

Digital communications – collection of solved problems and examples

M. Rakus, T. Palenik

2024

Digital communications – collection of solved problems and examples

M. Rakus, T. Palenik

2024

Published by
Emona Instruments Pty Ltd,
78 Parramatta Road
Camperdown NSW 2050
AUSTRALIA.

web: www.emona-tims.com
telephone: +61-2-9519-3933

First published 2024
ISBN 978-1-921903-15-1

Copyright © 2024 Emona Instruments Pty Ltd and its related entities. All rights reserved.

Foreword

This book is intended as complementary to the widely known venerable books by esteemed authors such as Digital Communications by John Proakis or Digital Communications – fundamentals and applications by B. Sklar, with an emphasis on maximal streamlining. The target audience is students new to the area of digital communications, who will find the minimal theoretical background here, along with basic exercises. There are two types of exercises we think are necessary for the basic comprehension of the areas covered: the manual exercises, calculated on paper by hand, and a complementary MATLAB exercises to be programmed. Some exercises are also possible to model using TIMS (Telecommunications Instructional Modelling System by EMONA - Australia). In this cases students can get a true hands-on experience with real signals and systems. We define the exercise for the commercial MATLAB platform, but are careful to use only a subset of MATLAB, focusing on built-in functionality and intentionally avoiding the use of toolboxes, in order to facilitate the use of the free and open source platform GNU Octave as well. In fact, most of the MATLAB exercises implementations were also tested on GNU Octave. We provide our reference implementations on our GitHub page under the GPL3.0 license:

<https://github.com/talenik/Digital-Communications-Problems-Solved.git>

Aknowledgements

This manuscript was supported by the Slovak Research and Development Agency under Contract no. APVV-19-0436 (New information and communication technologies for the future information infrastructure).

The content of this textbook was reviewed by Ing. Matus Turcsany, PhD., and by Ing. Michal Polasko, PhD..

Contents

Foreword	5
Introduction	11
Chapter 1: MATLAB recap.....	12
Lab 1.1: MATLAB basics	12
Lab 1.2: Matrices and vectors in MATLAB	13
Lab 1.3: Matrix partitioning	15
Chapter 2: Source coding	19
Universal coding.....	20
LZ77– sliding window	20
LZ78 – dictionary method	24
Entropic coding	26
Chapter 3: Channel coding	33
Linear Block Codes.....	33
Hamming codes	36
LBC decoding.....	38
Hardware realization of cyclic codes	48
Convolutional codes	52
Description of convolutional codes	53
Error correction capabilities of convolutional codes	57
Decoding of convolutional codes	61
Convolutional codes with catastrophic error propagation	64
Lab 3.1: MATLAB exercises - Hamming codes	65
Chapter 4: Signal analysis	67
Signals and their characteristics	67
Autocorrelation function and spectral density	74
Signal bandwidth	77
GFT – Generalized Fourier Transform.....	80
Gram–Schmidt orthogonalization procedure	83
Lab 4.1: MATLAB exercises - Signals and signal processing	91
Chapter 5: Digital modulation	105
Binary modulations:	105
Coherent BPSK (Binary Phase Shift Keying):	105
Differential BPSK: BDPSK	106
Non-coherent BFSK	106
M-ary modulations	106

Coherent MPSK	106
Non-coherent MPSK: DMPSK	107
Coherent MFSK	107
Non-coherent MFSK	107
M-QAM.....	107
Lab 5.1: MATLAB exercises: BPSK & 4QAM in signal space	113
Chapter 6: Multiple Access (MA)	117
FDMA	117
TDMA	118
SDMA	119
PDMA	119
CDMA	119
FH-CDMA.....	122
DS-CDMA.....	127
Chapter 7: Communication channels.....	143
BSC channel	143
Joint probability distribution of the BSC channel	144
AWGN Channel	147
Lab 7.1: Random number generation, random noise generation, properties of AWGN	151
Chapter 8: Signal Detection	165
Detection of binary signals under the presence of AWGN	165
Analog MF	166
Digital MF	169
MAP and ML decision criteria	172
Error probability of binary UNRZ signaling in AWGN channel	175
Error probability of binary BPNRZ signaling in AWGN channel	176
Inter-symbol interference (ISI).....	177
Lab 8.1: MATLAB exercises: Simulations of communications systems.....	181
Chapter 9: Reed-Solomon Codes	190
Galois fields.....	190
Reed Solomon codes	193
RS codes encoding	193
RS codes decoding	197
Chapter 10: LDPC codes in modern communications standards	204
QC-LDPC codes parity check matrix.....	204
CQ-LDPC codes in modern communications standards	211

Lab 10a – MATLAB exercises: uncompressing H matrix	214
Direct encoding of QC-LDPC codes.....	216
Direct encoding Method I: matrix algorithm.....	217
Direct encoding Method II: Low-level algorithm	218
Lab 10b MATLAB exercises: QC-LDPC Encoding.....	222
Decoding LDPC codes	225
Tanner graph.....	225
LLR – the Log-Likelihood ratio.....	227
The Min-Sum decoding algorithm	228
Lab 10c: MATLAB exercises: Min-Sum decoding of LDPC codes.....	235
MATLAB Exercise 10.17 - find $N(m)$ and $M(n)$ sets.....	235
Debugging	236
MATLAB Exercise 10.21: LDPC encoded transmission in AWGN channel – simulation results	238
List of acronyms.....	239
References	243

Introduction

In the following we will assume a digital communication system as shown in the figure bellow. Each chapter of this textbook is devoted to one block from the block diagram of the digital communication system. Selected problems are analyzed also by simulation using MATLAB software. At the end of some exercises there is a note that the analyzed problem can be also modeled using Emona TIMS hardware.

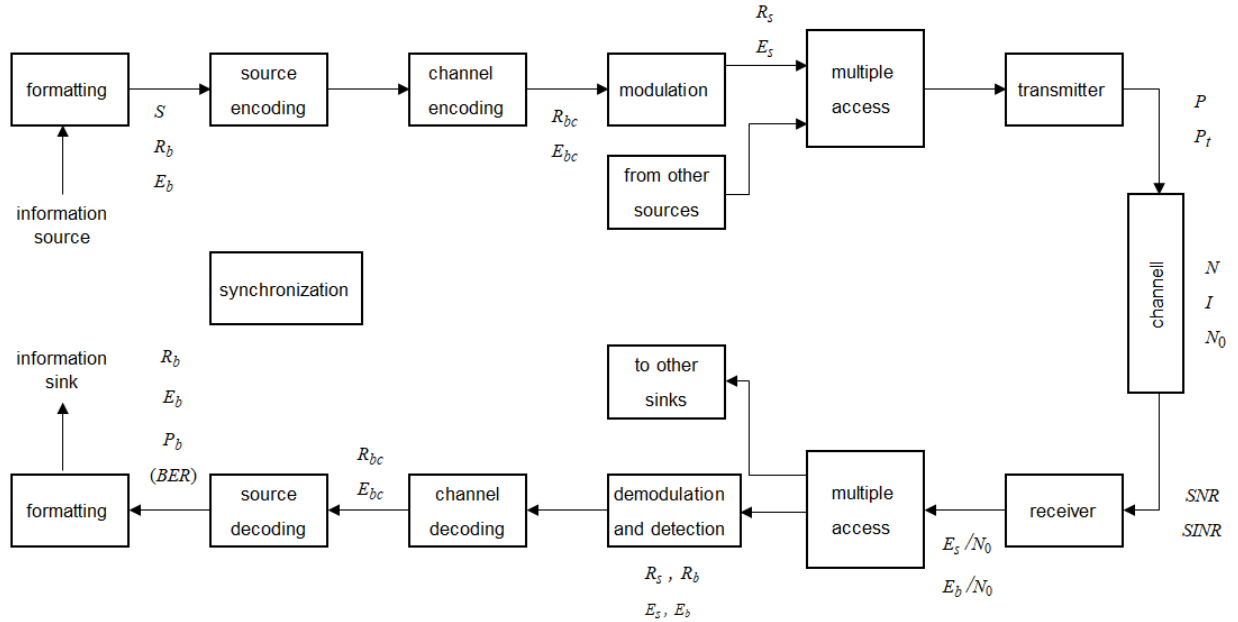


Fig. 1: Block diagram of digital communication system.

The following text covers only selected topics of the digital communication chain. The analyzed problems should serve students as a basic starting platform in their further study to achieve a deeper understanding of digital communications.

Chapter 1: MATLAB recap

It is assumed the reader has some basic MATLAB experience. However, to make sure, we provide some simple exercises to make sure the reader is with us. You can find the reference implementations of these exercises in the accompanying source files `lab1x.m`. Inexperienced reader is encouraged to read the Getting started section within the official MATLAB documentation portal: <https://www.mathworks.com/help/matlab/getting-started-with-matlab.html>

Lab 1.1: MATLAB basics

MATLAB Exercise 1.1.1: Variables and expressions basics.

- Define a variable a as a scalar with value 2π .
- Define a variable $\mathbf{v}$ as a vector of integer values 1, 2 ... 8.
- Define a variable $\mathbf{A}$ of size 2x2 as a matrix of any integer values.
- Copy variable a to a new variable b . Also copy $\mathbf{v}$ to a new variable $\mathbf{u}$.
Copy also $\mathbf{A}$ to a new matrix $\mathbf{B}$.
- Create a new variable $c = a + 2b$, and a new variable $\mathbf{w} = \mathbf{v} + 2\mathbf{u}$, and a new variable $\mathbf{C} = \mathbf{A} + 2\mathbf{B}$.
- Display a list of workspace variables along with their sizes.
- Try evaluating the expressions: $a + v$, $a + A$, $a * v$, $a * A$, $A + v$, $v + u'$, explain what's happening.
- Delete variable a from a workspace.

MATLAB Exercise 1.1.2: Calculation of values of various expressions.

- Given the theoretical formula and table of values of X_i and $p(X_i)$:

$$H(X) = -\sum_{\forall X_i} p(X_i) \cdot \log_2(p(X_i)) = -\sum p_i \log_2(p_i)$$

calculate the value of $H(X)$ using the values of $p(X_i)$ provided below:

X_i	0	1	2
$p(X_i)$	1/2	1/4	1/4

- Evaluate the following expression for $x = 1, 2$, and π :

$$f(x) = \sqrt{\frac{e^{-2\left(\frac{x-3}{x-2}\right)}}{2\pi i}}$$

MATLAB Exercise 1.1.3: Trying different display formats and precision.

Try the following commands:

```
>>format compact
>>e = 0.000000000000000022204
>>eps
>>e == eps
```

Explain what's wrong. Hint: try changing the output format.

What are all the supported formats?

MATLAB Exercise 1.1.4: Compare with tolerance.

Write expressions that compare $\sin(\pi)$ and $\tan(\pi/2)$ with their theoretical values. Be careful, with the usual comparison operator "=", compare the difference with a defined tolerance instead.

MATLAB Exercise 1.1.5: various versions of transpose

Create a complex matrix **B** and compare various transpose options:

```
>>B = [ 16 2+i 3-2i ; 5 11 10 ]  
>>C = B'  
>>D = B.'  
>>E = conj(B)
```

Which one is true: **C == D** ? **D == E** ? **E == C** ? Is it easy to compare matrices this way?

```
>>format short
```

Use the documentation online to find the proper function for simple test of matrix equality.

MATLAB Exercise 1.1.6: Solving systems of linear equations:

Find the solution of the following system using one line of code:

$$\begin{aligned} 3x + 2y - z &= 1 \\ 2x - 2y + 4z &= -2 \\ -x + \frac{1}{2}y - z &= 0 \end{aligned}$$

Lab 1.2: Matrices and vectors in MATLAB

Try help for the following built-in functions: `whos`, `rand`, `rank`, `det`, `fliplr`, `floor`, `zeros`, `eye`, `size`, `sum`. Do NOT use any loops (`for` / `while`) in your implementation. Use the vector features of MATLAB instead. You can find the reference implementations of these exercises in the accompanying source file `lab1b.m`.

MATLAB Exercise 1.2.1:

Using range operator and choose an integer parameter N stored in a variable to generate:

- A column vector **u** with values 0 to $N-1$.
- A row vector **v** with values N to $2N-1$.
- A row vector **w** with N values with first element $w_1 = 3$ and each other element $w_{i+1} = w_i + 2$.
- Using only built-in transpose and multiply operators, compute *inner* and *per-element* products of vectors **u** and **v**.

MATLAB Exercise 1.2.2:

Generate a pseudo-random square matrix of integers stored as double **A** of size $m = 4$. Matrix must contain only single-digit integer values.

- Find the rank and determinant of matrix **A**.
- Find the sums of all columns of **A**, the sums of all rows of **A**, and the sum of all elements of **A**.
- Flip **A** in horizontal and vertical direction, rotate the matrix by 90 degrees and transpose the matrix. Compare the results.

MATLAB Exercise 1.2.3:

Use range indexing on matrix **A** to:

- Copy the second column of matrix **A** to a column vector **v**.
- Copy the first row of matrix **A** to a row vector **u**.

- c) Copy the last row of matrix **A** to a row vector **w**.
- d) Copy the left half of the matrix **A** to a new matrix **A₁**.
- e) Copy the bottom half of the matrix **A** to a new matrix **A₂**.
- d) Using range indexing create a:
 - i. row vector **v₁** containing all the elements of matrix **A** scanned *column-wise*.
 - ii. row vector **v₂** containing all the elements of matrix **A** scanned *row-wise*.

MATLAB Exercise 1.2.4:

By using logical indexing:

- e) find all the elements of matrix **A** that are greater than 5.
- f) Copy all these elements to matrix **C** of same size as **A**, that will contain these elements at their proper positions and all other elements will be zero.
- g) Set all elements in matrix **A**, that are greater than 5, to zero.

MATLAB Exercise 1.2.5:

Using range indexing, cut the matrix **A** (of size 4x4) to submatrices as shown in the picture below and then transpose **A** per partes – by transposing and repositioning the pieces S1 – S6 into an all-zero matrix **D** of the same size as **A**. Note: it will be helpful to draw the transposed matrix **A**, along with the submatrix partitioning first.

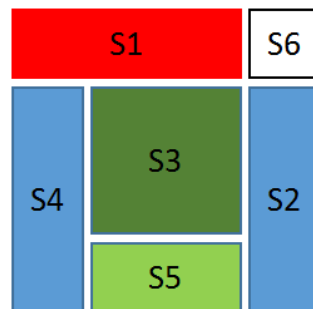


Fig. 1.1: Partitioning of matrix **A** to submatrices.

Divide this task into two separate m-files/functions with the following signatures:

```
function [ S1, S2, S3, S4, S5, S6 ] = foo( A )
```

divides the matrix **A** into submatrices.

```
function D = bar( S1, S2, S3, S4, S5, S6 )
```

transposes the matrix using the submatrices show above as inputs.

MATLAB Exercise 1.2.6:

Implement an approximation of binary arithmetics using the default MATLAB floating point data type `double`: Only store integer values 0 and 1 in a matrix **A^{N×N}** (choose the matrix size **N** and also the values randomly as you like) and multiply it with an equally “binary” vector of size **N**. Observe the result – it should contain various integer numbers. A true binary vector addition would be implemented by the bitwise XOR operation, while a binary multiplication is equivalent to bitwise AND. A binary subtraction is the same as addition. Instead of these logical/bitwise operations use the integer modulo operation on

the result of the vector-matrix multiplication. Observe the validity of this approximation to a true binary matrix multiplication by calculating it also by hand on paper. Also test the `logical()` built-in function to see another binary approximation available in MATLAB. What is the difference between the results after converting them using `logical()` ?

Lab 1.3: Matrix partitioning

The *Monte Carlo simulations* are widely used for evaluating many kinds of systems. The name comes from the utilization of chance in the form of random numbers generation. Apart from true random generators, that usually require some hardware input, most available software random generators are pseudorandom. That means they are not random at all, but look and behave as random. You can find the reference implementations of these exercises in the accompanying source file `lab1c.m`.

MATLAB Exercise 1.3.1: Generate pseudo-random matrices and matrix-vector multiplication

- Use the built-in function `randi()` or `rand()` to generate:
 - 4x4 matrix **A** containing only values from the interval 1 to 10.
 - A 4x1 column vector **b**, containing only integer values in from the interval 1 to 3 (including).
 - A 4x1 column vector **c**, containing only integer values in from the interval 3 to 6 (including).
 - A 4x1 column vector **e**, containing only integer values in from the interval 6 to 9 (including).
- Multiply the vectors with matrix **A**:

$$\mathbf{B} = \mathbf{A} * \mathbf{b}, \mathbf{C} = \mathbf{A} * \mathbf{c}, \mathbf{D} = \mathbf{A} * \mathbf{e}$$
- Use horizontal concatenation to store the resulting vectors **B**, **C**, **D** as columns in matrix **F**.
- Join the original vectors **b**, **c**, **d** horizontally together to form a matrix **e**.
- Multiply the matrix **e** by matrix **A** and compare the result with matrix **F**. The result of the comparison must a scalar logical value (0 or 1, not a vector or matrix).

In engineering practice, several subsystems of a larger system are often modelled by matrix multiplication. It is then advantageous to combine the matrices of the subsystems to a larger matrix that describes the whole system. This often results in a special structure of the resulting system matrix – the block matrix. One can view such a matrix as a matrix of matrices, or simply as a large matrix, that can be divided into submatrices. Fig. 1.2 shows a matrix **A** divided into four smaller submatrices.

$$\mathbf{A}^{5 \times 6} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11}^{3 \times 4} & \mathbf{A}_{12}^{3 \times 2} \\ \mathbf{A}_{21}^{2 \times 4} & \mathbf{A}_{22}^{2 \times 2} \end{pmatrix} \equiv \begin{pmatrix} \mathbf{B} & \mathbf{C} \\ \mathbf{D} & \mathbf{E} \end{pmatrix}$$

Fig. 1.2: Partitioning of matrix **A** into a grid of 2x2 submatrices.

From the figure above, it should be clear that the submatrices are defined in the following way:

$$\mathbf{B} \equiv \mathbf{A}_{11} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}, \quad \mathbf{C} \equiv \mathbf{A}_{12} = \begin{pmatrix} a_{15} & a_{16} \\ a_{25} & a_{26} \\ a_{35} & a_{36} \end{pmatrix},$$

$$\mathbf{D} \equiv \mathbf{A}_{21} = \begin{pmatrix} a_{41} & a_{42} & a_{43} & a_{44} \\ a_{51} & a_{52} & a_{53} & a_{54} \end{pmatrix}, \quad \mathbf{E} \equiv \mathbf{A}_{22} = \begin{pmatrix} a_{45} & a_{46} \\ a_{55} & a_{56} \end{pmatrix}$$

The submatrices of $\mathbf{A}$ need not be of the same size, but their sizes do indeed need to be compatible: Since the submatrices $\mathbf{A}_{11}$ and $\mathbf{A}_{12}$ form a first block-row of $\mathbf{A}$, they must have the same number of rows in order to be successfully concatenated in the horizontal direction. In Fig. 1.2 this number of rows is 3. The second block-row of matrix $\mathbf{A}$ consists of submatrices $\mathbf{A}_{21}$ and $\mathbf{A}_{22}$. Again these two matrices must have the same number of rows, which is 2. Similar situation arises with the columns: submatrices $\mathbf{A}_{11}$ and $\mathbf{A}_{21}$ need to have the same number of columns in order to be concatenated in the vertical direction to form the first block-column of matrix $\mathbf{A}$. The number of columns is 4 in the example. Finally matrices $\mathbf{A}_{12}$ and $\mathbf{A}_{22}$ need to have the same number of columns (2 in the example) to form the second block-column of the matrix $\mathbf{A}$.

Since MATLAB supports range indexing, the partitioning of an existing large matrix $\mathbf{A}$ to submatrices, or combining several smaller matrices to a block matrix shouldn't be very hard. It is only necessary to be careful with indexing: the index 1,1 of a submatrix at block-position (ib, jb) corresponds to a different index in the large matrix, depending on the size of the matrices preceding the given submatrix in the grid.

Example 1.3.1:

Let matrices $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$, and $\mathbf{E}$ constitute a partitioning of the matrix $\mathbf{A}$ given in the previous text. Given the indices inside these matrices, calculate the indices of corresponding elements in the original matrix $\mathbf{A}$:

$$\mathbf{A}_{11} \equiv \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{pmatrix}, \quad \mathbf{A}_{12} \equiv \mathbf{C} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{pmatrix},$$

$$\mathbf{A}_{21} \equiv \mathbf{D} = \begin{pmatrix} d_{11} & d_{12} & d_{13} & d_{14} \\ d_{21} & d_{22} & d_{23} & d_{24} \end{pmatrix}, \quad \mathbf{E} \equiv \mathbf{A}_{22} = \begin{pmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{pmatrix}$$

Where $\mathbf{A}$ is defined as:

$$\mathbf{A}^{5 \times 6} = \begin{pmatrix} \mathbf{A}_{11}^{3 \times 4} & \mathbf{A}_{12}^{3 \times 2} \\ \mathbf{A}_{21}^{2 \times 4} & \mathbf{A}_{22}^{2 \times 2} \end{pmatrix} \equiv \begin{pmatrix} \mathbf{B} & \mathbf{C} \\ \mathbf{D} & \mathbf{E} \end{pmatrix}$$

Since the $\mathbf{B}$ submatrix starts at block-position $(ib, jb) = (1, 1)$, its elements have the same indices as elements of $\mathbf{A}$, e.g.: $b_{ij} = a_{ij}$. Since the first column of the matrix $\mathbf{C}$ corresponds to the fifth column of $\mathbf{A}$, the indices of the elements relate in the following way: $a_{ij} = c_{i(j - cb)}$ where cb is the number of columns of the matrix $\mathbf{B}$ (now $cb = 4$). Therefore: $a_{15} = c_{1(5-4)} = c_{11}$, $a_{16} = c_{1(6-4)} = c_{12}$. The same holds true for all rows of $\mathbf{C}$: $a_{25} = c_{2(5-4)} = c_{21}$, $a_{26} = c_{2(6-4)} = c_{22}$.

Similarly the submatrix **D** is located below **B** in the grid, so the row indices are offset: Let $rb = 3$ be the number of rows of the **B** matrix, we can then relate the indices of the elements: $a_{41} = d_{(4-3)1} = d_{11}$, $a_{51} = d_{(5-3)1} = d_{21}$. We can use the same formula for all other column: $a_{44} = d_{(4-3)4} = d_{14}$, $a_{54} = d_{(5-3)4} = d_{24}$.

The most important case is the **E** matrix, at grid (or block) position $(ib, jb) = (2, 2)$. When recalculating indices, we must first note the position in **A** to which the element e_{11} corresponds. From the equations above it is not hard to see directly that: $e_{11} = a_{(1+rb)(1+cb)}$ or stated differently: $a_{ij} = a_{(i-rb)(j-cb)}$. Given $rb = 3$ and $cb = 4$, we can recalculate the indices: $a_{45} = e_{(4-3)(5-4)} = e_{11}$, $a_{46} = e_{(4-3)(6-4)} = e_{12}$, $a_{55} = e_{(5-3)(5-4)} = e_{21}$, $a_{56} = e_{(5-3)(6-4)} = e_{22}$. We can also write these results in a matrix form that looks trivial:

$$\begin{pmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{pmatrix} = \begin{pmatrix} a_{45} & a_{46} \\ a_{55} & a_{56} \end{pmatrix}$$

With block-matrix explained, we can now continue with reformulating the matrix multiplication operation using submatrices. The following figure shows how the matrix multiplication **AB** can be calculated by partitioned matrices:

$$\mathbf{C}^{5 \times 3} = \mathbf{A}^{5 \times 6} \mathbf{B}^{6 \times 3} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & & & & & \\ a_{31} & & a_{34} & & & \\ a_{41} & & a_{44} & a_{45} & a_{46} & \\ a_{51} & & a_{54} & & a_{56} & \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{13} \\ \vdots & \\ b_{41} & b_{43} \\ b_{51} & b_{53} \\ b_{61} & b_{63} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11}^{3 \times 4} & \mathbf{A}_{12}^{3 \times 2} \\ \mathbf{A}_{21}^{2 \times 4} & \mathbf{A}_{22}^{2 \times 2} \end{pmatrix} \times \begin{pmatrix} \mathbf{B}_{11}^{4 \times 3} \\ \mathbf{B}_{21}^{2 \times 3} \end{pmatrix}$$

$$\mathbf{C}^{5 \times 3} = \begin{pmatrix} \mathbf{C}_{11}^{3 \times 3} \\ \mathbf{C}_{21}^{2 \times 3} \end{pmatrix}, \quad \mathbf{C}_{11} = \mathbf{A}_{11} \mathbf{B}_{11} + \mathbf{A}_{12} \mathbf{B}_{21}, \quad \mathbf{C}_{21} = \mathbf{A}_{21} \mathbf{B}_{11} + \mathbf{A}_{22} \mathbf{B}_{21}$$

The resulting matrix **C** is the same when calculated in any of the two demonstrated ways. Sometimes, however, it is more convenient, or more descriptive with regards to a given context, to use the second method.

MATLAB Exercise 1.3.2: Block matrix multiplication.

- Generate six constant matrices $\mathbf{A}_{11} \dots \mathbf{A}_{22}$, and $\mathbf{B}_{11}$, $\mathbf{B}_{21}$ as shown below (use the built-in MATLAB function `ones()`):

$$\mathbf{A}_{11} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}, \quad \mathbf{A}_{12} = \begin{pmatrix} 2 & 2 \\ 2 & 2 \\ 2 & 2 \end{pmatrix}, \quad \mathbf{A}_{21} = \begin{pmatrix} 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 \end{pmatrix}, \quad \mathbf{A}_{22} = \begin{pmatrix} 4 & 4 \\ 4 & 4 \end{pmatrix}$$

$$\mathbf{B}_{11} = \begin{pmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \\ 5 & 5 & 5 \\ 5 & 5 & 5 \end{pmatrix}, \quad \mathbf{B}_{21} = \begin{pmatrix} 6 & 6 & 6 \\ 6 & 6 & 6 \end{pmatrix}$$

- Join the matrices using the horizontal and vertical concatenation operators to form matrices **A** and **B** as defined below. Do not use the `horzcat()` and `vercat()` functions.

$$\mathbf{A}^{5 \times 6} = \begin{pmatrix} \mathbf{A}_{11}^{3 \times 4} & \mathbf{A}_{12}^{3 \times 2} \\ \mathbf{A}_{21}^{2 \times 4} & \mathbf{A}_{22}^{2 \times 2} \end{pmatrix}, \mathbf{B}^{6 \times 3} = \begin{pmatrix} \mathbf{B}_{11}^{4 \times 3} \\ \mathbf{B}_{21}^{2 \times 3} \end{pmatrix}$$

- c. Multiply the matrices **A** and **B** in two ways:
 - i. Directly using matrix multiplication operation on **A** and **B**. Store result to variable **C**.
 - ii. Per-partes using the submatrices **A**₁₁ ... **A**₂₂, and **B**₁₁, **B**₂₁ multiplication, addition and matrix concatenation. Store result to variable **C2**.
- d. Compare the resulting matrices **C** and **C2**. The result of the comparison must a scalar logical value (not a vector or matrix).

MATLAB Exercise 1.3.3: Decimal to binary conversion

Write a public function given the prototype: `function bin = deToBi( dec, width )` that converts a nonnegative decimal scalar value to a binary row vector of length given by the second parameter. Note: a public function must be placed in a separate .m file.

The MSB must be the first bit in word - bit with column index 1. If the parameter `width` is larger than the number of bits necessary for the binary representation, zeros must be filled from the left. If it is smaller, an error must be thrown. You may use the built-in function `dec2bin()` in your implementation. Note: In MATLAB a convenient toolbox function `de2bi()` exists. This is not true in Octave. This exercise serves as a workaround to implements portable MATLAB/Octave code. It will be used in later exercises.

MATLAB Exercise 1.3.4: Generate all binary words

Write a function given the prototype: `function data = generateAllBinaryWords( k )` that generates a matrix of all binary vectors of width `k` bits in natural order - starting with all zero vector, ending with all ones vector. Each row of the resulting matrix must be one binary vector; MSB must be the first bit in word - bit with column index 1. Use your implementation of the function `deToBi()` in your implementation.

Chapter 2: Source coding

The main task of the source encoder is to express the information from the source as efficiently as possible using the binary symbols 0 and 1. In the next we will assume discrete binary information source. This source will be modeled using a random variable. Further we will define symbols of a source alphabet m_i , each occurring with probability p_i . Then we can express the amount of information as:

$$I(m_i) = -\log_2(p_i) \text{ [bit]} \quad (2.1)$$

and the entropy as:

$$H(M) = E(I(M)) = \sum_{i=1}^N p_i I(m_i) = -\sum_{i=1}^N p_i \log_2(p_i) \text{ [bit / symbol]} \quad (2.2)$$

Exercise 2.1:

Let's have a source with the following distribution of symbol probability. The information is transferred in BCD code with a symbol rate: $R_s = 9600 \text{ [s / s]}$:

m_i	p_i	BCD
A	1/2	000
B	1/4	001
C	1/8	010
D	1/16	011
E	1/16	100

Calculate:

- A) The amount of information carried by each of the symbols
- B) Source entropy - the average amount of information per symbol
- C) Information rate: R_i
- D) Data rate: R_b

Solution:

A) $I(m_i) = -\log_2(p_i) \text{ [bit]}$

m_i	p_i	$I(m_i) \text{ [bit]}$
A	1/2	1
B	1/4	2
C	1/8	3
D	1/16	4
E	1/16	4

B)

$$\begin{aligned} H(M) &= -\sum p_i \log_2(p_i) = \\ &= -\left[\left(\frac{1}{2}\right) \log_2\left(\frac{1}{2}\right) + \left(\frac{1}{4}\right) \log_2\left(\frac{1}{4}\right) + \left(\frac{1}{8}\right) \log_2\left(\frac{1}{8}\right) + 2 \times \left(\frac{1}{16}\right) \log_2\left(\frac{1}{16}\right) \right] = \\ &= -\left[\left(\frac{1}{2} \times (-1)\right) + \left(\frac{1}{4} \times (-2)\right) + \left(\frac{1}{8} \times (-3)\right) + \left(\frac{1}{8} \times (-4)\right) \right] = \\ &= \left[\left(\frac{1}{2}\right) + \left(\frac{1}{2}\right) + \left(\frac{3}{8}\right) + \left(\frac{1}{2}\right) \right] = \left(\frac{15}{8}\right) = 1.875 \text{ [bit / symbol]} \end{aligned}$$

$$\text{C) } R_i = H \cdot R_s = 1.875 \text{ [bit / symbol]} \times 9600 \text{ [symbol / second]} = 18000 \text{ [bit / s]}$$

$$\text{D) } R_b = k \cdot R_s = 3 \text{ [bit / symbol]} \times 9600 \text{ [symbol / second]} = 28800 \text{ [binit / s]}$$

Since: $R_i \ll R_b \Rightarrow$ there is still a lot of room for compaction

■

In the next we will assume lossless compression = compaction.

Universal coding

It is a lossless data compression type based on dictionary. We will describe the original Lempel-Ziv algorithms denoted as LZ77 and LZ78. Main characteristics of the algorithms:

- encode multiple symbols at once
- exploit dependencies between symbols in the data stream
- achieve a better compression ratio than Huffman code
- do not need to know the statistics of the source
- it is a lossless encoding – a compaction

Other variants:

LZW Welch .pdf .tiff

LZSS Storer Szymanski .rar .arj

LZMA Markov .7z

LZH Haruyasu .zip

LZ77– sliding window

Parameters:

N – window size

L_s – the size of the actual window

The encoding output is a triple: $\langle p, l, c \rangle$, where:

p is the pointer – the address within the buffer

l is the length of the match

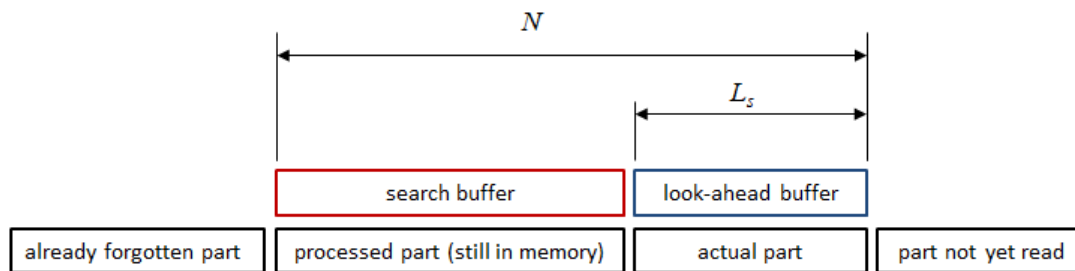
c is the symbol in the look-ahead buffer that follows the match

The sliding window scrolls from left to right. Its size is fixed. It is assumed that the lengths of the search patterns are within the window. The window is divided into 2 parts:

- left part - search buffer, in it there are auxiliary data through which we code

- the right part - lookahead buffer, it contains the actual data we want to compress

The sizes of both windows are predetermined. N is usually equal to L_s , but this is not a requirement.



Coding algorithm:

1. stack initialization - the search buffer is filled with zeros
2. the lookahead buffer is loaded with the maximum possible number of symbols of the input stream
3. codeword creation - the longest possible string of input sequence symbols is searched for in the lookahead buffer, which is identical to the string in the search buffer. The found string is encoded:
 - b. address - which is the value of the pointer where the string starts in the buffer
 - c. the length of the matching sequence in both buffers
 - d. the symbol in the look-ahead buffer that follows the match

Note: if the loaded string is not yet in the stack then the address is a null vector

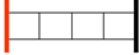
4. the encoded string from the lookahead buffer is moved to the search buffer
5. a new string of the same length as the one that was moved to the search buffer is loaded into the lookahead buffer
6. the procedure is repeated from step 3

Decoding algorithm:

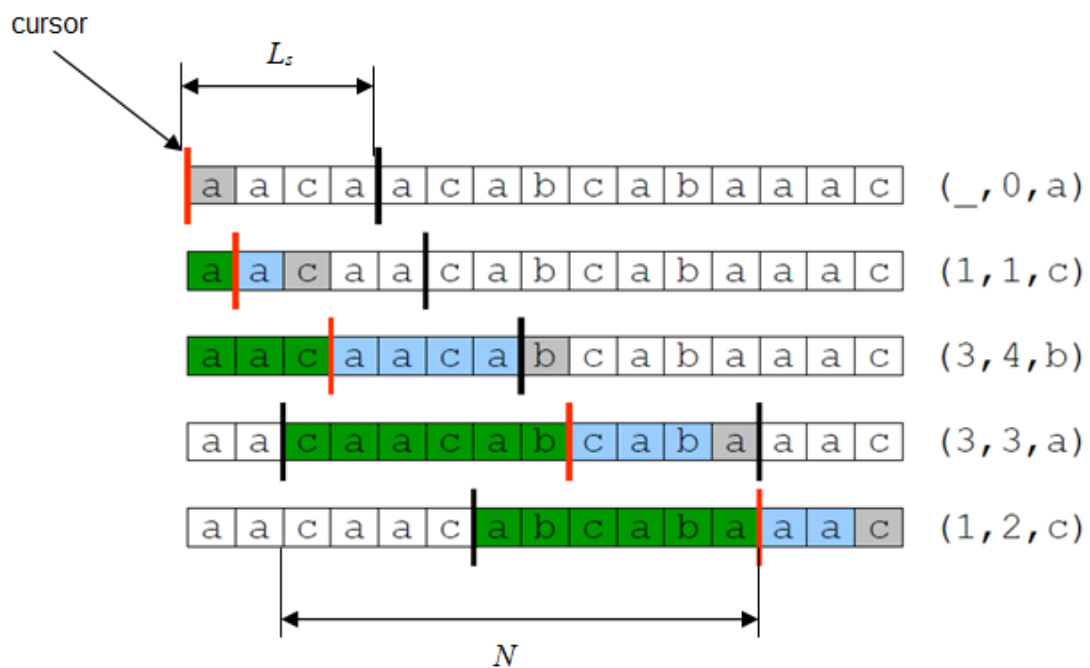
1. stack initialization - the search buffer is filled with zeros
2. read: encoded address (pointer), length and symbol in the look-ahead buffer that follows the match. A string of given length is copied from the given address from the search buffer to the look-ahead buffer and the symbol in the look-ahead buffer that follows the match is appended. The content of the look-ahead buffer is then the decoded string.
3. the decoded string is moved from the active window to the search buffer
4. the procedure is repeated from point 2

Exercise 2.4:

- A) Encode the following string of characters: **a a c a a c a b c a b a a a** using LZ77 method.
Parameters are: $N = 10$; $L_s = 4$.

Look-ahead buffer:   search buffer - length of the match: $N - L_s = 10 - 6 = 4$
 length of the match in both buffers
 symbol in the look-ahead buffer that follows the match

Solution:



- B) Decode the encoded sequence: **_ 0 a 1 1 c 3 4 b 3 3 a 1 2 c**

Solution:

We divide the sequence into triads: $(_, 0, a)$ $(1, 1, c)$ $(3, 4, b)$ $(3, 3, a)$ $(1, 2, c)$

We do the reverse procedure - instead of comparing, we copy from the lookahead buffer:

$_, 0, a \rightarrow a$
 $1, 1, c \rightarrow a a c$
 $3, 4, b \rightarrow a a c a a c a b$
 $3, 3, a \rightarrow a a c a a c a b c a b a$
 $1, 2, c \rightarrow a a c a a c a b c a b a a a c$

Balance sheet:

The original string: length = 14 characters (we assume 8 bit ASCII encoding) $\Rightarrow$ need $8 \times 14 = 112$ bits

Encoded string:

To encode search buffer is necessary:

$$\lceil \log_2(N - L_s) \rceil = \lceil \log_2(10 - 4) \rceil = \lceil \log_2(6) \rceil = \lceil 2.58 \rceil = 3 \text{ bits}$$

To encode look-ahead buffer is necessary: $\lceil \log_2(L_s) \rceil = \lceil \log_2(4) \rceil = 2 \text{ bits}$

The symbol in the look-ahead buffer that follows the match is one ASCII character, i.e. 8 bits

In total: $5 \times 3 + 5 \times 2 + 5 \times 8 = 65$ bits

Compaction ratio: $\frac{112}{65} \cong 1.723$



Exercise 2.5:

The ternary data stream is formed by the following string:

S = 0 0 1 0 1 0 2 1 0 2 1 0 2 1 0 2 1 2 0 2

Encode the given string using the LZ77 method using the ternary alphabet where the length of the match will be encoded with two ternary symbols.

Solution:

For all possible starting positions of the pointer (address) we are looking for the longest possible match - this is limited only by the number of bits to encode the parameter: length of the match. In this example, the min. length is: $00_{\text{ter}} \rightarrow 0$ and max. encodable length is: $22_{\text{ter}} \rightarrow 8$.

Additional rule: in case of ambiguity, choose the pointer value as small as possible. For the maximum encodable length, we need one more position in the buffer for the symbol that follows the match.

(0,2,1)	<table border="1"> <thead> <tr> <th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th> <th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th> </tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td> <td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>2</td><td>1</td><td>0</td> </tr> </tbody> </table>	0	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	2	1	0
0	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9																				
0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	2	1	0																				
(7,3,2)	<table border="1"> <thead> <tr> <th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th> <th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th> </tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td> <td>0</td><td>1</td><td>0</td><td>2</td><td>1</td><td>0</td><td>2</td><td>1</td><td>0</td> </tr> </tbody> </table>	0	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	0	0	0	0	0	0	0	0	1	0	1	0	2	1	0	2	1	0
0	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9																				
0	0	0	0	0	0	0	0	1	0	1	0	2	1	0	2	1	0																				
(6,8,2)	<table border="1"> <thead> <tr> <th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th> <th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th> </tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>2</td> <td>1</td><td>0</td><td>2</td><td>1</td><td>0</td><td>2</td><td>1</td><td>0</td><td>2</td> </tr> </tbody> </table>	0	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	0	0	0	0	1	0	1	0	2	1	0	2	1	0	2	1	0	2
0	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9																				
0	0	0	0	1	0	1	0	2	1	0	2	1	0	2	1	0	2																				
(0,1,2)	<table border="1"> <thead> <tr> <th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th> <th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th> </tr> </thead> <tbody> <tr> <td>1</td><td>0</td><td>2</td><td>1</td><td>0</td><td>2</td><td>1</td><td>0</td><td>2</td> <td>1</td><td>2</td><td>0</td><td>2</td><td></td><td></td><td></td><td></td><td></td> </tr> </tbody> </table>	0	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	1	0	2	1	0	2	1	0	2	1	2	0	2					
0	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9																				
1	0	2	1	0	2	1	0	2	1	2	0	2																									
(2,1,2)	<table border="1"> <thead> <tr> <th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th> <th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th> </tr> </thead> <tbody> <tr> <td>2</td><td>1</td><td>0</td><td>2</td><td>1</td><td>0</td><td>2</td><td>1</td><td>2</td> <td>0</td><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </tbody> </table>	0	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	2	1	0	2	1	0	2	1	2	0	2							
0	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9																				
2	1	0	2	1	0	2	1	2	0	2																											

Encoded string: SLZ77 = 00021 21102 20222 00012 02012 00000

LZ78 – dictionary method

Exercise 2.6:

Solution:

24

dictionary			input	output	
index	index ternary	content		integer	ternary
0	00	∅	0	(0,0)	000
1	01	0	01	(1,1)	011
2	02	01	2	(0,2)	002
3	10	2	1	(0,1)	001
4	11	1	21	(3,1)	101
5	12	21	210	(5,0)	120
6	20	210	2101	(6,1)	201
7	21	2101	21012	(7,2)	212
8	22	21012	21011	(7,1)	211

Encoded output:

SLZ78 = 0 0 0 0 1 1 0 0 2 0 0 1 0 1 1 2 0 2 0 1 2 1 2 2 1 1

Length of the string:

prior encoding: 24 symbols

after encoding: 27 symbols



Exercise 2.7:

Let add to S from Exercise 2.6 another 12 symbols: $S_e = 2\ 1\ 0\ 1\ 2\ 1\ 2\ 1\ 0\ 1\ 2\ 2$ and encode them.

Solution:

$S_e = 2\ 1\ 0\ 1\ 2\ 1\ 2\ 1\ 0\ 1\ 2\ 2$

$T \Rightarrow \wedge \quad \quad \quad \wedge$

$Out = (8,1) = (22,1)$

$Out = (8,2) = (22,2)$

Length of the string S + S_e :

prior encoding: 36 symbols

after encoding: 33 symbols



Exercise 2.8:

Decode the encoded data from the Exercise 2.7.

Solution:

SLZ78 = 0 0 0 0 1 1 0 0 2 0 0 1 0 1 1 2 0 2 0 1 2 1 2 2 1 1

dictionary			input	output
index	index ternary	content		
0	00	∅	000	0
1	01	0	011	01
2	02	01	002	2
3	10	2	001	1
4	11	1	101	21
5	12	21	120	210
6	20	210	201	2101
7	21	2101	212	21012
8	22	21012	211	21011

■

Entropic coding

It is a lossless data compression type based on entropy. We will describe two common prefix codes: Shannon-Fano and Huffman. Prefix codes have the following characteristics:

- variable code word length
- one codeword represents one resource symbol
- more frequent messages are coded with fewer binary bits
- minimise the mean codeword length
- no codeword is a prefix of another codeword
- disadvantage - need to know the statistical characteristics of the source before coding

Important parameters:

$$\text{Mean code word length: } \bar{d} = E(d) = \sum_{i=1}^N p_i d_i \quad (2.3)$$

where: p_i – is the probability of sending the message i ; $\sum_{i=1}^N p_i = 1$

d_i – is the length of the codeword belonging to message i

N – is the number of symbols

$$\text{Lower bound of the code word length: } d_{\min} = \frac{H(M)}{\log_2 N} \quad (2.4)$$

where: N – is the number of elements of the alphabet we are encoding

$$\text{Compression factor: } CF = \frac{d}{\bar{d}} \quad (2.5)$$

$$\text{Code efficiency: } CE = \frac{R_i}{R_b} \quad (2.6)$$

$$\text{Kraft inequality: } \sum_{i=1}^N 2^{-d_i} \leq 1 \quad (2.7)$$

It is a necessary and sufficient condition for the existence of binary code that has codeword lengths: $d_i, i = 1, \dots, N$.

Shannon-Fano code

This prefix code is suboptimal because the tree is formed from the most frequent symbols to the least frequent. The Shannon-Fano code is not optimal in terms of minimizing the average codeword length.

Algorithm:

1. the messages are sorted by descending probability
2. divide the messages into two groups so that the sum of the probabilities in them is as close as possible and at the same time the difference of the sums of the probabilities of the messages in the groups is as small as possible
3. assign the first group the sign 0, the second group the sign 1
4. recursively repeat until we get groups of one message

Huffman code

It is the optimal (most efficient) prefix code for a flow of stochastically independent symbols. The tree is formed from the least frequent symbols to the most frequent (from the bottom). There are variants of the described algorithm (also adaptive).

Algorithm:

1. the messages are sorted in descending order of probability - they form the leaf nodes of the tree
2. sequentially connect the two nodes with the lowest probability into one:
 - its probability will be equal to the sum of the probabilities of the merged nodes
 - if there are more possibilities, follow the additional criteria:
 - minimum variance of codeword length
3. this is repeated until there is only one node left - the root of the tree

The tree:

- has n leafs (n is the size of the alphabet)
- has $n-1$ intermediate nodes
- its complexity depends on the implementation of $O(n \log n)$ or $O(n)$ (if sort is not included)

Exercise 2.9:

The source transmits symbols with the occurrence frequencies described by the following table in BCD format at a rate of $R_s = 9600 [s / s]$.

	symbol m_i	frequency f_i
1	a	57
2	e	35
3	i	29
4	d	23
5	c	17
6	k	13
7	v	9
8	u	7
9	n	5
10	␣	5

Encode the source symbols using:

- BCD code.
- Shannon-Fano code.
- Huffman code.
- Calculate the mean codeword length $E(d)$ for all encodings.
- For each code, calculate:
 - The information rate (rate of information transfer): R_i .
 - Data rate (data rate): R_b .
 - Compression ratio
 - Code efficiency

F) Decode the following received message encoded using the Shannon-Fano code:

00000010000010011000000110000000000100000101000111

Solution

BCD code:

	symbol: m_i	frequency: f_i	p_i	BCD code	d_i
1	a	57	0.285	0000	4
2	e	35	0.175	0001	4
3	i	29	0.145	0010	4
4	d	23	0.115	0011	4
5	c	17	0.085	0100	4
6	k	13	0.065	0101	4
7	v	9	0.045	0110	4
8	u	7	0.035	0111	4
9	n	5	0.025	1000	4
1	␣	5	0.025	1110	4

$$p_i = \frac{f_i}{\sum f_i} \Rightarrow p_1 = \frac{f_1}{\sum f_i} = \frac{57}{200} = 0.285 \quad p_2 = \frac{f_2}{\sum f_i} = \frac{35}{200} = 0.175 \quad \dots$$

$$\text{Check: } p = \sum_{i=1}^N p_i = 1.0$$

We have to encode with BCD code (naturally) 10 messages $\Rightarrow$

$$d_i = \lceil \log_2(N) \rceil = \lceil \log_2(10) \rceil = \lceil 3.3219 \rceil = 4$$

$$\bar{d} = E(d) = \sum_{i=1}^N p_i d_i = 4 \text{ [bit / symbol]}$$

$$H(M) = -\sum p_i \log_2(p_i) = 2.91428 \text{ [bit / symbol]}$$

$$R_i = H \cdot R_s = 2,91428 \text{ [bit / symbol]} \times 9600 \text{ [symbol / second]} = 27977.1 \text{ [bit / s]}$$

$$R_b = \bar{d} \cdot R_s = 4 \text{ [bit / symbol]} \times 9600 \text{ [symbol / second]} = 38400 \text{ [binit / s]}$$

$R_i < R_b \Rightarrow$ there is still room for compaction

$$CF = \frac{d(BCD)}{\bar{d}} = \frac{4}{4} = 1 \rightarrow \text{no compression}$$

$$CE = \frac{R_i}{R_b} = \frac{27977.1}{38400} = 0.728569$$

■

Shannon-Fano code:

$p_i = \frac{f_i}{\sum f_i} \Rightarrow f_i = p_i \sum f_i \rightarrow$ we can use frequency instead of probability because it is linearly dependent on the probability

We have to divide f_i into two groups: $\frac{\sum f_i}{2} = \frac{200}{2} = 100 \Rightarrow$ 1st group: $\sum f_i = 92$, 2nd group:

$\sum f_i = 108$ so that the sums of abundances in both groups are as close as possible. Assign 0 to group 1, assign 1 to group 2. This process continues until the groups are single-element.

Additional rules if it is possible to split the messages in different ways - if there are ambiguities:

1. in even number of messages in groups if possible.
2. place a larger sum on top.

	m_i	f_i							Shanon-Fano	d
1	c	11	0	0	0				000	3
2	e	8			1				001	3
3	s	6		1	0				010	3
4	t	6			1				011	3
5	u	5	1	0	0	0			1000	4
6	r	5				1			1001	4
7	i	4			1	0			1010	4
8	a	4				1			1011	4
9	d	3		1	0	0	0		11000	5
10	o	3					1		11001	5
11	v	3				1			1101	4
12	m	2			1	0	0		11101	5
13	n	2					1		11110	5
14	k	1			1	0			11110	5
15	l	1					1		11111	5

$$64 = 31 + 33$$

$$31 = 19 + 12$$

$$19 = 11 + 8 \quad 12 = 6 + 6$$

$$33 = 18 + 15$$

$$18 = 10 + 8 \quad 15 = 9 + 6$$

$$10 = 5 + 5 \quad 8 = 4 + 4 \quad 9 = 6 + 3 \quad 6 = 4 + 2$$

$$6 = 3 + 3 \quad 4 = 2 + 2 \quad 2 = 1 + 1$$

$$\bar{d} = E(d) = \sum_{i=1}^N p_i d_i = 3.70313 \text{ [symbol]} [\text{binit}]$$

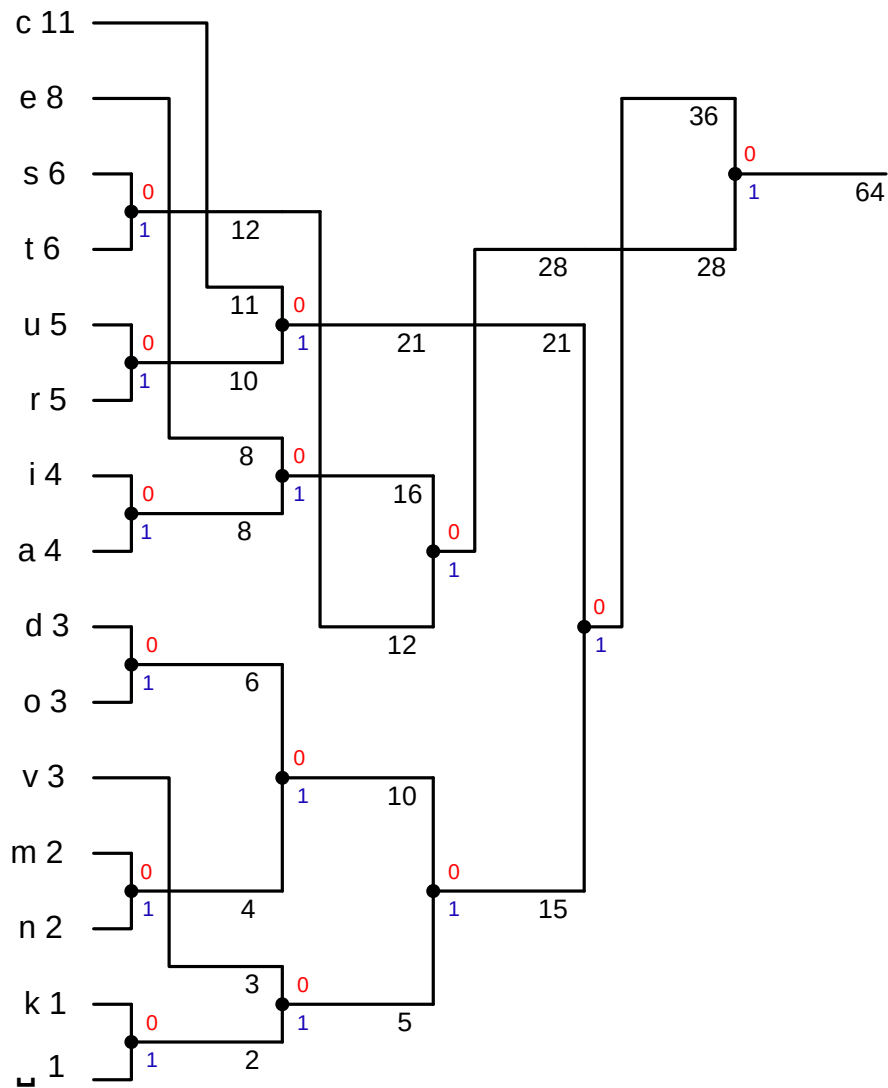
$$R_b = \bar{d} \cdot R_s = 3.70313 \text{ [bit / symbol]} \times 9600 \text{ [symbol / second]} = 35550.048 \text{ [binit / s]}$$

$$CF = \frac{d(BCD)}{\bar{d}} = \frac{4}{3.70313} = 1.080167 \rightarrow \text{slight compression}$$

$$CE = \frac{R_i}{R_b} = \frac{35016.384}{35550.48} = 0.985$$

■

Huffman code:



symbol	frequency	codeword	d_i
c	11	000	3
e	8	100	3
s	6	110	3
t	6	111	3
u	5	0010	4
r	5	0011	4
i	4	1010	4
a	4	1011	4
d	3	01000	5
o	3	01001	5
v	3	0110	4
m	2	01010	5
n	2	01011	5
k	1	01110	5
␣	1	01111	5

Since the distribution of d_i is identical to that for the Shannon-Fano code, the code parameters are the same. The Huffman code is optimal in terms of minimizing the average number of bits needed to represent a single source symbol.

Exercise 2.10:

Verify that Shannon-Fano code satisfies the Kraf inequality.

Solution:

$$\sum_{i=1}^{15} 2^{-d_i} = 4 \times 2^{-3} + 5 \times 2^{-4} + 6 \times 2^{-5} = 1.0$$

■

Chapter 3: Channel coding

The task of the channel coding is to adapt the conveyed information to the channel properties and at the same time to provide error (or erasure) protection against channel impairments such as noise or interference. The selection of the appropriate channel encoding scheme depends on the channel characteristics: burst channel, random error channel, etc. and also on the required bit error rate – BER or latency.

Linear Block Codes

The classical mathematical definition of a Linear Block Code (LBC) states that an LBC is a k -dimensional subspace of an n -dimensional vector space, usually but not necessarily defined over GF(2). This definition is sufficient for the purposes of this chapter. LBCs divide the input and output data streams into vectors of constant size: the input data word length is by convention denoted k , while the output codeword size is n . These basic parameters are then used for basic code identification, together with a third parameter – the minimal Hamming distance between any two codewords d_{min} . The triplet $[n, k, d_{min}]$ however, does not really define the code itself: the set of all the datawords, codewords, and their mapping does. Because of the linearity, we do not need to enumerate these sets, only the k basis vectors of the code are necessary. This is simply achieved by organizing them as rows of the *code generator matrix* $\mathbf{G}$ of size: $k \times n$, which then completely defines the code.

Since n is larger than k , and both are positive integer values, we may define also a positive parameter $m = n - k$ and a rational *code rate*: $R = k / n$. Regardless of the algorithm used, the decoding process then utilizes the *null space* of the code described by a *parity check matrix* $\mathbf{H}$ of size $m \times n$, that is *orthogonal* to $\mathbf{G}$ (the T in the superscript in the following equation indicates the transpose operation):

$$\mathbf{G}^{k \times n} \cdot \mathbf{H}^{T(n \times m)} = \mathbf{0}^{k \times m} \quad (3.1)$$

As mentioned above, this matrix multiplication takes place in GF(2), where the elements may only have one of the two possible binary values: 0 and 1, the addition operation is in fact binary XOR and the multiplication operation is binary AND.

The encoding process is in theory described easily by multiplying of the data word $\mathbf{i}$ by matrix $\mathbf{G}$ over GF(2), and obtaining the codeword $\mathbf{c}$. Most practical applications use a specific form of *systematic matrix* $\mathbf{G}$ in order to implement encoding in which the data word $\mathbf{i}$ is a subvector of $\mathbf{c}$. The rest of the codeword consists of redundant or parity bits and may be denoted as parity vector $\mathbf{p}$ of size m bits:

$$\mathbf{c}^{1 \times n} = \begin{bmatrix} \mathbf{i}^{1 \times k} & \mathbf{p}^{1 \times m} \end{bmatrix} = \mathbf{i}^{1 \times k} \cdot \mathbf{G}^{k \times n} \quad (3.2)$$

Because of the orthogonality of $\mathbf{G}$ and $\mathbf{H}$, all codewords are orthogonal to $\mathbf{H}$ and we may denote the result of multiplication of any vector $\mathbf{r}$ of length N as a *syndrome* vector $\mathbf{s}$:

$$\mathbf{s}^{1 \times m} = \mathbf{c}^{1 \times n} \cdot \mathbf{H}^{T(n \times m)} = \mathbf{0}^{1 \times m} \quad (3.3)$$

For all the codewords the syndrome is obviously a zero vector. For a practical systematic transmission both the $\mathbf{G}$ and $\mathbf{H}$ matrices have special structure and may easily be converted from one to the other: The $\mathbf{G}$ matrix can be partitioned vertically into two submatrices: an identity matrix $\mathbf{I}$ of size $k \times k$ and a parity matrix $\mathbf{P}$:

$$\mathbf{G} = (\mathbf{I}_k \mid \mathbf{P}^{k \times m}) \quad (3.4)$$

If the $\mathbf{G}$ is in this form, the $\mathbf{H}$ matrix can be directly calculated by:

$$\mathbf{H} = (\mathbf{P}^T \mid \mathbf{I}_m) \quad (3.5)$$

Note: Sometimes the systematic form is defined with $\mathbf{I}$ and $\mathbf{P}$ matrices switched.

