0)}}$$

(6.24)

and the possible code rates of the pruned Turbo Codes are

$$\frac{1}{2 \cdot n - 1} \leq R_{c,TC}^{(i)} \leq \frac{k^{(0)} - 1}{2 \cdot n - k^{(0)} + 1}.$$  

(6.25)

For example, if the unpruned convolutional code is of rate $R_c^{(0)} = 2/3$ and the pruned convolutional code forces the code rate to decrease to $R_c^{(1)} = 1/3$, the code rates of the unpruned and of the pruned Turbo Codes are $R_{c,TC}^{(0)} = 2/4$ and $R_{c,TC}^{(1)} = 2/5$, respectively.

### 6.2 Single-Trellis Decoding

Like for punctured convolutional codes, we would like to decode all sub-codes with the same decoder, i.e., with the decoder of the mother code.

The condition for single-trellis decoding is that the path-pruned code and the mother code have the same number of memory cells and thus, the same number of possible states. This means that the trellises of both codes are of the same size and we just have to modify the possible transitions between the states.

A mathematical description of this condition is given in [10], but we will not discuss this in detail. This paper provides a listing of good pruned (and original) codes for $n = 3, 4, 5$, which fulfill the condition for single-trellis decoding.
6.3 Path-Compatible Pruned Turbo Codes

Similar to punctured Turbo Codes, we can apply pruned Turbo Codes for unequal error protection. Suppose, there are $L$ blocks of different importance. We thus generate a family of $L$ codes by pruning a mother code. Let $\Gamma_i$ be the set of all paths in one trellis segment of code $C^{(i)}$. When switching between the different sub-codes, the code can suffer from path discontinuities.

To explain this in more detail, assume two sub-codes with $\Gamma_i$ and $\Gamma_{i+1}$ being the path collections of two subsequent blocks of different importance. If the two trellises are not compatible, the transition from $C^{(i)}$ to $C^{(i+1)}$ might not be possible. The encoding path would suffer from a path discontinuity. This would mean that this code combination is not realisable. Figure 6.2 illustrates the situation. The trellises are just an arbitrary example and not constructed from real codes. If the two trellis segments are placed next to each other, some paths of $\Gamma_i$ cannot be picked up after the transition from Code $C_i$ to $C_{i+1}$ and a path discontinuity results.

![Figure 6.2: Incompatible trellises](image)

To overcome these problems, we define a path-compatibility criterion as follows. Let $(\Gamma_1, \Gamma_2, \ldots, \Gamma_L)$ be a family of sub-codes, where the required average bit-error rates would be related as $P_{e,1} \geq P_{e,2} \geq \ldots \geq P_{e,L}$. These codes are path-compatible if, and only if,

$$\Gamma_L \subseteq \Gamma_{L-1} \subseteq \ldots \subseteq \Gamma_1 .$$

(6.26)

Let us assume that $\Gamma_2 \subseteq \Gamma_1$ is valid. The distance can be bounded similar to the observations from Fig. 5.3. Consider Fig. 6.3.
6.4 Results

In this section, we present some results for pruned Turbo Codes. We generally use equal encoders as constituent encoders. A table of mother codes and their pruned codes with good distance properties is given in [10]. We consider an example with a mother code of rate \( R_c^{(0)} = 2/3 \) and \( m = 2 \) memory cells. The mother generator matrix for the two component codes of the Turbo Code is

\[
G^{(0)}(D) = \begin{bmatrix}
1 & 1 + D & 1 + D \\
1 + D & D & 1 + D
\end{bmatrix}.
\] (6.28)

The path locating matrix is

\[
\Theta(D) = \begin{bmatrix}
1 & D
\end{bmatrix}.
\] (6.29)
The path-pruned generator $G^{(1)}(D)$ is calculated by pre-multiplying the generator of the mother code $G^{(0)}(D)$ by the path locating matrix $\Theta(D)$, which yields

$$G^{(1)}(D) = \Theta(D) \cdot G^{(0)}(D)$$

$$= \begin{bmatrix} 1 & D \\ 1+D & D & 1+D \end{bmatrix}$$

$$= \begin{bmatrix} 1+D+D^2 & 1+D+D^2 & 1+D^2 \end{bmatrix}$$

Figure 6.4 shows the trellis segments of the mother code and of the path-pruned code. We see that the paths in the pruned trellis segment are a selection from the paths in the unpruned trellis segment. Furthermore, the input bits of the pruned code are equal to the first input bit of the unpruned code.