The transformation is equally simple if the systematic $\mathbf{H}$ matrix is known. This doesn't work directly with non-systematic matrices, but those can be converted to systematic form. Fig. 3.1 shows how the encoded data transmission may then be modelled:

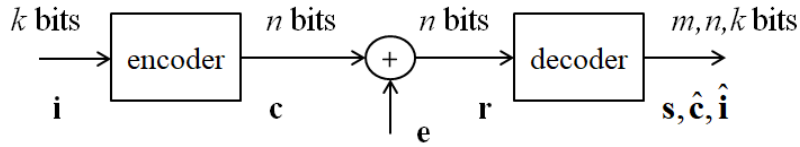


Fig. 3.1: Modelling of binary data transmission with LBC encoding and decoding.

The encoder introduces redundancy by adding m extra bits (known as parity bits) to each input block, while the channel introduces errors by adding an error vector $\mathbf{e}$ to $\mathbf{c}$. Each 1 in $\mathbf{e}$ represents a single-bit error. We model the transmission with a simple equation, utilizing the fact that any codeword $\mathbf{c}$ is orthogonal to $\mathbf{H}$ and therefore the term $\mathbf{cH}^T$ is always zero:

$$\mathbf{r} = \mathbf{c} + \mathbf{e} \quad (3.6)$$

$$\mathbf{rH}^T = (\mathbf{c} + \mathbf{e})\mathbf{H}^T = \mathbf{cH}^T + \mathbf{eH}^T = \mathbf{0} + \mathbf{eH}^T = \mathbf{s} \quad (3.7)$$

$$\mathbf{s} := \mathbf{rH}^T = \mathbf{eH}^T \quad (3.8)$$

We can use the left part of eq. (3.8) as a syndrome definition. The resulting syndrome vector is calculated in the receiver and if it equals zero, the codeword was received without any errors, or the number of errors exceeded the maximum number of correctable errors. The receiver then utilizes the redundancy and the orthogonality property to *correct* and/or *detect* errors.

Here the parameter d_{min} is most important since it defined how many errors can be corrected within each codeword, or how many errors can be detected. It is the minimal Hamming distance between any two codewords $\mathbf{c}$. The calculation of Hamming distance is based on the Hamming weight defined in the following way:

Hamming weight: $w_H(\mathbf{v})$ of vector $\mathbf{v}$ is the number of its non-zero elements. For binary vectors, this equals to the number of ones. However, the definition is general enough to work consistently also with non-binary vectors.

Hamming distance: $d_H(\mathbf{u}, \mathbf{v})$ of two vectors is the number of positions of these vectors with different values. For binary vectors, this can be calculated by the addition operation (XOR) as:

$$d_H(\mathbf{u}, \mathbf{v}) = w_H(\mathbf{u} + \mathbf{v}) \quad (3.9)$$

For a linear code, the task of finding the Hamming distances between all combinations of any two codewords is simplified by finding the minimum weight of all non-zero codewords. The linearity also implies existence of an all-zero codeword.

For a general LBC structure calculating d_{min} is an exhaustive procedure where all codewords have to be found by encoding all possible 2^k data words and applying eq. (3.2), and then finding their Hamming

weights. If we calculate a histogram where the Hamming weight of the codeword lies on the x -axis, and the number of codewords with this weight (denoted $N(\mathbf{c})$ or $\# \mathbf{c}$) on the y -axis, we call this important graph the *weight spectrum* of the code. Fig. 3.2 shows the code weight spectrum for a $[7, 4, 3]$ Hamming code (defined later).

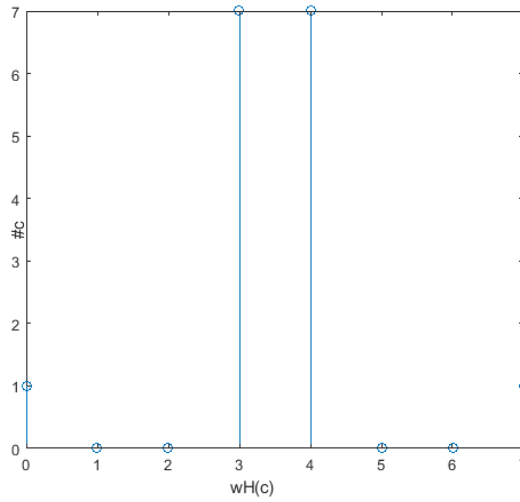


Fig. 3.2: Weight spectrum of a $[7, 4, 3]$ Hamming code.

Regardless of the form of the $\mathbf{G}$ and $\mathbf{H}$ matrices, a $[7, 4, 3]$ Hamming code weight spectrum will always contain 1 zero weight codeword, 7 codewords with $w_H(\mathbf{c}) = 3$, 7 codewords with $w_H(\mathbf{c}) = 4$, and 1 codeword of weight 7.

The decoder can work in three different modes:

1. Pure error detection
2. Pure error correction
3. A combination of both

For the first mode, the maximum number of errors that can be successfully detected in a received word $\mathbf{r}$ is given by:

$$t_{det} = d_{min} - 1 \quad (3.10)$$

In this mode, the detector can obtain information, that the received word contains errors, but cannot really correct any of them. However, this is still useful since it can signal to higher level routines in the receiver that can arrange a retransmission of the block from the sender.

The second decoder mode is pure error correction, in which the decoder tries to correct the errors in the receiver word. The relationship between the number of maximum correctable errors and d_{min} is given by:

$$t_{cor} = \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor \quad (3.11)$$

The principle of error correction lies in finding the codeword $\mathbf{c}$ closest to the received word $\mathbf{r}$ in terms of Hamming distance. We can rewrite this sentence more precisely as:

$$\hat{\mathbf{c}} := \arg_{\mathbf{c} \in \mathbf{C}} \min\{d_H(\mathbf{r}, \mathbf{c})\} \quad (3.12)$$

where $\mathbf{C}$ represents the set of all codewords in the code.

The third mode of operation is most interesting and its utility depends on the actual value of d_{min} . It will be explained later.

Hamming codes

Hamming discovered his famous class of codes when working in the Bell labs in the 1940's when trying to reduce the errors arising from the usage of punch-cards. Their design is still extremely useful today for LBC teaching and their various forms – modifications and extensions, such as SECDED can be found in actual, but usually historic digital systems.

The Hamming codes have a basic integer parameter r , from which the n and k can be calculated easily:

$$k = 2^r - r - 1 \quad (3.13)$$

$$n = k + r = 2^r - 1 \quad (3.14)$$

The minimal Hamming distance for all Hamming codes is $d_{min} = 3$, so the Hamming codes are often denoted as $[2^r - 1, 2^r - r - 1, 3]$. Explaining the insights of their construction from Karnaugh maps is beyond the scope of this book so we present the following simplified algorithm:

Construction of a Hamming code

1. Given the parameter r , we calculate n and k by means of (3.13) and (3.14).
2. The $\mathbf{H}$ matrix is created so that we enumerate all integer numbers from 0 to $2^r - 1$ in binary form and store them as columns of the $\mathbf{H}$ matrix.
3. We permute the columns of the $\mathbf{H}$ matrix so, that we take all the columns with exactly one symbol 1 and rearrange them so that they form an identity matrix on the left, and thus the $\mathbf{H}$ matrix has the systematic form as given by eq. (3.5).
4. It is now trivial to find a systematic form of $\mathbf{G}$ as defined by (3.4).

This is obviously not the only possible algorithm and many different $\mathbf{G}$ and $\mathbf{H}$ matrices exist for the Hamming code – permutations of rows and columns, or additions of linear combinations of rows and columns will not change the spectrum of the code, so the error detection/correction properties will remain.

Example 3.1:

Construct a Hamming code for $r = 3$ and find the systematic forms of matrices $\mathbf{G}$ and $\mathbf{H}$ using the above described algorithm.

Note: Examples 3.1 to 3.6 can be also implemented in MATLAB and the reference implementations are also provided in a separate file: `ch3_examples.m`. However, before you try them in MATLAB, you should implement all the MATLAB exercises defined at the end of this chapter (Open the file `lab3.m`).

Solution:

1. We calculate the basic code parameter: $[2^3 - 1, 2^3 - 3 - 1, 3] = [7, 4, 3]$ so that we immediately know the sizes of matrices $\mathbf{G}$ and $\mathbf{H}$.
2. We find the nonsystematic $\mathbf{H}_N$ matrix: First enumerate all binary vectors of length $r = 3$:

$$\mathbf{H}_N = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

3. We rearrange the columns of $\mathbf{H}_N$ to obtain one of the many possible systematic forms:

$$\mathbf{H} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

We can describe this process by a permutation: $p = [3, 5, 6, 7, 4, 2, 1]$ and if we create a corresponding permutation matrix $\mathbf{\Pi}$:

$$\mathbf{\Pi} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

We can use it to describe the process of converting $\mathbf{H}_N$ to $\mathbf{H}$ by a simple multiplication by a permutation matrix $\mathbf{\Pi}$:

$$\mathbf{H}^T = \mathbf{\Pi} \mathbf{H}_N^T \quad (3.15)$$

4. It is now a simple matter to extract the parity submatrix $\mathbf{P}$ from the systematic $\mathbf{H}$ and construct systematic form of $\mathbf{G}$:

$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

It is prudent to check we did everything well by evaluating the orthogonality of the $\mathbf{G}$ and $\mathbf{H}$ matrices by multiplying them over GF(2). The $\mathbf{G}$ should be also orthogonal to the nonsystematic form – the matrix $\mathbf{H}_N$:

$$\mathbf{GH}^T = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\mathbf{GH}_N^T = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

■

LBC decoding

There are many decoding algorithms; some are effective for a given subclass of LBC only. Here we present the most basic (and impractical) one, simple enough to give a basic understanding and be computed even by hand. A more sophisticated LBC decoding algorithm, specific to a subclass of LDPC codes is described in detail in later chapters.

The syndrome method for error correction

The syndrome decoding is the simplest decoding method and relies on a precomputed syndrome table: a table that maps all the potentially existing error words to their syndromes. The number of error word depends on two parameters: the codeword length n , and the number of correctable errors t_{cor} . The table is created as follows: first all the binary vectors of length n containing exactly one symbol 1 must be generated, then all the binary vectors containing two symbols 1, then 3, etc.... Until the number of ones reaches: t_{cor} . If we put all these vectors inside an array, we can then calculate a syndrome corresponding to each of them by applying eq. (3.10). Since the syndrome will be later used for addressing the table, it is important to implement the table in such way, as to preserve the mapping between $\mathbf{e}$ and $\mathbf{s}$. The number of entries of a syndrome table can be therefore calculated as follows:

$$|ST| = \sum_{t=0}^{t_{cor}} \binom{n}{t} \quad (3.16)$$

, where the term inside the sum is a binominal coefficient. The term for $t = 0$ may be added to account for the case of no error or may be omitted in actual implementation of the method. It is easy to see, that the sum in (3.16) grows really fast with increasing n and t_{cor} and it is indeed the large size of the syndrome table that prohibits its use in practical decoders, where n goes to hundreds or even thousands.

Example 3.2:

Calculate the size of a syndrome table for an LBC with parameters:

b) [7, 4, 3]

c) [16, 6, 7]

Solution:

The maximum number of correctable errors t_{cor} is by eq. (3.7): a) $= (3-1)/2 = 1$ b) $(7-1)/2 = 3$.

Therefore the number of rows of a syndrome table will be:

$$\text{a) } |ST| = \sum_{t=0}^1 \binom{7}{t} = \binom{7}{0} + \binom{7}{1} = 1 + 7 = 8$$

$$\text{b) } |ST| = \sum_{t=0}^3 \binom{16}{t} = \binom{16}{0} + \binom{16}{1} + \binom{16}{2} + \binom{16}{3} = 1 + 16 + 120 + 560 = 697$$

■

The syndrome method decoding algorithm

1. Out of the received, potentially erroneous word $\mathbf{r}$ we calculate the syndrome $\mathbf{s}$ according to eq. (3.8). This means the receiver must know the $\mathbf{H}$ matrix (and doesn't need to know $\mathbf{G}$).
2. The syndrome serves as an address for lookup to the syndrome table, and the result of this lookup is an error word estimate $\hat{\mathbf{e}}$. We must be careful to understand that this is only an estimate and we cannot be 100% sure this really is exactly the error vector $\mathbf{e}$.
3. The estimate $\hat{\mathbf{c}}$ of the transmitted codeword is obtained by modulo 2 addition of the received word $\mathbf{r}$ and the error word estimate $\hat{\mathbf{e}}$:

$$\hat{\mathbf{c}} = \mathbf{r} + \hat{\mathbf{e}}$$
4. For a systematic code, the desired data word estimate $\hat{\mathbf{i}}$ is a subvector of $\hat{\mathbf{c}}$. For a rarely used nonsystematic code, additional transformation must take place.

Example 3.3:

A linear block [7, 4, 3] code is defined by a non-systematic form of matrix $\mathbf{G}$:

$$\mathbf{G} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Verify, that such a code is really a 4-dimensional subspace of a 7-dimensional space, and that it indeed is one of the Hamming codes.

Solution:

To verify the number of dimensions, we must calculate how many of the matrix $\mathbf{G}$ rows are linearly independent. If there are k linearly independent rows, these form k basis vectors $\mathbf{g}_i$. The simplest way how to verify this is to use the Gaussian elimination method to reduce the $\mathbf{G}$ matrix to a reduced row echelon form, or even to a systematic form:

$$\begin{aligned} \mathbf{G} &= \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \xrightarrow{R2 := R2 + R4} \approx \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \xrightarrow{R4 := R4 + R3} \\ &\approx \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \xrightarrow{R1 := R1 + R4} \approx \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \xrightarrow{\text{switch } R2 \text{ and } R4} \\ &\approx \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} = \mathbf{G}_s \end{aligned}$$

Now we have the $\mathbf{G}$ matrix in systematic form, with obviously 4 linearly independent rows, thus $k = 4$. From the size of $\mathbf{G}$, we immediately know the parameters $[n, k] = [7, 4]$. In MATLAB, we can simply use the `rank()` or `rref()` built-ins.

However, the matrix $\mathbf{G}_s$ is here slightly different than the one shown in Example 3.1. We should now verify, that the matrix $\mathbf{G}_s$ does in fact specify a code with exactly the same error detection/correction properties as a known Hamming code. We do this by calculating the code spectrum: First, we generate all possible data words: binary vectors of length k , and then calculate the Hamming weight of each codeword. We calculate the codewords (denoted $\mathbf{c}_q$) for the systematic $\mathbf{G}_s$ of the code under scrutiny, and also for a known Hamming code generator matrix $\mathbf{G}$ from Example 3.1 (denoted $\mathbf{c}_H$):

$\mathbf{i}$	$\mathbf{c}_q = \mathbf{iG}_s \text{ (over GF(2))}$	$w_H(\mathbf{c}_q)$	$\mathbf{c}_H = \mathbf{iG} \text{ (over GF(2))}$	$w_H(\mathbf{c}_H)$
0000	0000000	0	0000000	0
0001	0001110	3	0001111	4
0010	0010011	3	0010110	3
0011	0011101	4	0011001	3
0100	0100111	4	0100101	3
0101	0101001	3	0101010	3
0110	0110100	3	0110011	4
0111	0111010	4	0111100	4
1000	1000101	3	1000011	3
1001	1001011	4	1001100	3
1010	1010110	4	1010101	4
1011	1011000	3	1011010	4
1100	1100010	3	1100110	4
1101	1101100	4	1101001	4
1110	1110001	4	1110000	3
1111	1111111	7	1111111	7

Table 3.1: Calculation of a weight spectrum of a linear block code and a known Hamming code.

While the exact mapping of datawords to codewords is different, the code spectrum will always contain 1 zero weight codeword, 7 codewords with $w_H(\mathbf{c}) = 3$, 7 codewords with $w_H(\mathbf{c}) = 4$, and 1 codeword of weight 7. We can put these values to a table Tab. 3.2 where $N(\mathbf{c})$ denotes the number of codewords with Hamming weight $w_H(\mathbf{c})$:

$w_H(\mathbf{c})$	$N(\mathbf{c})$
0	1
3	7
4	7
7	1

Table 3.2: Weight spectrum of a Hamming [7, 4, 3] code shown as a table.

The equality of the spectra is what makes the codes equivalent in terms of error performance. ■

Example 3.4:

Transmission of the encoded data over a channel with errors. Let the system use a systematic [7, 4, 3] Hamming code with **G** and **H** matrices defined in Example 3.1:

$$\mathbf{H} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The system supports transmission of a subset of the alphabet chosen based on their relative frequencies { E, T, A, O, I, N, S, R, H, D, L, U, C, M }. These letters will be encoded in binary form as numbers 1 to 14. Let zero value represent space to separate words and let the symbol ‘.’ indicate any one of the remaining alphabet letters.

A) Encode the message: “HI DAVE”

B) Let the channel introduce the following error vectors:

$$\mathbf{e}_1 = (0000001), \mathbf{e}_2 = (0000000), \mathbf{e}_3 = (0100000), \mathbf{e}_4 = (0100001)$$

$$\mathbf{e}_5 = (0000010), \mathbf{e}_6 = (0010000), \mathbf{e}_7 = (1000000)$$

Calculate the received vectors $\mathbf{r}_i$.

C) Decode the received message using the syndrome decoding method.

D) Observe any residual errors left after decoding.

Solution:

For the encoding table we can recycle portion of Table 3.1, which we extend by mapping of letters to numbers and binary data words:

Letter	Decimal	Binary $\mathbf{i}$	$\mathbf{c}_H = \mathbf{iG}$ (over GF(2))
\ \	0	0000	0000000
E	1	0001	0001111
T	2	0010	0010110
A	3	0011	0011001
O	4	0100	0100101
I	5	0101	0101010
N	6	0110	0110011
S	7	0111	0111100
R	8	1000	1000011
H	9	1001	1001100
D	10	1010	1010101
L	11	1011	1011010
U	12	1100	1100110
C	13	1101	1101001
M	14	1110	1110000
\ .'	15	1111	1111111

Table 3.3: Mapping of letters to binary datawords and codewords.

The user message of 7 letters: “HI DAVE” translates to 7 data words and is encoded to $7 \times 7 = 49$ bits transmitted. Since the $k = 4$ define 2^k possible datawords, the system is designed to replace all of the remaining least used characters {F, Y, W, G, P, B, V, K, X, Q, J, Z} with the symbol ‘.’ as a form of lossy source compression. This indicates hope, that most of the messages will be readable, if only

dictionary words are transmitted. The letter 'V'; will then be replaced by '.' So the actual message transmitted will be: 'HI DA.E'.

In serial transmission, the whole logical frame will be:

1001100,0101010,0000000,1010101,0011001,1111111,0001111

And will actually be transmitted without any helping separators:

1001100010101000000001010101001100111111110001111

Similarly the errors in the channel:

0000001,0000000,0100000,0100001,0000010,0010000,1000000

Will look more like an uninterrupted string:

0000001000000001000000100001000001000100001000000

The receiver then observes the modulo 2 addition, or bitwise XOR of the two strings:

1001101010101001000001110100001101111011111001111

Its first task will be to segment to received string to blocks of length n :

1001101,0101010,0100000,1110100,0011011,1101111,1001111

The next one is to implement the syndrome method for each block. For the syndrome method, the syndrome table Tab. 3.4 must be present (computed or loaded) in the receiver first:

e	s = eH^T
0000000	000
0000001	001
0000010	010
0000100	100
0001000	111
0010000	110
0100000	101
1000000	011

Table 3.4: Syndrome table for a systematic [7, 4, 3] Hamming code defined in Example 3.1.

The actual decoding then follows the syndrome decoding algorithm. We show details for the first received word, and put the results for the rest in the table below:

1. Based on the received vector $\mathbf{r}$, a syndrome is calculated:

$$\mathbf{s} = \mathbf{r}\mathbf{H}^T = (1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1) \cdot \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = (0 \ 0 \ 1)$$

2. Syndrome is used as an address to the syndrome table to obtain error estimate:

$$\mathbf{s} = (0 \ 0 \ 1) \rightarrow \mathbf{e} = (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1)$$

3. Estimate of the codeword is calculated:

$$\hat{\mathbf{c}} := \mathbf{r} + \mathbf{e} = (1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1) + (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1) = (1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0)$$

4. Datword is extracted as a subvector of $\hat{\mathbf{c}} : \hat{\mathbf{i}} = (1 \ 0 \ 0 \ 1)$

Results for all codewords:

$\mathbf{c}$	$\mathbf{e}$	$\mathbf{r}$	$\mathbf{s}$	$\hat{\mathbf{e}}$	$\hat{\mathbf{c}}$	$\hat{\mathbf{i}}$	$\hat{\mathbf{i}}_{\text{dec}}$	$\hat{\mathbf{i}}_{\text{ALPHA}}$
1001100	0000001	100110 1	001	0000001	100110 0	1001	9	H
0101010	0000000	0101010	000	0000000	0101010	0101	5	I
0000000	0100000	0 1 00000	101	0100000	0 0 00000	0000	0	\ ' \
1010101	0100001	1 1 1010 0	100	0000100	1110 0 00	1110	14	M
0011001	0000010	00110 1 1	010	0000010	00110 0 1	0011	3	A
1111111	0010000	11 0 1111	110	0010000	11 1 1111	1111	15	\ . \
0001111	1000000	1 001111	011	1000000	0 001111	0001	1	E

Table 3.5: Results after decoding. Some errors still occur.

The message then presented to the user at the receiver will be: 'HI MA.E'

The reader may have noticed that something is not right with the fourth codeword: the actual number of errors introduced in the transmission was two, even though the maximum number of correctable errors of the Hamming code is one. This in fact resulted in a wrong decoding. ■

As demonstrated by the previous example, the ability to correct errors may seem to be lost in some situations. In order to gain better understanding of how and when this happens and what actually happens, we must return back to the definitions of the codes maximum error detection and correction ability given by equations (3.10) and (3.11):

These limits are coming from the geometry of the code: the k -dimensional subspace of an n -dimensional vector space. In order to understand it, we must somehow visualize the important aspects of the code geometry. Fig. 3.3 attempts to do just that by showing two codewords $\mathbf{c}_1$ and $\mathbf{c}_2$, along with vectors that are part of the n -dimensional vector space, but are not codewords –don't belong to the k -dimensional subspace.

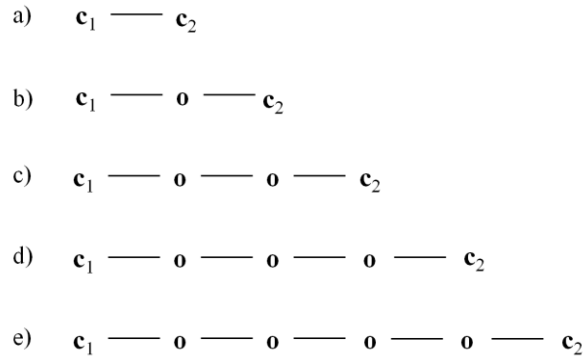


Fig. 3.3: Codewords $\mathbf{c}_1$, $\mathbf{c}_2$ separated by: 0 a), 1 b), 2 c), 3 d), 4 e) non-codewords.

Based on how many non-codewords are in-between $\mathbf{c}_1$ and $\mathbf{c}_2$, the Hamming distance between $\mathbf{c}_1$ and $\mathbf{c}_2$ increases:

- a) $d_H(\mathbf{c}_1, \mathbf{c}_2) = 1$
- b) $d_H(\mathbf{c}_1, \mathbf{c}_2) = 2$
- c) $d_H(\mathbf{c}_1, \mathbf{c}_2) = 3$
- d) $d_H(\mathbf{c}_1, \mathbf{c}_2) = 4$
- e) $d_H(\mathbf{c}_1, \mathbf{c}_2) = 5$

If a code has a minimum Hamming distance d_{min} , it means there is no combination of any two codewords, with more that $d_{min} - 1$ intermediate non-codewords. This allows for detection and correction on errors, and also for a combination of both.

As shown in Fig 3.4. a), b), c) for $d_{min} = 4$ if a 1, 2, 3-bit error affects the transmitted codeword $\mathbf{c}_1$, the resulting received word $\mathbf{r}$ is still not a codeword (each arrow represents a single-bit error). Therefore, if the receiver attempts to verify the orthogonality of $\mathbf{r}$ with $\mathbf{H}$, it can detect an error by calculating a non-zero syndrome value. However, if there are 4 bits in error – case d), the codeword is damaged so badly, that it actually is another codeword $\mathbf{c}_2$. The receiver observes only $\mathbf{r}$ and doesn't know which one of $\mathbf{c}_1$ and $\mathbf{c}_2$ was really transmitted. The syndrome calculated will be zero and the 4-erroneous bits will be completely undetected. A mirrored picture could be drawn with transmitted codeword $\mathbf{c}_2$.

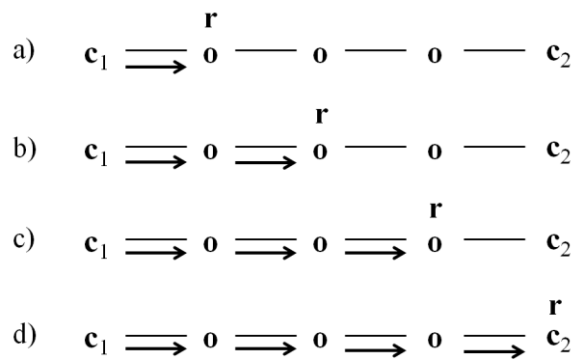


Fig. 3.4: Multiple errors introduced in channel: 1 a), 2 b), 3 c), 4 d) for $d_H(\mathbf{c}_1, \mathbf{c}_2) = 4$.

Figure 3.4 demonstrates the first mode of the decoder operation mentioned earlier – the pure error detection receiver. Let us now analyze the second mode – a pure error correction decoder. Fig. 3.5 should bring some light into a $d_{min} = 3$ situations:

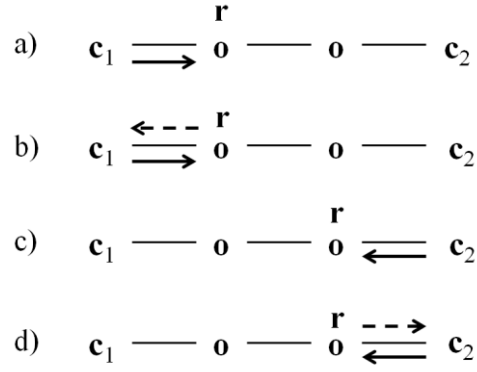


Fig. 3.5: For c_1 transmitted an error is introduced a) and corrected b).
The same principle applies if c_2 is transmitted c) and d).

The decoder receives r and is designed to operate based on eq. (3.12): $\hat{c} := \arg_{c \in C} \min\{d_H(r, c)\}$.

This means that out of the set of codewords, it selects the one closest to the receiver vector r based on the Hamming distance. As shown in Fig. 3.5, a) $d_H(r, c_1) = 1$ while $d_H(r, c_2) = 2$, so the closer codeword is selected as decoder estimate of what's transmitted. The dashed arrows in b) indicate this error correction. Similar situation is shown in c) and d), with this time the c_2 transmitted.

If we recall eq. (3.11), for a $d_{min} = 3$ the maximum number of correctable errors is $t_{cor} = 1$, which is intuitively compatible with Fig. 3.5. Let us now analyze, what happens if this limit is exceeded. Fig. 3.6 illustrates this eventuality:

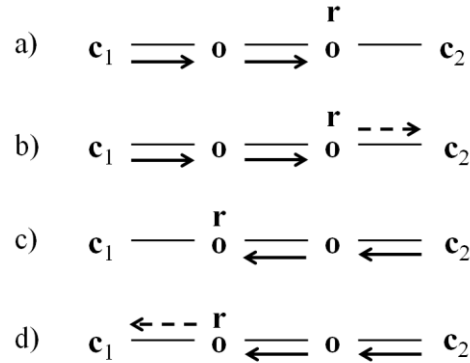


Fig. 3.6: For c_1 transmitted a double error is introduced a) and erroneously corrected b).
The same principle applies if c_2 is transmitted c) and d).

Since the number of errors introduced in the transmission exceeds the correcting capability of the code, the decoder operating on the minimum Hamming distance principle introduces further errors: If c_1 was transmitted and double error introduces a), then c_2 will be decoded b), meaning 3 bits will be in error after decoding, which is worse than if no decoding was performed at all. Again c) and d) show the mirror situation for the transmission of c_2 .

To sum up: even a good ECC might in some situation increase the number of errors. This depends entirely on the probability of error during the transmission: if the likelihood of error is too high, the use of an ECC might actually worsen the situation. It is up to the designer of the system to make sure, the code is used properly.

With these insights at hand, we may finally tackle the third form of a decoder: a decoder that combines error detection and error correction. This situation is shown in Fig. 3.7 for a $d_{min} = 4$ code.

As mentioned earlier, this operation depends on suitable code parameters, so it won't work for a $d_{min} = 3$ codes:

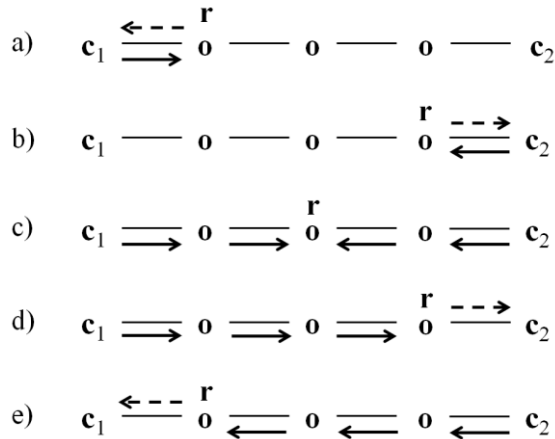


Fig. 3.7: Combined single error correction a), b), with double error detection c). Triple error causes the induction of a fourth error in the decoder d), e).

Given the minimal Hamming distance decoding criteria, the situation is quite clear in a) where the receiver word $\mathbf{r}$ is clearly closer to c_1 , and b) where it is closer to c_2 . Situation depicted in c) is more interesting: $\mathbf{r}$ is exactly in the middle between c_1 and c_2 (both Hamming distances equal 2) so it is not possible to correct this error. However, what is possible is to detect, that such a double error occurred. This detection has its value, since it allows the decoder routine to signal to higher level routines that can deal with it, perhaps by initiating a retransmission of this codeword. d) and e) then depict the eventuality of exceeding the maximum error limit, and the minimal Hamming distance decoder will erroneously “correct them” inducing a fourth bit error. It is necessary to note, that the same $d_{min} = 4$ code shown in Fig. 3.7 could be used with a detect-only decoder, which would tradeoff the single error correction for an enhanced triple error detection.

Figures 3.3 to 3.7 gave us some understanding of the code and decoding, but do not really provide a full picture. As mentioned before, the key code parameter d_{min} determines the error correction and detection capability of a code. If we recall the weight spectrum of a $[7, 4, 3]$ Hamming code:

$w_H(\mathbf{c})$	$N(\mathbf{c})$
0	1
3	7
4	7
7	1

We see that for any given codeword $\mathbf{c}$, there are 7 codewords with Hamming distance 3, and another 7 codewords with Hamming distance 4 (let us now ignore the codeword that is a complete inversion). If we know this information, couldn't we design an improved decoder, relatively to the syndrome decoder presented earlier?

Example 3.5:

Let the transmitted word of a $[7, 4, 3]$ Hamming code be $\mathbf{c} = (1, 0, 0, 1, 1, 0, 0)$. Let us choose the codewords $\mathbf{c}_3 = (0, 0, 0, 1, 1, 1, 1)$ and $\mathbf{c}_4 = (1, 1, 1, 0, 0, 0, 0)$ as representatives of the sets of codewords with Hamming distance 3 and 4 from $\mathbf{c}$: $d_H(\mathbf{c}, \mathbf{c}_3) = 3$, $d_H(\mathbf{c}, \mathbf{c}_4) = 4$. Let us analyze the situation where

$\mathbf{c}$ was transmitted, and the channel introduced 0 to 4 errors, first in the direction of codeword $\mathbf{c}_3$ (suffix a), then in the direction of codeword $\mathbf{c}_4$ (suffix b). Fig.3.8 provides visualization:

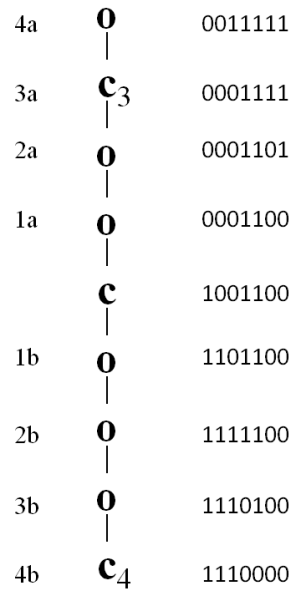


Fig. 3.8: Transmitted codeword $\mathbf{c}$ and example received words with increasing number of bit errors.

If we contemplate the information given in previous sections, and recall eq. (3.12) with its decoding definition: $\hat{\mathbf{c}} := \arg_{\mathbf{c} \in C} \min\{d_H(\mathbf{r}, \mathbf{c})\}$

We see that in the direction a) (from $\mathbf{c}$ to $\mathbf{c}_3$) this criterion doesn't introduce any new phenomenon: if a single bit error occurs (1a) it gets corrected, a double bit error (2b) results of a decoder inducing additional erroneous bit and resulting in $\mathbf{c}_3$ as decoder output. A triple error (3a) results in outputting $\mathbf{c}_3$ without any indication of error, while a quadruple error (4a) is again "corrected" to $\mathbf{c}_3$, so in fact one bit error gets corrected, while three remain.

Looking at direction b) (from $\mathbf{c}$ to $\mathbf{c}_4$), a single bit error (1a) gets corrected. For a double bit error (2b) we might be tempted to conclude, that since the Hamming distance here is 4, the word received could provide additional error detection capability, similarly to figure 3.7. However, this intuition is wrong because of the following reason: the received word $\mathbf{r}_{2b} = (1,1,1,1,1,0,0)$ might be equally far from both $\mathbf{c}$ and $\mathbf{c}_4$, but is actually closer to another codeword, not shown in the figure, to $\mathbf{c}_3' = (0,1,1,1,1,0,0)$. Clearly $d_H(\mathbf{r}_{2b}, \mathbf{c}) = 2$ and $d_H(\mathbf{r}_{2b}, \mathbf{c}_3') = 1$. So applying a minimum distance decoding criteria, the output of the decoder will be $\mathbf{c}_3'$, again inducing an extra error in the decoder. Similarly an extra error is induced for a triple error (3b) that gets corrected to $\mathbf{c}_4$.

This example serves as a cautionary tale to warn the reader from oversimplifying their analysis: the geometries of an n -dimensional vector space are usually far more complex than what is clear from simplified schemes.

Example 3.6:

Use the received codewords from the previous example and decode them using the syndrome method defined earlier and calculate the Hamming distance between decoded codeword estimates, and the actual transmitted codeword.

Solution:

Let us reuse the matrices and syndrome table from Example 3.2 to fill the results table:

# of errors	e	r	s	$\hat{e}$	$\hat{c}$	$d_H(c, \hat{c})$
0	0000000	1001100	000	0000000	1001100	0
1a	1000000	0001100	011	1000000	1 001100	0
1b	0100000	1101100	101	0100000	1 0 01100	0
2a	1000001	0001101	010	0000010	00011 11	3
2b	0110000	1111100	011	1000000	0 111100	3
3a	1000011	0001111	000	0000000	0001111	3
3b	0111000	1110100	100	0000100	1110 0 00	4
4a	1010011	0011111	110	0010000	00 0 1111	3
4b	0111100	1110000	000	0000000	1110000	4

■

As demonstrated in this chapter, ECC can detect and even correct errors, but are not omnipotent: the presence of an ECC in a system doesn't guarantee error-free operation. The actual practical goal of an ECC is to *reduce* the error probability, and its success in this task is determined by the existing probability of error within the system.

Hardware realization of cyclic codes

In case of cyclic codes any circular shift of codeword is another codeword. Cyclic code encoders are implemented using binary shift registers. In case of non-systematic codes these are non-feedback registers, for systematic codes feedback registers. For codeword generation LSFR - Linear Shift Feedback Register are often used. There are many possible HW implementations. In the next we will be interested in the canonical form of LFSR - the form with the smallest number of registers.

The generating polynomial has the form:

$$g(x) = g_m x^m + g_{m-1} x^{m-1} + \dots + g_1 x^1 + g_0 x^0 \quad g_i \in \{0,1\} \quad (3.13)$$

$$m = \deg\{g(x)\}$$

In case of non-systematic encoding the codeword polynomial has the form:

$$c(x) = i(x) \cdot g(x) \quad (3.14)$$

The general structure of a non-systematic encoder (the so-called fast implementation) is shown in Fig. 3.9.

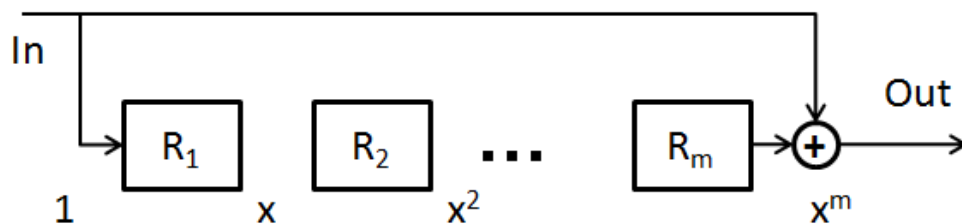


Fig. 3.9: General structure of a non-systematic encoder.

The connections between the register cells will then be given by the values of the coefficients $g(x)$ at given powers of x :

if $g_i = 0$ then we just link the output R_i to the input R_{i+1} .

if $g_i = 1$ then between R_i and R_{i+1} is placed XOR. The second XOR input is connected with the encoder input.

In case of non-systematic encoding the codeword polynomial has the form:

$$c(x) = i(x) \cdot x^{st\{g(x)\}} + i(x) \cdot x^{st\{g(x)\}} \bmod(g(x)) \quad (3.15)$$

The general structure of systematic encoder is shown in Fig. 3.10.

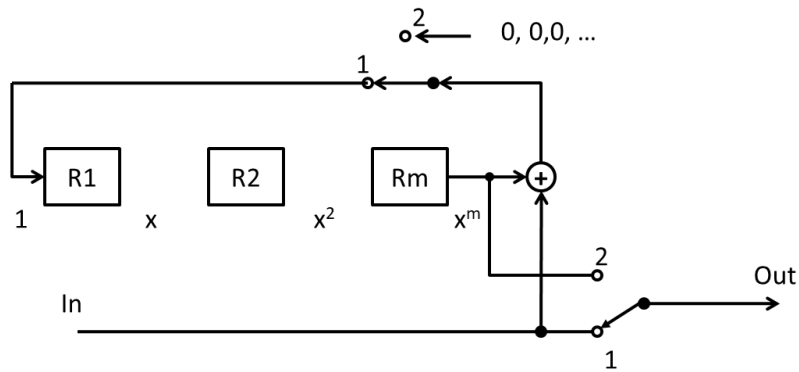


Fig. 3.10: General structure of systematic encoder.

Note: If the top switch is in position 2, zeros are fed to the circuit.

Exercise 3.1:

- Draw the hardware encoder circuit for the non-systematic form of the cyclic code $[7, 4, 3]$, given generating polynomial is: $g(x) = x^3 + x + 1$
- Encode the information vector $i(x) = x^3 + x \rightarrow \mathbf{i} = (1, 0, 1, 0)$.

Solution:

a)

In our case the encoder is shown in Fig. 3.11.

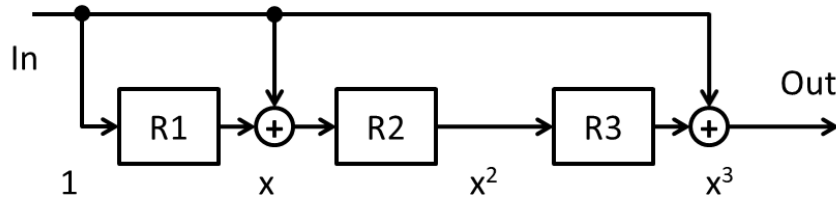


Fig. 3.11: Hardware realization of $[7,4,3]$ non-systematic encoder.

b)

Input information enters the encoder MSB first. Prior encoding the encoder is reset – in all registers are zeros. For Fig. 3.11 we can write the following equations:

$$\begin{aligned} R_1(t) &= \text{In}(t) \\ R_2(t+1) &= \text{In}(t+1) \oplus R_1(t) \quad \oplus = \text{XOR} \\ R_3(t+1) &= R_2(t) \\ \text{Out}(t+1) &= \text{In}(t+1) \oplus R_3(t) \end{aligned}$$

The solution is in the following table (boldface is the initial values):

time	In	R1	R2	R3	Out
0	-	0	0	0	-
1	1	1	1	0	1
2	0	0	1	1	0
3	1	1	1	1	0
4	0	0	1	1	1
5	0	0	0	1	1
6	0	0	0	0	1
7	0	0	0	0	0

0 are zeros necessary to reset the encoder. After $t = 7$ the result is:

$$c = (1, 0, 0, 1, 1, 1, 0), \quad c(x) = x^6 + x^3 + x^2 + x$$

Verification:

$$\begin{aligned} c(x) &= i(x) \cdot g(x) = (x^3 + x) \cdot (x^3 + x + 1) = x^6 + x^4 + x^3 + x^4 + x^2 + x \\ &= x^6 + x^3 + x^2 + x \end{aligned}$$

■

Exercise 3.2:

- Draw the hardware encoder circuit for the systematic form of the cyclic code $[7, 4, 3]$, given generating polynomial is: $g(x) = x^3 + x + 1$
- Encode the information vector $i(x) = x^3 + x \rightarrow \mathbf{i} = (1, 0, 1, 0)$.

Solution:

a)

The systematic encoder is shown in Fig. 3.12.



Coding and decoding of [7, 4, 3] Hamming code is analyzed in [8] in the TIMS experiments: “Block coding & decoding”, Emona TIMS Volume D2 – Further & Advanced Digital Experiments, experiment: D2-07 and “Block coding & coding gain”, Emona TIMS Volume D2 – Further & Advanced Digital Experiments, experiment: D2-08.

Convolutional codes

Convolutional codes are the most important of all trellis codes. Mostly they are used concatenated with block codes. They are suitable to correct independent errors. Convolutional codes are described by 3-tuple: (n, k, K) , where n is the output n -tuple produced each time the input k -tuple enters the encoder. Most often these codes are constructed as binary, then $k = 1$. In the next we will assume binary convolutional codes. $K = m$ is the *constraint length*, representing the length of the shift register of the encoder. An example diagram of the convolutional encoder is shown in Fig. 3.13.

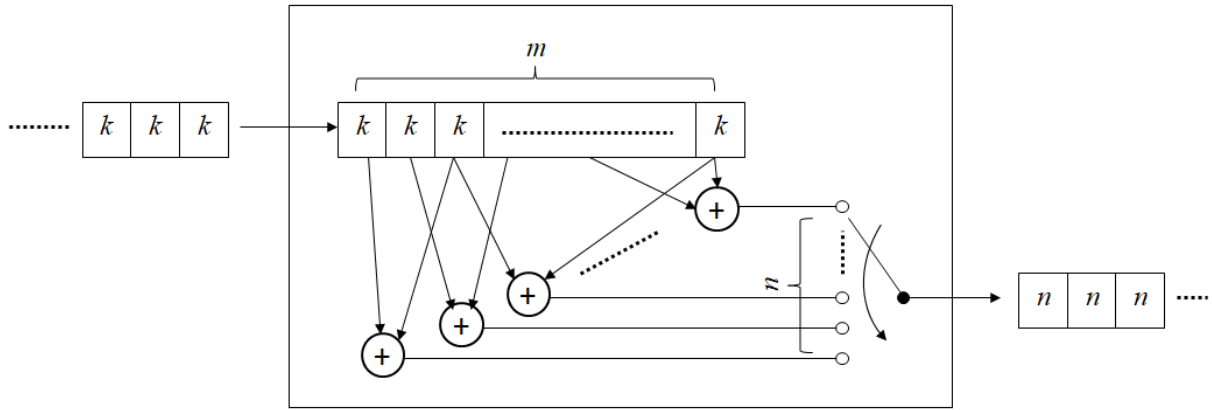


Fig. 3.13: Block diagram of a convolutional encoder.

It consists of m -stage shift register, n modulo-2 adders (XOR circuits) and n -poles commutator. Each time one bit enters the encoder, the commutator runs over all n -poles and produces the output n -tuple. The output n -tuple is generated based on the input bit and $m-1$ bits stored in the shift register. The measure of the added redundancy is expressed as the code rate: $R_k = k/n$. Unlike block code, in convolutional code it is necessary to reset the encoder after all the information bits of the message have passed through the encoder. (The information sequence is periodically divided into fixed-length blocks (frames), so the encoder must be reset after each block). This creates a "tail" which is essential for proper decoding. This tail reduces the code rate. To reset the encoder, a tail that is one zero shorter than the number of cells in the shift register, i.e., $m-1$ zeros, must be appended to the message. This will result in a corresponding longer tail also in the encoded sequence and the code rate will be reduced to “effective code rate” R_{keff} , where l is the length of the input block.

$$R_{keff} = \frac{l \times k}{(l + m - 1) \times n} \quad (3.16)$$

In practice: $l \gg 1$, then $R_{keff} \approx R_k$ and $R_{keff} = 1/n \Rightarrow$ convolutional codes can only have a code rate from the set $\{1/2, 1/3, 1/4, \dots\}$.

Convolutional encoder can be represented by several equivalent ways:

1. hardware representation (a circuit diagram)
2. connection vectors
3. connection polynomials
4. impulse response
5. state diagram
6. tree diagram
7. trellis diagram

Description of convolutional codes

Exercise 3.3:

1. Let's have a convolutional encoder defined by the circuit diagram shown in Fig. 3.14.

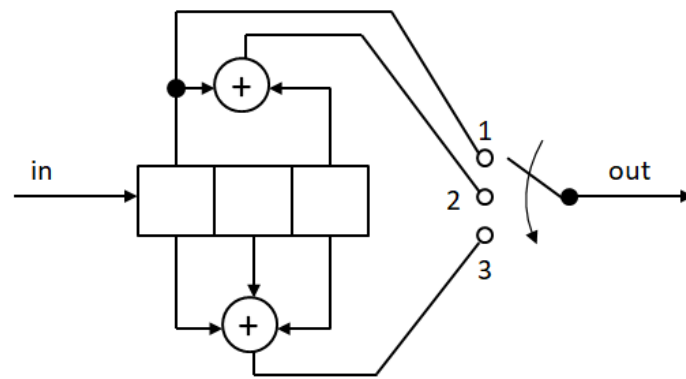


Fig. 3.14: Convolutional code encoder circuit.

It is obvious that: $K = 3$ and $R_k = \frac{1}{3}$.

2. Representation by *connection vectors*: if the corresponding register cell has is connected to the adder, the corresponding coordinate of vector is 1, otherwise is 0. Connection vectors are numbered accordingly to the commutator. In our case:

$$\bar{g}_1 = [1, 0, 0]$$

$$\bar{g}_2 = [1, 0, 1]$$

$$\bar{g}_3 = [1, 1, 1]$$

3. Representation by *connection polynomials*: if the corresponding register cell has is connected to the adder, the corresponding coefficient of the polynomial is 1, otherwise is 0. Connection polynomials are numbered accordingly to the commutator. The degree of each connection polynomial is $\leq K - 1$. The lowest power corresponds to the input cell of the shift register. In our case:

$$g_1(x) = 1$$

$$g_2(x) = 1 + x^2$$

$$g_3(x) = 1 + x + x^2$$

In literature convolutional encoders are described using octal notation. The analyzed encoder could be then expressed as: $(1,5,7)_{\text{OCT}}$. In case that the degree of $g(x)$ is 0 then we have a systematic encoder. In this case the first bit of the output n -tuple is the input bit. Encoding the input sequence using individual generating polynomials produces encoded sequence associated with each of the commutator input. The resulting encoded sequence is obtained by writing off the coefficients of the individual output sequences at the same powers, in the order the commutator goes. Let encode the input sequence: $i(x) = 1 + x^2$. In our example:

$$c_1(x) = i(x)g_1(x) = (1 + x^2)1 = 1 + x^2$$

$$c_2(x) = i(x)g_2(x) = (1 + x^2)(1 + x^2) = 1 + x^2 + x^2 + x^4 = 1 + x^4 \rightarrow \text{addition mod 2}$$

$$c_3(x) = i(x)g_3(x) = (1 + x^2)(1 + x + x^2) = 1 + x + x^2 + x^2 + x^3 + x^4 = 1 + x + x^3 + x^4$$

$$\begin{array}{l} c(x): 1x^0 + 0x + 1x^2 + 0x^3 + 0x^4 \rightarrow c_1(x) \\ \quad 1x^0 + 0x + 0x^2 + 0x^3 + 1x^4 \rightarrow c_2(x) \\ \quad 1x^0 + 1x + 0x^2 + 1x^3 + 1x^4 \rightarrow c_3(x) \end{array} \quad \downarrow$$

$$c(x) = (1,1,1)x^0 + (0,0,1)x + (1,0,0)x^2 + (0,0,1)x^3 + (0,1,1)x^4$$

$$\bar{c} = 111 \ 001 \ 100 \ 001 \ 011$$

tail for resetting the encoder

4. The *impulse response* – it is the response of the encoder to the special input – a single bit that moves through the encoder. Let assume the same input as in the previous case: $\bar{i} = 101$

register	output
1 0 0	1 1 1
0 1 0	0 0 1
0 0 1	0 1 1

In case that one 1 moves through the encoder (the impulse response) is: $1 \rightarrow 111 \ 001 \ 011$. In case that one 0 moves through the encoder the output is: $0 \rightarrow 000 \ 000 \ 000$. Therefore for our input:

$$\begin{array}{l} 1: \quad 111 \ 001 \ 011 \\ 0: \quad 000 \ 000 \ 000 \\ \hline 1: \quad 111 \ 001 \ 011 \\ \hline \bar{c} = 111 \ 001 \ 100 \ 001 \ 011 \end{array}$$

the tail for resetting the encoder

The output sequence is the convolution of the impulse response and the input sequence [convolution in digital form is shift and add]. A convolutional coder is essentially a binary linear FIR transversal filter, and its output is the convolution of the input vector and the impulse response.

5. The *state diagram* - it is based on the idea of the coder as a finite automaton. We define the state of a convolutional encoder by the contents of its last $K-1$, which are all to the right. If we know the state of the encoder (the contents of the register) and the next input bit, we can determine what the output will be. In our example, the output is given by the input bit and the last 2 bits of the previous state $\Rightarrow$ 4 possible states in the state diagram (2^2). Generally the number of states in a state diagram for binary convolutional code is: $(K-1)^2$. A state diagram is a graph showing all possible states of the encoder and all possible allowed transitions between each state. However, this description is not very suitable for describing encoding over a long period of time. The state diagram of the analyzed code is shown in Fig. 3.15.

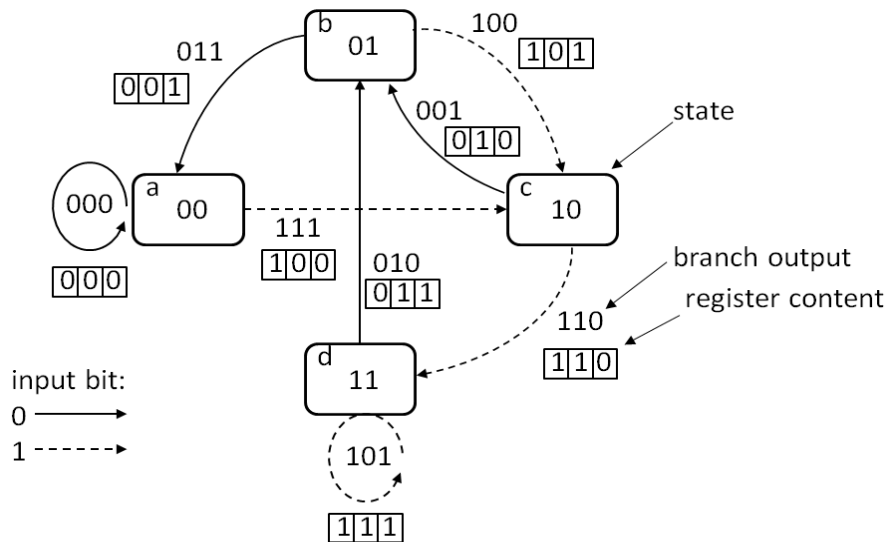


Fig. 3.15: State diagram for the convolutional code encoder circuit defined in Fig. 3.14.

Note: it is assumed that content of the register prior encoding is: 000.

6. The *tree diagram* - it is essentially an extension of a state diagram with a time dimension. The nodes of the tree correspond to individual states and branches correspond to outputs. From a certain point in time, certain regularity appears in the tree diagram: i.e., the top and bottom halves of the tree are the same. We assume the same initial state: 000. The tree diagram of the analyzed code is shown in Fig. 3.16.

Note: if the input bit is 0 we are moving up in the diagram and if it is 1 we are moving down. In the following example in red it is encoded the input sequence: 101 + 00 $\rightarrow$ for resetting the encoder. To encode the given sequence we follow the diagram correspondingly to the input bits and readout the branch values for the corresponding time slots. In our case the input is: $\bar{i} = (1, 0, 1)$. The output is: $\bar{c} = 111 \ 001 \ 100 \ 001 \ 011$


 tail for resetting the encoder

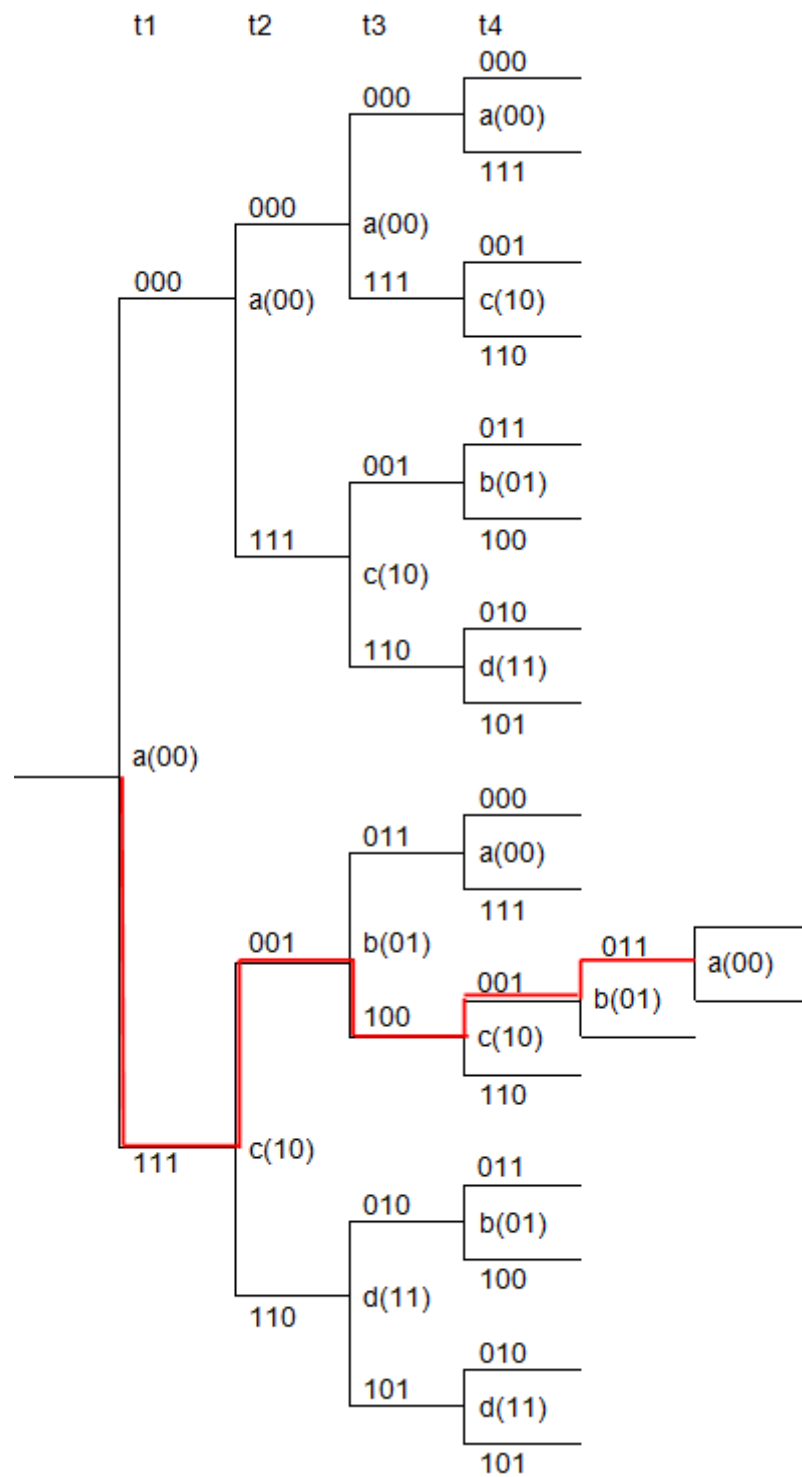


Fig. 3.16: Tree diagram for the convolutional code encoder circuit defined in Fig. 3.14.

7. trellis diagram - A trellis is actually a tree diagram in which identical states at a particular time t_i are merged and represented by a single node. In our case the trellis looks as follows. In the following example in red it is encoded the input sequence: $101 + 00 \rightarrow$ for resetting the encoder. To encode the given sequence we follow the trellis correspondingly to the input bits and readout the output for the corresponding time slots. In our case the input is: $\bar{i} = (1,0,1)$. The trellis diagram of the analyzed code is shown in Fig. 3.17. The output is: $\bar{c} = 111 \ 001 \ 100 \ 001 \ 011$


 tail for resetting the encoder

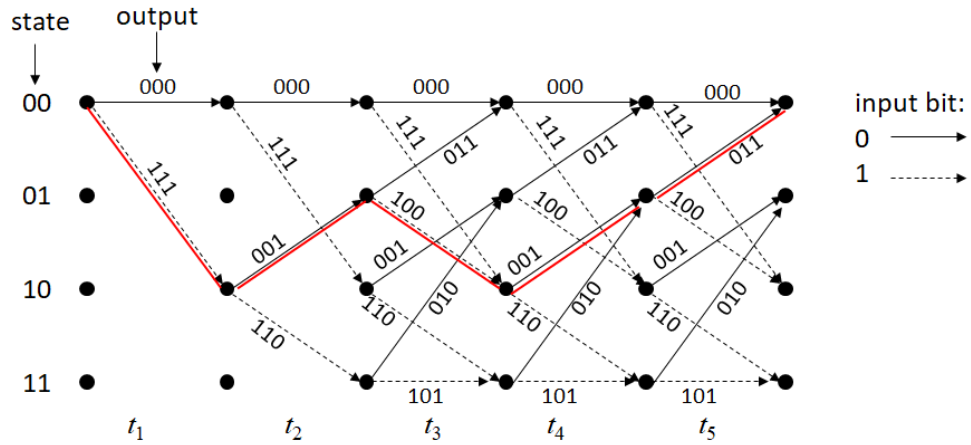


Fig. 3.17: Trellis diagram for the convolutional code encoder circuit defined in Fig. 3.14.

Error correction capabilities of convolutional codes

The ability of linear block codes to correct errors was related to the Hamming distance between codewords. Similarly for convolutional codes, with the difference that codewords are semi-infinite sequences of bits. Thus, if we consider the total distance, we would find that all the different codewords have infinite distance. This, however, is of no practical significance. Therefore, it is more useful to know the distance characteristics on finite codeword sections. Due to linearity, it is sufficient to examine the distances of all codewords from the null sequence without loss of generality (which is the same as examining all the distances from a particular codeword to the rest of all the others). Thus, we assume that a null codeword has been transmitted. Then a decoding error will occur if the decoded received sequence is closer to another codeword sequence than to the null sequence. If we label the branches in the trellis with the respective Hamming distances of each branch from the null sequence, we see that the sequences separate from the null sequence and then rejoin with it. The minimum Hamming distance in the set of arbitrarily long sequences that are separated from the zero sequence and then rejoin to it is called the minimum free distance - d_{free} .

Relation of the number of maximum correctable errors (per constraint length) is:

$$t = \left\lfloor \frac{d_{free} - 1}{2} \right\rfloor \quad (3.17)$$

The minimum free distance of a given code can be computed using the so-called transfer function. The transfer function expresses the response of the encoder to a particular input:

$$T = \frac{\text{output}}{\text{input}}$$

Convolutional encoder is a final state automaton – it does not have the input and output. In order to obtain input and output we split the initial state (usually 000...) into two identical states. From our previous example we will modify the existing state diagram to obtain new states:

- “a” will represent the input
- “e” will represent the output

In the original state diagram we will split state 00 as is shown in Fig. 3.18.

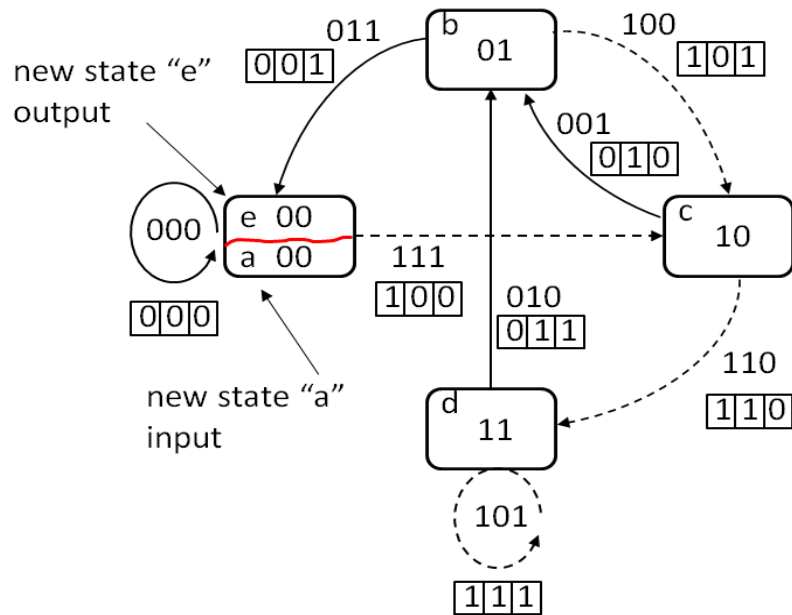


Fig. 3.18: Splitting of a state of the state diagram.

In the split state diagram each branch is evaluated using an auxiliary variable D such that the exponent of D expresses the Hamming distance of that branch to zero branch (i.e., the number of non-zero elements). We do not draw the loop around zero in the new diagram because it does not contribute to the distance to the null codeword. Modified (split) state diagram now has the input and the output and is shown in Fig. 3.19.

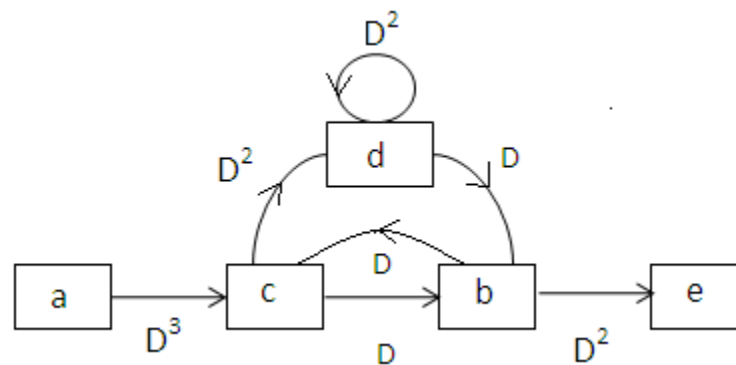


Fig. 3.19: State diagram with split state.

Next we have to write the state equations. We construct the state equations by expressing the individual state variables X_i in terms of the evaluated (using the variable D) branches entering a given state. In our case:

$$\begin{aligned}X_c &= D^3 X_a + DX_b \\X_b &= DX_c + DX_d \\X_d &= D^2 X_c + D^2 X_d \\X_e &= D^2 X_b\end{aligned}$$

The transfer function is the function of variable D : $T(D) = \frac{X_e}{X_a}$. Next we have to express X_e and X_a from the state equations:

$$\begin{aligned}X_c &= D^3 X_a + DX_b \\X_b &= DX_c + DX_d \Rightarrow X_d = DX_b \\X_d &= D^2 X_c + D^2 X_d \\X_e &= D^2 X_b \Rightarrow X_b = \frac{X_e}{D^2}\end{aligned}$$

Then:

$$\begin{aligned}X_e &= D^2 X_b = D^2 (DX_c + DX_d) = D^3 (X_c + X_d) = D^3 (X_c + DX_b) = D^3 \left(X_c + D \frac{X_e}{D^2} \right) = D^3 \left(X_c + \frac{X_e}{D} \right) \\&= D^3 \left(D^3 X_a + DX_b + \frac{X_e}{D} \right) = D^3 \left(D^3 X_a + \frac{X_e}{D} + \frac{X_e}{D} \right) = D^2 (D^4 X_a + 2X_e) = D^6 X_a + 2D^2 X_e\end{aligned}$$

$$\text{Finally: } T(D) = \frac{X_e}{X_a} = \frac{D^6}{1 - 2D^2}$$

Next we have to perform polynomial division:

$$\begin{array}{r}D^6 : 1 - 2D^2 = 1D^6 + 2D^8 + 4D^{10} + 8D^{12} \dots \Rightarrow \text{There is 1 sequence with Hamming distance 6} \\ \underline{-(1D^6 - 2D^8)} \qquad \qquad \text{which deviates from the zero sequence and then rejoins} \\ \qquad 2D^8 \qquad \qquad \text{it, there are 2 sequences with Hamming distance 8} \\ \underline{-(2D^8 - 4D^{10})} \qquad \text{which deviate from the zero sequence and then rejoin} \\ \qquad \qquad 4D^{10} \qquad \text{it, there are 4 sequences with Hamming distance 10} \\ \underline{-(4D^{10} - 8D^{12})} \qquad \text{which deviate from the zero sequence and then rejoin it} \\ \qquad \qquad \qquad 8D^{12} \qquad \dots \\ \qquad \qquad \qquad \vdots\end{array}$$

Path with the minimal Hamming distance which deviates from the zero sequence and it rejoins it (in our case the sequence of states: $acbe = a$) has: $d_{free} = 6$.

$$t = \left\lfloor \frac{d_{free} - 1}{2} \right\rfloor = \left\lfloor \frac{6 - 1}{2} \right\rfloor = 2 \rightarrow \text{it means this code is capable of correcting 2 errors on one constraint}$$

length $\rightarrow$ it corrects 2 bits on the sequence of $K \frac{1}{R_k} = 3 \times 3 = 9$ bits. For practical implementation,

typically 3÷5 constrain lengths are stored in memory. Expressing d_{free} using $T(D)$ for large K is elaborate because the complexity of $T(D)$ increases exponentially with K .

The transfer function $T(D)$ can also provide other information besides Hamming distances of different paths. Let introduce an additional variable L into each branch of the state diagram such that the exponent of L in the transfer function serves as a count of the number of branches in any path from state "a" to state "e". An additional variable N in each branch will indicate a transition caused by input bit 1, so the exponent of N in the transfer function is cumulatively incremented only if that transition was caused by input bit 1. The modified state diagram for our example is shown in Fig. 3.20.

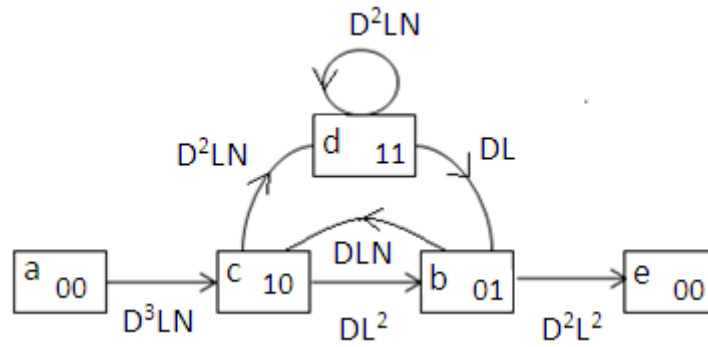


Fig. 3.20: Modified State diagram.

State equations:

$$X_c = D^3LN X_a + DLN X_b$$

$$X_b = DLX_c + DLX_d$$

$$X_d = D^2LN X_c + D^2LN X_d \Rightarrow X_d = DN X_b$$

$$X_e = D^2LX_b \Rightarrow X_b = \frac{X_e}{D^2L}$$

$$X_e = D^2LX_b = D^2L(DLX_c + DLX_d) = D^3L^2(X_c + X_d) = D^3L^2(X_c + DN X_b)$$

$$= D^3L^2 \left(X_c + DN \frac{X_e}{D^2L} \right) = D^3L^2 \left(D^3LN X_a + DLN X_b + N \frac{X_e}{DL} \right)$$

$$= D^3L^2 \left(D^3LN X_a + DLN \frac{X_e}{D^2L} + N \frac{X_e}{DL} \right) = D^3L^2 \left(D^3LN X_a + \frac{LN X_e + NX_e}{DL} \right)$$

$$= D^3L^2 \frac{(D^4L^2NX_a + X_e(LN + N))}{DL} = D^2L [D^4L^2NX_a + X_e(LN + N)]$$

$$= D^6L^3NX_a + D^2L(LN + N)X_e$$

$$X_e [1 - D^2L(LN + N)] = D^6L^3NX_a \Rightarrow \frac{X_e}{X_a} = \frac{D^6L^3N}{1 - D^2L(1 + L)N}$$

$$\begin{aligned}
T(D, L, N) &= D^6 L^3 N : 1 - D^2 L(1+L)N = D^6 L^3 N + D^8 L^4 (1+L)N^2 + D^{10} L^5 (1+L)^2 N^3 + \dots \\
&\quad - \frac{(D^6 L^3 N - D^8 L^4 (1+L)N^2)}{D^8 L^4 (1+L)N^2} \quad \quad \quad * \quad \quad \quad ** \quad \quad \quad *** \\
&\quad - \frac{(D^8 L^4 (1+L)N^2 - D^{10} L^5 (1+L)^2 N^3)}{D^{10} L^5 (1+L)^2 N^3} \dots
\end{aligned}$$

*

in trellis exists 1 path of length 3 (L^3), with Hamming distance 6 (D^6), that differs in 1 (N^1) bit from the zero sequence

**

in trellis there are 2 paths with Hamming distance 8 from the zero sequence, 1st path has length 4 (L^4), 2nd path has length 5 (L^5), these paths differ in 2 bits from the zero seq. (N^2)

in trellis there are 4 paths with Hamming distance 10 from the zero sequence, 1st path has length 5 (L^5), 2 paths have length 6 (L^6), 1 path has length 7 (L^7). These paths differ in 3 bits from the zero sequence (N^3) ...

Decoding of convolutional codes

The output of the detector $z(T)$ can be configured in two ways:

1. $z(T) = \begin{cases} 0 \\ 1 \end{cases}$ - in this case we are talking about hard decision decoding
2. $z(T)$ is quantized to more than 2 levels – usually ≥ 8 - in this case we are talking about soft decision decoding. In this case the output $z(T)$ represents the probability of $z(T)$ to be 0 or 1.

The soft decision decoding has greater coding gain than the hard decision decoding but it is more complex. An algorithm which minimizes the error of the output decoded sequence is Viterbi algorithm. In the next will be described hard decision decoding using Viterbi algorithm and the encoder described above.

For an easy following the decoding process it is useful to display one part of a trellis with complete transitions is shown in Fig. 3.21.

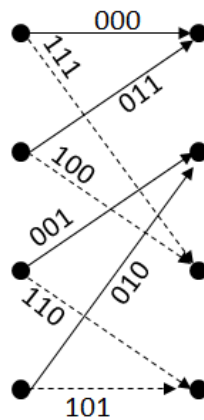
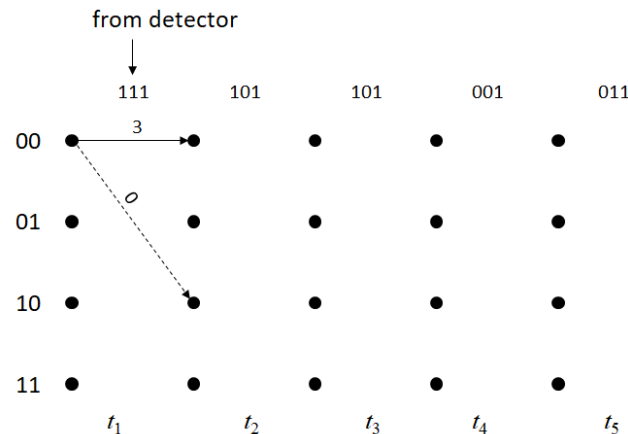


Fig. 3.21: The complete trellis segment for the code defined in Fig. 3.14.

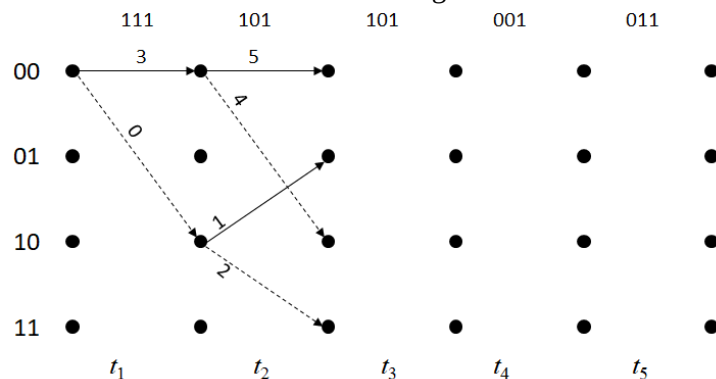
The input bit stream from the detector has to be partitioned into n -tuples of bit, where n is the size of the output n -tuple. The assumption is to have a reset decoder. The next step is to draw an empty trellis with so many time intervals as we have the input n -tuples. Let have the following input sequence from detector: 111 101 101 001 011. Since the decoder is reset we have to start from the state 00. In time t_1 we received 111. From state 00 we can go to the state: 00 or to 10. If we go to 00 the Hamming distance between the received 3-tuple 111 and the 00 to 00 transition is 3. In time t_1 the other possibility is to go from state 00 to 10. This transition has Hamming distance 0 between the received 3-tuple 111.

Exercise 3.4: Progress of hard Viterbi decoding.

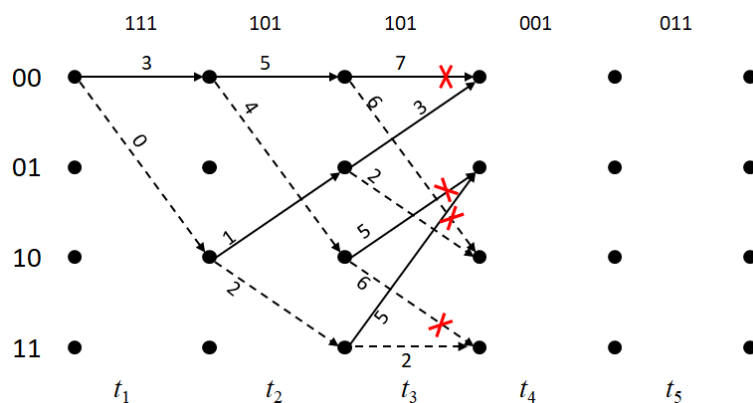
a) at t_1 :



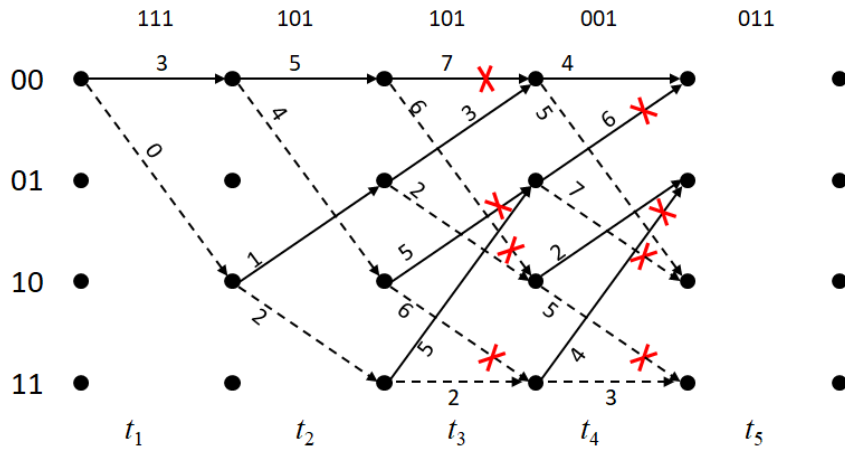
b) at t_2 – Hamming distances of transitions are cumulating:



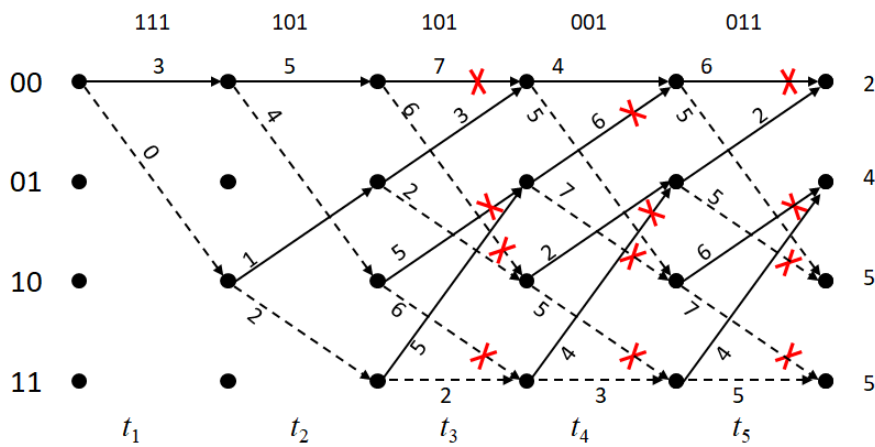
c) at t_3 – to each of the last states (timewise) enter two transitions. In order to keep only surviving paths, transitions with higher Hamming distance are cancelled out. In case of equal distances it does not matter which one is cancelled:



d) at t_4 :



e) at t_5 :



On the right side we see the cumulative Hamming distances (weights) of the surviving paths at time t_5 . The resulting number show the number of errors accumulated during that particular sequence. At this time the decoder stops and selects the surviving path with the lowest weight – in our case 2. Decoder follows this survived path backwards to time t_1 . As it proceeds along this path it outputs bit sequence corresponding to the elements of the survived path according to the state (or trellis) diagram. From the figure bellow can be seen that the decoder corrected two errors in one constraint length, what corresponds with theory.

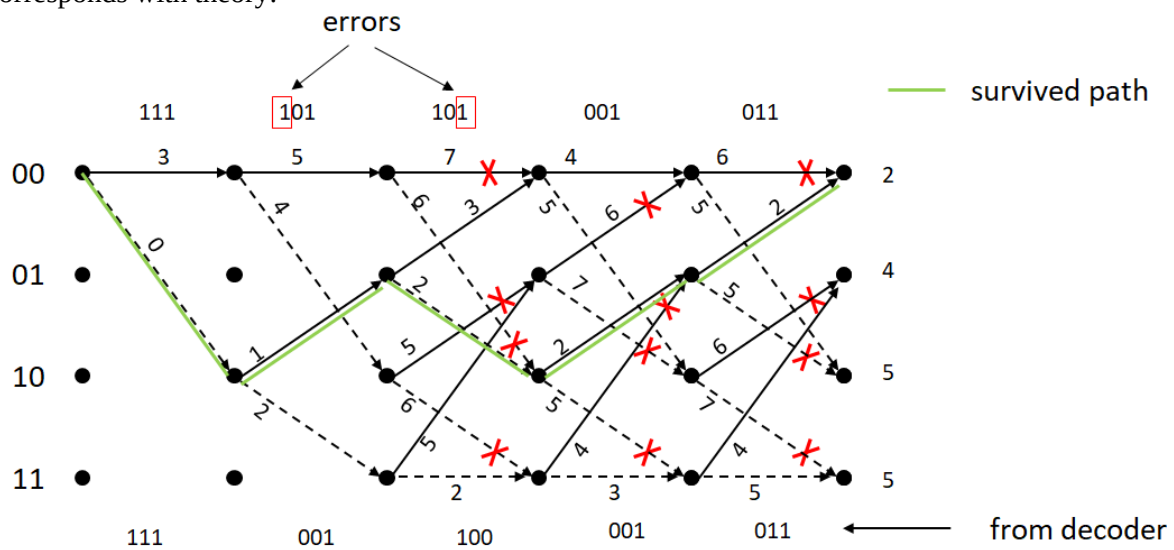


Fig. 3.22: The result of Viterbi decoding.

Convolutional codes with catastrophic error propagation

In case of catastrophic error propagation finite number of input errors will cause an infinite number of errors at the decoder output. Any systematic convolutional code is non-catastrophic. In the following example we will show a catastrophic convolutional code. Let have the following hardware realization shown in Fig. 3.23.

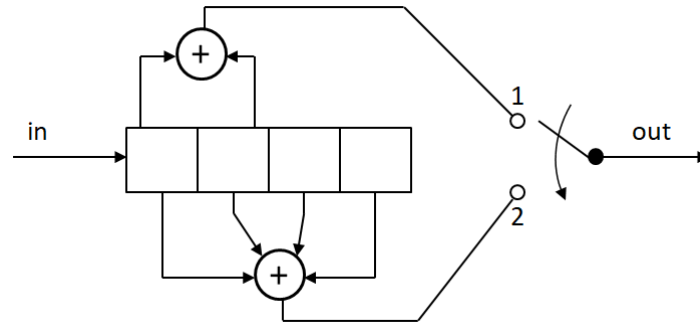


Fig. 3.23: A catastrophic CC encoder circuit.

Generating polynomials are: $g_1(x) = 1 + x$ $g_2(x) = 1 + x + x^2 + x^3$

Let have the input sequence in the form: $i(x) = \frac{1}{1+x} = 1 + x + x^2 + x^3 + \dots$ or $\bar{i} = (1, 1, 1, \dots)$

Then:

$$c_1(x) = i(x)g_1(x) = \frac{1}{1+x} (1+x) = 1 \text{ and } c_2(x) = i(x)g_2(x) = \frac{1}{1+x} (1+x+x^2+x^3) = 1+x^2$$

The resulting code sequence is: $c(x) = (1, 1)x^0 + (0, 0)x + (0, 1)x^2 + (0, 0)x^3 + \dots$ or $\bar{c} = (1, 1, 0, 0, 0, 1, 0, 0, 0, \dots) \rightarrow$ the Hamming weight of the output sequence is finite ($w(\bar{c}) = 3$) in spite of the fact that $w(\bar{i}) = \infty$. In case that in the channel error vector $\bar{e} = (1, 1, 0, 0, 0, 1, 0, 0, 0, \dots)$ occurs then it is obvious that the decoded sequence is a zero sequence ($\hat{\bar{i}} = \bar{0}$). If we now measure the Hamming distance between the sent and received sequence $w(\bar{i} - \hat{\bar{i}}) = \infty$. It means finite number of errors ($w(\bar{e}) = 3$) in the channel case an infinite number of errors at the decoder output. It is possible to show that if the generators polynomials have a common polynomial factor of degree at least 1, then the code is catastrophic.



Coding and decoding of a simple convolutional code is analyzed in [8] in the TIMS experiments: “Convolutional coding”, Emona TIMS Volume D2 – Further & Advanced Digital Experiments, experiment: D2-09.

Lab 3.1: MATLAB exercises - Hamming codes

Unless explicitly allowed in the exercise, the use of any form of cycles or toolbox functions in the implementation is prohibited. Using the vector features of MATLAB is mandatory. Let us not use the Galois toolkit, instead we interpret the default `double` type storing only integer values 0 and 1 as a binary matrix. We can then easily emulate GF(2) calculation using the modulo 2 operation provided by the `mod()` built-in. Furthermore, in this lab we often intentionally store a set of vectors inside one variable – a matrix. We do this in a (for MATLAB) suboptimal row-wise fashion.

Open the provided source file `lab3.m` in MATLAB and use it as a unit test for your implementations of the following exercises.

MATLAB Exercise 3.1.1: Hamming code parameters

Write a public function given the prototype: `function [ n, k ] = HammingCodeNK( r )` that finds the codeword size N and data word size K for a given parameter r of a Hamming code.

MATLAB Exercise 3.1.2: Hamming code matrices

Write a public function given the prototype: `function [ H, G ] = HammingCodeHG( r )` that finds the parity check matrix $\mathbf{H}$ and generator matrix $\mathbf{G}$ for a Hamming code defined by its parameter r . Both matrices must be in systematic form and the orientation of the matrices should facilitate the use of a row-vector data word representation. Do NOT use the toolbox function `hammgen()`.

MATLAB Exercise 3.1.3: Matrix orthogonality test

Write a function given the prototype: `function ort = areOrthogonal( G, H )` that tests the orthogonality of binary matrices $\mathbf{G}$ and $\mathbf{H}$, specified as parameters. The function should return a logical 1 if matrices are orthogonal, 0 otherwise.

MATLAB Exercise 3.1.4: Find all codewords of an LBC

Write a public function given the prototype:

`function Codewords = findAllCodewords( G, datawords )`

that finds all codewords for a given LBC generator matrix $\mathbf{G}$. The second argument is optional, and specifies the data word set in a matrix, with datawords stored row-wise. If the second parameter is present, `findAllCodewords( G, Datawords )` just encodes these using the matrix $\mathbf{G}$. In both cases the resulting binary codewords are stored as rows of a single output matrix. For Octave compatibility, do NOT use the toolbox function `encode()`.

MATLAB Exercise 3.1.5: Compute syndromes

Write a function given the prototype:

`function Syndromes = computeSyndromes( H, words )`

that computes syndromes for a given matrix of row-wise stored binary words and parity check matrix $\mathbf{H}$. Store resulting syndromes row-wise in a single matrix.

MATLAB Exercise 3.1.6: Find code spectrum

Write a function given the prototype:

`function [ Spectrum, dmin ] = findCodeSpectrum( G )`

that finds the distance spectrum of a LBC specified by its generator matrix $\mathbf{G}$ and returns it as a row vector of length N , in which the value at index i stores the number of codewords with Hamming weight i . Optionally it also finds the minimum Hamming distance and returns it as a second output.

MATLAB Exercise 3.1.7: binary combinations matrix

Write a function given a prototype: `function Errors = findGF2Combinations( n, k )` that finds all GF(2) vectors of length n , with exactly k nonzero entries and stores all these vectors row-wise in single output matrix. You may use the functions `nchoosek()` and `sub2ind()` in your implementation.

MATLAB Exercise 3.1.8: all error vectors

Write a function given the prototype:

`function ErrorVectors = findAllErrorVectors( n, tcor )`

that finds all possible error vectors for a syndrome table decoding: For a given codeword size n , and maximum number of correctable errors t_{cor} , generate all error vectors - generate all combinations of binary vectors with number of ones between 0 and t_{cor} (including). Store these as rows in a matrix.

MATLAB Exercise 3.1.9: Syndrome table

Write a function given the prototype:

`function SyndromeTable = findSyndromeTable( H, dmin )`

that finds a syndrome table useful for syndrome decoding of an LBC defined by matrix H .

The Resulting syndromeTable must a structure of two matrices:

`SyndromeTable.Words` - contains all possible error words

`SyndromeTable.Syndromes` - contains corresponding syndromes

This table will be used for mapping error words to syndromes and vice versa. Use the implementation of the function `findGF2Combinations( n, k )` from the previous exercise.

MATLAB Exercise 3.1.10: Syndrome decoding

Write a function given the prototype:

`function CodewordEstimates = decodeSyndromeMethod( H, syndromeTable, Received )`

that decodes received vectors - codewords of an LBC potentially affected by errors, using the syndrome method.

The syndrome table structure is defined in the previous exercise. The third parameter is the set of received binary words to be decoded, stored row-wise in a single matrix. The resulting codewords estimated are stored in the same fashion. You may find the function `ismember()` helpful in your implementation.

MATLAB Exercise 3.1.11: Test syndrome decoding with predictable errors

Use `repmat()` to duplicate the error vector matrix from Exercise 8 to match the size of the codewords matrix from Exercise 4 and sum them (modulo 2) to obtain a set of erroneous receiver vectors, stored as rows of a single matrix. Test your syndrome method decoder using this received matrix: all codewords should be decoded successfully and all errors should get corrected.

MATLAB Exercise 3.1.12: Hamming distance

Write a function given the prototype: `function dh = dHam( Word, Words )` that calculates Hamming distances between the vector word, and all words in matrix Words stored as rows. If matrix is in fact a vector, a single Hamming distance is computed.

Chapter 4: Signal analysis

Signals and their characteristics

According to the probability of their occurrence over time we can divide signals into two big groups:

1. Deterministic - which can be further divided into:
 - a) Periodic: $\rightarrow$ harmonic/inharmonic
 - b) Non-periodic
 - c) Quasiperiodic
2. Random (stochastic) - which can be further divided into:
 - a) Non-stationary
 - b) Stationary
 - c) Ergodic

Deterministic signals: are signals that can be described by a suitable time function that corresponds to a particular transmitted message.

- a) Periodic signals: can be described by harmonic functions: $\sin/\cos$. Their mathematical description satisfies the condition:

$$f(t) = f(t + nT), \text{ where: } t \in (-\infty, \infty), n \in Z, T \text{ is period} \quad (4.1)$$

An example of harmonic and inharmonic signals is on Fig. 4.1:



Fig.4.1: harmonic signal (left), non-harmonic signal (right).

Periodic signals can be decomposed by Fourier series into the sum of harmonic signals:

$$f(t) = \sum_{n=-\infty}^{\infty} \bar{F}_n e^{j\Omega_n t}, \text{ where: } \Omega_n = n\Omega_1. \quad (4.2)$$

Their spectrum is discrete, see Fig.4.2.

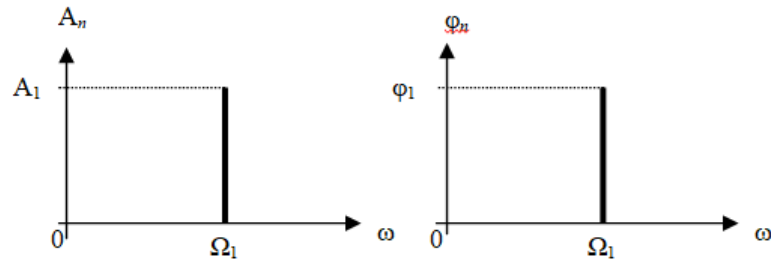


Fig.4.2 An amplitude and phase spectrum of signal: $x(t) = A_1 \cos(\Omega_1 t)$.

b) Non-periodic signals: their mathematical model is a time function defined on:

- a finite interval: $t \in \langle t_1, t_2 \rangle$

- a semi-infinite interval: $t \in (t, \infty)$

outside these intervals, the function has zero value. Their spectrum is continuous.

c) Quasiperiodic signals: are created by the sum of a set of harmonic signals, which does not satisfy the periodicity condition—that is, for at least one pair of frequencies, their ratio is an irrational number:

$$f(t) = \sum_{k=-\infty}^{\infty} \bar{F}_k e^{j\Omega_k t}, \text{ but: } \Omega_k \neq k\Omega_1 \quad (4.3)$$

According to the mean normalized power or energy we can divide signals into two big groups:

1. Power signals
2. Energy signals

1. Power signals - have a non-zero but finite mean normalized power: $0 < P_x < \infty$ – (the subscript "x" denotes normalization at resistance $R = 1\Omega$)

Note: $P = U I = I^2 R = \frac{U^2}{R}$, if: $R = 1\Omega \rightarrow P = x^2$, where x can be U or I.

Mean normalized power for a real signal $x(t)$ with a period $T > 0$ can be expressed as:

$$P_x = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} x^2(t) dt \quad [W] \quad (4.4)$$

P_x is the mean normalized power dissipated on the load during a finite time interval: $\left\langle -\frac{T}{2}, \frac{T}{2} \right\rangle$.

The following classes of signals belong to power signals:

- Periodic signals
- Quasiperiodic signals
- Stochastic signals

Their spectrum is discrete and we can calculate it using Fourier series.

2. Energy signals - have a non-zero but finite mean normalized energy: $0 < E_x < \infty$

Mean normalized energy for a real signal $x(t)$:

$$E_x = \int_{-\infty}^{\infty} x^2(t) dt \quad [J] \quad (4.5)$$

In practice, we are mostly interested in the mean normalized energy dissipated at the load

$R = 1\Omega$ during a finite time interval $\left\langle -\frac{T}{2}, \frac{T}{2} \right\rangle$, where T is the symbol period:

$$E_x^T = \int_{-\frac{T}{2}}^{\frac{T}{2}} x^2(t) dt \quad (4.6)$$

To energy signals belong:

- Non-periodic signals

Their spectrum is continuous and we can calculate it using Fourier transform.

PSD (Power Spectral Density) - expresses the distribution of signal power in the frequency domain and indicates how much signal power is per [Hz] of bandwidth. It is denoted by $G_x(f)$. Formally:

$G_x(f) = |X(f)|^2 \quad [W / Hz]$, where $X(f)$ is the amplitude spectrum of the signal.

For real periodic signals: $G_x(f) = \sum_{n=-\infty}^{\infty} |c_n|^2 \delta(f - nf_0)$, where n is the harmonic component index.

ESD (Energy Spectral Density) - expresses the distribution of signal energy in the frequency domain and indicates how much signal energy is per [Hz] of bandwidth. It is denoted by $\psi_x(f)$. Formally:

$\psi_x(f) = |X(f)|^2 \quad [J / Hz]$, where $X(f)$ is the amplitude spectrum of the signal.

Time domain and frequency domain are two different views of the same thing, see Fig.4.3.

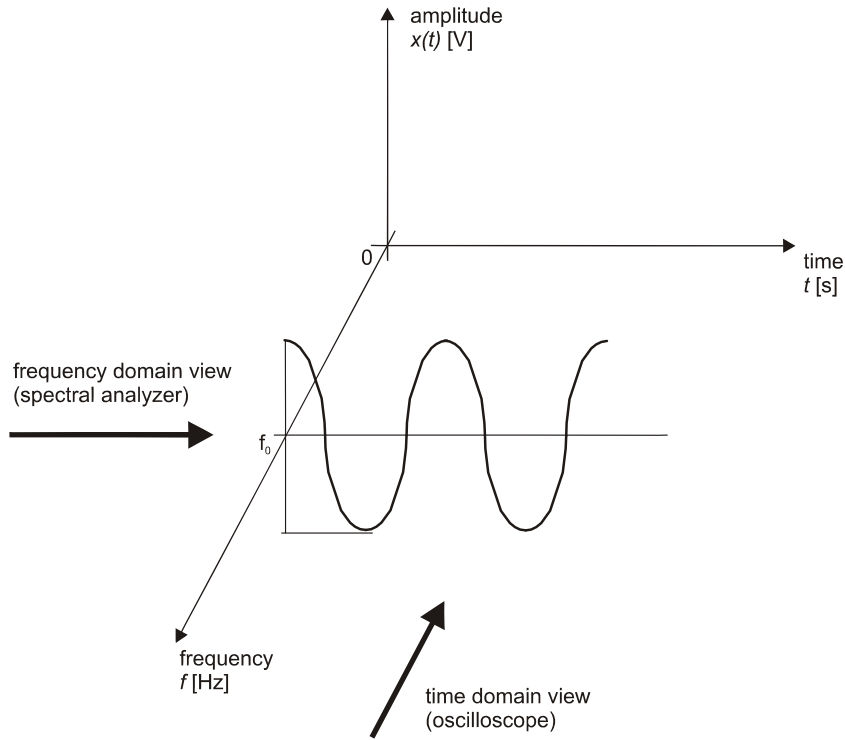


Fig. 4.3: Time and frequency domain.

Therefore, it is logical that the power/energy of the signal in the time domain must be equal to the power/energy of the signal in the frequency domain. This is formally expressed by Parseval's theorem:

$$\text{Real power signals: } P_x = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} x^2(t) dt = \sum_{n=-\infty}^{\infty} |c_n|^2 \quad x(t) \xleftrightarrow{FR} |c_n| \quad (4.7)$$

$$\text{Real energy signals: } E_x = \int_{-\infty}^{\infty} x^2(t) dt = \int_{-\infty}^{\infty} |X(f)|^2 df \quad x(t) \xleftrightarrow{FT} X(f) \quad (4.8)$$

Then:

$$P_x = \int_{-\infty}^{\infty} G_x(f) df = \int_0^{\infty} G_x(f) df \quad , \text{ or: } P_x = \int_{-\infty}^{\infty} G_x(f) df = 2 \int_0^{\infty} G_x(f) df$$

double-sided
single sided
double-sided

$$E_x = \int_{-\infty}^{\infty} \psi_x(f) df = \int_0^{\infty} \psi_x(f) df \quad , \text{ or: } E_x = \int_{-\infty}^{\infty} \psi_x(f) df = 2 \int_0^{\infty} \psi_x(f) df$$

double-sided
single sided
double-sided



Parseval's theorem is analyzed in [9] the TIMS experiment: "Parseval's Theorem: Relationship between time & frequency domain", Signals & Systems Experiments with Emona TIMS Volume S1 - Fundamentals of Signals & Systems, experiment: S1-13.

Exercise 4.1:

Calculate the mean normalized power of signal: $x(t) = A \cos(2\pi f_0 t)$ in:

- (a) Time domain
- (b) Frequency domain

Solution:

a)

We applied the formula: $\cos^2 x = \frac{1}{2}(1 + \cos 2x)$

$$\begin{aligned} P_x &= \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} x^2(t) dt = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} A^2 \cos^2(2\pi f_0 t) dt = \frac{A^2}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} \left(\frac{1}{2} + \frac{\cos(4\pi f_0 t)}{2} \right) dt = \\ &= \frac{A^2}{T} \left[\frac{1}{2} t \right]_{-\frac{T}{2}}^{\frac{T}{2}} + \frac{A^2}{2T} \left[\frac{\sin(4\pi f_0 t)}{4\pi f_0} \right]_{-\frac{T}{2}}^{\frac{T}{2}} = \frac{A^2}{2T} \left[\left(\frac{T}{2} + \frac{T}{2} \right) + \frac{1}{4\pi f_0} \left(\sin(4\pi f_0 \frac{T}{2}) - \sin(-4\pi f_0 \frac{T}{2}) \right) \right] = \\ &= \frac{A^2}{2T} \left[T + \frac{1}{4\pi f_0} 2\sin(2\pi f_0 T) \right] = \frac{A^2}{2} \end{aligned}$$

b)

We have to determine the complex Fourier coefficients, what is easy since we have a monochromatic signal with power spectrum:

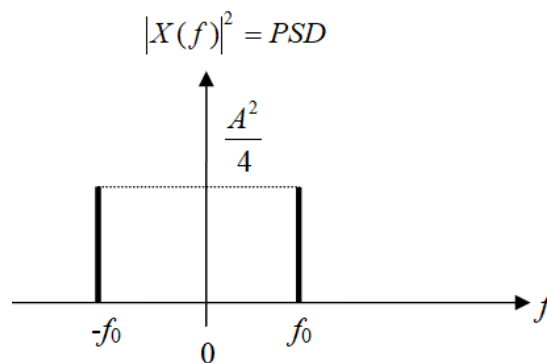


Fig. 4.4: Two-sided PSD of a cosine signal.

In general: $|c_n| = \frac{1}{2} \sqrt{a_n^2 + b_n^2}$.

In our case: $a_0 = 0$ - there is no DC component

$$a_1 = A$$

$b_0 = 0$ - $\cos(x)$ is an even function

$$c_1 = c_{-1} = \frac{A}{2}, \text{ then: } P_x = \sum_{n=-1}^1 |c_n|^2 = 2 \times |c_1|^2 = 2 \times \left(\frac{A}{2}\right)^2 = \frac{A^2}{2}$$

■

Exercise 4.2:

The amplitude spectrum of a real periodic signal representing a voltage is given in Fig. 4.5. Calculate the power of the signal in [dB] dissipated on a real load $R = 0.1 [\Omega]$.

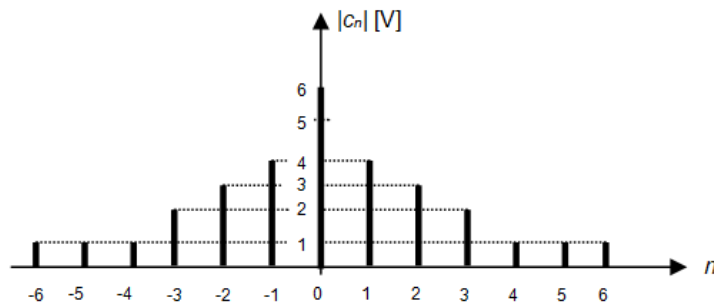


Fig. 4.5: Two sided PSD of a sum of harmonic signals.

Solution:

The easiest way is to use a frequency domain:

$$P_x = \sum_{n=-6}^6 |c_n|^2 = 6^2 + 2 \times (4^2 + 3^2 + 2^2 + 1^2 + 1^2 + 1^2) = 100 [W]$$

$$P_R = \frac{P_x}{R} = \frac{100}{0.1} = 1000 \quad P_R [dB] = 10 \times \log_{10}(1000) = 30 [dB]$$

■

Exercise 4.3:

For a signal: $x(t) = A \text{rect}\left(\frac{t}{T_0}\right)$, $A > 0$, $A \in \mathbb{R}$ calculate:

- ESD
- E_x using frequency and time domain

If $A = 1$ and $T_0 = 1$ then the signal looks like in Fig. 4.6.

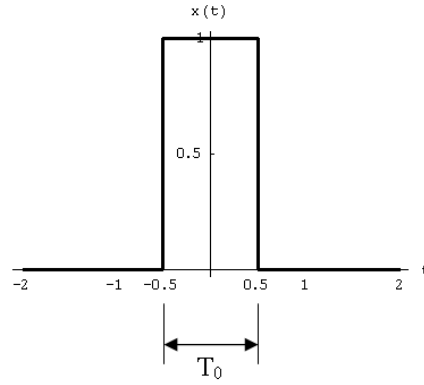


Fig. 4.6: A rectangular pulse – time domain.

Solution:

a) Since: $ESD = \psi(f) = |X(f)|^2$ we have to first calculate the amplitude spectrum:

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft} dt = \int_{-\frac{T_0}{2}}^{\frac{T_0}{2}} A e^{-j2\pi ft} dt = A \left[\frac{e^{-j2\pi ft}}{-j2\pi f} \right]_{-\frac{T_0}{2}}^{\frac{T_0}{2}} =$$

$$A \frac{e^{-j2\pi f \frac{T_0}{2}} - e^{j2\pi f \frac{T_0}{2}}}{-j2\pi f} = AT_0 \frac{\sin 2\pi f \frac{T_0}{2}}{\pi f T_0} = AT_0 \frac{\sin \pi f T_0}{\pi f T_0} = AT_0 \text{ sinc}(f T_0) = AT_0 \text{ si}(\pi f T_0)$$

Then: $ESD = \psi(f) = |X(f)|^2 = A^2 T_0^2 \text{ sinc}^2(f T_0)$, see Fig. 4.7.

b) Using frequency domain: $E_x = \int_{-\infty}^{\infty} \psi(f) df = \int_{-\infty}^{\infty} |X(f)|^2 df = \int_{-\infty}^{\infty} A^2 T_0^2 \text{ sinc}^2(f T_0) df = A^2 T_0$

This integral can be evaluated using e.g. Mathematica or MATLAB.

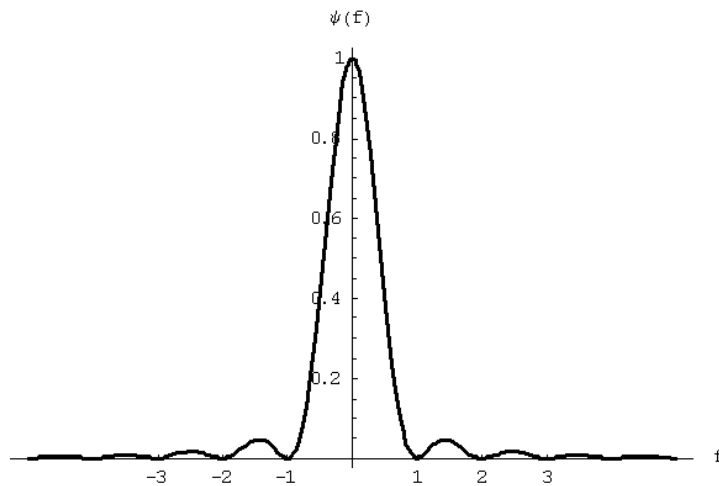


Fig. 4.7: ESD of a rectangular pulse.

Using time domain: $E_x = \int_{-\frac{T_0}{2}}^{\frac{T_0}{2}} x^2(t)dt = \int_{-\frac{T_0}{2}}^{\frac{T_0}{2}} A^2 dt = A^2 \left[\frac{T_0}{2} - \left(-\frac{T_0}{2} \right) \right] = A^2 T_0$

■

Autocorrelation function and spectral density

Energy signals: $R_x(\tau) = \int_{-\infty}^{\infty} x(t)x(t+\tau)dt$

Properties of $R_x(\tau)$:

1. $R_x(\tau) = R_x(-\tau)$
2. $R_x(\tau) \leq R_x(0)$
3. $R_x(\tau) \xleftrightarrow{FT} \psi_x(f)$
4. $R_x(0) = E_x$

Power signals: $R_x(\tau) = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} x(t)x(t+\tau)dt$

Properties of $R_x(\tau)$:

1. $R_x(\tau) = R_x(-\tau)$
2. $R_x(\tau) \leq R_x(0)$
3. $R_x(\tau) \xleftrightarrow{FT} G_x(f)$
4. $R_x(0) = P_x$

Properties of spectral densities:

a) ESD

1. $\psi_x(f) \geq 0$
2. $\psi_x(f) = \psi_x(-f)$
3. $\int_{-\infty}^{\infty} \psi_x(f)df = E_x$
4. $\psi_x(f) \xrightarrow{F^{-1}} R_x(\tau)$

b) PSD

1. $G_x(f) \geq 0$
2. $G_x(f) = G_x(-f)$
3. $\int_{-\infty}^{\infty} G_x(f)df = P_x$
4. $G_x(f) \xrightarrow{F^{-1}} R_x(\tau)$

Exercise 4.4:

- a) Calculate the autocorrelation function of a signal: $x(t) = A \cos(2\pi f_s t)$, $A > 0$, $A \in \mathbb{R}$
 b) Calculate the mean normalized power using the property #4 of $R_x(\tau)$

Solution:

a)

$$\begin{aligned}
 R_x(\tau) &= \frac{1}{T_s} \int_{-\frac{T_s}{2}}^{\frac{T_s}{2}} x(t)x(t+\tau)dt = \frac{1}{T_s} \int_{-\frac{T_s}{2}}^{\frac{T_s}{2}} A \sin(2\pi f_s t) A \sin[2\pi f_s (t+\tau)]dt \\
 &= \frac{A^2}{T_s} \int_{-\frac{T_s}{2}}^{\frac{T_s}{2}} \sin(2\pi f_s t) [\sin(2\pi f_s t) \cos(2\pi f_s \tau) + \cos(2\pi f_s t) \cos(2\pi f_s \tau)]dt \\
 &= \frac{A^2}{T_s} \int_{-\frac{T_s}{2}}^{\frac{T_s}{2}} \sin^2(2\pi f_s t) \cos(2\pi f_s \tau) + \sin(2\pi f_s t) \cos(2\pi f_s t) \cos(2\pi f_s \tau)dt \\
 &= \frac{A^2}{T_s} \cos(2\pi f_s \tau) \int_{-\frac{T_s}{2}}^{\frac{T_s}{2}} \sin^2(2\pi f_s t)dt = \frac{A^2}{2} \cos(2\pi f_s \tau)
 \end{aligned}$$

b) $P_x = R_x(0) = \frac{A^2}{2}$

■

Exercise 4.5:

Find the autocorrelation function of the function $x(t)$ shown in the Fig. 4.8.

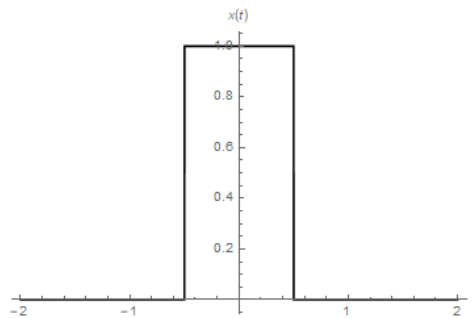


Fig. 4.8: A rectangular pulse – time domain.

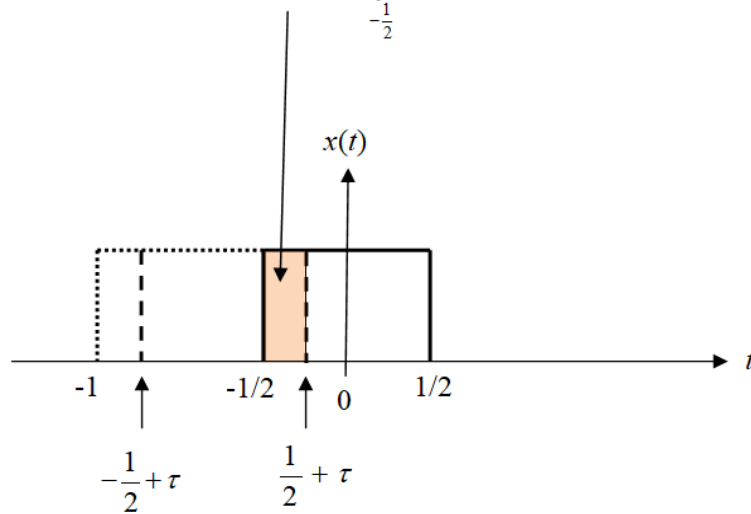
Solution:

One way is to find an amplitude spectrum of the given function, then calculate ESD and use the inverse Fourier transform to find the autocorrelation function. The other way is to use the definition:

The autocorrelation function must be calculated over all significant displacements – in this case 2:

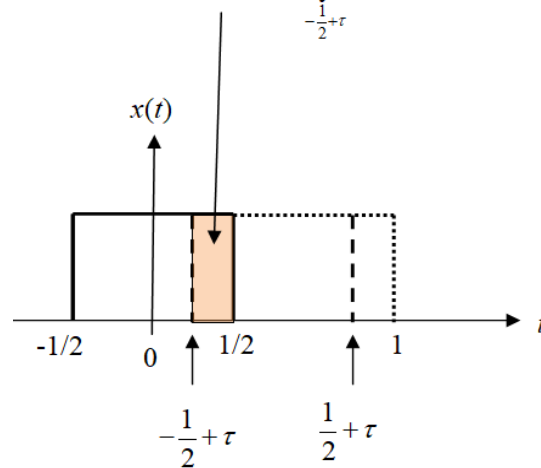
For: $\tau \in \langle -1, 0 \rangle$:

$$R_x(\tau) = \int_{-\frac{1}{2}}^{\frac{1}{2}+\tau} 1 \times 1 dt = 1 + \tau$$



For: $\tau \in \langle 0, 1 \rangle$:

$$R_x(\tau) = \int_{-\frac{1}{2}+\tau}^{\frac{1}{2}} 1 \times 1 dt = 1 - \tau$$



The resulting $R_x(\tau)$:

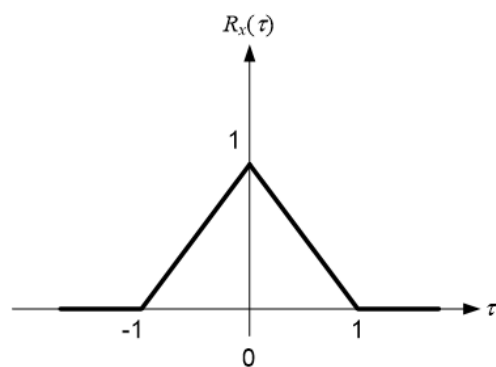


Fig.4. 9: Autocorrelation function of a rectangular pulse.

■

Signal bandwidth

In order to control the degree of interference between devices sharing the same (or adjacent) frequency band in a defined way, it was necessary to introduce (application-specific) frequency band definitions.

The PSD of real signals (valid for most digital modulations) has the shape of the square of the $\text{si}(x)$ function:

$$G_x(f) = T \left\{ \frac{\sin[\pi(f - f_c)T]}{\pi(f - f_c)T} \right\}^2 \quad (4.9)$$

where f_c is the carrier frequency, T is the pulse period.

Practical Frequency band definitions:

1. Half-power bandwidth: W_{HP} - the bandwidth within which the PSD drops to half of the maximum value (i.e. by 3 [dB])
2. Noise-equivalent bandwidth: W_{NE} - is the bandwidth of a Nyquist filter, with the same gain in the passband as the real system, which passes the same AWGN power as the real system:

$$P_n = W_{NE} \times G_x(f_c) \Rightarrow W_{NE} = \frac{P_n}{G_x(f_c)} \quad (4.10)$$

3. Null-to-null bandwidth: $W_{null-to-null}$ - is the bandwidth defined by the first zeros around the PSD maximum. It is the most widely used definition of bandwidth since it is the easiest to measure; it can be read directly from a spectrum analyzer.
4. Fractional power containment bandwidth: W_{FPC} - is the band in which 99% of the signal power is located, such that 0.5% of P_x is above the upper cutoff frequency f_u and 0.5% of P_x is below the lower cutoff frequency f_l .
5. Bounded PSD: W_{BPSD} - defines the maximum out-of-band PSD value, where the PSD must not exceed - 35 [dB] or - 50 [dB] relative to the maximum.

All the above mentioned frequency band definitions are graphically presented in Fig. 4.10.

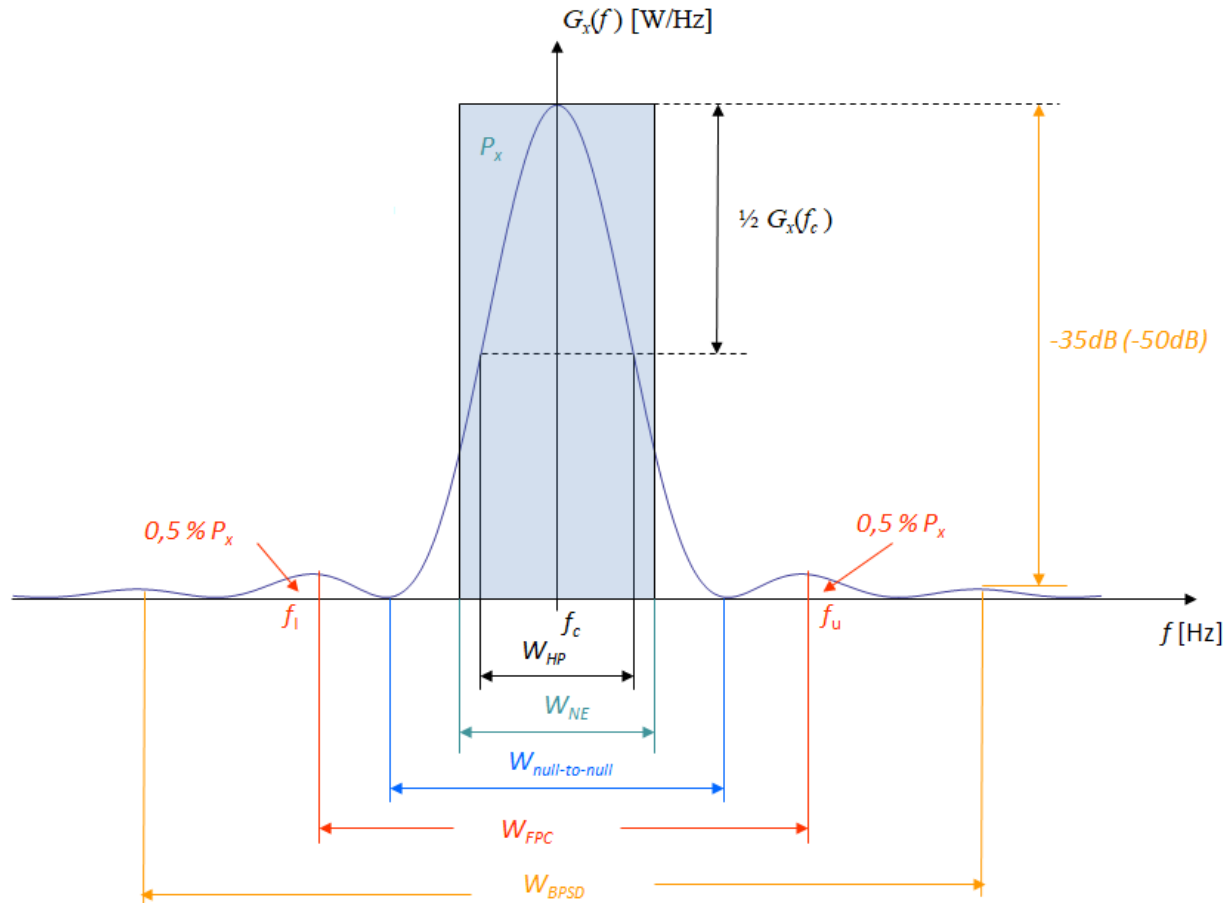


Fig. 4.10: Frequency band definitions. Note: W_{FPC} and W_{BPSD} are not to scale.