![Trellis segments of the unpruned (left) and the pruned (right) code](image)

**Figure 6.4:** Trellis segments of the unpruned (left) and the pruned (right) code

Figures 6.5 and 6.6 show the EXIT charts of the mother code $G^{(0)}$ and the path-pruned sub-code $G^{(1)}$ at $E_b/N_0 = 0.8$ dB and $E_b/N_0 = -1.2$ dB, respectively. The mother code and the sub-code need 10 and 6 decoding iterations to reach the point (1,1), respectively.
Figure 6.5: EXIT chart of the unpruned Turbo Code

Figure 6.6: EXIT chart of the path-pruned Turbo Code
Figure 6.7 shows the bit-error curves of both codes. One can see, that the performance of the pruned code outperforms the mother code by approximately 1 dB at a bit-error rate of $10^{-4}$.

Like in Section 5.3, we inserted the Shannon limits of both codes as vertical lines. They are approximately $-0.55$ dB and $-1.02$ dB for the mother code and for the path-pruned code, respectively. The distance of the respective code to its Shannon limit is about $2.5$ dB and $3.5$ dB for the path-pruned code and for the mother code, respectively.

Let us consider another example. We denote the mother code and the path-pruned codes by $C^{(A)}$ and $C^{(B)}$ to distinguish this example from the last. Let the generators of the mother code and the path locating matrix be

$$G^{(A)}(D) = \begin{bmatrix} D & D & 1 + D \\ 1 + D & 1 & 0 \end{bmatrix} \quad (6.33)$$

and

$$\Theta(D) = \begin{bmatrix} D & 1 \end{bmatrix}, \quad (6.34)$$

respectively. Thus, the path-pruned code is given by $\Theta(D) \cdot G^{(A)}(D)$, which yields
\[ G^{(B)}(D) = \Theta(D) \cdot G^{(A)}(D) \]  \hfill (6.35)

\[ = \left[ \begin{array}{c} D & 1 \\ \end{array} \right] \cdot \left[ \begin{array}{ccc} D & D & 1 + D \\ 1 + D & 1 & 0 \\ \end{array} \right] \]  \hfill (6.36)

\[ = \left[ \begin{array}{ccc} 1 + D + D^2 & 1 + D^2 & D + D^2 \end{array} \right] . \]  \hfill (6.37)

The bit-error rates of these codes are depicted in Fig. 6.8, again containing the Shannon limits.

![Figure 6.8: Bit-error curves of the mother code and the path-pruned code](image)

For this example, the enhancement due to path-pruning is even larger than in the first example. The pruned code achieves a \( E_b/N_0 \)-gain of approximately 4 dB at a bit-error rate of \( 10^{-5} \) compared to its mother code. The distances of the mother and the pruned codes to their Shannon limits are about 6 dB and 2 dB, respectively.

It should be noted that the last two examples are codes with good properties according to [10]. Nevertheless, these codes have not been optimised when embedding them into Turbo Codes but only as single convolutional codes. The optimisation of the pruned component codes has not been done so far. One could, e.g., use different pruned codes as component codes in order to achieve asymmetric Turbo Codes.
We now consider a scenario, where we transmit one frame which is divided into two blocks of different importance. Let us assume the first block of length $K_1 = 256$ to be more important. It will be encoded by the sub-code of rate $R_c = 1/3$ from the first of the two above examples, i.e.,

$$G_{\text{block}1}(D) = G^{(1)} = \left[ \begin{array}{ccc} 1 + D + D^2 & 1 + D + D^2 & 1 + D^2 \end{array} \right].$$

(6.38)

The second, less important block of length $K_2 = 512$ is encoded by the corresponding mother code.

$$G_{\text{block}2}(D) = G^{(0)} = \left[ \begin{array}{ccc} 1 & 1 + D & 1 + D \\ 1 + D & D & 1 + D \end{array} \right].$$

(6.39)

Figure 6.9 shows the bit-error rates of the bits near the transition region from the first to the second block at $E_b/N_0 = 1$ dB. The bit-error rates do not correspond to those in Fig. 6.7 due to quite small interleaver sizes, which leads to a performance degradation.

![Figure 6.9: Transition region of two unequally important blocks](image)

We see that the bit-error rates of the bits around the transition region are between the bit-error rates of the two blocks. This corresponds to the observation in Section 6.3 that the minimum distance of a transitional path is lower bounded by the free distance of the weaker code. Of course, it is upper bounded by the free distance of the stronger code. This means that the average bit-error rate of a transitional path lies between the bit-error rates of the two blocks, which is well confirmed in Fig. 6.9.
Chapter 7
Conclusions

In this work, we discussed possible solutions for achieving unequal error protection based on Turbo Codes. There exist several reasons that motivate the application of unequal error protection, for example if some kind of media should be handled by different terminal equipment, or if the bit-error rates have to be adapted to channel conditions.

One possibility of applying unequal error protection to Turbo Codes is puncturing, which means not transmitting all of the generated bits of a code sequence but puncturing a certain amount of them. This method enlarges the code rate $R_c = k/n$ by reducing the denominator, i.e., the number of output bits of the code. The increase of the code rate leads to a performance degradation, i.e., to a less powerful code. Thus, we can construct a family of unequally protecting codes by applying different puncturing patterns.

As another option, we studied path-pruning. In this case, the code rate $R_c = k/n$ of a mother code is modified by varying the numerator, i.e., the number of input bits. We presented a procedure for constructing a more powerful code from a mother code, where the sub-code has a smaller number of information bits and the set of transitions in a trellis segment of the sub-code is a subset of the transitions of the mother code. If we follow a condition for ”single-trellis decoding”, we only have to modify the state transitions of the trellis and are able to use the decoder of the mother code for decoding the sub-codes.

For punctured and pruned Turbo Codes, we defined a rate-compatibility and a path-compatibility criterion, respectively, which guarantee that in transition regions between different sub-codes no distance losses or path discontinuities can occur.

When comparing the properties and results of punctured and pruned Turbo Codes, we should, first of all, note that both methods achieve unequal error protection without additional complexity. In both cases, we are able to decode all sub-codes by the
decoders of their mother codes and therefore, do not have to implement additional decoders.

One advantage of puncturing compared to pruning is that we can achieve a large number, i.e., a high resolution of different code rates just by enhancing the length of the puncturing period. When applying pruning to a mother code of rate $R_c = k/n$, we are able to construct only $k - 1$ sub-codes, which is usually restricted to a small number, since we assume small component codes for Turbo Codes.

Regarding the performance of punctured and pruned Turbo Codes, the obtained results were expected. For a family of $N$ codes with code rates $R_{c,1}$, $R_{c,2}$, ..., and $R_{c,N}$ with $R_{c,1} > R_{c,2} > \ldots > R_{c,N}$, we obtain staggered bit-error rate curves, which are related according to $P_{b,1} > P_{b,2} > \ldots > P_{b,N}$. The exact gain is, of course, dependent on the properties of the used codes and on the system components, e.g., on the interleaver size.

We do not list the gains again, but would like to mention that further optimisation of the constituent codes needs to be done. However, the principal behaviour shows up as expected and desired.

As future tasks, we may especially investigate pruned Turbo Codes in more detail. The optimization of possible component codes has not yet been done. One can, for example, apply differently pruned codes as component codes. Furthermore, the evaluation of source significance information and appropriate assignment to certain codes has to be investigated.
Bibliography