Exercise 4.5:

Single sided PSD is defined as: $G_x(f) = T_s \left\{ \frac{\sin[\pi(f - 2.4 \times 10^9)T_s]}{\pi(f - 2.4 \times 10^9)T_s} \right\}^2$ where $T_s = 10^{-7}$ [s].

Calculate:

- W_{HP}
- W_{NE}
- $W_{null-to-null}$

Solution:

As follows from the assignment in our case – the carrier frequency: $f_c = 2.4$ [GHz]

- We have to find max of PSD: $\max \{G_x(f)\} = G_x(f_c) = T_s = 10^{-7} \left[\frac{W}{Hz} \right]$, since:

$$\lim_{x \rightarrow 0} \frac{\sin(x)}{x} = \lim_{x \rightarrow 0} \frac{\sin(x)'}{x'} = \lim_{x \rightarrow 0} \frac{\cos(x)}{1} = 1$$

From the definition of W_{HP} follows: $G_x(f') = \frac{1}{2}G_x(f_c)$, where f' is the unknown frequency.

$$10^{-7} \times \left\{ \frac{\sin \left[\pi(f' - 2.4 \times 10^9) \times 10^{-7} \right]}{\pi(f' - 2.4 \times 10^9) \times 10^{-7}} \right\}^2 = \frac{1}{2} \times 10^{-7}$$

$$\frac{\sin \left[\pi(f' - 2.4 \times 10^9) \times 10^{-7} \right]}{\pi(f' - 2.4 \times 10^9) \times 10^{-7}} = \pm \sqrt{\frac{1}{2}}$$

We will use the substitution: $\pi(f' - 2.4 \times 10^9) \times 10^{-7} = \alpha$

Then: $\sin \alpha = \pm \frac{1}{\sqrt{2}}$ $\rightarrow$ this equation has only numerical solution. We can use iteration to solve

it. As the initial value we can use: $\alpha_0 = 1, 0$

$$\alpha_i = \pm \sin(\alpha_{i-1}) \times \sqrt{2}$$

$\vdots$

$$\alpha = \pm 1.39156 \text{ [rad]}$$

$$\pi(f' - 2.4 \times 10^9) \times 10^{-7} = \pm 1.39156$$

$$f' - 2.4 \times 10^9 = \pm \frac{1.39156 \times 10^7}{\pi}$$

$$f' = 2.4 \times 10^9 \pm 4.429 \times 10^6 \text{ [Hz]}$$

$$f_u = 2.4 \times 10^9 + 4.429 \times 10^6 \text{ [Hz]} \quad \text{and} \quad f_l = 2.4 \times 10^9 - 4.429 \times 10^6 \text{ [Hz]}$$

$$W_{HP} = f_u - f_l \cong 8,858 \times 10^6 \text{ [Hz]}$$

b)

$W_{NE} = \frac{P_n}{G_x(f_c)} \rightarrow$ We need to calculate the noise power. There are two ways how to do it:

$$1. P_n = \int_{-\infty}^{\infty} G_x(f) df$$

$$2. P_n = R_x(0)$$

Obviously the second way is easier: $R_x(\tau) = FT^{-1} \{G_x(f)\}$:

$$T_s \sin^2(fT_s) \xrightarrow{FT^{-1}} R_x(\tau) = \begin{cases} 1 - \frac{|\tau|}{T_s} & |\tau| \leq T_s \\ 0 & |\tau| > T_s \end{cases} \Rightarrow R_x(0) = 1 = P_n$$

$$\text{Therefore: } W_{NE} = \frac{1}{10^{-7}} = 10 \times 10^6 \text{ [Hz]}$$

c)

We will start with the bandwidth definition: $G_x(f') \stackrel{!}{=} 0$ and:

$$10^{-7} \times \left\{ \frac{\sin \left[\pi(f' - 2.4 \times 10^9) \times 10^{-7} \right]}{\pi(f' - 2.4 \times 10^9) \times 10^{-7}} \right\}^2 = 0$$

We use the same substitution as in case b): $10^{-7} \times \sin^2 \alpha = 0 \Rightarrow \alpha = 0 \pm k \times \pi, k = 1, 2, \dots$

We are interested in case when: $k = \pm 1$:

$$10^{-7} \times \pi(f' - 2.4 \times 10^9) = \pm \pi$$

$$f' - 2.4 \times 10^9 = \pm 10^7 \text{ [Hz]} \Rightarrow f_u = 2.4 \times 10^9 + 10^7 \quad f_l = 2.4 \times 10^9 - 10^7 \text{ [Hz]}$$

$$\text{Therefore: } W_{\text{null-to-null}} = f_u - f_l = 20 \times 10^6 \text{ [Hz]}$$

■

GFT – Generalized Fourier Transform

An arbitrary information signal can be represented as a vector in the so-called signal space. The basis of this signal space is a set of orthogonal functions. Then, an arbitrary information signal can be expressed as a linear combination of the basis functions of the given signal space. It can be shown that the mutual distance of two signals in a signal space (called Euclidean) can be expressed as: $d = \sqrt{E_d}$, where E_d is the energy of the difference signal:

$$E_d = \int_0^T [s_1(t) - s_2(t)]^2 dt \quad (4.11)$$

An example of antipodal signaling is given in Fig. 4.1.

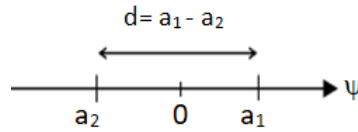


Fig. 4.11: Antipodal signalling in one dimensional signal space.

In Euclidean space, the farther apart 2 points are, the less similar they are, and their "dissimilarity" is related to the energy of the difference signal: $d = \sqrt{E_d}$. Let's have a signal alphabet:

$S = \{s_1(t), s_2(t) \dots s_M(t)\}$. The problem is to find $N \leq M$ basis functions of an N - dimensional orthogonal space that can be used to express an arbitrary signal from the signal alphabet as its linear combination with the basis functions. An N - dimensional orthogonal space is characterized by N linearly independent functions, which are called basis functions: $\{\psi_j(t)\} \quad j = 1, \dots, N$. All basis functions must satisfy the orthogonality conditions:

$$\int_0^T \psi_j(t) \psi_k(t) dt = K_j \delta_{jk} \quad 0 \leq t \leq T \quad j, k = 1, 2, \dots, N, \quad K_j \text{ is a constant}, \quad \delta_{jk} = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases} \quad (4.12)$$

If the basis functions are normalized, i.e. $K_j = 1$, the space is called orthonormal. Orthogonality implies that each $\psi_j(t)$ must be linearly independent of the others. Example of a 3D signal space is shown in Figure 4.12.

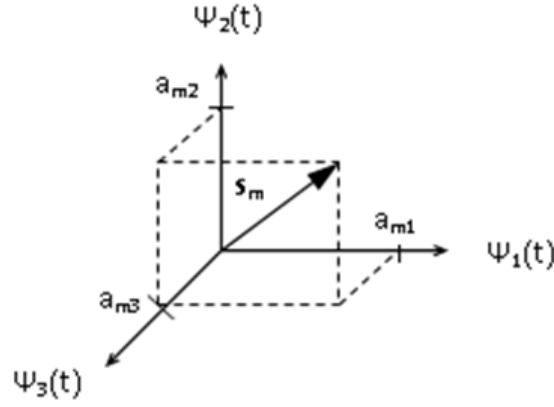


Fig. 4.12: An example of orthogonal signals in 3D signal space.

Each point of signal space represents one signal. The distance of the point from the origin is the signal amplitude. In this case a_{m1}, a_{m2}, a_{m3} are signal coordinates of s_m . Geometrically: each $\psi_j(t)$ is perpendicular to all the others $\psi_k(t)$ for $j \neq k$. If $\psi_j(t)$ corresponds to a voltage or current on the load of $1 [\Omega]$, then the normalized energy dissipated on the load in T seconds is:

$$E_j = \int_0^T \psi_j^2(t) dt = K_j \quad (4.13)$$

It can be shown that any signal from the signal alphabet can be expressed as a linear combination of N - orthogonal (basis) functions $\psi_1(t), \dots, \psi_N(t)$, where $N \leq M$:

$$\begin{aligned} s_1(t) &= a_{11}\psi_1(t) + a_{12}\psi_2(t) + \dots + a_{1N}\psi_N(t) \\ s_2(t) &= a_{21}\psi_1(t) + a_{22}\psi_2(t) + \dots + a_{2N}\psi_N(t) \\ &\vdots \\ s_M(t) &= a_{M1}\psi_1(t) + a_{M2}\psi_2(t) + \dots + a_{MN}\psi_N(t) \end{aligned} \quad (4.14)$$

In a compact form: $s_i(t) = \sum_{j=1}^N a_{ij} \psi_j(t)$ $i = 1 \dots M, N \leq M$ (2). The coefficient a_{ij} is the value of the component $\psi_j(t)$ in the signal $s_i(t)$:

$$a_{ij} = \frac{1}{K_j} \int_0^T s_i(t) \psi_j(t) dt \quad i = 1, 2, \dots, M \quad j = 1, 2, \dots, N \quad 0 \leq t \leq T \quad (4.15)$$

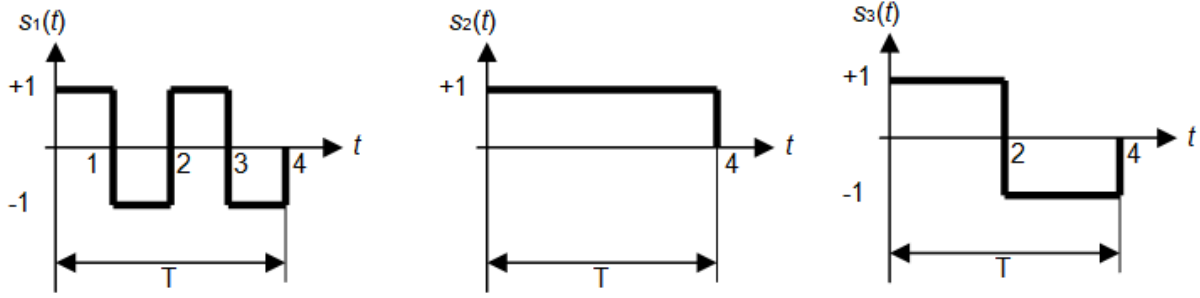
Expressions (4.12), (4.14), and (4.15) form the Generalized Fourier Transform - GFT. Using GFT An arbitrary signal can be expressed as a vector $\bar{s}_i = (a_{i1}, \dots, a_{iN})$ with coordinates in N - dimensional signal space. Thus, if the set of basis functions is already determined, then each transmitted signal is completely determined by its coefficients. Normalized energy of $\bar{s}_i$ over time T :

$$E_i = \sum_{j=1}^N a_{ij}^2 K_j \quad (4.16)$$

In the case of orthonormal functions: $K_j = 1 \rightarrow E_j = \sum_{i=1}^N a_{ij}^2$

Exercise 4.6:

Find out whether the given set of functions is orthogonal:



Solution: Let us evaluate the equation (4.12) for all combinations of $s_1(t)$, $s_2(t)$, $s_3(t)$:

$$\begin{aligned} \int_0^T s_1(t) s_2(t) dt &= \int_0^1 s_1(t) s_2(t) dt + \int_1^2 s_1(t) s_2(t) dt + \int_2^3 s_1(t) s_2(t) dt + \int_3^4 s_1(t) s_2(t) dt \\ &= \int_0^1 1 \times 1 dt + \int_1^2 (-1) \times 1 dt + \int_2^3 1 \times 1 dt + \int_3^4 (-1) \times 1 dt = 0 \end{aligned}$$

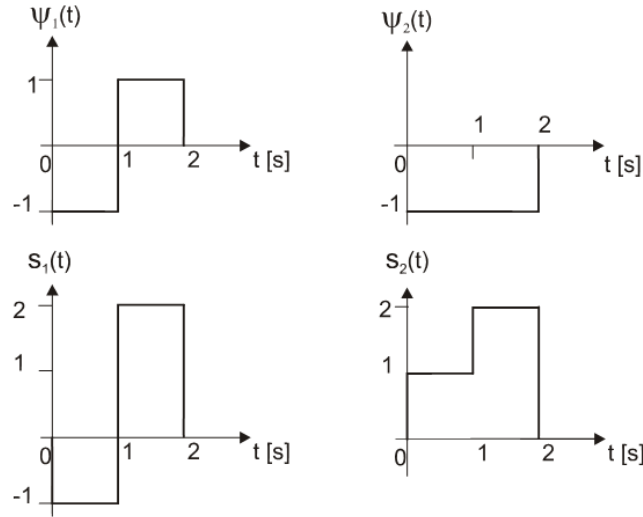
$$\begin{aligned} \int_0^T s_1(t) s_3(t) dt &= \int_0^1 s_1(t) s_3(t) dt + \int_1^2 s_1(t) s_3(t) dt + \int_2^3 s_1(t) s_3(t) dt + \int_3^4 s_1(t) s_3(t) dt \\ &= \int_0^1 1 \times 1 dt + \int_1^2 (-1) \times 1 dt + \int_2^3 1 \times (-1) dt + \int_3^4 (-1) \times (-1) dt = 0 \end{aligned}$$

$$\begin{aligned} \int_0^T s_2(t) s_3(t) dt &= \int_0^1 s_2(t) s_3(t) dt + \int_1^2 s_2(t) s_3(t) dt + \int_2^3 s_2(t) s_3(t) dt + \int_3^4 s_2(t) s_3(t) dt \\ &= \int_0^1 1 \times 1 dt + \int_1^2 1 \times 1 dt + \int_2^3 1 \times (-1) dt + \int_3^4 1 \times (-1) dt = 0 \end{aligned}$$

Therefore the given set is orthogonal. ■

Exercise 4.7:

The base of the signal space is given by the functions: $\psi_1(t), \psi_2(t)$. Analytically find the coordinates of the signals: $s_1(t), s_2(t)$ in the given basis:



Solution:

Simply evaluate the integral (4.13):

$$K_1 = \int_0^T \psi_1^2(t) dt = \int_0^1 (-1)^2 dt + \int_1^2 1^2 dt = 1 + 1 = 2 \quad K_2 = \int_0^T \psi_2^2(t) dt = \int_0^2 (-1)^2 dt = 2$$

$$a_{11} = \frac{1}{K_1} \int_0^T s_1(t) \psi_1(t) dt = \frac{1}{2} \times \left[\int_0^1 (-1) \times (-1) dt + \int_1^2 2 \times 1 dt \right] = \frac{3}{2}$$

$$a_{12} = \frac{1}{K_2} \int_0^T s_1(t) \psi_2(t) dt = \frac{1}{2} \times \left[\int_0^1 (-1) \times (-1) dt + \int_1^2 2 \times (-1) dt \right] = -\frac{1}{2}$$

$$a_{21} = \frac{1}{K_1} \int_0^T s_2(t) \psi_1(t) dt = \frac{1}{2} \times \left[\int_0^1 1 \times (-1) dt + \int_1^2 2 \times 1 dt \right] = \frac{1}{2}$$

$$a_{22} = \frac{1}{K_2} \int_0^T s_2(t) \psi_2(t) dt = \frac{1}{2} \times \left[\int_0^1 1 \times (-1) dt + \int_1^2 2 \times (-1) dt \right] = -\frac{3}{2}$$

■

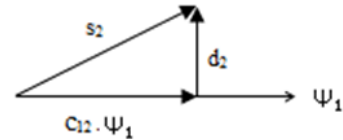
Gram–Schmidt orthogonalization procedure

This method enables to find an appropriate basis function set for any given signal alphabet. Let's have signal alphabet: $S = \{s_1(t), s_2(t) \dots s_M(t)\}$. It doesn't matter which function we start with, e.g. $s_1(t)$:

$$1. \quad \text{First basis function: } \psi_1(t) = \frac{s_1(t)}{\sqrt{E_1}} \quad E_1 = \int_0^T s_1^2(t) dt$$

2. Second basis function (it has to be orthogonal to the first one):

$$\psi_2(t) = \frac{d_2(t)}{\sqrt{E_2}} \quad E_2 = \int_0^T d_2^2(t) dt$$

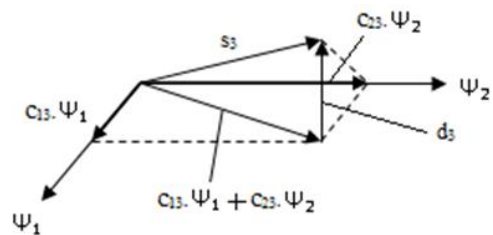


$d_2(t) = s_2(t) - c_{12}\psi_1(t) \rightarrow$ removal of the component of $s_2(t)$ projected in $\psi_1(t)$ - results in signal $d_2(t)$ that is perpendicular to $\psi_1(t)$.

$$c_{12} = \int_0^T s_2(t)\psi_1(t) dt$$

3. Third basis function (it has to be orthogonal to the previous two):

$$\psi_3(t) = \frac{d_3(t)}{\sqrt{E_3}} \quad E_3 = \int_0^T d_3^2(t) dt$$



$$d_3(t) = s_3(t) - c_{13}\psi_1(t) - c_{23}\psi_2(t)$$

$$c_{13} = \int_0^T s_3(t)\psi_1(t) dt \quad c_{23} = \int_0^T s_3(t)\psi_2(t) dt$$

$\vdots$

and so on up to:

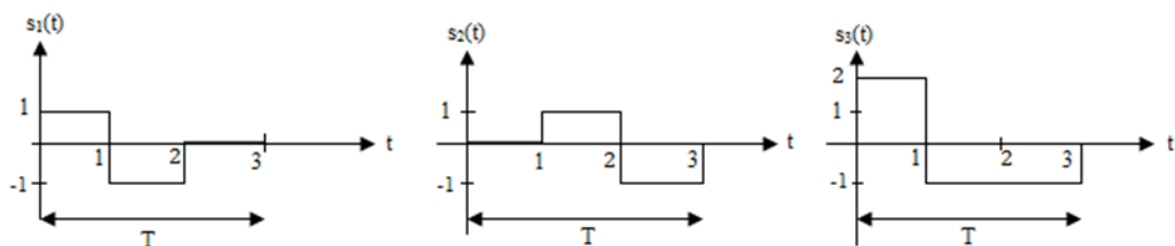
$$\psi_l(t) = \frac{d_l(t)}{\sqrt{E_l}} \quad E_l = \int_0^T d_l^2(t) dt \quad d_l(t) = s_l(t) - \sum_{j=1}^{l-1} c_{jl}\psi_j(t)$$

Note:

- The set $\{\psi_i(t)\}$ is not unique for a given $\{s_i(t)\}$, but in any case the distances between the signals remain unchanged!
- If $N < M$ some signals from S are linearly dependent (zero signals cannot be an element of the basis).

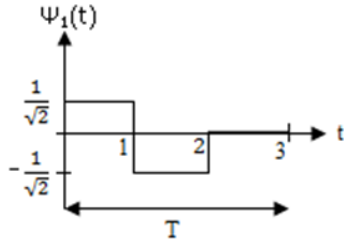
Exercise 4.8:

Find the basis functions of the signals:



Solution:

$$\Psi_1(t) = \frac{s_1(t)}{\sqrt{E_1}} \quad E_1 = \int_0^T s_1^2(t) dt = \int_0^1 1^2 dt + \int_1^2 (-1)^2 dt + \int_2^3 0^2 dt = 1 + 1 = 2$$



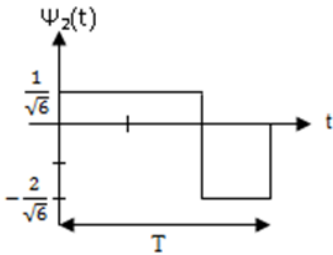
$$\psi_2(t) = \frac{d_2(t)}{\sqrt{E_2}} \quad d_2(t) = s_2(t) - c_{12}\psi_1(t) = s_2(t) + \frac{1}{\sqrt{2}}\psi_1(t)$$

$$E_2 = \int_0^T d_2^2(t) dt = \int_0^1 \left(\frac{1}{2}\right)^2 dt + \int_1^2 \left(\frac{1}{2}\right)^2 dt + \int_2^3 (-1)^2 dt = \frac{1}{4} + \frac{1}{4} + 1 = \frac{3}{2}$$

$$c_{12} = \int_0^T s_2(t)\psi_1(t) dt = \int_0^1 0 \times \frac{1}{\sqrt{2}} dt + \int_1^2 1 \times \left(-\frac{1}{\sqrt{2}}\right) dt + \int_2^3 (-1) \times 0 dt = -\frac{1}{\sqrt{2}}$$

$$d_2(t) = s_2(t) + \frac{1}{\sqrt{2}}\psi_1(t) = s_2(t) + \frac{1}{\sqrt{2}} \frac{s_1(t)}{\sqrt{2}} = \frac{1}{2}s_1(t) + s_2(t)$$

$$\psi_2(t) = \frac{1}{2\sqrt{\frac{3}{2}}}s_1(t) + \frac{1}{\sqrt{\frac{3}{2}}}s_2(t) = \frac{1}{\sqrt{6}}s_1(t) + \frac{2}{\sqrt{6}}s_2(t)$$



$$\psi_3(t) = \frac{d_3(t)}{\sqrt{E_3}} \quad E_3 = \int_0^T d_3^2(t) dt = \int_0^T 0^2 dt = 0$$

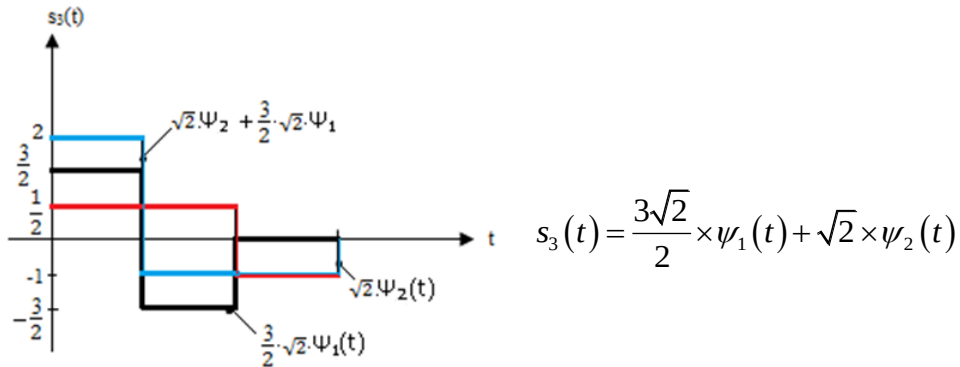
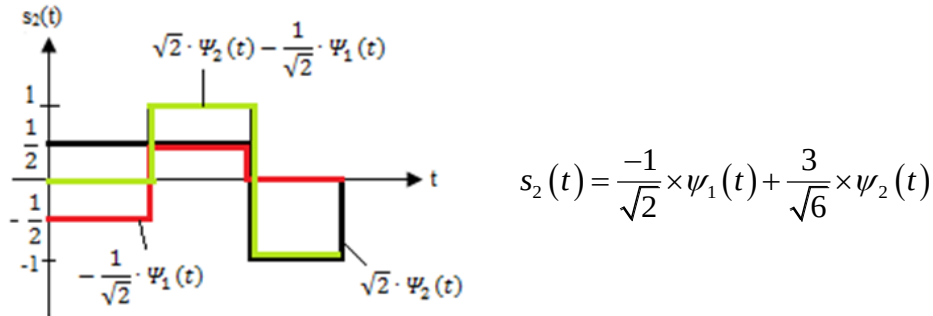
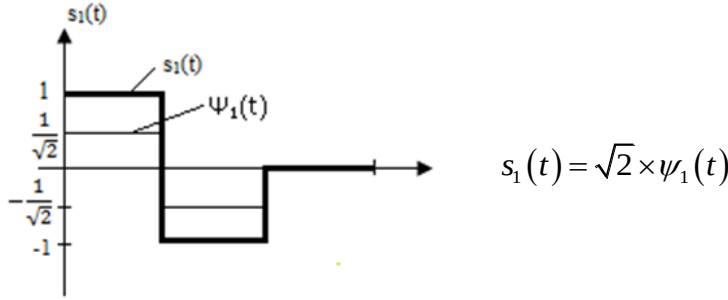
$$d_3(t) = s_3(t) - c_{13}\psi_1(t) - c_{23}\psi_2(t)$$

$$c_{13} = \int_0^T s_3(t)\psi_1(t) dt = \int_0^1 2 \times \frac{1}{\sqrt{2}} dt + \int_1^2 (-1) \times \left(-\frac{1}{\sqrt{2}}\right) dt + \int_2^3 (-1) \times 0 dt = \frac{2+1}{\sqrt{2}} = \frac{3}{\sqrt{2}}$$

$$c_{23} = \int_0^T s_3(t) \psi_2(t) dt = \int_0^1 2 \times \frac{1}{\sqrt{6}} dt + \int_1^2 (-1) \times \frac{1}{\sqrt{6}} dt + \int_2^3 (-1) \times \left(-\frac{2}{\sqrt{6}}\right) dt = \frac{2-1+2}{\sqrt{6}} = \frac{3}{\sqrt{6}}$$

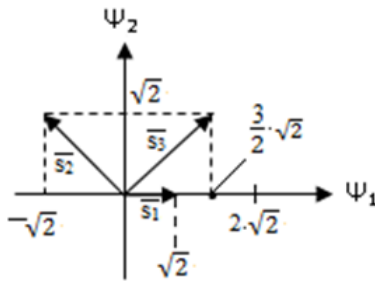
$$\begin{aligned} d_3(t) &= s_3(t) - \frac{3}{\sqrt{2}} \times \frac{s_1(t)}{\sqrt{2}} - \frac{3}{\sqrt{6}} \times \left(\frac{1}{2\sqrt{\frac{3}{2}}} \times s_1(t) + \frac{1}{\sqrt{\frac{3}{2}}} \times s_2(t) \right) \\ &= s_3(t) - \frac{3}{2} \times s_1(t) - \frac{1}{2} \times s_1(t) - s_2(t) = s_3(t) - 2 \times s_1(t) - s_2(t) = 0 \end{aligned}$$

In this case with only 2 basis function we can describe all 3 signals:



All signals can be expressed using their coordinates in a signal space defined by $\psi_1(t), \psi_2(t)$:

$$\begin{aligned} s_1(t) &= (\sqrt{2}, 0) \\ s_2(t) &= \left(-\frac{1}{\sqrt{2}}, \sqrt{2}\right) \\ s_3(t) &= \left(\frac{3\sqrt{2}}{2}, \sqrt{2}\right) \end{aligned}$$



Let check whether the found base is orthogonal:

$$\int_0^T \psi_1(t) \psi_2(t) dt = \int_0^1 \frac{1}{\sqrt{2}} \times \frac{1}{2\sqrt{2}} dt + \int_1^2 \left(-\frac{1}{\sqrt{2}}\right) \times \frac{1}{2\sqrt{2}} dt + \int_2^3 0 \times \left(-\frac{1}{\sqrt{2}}\right) dt = \frac{1}{4} - \frac{1}{4} + 0 = 0$$

■

Exercise 4.9:

Find the QPSK modulation basis functions.

Solution:

The QPSK signal space has 4 states $\Rightarrow$ each modulation symbol carries 2 bits of information.

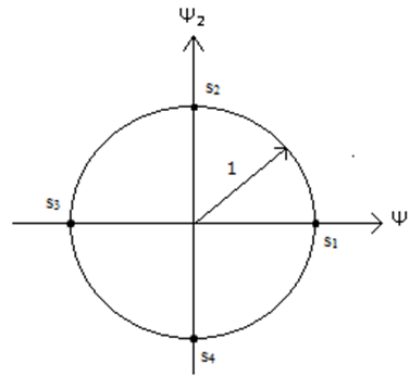
Signal alphabet: $s_i(t) = A \cos(\omega t + \varphi_i)$ $\varphi_i = 0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}$, $0 \leq t \leq T$ $A = \sqrt{\frac{2E}{T}}$

$$s_1(t) = \sqrt{\frac{2E}{T}} \cos(\omega t)$$

$$s_2(t) = \sqrt{\frac{2E}{T}} \cos\left(\omega t + \frac{\pi}{2}\right)$$

$$s_3(t) = \sqrt{\frac{2E}{T}} \cos(\omega t + \pi)$$

$$s_4(t) = \sqrt{\frac{2E}{T}} \cos\left(\omega t + \frac{3\pi}{2}\right)$$



$$\psi_1(t) = \frac{s_1(t)}{\sqrt{E_1}} = \frac{\sqrt{\frac{2E}{T}} \cos(\omega t)}{\sqrt{E}} = \sqrt{\frac{2}{T}} \cos(\omega t) \dots 0 < t < T, \text{ otherwise } 0$$

$$\begin{aligned}
c_{12} &= \int_0^T \sqrt{\frac{2E}{T}} \cos\left(\omega t + \frac{\pi}{2}\right) \sqrt{\frac{2}{T}} \cos(\omega t) dt = \frac{2\sqrt{E}}{T} \int_0^T \underbrace{\cos\left(\omega t + \frac{\pi}{2}\right)}_{=-\sin(\omega t)} \cos(\omega t) dt \\
&= -\frac{2\sqrt{E}}{T} \int_0^T \sin(\omega t) \cos(\omega t) dt = -\frac{\sqrt{E}}{T} \int_0^T \sin(2\omega t) dt = -\frac{\sqrt{E}}{T} \left[\frac{\cos(2\omega t)}{2\omega} \right]_0^T \\
&= -\frac{\sqrt{E}}{T} \left[\frac{\cos(2\omega T)}{2\omega} - \frac{1}{\omega} \right] = \frac{\sqrt{E}}{2\omega T} \left[1 - \underbrace{\cos(2\omega T)}_{=1} \right] = 0
\end{aligned}$$

Note: $\sin(2\omega t) = 2\sin(\omega t) \cos(\omega t)$

$$d_2(t) = s_2(t) - c_{12} \psi_1(t) = \sqrt{\frac{2E}{T}} \cos\left(\omega t + \frac{\pi}{2}\right)$$

$$\begin{aligned}
\psi_2(t) &= \frac{d_2(t)}{\sqrt{E}} = \frac{\sqrt{\frac{2E}{T}} \cos\left(\omega t + \frac{\pi}{2}\right)}{\sqrt{E}} = \sqrt{\frac{2}{T}} \cos\left(\omega t + \frac{\pi}{2}\right) \\
&= -\sqrt{\frac{2}{T}} \sin(\omega t) \dots \quad 0 \leq t \leq T, \text{ otherwise } 0
\end{aligned}$$

With the use of: $\int_{-\pi}^{\pi} \left(\underbrace{\cos(m\lambda)}_{\text{if } m=n} \right)^2 d\lambda = \int_{-\pi}^{\pi} \cos^2(\omega t) dt = \frac{T}{2} :$

$$\begin{aligned}
c_{13} &= \int_0^T s_3(t) \psi_1(t) dt = \int_0^T \sqrt{\frac{2E}{T}} \cos(\omega t + \pi) \sqrt{\frac{2}{T}} \cos(\omega t) dt \\
&= \frac{2\sqrt{E}}{T} \int_0^T \underbrace{\cos(\omega t + \pi)}_{=-\cos(\omega t)} \cos(\omega t) dt = -\frac{2\sqrt{E}}{T} \int_0^T \cos^2(\omega t) dt \\
&= -\frac{2\sqrt{E}}{T} \int_0^T \frac{1}{2} (1 + \cos(2\omega t)) dt = -\frac{2\sqrt{E}}{T} \frac{1}{2} [t]_0^T - \frac{2\sqrt{E}}{T} \frac{1}{2} \int_0^T \cos(2\omega t) dt \\
&= -\frac{2\sqrt{E}}{2T} T - \frac{2\sqrt{E}}{2T} \left[\frac{\sin(2\omega t)}{2\omega} \right]_0^T = -\sqrt{E} - \frac{\sqrt{E}}{T} \left[\underbrace{\frac{\sin \frac{4\pi T}{T}}{\frac{4\pi T}{T}}}_0 - C \right] = -\sqrt{E}
\end{aligned}$$

$$\begin{aligned}
c_{23} &= \int_0^T s_3(t) \psi_2(t) dt = \int_0^T \sqrt{\frac{2E}{T}} \cos(\omega t + \pi) \sqrt{\frac{2}{T}} \cos\left(\omega t + \frac{\pi}{2}\right) dt \\
&= \frac{2\sqrt{E}}{T} \int_0^T \cos(\omega t + \pi) \cos\left(\omega t + \frac{\pi}{2}\right) dt = \frac{2\sqrt{E}}{T} \int_0^T (-\cos(\omega t))(-\sin(\omega t)) dt \\
&= \frac{2\sqrt{E}}{T} \int_0^T \underbrace{\cos(\omega t)}_{u=\cos(\omega t)} \sin(\omega t) dt = \frac{2\sqrt{E}}{T} \int_1^{-1} \frac{\sin(\omega t)}{-\sin(\omega t)} du = 0
\end{aligned}$$

$$\frac{du}{dt} = -\sin(\omega t) \rightarrow dt = -\frac{du}{\sin(\omega t)} \quad t=0 \rightarrow u=1, T=t \rightarrow u=-1$$

$$\begin{aligned}
d_3(t) &= s_3(t) - c_{13}(t) \psi_1(t) - c_{23}(t) \psi_2(t) = \sqrt{\frac{2E}{T}} \cos(\omega t + \pi) \\
&+ \sqrt{E} \sqrt{\frac{2}{T}} \cos(\omega t) - 0 \times \psi_2(t) = -\sqrt{\frac{2E}{T}} \cos(\omega t) + \sqrt{\frac{2E}{T}} \cos(\omega t) = 0 \rightarrow \psi_3(t) = 0
\end{aligned}$$

$$\begin{aligned}
c_{14} &= \int_0^T s_4(t) \psi_1(t) dt = \int_0^T \sqrt{\frac{2E}{T}} \underbrace{\cos\left(\omega t + \frac{3}{2}\pi\right)}_{+\sin(\omega t)} \sqrt{\frac{2}{T}} \cos(\omega t) dt \\
&= \frac{2\sqrt{E}}{T} \int_0^T \sin(\omega t) \cos(\omega t) dt = 0
\end{aligned}$$

$$\begin{aligned}
c_{24} &= \int_0^T s_4(t) \psi_2(t) dt = \int_0^T \sqrt{\frac{2E}{T}} \underbrace{\cos\left(\omega t + \frac{3}{2}\pi\right)}_{+\sin(\omega t)} \sqrt{\frac{2}{T}} \underbrace{\cos\left(\omega t + \frac{\pi}{2}\right)}_{-\sin(\omega t)} dt \\
&= -\frac{2\sqrt{E}}{T} \int_0^T \sin^2(\omega t) dt = -\frac{2\sqrt{E}}{T} \int_0^T \frac{1}{2} (1 - \cos(2\omega t)) dt = -\frac{2\sqrt{E}}{2T} [t]_0^T - \frac{2\sqrt{E}}{2T} \int_0^T \cos(2\omega t) dt \\
&= -\sqrt{E} - \frac{\sqrt{2}}{T} \left[\underbrace{\frac{\sin \frac{4\pi t}{T}}{4\pi}}_0 \right]_0^T = -\sqrt{E}
\end{aligned}$$

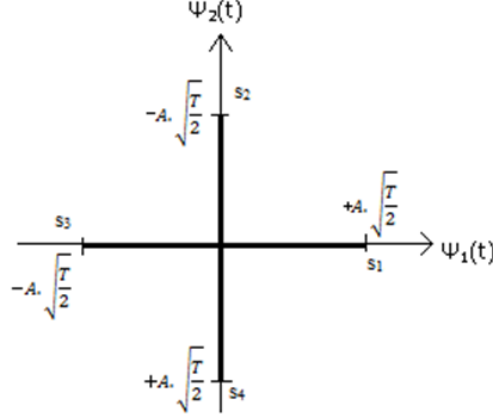
$$\int_{-\pi}^{\pi} \sin^2(\omega t) dt = \frac{T}{2} \quad \omega = \frac{2\pi}{T} \rightarrow \pi = \frac{T}{2} \quad \text{for } \omega=1 \quad \sin^2(x) = \frac{1}{2}(1 - \cos(2x))$$

$$c_{34} = \int_0^T s_4(t) \psi_3(t) dt = 0$$

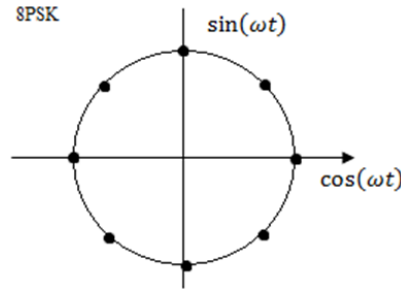
$$\begin{aligned}
d_4(t) &= s_4(t) - c_{14}(t) \psi_1(t) - c_{24}(t) \psi_2(t) - c_{34}(t) \psi_3(t) = s_4(t) - c_{24}(t) \psi_2(t) \\
&= \sqrt{\frac{2E}{T}} \underbrace{\cos\left(\omega t + \frac{3}{2}\pi\right)}_{+\sin(\omega t)} + \sqrt{E} \sqrt{\frac{2}{T}} \underbrace{\cos\left(\omega t + \frac{\pi}{2}\right)}_{-\sin(\omega t)} = \sqrt{\frac{2E}{T}} (\sin(\omega t) - \sin(\omega t)) = 0
\end{aligned}$$

From the previous follows that only 2 basis functions are sufficient to express QPSK modulation in the signal space:

$$\psi_1(t) = \sqrt{\frac{2}{T}} \cos(\omega t) \quad \psi_2(t) = -\sqrt{\frac{2}{T}} \sin(\omega t)$$



Note: For a MPSK with M states, the base of the signal space consists of 2 functions $\sin$ and $\cos$ that are orthogonal to each other:



Test whether the found basis is orthogonal:

$$\begin{aligned} \int_0^T \psi_1(t) \psi_2(t) dt &= \int_0^T \sqrt{\frac{2}{T}} \cos(\omega t) \left(-\sqrt{\frac{2}{T}} \sin(\omega t) \right) dt = -\frac{2}{T} \int_0^T \cos(\omega t) \sin(\omega t) dt \\ &= -\frac{2}{T} \times \frac{1}{2} \int_0^T (\sin(2\omega t) - \sin(0)) dt = -\frac{1}{T} \int_0^T \sin(2\omega t) dt \\ &= -\frac{1}{T} \left[-\frac{\cos(4\omega t)}{4\omega} \right]_0^T = \frac{1}{T} \left[\frac{\cos\left(\frac{8\pi T}{T}\right)}{\frac{8\pi}{T}} - \frac{\cos(0)}{\frac{8\pi}{T}} \right] = \frac{1}{T} \left[\frac{1}{\frac{8\pi}{T}} - \frac{1}{\frac{8\pi}{T}} \right] = 0 \end{aligned}$$

Signal coordinates:

$$s_1(t) = \left(A\sqrt{\frac{T}{2}}, 0 \right) \quad s_2(t) = \left(0, A\sqrt{\frac{T}{2}} \right) \quad s_3(t) = \left(-A\sqrt{\frac{T}{2}}, 0 \right) \quad s_4(t) = \left(0, -A\sqrt{\frac{T}{2}} \right)$$

■

Lab 4.1: MATLAB exercises - Signals and signal processing

Implement the following exercises in separate .m-files. Use the provided `lab4.m` file as a unit test for them.

MATLAB Exercise 4.1.1: Comparing data with tolerance.

Implement a public function with the prototype: `function [ E, H, D ] = equals( A, B, tol )` that will compare the matrices **A** and **B** for equality and optionally calculate the Hamming and absolute distance between the matrices. The third input argument is optional, and if it's not specified, the function will behave the same as the `isequal()` built-in. If specified, the `tol` defines the tolerance with which we assume the matrices are "close enough", or to be considered "practically equal". That means that if the distance between matrices is less than `tol`, the function returns true. For the purpose of this exercise we define the distance as the sum of absolutes of differences of all the elements of **A** and **B**. Several different metrics, called norms, could also be used.

Furthermore, regardless of the number of input parameters, the function must return an optional second and third output parameter: the Hamming distance between matrices and the absolute distance defined above. The Hamming distance was defined in previous chapters for binary matrices and can be easily extended as the number of non-equal elements (Assume true binary equality for this exercise).

The function must not produce an error if the matrices are of different sizes and must provide sensible return values for all combinations input / output parameter options.

MATLAB Exercise 4.1.2: Unitary Discrete Fourier Transform (DFT) matrix.

Implement a public function with the prototype: `function W = ftn( n )` that generates a unitary DFT matrix of size $n \times n$. The DFT matrix is defined as follows:

$$W = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \dots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

where: $\omega = e^{\left(\frac{-2\pi i}{N}\right)}$

Hint: generate a matrix of exponents first. Do it without using any loops, use MATLAB vector features (vector multiplication) instead to create a matrix of exponents first:

$$C = \begin{pmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 2 & 3 & \dots & N-1 \\ 0 & 2 & 4 & 6 & \dots & 2(N-1) \\ 0 & 3 & 6 & 9 & \dots & 3(N-1) \\ \dots & & & & & \\ 0 & N-1 & 2(N-1) & 3(N-1) & \dots & (N-1)(N-1) \end{pmatrix}$$

Compare transforming a complex-valued signal stored in a column vector using the built-in `fft()` function and transformation using matrix multiplication (You have to multiply the output of the `fft()` function with a scale factor). Use the `equals` function from Exercise 1 for comparison with tolerance and compare its results with comparison using `isequal()`. Test both left and right matrix-vector

multiplication formulation. Observe the differences between transpose and Hermitian transpose. Is the DFT matrix Hermitian or just symmetric?

MATLAB Exercise 4.1.3: Visualizing complex signals

Write a public function given the prototype: `function h = plotComplex( f, type, Signal, x )` that plots all four basic plots of a complex signal into one figure consisting of a grid of 2x2 plots: First the real part, second the imaginary part, third the magnitude, and fourth the phase. The first parameter should be an integer number specifying the figure number, the second parameter may take on one of the character array values 'plot' or 'stem' to specify continuous or discrete plot, the `Signal` is a vector (row or column) of the signals y-values and the last parameter `x` is an optional vector of corresponding x values. If not specified, the indices of the signal array should be used (starting at 1). The function should return the figure handle. Use function handle operator: `@` for switching between calling of `plot()` or `stem()` built-ins in your implementation. Figure 4.13 show the desired outputs for a continuous x-axis plot:

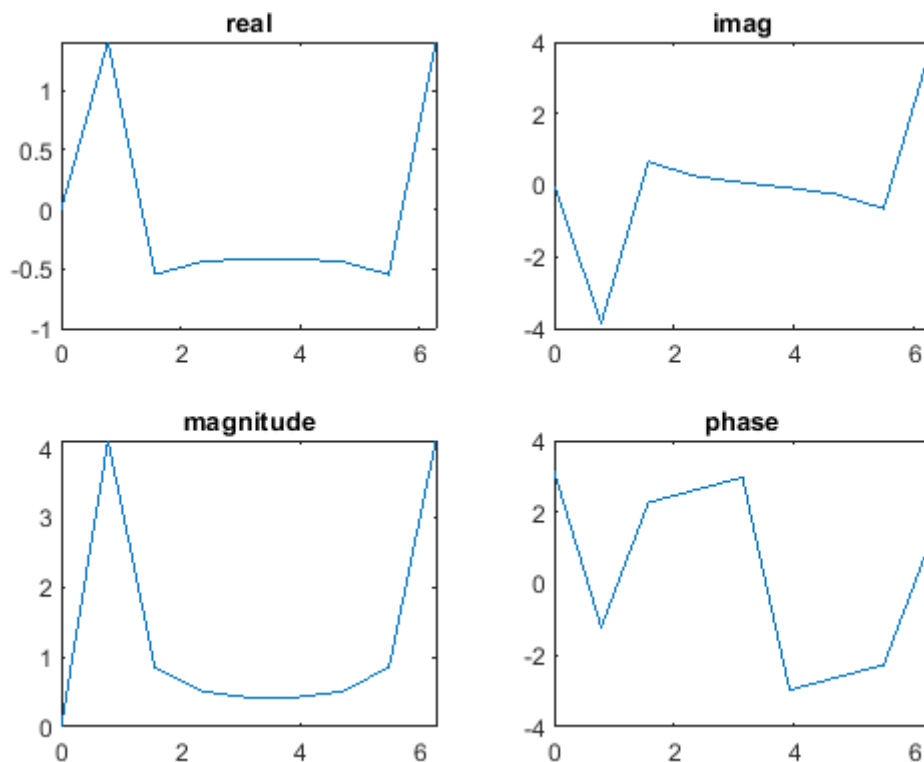


Fig. 4.13: Example plot for a continuous display with implicit x-axis values.

Fig. 4.14 then shows an example of discrete-time discrete-x-axis plot. Note that in the basic signal-processing context, the x-axis may in some setting represent time, while in other settings it may represent frequency.

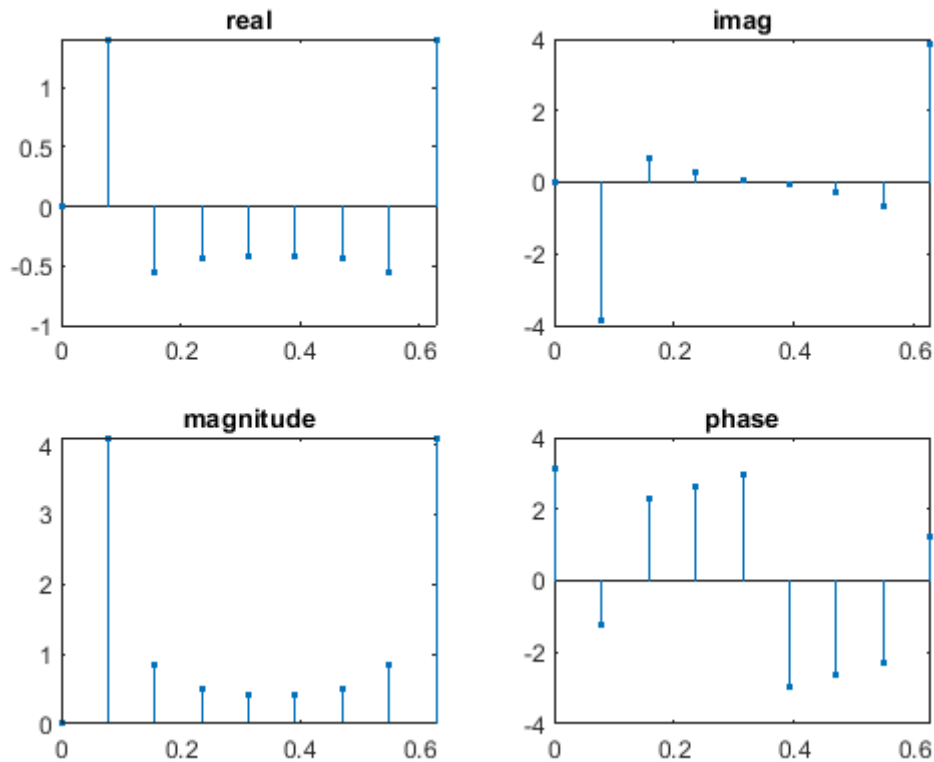


Fig. 4.14: Example plot for a discrete display with user supplied x-axis values.

Use the MATLAB code snippet shown below to test your implementation: open 4 figure windows: one for the sine signal, one for the cosine, the third one for the sine spectrum, the last one for the cosine spectrum.

MATLAB Exercise 4.1.4: Basic Fourier analysis - introduction

Let one period of a pure sine and cosine signals be given by the samples of their first period: $\langle 0, 2\pi \rangle$. Let the desired number of samples per period $N = 8$. Someone used the definition of a time samples vector for functions evaluation given by the following code snippet:

```
s = ( 2 * pi ) / N ;
t = [ 0 : s : 2 * pi ] ;
fsin = sin( t ) ;
fcos = cos( t ) ;
FSIN = fft( fsin ) ;
FCOS = fft( fcos ) ;
```

After evaluating the function and printing using `plot()`, the approximations of the signals looks fine:

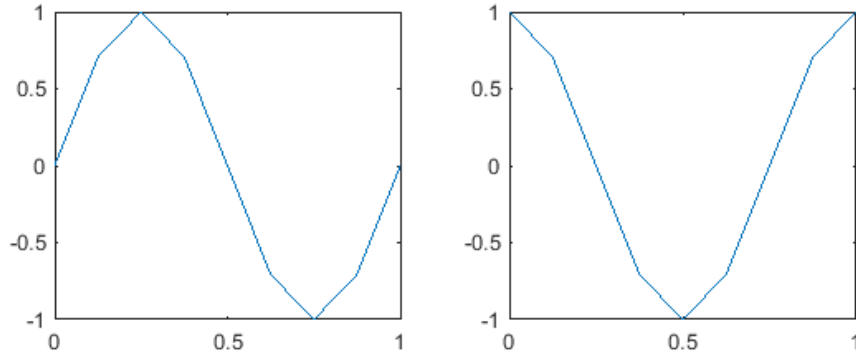


Fig. 4.15: The approximations of a sine and cosine signal for $N = 8$ samples per period.

Note the artefacts of the approximation: we intentionally use only a small number of samples, and the `plot()` function in MATLAB performs linear interpolation when rendering the pixels of the image.

From the well known theoretical Fourier analysis, we know that the spectra of continuous sine and cosine signals consist of just two Dirac impulses, symmetric and real for cosine, antisymmetric and imaginary for sine:

$$\begin{aligned}\cos(2\pi f_0 t) &\xleftrightarrow{FT} \frac{1}{2} \delta(f - f_0) + \frac{1}{2} \delta(f + f_0) \\ \sin(2\pi f_0 t) &\xleftrightarrow{FT} \frac{1}{2j} \delta(f - f_0) - \frac{1}{2j} \delta(f + f_0)\end{aligned}\tag{4.17}$$

Fig. 4.16 provides a simple and intuitive visualization of these equations for a normalized frequency $f_0 = 1$.

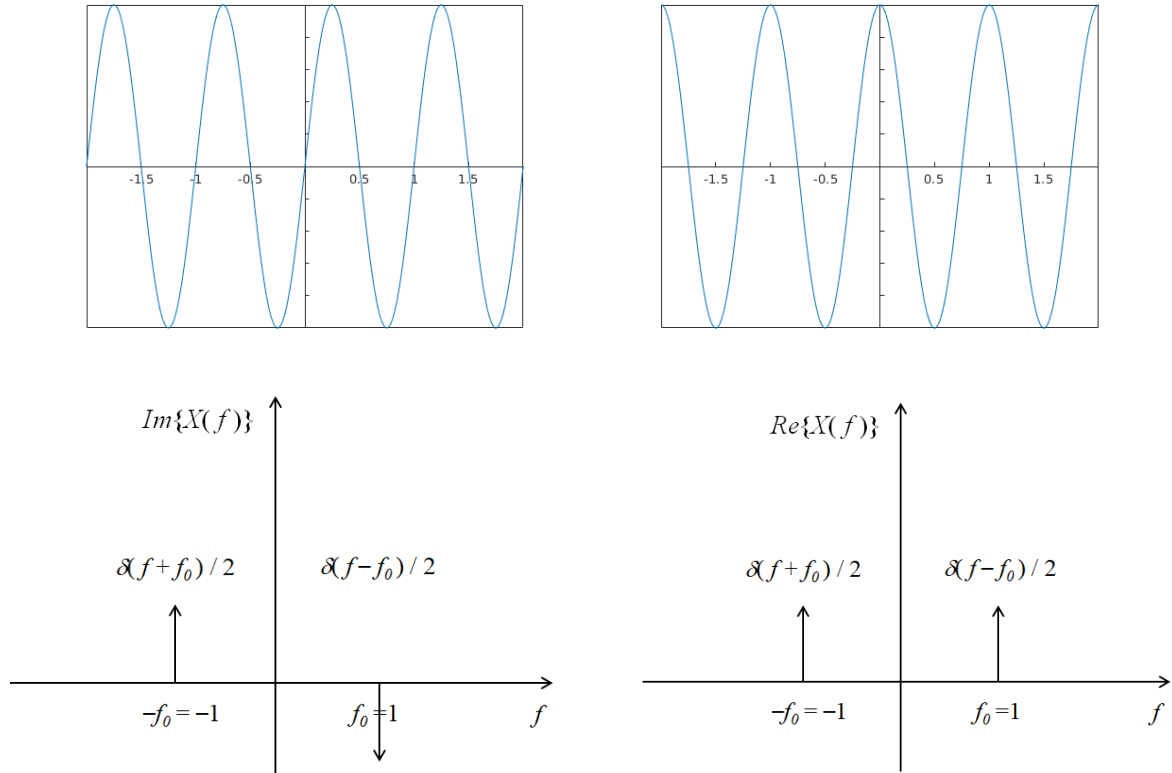


Fig. 4.16: The continuous-time domain of normalized sine and cosine (up) and their spectra (down).

Let us now investigate, what happens if we try to obtain these results in MATLAB by running the code snippet above: after finding the spectrum using the `fft()` built-in, the resulting `stem()` plots of the spectra show surprising artefacts, as demonstrated by the following figures:

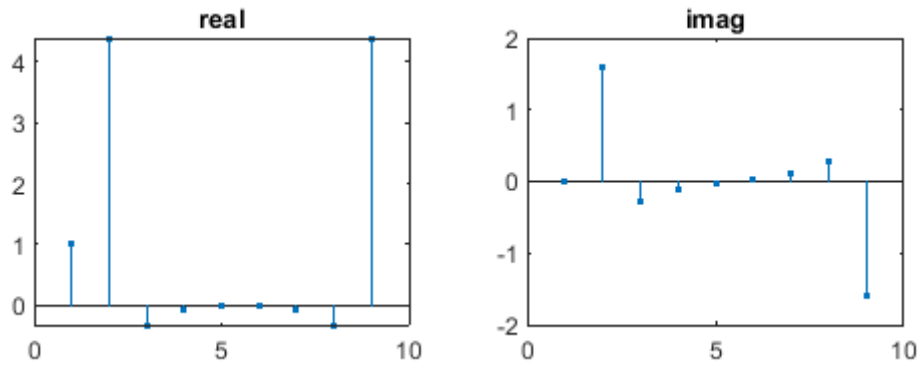


Fig. 4.17: Real (left) and imaginary (right) part of the spectrum of a cosine signal.

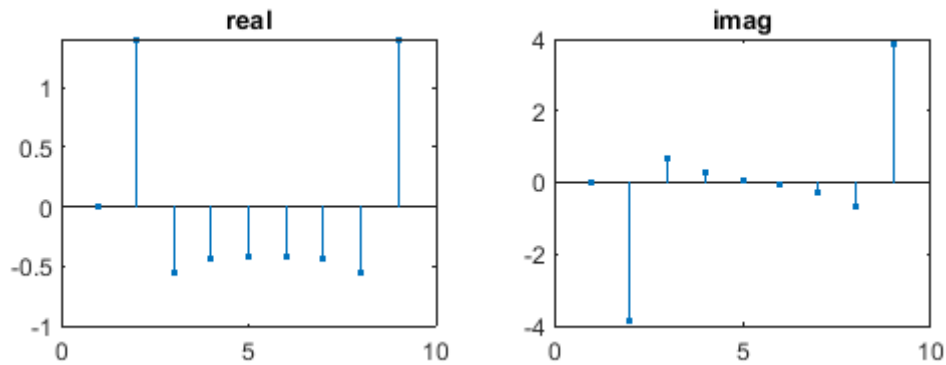


Fig. 4.18: Real (left) and imaginary (right) part of the spectrum of a sine signal.

While some features of the actually calculated plots manifest similarities with the theoretical values, others are completely different: the imaginary part is not zero for the cosine signal, and real part not zero for the sine. Comparing to the theoretical figures, the spectral lines are at different positions, extra non-zero values are present and also the amplitudes at the Dirac pulses seem to be off. Furthermore, the labeling at the x-axis seems to not to correspond to any frequency at all.

A careful reader already knows what went wrong in this example: by defining the time vector on line 2 of the code snippet from 0 to 2π , we unintentionally included the last 9th sample that actually belongs to the next period. So we accidentally took 9 samples instead of 8. This is easy to see from the `stem()` output in figures 4.17 and 4.18, but very hard to identify from the `plot()` in Fig. 4.15. The solution is simple: exclude the last sample from the time samples vector and re-evaluate. For a small N , if we plot the sine function using `stem()`, we may be able to see the subtle difference directly:

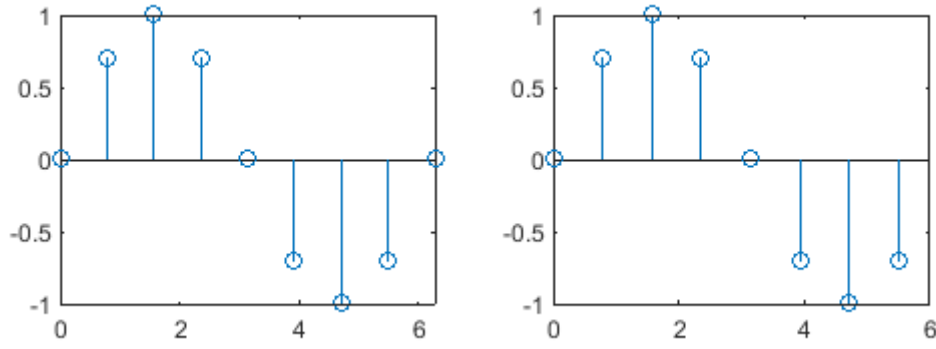


Fig. 4.19: Stem plot of one period of the sine function with extra last sample (left), and correct (right).

If we now calculate the samples of the spectrum again, we get a superficially similar, but in fact much different plot. Fig. 4.20 shows the result for the sine function.

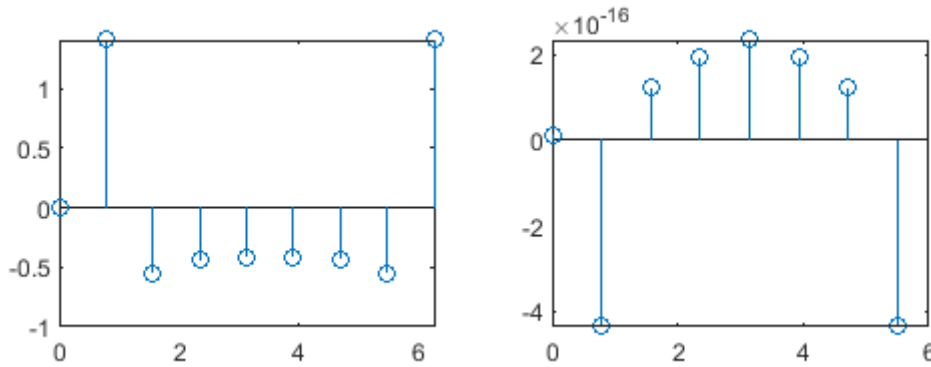


Fig. 4.20: Stem plot of the real part of a sine function spectrum: with extra last sample (left), and “almost” correct (right).

As we can see, the spectrum is still not what we expected: the real part was supposed to be all zero, which is not the case at all, if an extra sample is taken, while being extremely close to zero for correct sampling. It is just the auto scaling feature performed an automatic zoom in on the y-axis. If we notice the scaling factor in Fig. 4.20 right: 10^{-16} and remember to compare the results of floating point calculations with a tolerance, we can safely round the whole real part of the spectrum to zero. This still leaves us with a little problem: the sine spectrum in Fig. 4.18 right is still not positioned properly on the x-axis. No negative frequencies are shown in the figure. This is because what we’re actually using on a computer is the DFT: the Discrete Fourier Transform that transforms a vector of samples of one period of a discrete-time periodic signal, to a vector of samples of a discrete-frequency periodic spectrum. The defining formulas of direct and inverse DFT :

$$\begin{aligned}
 X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{\left(-\frac{2\pi j}{N} kn\right)} \\
 x_n &= \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{\left(\frac{2\pi j}{N} kn\right)} \\
 &\text{for } n, k \in \{0, 1, \dots, N-1\}
 \end{aligned}
 \tag{4.18}$$

Deal only with integer indices n and k of elements of time-domain and frequency-domain vectors $\mathbf{x}$ and $\mathbf{X}$. This is clearly visible in all the figures, since the x-axes labels are allways just the indices of frequency bins, not the actual frequencies. Furthermore, because of the DFT properties, the bins with index larger than $N / 2$ actually represent negative frequencies. This comes from the periodicity of the DFT spectra. The situation is, however, a little bit more complicated: the proper tool for a continuous-time periodic signal is the *Fourier series*, which transforms a continuous-time periodic signal to a discrete-frequency non-periodic spectrum. However, in a computer, any continous time-or-frequency signal must be in fact approximated by a discrete time-and-frequency signal, leaving the DFT as the only tool available. We must take advantage of known theoretical analogies between various forms of the Fourier analysis for different settings. The details are out of the scope of this book, but the short solution for approximating the theoretical plot of a continuous-time sine spectrum shown in Fig 4.16 and its discrete-frequency nonperiodic spectrum equivalent is to manullay assign the proper frequency bins to the spectrum samples by shifting them by $N/2$. Luckily this is done easily in MATLAB, by dividing the `fft()` output vector in halves and schwitching them. This is shownm in Fig. 4.21, this time for a $\sin(2\pi f_0 t)$ signal wher $f_0 = 1$ Hz. However the x-axis still shows the indices of the frequency bins, not actual frequencies. These must be calculated by the programmer.

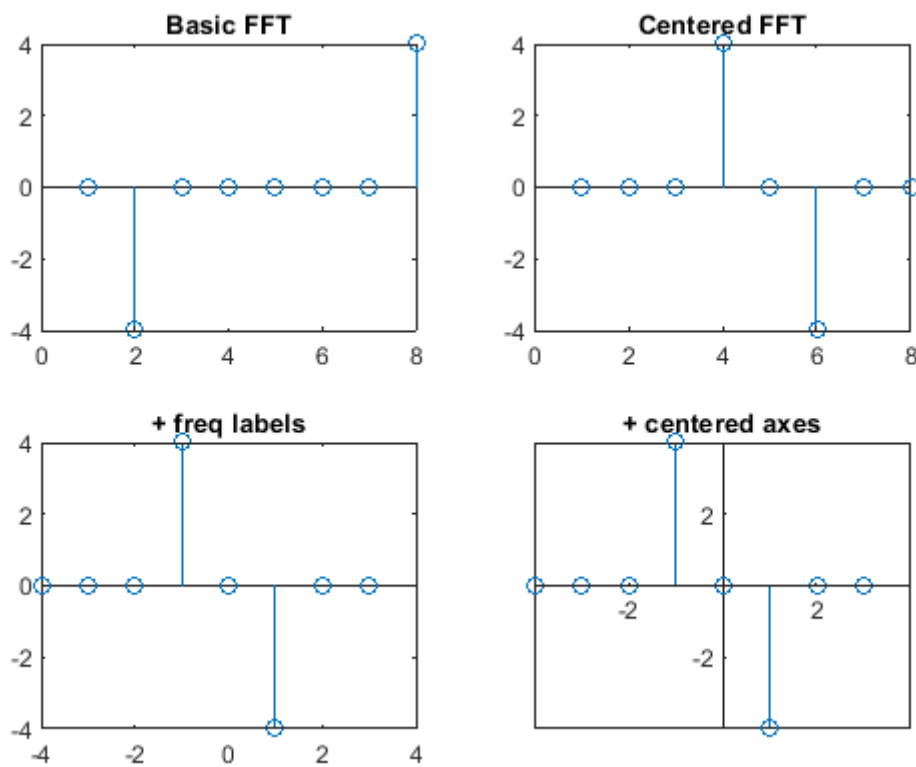


Fig. 4.21: Four visualization of the sine signal imaginary spectrum: left up: standard `fft()` output, right up: `fft()` with negative frequencies shifted, left down: added correct x-axis labels defining actual frequencies, right down: axes centered to origin.

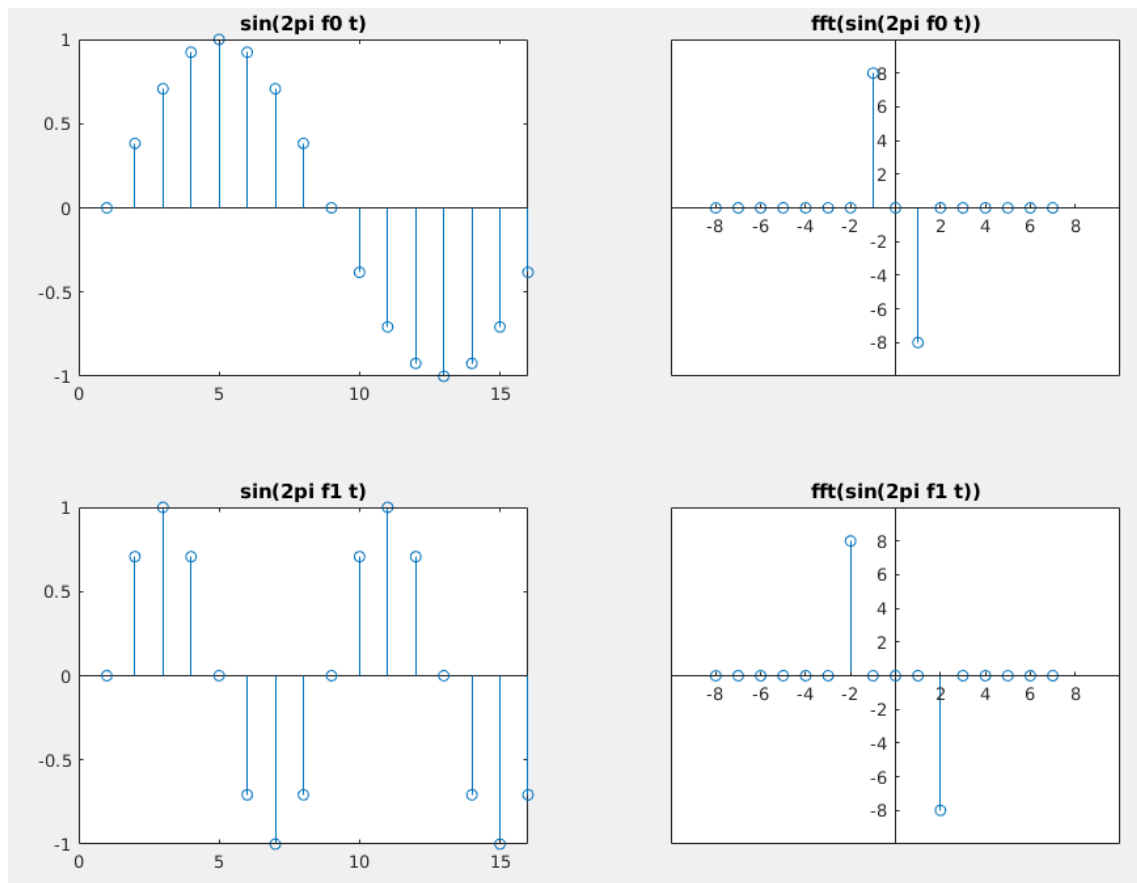


Fig. 4.22: The expected effect of doubling the sine signal frequency on the spectrum from $f_0 = 1$ to $f_1 = 2$, while processing the same amount of samples. Note the number of samples per one period: N is equal to 16 for the lower frequency sine (top), while $N = 8$ for the higher frequency (bottom).

The task of this exercise is to implement an *m*-file that replicates the demonstration described above, and adds also the correct DFT analysis of the cosine samples.

MATLAB Exercise 4.1.4: Basic Fourier analysis – assignments:

MATLAB Exercise 4.1.4a:

Implement a function given the prototype: `function Spectr = bft( Signal, N, unitary, tol )` that calculates an enhanced Fourier transform: it should really be just a wrapper around the `fft()` built-in that introduces the following extra functionality: it should automatically shift the `fft()` result to shift the zero frequency bin to the center. It should check that the number of samples N is even and produce an error if it's not. It should take an optional third boolean parameter that specifies whether or not should the spectrum be normalized to be unitary (automatically defaulting to false), it should take an optional fourth parameter: the tolerance value. If all the samples of the real or imaginary part of the spectrum are smaller in absolute value than the tolerance, this part of the spectrum should not be included in the output (so that for a cosine input produces a pure real spectrum, while a sine input produces a pure imaginary one).

MATLAB Exercise 4.1.4b:

Implement a correction to the code snippet from the beginning of this exercise to include only the proper number of samples. Also define the period $T = 1$ [s], and modify the observed signals to be normalized: $\sin(2\pi f_0 t)$ and $\cos(2\pi f_0 t)$. Properly calculate the value of $f_0 = 1/T$, sampling time $t_s = T/N$, and frequency axis labels and plot the resulting discrete-time signal and discrete-frequency spectrum into two separate figures: one for the sine, and one for the cosine signal. The resulting figures should look like the figure Fig. 4.23 (for a non-unitary Fourier transform).

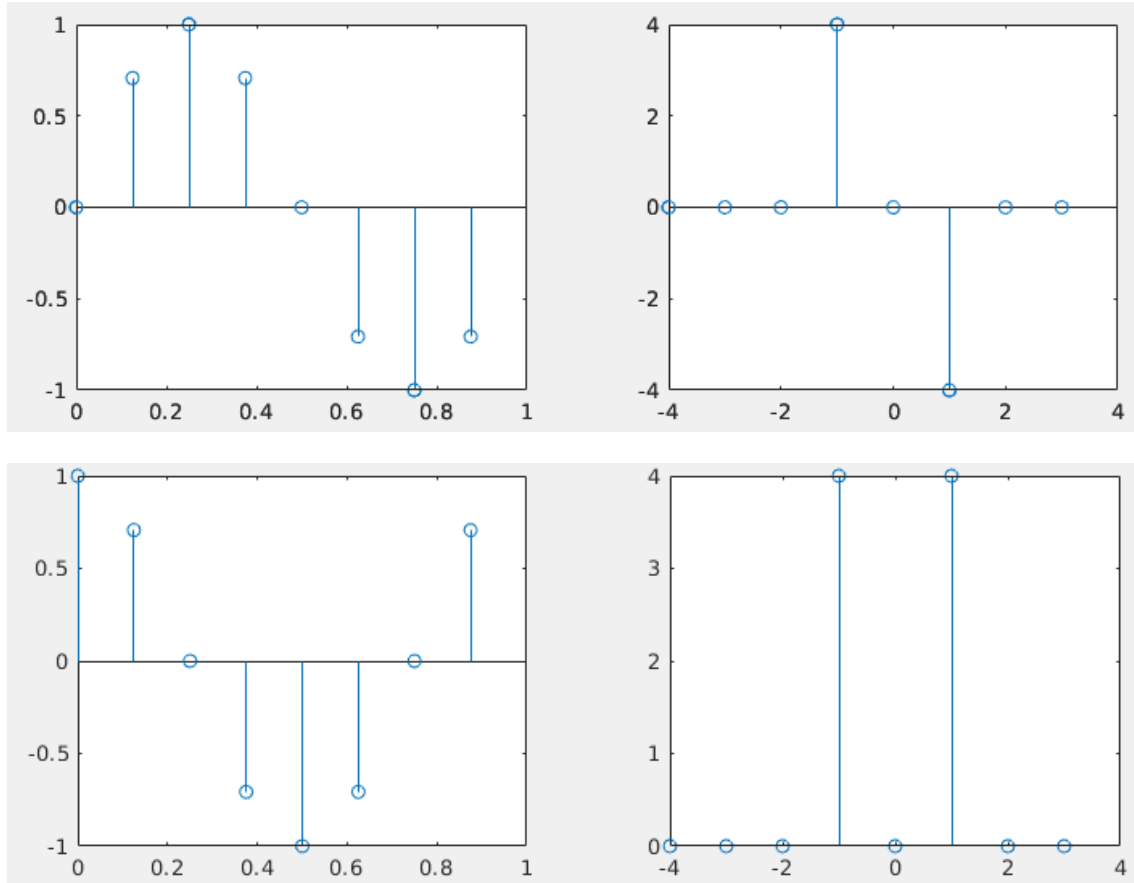


Fig. 4.23: Discrete-time sine signal and its discrete frequency imaginary spectrum (top), normalized cosine and its real spectrum (bottom).

MATLAB Exercise 4.1.4c:

Copy the source code from the previous exercise and perform the Fourier transform for two periods of the signal, while maintaining $N = 8$ and increasing f_1 to $2f_0$. Use a unitary Fourier transform this time. Fig. 4.24 shows the desired resulting figures:

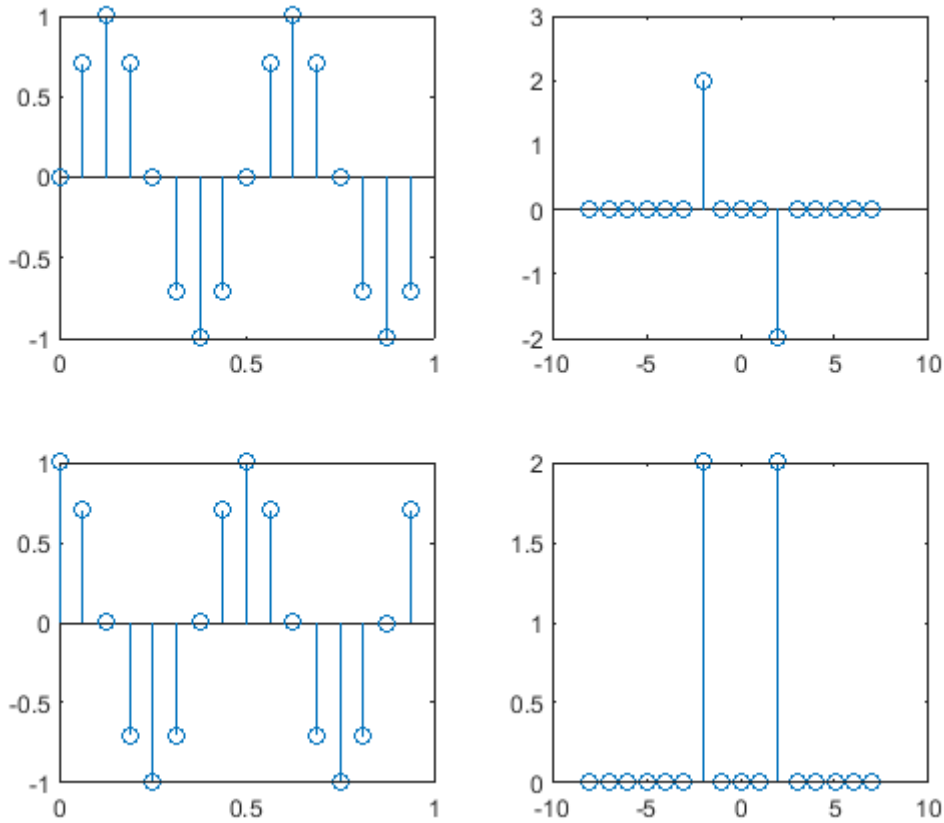


Fig. 4.24: Discrete-time sine signal and its discrete frequency imaginary unitary spectrum (top), cosine and its real unitary spectrum (bottom).
Two periods were transformed and frequency doubled.

MATLAB Exercise 4.1.5: Signal energy and power

We may recall the important definitions of energy and power from the theoretical part of this chapter, the formulas seem simple, for a power signal, the definition of a mean normalized power is given by:

$$P_x = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} x^2(t) dt \quad [W] \quad (4.19)$$

while for an energetic signal, the mean normalized energy definition is:

$$E_x = \int_{-\infty}^{\infty} x^2(t) dt \quad [J] \quad (4.20)$$

Both these definitions deal with real-valued signal only, which is insufficient for a practical communications systems analysis. Furthermore, they apply for a continuous-time signal that is unavailable in the context of Digital Signal Processing on the computer. The transformation of these equations to their discrete-time equivalents should be relatively straightforward:

Several formulations of the power definition may be found in literature. They vary in few details: if we are dealing with a signal symmetrical around the y-axis, we use the first of the forms shown below (left).

For a complex signal we always need to use the absolute value, or better, a multiplication with the signals complex conjugate, as shown in the middle. This definition is perfectly compatible with real signals, and is in fact preferable, because it explicitly acknowledges the fact that we are often dealing with complex signals. Finally the last definition on the right deals with a signal uncentered around the y-axis. (The symbol N is obvious equal to $2N' + 1$):

$$Px = \frac{1}{2N'+1} \sum_{n=-N'}^{N'} |x_n|^2 = \frac{1}{2N'+1} \sum_{n=-N'}^{N'} x_n x_n^* \approx \frac{1}{N} \sum_{n=0}^{N-1} x_n x_n^* \quad (4.21)$$

Very similar situation applies in the case of energetic signals:

$$Ex = \sum_{n=-\infty}^{\infty} |x_n|^2 = \sum_{n=-\infty}^{\infty} x_n x_n^* \approx \sum_{n=0}^{N-1} x_n x_n^* \quad (4.22)$$

Similarly, the very usefull Parseval theorem was defined before in terms of Fourier series and Fourier transform. Since we already know the DFT is the only true option available in a computer, we should also reformulate the Parseval theorem in terms of the DFT:

$$\sum_{n=0}^{N-1} |x_n|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X_k|^2 \approx \sum_{n=0}^{N-1} |x_n|^2 = \sum_{k=0}^{N-1} |X_k|^2 \quad (4.23)$$

This time we use the classic DFT on the left, or the rescaled unitary DFT on the right. The diffences are subte, but important.

MATLAB Exercise 4.1.5a:

Implement a function given the prototype: `function [ E, P ] = ep( Signal )` that calculates the energy and power of a discrete complex-valued Signal. If the signal is a vector (regardles if row or column), the results are scalars. If the Signal is a matrix, the function should calculate energy and power for each column separately, so the resulting outputs are row vectors.

MATLAB Exercise 4.1.5b:

Write a test script, where you generate one period of a cosine signal $A \cos(2\pi f_0 t)$ sampled by $N = 8$ samples per period and calculate its energy and power in time and also frequency domain using the function `ep()`. Use both forms of the DFT: unitary and standard. Verify the validity of the Parseval theorem. Run the same script again with increased $N = 16$. If ewerything is done correctly, the resulting power should stay the same, regardless if calculated in time or frequency domain, and should not change with N . Change the frequency to $f_1 = 2f_0$ and calculate the energy and power of the signal over two periods. Will the power stay the same?

MATLAB Exercise 4.1.6: Convolution and filtering

Matrix multiplication is an important tool in signal processing applications, especially in discrete-time case. Every time a signal is processed using a Finite Impulse Response (FIR) digital filter, the process of convolution of the signal $\mathbf{t}$ with the filter impulse response $\mathbf{h}$ may be described using multiplication by a matrix representing the filter. In practical system, it is not unreasonable to expect a block orientation of the calculation, thus $\mathbf{t}$ and $\mathbf{h}$ will both be finite-length vectors. Convolution is a well-known operation, for an input vector of length N and filter of length ν , the resulting vector $\mathbf{r}$ will be $N + \nu - 1$ sample long and its samples relate to the input and filter samples by the system of equations:

$$r_n = \sum_{m=1}^{\nu} h_m \cdot t_{n-m+1}, \quad n = 1, \dots, N + \nu - 1 \quad (4.24)$$

Usually the n index represent time sample while the m index represent a delay, relative to a given n . All samples of the input signal for the time index outside of the range 1 to N are considered zero, similarly to the taps of the filter outside of the range 1 to ν . We can the rewrite (4.24) to show that it actually describes a whole system of equations:

$$\begin{aligned} r_1 &= h_1 \cdot t_1 \\ r_2 &= h_1 \cdot t_2 + h_2 \cdot t_1 \\ r_3 &= h_1 \cdot t_3 + h_2 \cdot t_2 + h_3 \cdot t_1 \\ &\dots \\ r_{N+\nu-1} &= h_{\nu} \cdot t_N \end{aligned} \quad (4.25)$$

It is now just a matter of representing a system of equations by a matrix as shown in the example below. From the perspective of modelling a cascade of blocks in a system, we can focus on the important fact: the process of convolution shown by a system of equations (4.25) can conceptually be replaced by a much simpler matrix multiplication equation:

$$\mathbf{r}^{((N+\nu-1) \times 1)} = \mathbf{H}^{((N+\nu-1) \times N)} \cdot \mathbf{t}^{(N \times 1)} \quad (4.26)$$

If we omit the explicit sizes indication (which we often do), the equation representing a FIR filter operation really starts to look simple:

$$\mathbf{r} = \mathbf{H}\mathbf{t} \quad (4.27)$$

Furthermore, if a cascade of FIR filters is used in a more complex system, the individual filter matrices $\mathbf{H}_1$ and $\mathbf{H}_2$ can be simply combined together using the matrix multiplication associativity rule:

$$\mathbf{r} = \mathbf{H}_2(\mathbf{H}_1\mathbf{t}) = (\mathbf{H}_2\mathbf{H}_1)\mathbf{t} = \mathbf{H}_c\mathbf{t} \quad (4.28)$$

Example 4.1: Linear convolution matrix for a FIR filter with length $v = 4$ and input block size $N = 8$.

$$\mathbf{r} = \mathbf{H}\mathbf{t} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \\ r_8 \\ r_9 \\ r_{10} \\ r_{11} \end{bmatrix} = \begin{bmatrix} h_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ h_2 & h_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ h_3 & h_2 & h_1 & 0 & 0 & 0 & 0 & 0 \\ h_4 & h_3 & h_2 & h_1 & 0 & 0 & 0 & 0 \\ 0 & h_4 & h_3 & h_2 & h_1 & 0 & 0 & 0 \\ 0 & 0 & h_4 & h_3 & h_2 & h_1 & 0 & 0 \\ 0 & 0 & 0 & h_4 & h_3 & h_2 & h_1 & 0 \\ 0 & 0 & 0 & 0 & h_4 & h_3 & h_2 & h_1 \\ 0 & 0 & 0 & 0 & 0 & h_4 & h_3 & h_2 \\ 0 & 0 & 0 & 0 & 0 & 0 & h_4 & h_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_4 \end{bmatrix} \times \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \end{bmatrix}$$

On the other hand, these matrices can become quite large. For instance if $\mathbf{t}$ represents samples of a Wi-Fi frame 1500 bytes long, modulated by a binary modulation scheme, and oversampled so that each symbol is represented by 4 samples, the total length of the sampled frame will be $1500 \times 8 \times 4 = 48000$ samples. If such a signal is processed by a 32-tap FIR filter the size of the $\mathbf{H}$ matrix will be $48031 \times 32 = 1.536 \times 10^6$.

Another important use of the segmentation of a matrix into a grid of submatrices and dividing the matrix multiplication to smaller submatrices operations is whenever multi-channel discrete-time system is modelled. If we model the multiple parallel channels as columns of an input matrix $\mathbf{T}$, we can again describe a multichannel filter by a simple matrix equation:

$$\mathbf{R}^{(L \times M)} = \mathbf{H}^{(L \times N)} \mathbf{T}^{(N \times M)} =$$

$$= (\mathbf{r}_1 \quad \mathbf{r}_2 \quad \dots \quad \mathbf{r}_M) = \mathbf{H} \cdot (\mathbf{t}_1 \quad \mathbf{t}_2 \quad \dots \quad \mathbf{t}_M) = \mathbf{H} \cdot \left(\begin{pmatrix} t_{11} \\ t_{21} \\ \dots \\ t_{N1} \end{pmatrix} \begin{pmatrix} t_{12} \\ t_{22} \\ \dots \\ t_{N2} \end{pmatrix} \dots \begin{pmatrix} t_{1M} \\ t_{2M} \\ \dots \\ t_{NM} \end{pmatrix} \right)$$

where each of the output vectors $\mathbf{r}_i$ is a column vector L samples long and each columns vector $\mathbf{t}_i$ represents one signal among the M signals processed in parallel

MATLAB Exercise 4.1.6a:

Implement a function given the prototype: `function H = convmtx( h, N )` that generates and returns a convolution matrix for a given impulse response vector $\mathbf{h}$ and input vector size N . The function should behave in the following way: if $\mathbf{h}$ is a column vector, then the matrix $\mathbf{H}$ should be directly usable in a left multiplication:

$$\mathbf{r} = \mathbf{H}\mathbf{t}$$

if $\mathbf{h}$ is a row vector, the generated convolution matrix must be transposed to be directly usable in a right multiplication:

$$\mathbf{r} = \mathbf{t}\mathbf{H}$$

If **h** is a matrix or scalar, the function should throw an error.

MATLAB Exercise 4.1.6b:

Test your implementation of the `convMtx()` function with the following inputs: **h** = [3 2 1 0 -1 1], **t** = [1 1 -1 -1 1 1 -1 -1 1 1]. Test the correct results by comparing its output with the output of the built-in `conv()`. Test both a column and row vector orientation.

Chapter 5: Digital modulation

The task of digital modulation is to convert signals conveying the information in form of binitis to the waveforms which are compatible with physical transmission channel characteristics.

Number of modulation states: $M = 2^k$, or: $k = \log_2 M$, where k is the number of bits per modulation symbol.

If:

$M = 2 \rightarrow$ binary modulations, used signals are:

- antipodal signals: $s_1(t) = -s_2(t)$, e.g. BPSK modulation
- orthogonal signals: $\sin(x)$, $\cos(x)$, e.g. BFSK modulation

$M > 2 \rightarrow$ we have M -ary modulations

Detection can be:

- coherent – the receiver has to be phase-synchronized with the receiving signal
- non-coherent - the receiver does not have to be phase-synchronized with the receiving signal

Binary modulations:

Coherent BPSK (Binary Phase Shift Keying):

Modulation signals:

$$s_1(t) = \sqrt{\frac{2E}{T_b}} \cos(\omega_0 t) \quad s_2(t) = \sqrt{\frac{2E}{T_b}} \cos(\omega_0 t + \pi) \quad 0 \leq t \leq T_b$$

Signal constellation (Fig. 5.1):

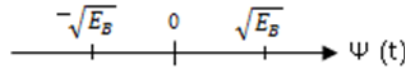


Fig. 5.1: Coherent BPSK.

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

Coherent BFSK (Binary Frequency Shift Keying):

Modulation signals:

$$s_1(t) = \sqrt{\frac{2E}{T_b}} \cos(\omega_1 t) \quad s_2(t) = \sqrt{\frac{2E}{T_b}} \cos(\omega_2 t)$$

In order to be orthogonal frequency spacing has to be:

a) $\Delta f = \frac{1}{2T_b}$ - in case of coherent detection

b) $\Delta f = \frac{1}{T_b}$ - in case of non-coherent detection

Signal constellation (Fig. 5.2):

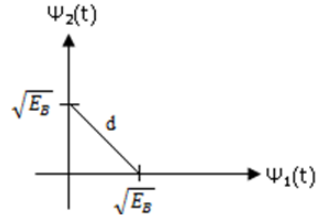


Fig. 5.2: Coherent BFSK modulation.

$$P_b = Q\left(\sqrt{\frac{E_b}{N_0}}\right)$$

Note: Coherent BPSK requires approximately 3dB smaller E_b / N_0 than coherent BFSK for $P_b \leq 10^{-4}$ under the same conditions because the argument of $Q(x)$ is larger for BPSK than for BFSK.

Differential BPSK: BDPSK

The information is conveyed in the difference between the previous and the current phase of the received signal.

$$P_b = \frac{1}{2} e^{-\frac{E_b}{N_0}}$$

For $P_b \leq 10^{-4}$ BDPSK requires by 1 dB greater E_b / N_0 than coherent BPSK to maintain the same P_b .

Non-coherent BFSK

Tone spacing in order to maintain orthogonality: $\Delta f = \frac{1}{T}$ $P_b = \frac{1}{2} e^{-\frac{E_b}{2N_0}}$

For $P_b \leq 10^{-4}$ non-coherent BFSK requires by 1 dB greater E_b / N_0 than coherent BFSK to maintain the same P_b .

M-ary modulations

Assumption: Gray code is used to map bits to symbols. Assumed channel is AWGN.

Coherent MPSK

Probability of symbol error: $P_s(M) \cong 2Q\left(\sqrt{\frac{2E_s}{N_0}} \sin\left(\frac{\pi}{M}\right)\right)$

Non-coherent MPSK: DMPSK

Probability of symbol error: $P_s(M) \cong 2Q\left(\sqrt{\frac{2E_s}{N_0}} \sin\left(\frac{\pi}{\sqrt{2}M}\right)\right)$

Coherent MFSK

Probability of symbol error: $P_s(M) \leq (M-1)Q\left(\sqrt{\frac{E_s}{N_0}}\right)$

Non-coherent MFSK

Probability of symbol error: $P_s(M) = \frac{1}{M} \exp\left[-\frac{E_s}{N_0}\right] \sum_{j=2}^M (-1)^j \binom{M}{j} \exp\left(\frac{E_s}{jN_0}\right)$

M-QAM

Square M-QAM constellations are most often used because they are the easiest to implement. A square M-QAM is formed by the product of two orthogonal M-ary PAM, (M must be even), see Fig. 5.3.

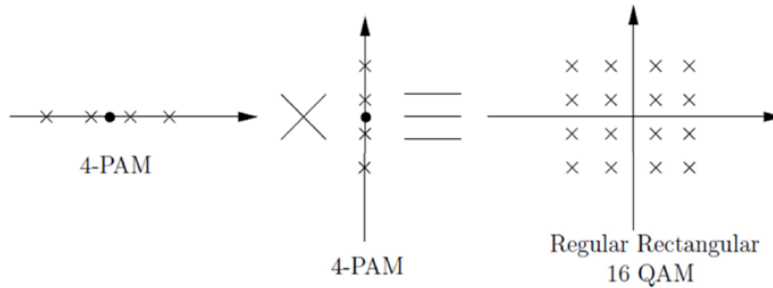


Fig. 5.3: The Cartesian product of two orthogonal 1 dimensional PAM constellation forms a 2D QAM constellation in signal space.

For a sufficient SNR, an approximate relationship for probability of symbol error can be used:

$P_s \cong 2P_{\sqrt{M}} = \frac{4(\sqrt{M}-1)}{\sqrt{M}} Q\left(\sqrt{\frac{3E_{sav}}{(M-1)N_0}}\right)$, where: $P_{\sqrt{M}} = \frac{2(\sqrt{M}-1)}{\sqrt{M}} Q\left(\sqrt{\frac{3E_{sav}}{(M-1)N_0}}\right)$ is the symbol error probability of the PAM from which M-QAM was created. E_{sav} is the mean energy per symbol.

In order to compare error probability of different types of modulation it is necessary to convert the probability of symbol error to bit error probability:

a) orthogonal modulations: $P_b = \frac{\frac{M}{2}}{M-1} P_s = \frac{2^{k-1}}{2^k - 1} P_s$

b) non-orthogonal modulations: $P_b \doteq \frac{P_s}{\log_2 M} = \frac{P_s}{k}$

Exercise 5.1:

The system uses BPSK signaling with parameter $E_b / N_0 = 5.0 \text{ [dB]}$. For detection, matched filters are used. We consider optimal detection with optimal decision threshold: $\gamma_0 = 0$. Assume ML decision criterion.

- Calculate bit error probability in case of optimal decision threshold.
- Calculate in % by how much the bit error probability will be increased (comparing to optimal detection and decision) if the decision threshold changes to: $\gamma_0 = 0.15 \times \sqrt{E_b}$

Solution:

a)

$$\text{In case of BPSK: } P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) = Q\left(\sqrt{2 \times 10^{0.5}}\right) = Q(2.515) \approx 5.954 \times 10^{-3}$$

b)

In this case we have to calculate probabilities when the sent signal was $s_1(t)$ but due to the noise it was detected as $s_2(t)$ and vice versa.

$$\begin{aligned} P_{b(0 \rightarrow 1)} &= Q\left(\frac{0.15 \times \sqrt{E_b} - (-\sqrt{E_b})}{\sqrt{\frac{N_0}{2}}}\right) = Q\left(\frac{1.15 \times \sqrt{E_b}}{\sqrt{\frac{N_0}{2}}}\right) = Q\left(1.15 \times \sqrt{\frac{2E_b}{N_0}}\right) = Q\left(1.15 \times \sqrt{\frac{2E_b}{N_0}}\right) \\ &= Q(1.15 \times 2.515) = Q(2.892) \approx 1.913 \times 10^{-3} \end{aligned}$$

$$\begin{aligned} P_{b(1 \rightarrow 0)} &= Q\left(\frac{0.15 \times \sqrt{E_b} - \sqrt{E_b}}{\sqrt{\frac{N_0}{2}}}\right) = Q\left(\frac{-0.85 \times \sqrt{E_b}}{\sqrt{\frac{N_0}{2}}}\right) = Q\left(0.85 \times \sqrt{\frac{2E_b}{N_0}}\right) \\ &= Q(0.85 \times 2.515) = Q(2.138) \approx 1.627 \times 10^{-2} \end{aligned}$$

Note:

$$Q(x) + \Phi(x) = 1$$

$$\Phi(x) = 1 - \Phi(-x) \Rightarrow \Phi(-x) = 1 - \Phi(x) = Q(x)$$

Total bit error probability:

$$P_{b_total} = P(1 \rightarrow 0)P(1) + P(0 \rightarrow 1)P(0) = 1.627 \times 10^{-2} \times \frac{1}{2} + 1.913 \times 10^{-3} \times \frac{1}{2} = 9.093 \times 10^{-3}$$

Relative increase of bit error probability due to the offset of the optimal threshold value:

$$error = \left(\frac{P_{b_total}}{P_b} - 1 \right) \times 100 = \left(\frac{9.093 \times 10^{-3}}{5.954 \times 10^{-3}} - 1 \right) \times 100 = 52.729 \text{ [%]}$$

■



Coherent BPSK modulation is analyzed in [10] in the TIMS experiment: “BER measurement of coherent BPSK signaling in an ideal distortion less channel”, Emona TIMS Volume D4 – Further Advanced Digital Experiments, experiment: D4-03.

Exercise 5.2:

The system with coherent BFSK modulation records in the average 10 errors per hour. Bit rate is: $R_b = 1200 \text{ [b / s]}$ and $N_0 / 2 = 5 \times 10^{-11} \text{ [W / Hz]}$.

- Calculate the average bit error rate – *BER*.
- Calculate the average received power *S* which will provide the same bit error probability as is *BER* from a).

Solution:

a)

10 errors per hour corresponds to 2.777×10^{-3} errors per second

$$BER = \frac{2.777 \times 10^{-3}}{1200} = 2.315 \times 10^{-6}$$

b)

For coherent BFSK: $P_b = Q\left(\sqrt{\frac{E_b}{N_0}}\right) \rightarrow$ we have to find an argument of $Q(x)$, such that

$P_b = 2.315 \times 10^{-6}$. Using tables for $Q(x)$ or software e.g. Mathematica it is possible to find an approximate solution: $Q(4.520089) = 2.315 \times 10^{-6}$

$$E_b = S T_b = \frac{S}{R_b} \Rightarrow S = E_b R_b$$

$$\sqrt{\frac{E_b}{N_0}} = 4.58089 \Rightarrow E_b = 4.58089^2 N_0$$

$$S = 4.58089^2 \times 2 \times 5 \times 10^{-11} \times 1200 = 2.518 \times 10^{-6} \text{ [W]}$$

■



Coherent BFSK modulation is analyzed in [11] in the TIMS experiment: “BER measurement of coherent BFSK signaling in an ideal distortion less channel”, Emona TIMS Volume D7 – Advanced BER Experiments, experiment: D7-02.

Exercise 5.3:

We have 4 binary communication systems operating over AWGN channel. Calculate the bit error probability for each system:

- a) coherent BPSK system with $E_b / N_0 = 8.0 \text{ [dB]}$
- b) Differential BDPSK system with $E_b / N_0 = 9.0 \text{ [dB]}$
- c) coherent BFSK system with $E_b / N_0 = 11.0 \text{ [dB]}$
- d) non-coherent BFSK system with $E_b / N_0 = 12.0 \text{ [dB]}$

Solution:

a)

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) = Q\left(\sqrt{2 \times 10^{0.8}}\right) = Q(3.552) \approx 1.909 \times 10^{-4}$$

b)

$$P_b = \frac{1}{2} e^{-\frac{E_b}{N_0}} = \frac{1}{2} e^{-10^{0.9}} \approx 1.775 \times 10^{-4}$$

c)

$$P_b = Q\left(\sqrt{\frac{E_b}{N_0}}\right) = Q\left(\sqrt{10^{1.1}}\right) = Q(3.548) \approx 1.940 \times 10^{-4}$$

d)

$$P_b = \frac{1}{2} e^{-\frac{E_b}{2N_0}} = \frac{1}{2} e^{-\frac{1}{2} 10^{1.2}} \approx 1.810 \times 10^{-4}$$

■

The results confirms the above stated theory with respect to the comparison of P_b for BPSK and BFSK modulations as well as for comparing coherently and non-coherently detected BPSK and BFSK modulations.



Non-coherent BFSK modulation is analyzed in [11] in the TIMS experiment: “BER measurement of noncoherent BFSK signaling in an ideal distortion less channel”, Emona TIMS Volume D7 – Advanced BER Experiments, experiment: D7-01.



DBPSK modulation is analyzed in [11] in the TIMS experiment: “BER measurement of DBPSK signaling in an ideal distortion less channel”, Emona TIMS Volume D7 – Advanced BER Experiments, experiment: D7-03.

Exercise 5.4:

We have 2 communication systems operating over AWGN channel. 1st system is using coherent 8FSK modulation with $E_b / N_0 = 10 \text{ [dB]}$. 2nd system is using coherent 16PSK modulation with $E_b / N_0 = 19 \text{ [dB]}$. We assume that the Gray code is used to map bits to symbols. Determine which system has a lower P_b .

Solution:

$$1^{\text{st}} \text{ system: } M = 8 \Rightarrow k = 3 \text{ and } \frac{E_s}{N_0} = k \frac{E_b}{N_0}$$

For coherent 8PSK the probability of symbol error is:

$$P_s(8) \leq (8-1)Q\left(\sqrt{\frac{E_s}{N_0}}\right) = 7Q\left(\sqrt{3 \times 10^{1.0}}\right) = 7Q(5.477) \approx 1.512 \times 10^{-7}$$

$$P_b = \frac{\frac{M}{2}}{M-1} P_s = \frac{\frac{8}{2}}{8-1} \times P_s = \frac{4}{7} \times P_s = \frac{4}{7} \times 1.512 \times 10^{-7} = 8.641 \times 10^{-8}$$

$$2^{\text{nd}} \text{ system: } M = 16 \Rightarrow k = 4 \text{ and } \frac{E_s}{N_0} = k \frac{E_b}{N_0}$$

For coherent 16PSK the probability of symbol error is:

$$P_s(M) \leq 2Q\left(\sqrt{\frac{2E_s}{N_0}} \sin\left(\frac{\pi}{M}\right)\right) = 2Q\left(\sqrt{2 \times 4 \times 10^{1.9}} \sin\left(\frac{\pi}{16}\right)\right) = 7Q(4.918) \approx 8.747 \times 10^{-7}$$

$$P_b = \frac{P_s}{\log_2 M} = \frac{8.747 \times 10^{-7}}{\log_2 16} \approx 2.187 \times 10^{-7}$$

The first system has lower P_b in spite of the fact that the second system has by 9 [dB] greater value of E_b/N_0 . It is caused by smaller inter-symbol distances in case of 16 PSK.

■



Different types of MSK modulations are analyzed in [10], Emona TIMS Volume D4 – Further Advanced Digital Experiments, experiments: D4-04, D4-05, D4-06.

Exercise 5.5:

Communication system has bit rate 1 [Mb/s], the average amount of signal power pre modulation symbol is $S_{avg} = 10^{-3}$ [W]. Let assume AWGN with $N_0 = 10^{-10}$ [W / Hz]. Calculate bit error probability in case of 16QAM and 64QAM modulations – let assume square constellations. We assume that the Gray code is used to map bits to symbols.

Solution:

16QAM:

$$R_s = \frac{R_b}{k} = \frac{10^6}{4} = 250 \text{ [ks / s]} \Rightarrow T_s = \frac{1}{R_s} = 4 \times 10^{-6} \text{ [s]}$$

$$\text{average energy per symbol: } E_s = S_{avg} \cdot T_s = 10^{-3} \times 4 \times 10^{-6} = 4 \times 10^{-9} \text{ [J]}$$

$$\frac{E_s}{N_0} = \frac{4 \times 10^{-9}}{10^{-10}} = 40$$

$$P_{\sqrt{16}} = \frac{2(\sqrt{16}-1)}{\sqrt{16}} Q\left(\sqrt{\frac{3E_{sav}}{(16-1)N_0}}\right) = \frac{6}{4} \times Q\left(\sqrt{\frac{3}{15} \times 40}\right) = 1.5 \times Q(\sqrt{8}) = 1.5 \times Q(2.828) \approx 3.508 \times 10^{-3}$$

$$P_s \approx 2P_{\sqrt{16}} = 2 \times 3.508 \times 10^{-3} = 7.017 \times 10^{-3}$$

64QAM:

$$R_s = \frac{R_b}{k} = \frac{10^6}{6} = 166.667 \text{ [ks / s]} \Rightarrow T_s = \frac{1}{R_s} = 6 \times 10^{-6} \text{ [s]}$$

$$\text{average energy per symbol: } E_s = S_{avg} \cdot T_s = 10^{-3} \times 6 \times 10^{-6} = 6 \times 10^{-9} \text{ [J]}$$

$$\frac{E_s}{N_0} = \frac{6 \times 10^{-9}}{10^{-10}} = 60$$

$$P_{\sqrt{64}} = \frac{2(\sqrt{64}-1)}{\sqrt{64}} Q\left(\sqrt{\frac{3E_{sav}}{(64-1)N_0}}\right) = \frac{14}{8} \times Q\left(\sqrt{\frac{3}{63} \times 60}\right) = 1.75 \times Q(\sqrt{2.857}) = 1.75 \times Q(1.690) \approx 7.960 \times 10^{-2}$$

$$P_s \approx 2P_{\sqrt{64}} = 2 \times 7.960 \times 10^{-2} = 1.592 \times 10^{-1}$$

■



16QAM modulation is analyzed in [12] in the TIMS experiment: “16QAM LTE modulation”.



BER measurement of 16QAM modulation is analyzed in [12] in the TIMS experiment: “BER measurement of 16QAM LTE modulation”.

Lab 5.1: MATLAB exercises: BPSK & 4QAM in signal space

The BPSK modulation is considered a one dimensional (1D) modulation. The quaternary QPSK consists of four prototypes placed as corners of a square in the 2D complex (or IQ) plane, aka the signal space.

However, we can design three main forms of the BPSK constellation: a real-one, and imaginary one, and an actual two-dimensional one. Fig. 5.3 illustrates all of them.

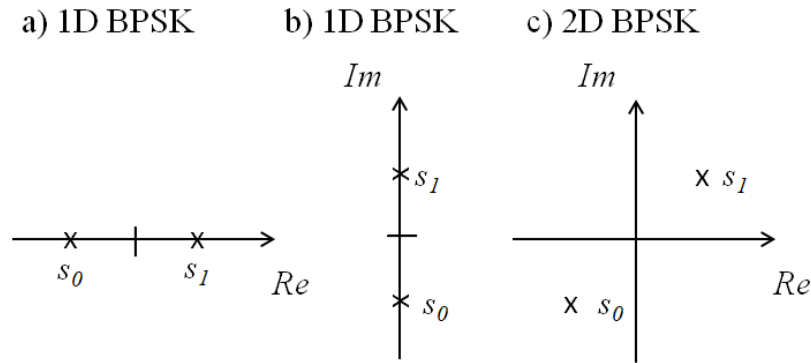


Fig. 5.3: Three forms of BPSK: first two are considered 1D (a, b)). The third one c) may be sometimes considered 2D.

While the first two constellations are very simple, the third, truly complex form of BPSK seems to be unnecessarily complicated. Especially when it comes to detection in the receiver: while the detector for cases a) and b) only needs to compare the received sample with zero and can immediately return the bit estimate (for instance by returning the sign bit), the source code for determining, whether a received complex sample z is closer to s_1 or to s_0 involves squaring of x and y component differences. More on detection will be given in later chapters, but it is important to understand now, that the constellation design is affected by the detection process in the receiver.

The 1D BPSK is also useful when thinking about the 2D QPSK modulation. This is illustrated in Fig. 5.4.

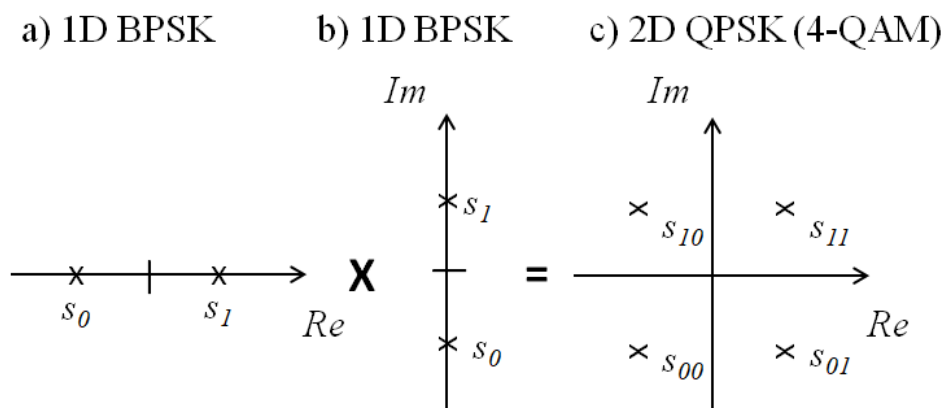


Fig. 5.4: A 2D QPSK can be understood as a Cartesian products of two 1D BPSK modulations.

There are many similar forms of QPSK which will differ only by their bit mapping and rotation of the constellation from the positive x -axis direction. The constellation in Fig. 5.2 c) is usually called 4-QAM

and from the perspective of its error performance in AWGN channel is equivalent to both other variants shown in Fig. 5.5.

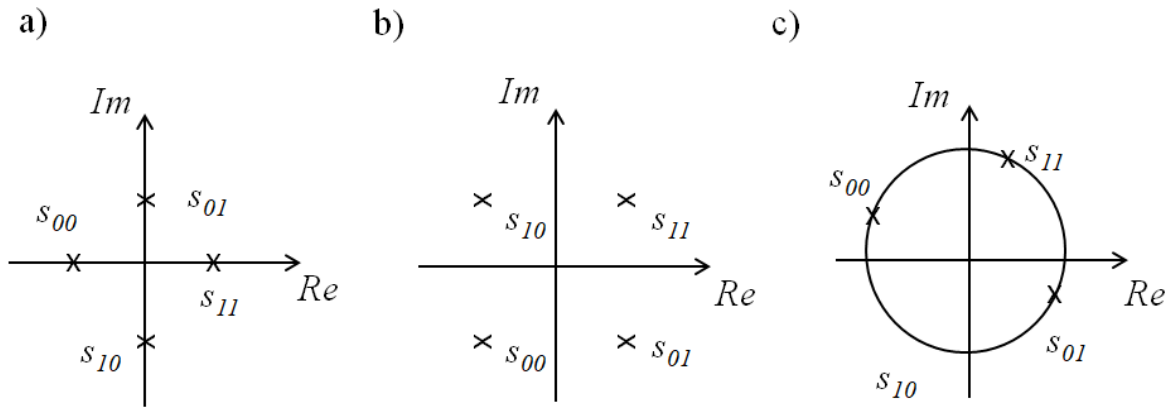


Fig. 5.5: Various forms of QPSK: The difference lies mainly in the rotation of the constellation.

Since we like to simplify our calculations as much as possible, we may sometimes assume normalized transmitter energy per bit $E_b = 1$ and zero channel attenuation. This means, the magnitudes of the prototypes will be equal to 1 for all the modulations in question. This determines the exact values of the prototypes:.

Real-valued BPSK: $s_0 = -1, s_1 = 1$

Imaginary-valued BPSK: $s_0 = -j, s_1 = j$

Complex-BPSK rotated $\pi/4$: $s_0 = -\left(\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j\right), s_1 = \left(\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j\right)$

QPSK: $s_{11} = 1, s_{10} = j, s_{00} = -1, s_{01} = -j$

4-QAM: $s_{00} = -\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}j, s_{01} = \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}j, s_{11} = \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j, s_{10} = -\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j$

MATLAB Exercise 5.1.1: Real-valued BPSK modulator

Implement a public function given the prototype:

```
function SignalSpaceSymbols = modulateBPSK( LogicalSymbols )
```

that performs digital BPSK modulation. The output should be signal space prototypes with no imaginary component. Use the following mapping of the logical binary signal to signal space prototype: 0 $\rightarrow$ -1, 1 $\rightarrow$ +1. The input and output may be a scalar, vector or a matrix.

MATLAB Exercise 5.1.2: Universal BPSK modulator

Implement a public function given the prototype:

```
function SignalSpaceSymbols = modulateComplexBPSK( LogicalSymbols, Map, A, rot )
```

that performs digital BPSK modulation. The output should be signal space (IQ plane) prototypes. The Map vector defines the mapping:

[0 1] defines the default mapping: 0 $\rightarrow$ -A, 1 $\rightarrow$ +A

[1 0] defines the other mapping: 0 $\rightarrow$ +A, 1 $\rightarrow$ -A

Parameter A defines the required magnitude of the prototypes, and an optional parameter `rot` is the rotation of the constellation from the positive x-axis in radians counterclockwise. The input and output may be a scalar, vector or a matrix, and the output values are allway complex numbers, even if their real or imagirany parts are zero.

MATLAB Exercise 5.1.3a: Implement a public function given the prototype:

```
function index = bits2Index( bits )
```

That converts a binary vector of two bits to an integer scalar index (converts from binary to decimal representation). Use this function in both your implementations of QPSK and 4QAM modulators.

MATLAB Exercise 5.3: Basic QPSK modulator

Implement a public function given the prototype:

```
function SignalSpaceSymbols = modulateQPSK( LogicalSymbols, A )
```

that performs digital QPSK modulation. The output should be signal space (IQ plane) prototypes with Gray mapping of bits (logical symbols) as defined by Fig. 5.4. Parameter A defines the required magnitude of the prototypes. `LogicalSymbols` may be any matrix with an even number of row: since two bits in neighbouring rows will be mapped to one prototype, each pair of bits in rows i and $i+1$ will determine the mapping to symbol index: $S_{i(i+1)}$. For example a matrix:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

will be mapped to indices, shown first as binary vectors, and also converted to decimal indices:

$$\begin{pmatrix} 00 & 01 & 10 \\ 11 & 10 & 01 \end{pmatrix} \equiv \begin{pmatrix} 0 & 1 & 2 \\ 3 & 2 & 1 \end{pmatrix}$$

The indices may then be used for indexing a map of prototype values. The exact numeric values of the prototypes will be determined by the constellation diagram in Fig. 5.4 a) and the parameter A . For unit energy the values will be:

$$\begin{pmatrix} s_{00} & s_{01} & s_{10} \\ s_{11} & s_{10} & s_{01} \end{pmatrix} = \begin{pmatrix} -1 & j & -j \\ 1 & -j & j \end{pmatrix}$$



BER measurement of coherent QPSK modulation is analyzed in [11] in the TIMS experiment: “BER measurement of coherent QPSK signaling in an ideal distortion less channel”, Emona TIMS Volume D7 – Advanced Advanced BER Experiments, experiment: D7-04.

MATLAB Exercise 5.1.4: 4-QAM modulator

Implement a public function given the prototype:

```
function SignalSpaceSymbols = modulate4QAM( LogicalSymbols, A )
```

that performs digital 4QAM modulation with Gray mapping as defined by Fig. 5.4. The output should be signal space (IQ plane) prototypes. The optional parameter `A` defines the required prototype magnitude and `LogicalSymbols` may be any matrix with an even number of row with mapping of bits to symbol indices exactly the same as defined in the previous exercise. For example a matrix:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

will be mapped to symbols:

$$\begin{pmatrix} s_{00} & s_{01} & s_{10} \\ s_{11} & s_{10} & s_{01} \end{pmatrix}$$

which for unit energy per bit translates to a following prototype matrix:

$$\begin{pmatrix} -\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}j & \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}j & -\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j \\ \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j & -\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j & \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}j \end{pmatrix}$$

Chapter 6: Multiple Access (MA)

Communication Resource (CR) represents the time and bandwidth available for the purpose of signalling (message transmission) for a given communication system. The CR can be represented by a time-frequency plane:

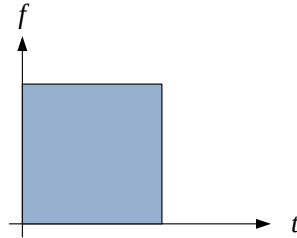


Fig. 6.1: Time-frequency plane.

For efficient use of the communication system, it is important to plan in advance how to allocate resources to individual users so that no area of the available time-space plane is unused and, at the same time, so that the allocation of user resources is fair.

The term multiplexing, or multiple access, describes a method of allocating available communication resources. There is little difference between these terms.

Multiplexing - describes a method of allocating resources on a fixed (predefined) basis. Alternatively, this method may vary slowly. Resource sharing here is done on a local basis, e.g. on a given device.

Multiple access describes a method of sharing a remote resource, e.g. a satellite link by different users. The method of this sharing can change dynamically based on user requirements, e.g. in LTE.

The essence of all multiple channel sharing techniques is to prevent the generation of interference at the receiving end of the users that would prevent them from communicating. That is, the signals of other users must not cause a significant increase in the error rate of a given user. The mutual orthogonality of the signals minimizes this interference (see eq. 4.12).

Basic MA techniques:

- FDMA (Frequency Division Multiple Access)
- TDMA (Time Division Multiple Access)
- CDMA (Code Division Multiple Access)
- SDMA (Space Division Multiple Access)
- PDMA (Polarization Division Multiple Access)

FDMA

Each user is assigned a certain part of the available frequency spectrum (channel, frequency slot) within which only he transmits his data. To prevent interference between adjacent bands, they are separated by guard bands, see Fig. 6.2. The time-frequency plane is divided horizontally, i.e. each user is allocated a specific frequency band for a theoretically unlimited amount of time. To avoid mutual interference between different frequency bands, their signals must have been mutually orthogonal in frequency domain:

$$\int_{-\infty}^{\infty} X_i(f) X_j(f) df = \begin{cases} K & \text{for } i = j \\ 0 & \text{otherwise} \end{cases} \quad (6.1)$$

where K is a non-zero constant and: $X_i(f) \xleftrightarrow{FT} x_i(t)$ $i = 1, 2, \dots, N$, N is the number of users.

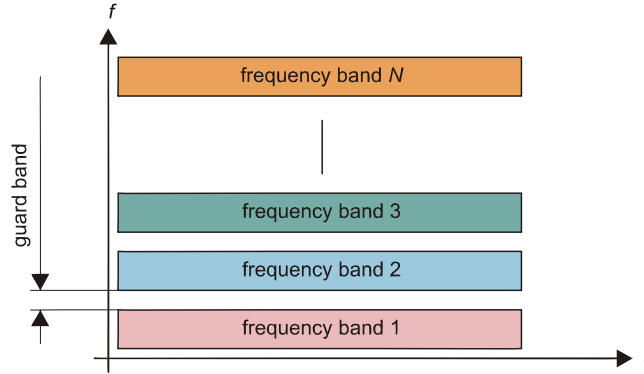


Fig. 6.2: FDMA principle.

TDMA

Each user is allocated the entire available frequency spectrum at any given time, but only for a certain period of time within which only he or she transmits his or her data (time slot). To avoid interference between adjacent time slots, these are separated by guard times:

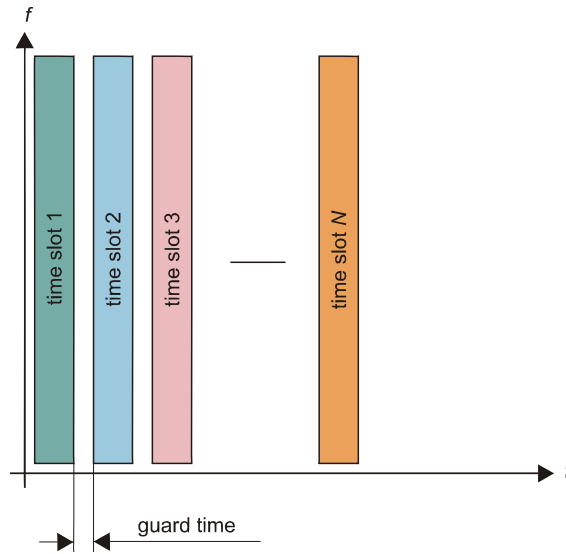


Fig. 6.3: TDMA principle.

The time-frequency plane is divided vertically, i.e. each user is allocated the entire frequency band, but only for a certain amount of time. To avoid mutual interference between signals from different time intervals their signals must be mutually orthogonal:

$$\int_{-\infty}^{\infty} x_i(t) x_j(t) dt = \begin{cases} K & \text{for } i = j \\ 0 & \text{otherwise} \end{cases} \quad (6.2)$$

where K is a non-zero constant and: $x_i(t)$ $i = 1, 2, \dots, N$ are the time waveforms of the signals of the individual users, where N is the number of users.

SDMA

To each user is allocated a separate beam generated by the multielement antenna array. Users are separated in space. Such arrangement allows using the same frequency within the same geographical area e.g. a cell.

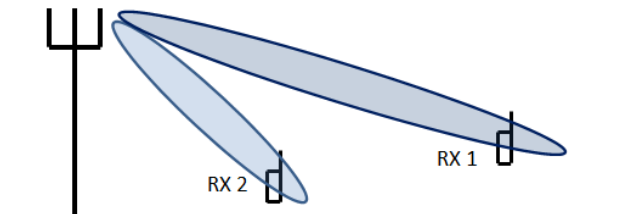


Fig. 6.4: SDMA principle.

PDMA

Each user uses a different type of polarization within the same frequency band. Used polarizations have to be mutually orthogonal.

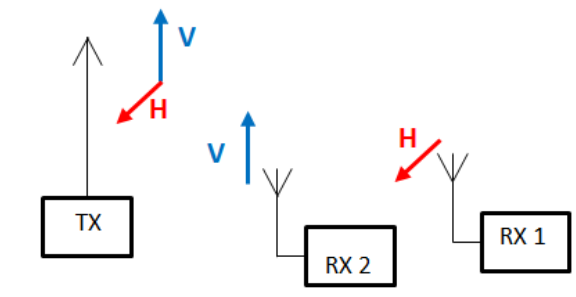


Fig. 6.5: PDMA principle.

CDMA

In the next we will focus on CDMA methods. Advantages of CDMA over FDMA and TDMA:

- privacy – without knowledge of the PN code, the received signal cannot be decoded
- resistance to interference – statistically, if the interfering signal damages less than half of the chips used to send one modulation symbol, it is still possible to restore it without errors (possibility of communication below the noise limit)
- resistance to multipath propagation - statistically, if the reflected signals arrive at the antenna in a time greater than the chip duration, they are decorrelated.
- asynchronous operation – compared to TDMA, CDMA does not need synchronization of all users. Thanks to the orthogonality of PN codes, each user can broadcast at any time.

As PN codes are used various PN sequences satisfying the general requirements placed on PN sequences for CDMA systems, which are:

1. Balance property – in each period of PN sequence the number of binary ones should ideally differ from the number of binary zeros by at most one digit.
2. Run-length property. A run is defined as the sequence of a single type of binary digit(s). The occurrence of the alternate digit in a sequence marks a new run. The number of digits in the run defines the length of the run. Among the runs of ones and zeros in each period, it is desirable that about $\frac{1}{2}$ the runs of each type are of length 1, about $\frac{1}{4}$ are of length 2, about $\frac{1}{8}$ are of length 3, and so on.
3. Correlation property. If a period of the sequence is compared term by term with any cyclic shift of itself, in an ideal case the number of agreements differs from the number of disagreements by not more than one count (except for a zero shift).

The best PN sequences with regard to the above mentioned properties are so called maximal length sequences, or m -sequences. m -sequences are the longest sequences generated by a given shift register with m stages. Such register (defined by a primitive polynomial of a degree m) generates m -sequence of length:

$$N_c = 2^m - 1 \quad (6.3)$$

, where N_c is the period of m -sequence in chips [c]. Chip is one single PN sequence symbol. Chip duration T_c is the shortest time interval of a given sequence, see Figure 6.6.

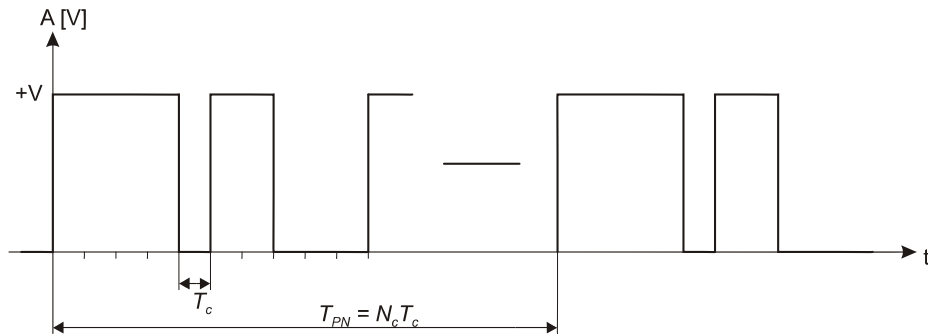


Fig.6.6: Illustration of time relationships in PN sequence.

As can be seen from Fig. 6.6 each PN sequence repeats itself – its period T_{PN} is directly proportional to its length in [c]:

$$T_{PN} = N_c T_c \quad (6.4)$$

A shift register sequence generator consists of a shift register working in conjunction with the appropriate logic, which feeds back a logical combination of states of two or more of its stages to its input. The output of a sequence generator, and the content of its m stages at any clock interval, is a function of the outputs of the stages fed back at the preceding clock interval. The clock rate of shift register sequence generator determines the chip time T_c :

$$T_c = \frac{1}{R_c} = \frac{1}{R_{clk}} \quad (6.5)$$

, where R_c is the chip rate in [c/s] and R_{clk} is the clock rate of a sequence generator. As was mentioned earlier in CDMA system different users are separated by unique PN codes assigned to each user. To assure the effective user separation used PN codes have to be orthogonal (or nearly orthogonal). So the

degree of mutual orthogonality relates to cross-correlation property. Two real valued waveforms $x(t)$ and $y(t)$ are orthogonal if the value of their cross-correlation $R_{xy}(\tau)$ over the symbol interval T equals to zero:

$$R_{xy}(\tau) = \int_0^T x(t) y(t + \tau) dt = 0 \quad (6.6)$$

The ability of resistance of CDMA systems against jamming is a measure of the system performance and can be expressed with signal-to-jammer ratio:

$$SJR = \frac{E_s}{E_j} \frac{n}{N} \quad (6.7)$$

, where E_s is the energy of the signal from the signaling alphabet: $S = \{s_1, s_2, \dots, s_N\}$. N is the alphabet size. E_j is the jammer energy; n is the signal dimension after spreading. Signal-to-jammer ratio after post processing (6.7) gives to communicator an advantage of n/N over the jammer. Factor n/N is called the processing gain (denoted as P_G) and it is exactly equal to the ratio of the dimensionality of the possible signal space (and therefore the space in which the jammer must seek to operate) to the dimensions needed to actually transmit the signals. Processing gain can be defined as:

$$P_G = \frac{n}{N} = \frac{2W_{ss}T_s}{2WT_s} = \frac{W_{ss}}{W} \quad (6.8)$$

, where W_{ss} is the bandwidth of the spread spectrum signals, W is the minimum bandwidth of unspread signals.

Depending on the jamming waveforms, jammers can be classified into the following classical categories:

- a) broadband noise jammers – jammer occupies the whole system bandwidth. Jammer distributes jamming power J evenly over the W_{ss} , therefore his equivalent single-sided PSD, denoted as J_0 equals to:

$$J_0 = \frac{J}{W_{ss}} \quad (6.9)$$

It means that jammer applies a little power to each of the signal coordinates. The broadband noise jammer is a brute force jammer that does not exploit any knowledge of the communication system except its bandwidth W_{ss} .

- b) narrowband noise jammers – also referred to as partial-band jammers spreads the noise of total power J evenly over some portion of the system bandwidth. Let define ρ as the ratio of the jammer's bandwidth to the system bandwidth:

$$\rho = \frac{W_J}{W_{ss}} \leq 1 \quad (6.10)$$

, using (6.8) and (6.9) jammer's PSD equals to: $\frac{J}{W_j} = \frac{J_0}{\rho}$. In this case jammer trades his

bandwidth for an increased PSD (he jams a few signal coordinates, but with higher power – assuming J to be constant).

- c) stepped noise jammers – as in case b), but now jammer steppes through different regions of the system bandwidth at random times.
- d) CW and multitone jammers – jammer is using either a continuous wave (CW) – a single tone with high power, or a several tones of equal power – a multitone jammer.
- e) stepped CW and multitone jammers – as in case d), but now jammer steppes choices different frequencies for CW or multitones at random times.
- f) pulse jammers – jammer transmits pulses of bandlimited white Gaussian noise having time-averaged received power J . The actual jamming power during a jamming pulse is larger. Also jammer can trade duty cycle for increased jamming power. If the jammer's pulse duration is denoted as ρ , then jammer's PSD is during that time increased to:

$$J_0 = \frac{J}{W_{ss} \rho} \quad (6.11)$$

, maintaining a constant time-averaged jamming power J . So jamming is present during a fraction ρ of time, for the remaining fraction $(1 - \rho)$ of time jamming is not present. Jammer occupies the whole system bandwidth.

- g) repeat-back jammers – this type of jamming is only applicable to FH SS systems. In this case a jammer is monitoring communicator's communication based on which they are able to generate jamming signal (fast enough) in the spectral vicinity of communicator's FH/FSK signal.

See Figure 6.7 for categories a) through e).

FH-CDMA

Each user is allocated only a certain part of the available spectrum at any given time for a certain period of time within which only he transmits his data. To avoid interference between adjacent time slots and frequency bands, these are separated from each other by guard time and frequency slots. The assignment of a given frequency subband to a given source is determined by the PN generator. This allocation changes at each time interval. An example of FH-CDMA applications are Bluetooth Technology and WISSCE (Wireless Indoor Spread Spectrum Communication Equipment). In FH-CDMA system N users share the same bandwidth, transmit simultaneously and each user has its own PN code determining its hopping pattern, see Figure 6.8. The general block diagram of FH-CDMA communication system is shown in Figure 6.9.

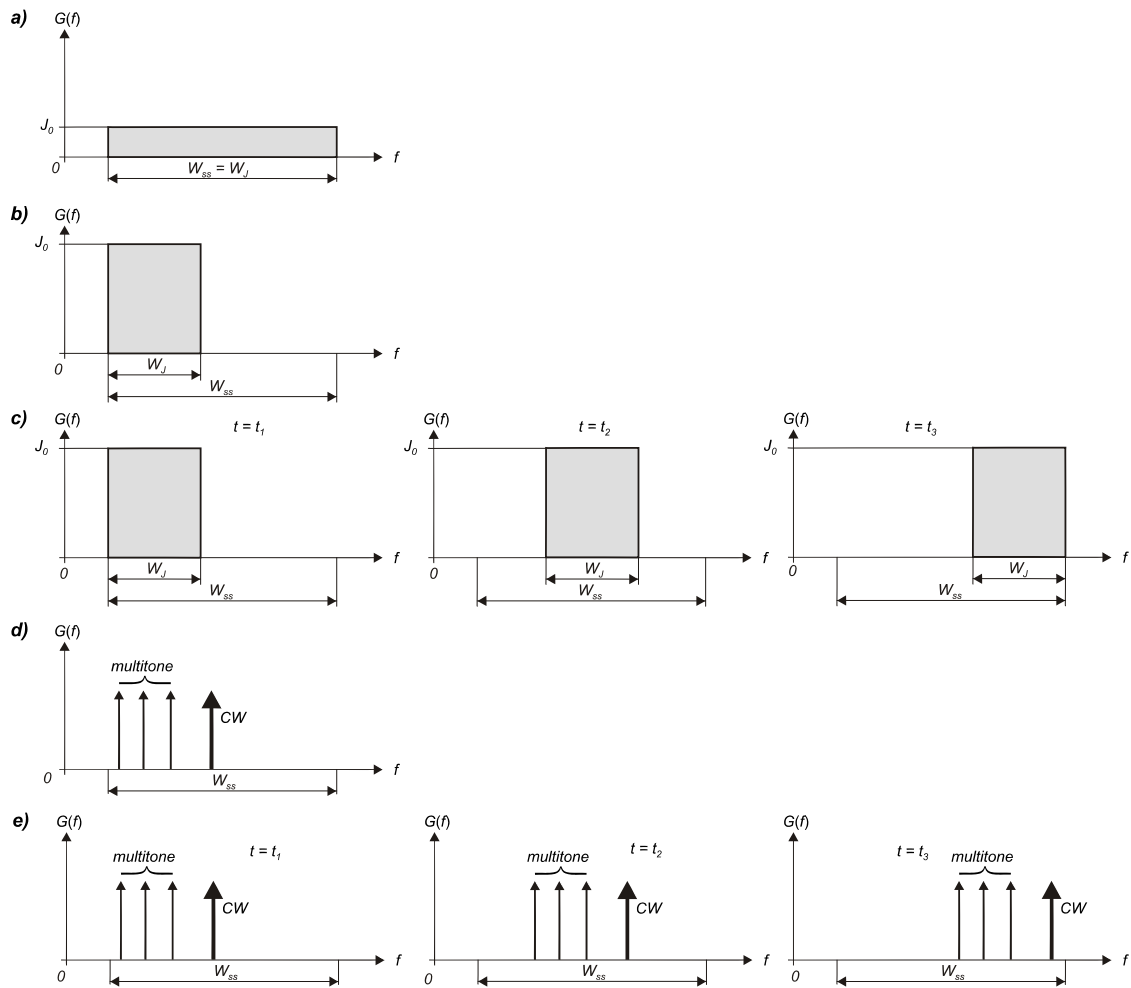


Fig. 6.7: The most common jamming strategies.

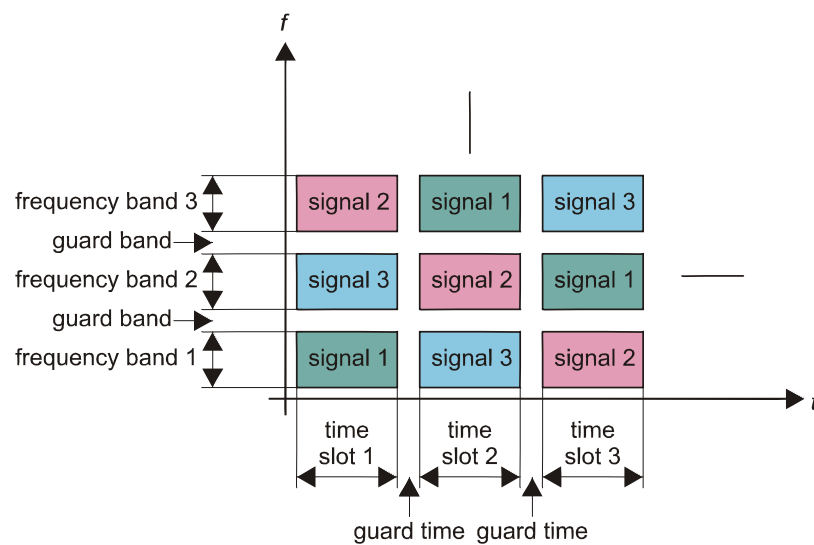


Fig. 6.8: FH-CDMA principle.

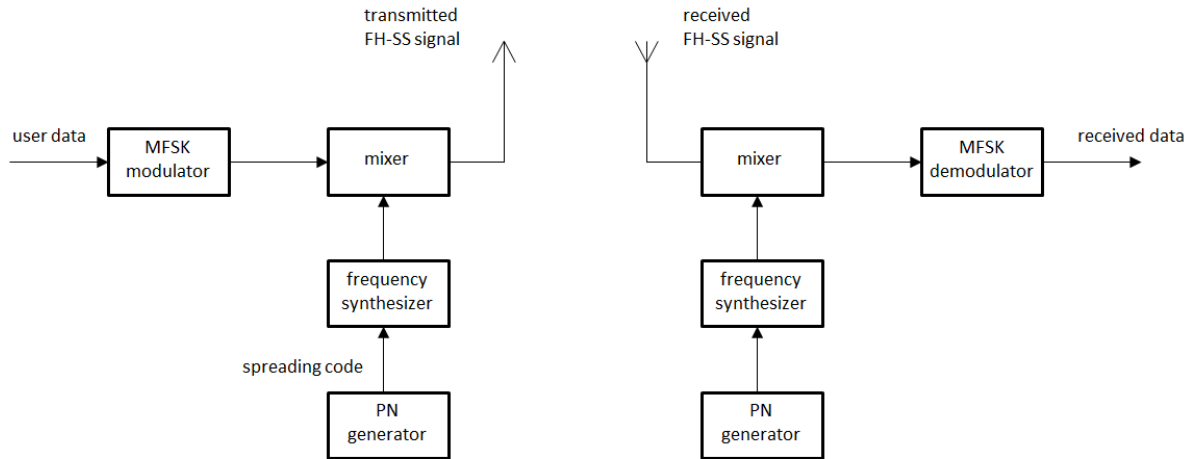


Fig. 6.9: General block diagram of FH-CDMA communication system.

In FH-CDMA communication system the available bandwidth W_{ss} is subdivided into a large number of frequency slots: $S_1, S_2, \dots, S_J$, see Fig. 6.10. Each slot has bandwidth W_s determined by:

- number of used modulation states: M
- frequency of the individual tones of used symbol alphabet
- type of used detection: coherent or non-coherent.

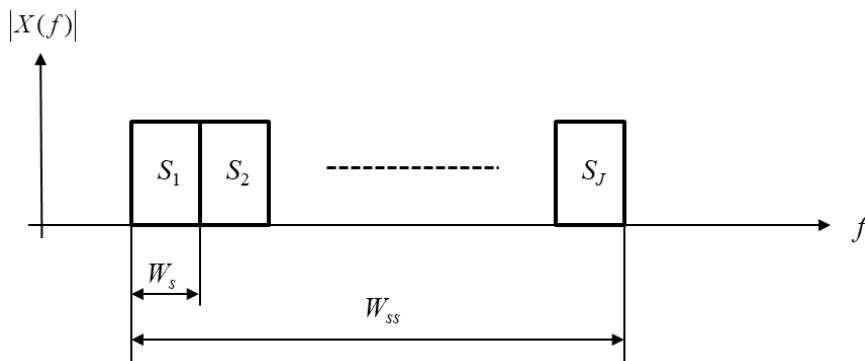


Fig. 6.10: FH-CDMA bandwidth.

The number of frequency slots is set by the PN spreading generator, realized as m – stages linear feedback shift register generates which generates:

$$J = 2^m - 1 \quad (6.12)$$

distinguished sequences m bits long (frequency words) that control the output frequency of frequency synthesizer. The conventional FSK modulator depending on the input data produces:

$$M = 2^k \quad (6.13)$$

different frequencies (tones), where k denotes the number of bits in each modulation symbol, thus: $k = \log_2(M)$. An example of simple FH pattern is shown in Fig. 6.11.

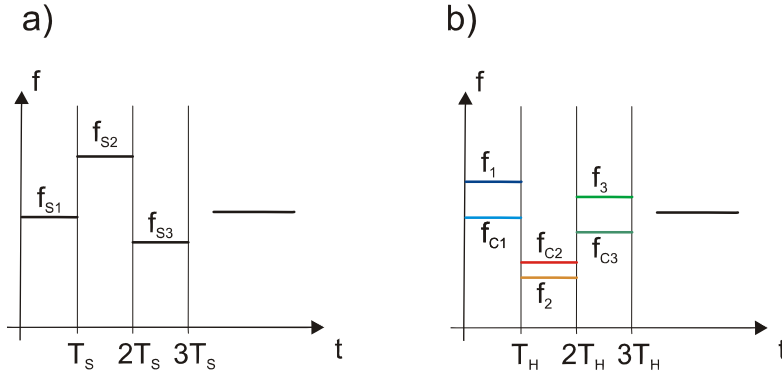


Fig. 6.11: An example of simple FH pattern.

In Fig. 6.11a) each k – tuple of the input data bits determines frequency f_{si} , where $i \in \{1, \dots, M\}$ of one modulation symbol. Time duration of one modulation symbol is: $T_s = k \cdot T_b$, where T_b is data bit duration. In Fig. 6.11b) the resulting MFSK signal f_{si} is translated in frequency by the amount that is determined by the output sequence from the PN generator, which in turn controls the output frequency f_{cj} , where $j \in \{1, \dots, J\}$ synthesized by the frequency synthesizer. Time duration of generating one single frequency at the output of the frequency synthesizer is denoted as hop time T_H . This frequency is mixed with the output of the MFSK modulator and the resultant frequency–translated signal f_l (l is time index, $l \in \{1, \dots\}$) is transmitted over the channel. In FH-CDMA systems chip interval T_c determines the shortest duration of transmitting of one single frequency (an uninterrupted waveform). If we are referring to a chip in conjunction with PN sequence controlling frequency synthesizer in FH-CDMA systems, then chip has the same meaning as in DS-SS described next. Therefore in order to avoid confusion, it has to be paid an attention under which circumstances this term is used.

In FH-CDMA systems the relationship between symbol interval T_s and hop interval T_H classifies the systems into one of two categories:

1. SFH (Slow Frequency Hopping) systems – in this case: $T_H \geq T_s$, and $T_c = T_s$. This means that frequency synthesizer produces one constant frequency during one or several modulation symbols:

$$T_H = n T_s \quad (6.14)$$

, where n is the number of modulation symbols per one frequency hop. MFSK modulator (depending on input data) produces one constant tone lasting one-symbol interval T_s . Since MFSK signal offsets frequency synthesizer output during each modulation symbol independently, time for transmitting an uninterrupted waveform (a single frequency) is symbol interval, therefore chip interval equals symbol interval. Reciprocal value of hop time defines a hop rate:

$$R_H = \frac{1}{T_H} = \frac{1}{n T_s} = \frac{1}{n} R_m = \frac{R_b}{nk} [\text{hops} / s] \quad (6.15)$$

, where R_m is modulation rate, R_b is data rate; k is number of data bits per modulation symbol. An example of SFH is shown in Fig. 6.12. In this example: $T_H = 2 T_s$. It means

that frequency synthesizer generates one particular frequency during two modulation symbols.

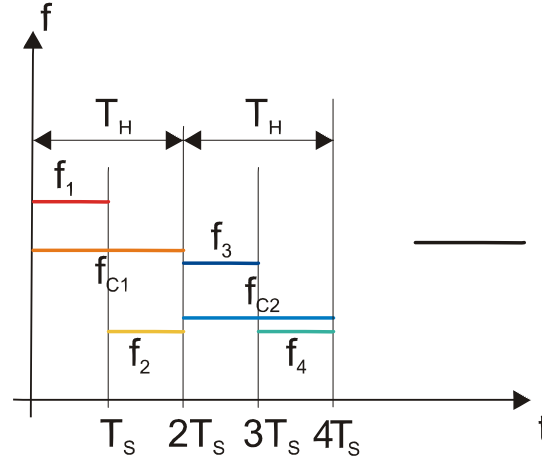


Fig. 6.12: An example of SFH hopping pattern.

2. FFH (Fast Frequency Hopping) systems – in this case: $T_H < T_s$, and $T_c = T_H$. This means that frequency synthesizer changes frequency several times per modulation symbol. Therefore the shortest time interval of transmitting an uninterrupted waveform is hop time, what means that chip time equals hop time. Let denote the number of frequency hops per modulation symbol as N , it follows that:

$$T_s = N \cdot T_H \quad (6.16)$$

FFH SS systems (with $N > 1$) are sometime called systems with *diversity*. In case of FFH hop rate is defined as follows:

$$R_H = \frac{1}{T_H} = \frac{N}{T_s} = NR_m = N \frac{R_b}{k} [\text{hops} / \text{s}] \quad (6.17)$$

An example of FFH is shown on Fig. 6.13. In this example: $T_s = 4T_H$. It means that frequency synthesizer changes frequency four times per one modulation symbol. During one modulation symbol interval MFSK modulator offsets the frequency synthesizer output frequency by the same amount.

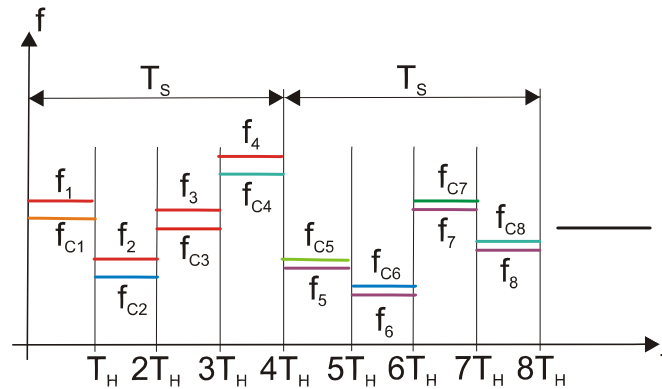


Fig. 6.13: An example of FFH hopping pattern.

DS-CDMA

Each user is assigned a unique PN code with which they multiply (XOR) their data. This has the effect of extending the spectrum of the user to the entire available spectrum. All users can broadcast simultaneously within the entire available band at any time, see Figure 6.14. DS-CDMA systems are used in GNSS: e.g. Galileo or GPS. (They were also used in 3G mobile communication systems).

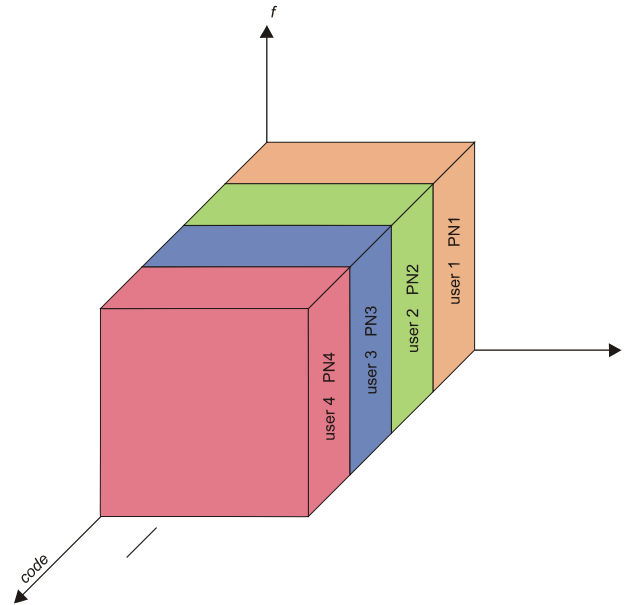


Fig. 6.14: DS-CDMA principle.

In DS-CDMA system N users share the same bandwidth, transmit simultaneously and each user has its own PN code, see Figure 6.15.

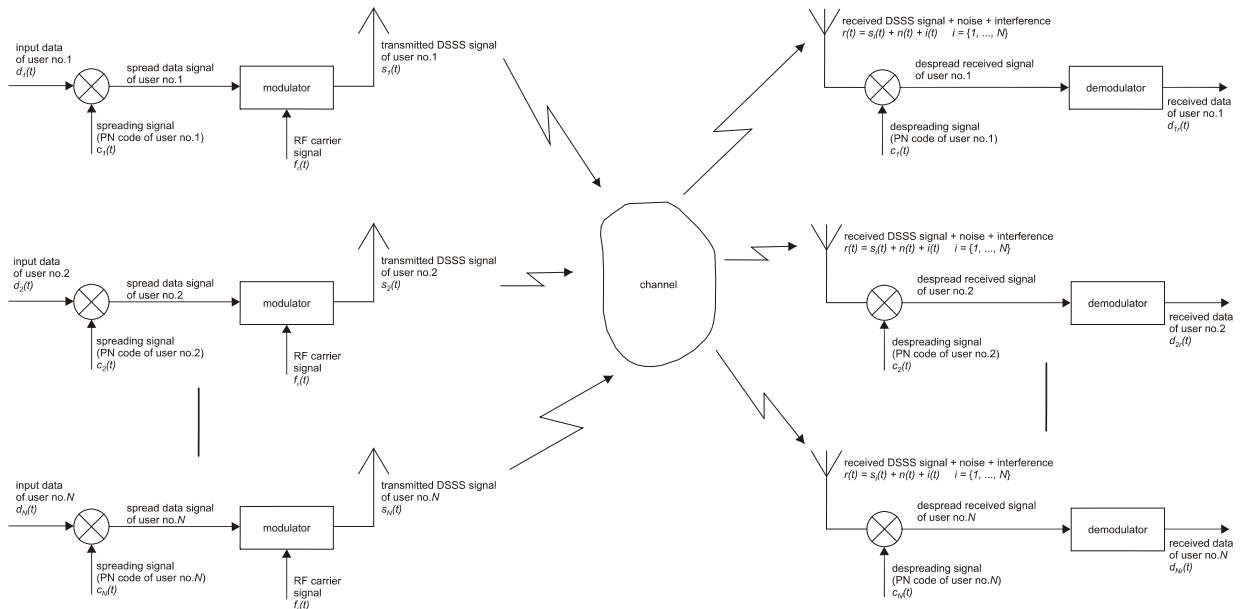


Fig. 6.15: General model of DS-CDMA communication system.

Each pair transmitter – receiver in Figure 6.15 consists of a classical DS-SS transmitter/receiver pair. On the transmitting side DS-CDMA transmitter is a classical DS-SS transmitter using such a spreading code

that satisfies the CDMA system requirements (number of simultaneous users, environment, etc.). Block diagram of DS-SS transmitter is shown in Figure 6.16, where RF carrier signal $f_c(t)$ is first modulated with data signal $d(t)$, then data – modulated signal is again modulated with high-speed (wideband) spreading signal $c(t)$, also called the code.

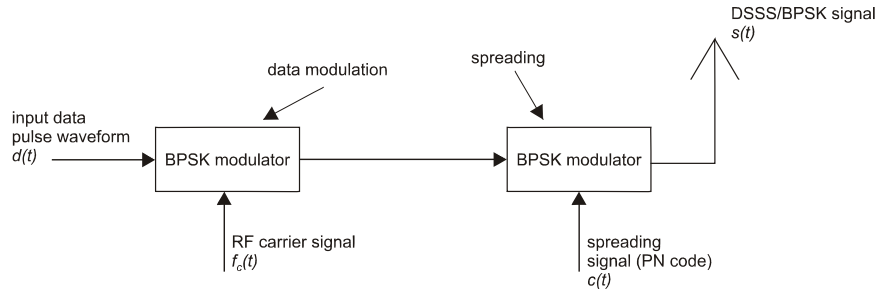


Fig. 6.16: Block diagram of DSSS/BPSK transmitter.

If data signal is an antipodal pulse stream (bipolar non-return to zero signal - BPNRZ) with values of pulses +1 or -1 and the spreading PN signal is also a BPNRZ signal, then simplified DSSS/BPSK transmitter can be realized as is shown in Figure 6.17. Data signal is spread by a direct multiplication (analog) with spreading signal.

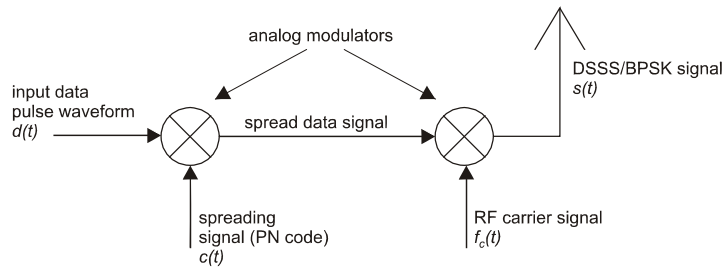


Fig. 6.17: Simplified block diagram of DSSS/BPSK transmitter.

In case that the clock frequency of the data signal is a sub-multiple of the clock frequency of the PN spreading signal then spreading of the data signal can be achieved by performing modulo two sum of the data signal and the spreading signal. In this case both signals are assumed to be unipolar non-return to zero signals (UNRZ) with values of pulses +1 or 0. In order to modulate such a spread data signal onto RF carrier a conversion from UNRZ to BPNRZ signal is necessary, see Figure 6.18.

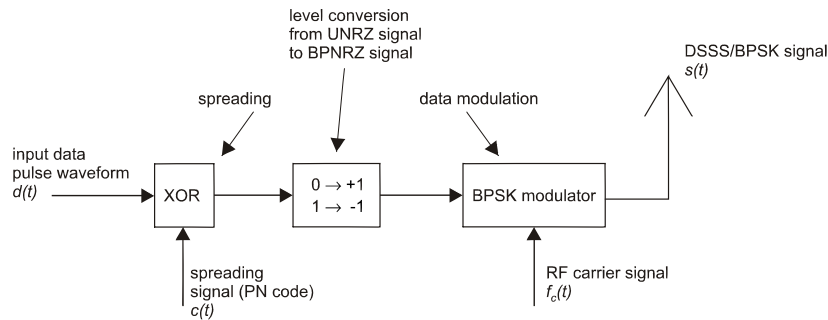


Fig. 6.18: Another alternative of block diagram of DSSS/BPSK transmitter.

On the receiving side spread spectrum signal cannot be detected by a conventional narrowband receiver. For the correct detection of the received signal of the given user, the incoming signal has to be first

“despread” using the same PN sequence as was used for spreading and then demodulated as in a conventional receiver. The other important condition for a successful despreading and consequently for a successful detection is a correct alignment of the received signal and the locally generated PN sequence in the receiver. The simplified block diagram of DSSS/BPSK receiver is shown in Figur 6.19 – block for PN sequence alignment is omitted. In Figure 6.18 it is assumed that both sequences are perfectly aligned. In real world applications the initial alignment of PN sequences (acquisition) and maintaining the alignment (tracking) are complex tasks and one of the main challenges in the application of DS-CDMA techniques.

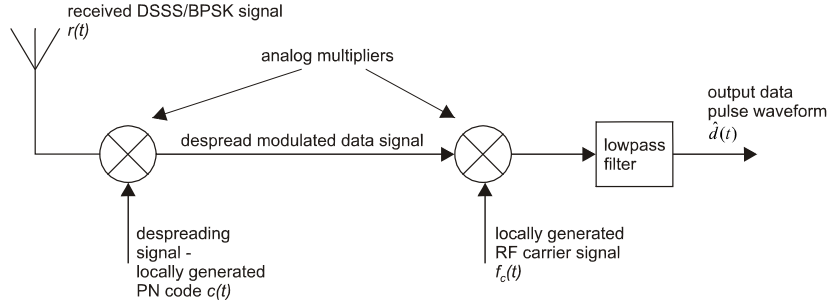


Fig. 6.19: Simplified block diagram of DSSS/BPSK receiver.

Baseband bandwidth of data signal is inversely proportional to bit duration T_b :

$$W_{d0-null} = \frac{1}{T_b} \quad (6.18)$$

Baseband bandwidth of the spreading signal is inversely proportional to chip duration T_c :

$$W_{c0-null} = \frac{1}{T_c} \quad (6.19)$$

Bandpass bandwidth of the DSSS/BPSK signal is:

$$W_{ss} = \frac{2}{T_c} \quad (6.20)$$

Processing gain in cse of DSSS is:

$$P_{PG} = \frac{T_b}{T_c} = \frac{R_c}{R_b} \quad (6.21)$$

Exercise 6.1:

PN signal has rate: 1.2×10^6 [c/s], data rate: 1200 [b/s]. Specify the processing gain in [dB]. (Consider BPSK-DS/SS).

Solution:

$$\text{Bit duration: } T_b = \frac{1}{R_b} = \frac{1}{1200} = 8.33\bar{3} \times 10^{-4} \text{ [s]}$$

$$\text{Chip duration: } T_c = \frac{1}{R_c} = \frac{1}{1200} = 8.33\bar{3} \times 10^{-7} \text{ [s]}$$

$$\text{Processing gain: } P_G = \frac{T_b}{T_c} = \frac{0.833\bar{3} \times 10^{-4}}{8.33\bar{3} \times 10^{-7}} = 10^3 \cong 30 \text{ [dB]}$$

An alternative solution using definition of P_G :

$$W_{ss} = \frac{2}{T_c} \quad R_c = \frac{1}{T_c} \rightarrow W_{ss} = 2R_c$$

$$W_B = \frac{1}{T_b} \quad R_b = \frac{1}{T_b} \rightarrow W_B = R_b$$

$$P_G = \frac{W_{ss}}{2W_b} = \frac{2}{T_c} \times \frac{1}{\frac{2}{T_b}} = \frac{T_b}{T_c} = \frac{R_c}{R_b} = \frac{1.2 \times 10^6}{1.2 \times 10^3} = 1000 \cong 30 \text{ [dB]}$$

■

Exercise 6.2:

DS/SS – BPSK system is supposed to transmit data at rate: $R_b = 10 \text{ [kb/s]}$. A PN sequence has a period: $N = 2047 \text{ [c]}$. Assume that one PN period is used to spread one bit. Determine the $W_{\text{null-to-null}}$ bandwidth of such a system.

Solution:

$$\text{Bit duration: } T_b = \frac{1}{10^4} = 10^{-4} \text{ [s]}$$

$$\text{Chip duration: } T_b = N T_c \Rightarrow T_c = \frac{T_b}{N} = \frac{10^{-4}}{2047} = 4.885 \times 10^{-8} \text{ [s]}$$

$$\text{Bandwidth: } W_{ss(0-0)} = \frac{2}{T_c} = \frac{2}{4.885 \times 10^{-8}} = 40.94 \text{ [MHz]}$$

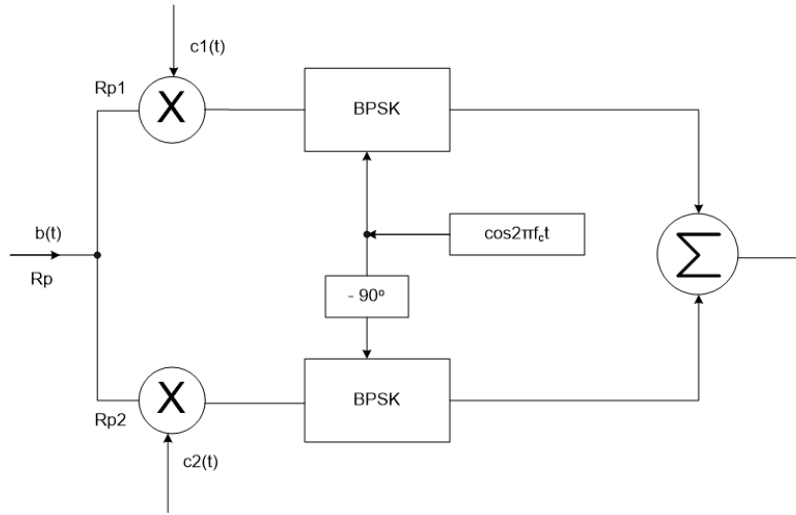
$$P_G = \frac{W_{ss}}{2W_b} \Rightarrow W_{ss} = 2 P_G W_b = 2 \times \frac{T_b}{T_c} \times \frac{1}{T_b} = \frac{2}{T_c} \text{ in case of BPSK baseband bandwidth is: } W_B = \frac{1}{T_b}$$

if: $T = N T_c \Rightarrow P_G = N$, then: $P_G = \frac{T_b}{T_c} = \frac{10^{-4}}{4.885 \times 10^{-8}} = 2047 \cong 33.11 [dB]$

■

Exercise 6.3:

Consider DS/SS-QPSK system as per figure. The chip rate of both spreading sequences is $10^7 [c/s]$, processing gain is $30 [dB]$. Calculate the system bit rate. What would be the advantage of such a system comparing it to a conventional DS/SS-BPSK system with the same parameters?



Solution:

$$P_G = 30 [dB] \cong 10^3$$

$$P_G = \frac{T}{T_c} = 10^3 \Rightarrow T = 10^3 \times T_c = 10^3 \times 10^{-7} = 10^{-4} [s]$$

$$R_{p1} = \frac{1}{T} = 10 kbps = R_{p2} = R_p$$

The bit rate (and bandwidth) is the same as for DS/SS-BPSK systems, but the output of the demodulator: $z_i = AT$, which is twice that of BPSK, thus reducing the error rate of the system.

■

Exercise 6.4:

Let the DS/SS-BPSK system transmit a speech signal that is sampled at a frequency $f_v = 8 [kHz]$, using $k = 8$ bins per sample. At a frequency $f_c = 1.9 [GHz]$, system bandwidth is $W_{ss} = 80 [MHz]$. We assume that one period of the PN sequence is used to spread one binit. Determine:

- the maximum processing gain in [dB]
- what is the maximum usable chip rate of spreading PN sequence?

Solution:

a)

$$R_b = 8 \times 8 \times 10^3 = 64 \times 10^3 [b / s] \rightarrow T_b = \frac{1}{R_b} = \frac{1}{64 \times 10^3} = 15.625 \times 10^{-6} [s]$$

$$W_{ss} = \frac{2}{T_c} \Rightarrow T_c = \frac{2}{W_{ss}} = \frac{2}{80 \times 10^6} = 2.5 \times 10^{-8} [s]$$

$$P_G = \frac{T_b}{T_c} = \frac{15.625 \times 10^{-6}}{2.5 \times 10^{-8}} = 625 \cong 27.958 [dB]$$

b)

$$R_c = \frac{1}{T_c} = \frac{1}{2.5 \times 10^{-8}} = 40 \times 10^6 [c / s], \text{ since } R_c = \frac{1}{T_c} = \frac{W_{ss}}{2}$$

■



DSSS/BPSK system is analyzed in [13] in the TIMS experiment: “Spread Spectrum - Analysis of Direct Sequence Spread Spectrum”, Emona TIMS Volume D5 – Basic Spread Spectrum Techniques, experiment: D5-01.



3 channel DSSS CDMA system is modeled in [14] in the TIMS experiment: “CDMA - DSSS 3 Channel Basic System”, Emona TIMS Volume D6 – Advanced Spread Spectrum Experiments, experiment: D6-01.

Exercise 6.5:

DS/SS-BPSK system has a processing gain of $P_G = 2000$. At the receiver input we assume AWGN, let $E_b / N_0 = 20 [dB]$. The mean power of the received signal is $P = 1 [W]$, also let $T_b = NT_c$, where N is the period of used PN sequence. Determine:

- SNR_0 at the output of the detector (integrator) in $[dB]$ (before the decision block)
- SNR_0 if we assume a narrowband jammer with power $P_j = 50 [W]$.

a)

$$SNR = \frac{1}{2} SNR_0 \Rightarrow SNR_0 = 2 \times SNR = 200 \cong 23.01 [dB]$$

b)

$$\text{in case of narrowband jamming: } SNR_{0j} = \frac{A^2 T}{N_0 + P_j T_c} = \frac{A^2}{\frac{N_0}{T} + \frac{P_j}{P_G}}$$

$$\text{in case of BPSK modulation: } P = \frac{A^2}{2} \Rightarrow A^2 = 2 \times P = 2[W]$$

$$\text{without jamming: } SNR_0 = \frac{A^2 T}{N_0} \Rightarrow \frac{N_0}{T} = \frac{A^2}{SNR_0} = \frac{2}{200} = 0.01$$

$$\text{with jamming: } SNR_{0j} = \frac{2}{0.01 + \frac{50}{2000}} = 57.143 \cong 17.57 [dB] \quad \blacksquare$$

Exercise 6.6:

We have a DS/SS-BPSK system that transmits data at a rate of $R_b = 1 [Mb/s]$. The mean power of the narrowband jammer is $P_j = 50 [W]$. AWGN noise is present in the system, whose PSD is $N_0 = 10^{-9} [W/Hz]$. The mean power of the data signal is $P = 1 [W]$. Determine:

- how long a PN sequence must be used so that at the input of decision block of an optimal DS/SS-BPSK receiver is a minimum $SNR_{0j} = 16.03 [dB]$. Assume: $T_b = N T_c$, where N is the period of the used PN sequence.
- the length of shift register necessary to generate PN sequence from a).

Solution:

a)

$$P = \frac{A^2}{2} \Rightarrow A^2 = 2P \quad R_b = \frac{1}{T_b} \Rightarrow T_b = \frac{1}{R_b}$$

$$SNR_{0j} = \frac{A^2 T_b}{N_0 + P_j T_c} = \frac{A^2 T_b}{N_0 + P_j \frac{T_b}{N}} = \frac{A^2 T_b}{\frac{N_0 N + P_j T_b}{N}} = A^2 T_b \frac{N}{N_0 N + P_j T_b}$$

$$SNR_{0j} N N_0 + SNR_{0j} T_b P_j = A^2 T_b N$$

$$N(SNR_{0j} N_0 - A^2 T_b) = -SNR_{0j} T_b P_j$$

$$N = \frac{SNR_{0j} P_j T}{A^2 T - SNR_{0j} N_0} \quad N = \left\lceil \frac{10^{1.603} \times 50 \times 10^{-6}}{2 \times 10^{-6} - 10^{1.603} \times 10^{-9}} \right\rceil = \lceil 1022.66 \rceil = 1023 [c]$$

b)

$$N = 2^m - 1 \Rightarrow m = \log_2(N + 1) = \log_2(1024) = 10$$

■

Exercise 6.7:

What can be the maximum one-sided power density of broadband jammer so that the SNR at the input of an optimal DS/SS-BPSK receiver is -25.64 [dB], assuming that the transmission rate is $R_b = 2$ [Mb/s] and the mean power of the received signal is $P = 300$ [mW]? We consider PSD of AWGN $N_0 = 10^{-6}$ [W/Hz].

Solution:

$$T_b = \frac{1}{R_b} = 5 \times 10^{-7} \text{ [s]} \quad P = \frac{A^2}{2} = 0.3 \text{ [W]}$$

$$SNR_j = \frac{\frac{A^2 T_b}{2}}{N_0 + \frac{P_j}{W_j}}$$

$$SNR_j \left(N_0 + \frac{P_j}{W_j} \right) = P T_b$$

$$\frac{P_j}{W_j} = \frac{P T_b - SNR_j N_0}{SNR_j} = \frac{0.3 \times 5 \times 10^{-7} - 10^{-2.564} \times 10^{-6}}{10^{-2.564}} = 5.396 \times 10^{-5} \left[\frac{\text{W}}{\text{Hz}} \right]$$

■

Exercise 6.8:

DS/SS-BPSK system has bit rate $R_b = 9.6$ [kb/s], its bandwidth is $W_{ss} = 96$ [MHz]. Mean received power of the useful signal $P = 28.8$ [μ W], PSD of AWGN at the receiver input is $N_0 = 10^{-9}$ [W/Hz]. Mean power of broadband jammer $P_j = 10$ [W]. Determine:

- the probability of error per bit P_b - without jamming
- probability of error per bit P_{bj} - with jammer
- processing gain in [dB]

Solution:

a)

we assume a coherent BPSK modulation:

$$E_b = \frac{P}{R_b} = \frac{28.8 \times 10^{-6}}{9.6 \times 10^3} = 3 \times 10^{-9} [J]$$

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) = Q\left(\sqrt{\frac{2 \times 3 \times 10^{-9}}{10^{-9}}}\right) = Q(\sqrt{6}) = Q(2.449) \cong 0.0071$$

b)

in case of broadband jammer: $W_{ss} = W_j$, then:

$$J_0 = \frac{P_j}{W_j} \quad P_b = Q\left(\sqrt{\frac{2E_b}{J_0 + N_0}}\right) = Q\left(\sqrt{\frac{2 \times 3 \times 10^{-9}}{10^{-9} + \frac{10}{96 \times 10^6}}}\right) = Q(0.2388) = 0.4052$$

c)

$$W_{ss} = \frac{2}{T_c} \Rightarrow T_c = \frac{2}{W_{ss}}$$

$$T = \frac{1}{R_b}$$

$$P_G = \frac{T}{T_c} = \frac{1}{R_b} \frac{W_{ss}}{2} = \frac{96 \times 10^6}{2 \times 9.6 \times 10^3} = 5 \times 10^3 \Rightarrow \cong 37[dB]$$

■

Exercise 6.9:

c) What has to be the minimal chip rate so in DS/SS-BPSK system is eliminated the reflected path which is by 15 [m] longer than direct path?

d) Calculate the bandwidth of DS/SS-BPSK from a).

Solution:

a)

The time it takes for the signal to overcome the distance difference: $\Delta l = l_2 - l_1$ is:

$$\tau = \frac{l_2 - l_1}{c} = \frac{15}{300 \times 10^6} = 5 \times 10^{-8} [s]$$

In order to suppress the reflected path (decorrelation): $T_c \leq \tau \Rightarrow R_c \geq \frac{1}{T_c} = 2 \times 10^7 [c/s]$

b)

$$W_{ss} = \frac{2}{T_c} = \frac{2}{5 \times 10^{-8}} = 40 [MHz]$$

■

Exercise 6.10:

The maximum value of the autocorrelation function of the narrowband jammer is 2.5 [W] (already after recalculation over the symbol interval). What can be the maximum transmission rate of a DS/SS- BPSK system if the period of the used PN sequence $N = 1023$ [c], while the total power of the jammer that enters the detection process is $E\{j_0^2\} = 7.4578 \times 10^{-14}$ [W]? The $E\{.\}$ is the expected value operator. Assume narrowband jammer and that the jammer knows our system (i.e., $f_j - f_c$ is small).

Solution:

Jamming power: $P_j = R_j(0) = 2.5$ [W]

$$E\{j_0^2\} = \frac{P_j T_b^2}{4N} \Rightarrow T_b = \sqrt{\frac{E\{j_0^2\} 4N}{P_j}} = \sqrt{\frac{7.4578 \times 10^{-14} \times 4 \times 1023}{2.5}} = 1.105 \times 10^{-5} \Rightarrow R_b = 90.498 [kb / s]$$

■

➡ Processing gain measurement of baseband DSSS/BPSK system is analyzed in [14] in the TIMS experiment: “DS SS Baseband System – Processing Gain Measurement”, Emona TIMS Volume D6 – Advanced Spread Spectrum Experiments, experiment: D6-02.

➡ ISI rejection in DSSS/BPSK system is analyzed in [15] in the TIMS experiment: “ISI Rejection in DS SS”, Emona TIMS Volume D8 – Multipath Experiments, experiment: D8-02.

Exercise 6.11:

What must be the minimum difference of neighboring tones in baseband at the output of the BFSK modulator for the FH/SS –BFSK system in the case of:

a) non-coherent detection

b) coherent detection

Solution:

$$\text{hop time: } T_h = 25 [\mu s] \quad \text{bit time: } T_b = \frac{1}{R_b} = \frac{1}{20 \times 10^3} = 50 [\mu s]$$

since: $T_h < T_b = T_s \Rightarrow \text{FFH}$

$$\text{a) } \Delta f_{\min} = \frac{1}{T_h} = \frac{1}{25 \times 10^{-6}} = 40 [kHz]$$

$$b) \Delta f_{\min} = \frac{1}{2T_h} = \frac{1}{2 \times 25 \times 10^{-6}} = 20 \text{ [kHz]}$$

Note:

$$\text{non-coherent detection: } \Delta f = \frac{1}{T_c}$$

$$\text{coherent detection: } \Delta f = \frac{1}{2T_c}, \text{ where: } T_c = \begin{cases} T_s & \text{SFH} \\ T_h & \text{FFH} \end{cases}$$

■

Exercise 6.12:

The non-coherent BFSK-FFH/SS system with 5 hops per bit has a bit rate $R_b = 2500 \text{ [b/s]}$. The system uses $J = 1023$ carrier frequencies. Calculate:

a) clock rate: R_H of PN generator

b) system bandwidth W_{ss}

Solution:

a)

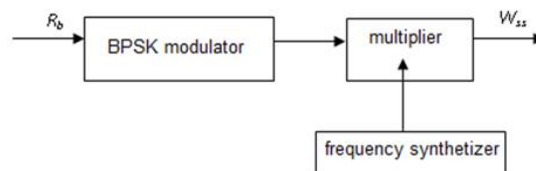
$$\text{BFSK : } \Rightarrow T_b = \frac{1}{R_b} = \frac{1}{2500} = 0.4 \text{ [ms]}$$

$$5 \text{ hops per bit (5-fold diversity)} \Rightarrow \text{hop time is: } T_h = \frac{T_b}{5} = \frac{0.4 \times 10^{-3}}{5} = 8 \times 10^{-5} \text{ [s]}$$

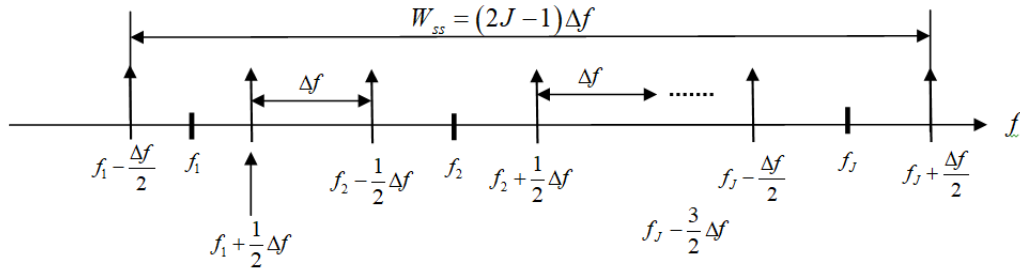
$$\text{clock rate is: } R_H = \frac{1}{T_h} = \frac{1}{8 \times 10^{-5}} = 12500 \text{ [c / s]}$$

b)

In case that we assume the following block diagram:



At the same time we assume that the individual subbands are closely adjacent and do not overlap (there is Δf distance between tones):



then: $W_{ss} = (2J - 1)\Delta f = (2 \times 1023 - 1) \times 12500 \cong 25.5625 \text{ [MHz]}$

■

Exercise 6.13:

8FSK FFH/SS transmits data at a rate of $R_p = 4500 \text{ [b/s]}$. The system uses 1023 carrier frequencies. The system uses 5-fold diversity (i.e., 5 hops per symbol). Assume non-coherent detection. Determine:

- the bandwidth of the system assuming mixing synthesized frequency and modulation tones as in Exercise 6.10.
- what will be the bandwidth of the system if the system turns into a SFH with 1 hop per 2 symbols?

Solution:

a)

8FSK: $M = 8 \Rightarrow k = 3$

modulation rate: $R_M = \frac{R_p}{k} = \frac{4500}{3} = 1500 \text{ [s / s]}$

symbol time: $T_s = \frac{1}{R_M} = \frac{1}{1500} = 6.66\bar{6} \times 10^{-4} \text{ [s]}$

hop time: $T_h = \frac{T_s}{5} = \frac{6.66\bar{6} \times 10^{-4}}{5} = 1.33\bar{3} \times 10^{-4} \text{ [s]}$

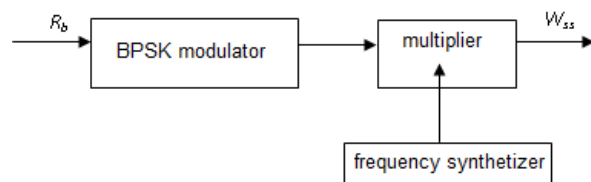
tone spacing: $\Delta f = \frac{1}{T_h} = R_h = 7500 \text{ [Hz]}$

$W_{ss} = (M \times J - 1)\Delta f = (8 \times 1023 - 1) \times 7500 \cong 61.3725 \text{ [MHz]}$

b)

SFH system: $T_h = 2T_s = \frac{2}{1500} = 1.33\bar{3} \times 10^{-3} \text{ [s]}$

$T_c = T_s$



$$\Delta f = \frac{1}{T_c} = 1500 \text{ [Hz]} \quad W_{ss} = (M \times J - 1) \Delta f = (8 \times 1023 - 1) \times 1500 \cong 12.2745 \text{ [MHz]}$$

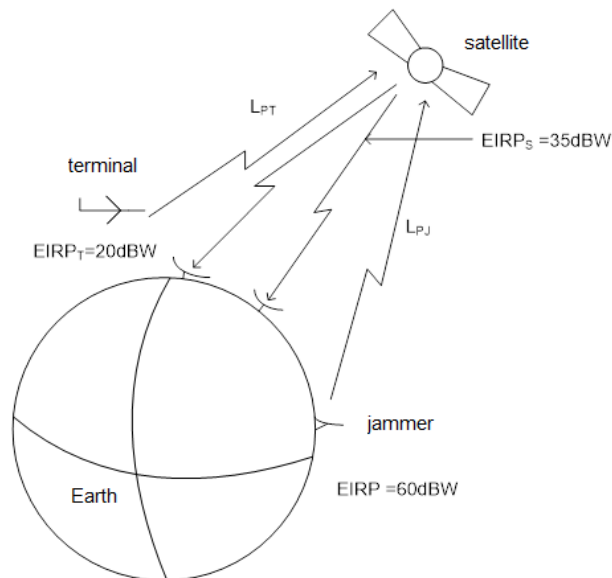
■

- ➡ Modeling of CDMA FHSS/BFSK system is analyzed in [14] in the TIMS experiment: “FH-CDMA/ BFSK”, Emona TIMS Volume D6 – Advanced Spread Spectrum Experiments, experiment: D6-03.
- ➡ Analyzing of FHSS/BFSK system is analyzed in [13] in the TIMS experiment: “Spread Spectrum - Analysis of Frequency Hop Spread Spectrum”, Emona TIMS Volume D5 – Basic Spread Spectrum Techniques, experiment: D5-02.

Exercise 6.14:

The terminal (aircraft) uses the FH/SS system to communicate with ground terminals via satellite. $EIRP_T = 20 \text{ [dBW]}$. Transmission rate $R_b = 100 \text{ [b/s]}$. The jammer transmits broadband AWGN, continuously with $EIRP_j = 60 \text{ [dBW]}$. Assume that required $(E_b / J_0)_{req} = 10 \text{ [dB]}$ and that the signal propagation losses with respect to the satellite are the same for both the aircraft and the jammer ($L_{PT} = L_{PJ}$). Determine:

- from the communicator's point of view, which is worse interference UL or DL
- if $M_{AJ} = 20 \text{ [dB]}$ is required, what must be the system bandwidth W_{ss}



Solution:

a)

From the communicator's point of view, UL jamming is much worse, because such "focused" jamming can degrade the communication of a number of terminals that simultaneously use the same transponder. (The jammer transmits a narrow beam directly to the satellite). To achieve the same effect for DL jamming, the jammer would have to jam each receiving station separately.

b)

If we assume: $L_{PJ} = L_{PT}$, then:

$$J = \frac{EIRP_J}{L_P} \quad S = \frac{EIRP_T}{L_P} \quad \left(\frac{J}{S} \right)_{received} = \frac{EIRP_J}{EIRP_T}$$

$$M_{AJ} [dB] = \left(\frac{J}{S} \right)_{req.} [dB] - \left(\frac{J}{S} \right)_r [dB]$$

$$M_{AJ} [dB] = \left(\frac{J}{S} \right)_{req.} [dB] - EIRP_J [dBW] + EIRP_T [dBW] =$$

$$= P_G [dB] - \left(\frac{E_b}{J_0} \right)_{req.} [dB] + EIRP_T [dBW] - EIRP_J [dBW] \Rightarrow$$

$$P_G [dB] = M_{AJ} [dB] + \left(\frac{E_b}{J_0} \right)_{req.} [dB] - EIRP_T [dBW] + EIRP_J [dBW] =$$

$$= 20 + 10 - 20 + 60 = 70 [dB]$$

$$P_G = \frac{W_{ss} S}{R_b} \Rightarrow W_{ss} S = P_G R_b \rightarrow W_{ss} [dBHz] = P_G [dB] + R_b [dBHz] = 70 + 20 = 90 [dBHz] \cong 1 [GHz]$$

If we consider a system with an additional multiplier, then: $W_{ss} = 2 P_G R_b = 2 [GHz]$

■

Exercise 6.15:

FH/SS system is using non-coherent BFSK modulation. System parameters are: processing gain is: $P_G = 6400$, bit rate is: $R_b = 64 [kb/s]$, $E_b / J_0 = 10$, Determine:

- the bandwidth of the jammer so that the bit error rate probability of the communicator is maximal
- P_{bmax} for given parameters

Solution:

In case of non-coherent BFSK bit error probability can be approximated as: $P_b = \frac{\rho}{2} \exp\left[-\rho \frac{E_b}{2J_0}\right]$

where: $\rho = \frac{W_J}{W_{ss}} \Rightarrow \rho \in \langle 0,1 \rangle$

Therefore it is necessary to determine $\rho = \rho_0$ so the bit error probability P_b is maximal:

$$\frac{\partial P_b}{\partial \rho} = \frac{1}{2} \exp\left(-\rho \frac{E_b}{2J_0}\right) + \frac{\rho}{2} \exp\left(-\rho \frac{E_b}{2J_0}\right) \left(-\frac{E_b}{2J_0}\right) = \frac{1}{2} \exp\left(-\rho \frac{E_b}{2J_0}\right) \left[1 - \rho \frac{E_b}{J_0}\right] = 0$$

$$1 - \rho \frac{E_b}{J_0} = 0 \Rightarrow \rho = \begin{cases} \frac{2}{\frac{E_b}{J_0}} & \text{if } \frac{E_b}{J_0} > 2 \\ 1 & \text{if } \frac{E_b}{J_0} \leq 2 \end{cases}$$

$$\rho = \frac{2J_0}{E_b} \frac{J_0}{1} = \frac{2}{\frac{E_b}{J_0}}$$

$$\text{In our case: } \frac{E_b}{J_0} = 10 \Rightarrow \rho = \frac{2}{\frac{E_b}{J_0}} = 0.2$$

$$\rho = \frac{W_J}{W_{ss}} \Rightarrow W_J = \rho W_{ss} = \rho P_G R_b = 0.2 \times 6400 \times 64 \times 10^3 = 81.92 \text{ [MHz]}$$

$$W_{ss} = 6400 \times 64 \times 10^3 = 409.6 \text{ [MHz]}$$

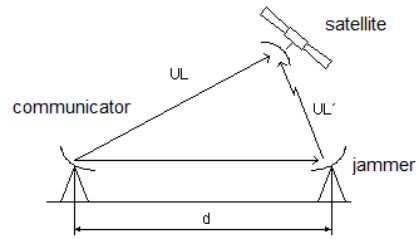
b)

$$\frac{E_b}{J_0} > 2 \Rightarrow P_b = \frac{e^{-1}}{\frac{E_b}{J_0}} = \frac{e^{-1}}{10} = 0.03678$$

■

Exercise 6.16:

The repeat-back jammer is $d = 30 \text{ [km]}$ away from the communicator and can monitor UL traffic to the satellite. How fast must the communicator change frequency to cancel the effect of the repeat-back jamming? Assume that the jammer is able to change frequency instantaneously and that the time difference between the UL signal of the communicator and the UL signal of the jammer is the propagation time from the communicator to the jammer.



Solution:

In order for the frequency of the communicator and the jammer not to overlap in time, the following

must apply: $T_H \leq \frac{d}{c}$ $T_H \leq \frac{30 \times 10^3}{3 \times 10^8} = 10^{-4} [s] \Rightarrow R_H \geq 10^4 [\text{hops} / s]$

■

Chapter 7: Communication channels

BSC channel

The Binary Symmetric Channel describes the data transmission as an independent series of transmissions of individual bits. Because of the independence, a single bit transmission is sufficient for the analysis of BSC channel properties. The BSC channel is completely described by the error probability q – a conditional probability that a transmitted bit zero gets corrupted to a received bit one, and vice versa. The transmitter (TX) and receiver (RX) are then described by discrete random variables, denoted X and Y in Fig. 7.1:

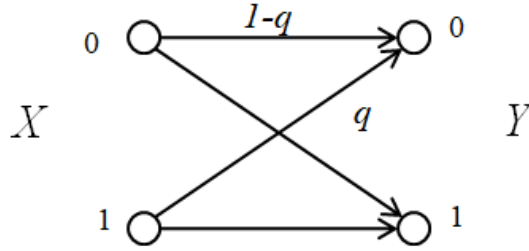


Fig. 7.1: The BSC channel model.

As shown in the figure, $1 - q$ is the conditional probability of successful transmission. The BSC channel may be also modelled in a matrix form:

$$Q = \begin{bmatrix} 1-q & q \\ q & 1-q \end{bmatrix}$$

The binary transmitter is described by an alternative probability distribution of binary values zero and one:

X_i	0	1
p_i	p	$1 - p$

while the same applies for the binary receiver:

Y_i	0	1
p_i	u	$1 - u$

The discrete probability distribution of Y is completely determined by the parameters of the transmitter, and BSC channel. We can use the law of total probability:

$$p(y) = \sum_{\forall x} p(x) \cdot p(y / x) \equiv p(Y_j) = \sum_{i=1}^N p(X_i) \cdot p(Y_j / X_i) \quad (7.1)$$

by applying it to the binary alternative distribution of X and conditional probabilities of the BSC channel:

$$u \equiv p(Y = 0) = p(X = 0) \cdot p(Y = 0 / X = 0) + p(X = 1) \cdot p(Y = 0 / X = 1)$$

$$u \equiv p(Y = 0) = p_0(1 - q) + p_1q = p(1 - q) + (1 - p)q$$

$$p(Y = 1) \equiv 1 - u = pq + (1 - p)(1 - q)$$

Joint probability distribution of the BSC channel

We obtain the joint probability distribution of a random vector (X, Y) consisting of two discrete random variables X and Y representing the transmitter and receiver in the following way:

The all possible combinations of values are given by the Cartesian product of all values of X and Y and we calculate their probabilities by using the conditional error probabilities of the channel as shown by the equation below:

$$p(x, y) = p((X = x) \vee (Y = y)) = p(X = x) \cdot p(Y = y / X = x) = p(x) \cdot p(y / x)$$

For a BSC channels both X and Y may only have two values: 0 and 1, which make for 4 different combinations, shown in Table 7.1:

$Y \setminus X$	0	1
0	$p(0,0) = p_0 \cdot p(0/0)$	$p(1,0) = p_1 \cdot p(0/1)$
1	$p(0,1) = p_0 \cdot p(1/0)$	$p(1,1) = p_1 \cdot p(1/1)$

Table 7.1: The joint probability distribution for the input and output of a BSC channel.

If the distributions of both X and Y is known, we must be careful and resist the temptation to fill the table by simply multiplying the probabilities: this would be correct if we know that X and Y are independent random variables, which in the context of data transmission would mean a channel with no capacity. We will elaborate in the examples below.

Table 7.1 completely describes the transmission and if it's known, we can calculate the probability distributions of both X and Y by the process of *marginalization*:

$$\begin{aligned} p(x) &= \sum_{\forall y} p(x, y) \\ p(y) &= \sum_{\forall x} p(x, y) \end{aligned} \tag{7.2}$$

If we apply the marginalization equations, we may extend Table 7.1 to also include the distributions of X and Y , see Table 7.2.

$Y \setminus X$	0	1	$p(Y)$
0	$p(0,0)$	$p(1,0)$	$p(0,0) + p(1,0)$
1	$p(0,1)$	$p(1,1)$	$p(0,1) + p(1,1)$
$p(X)$	$p(0,0) + p(0,1)$	$p(1,0) + p(1,1)$	

Table 7.2: Joint probability distribution and marginalizations.

What are important for the receiver design are actually the *posterior* probabilities:

$$p(x / y) = \frac{p(x, y)}{p(y)} \tag{7.3}$$

We will discuss them in more detail in later chapters.

Exercise 7.1:

A digital communication system transmitter transmits the symbols 0 and 1 with probabilities 0.6 and 0.4. A BSC channel introduces an error with a probability 0.3. Calculate:

- The Q matrix describing the BSC channel.
- Joint probability distribution of transmitter and receiver (X, Y).
- The probability distribution of Y using:
 - Marginalization
 - The law of total probability
- The probability of transmitting the symbol 0, given a received symbol 0
- The probability of transmitting the symbol 1, given a received symbol 0

Solution:

a)

$$Q = \begin{bmatrix} 0.7 & 0.3 \\ 0.3 & 0.7 \end{bmatrix}$$

b)

$Y \setminus X$	0	1	$p(Y)$
0	$0.6 \times 0.7 = 0.42$	$0.4 \times 0.3 = 0.12$	$0.42 + 0.12 = 0.54$
1	$0.6 \times 0.3 = 0.18$	$0.4 \times 0.7 = 0.28$	$0.18 + 0.28 = 0.46$
$p(X)$	$0.42 + 0.18 = 0.6$	$0.12 + 0.28 = 0.4$	Sum = 1

c)

- The marginalization is shown above in the last column of the table.
- Using the law of total probability:

$$\begin{aligned} p(Y=0) &= p(X=0) \cdot p(Y=0/X=0) + p(X=1) \cdot p(Y=0/X=1) = \\ &= 0.6 \times 0.7 + 0.4 \times 0.3 = 0.42 + 0.12 = 0.54 \end{aligned}$$

$$\begin{aligned} p(Y=1) &= p(X=1) \cdot p(Y=1/X=1) + p(X=0) \cdot p(Y=1/X=0) = \\ &= 0.4 \times 0.7 + 0.6 \times 0.3 = 0.28 + 0.18 = 0.46 \end{aligned}$$

$$d) \ p(X=0/Y=0) = \frac{p(X=0, Y=0)}{p(Y=0)} = \frac{0.42}{0.54} = \frac{7}{9} = 0.78$$

$$e) \ p(X=1/Y=0) = \frac{p(X=1, Y=0)}{p(Y=0)} = \frac{0.12}{0.54} = \frac{2}{9} = 0.\bar{2}$$

■

Exercise 7.2:

The same assignment as in exercise 7.1, but:

- A. Using a completely unreliable BSC channel.
- B. Using a completely reliable BSC channel.

Solution:

A.

$$a) \quad Q = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$$

b), c)

As shown in Table 7.2, we calculate the probability mass functions of the random variables X and Y by marginalization (7.2) – by summing all the probabilities in each row to get $p(Y)$, and summing all the probabilities in each column to get $p(X)$. In the table below, we can see that for every possible value of the random vector (X, Y) , it is true that $P(X, Y) = P(X) \cdot P(Y)$. Therefore the random variables X and Y are *independent* and no data transmission is actually possible.

$Y \setminus X$	0	1	$p(Y)$
0	$0.6 \times 0.5 = 0.3$	$0.4 \times 0.5 = 0.2$	$0.3 + 0.2 = 0.5$
1	$0.6 \times 0.5 = 0.3$	$0.4 \times 0.5 = 0.2$	$0.3 + 0.2 = 0.5$
$p(X)$	$0.3 + 0.3 = 0.6$	$0.2 + 0.2 = 0.4$	$Sum = 1$

The independence of X and Y is also demonstrated by the definition of independence, i.e. by the conditional probability $p(X/Y)$ being not affected by $p(Y)$:

$$d) \quad P(X = 0/Y = 0) = 0.3 / 0.5 = 0.6 = P(X = 0)$$

$$e) \quad P(X = 1/Y = 0) = 0.2/0.5 = 0.4 = P(X = 1)$$

B.

$$a) \quad Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

b), c)

$Y \setminus X$	0	1	$p(Y)$
0	$0.6 \times 1 = 0.6$	$0.4 \times 0 = 0$	$0.6 + 0 = 0.6$
1	$0.6 \times 0 = 0$	$0.4 \times 1 = 0.4$	$0 + 0.4 = 0.4$
$p(X)$	$0.6 + 0 = 0.6$	$0 + 0.4 = 0.4$	$Sum = 1$

As expected, a completely reliable channel makes Y a deterministic function of X . This is show also when anumerating the posterior probabilitites.

$$d) \quad P(X = 0/Y = 0) = 0.6 / 0.6 = 1$$

$$e) \quad P(X = 1/Y = 0) = 0 / 0.6 = 0$$

AWGN Channel

Natural - thermal (Johnson) noise is caused by the thermal movement of electrons and therefore cannot be eliminated from electrical devices. Good design can minimize its impact. It can be described by a Gaussian distribution with zero mean. An analytical model of thermal noise is AWGN, which has the following parameters:

$$G_n(f) = \frac{N_0}{2} \left[\frac{W}{Hz} \right] \rightarrow \text{double sided PSD} \quad (7.4)$$

$$G_n(f) = N_0 \left[\frac{W}{Hz} \right] \rightarrow \text{single sided PSD} \quad (7.5)$$

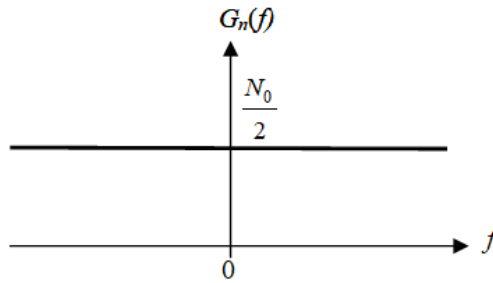


Fig. 7.2: PSD of an AWGN noise.

$$R_n(\tau) = FT^{-1}\{G_n(f)\} = \frac{N_0}{2} \delta(\tau) \rightarrow \text{any two samples of AWGN are uncorrelated} \quad (7.6)$$

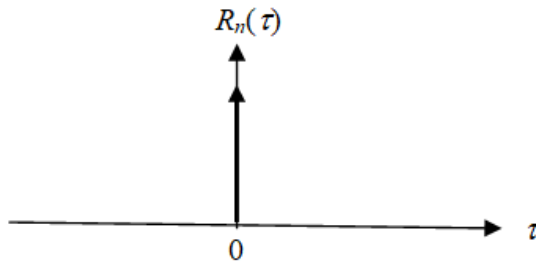


Fig. 7.3: Autocorrelation of an AWGN noise.

$$\text{Mean power: } P_n = \int_{-\infty}^{\infty} G_n(f) df = \infty \rightarrow \text{AWGN has an infinite bandwidth} \quad (7.7)$$

Normal distribution:

The random variable X has a normal probability distribution, denoted $N(a, \sigma^2)$ with parameters: μ - mean, σ^2 - variance if its probability density function (pdf) has the form:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (7.8)$$

The distribution (error) function $F(x)$ expresses the probability that the random variable X with $N(\mu, \sigma^2)$ takes the value $\leq x$:

$$F(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt \quad (7.9)$$

In case of normalized normal distribution with parameters: $N(0,1)$ the pdf has the form:

$$f(x)_{N(0,1)} = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \quad (7.10)$$

The distribution function of normalized normal distribution is then:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \quad (7.11)$$

The distribution function of normalized normal distribution is in form of tables or is programmed in e.g. Excel, Mathematica or MATLAB in the form:

$$\text{erf}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{t^2}{2}} dt \quad (7.12)$$

If we want to use this function or tables for normal distribution with $\mu \neq 0$, $\sigma^2 \neq 1$ we have to modify the argument of the distribution function:

$$F(x) = \Phi\left(\frac{x-\mu}{\sigma}\right) \quad (7.13)$$

Co-error function $Q(x)$ expresses the probability that the random variable X with $N(\mu, \sigma^2)$ takes the value $> x$:

$$Q(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_x^{\infty} e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt \quad (7.14)$$

Its normalized version has the form:

$$Q_n(x) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} e^{-\frac{t^2}{2}} dt \quad (7.15)$$

This function is also expressed in tables or is programmed in the form:

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt \quad (7.16)$$

Relation between $\text{erfc}(x)$ and $Q(x)$:

$$\begin{aligned} \text{erfc}(x) &= 2Q\left(x\sqrt{2}\right) \\ Q(x) &= \frac{1}{2} \text{erfc}\left(\frac{x}{\sqrt{2}}\right) \end{aligned} \quad (7.17)$$

If we want to use this function or tables for normal distribution with $\mu \neq 0$, $\sigma^2 \neq 1$ we have to modify the argument of the distribution function:

$$Q(x) = Q_n\left(\frac{x-\mu}{\sigma}\right) \quad (7.18)$$

Other important relationships:

$$\begin{aligned} Q(x) + \Phi(x) &= 1 \\ \Phi(x) &= 1 - \Phi(-x) \Rightarrow \Phi(-x) = Q(x) \end{aligned} \quad (7.19)$$

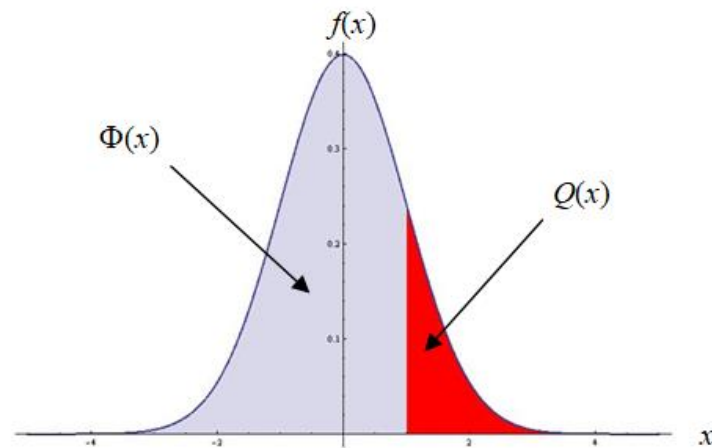


Fig. 7.4: The relationship between probability density $f(x)$, cumulative distribution function $\Phi(x)$, and the complementary error function $Q(x)$.

Exercise 7.3:

AWGN is defined by its variance: $\sigma_0^2 = 10^{-7}$. Calculate the probability that the voltage noise level will be in the interval:

a) $\langle +100, +500 \rangle [\mu V]$

b) $\langle +1, +3 \rangle [mV]$

Solution:

AWGN has a normal distribution; it means that the probability that the voltage noise level u_n is greater than some value of x is:

$$P(u_n > x) = Q(x)$$

If we want to calculate the probability that: $u_n \in \langle a, b \rangle$, then:

$$P(a \leq u_n \leq b) = P(u_n \geq a) - P(u_n \geq b) = Q(a) - Q(b) \rightarrow \text{see Fig. 7.5.}$$

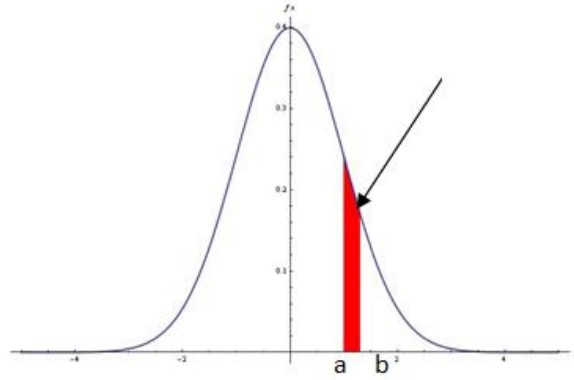


Fig. 7.5: Calculating probability by integrating probability density function.

Since: $\sigma_0^2 \neq 1 \Rightarrow$ we have to modify the argument of Q function:

a)

$$\begin{aligned} P(+100[\mu V] \leq u_n \leq +500[\mu V]) &= Q\left(\frac{10^{-4}}{\sqrt{10^{-7}}}\right) - Q\left(\frac{5 \times 10^{-4}}{\sqrt{10^{-7}}}\right) = Q(0.316) - Q(1.58) \\ &= 0.3745 - 0.0571 = 0.3174 \end{aligned}$$

b)

$$\begin{aligned} P(+1[mV] \leq u_n \leq +3[mV]) &= Q\left(\frac{10^{-3}}{\sqrt{10^{-7}}}\right) - Q\left(\frac{3 \times 10^{-3}}{\sqrt{10^{-7}}}\right) = Q(3.16) - Q(9.486) \\ &= 0.0008 - 1.203 \times 10^{-21} \cong 0.0008 \end{aligned}$$

For larger arguments ($x > 3$) we can use an approximation e.g.: $Q(x) \cong \left(\frac{1}{x\sqrt{2\pi}}\right)e^{\left(-\frac{x^2}{2}\right)}$ ■



Basic properties of generated AWGN are analyzed in [16] in the TIMS experiment: “Random Variable Analysis: AWGN and erfc”, Emona TIMS Signals & Systems Advanced Experiments – Volume 2, experiment: S&S Vol.2-06.

Lab 7.1: Random number generation, random noise generation, properties of AWGN

Many properties of the AWGN were mentioned in the theoretical section of this chapter. The actual definition of AWGN in literature looks, however, often quite different. It may be summarized in the following way: AWGN is a wide-sense stationary normally distributed ergodic stochastic process that is completely defined by its first and second moment: the mean μ and variance σ^2 . Sometimes, especially in the context of digital communications systems, the usual real-valued normal probability distribution $N(\mu, \sigma^2)$ is not sufficient and a complex-valued $CN(\mu, \sigma^2)$ must be used, introducing additional mathematics. Before going into these details, it is interesting to revisit the theoretical properties of AWGN and their formulation: The autocorrelation function $R_n(\tau)$ of AWGN show in Fig. 7.3 is a Dirac impulse indicating the complete lack of correlation between any two different time points. One of the mathematical properties of the autocorrelation function is that its Fourier transform is the AWGN noise Power Spectral Density (PSD) $G_n(f)$ shown in Fig. 7.2 as a constant line. This is compatible with the common knowledge that a white light consists of all the colors equally distributed, which translates directly to equal distribution of all frequencies within a white noise signal. The theory looks fine until we start to play with MATLAB and try to approximate the AWGN process by using built-in random number generators, such as the `randn()` function. If we use it to simply generate a vector of samples and then calculate an estimated $R_n(\tau)$ and $G_n(f)$ by calling `xcov()` (or `xcor()`) and `fft()` functions, after we plot the results we will be surprised by a lack of similarity of the plots in Fig. 7.6 below with figures 7.2 and 7.3.

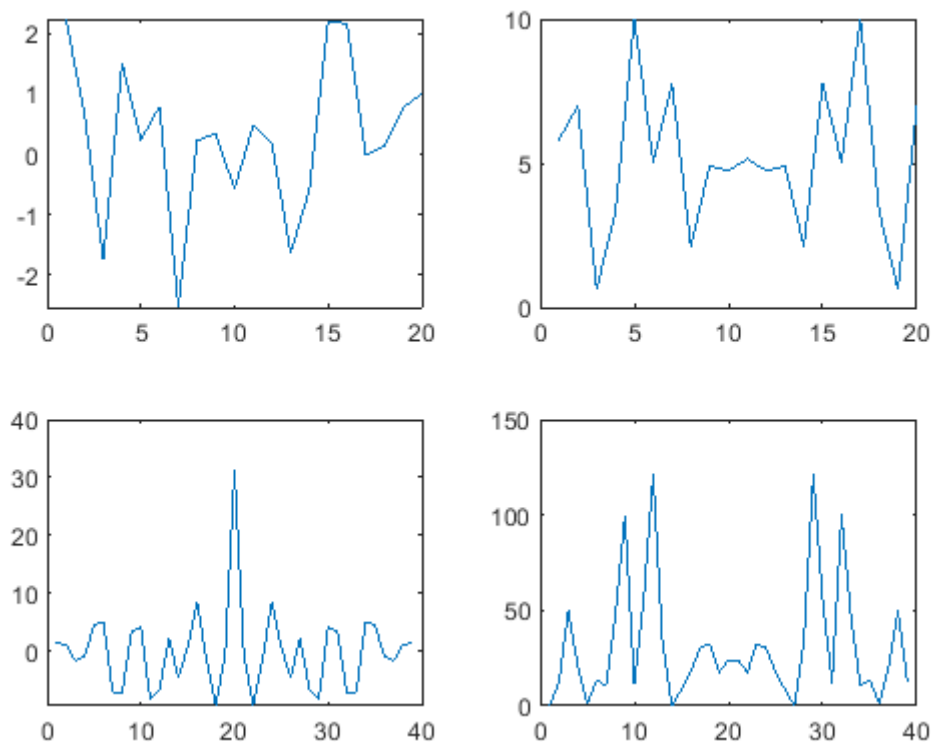


Fig. 7.6: Naïve Fourier analysis attempt: One realization of a random process (top left), its spectrum obtained by `fft()` (top right), autocorrelation $R_n(\tau)$ (bottom left), and PSD $G_n(f)$ (bottom right).

Obviously, when using the pseudo-random generators in MATLAB we must pay more attention to the details. First it is necessary to understand the difference between probabilistic and statistical formulation of the equation that define the basic *moments* we use to evaluate for (pseudo) random signals. For the purposes of this chapter we will be interested only in selected first and second moments: mean, variance and covariance.

As described in Chapter 4, the only option we have on a computer is to work with discrete-time and discrete-frequency signals represented by vectors or matrices, and we use the analogies between discrete and continuous form to infer properties of signals coming to and from a computer through some converter. It would be perhaps more appropriate to plot the signals in Fig. 7.6 using the `stem()` function to emphasize their discrete character:

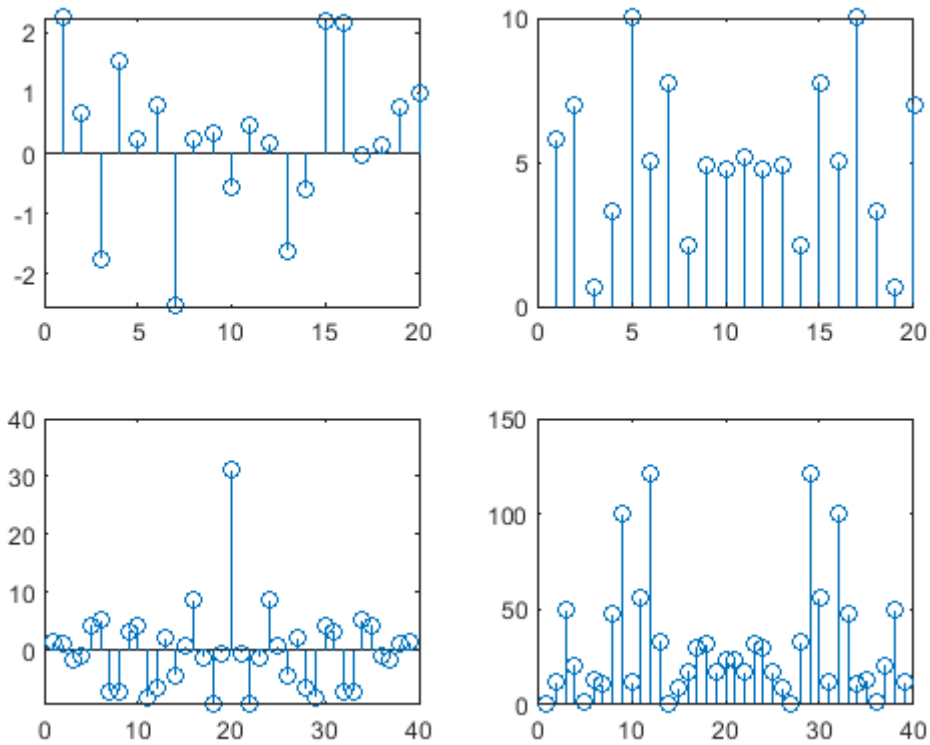


Fig. 7.7: Naïve Fourier analysis attempt – the same signals as in Fig. 7.6 but plotted as discrete-time (or discrete frequency) signals – vectors.

In the probabilistic view, we define the first moment of a random variable X separately for discrete random variables with probability mass function and continuous random variables with probability densities:

$$\text{Expected value or mean: } \mu \equiv m_x \equiv E[X] = \sum_{i=1}^{\infty} x_i p_i \text{ discr. } E[X] = \int_{-\infty}^{\infty} x f(x) dx \text{ cont.} \quad (7.20)$$

Then we can simply define selected second moments for random variables X and Y :

$$\text{Variance or dispersion: } \sigma^2 \equiv \text{Var}(X) = E[(X - \mu)^2] = E[X^2] - E[X]^2 \quad (7.21)$$

$$\text{Covariance: } \sigma_{XY} \equiv \text{cov}(X, Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y] \quad (7.22)$$

We can also define μ in a common formula for both discrete and continuous variables (and mixed ones) by using Lebesgue integration, but let's keep things simple for now. As shown in (7.20), these theoretical definitions often deal with infinite number of elements, or uncountable continuum, and will therefore be of little use for our practical real-world examples that always deal with finite and even quite small datasets.

Let us now summarize some important and often used properties of these moments:

$$E[X + Y] = E[X] + E[Y] \quad (7.23)$$

$$E[aX] = aE[X] \quad (7.24)$$

$$\text{Var}(X) \geq 0 \quad (7.25)$$

$$\text{Var}(X + a) = \text{Var}(X) \quad (7.26)$$

$$\text{Var}(aX) = a^2 \text{Var}(X) \quad (7.27)$$

$$\text{Var}(aX \pm bY) = a^2 \text{Var}(X) + b^2 \text{Var}(Y) \pm 2ab \text{cov}(X, Y) \quad (7.28)$$

$$\text{cov}(X, a) = 0 \quad (7.29)$$

$$\text{cov}(X, X) = \text{Var}(X) \quad (7.30)$$

$$\text{cov}(X, Y) = \text{cov}(Y, X) \quad (7.31)$$

$$\text{cov}(aX, bY) = ab \text{cov}(X, Y) \quad (7.32)$$

$$\text{cov}(X + a, Y + b) = \text{cov}(X, Y) \quad (7.33)$$

Now that we have the usefull properties summarized, let's look at the statistical formulation of equivalent metrics for finite dataset size N , usually obtained by measurement:

$$\text{Sample mean:} \quad sm \equiv \bar{x} = \frac{1}{N} \sum_{i=1}^N x_i \quad (7.34)$$

$$\text{Sample variance} \quad sv \equiv s_x^2 = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2 \quad (7.35)$$

$$\text{Sample covariance} \quad sc \equiv s_{x,y} = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})(y_i - \bar{y}) \quad (7.36)$$

We would intuitively expect, that the sample moments (also called statistics) will approximate the theoretical moments, and even converge to them with rising value of N . Indeed, we will show this in the following exercises.

MATLAB Exercise 7.1: Generate pseudo-random matrices

Using the `randn()` built-in, implement a function with a given the prototype:

```
function Rn = random( s, m, v )
```

Generate a matrix $\mathbf{R}_n$ of pseudo-random floating point values drawn from a normal distribution, Let the size of the output matrix be $M \times N$ (defined as input vector $s = [M \ N]$) and the optional parameters be:

m – desired sample mean value with default value 0 if not specified

v – desired sample variance with default value 1 if not specified

Implement the generator/wrapper using a simple transformation utilizing the moments properties defined above.

Note: You don't have to implement the formulas (7.34) to (7.36), just use the MATLAB built-in functions `mean()`, `var()` and `cov()` instead. See their help for their exact definition of sample moments.

Test your generator function for $m = 8$ and $v = 3$ for a small output matrix size $M = 5$, $N = 10$. Run your generator several times to see it really produces different numbers each time.

Also analyze the resulting matrix - calculate the sample mean and sample variance of the output using the built-in functions `mean()`, `var()` :

- for each column - the results will be row vectors of sample averages and variances.
- for each row - the results will be column vectors of sample averages and variances.
- for the whole matrix - the results will be two scalar numbers per each matrix: sample mean and sample variance.

Observe, that the resulting sample mean and sample variance will be slightly different than the specified desired values $m = 8$ and $v = 3$. This is the essence of dealing with random variables.

MATLAB Exercise 7.2: Convergence of sample mean and variance

Using your `random()` function, generate a pseudo random column vector V of varying length N (use a `for` cycle with exponentially growing N , such as $N = [1 \ 2 \ 5 \ 10 \dots]^T$), with given desired sample mean and variance values: $m = 10$, $v = 3$. For each vector length N calculate the sample mean sm and variance sv (using MATLAB built-ins) and for each calculated sm and sv calculate their relative errors in percent That means error with respect to the desired values $m = 10$, $v = 3$. Let us define the relative errors as:

$$\text{Relative error of the sample mean: } e_m = \frac{|sm - m|}{m} \cdot 100 [\%]$$

$$\text{Relative error of the sample variance: } e_v = \frac{|sv - v|}{v} \cdot 100 [\%]$$

Also print these values. The errors should converge to zero with large N – Fig. 7.8 shows one potential output:

N:	1,	mean:	9.613017,	var:	0.000000,	mean err:	3.869828%,	var err:	100.000000%
N:	2,	mean:	11.235384,	var:	0.198815,	mean err:	12.353842%,	var err:	93.372837%
N:	5,	mean:	9.780189,	var:	2.901786,	mean err:	2.198108%,	var err:	3.273812%
N:	10,	mean:	10.272244,	var:	1.388240,	mean err:	2.722436%,	var err:	53.725344%
N:	100,	mean:	10.105764,	var:	3.486989,	mean err:	1.057642%,	var err:	16.232971%
N:	1000,	mean:	9.990951,	var:	2.854616,	mean err:	0.090493%,	var err:	4.846142%
N:	10000,	mean:	10.008857,	var:	3.019976,	mean err:	0.088570%,	var err:	0.665860%
N:	100000,	mean:	10.006103,	var:	2.997498,	mean err:	0.061028%,	var err:	0.083413%

Fig. 7.8: Convergence of sample moments sm , and sv towards μ and σ^2 .

We have seen that a random variable can be represented by a set of points – values drawn from a given distribution (often times normal). This set can be stored in a column or row vector, or even in a matrix or volume. The matrix representation however is more suitable to represent a different entity in the context of communications systems simulations – a random or *stochastic process*. The rigorous definition of a random process is outside the scope of this book, but we may think of it as a random function – usually a random function of time: $X(t)$. Similarly to a random variable, the random process is practically described by its moments, which themselves are functions.

For a random process, two moments are especially important: the mean $m_x(t)$, and the autocorrelation function $R_x(t)$, defined for a real-valued process as:

$$\mu(t) \equiv E[X(t)] \quad (7.37)$$

$$R_x(t_1, t_2) \equiv E[X(t_1)X(t_2)] \quad (7.38)$$

Out of the many types of random processes, the Wide-Sense Stationary (WSS) processes are often used in analysis. For a random process to be WSS, the following conditions must be met:

$$\mu(t) \equiv \text{const} \quad (7.39)$$

$$R_x(t_1, t_2) \equiv R_x(\tau), \text{ where } \tau = t_1 - t_2 \quad (7.40)$$

This means that the autocorrelation function depend only on the difference between the time instants, not on their actual position on the time-axis. For a WSS with autocorrelation $R_x(\tau)$, a Power Spectral Density (PSD) function describes the process in the frequency domain:

$$G_x(f) = F[R_x(\tau)] \quad (7.41)$$

where F denotes the Fourier transform. Functions $R_x(\tau)$ and $G_x(f)$ are equivalent to functions with same names and symbols defined in Chapter 4 for deterministic signals along with their properties:

Properties of $R_x(\tau)$:

1. $R_x(\tau) = R_x(-\tau)$
2. $R_x(\tau) \leq R_x(0)$
3. $R_x(0) = P_x$

Properties of $G_x(f)$:

1. $G_x(f) \geq 0$

2. $G_x(f) = G_x(-f)$
3. $\int_{-\infty}^{\infty} G_x(f) df = P_x$

Another important type of random processes is the *Gaussian random process* – such a process if sampled at any time instants $t_1, t_2, \dots, t_N$, where N is finite, then the resulting random vector – vector of random variables $(X_1, X_2, \dots, X_N) \equiv (X(t_1), X(t_2), \dots, X(t_N))$ is a *Gaussian random vector* – random variables X_1 to X_N are *jointly Gaussian*. A random vector is simply a column vector $\mathbf{X} = (X_1, X_2, \dots, X_N)^T$, consisting of finite number N of random variables. Its expected value, or mean, is simply defined as a vector of mean values of the constituent random variables:

$$\mathbf{m} \equiv E[\mathbf{X}] = (E(X_i)) \quad (7.42)$$

The variance itself is insufficient second moment in this case, and is functionally replaced by the *covariance matrix* $\mathbf{C}$, defined as:

$$\mathbf{C} \equiv E[(\mathbf{X} - \mathbf{m})(\mathbf{X} - \mathbf{m})^T] = (c_{ij}) = (\text{cov}(X_i, X_j)) \quad (7.43)$$

Its entry at indices i and j represent the covariances between components – random variables X_i and X_j . Therefore $\mathbf{C}$ is symmetrical, and the elements on the main diagonal are the variances of the components X_i . In a Gaussian random vector, its N -dimensional probability density function (pdf) is called the joint pdf and defined as:

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{n/2} (\det(\mathbf{C}))^{1/2}} e^{\left(-\frac{(\mathbf{x} - \mathbf{m})^T \mathbf{C}^{-1} (\mathbf{x} - \mathbf{m})}{2} \right)} \quad (7.44)$$

Finally the famous AWGN noise model, ubiquitous in any theoretical communication channel description as well as in practical simulations, is a random process $N(t)$ that satisfies all of the following conditions:

1. Is a WSS
2. $\mu(t) \equiv 0$
3. Is a Gaussian process
4. $G_x(f) = \text{const} = \frac{N_0}{2}$

Similarly to a random variable that can be described by a potentially infinite set of realizations – random numbers, a random process can be described by a potentially infinite set of random functions. Figure 7.8 shows 4 realizations of a MATLAB generated model/approximation of a random process.

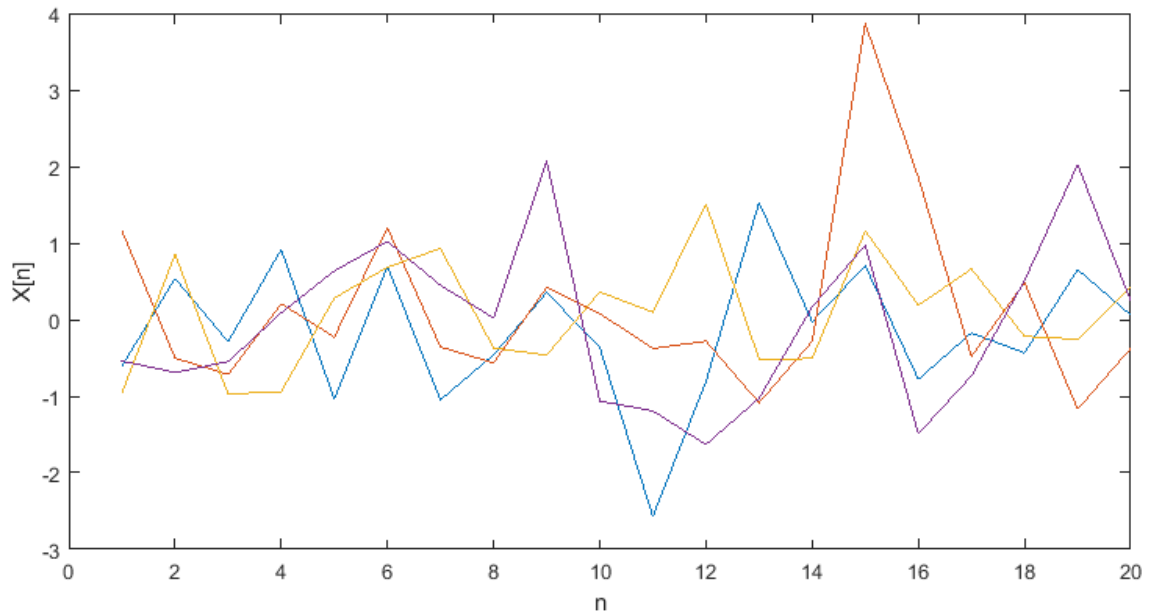


Figure 7.8: Four realizations of a discrete-time random process/ random function $X[n]$.

A random process $X(t)$ can be viewed from two perspectives: as a (potentially infinite) collection of functions (realizations $x_j(t)$) called an *ensemble* (of realizations), or as a time-series of random variables $X_i = X(t_i)$, taken at some (potentially infinite) set of time instants t_i . For practical purposes we will again focus on modelling a random process on a computer. This implies:

1. discrete-time representation of (random) realization $x[n] \equiv x_n \equiv xn$
2. discrete-frequency $X[k] \equiv X_k \equiv xk$ representation of their (random) spectra
3. finite set of realizations constituting an ensemble
4. representation of each function as a finite-length vector of samples - real or complex numbers

These properties make a matrix an ideal representation for a random process, where each column can represent one realization function with the row index representing index of time sample (With the realistic assumption of regular sampling at integer multiples of sampling time T_s).

MATLAB Exercise 7.3: Sample mean and variance of a random process

Using your `random()` function write a script, that generates a matrix $\mathbf{R}_N$ of random numbers drawn from the standard normal distribution $N(0,1)$. Interpret $\mathbf{R}_N$ as an approximation of an ensemble of random functions where each function is stores as one column. Use $N = 100$ as a number of time samples (rows). Using the built-in functions `mean()` and `var()` calculate the sample mean sm and sample variance sv of all the random variables X_i representing time samples of the random process. The resulting sm and sv will therefore be vectors representing discrete-time functions. Plot the resulting mean and variance functions for increasing number of realizations $s = [10 \ 100 \ 1000 \ 10000]$.

Figure 7.9 shows the resulting functions that are again random. It can be seen from the first subplot in the top left corner, that for a small number of realizations, the results vary quite wildly, from other subplots, we can see that we need an ensemble of at least 1000 functions in order to get intuitively expected results.

Note: Because of relatively large number of samples N , the plots are drawn using the `plot()` function and not `stem()`, which would intuitively be a more appropriate for discrete-time signals. Should we try to use `stem()`, the plot would get completely filled and very hard to read. The use of `stem()` is therefore practical only for a small number of displayed points. Also, we may justify this choice by

remembering, that the vectors in MATLAB represent samples of some continuous-time function that has been converted using an analog to digital converter for processing.

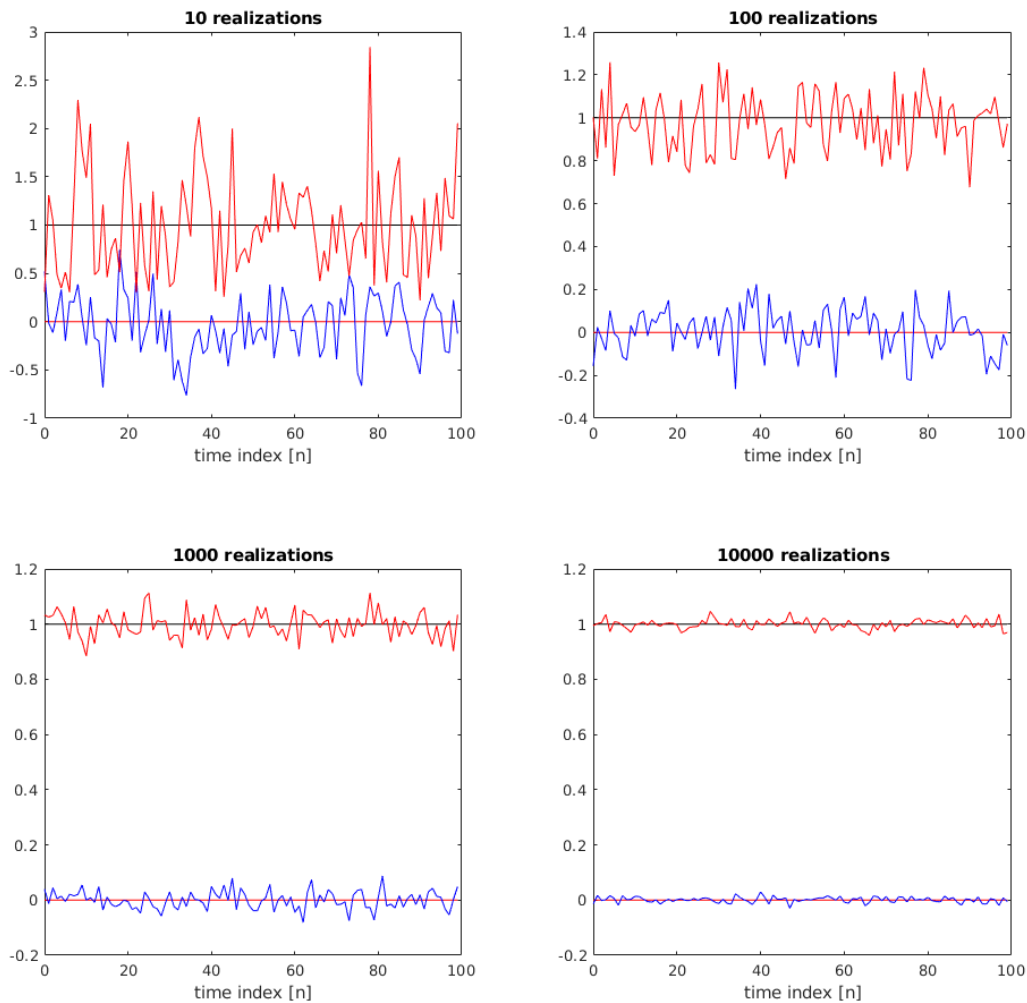


Figure 7.9: Sample mean and sample variance as a functions of time for a Gaussian random process with samples at each time instant are drawn from $N(0,1)$.

Let us now look at the same example from a different perspective – let’s try to analyze each realization – column of the $\mathbf{R}_N$ matrix separately, and let us view it as samples of a random variable. For sure we can calculate sample mean and sample variance.

MATLAB Exercise 7.4: Sample mean and variance of a random vector

Using your `random()` function write a script, that generates a matrix $\mathbf{R}_N$ of random numbers drawn from the standard normal distribution $N(0,1)$. Interpret $\mathbf{R}_N$ as sets of realizations of a random vector $\mathbf{X} = (X_1, X_2, \dots, X_N)$, where each column j of $\mathbf{R}_N$ represents values belonging to random variable X_j . Use $N = 10$ as a number components of the random vector (columns of $\mathbf{R}_N$). Vary the number of samples in each column $s = [10 \ 100 \ 1000 \ 10000]$ and draw the resulting sample means and sample variances of $\mathbf{X}$, obtained by `mean()` and `var()`. Use `stem()` to emphasize the discrete nature of a random vector.

Also, pay attention to the MATLAB auto-y-axis scaling feature. This can be quite tricky for an inexperienced user, so let us plot three graphs for each value of s : Just the means of $\mathbf{X}$, just the variances

of $\mathbf{X}$, and the means and variances together in the same graph. Fig. 7.10 shows two such graph triplets for sample size $s = 10$, right $s = 100$.

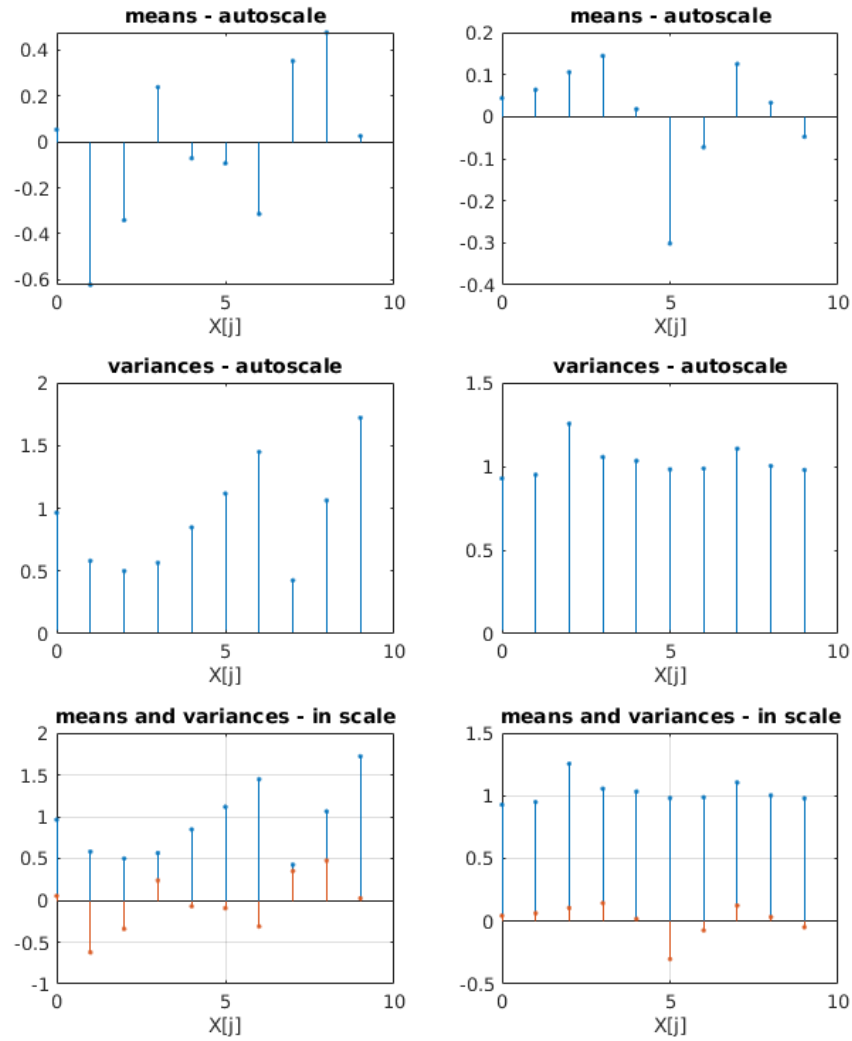


Fig. 7.10: Sample means and variances of components of a random vector $(X_1, X_2, \dots, X_{10})$. Left: sample size $s = 10$, right $s = 100$. Only the bottom row provides immediate visual clue regarding the scale of the random values sm and sv .

Fig. 7.10 shows what we expected: the means and variances of the random vector components X_j being themselves random, along with the fact that a larger sample size visibly converges to theoretical values. Similarly Fig. 7.11 shows further improvement for even larger dataset size.

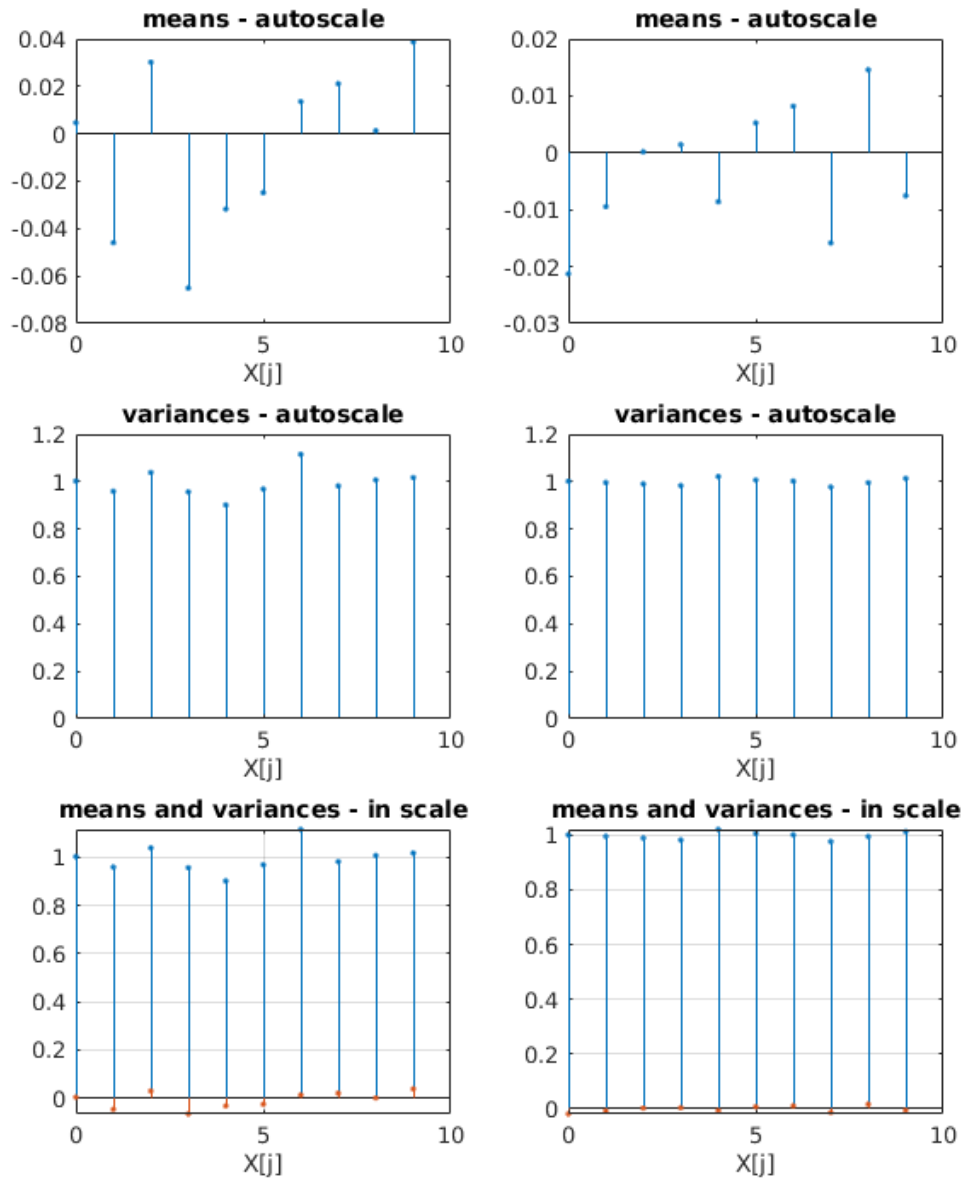


Fig. 7.11: Sample means and variances of components of a random vector $(X_1, X_2, \dots, X_{10})$. Left: sample size $s = 1000$, right $s = 10000$.

What did we do in the previous two exercises? We interpreted the same pseudo-randomly generated matrix of samples drawn from $N(0,1)$ in two different ways, with increasing number of ensemble realizations in Exercise 7.3 and time samples in Exercise 7.4, converging to very similar results – values sm and sv . Stochastic processes, for which these values converge, regardless if being calculated as ensemble moments, or from time-samples of one realization, are called *ergodic* processes. The AWGN random process is an example of an ergodic process.

With understanding of mean and variance calculation and behavior, let us now return to the other two important characteristics of AWGN: autocorrelation function $R_x(\tau)$ and PSD $G_x(f)$. Intuitively, we expect, that using a computer approximations and finite sample size, we will have to use the sample moments defined in equation 7.34 to 7.36, from which we will obtain a random sample autocorrelation $R_{xs}[m]$ – a function of discrete time delay m and a random sample PSD $G_{xs}[k]$ as a function of discrete frequency. We can also foresee, that these two random statistics shapes should converge to the clean theoretical charts show in Fig. 7.2 and 7.3 (repeated here as Fig. 7.12) with increasing dataset size.

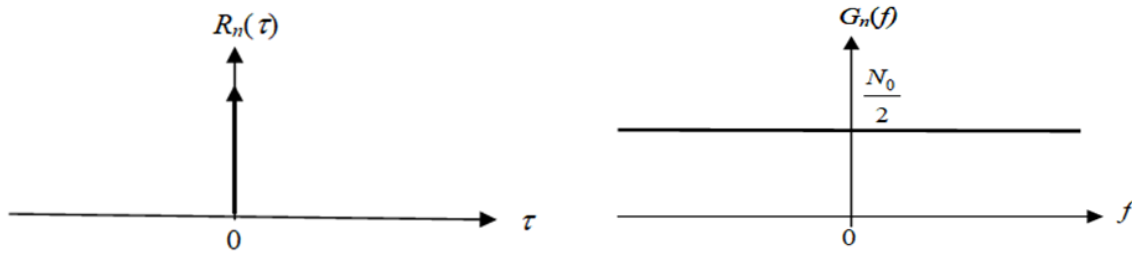


Fig. 7.12: Textbook depiction of the autocorrelation function $R_x(\tau)$ and power spectral density $G_x(f)$ of a white Gaussian stochastic process.

Just as we approached the mean and variance by averaging over some sample size in the previous examples, we can calculate the approximation of a random process autocorrelation by simply calculating autocorrelations of each of the random realizations and then average over these random realizations of the autocorrelation.

A non-normalized cross-correlation of a two vectors may be defined as:

$$R_{xy}[m] = \begin{cases} \sum_{n=0}^{N-m-1} x_{n+m} y_n, & m \geq 0 \\ R_{xy}[-m], & m < 0 \end{cases} \quad (7.45)$$

MATLAB Exercise 7.5: Simple cross-correlation of two vectors

Implement a public function with the prototype:

```
function [ R, lags, lag0i ] = xcor( a, b )
```

that will calculate simple non-normalized cross correlation of two column vectors **a** and **b** as defined by (7.45). Rewrite the equation for MATLAB indexing starting at 1. If we denote the lengths of the vectors la and lb , the result will be $la + lb - 1$ samples long. Also calculate the lags vector – vector of shifts for proper x axis labeling. Also calculate the index of the largest R_{xy} value - $lag0i$.

Read Chapter 4.1 again and use your implementation of the `convmtx()` from Chapter 4.1 in your implementation.

MATLAB Exercise 7.6: Ensemble cross-correlation

Implement a public function with the prototype:

```
function [ R, lags, lag0Index ] = excor( A, B )
```

that will calculate cross correlations for each of the realizations stored as columns of matrices **A** and **B**.

This functions should act as a simple wrapper to the function `xcor()` from the previous exercise.

The number of columns of **A** and **B** must be equal and if only one argument is supplied, the function will calculate the autocorrelations of the columns of **A**.

MATLAB Exercise 7.7: Random process autocorrelation function approximation

Using your `random()` function write a script, that generates a matrix **Noise** of random numbers drawn from the standard normal distribution $N(0,1)$. Interpret **Noise** as an approximation of an ensemble of random functions where each function is stores as one column. Use $N = 100$ as a number of time samples (rows). Use the `excor()` function to calculate an ensemble of random autocorrelations and then use simple average to obtain the desired random process approximation $R_x[m]$. Plot the resulting autocorrelation function estimate for increasing number of realizations $s = [1 \ 10 \ 100 \ 1000]$.

Fig. 7.13 shows the convergence of the sample $R_x[m]$ towards the theoretical value $R_x(\tau)$.

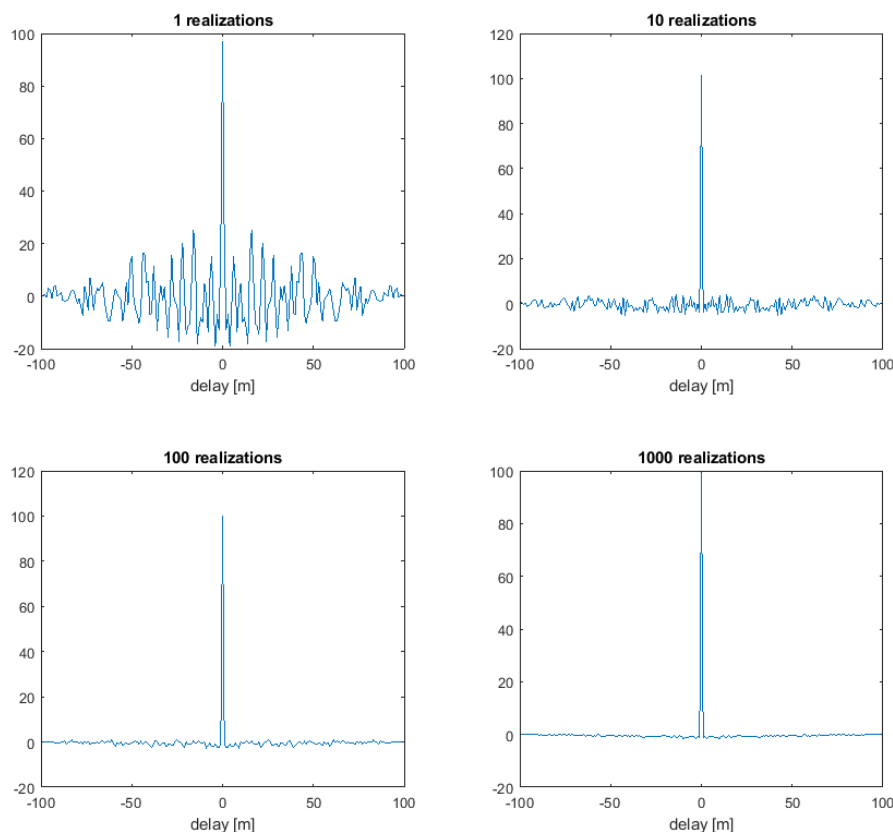


Fig. 7.13: AWGN autocorrelation function estimate as a result of averaging s cross-correlations realizations.

MATLAB Exercise 7.8: Random process PSD approximation

Using your `random()` function write a script, that generates a matrix **Noise** of random numbers drawn from the standard normal distribution $N(0,1)$. Interpret **Noise** as an approximation of an ensemble of random functions where each function is stores as one column. Use $N = 100$ as a number of time samples (rows). Use the `excov()` function to calculate an ensemble of random autocorrelations. Use `fft()` to calculate an ensemble of random spectra and then average these to form the resulting random process PSD estimate. Alternatively average the random autocorrelation and then use `fft()` to calculate the resulting PSD. The results should be the same.

Plot the resulting PSD function estimate for increasing number of realizations $s = [1 \ 10 \ 100 \ 1000]$. Observe the resulting plot, and add y-axis scaling so that the chart isn't automatically zooming in. See Fig. 7.14 and 7.15.

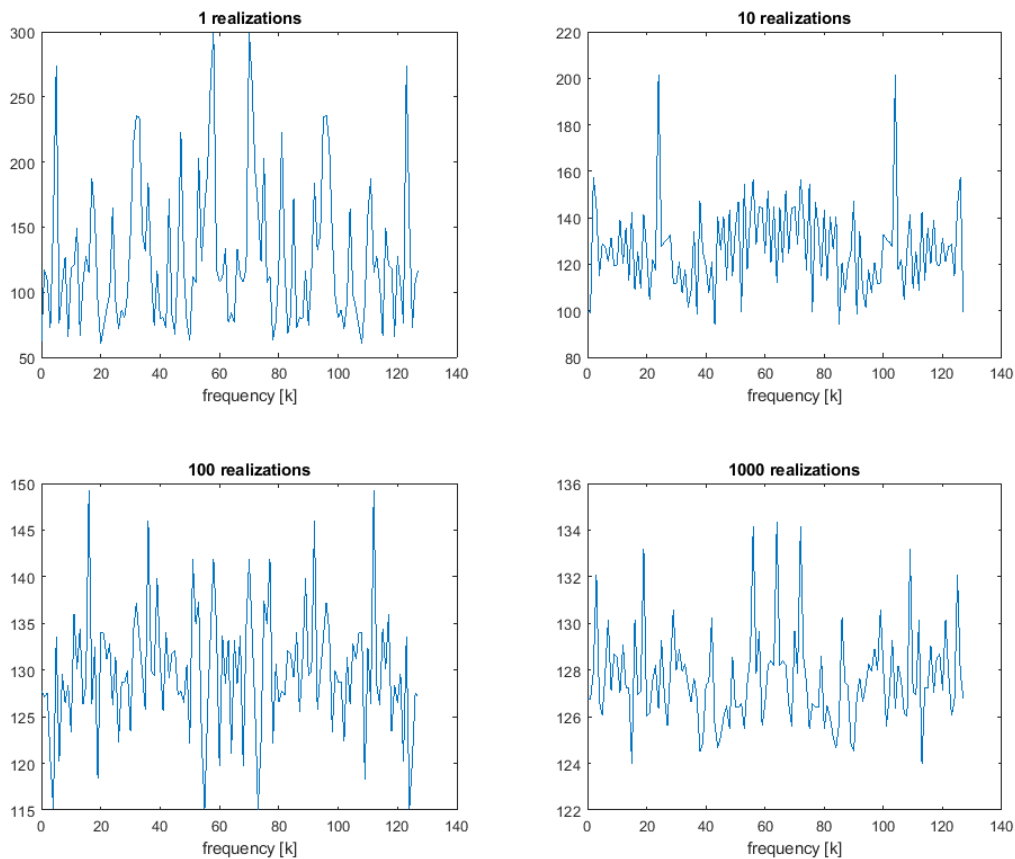


Fig. 7.14: AWGN power spectral density estimate as a result of averaging s random spectra. The auto-scaling of the y-axis introduces much confusion.

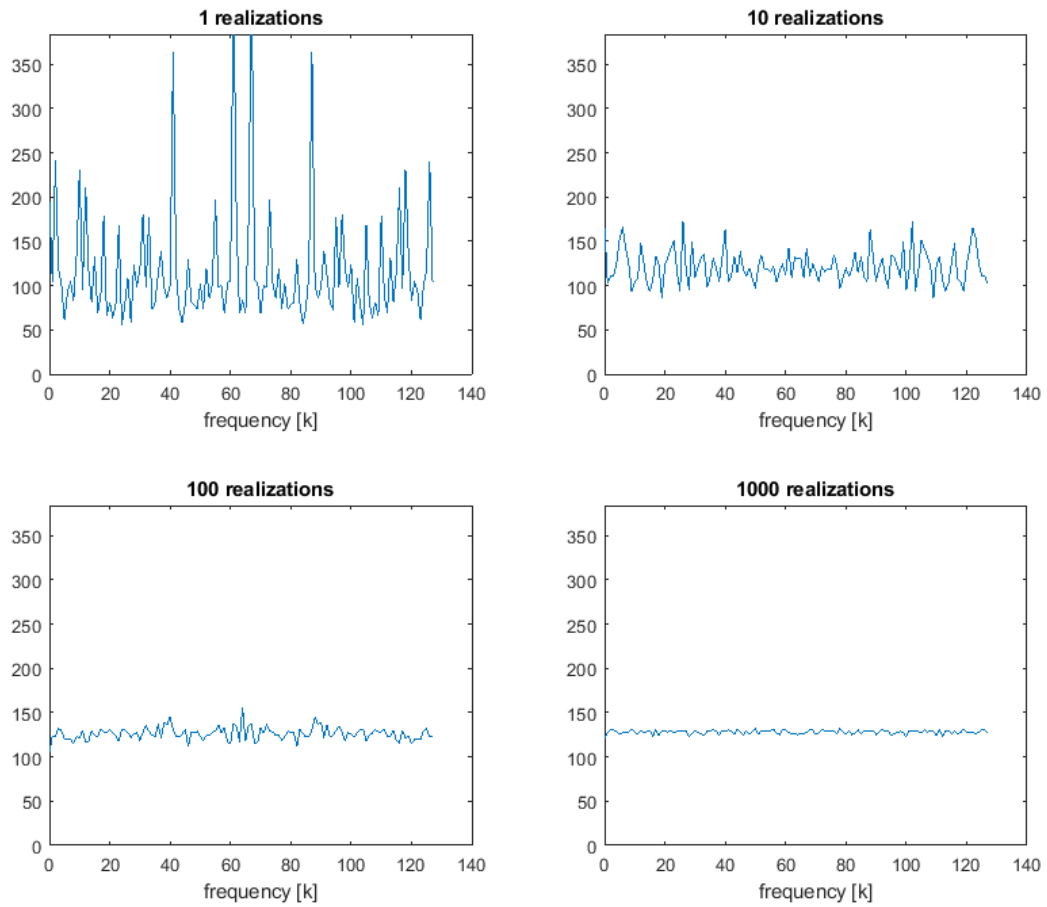


Fig. 7.15: AWGN power spectral density estimate as a result of averaging s random spectra. The y-axis limits were set manually for the user to be able to see the actual flattening of the spectrum.

Chapter 8: Signal Detection

Detection of binary signals under the presence of AWGN

Block diagram of the optimal receiver:

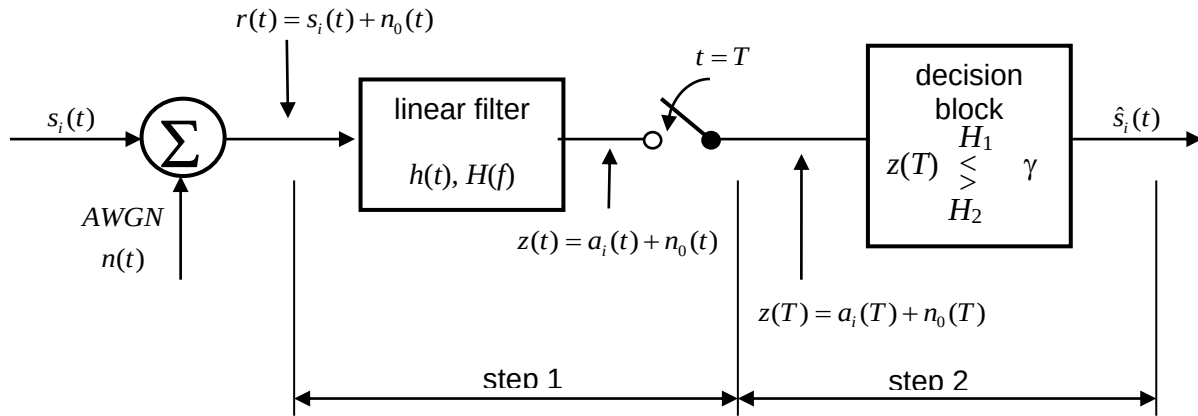


Fig. 8.1: Block diagram of the optimal receiver.

$s_i(t)$ - sent signal

$n(t)$ - AWGN

$r(t)$ - received signal

$z(t)$ - output of the linear filter – Gaussian random variable

$a_i(t)$ - signal component of $z(t)$ assuming that signal $s_i(t)$

$n_0(t)$ - noise component of $z(t)$

$z(T)$ - test statistic (a voltage proportional to the energy of the received symbol)

H_1, H_2 - statistical hypothesis

γ - decision threshold

$\hat{s}_i(t)$ - estimate of the sent signal

Detection consists of two steps:

Step 1 - transforming the received signal $r(t)$ into a sample $z(T)$ - which is the test statistic. It can be done by a linear filter, after which a sampling circuit is connected. For optimum reception (min P_b), the following filters should be used:

(a) a matched filter (MF)

or

(b) a correlator

Step 2 - deciding which signal has been transmitted by comparing the value of $z(T)$ and the decision threshold γ . The decision results in one of the possible hypotheses H_1, H_2 , which then determines the

estimate of the transmitted signal $\hat{s}_i(t)$. At the beginning of each symbol interval T , the initial conditions of the MF or correlator are set to zero values. At the end of the interval T , a sample appears at the output of the MF or correlator. The linearity of the filter causes that if the input is a random variable (sent signal + AWGN) with a given distribution, then at the output we will also get a random variable with the same distribution (in this case Gaussian).

Matched filter - is a linear filter constructed to maximize S/N at the output for a particular waveform.

It is the optimal filter. It is designed so: $\left(\frac{S}{N}\right) = \frac{a_i^2}{\sigma_0^2}$ is maximal. Main characteristics of MF:

- transfer function: $H(f) = k S^*(f) e^{-j2\pi f T}$, where $S^*(f)$ is a conjugate spectrum of the filtered signal
- impulse response: $h(t) = \begin{cases} k s(T-t) & 0 \leq t \leq T \\ 0 & \text{otherwise} \end{cases}$

At the time $t = T$ the output of the MF and correlator is identical:

$$z(t) = r(t) * h(t) = \int_0^t r(\tau) h(t-\tau) d\tau = \int_0^T k r(\tau) s[T-(t-\tau)] d\tau = k \int_0^T r(\tau) s(\tau) d\tau = z(T)$$

If the correlator and MF are time synchronized, then their outputs at the end of the symbol interval are identical. Since the correlator gives a single output value/symbol, it must have additional information about the start and end of the integration time. If there are timing errors in the correlator, the output to the detector will be significantly distorted. However, by additional circuitry, the digital MF can learn the optimum sampling time. Based on the reference correlator input we can distinguish two cases:

1. as a reference we take signal prototype – this is denoted as receiver of type I.
2. as a reference we take basis function – this is denoted as receiver of type II.

Analog MF

Exercise 8.1:

Calculate the output of the analog MF if the input signal is:

- a) UPNRZ with amplitude A
- b) BPNRZ with amplitude A

Solution:

a)

Let: $s_1(t) = A \quad 0 \leq t \leq T$ „log 1“ $A > 0$ and $s_2(t) = 0 \quad 0 \leq t \leq T$ „log 0“

Block diagram of the detector:

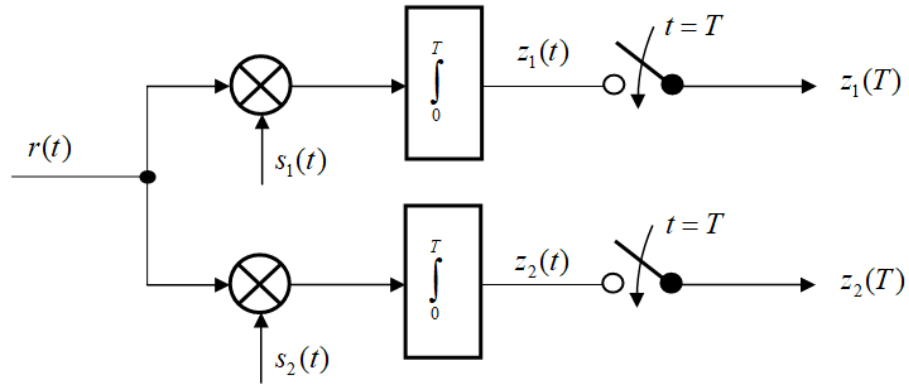


Fig. 8.2: Block diagram of a correlator of type I.

The expected value of the output $z_1(T)$, if a signal $s_1(t)$ has been sent:

$$\begin{aligned}
 z_1(T) &= \mathcal{E} \{ z(T) / s_1(t) \} \\
 &= \mathcal{E} \left\{ \int_0^T [s_1(t) + n(t)] s_1(t) dt \right\} \\
 &= \mathcal{E} \left\{ \int_0^T [A^2 + A n(t)] dt \right\} \\
 &= \int_0^T A^2 dt + \int_0^T A \mathcal{E} \{ n(t) \} dt \\
 &= A^2 T \quad [V]
 \end{aligned}$$

Similarly:

$$\left. \begin{aligned}
 z_1(T) &= \mathcal{E} \{ z(T) / s_2(t) \} = 0 \\
 z_2(T) &= \mathcal{E} \{ z(T) / s_1(t) \} = 0 \\
 z_2(T) &= \mathcal{E} \{ z(T) / s_2(t) \} = 0
 \end{aligned} \right\} \Rightarrow$$

The detector can be implemented with a single correlator since: $s_2(t) = 0 \quad 0 \leq t \leq T$

The random variable $z_i(T)$, $i=1,2$ - output from the MF depends on $s_i(t)$ and noise $n(t)$, as implied by the distribution of conditional probability densities:

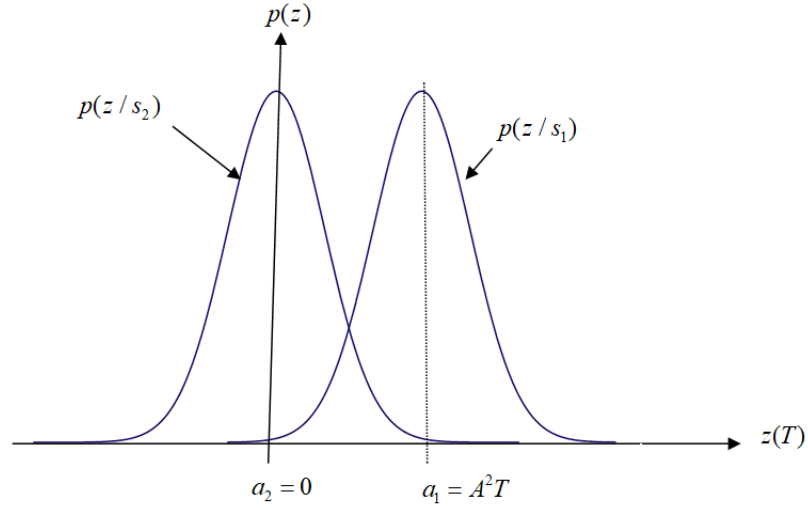


Fig. 8.3: Conditional probability densities for a unipolar signal in an AWGN channel.

b)

Let: $s_1(t) = A \quad 0 \leq t \leq T$ „log 1“ $A > 0$ and $s_2(t) = -A \quad 0 \leq t \leq T$ „log 0“

The outputs of MF:

$$z_1(T) = \varepsilon \{ z(T) / s_1(t) \} = A^2T$$

$$z_1(T) = \varepsilon \{ z(T) / s_2(t) \} = \varepsilon \left\{ \int_0^T [-A + n(t)] \times A dt \right\} = -A^2T$$

$$z_2(T) = \varepsilon \{ z(T) / s_1(t) \} = -A^2T$$

$$z_2(T) = \varepsilon \{ z(T) / s_2(t) \} = A^2T$$

Conditional pdfs:

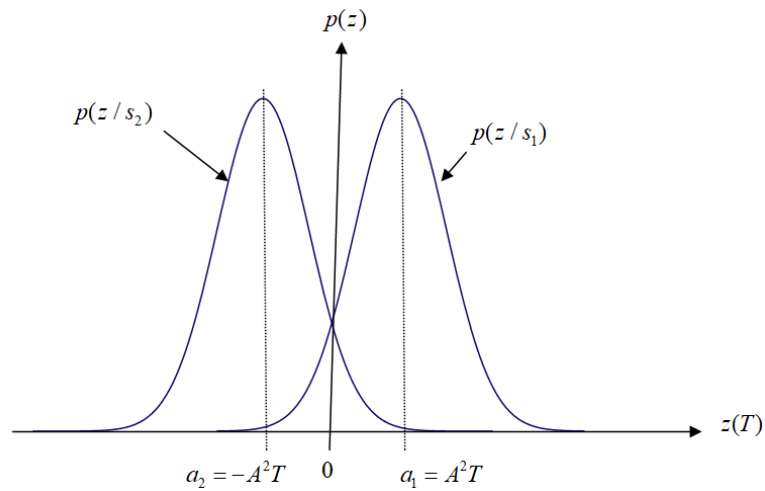


Fig. 8.4: Conditional probability densities for a bipolar signal in an AWGN channel.

■

Digital MF

Block diagram of a digital MF:

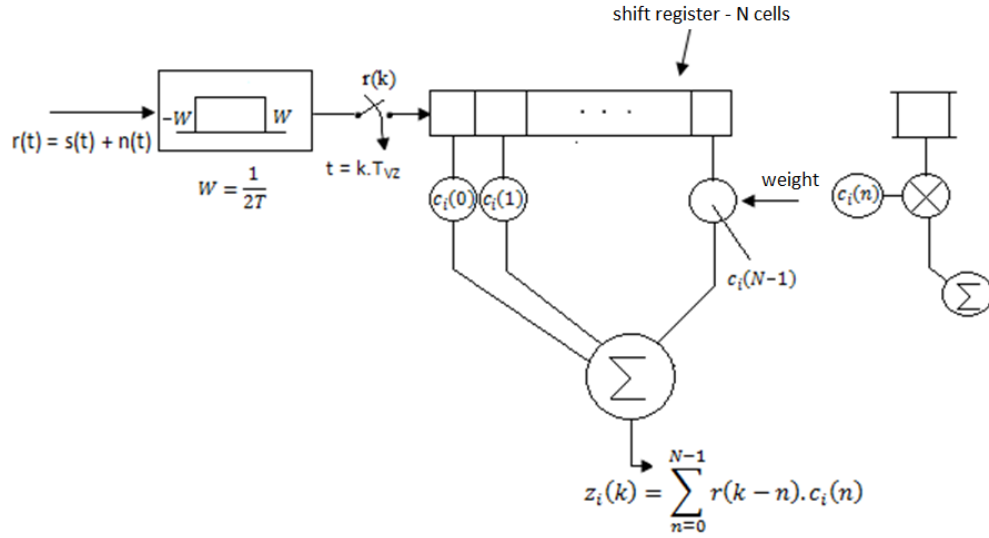


Fig. 8.5: Block diagram of a digital matched filter.

Nyquist bandwidth is: $W = \frac{1}{2T}$, where T is symbol duration. Therefore Nyquist sampling frequency is:

$f_0 = 2W = \frac{1}{T}$. In real systems sampling frequency f_s is usually: $\geq 4f_0$. During clock pulses at times $r(k)$ samples enter the register (earlier samples are to the right of later ones). Sampled received signal in binary case: $r(k) = s_i(k) + n(k)$ $i = 1, 2$ $k = 0, 1, \dots, \text{mod } N - k$ is the sampling time index). Weights - approximate $h(t)$ of MF, where $n = 0, 1, \dots, N - 1$ is the time index of the weights and register cells. N is the number of register cells and the number of samples per symbol. Summation occurs after all N samples of a symbol have been loaded into the register. In real systems, each sample is represented by $6 \div 8$ bits depending on the density of the signal constellation (their proximity in the signal space). The set of weights corresponds to $h(t)$ of MF: $c_i(n) = s_i[(N-1)-n] \leftrightarrow h(t) = s[T-t]$. The output can then be expressed using the discrete form of the convolution integral:

$$z_i(k) = \sum_{n=0}^{N-1} r(k-n) c_i(n) \quad k = 0, 1, \dots, \text{mod } N \quad (8.1)$$

Assuming that the system is synchronized and T is known, we also assume that the received signal noise has zero mean, then the expected (predicted) value of the received signal $s_i(t)$ is $\mathcal{E}\{r(k)\} = s_i(k)$ $i = 1, 2$. Assuming that the signal $s_i(t)$ has been transmitted, the expected value at the output of the MF will be:

$$\mathcal{E}\{z_i(k)\} = \sum_{n=0}^{N-1} s_i(k-n) c_i(n) \quad k = 0, 1, \dots, \text{mod } N \quad (8.2)$$

Exercise 8.2:

Calculate the output of the digital MF if we consider the transmission of antipodal signaling sampled with 4 samples/symbol by 3 level sampling, as per Fig. 8.6. We assume a noiseless transmission. Draw a block diagram of the filter and describe its operation.

4 samples per symbol $\Rightarrow N = 4$

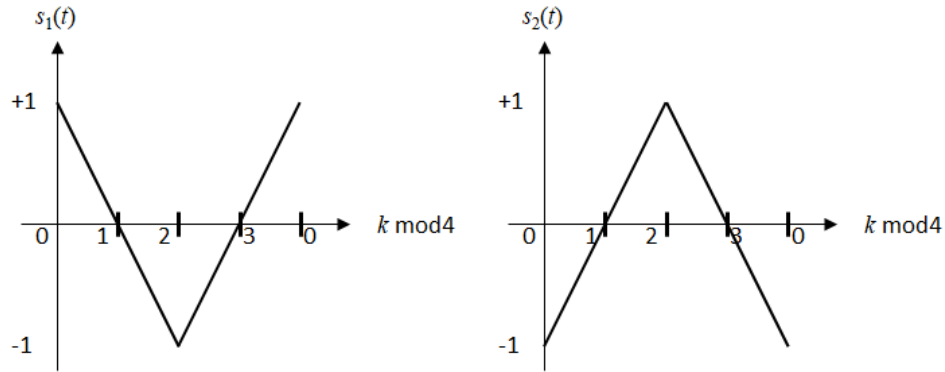


Fig. 8.6: Three-level antipodal signals.

Solution:

After sampling: $s_i(t) \rightarrow s_i(k)$

$$s_1(0) = +1$$

$$s_1(1) = 0$$

$$s_1(2) = -1$$

$$s_1(3) = 0$$

$$s_2(0) = -1$$

$$s_2(1) = 0$$

$$s_2(2) = +1$$

$$s_2(3) = 0$$

filter coefficients: $c_1(n) = s_1[(N-1)-n]$

$$c_1(0) = s_1(3) = 0$$

$$c_1(1) = s_1(2) = -1$$

$$c_1(2) = s_1(1) = 0$$

$$c_1(3) = s_1(0) = +1$$

$$c_2(0) = s_2(3) = 0$$

$$c_2(1) = s_2(2) = +1$$

$$c_2(2) = s_2(1) = 0$$

$$c_2(3) = s_2(0) = -1$$

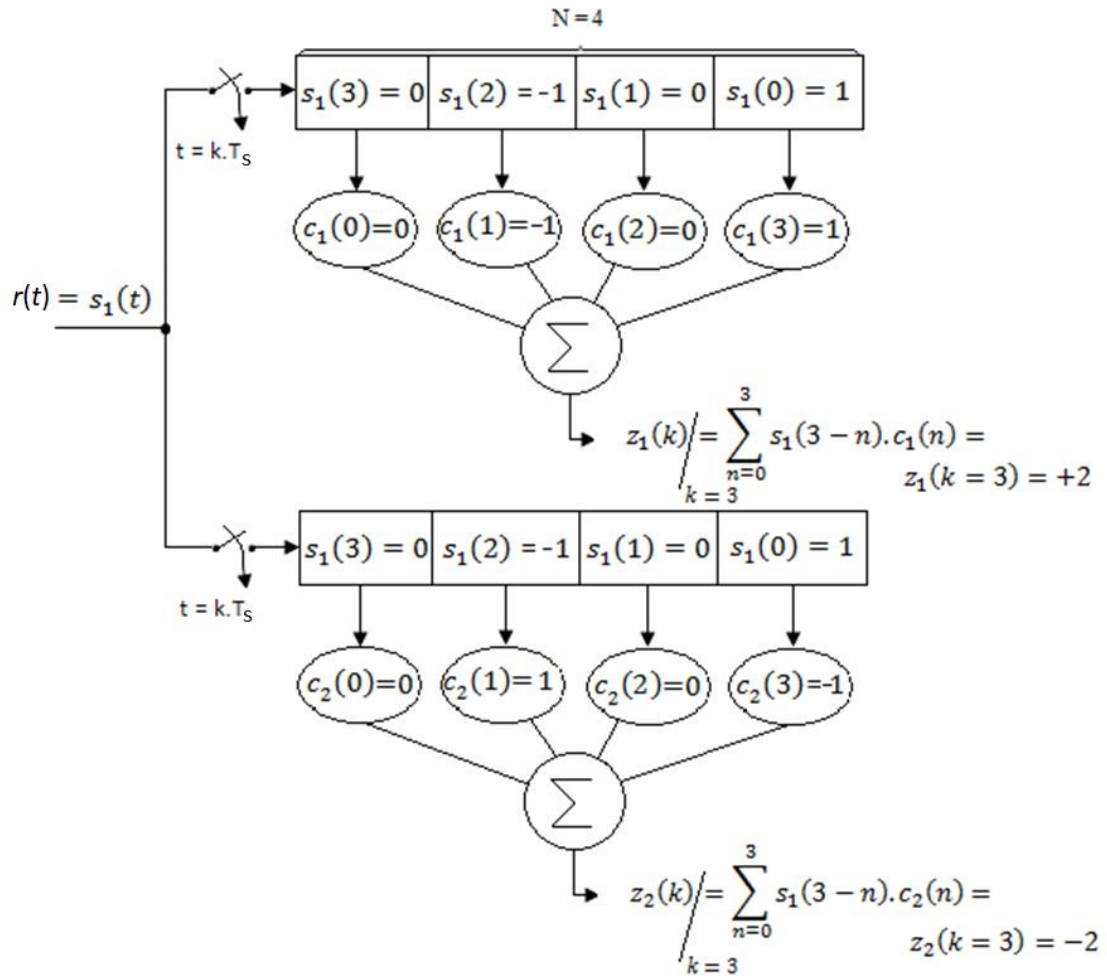


Fig. 8.7: Digital implementation of a matched filter.

Similarly:

$$\mathcal{E}\{z_1 / s_1\} = +2$$

$$\mathcal{E}\{z_1 / s_2\} = -2$$

$$\mathcal{E}\{z_2 / s_1\} = -2$$

$$\mathcal{E}\{z_2 / s_2\} = +2$$

■

MAP and ML decision criteria

The output of the detector is a random variable z - mostly continuous. Based on the value of z , the receiver decides whether the signal $s_1(t)$ or $s_2(t)$ has been transmitted. Therefore, it is useful to know $p(s_i / z)$ (the probability that a signal $s_i(t)$, $i = 1, 2$ has been transmitted given some particular value of z at the output of the MF). We can measure $p(z / s_i)$. Converting $p(z / s_i)$ to $p(s_i / z)$ can be done using Bayes formula:

$$p(s_i / z) = \frac{p(z / s_i) P(s_i)}{P(z)} \quad (8.3)$$

$$P(z) = \sum_i p(z / s_i) P(s_i) \quad (8.4)$$

MAP (Maximal A' posteriori Probability) - optimal only if all types of errors are equally important (have equal weight) we choose the signal $s_i(t)$ that has the highest aposteriori probability $p(s_i / z)$:

$$\begin{matrix} H_1 \\ p(s_1 / z) \propto p(s_2 / z) \\ H_2 \end{matrix} \quad (8.5)$$

If we substitute Bayes' formula into this relation we get:

$$\begin{matrix} H_1 \\ \frac{P(s_1) p(z / s_1)}{P(z)} \propto \frac{P(s_2) p(z / s_2)}{P(z)} \\ H_2 \end{matrix}$$

$$\begin{matrix} H_1 \\ P(s_1) p(z / s_1) \propto P(s_2) p(z / s_2) \\ H_2 \end{matrix}$$

$$\begin{matrix} H_1 \\ \frac{p(z / s_1)}{p(z / s_2)} \propto \frac{P(s_2)}{P(s_1)} \\ H_2 \end{matrix} \quad (8.6)$$

ML (Maximal Likelihood) – if we assume that: $P(s_1) = P(s_2)$ then:

$$\begin{matrix} H_1 \\ \frac{p(z / s_1)}{p(z / s_2)} \propto 1 \\ H_2 \end{matrix} \quad (8.7)$$

So far we assumed that noise is not present in the channel. In case of AWGN MAP criteria has to be modified to:

$$\begin{matrix} H_1 \\ z \propto \left\{ \ln \frac{P(s_2)}{P(s_1)} - \frac{a_2^2 - a_1^2}{2\sigma_0^2} \right\} \frac{\sigma_0^2}{a_1 - a_2} \\ H_2 \end{matrix} \quad (8.8)$$

Exercise 8.3:

The system uses binary signaling. The output of the detector in the receiver is z . The conditional probabilities are given graphically:

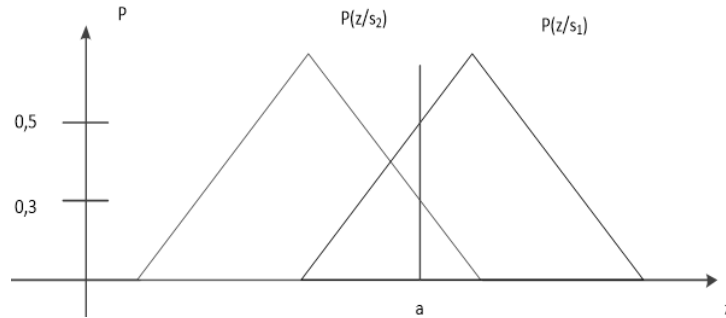


Fig. 8.8: Conditional probabilities of a hypothetical pdf.

Using the criteria:

(a) MAP

(b) ML

determine which signal was sent.

Solution:

a)

From the graph: $p(z_a / s_1) = 0.5$ $p(z_a / s_2) = 0.3$. Then:

$$p(s_1 / z_a) = \frac{p(z_a / s_1)P(s_1)}{p(z_a / s_1)P(s_1) + p(z_a / s_2)P(s_2)} = \frac{0.5 \times 0.3}{0.5 \times 0.3 + 0.3 \times 0.7} = 0.416$$

$$p(s_2 / z_a) = \frac{p(z_a / s_2)P(s_2)}{p(z_a / s_1)P(s_1) + p(z_a / s_2)P(s_2)} = \frac{0.3 \times 0.7}{0.5 \times 0.3 + 0.3 \times 0.7} = 0.58333$$

Since: $p(s_2 / z_a) > p(s_1 / z_a) \Rightarrow$ according to the MAP criterion, the signal $s_2(t)$ was sent.

b)

$p(z_a / s_1) = 0.5 > p(z_a / s_2) = 0.3 \Rightarrow$ according to the ML criterion, the signal $s_1(t)$ was sent – what contradicts to a). In this case a decision error occurred since: $P(s_1) \neq P(s_2)$.

■

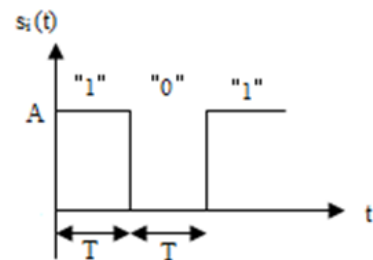
Exercise 8.4:

The unipolar NRZ signal with the following parameters is used for signaling:

$$A = 5V \quad \sigma_0^2 = 10^{-6} \quad R_b = 1 [kb / s]$$

$$P(0) = 0.3 \quad s_2(t) = 0 \quad 0 \leq t \leq T$$

$$P(1) = 0.7 \quad s_1(t) = A \quad 0 \leq t \leq T$$



Calculate the optimal decision level for detection by the MF filter in use:

- a) MAP criterion
- b) ML criterion

Solution:

a)

Based on the Eg.x the signal components at the output of MF are:

$$a_1 = A^2 T \quad s_1(t) = A \quad 0 \leq t \leq T$$

$$a_2 = 0 \quad s_2(t) = 0 \quad 0 \leq t \leq T$$

$R_b = 1 \text{ [kb / s]} \rightarrow T = 1 \text{ [ms]}$, therefore: $a_1 = A^2 T = 5^2 \times 10^{-3} = 0.025$ and $a_2 = 0$

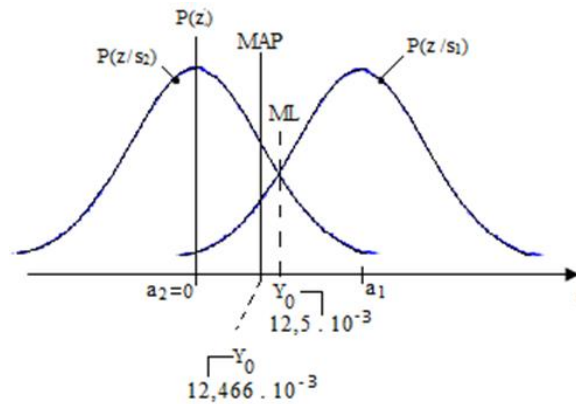
The optimum decision threshold is:

$$z \underset{H_2}{\overset{H_1}{\propto}} \left\{ \ln \frac{0.3}{0.7} - \frac{0^2 - 0.025^2}{2 \times 10^{-6}} \right\} \times \frac{10^{-6}}{0.025 - 0} \doteq 12.466 \times 10^{-3} = \gamma_0$$

γ_0 is shifted towards s_2 because: $P(s_1) > P(s_2)$

b)

$$P(s_1) = P(s_2) = \frac{1}{2} \rightarrow \gamma_0 = \frac{a_1 + a_2}{2} = \frac{0.025 + 0}{2} = 12.5 \times 10^{-3}$$



Signal detection in AWGN channel is analyzed in [12] in the TIMS experiment: “Detection on binary signals in AWGN channel”.

Error probability of binary UNRZ signaling in AWGN channel

Bit error probability at the output of the MF can be expressed as: $P_b = Q\left(\frac{a_1 - a_2}{2\sigma_0}\right)$. Further the output

of MF in case that the input is AWGN is: $\frac{N_0}{2} = \sigma_0^2 \quad [W] \Rightarrow 2\sigma_0 = \sqrt{2N_0}$ (under the assumption of

the type I. optimum receiver – the input signal is correlated with basis functions): $\psi_1(t) = \frac{s_1(t)}{\sqrt{E_b}}$,

$\psi_2(t) = 0$. For UNRZ signaling: $s_1(t) = A \rightarrow E_{b(1)} = A^2T$ and $s_2(t) = 0 \rightarrow E_{b(2)} = 0$ therefore the

mean energy per bit is: $E_b = \frac{1}{2} \times A^2T + \frac{1}{2} \times 0 = \frac{A^2T}{2}$. Then: $\psi_1(t) = \frac{A}{\sqrt{A^2T}} = \frac{1}{\sqrt{T}}$. Signal

components at the correlator output: $a_1(T) = \int_0^T A\psi_1(t) dt = \int_0^T A \frac{1}{\sqrt{T}} dt = A\sqrt{T}$ and $a_2(T) = 0$.

Finally:

$$P_b = Q\left(\frac{a_1 - a_2}{\sqrt{2N_0}}\right) = Q\left(\frac{A\sqrt{T}}{\sqrt{2N_0}}\right) = Q\left(\sqrt{\frac{A^2T}{2N_0}}\right) = Q\left(\sqrt{\frac{E_b}{N_0}}\right). \quad (8.9)$$

If we use only one correlator and at the reference input is applied the difference signal: $s_1(t) - s_2(t)$, then the bit error probability can be also expressed using the energy of the difference signal at the MF

input: $E_d = \int_0^T [s_1(t) - s_2(t)]^2 dt = \int_0^T [A - 0]^2 dt = A^2T$ and:

$$P_b = Q\left(\frac{a_1 - a_2}{\sqrt{2N_0}}\right) = Q\left(\sqrt{\frac{A^2T}{2N_0}}\right) = Q\left(\sqrt{\frac{E_d}{2N_0}}\right) \quad (8.10)$$

Exercise 8.5:

Calculate the bit error probability for unipolar NRZ signal with the following parameters:

$$A = 5V \quad \sigma_0^2 = 10^{-4} \quad R_b = 1 [kb/s] \quad P(s_1) = P(s_2)$$

$$s_1(t) = A \quad s_2(t) = 0 \quad 0 \leq t \leq T$$

Solution:

$$E_b = \frac{1}{2} (E_{b(1)} + E_{b(0)}) = \frac{1}{2} A^2T = \frac{1}{2} \times 5^2 \times 10^{-3} = 12.5 \times 10^{-3}$$

$$N_0 = 2\sigma_0^2 = 2 \times 10^{-4}$$

$$P_b = Q\left(\sqrt{\frac{E_b}{N_0}}\right) = Q\left(\sqrt{\frac{12.5 \times 10^{-3}}{2 \times 10^{-4}}}\right) = Q(7.9057) \doteq 1.353 \times 10^{-15}$$

■



BER measurement of UNRZ signal is analyzed in [10] in the TIMS experiment: “BER measurement of unipolar NRZ signals in a baseband distortionless channel”, Emona TIMS Volume D4 Further Advanced Digital Experiments, experiment: D4-01.

Error probability of binary BPNRZ signaling in AWGN channel

Bit error probability at the output of the MF can be expressed as in case of UNRZ signaling:

$P_b = Q\left(\frac{a_1 - a_2}{2\sigma_0}\right)$. The basis functions: $\psi_1(t) = \frac{s_1(t)}{\sqrt{E_b}}$, $\psi_2(t) = \frac{s_2(t)}{\sqrt{E_b}}$. For BPNRZ signaling:

$s_1(t) = A \rightarrow E_{b(1)} = A^2T$ and $s_2(t) = -A \rightarrow E_{b(2)} = A^2T$ therefore the mean energy per bit is:

$E_b = \frac{1}{2} \times A^2T + \frac{1}{2} \times A^2T = A^2T$. Then: $\psi_1(t) = \frac{A}{\sqrt{A^2T}} = \frac{1}{\sqrt{T}}$ and $\psi_2(t) = \frac{-A}{\sqrt{A^2T}} = \frac{-1}{\sqrt{T}}$. Signal

components at the correlator output: $a_1(T) = \int_0^T A\psi_1(t) dt = \int_0^T A \frac{1}{\sqrt{T}} dt = A\sqrt{T}$ and

$a_2(T) = \int_0^T A\psi_2(t) dt = \int_0^T A \frac{-1}{\sqrt{T}} dt = -A\sqrt{T}$. Finally:

$$\begin{aligned} P_b &= Q\left(\frac{a_1 - a_2}{\sqrt{2N_0}}\right) = Q\left(\frac{A\sqrt{T} - (-A\sqrt{T})}{\sqrt{2N_0}}\right) = Q\left(\frac{2A\sqrt{T}}{\sqrt{2N_0}}\right) \\ &= Q\left(\sqrt{\frac{4A^2T}{2N_0}}\right) = Q\left(\sqrt{\frac{2A^2T}{N_0}}\right) = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \end{aligned} \quad (8.11)$$

If we use only one correlator and at the reference input is applied the difference signal: $s_1(t) - s_2(t)$, then the bit error probability can be also expressed using the energy of the difference signal at the MF

input: $E_d = \int_0^T [s_1(t) - s_2(t)]^2 dt = \int_0^T [A - (-A)]^2 dt = 4A^2T$ and:

$$P_b = Q\left(\frac{a_1 - a_2}{\sqrt{2N_0}}\right) = Q\left(\sqrt{\frac{4A^2T}{2N_0}}\right) = Q\left(\sqrt{\frac{E_d}{2N_0}}\right) \quad (8.12)$$

Exercise 8.6:

Calculate the bit error probability for bipolar NRZ signal with the following parameters:

$$A = 5V \quad \sigma_0^2 = 10^{-4} \quad R_b = 1 \text{ [kb / s]} \quad P(s_1) = P(s_2)$$

$$s_1(t) = A \quad s_2(t) = -A \quad 0 \leq t \leq T$$

Solution:

$$E_b = \frac{1}{2} (E_{b(1)} + E_{b(2)}) = A^2T = 5^2 \times 10^{-3} = 25 \times 10^{-3}$$

$$N_0 = 2\sigma_0^2 = 2 \times 10^{-4}$$

$$P_b = Q\left(\sqrt{\frac{2A^2T}{N_0}}\right) = Q\left(\sqrt{\frac{2 \times 5^2 \times 10^{-3}}{2 \times 10^{-4}}}\right) = Q(15.811) = 1.311616 \times 10^{-56} \rightarrow \text{practically zero}$$

error, since antipodal signaling is used and the noise power is relatively small.

■



BER measurement of BPNRZ signal is analyzed in [10] in the TIMS experiment: “BER measurement of bipolar NRZ signals in a baseband distortionless channel”, Emona TIMS Volume D4 Further Advanced Digital Experiments, experiment: D4-02.

Exercise 8.7:

Calculate the bit error probability for bipolar NRZ signal with the following parameters:

$$A = 5V \quad \sigma_0^2 = 10^{-4} \quad R_b = 1 [Mb / s] \quad P(s_1) = P(s_2)$$

$$s_1(t) = A \quad s_2(t) = -A \quad 0 \leq t \leq T$$

Solution:

$$T = \frac{1}{R_b} = 10^{-6} [s] \rightarrow E_b = 25 \times 10^{-6} [J] \rightarrow \text{bit energy is decreased} \Rightarrow$$

$$P_b = Q\left(\sqrt{\frac{2 \times 25 \times 10^{-6}}{2 \times 10^{-4}}}\right) = Q(0.5) = 0.3085 \quad P_b \text{ if } R_b = 1 [kb / s] !$$

■

Inter-symbol interference (ISI)

ISI is an essential part of real wired and wireless communications. S. Morse was the first to realize this fact in his experiments with transatlantic telegraph transmission. He observed that if orthogonal pulses were transmitted at the receiving end, the pulses were distorted and merged into each other. Due to the effect of the transmission medium, the duration of each symbol is much longer than desired – signal dispersion. As can be seen in the figure bellow the resulting received signal is a superposition of all received distorted pulses. When two (or more) pulses overlap ISI is created. ISI can be caused by:

- the filtering effect of the HW used - for now we are only concerned with this effect
- selective frequency fading in case of mobile channels
- non-linearity of circuits
- charging effect

Increased ISI also increases BER, therefore ISI suppression is an essential part of any communication system (wired and wireless). The simplest way to suppress ISI is to reduce the transmission rate, i.e., to send a new pulse after the previous one decays. If the received pulse width $\leq$ symbol interval, no ISI occurs. However, reducing the transmission rate is an unacceptable solution.

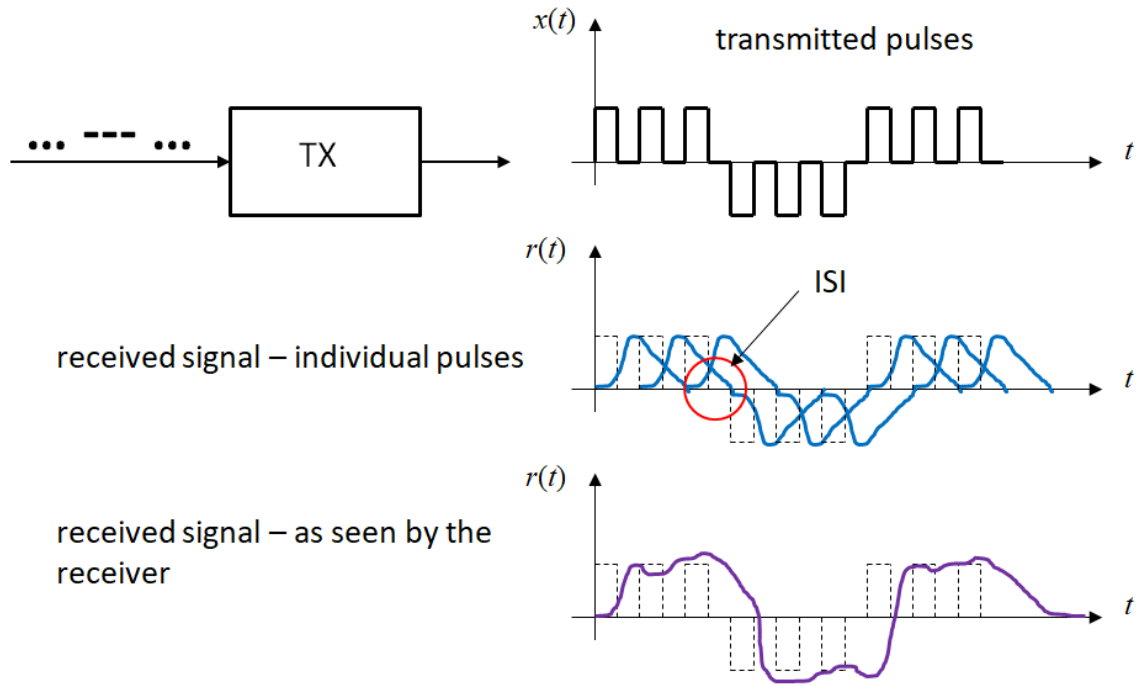


Fig. 8.9: Inter-symbol interference.

The theoretical background of analyzing mitigation of ISI laid down H. Nyquist. He showed that if we use ideal low-pass filter and at its input are Dirac pulses exactly separated by time T , then at the output a series of si pulses will occur. Since they are crossing each other at zero points no ISI is created:

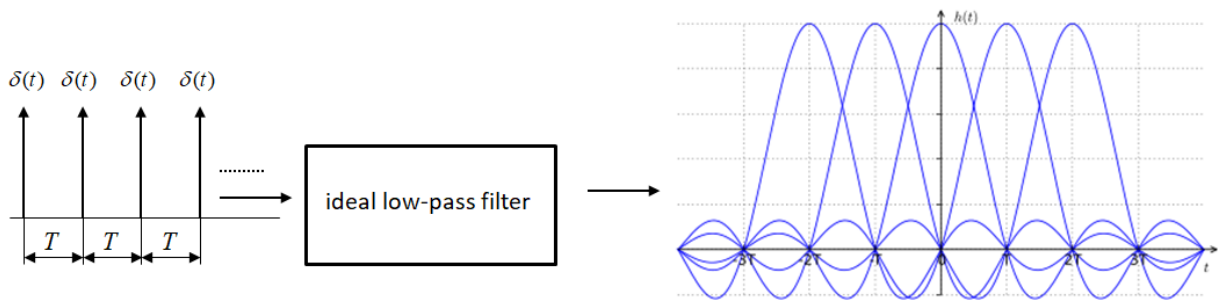


Fig. 8.10: Ideal LP filtering of a Dirac pulsetrain.

This ideal LPF has bandwidth $W_0 = \frac{1}{2T}$ [Hz] - what is also called Nyquist bandwidth. In this chapter we will assume 0-to-null or null-to-null bandwidth. From this bandwidth directly follows that the maximal symbol rate (without ISI) is $R_s = 2W_0$ [Baud] $\rightarrow$ it is possible to transmit maximum 2symbols per second per 1 Hz of bandwidth. Or if we want to transmit R_s [symbol/second] without ISI the minimal theoretical necessary bandwidth is $W_0 = \frac{R_s}{2}$ [Hz]. Nyquist limit is not possible to reach because:

1. Nyquist filter is not possible to construct (non-causal)
2. absolute ideal synchronization does not exist – in real system we also have some jitter

The main tool for ISI reduction is pulse shaping. The idea comes from the use of digital demodulation, i.e. we evaluate the received signal only at the sampling instances and are not interested in its level outside them. But we have to use such a transmission rate that the interference does not affect the received signal at the sampling instances.

Nyquist proposed forms of realizable pulses that have as good properties as the *sinc* pulse but fewer drawbacks. Nyquist pulses are pulses generated by the product of *sinc* and a real time function. The first class of Nyquist pulses is Raised Cosine (RC) pulses. RC pulses are modification of *sinc* pulses. RC pulse has selectable bandwidth from W_0 up to $2W_0$. Impulse response of a filter generating RC pulses has analytical form:

$$h(t) = \frac{\sin(\pi t / T_s)}{\pi t / T_s} \frac{\cos(\pi \rho t / T_s)}{1 - (2\rho t / T_s)^2} \quad (8.13)$$

, where T_s is pulse duration and ρ is roll-off factor

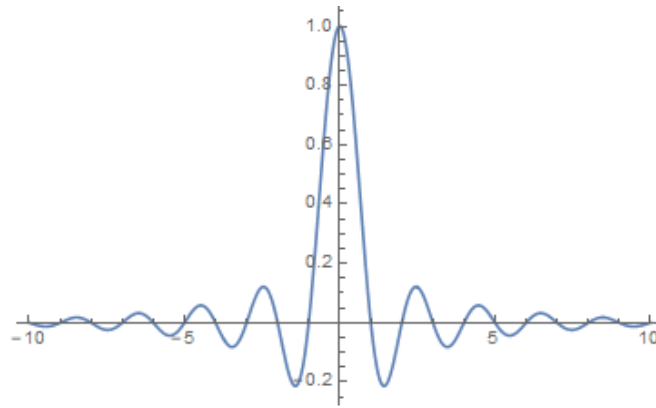


Fig. 8.11 RC pulse with $T_s = 1$ [s] and $\rho = 1$.

The actual pulse bandwidth is now by the factor $1 + \rho$ greater than the Nyquist bandwidth:

$$\rho = \frac{W - W_0}{W_0} \quad (8.14)$$

→ it indicates how the actual pulse bandwidth W relates to Nyquist bandwidth: W_0 . $\rho \in \langle 0, 1 \rangle$: if $\rho = 0$

→ pure *sinc* pulse is generated, if $\rho = 1$ pure rectangular pulse is generated. Roll-off factor defines a real bandwidth necessary to transmit pulses with symbol rate R_s :

$$\text{in case of baseband it is: } W = \frac{1}{2}(1 + \rho)R_s \quad (8.15)$$

$$\text{in case of bandpass it is: } W = (1 + \rho)R_s \quad (8.16)$$

Typical values for wireless communication is $\rho \in \langle 0.2; 0.4 \rangle$. In practice, we try to use as small a value of ρ as possible because it minimizes the signal bandwidth. RC pulse shaping filter can be also described in frequency domain:

$$H(f) = \begin{cases} 1 & |f| < 2W_0 - W \\ \cos^2 \left\{ \frac{\pi}{4} \frac{|f| + W - 2W_0}{W - W_0} \right\} & 2W_0 - W < |f| < W \\ 0 & |f| > W \end{cases} \quad (8.17)$$

Exercise 8.7:

Signal transmitted via GEO satellite has bandwidth: $W = 36 \text{ [MHz]}$. Used modulation is QPSK. Calculate the maximal bit rate if for pulse shaping is used RCF with: $\rho \in 0.3$?

Solution:

Satellite data signal is QPSK modulated onto some carrier frequency, therefore we have to consider bandpass bandwidth:

$$W = (1 + \rho) R_s \Rightarrow R_s = \frac{W}{1 + \rho} = \frac{36 \times 10^6}{1 + 0.3} = 27.7 \text{ [Mbaud]}$$

In case of QPSK: $M = 4 \Rightarrow k = \log_2 M = \log_2 4 = 2$

$$R_s = \frac{R_b}{k} \Rightarrow R_b = k R_s = 2 \times 27.7 \times 10^6 = 55.4 \text{ [Mb/s]}$$

■

Exercise 8.8:

Calculate the required signal bandwidth for data rate: $R_b = 100 \text{ [Mb/s]}$ if pulses are shaped using RCF with: $\rho \in 0.3$? Consider:

- a) 2 state modulation signal
- b) 8 state modulation signal

Solution:

a)

$$M = 2 \Rightarrow k = \log_2 M = \log_2 2 = 1$$

$$R_s = \frac{R_b}{k} = \frac{100 \text{ [Mb/s]}}{1} = 100 \text{ [Mbaud]}$$

$$W = \frac{1}{2} (1 + \rho) R_s = \frac{1}{2} \times (1 + 0.3) \times 100 \times 10^6 = 65 \text{ [MHz]}$$

b)

$$M = 8 \Rightarrow k = \log_2 M = \log_2 8 = 3$$

$$R_s = \frac{100 \text{ [Mb/s]}}{3} \doteq 33.333 \text{ [Mbaud]}$$

$$W = \frac{1}{2} (1 + 0.3) \times 33.333 \times 10^6 \doteq 21.667 \text{ [MHz]}$$

■

Note: for multi-state modulation, a narrower transmitted bandwidth is sufficient (at the cost of a higher error rate (for the same power) and more complex equipment).

Lab 8.1: MATLAB exercises: Simulations of communications systems

In this chapter we will revisit elements of previous chapter and finally put everything together by showing how various components of the system interact and how to simply simulate the system using MATLAB. We also provide insights into statistical evaluation of the systems error performance – the calculation of the waterfall curve: the dependence of the system bit-, symbol- and frame- (or block-) error ratios *BER*, *SER*, *FER* (aka. *BLER*) and the key transmission parameter E_b/N_0 .

In all the exercises we will simulate a block oriented transmission, that is compatible with all digital communications systems: the link layer frame can be several hundreds or even thousands bytes long, and can be divided into smaller subblock: sometimes the subblock may represent each byte, with bits transmitted serially, other times, the codeword of the ECC present in the system is a good choice for the subblock. Depending on the modulation order, the bits of the codewords may further be grouped to form modulation symbols. In any case, we take advantage of the MATLABs natural matrix representation, where each block (or frame) will be represented by a matrix, and the columns of this matrix will represent subblocks. Fig . 8.12 show the block structure, used in all the simulations.

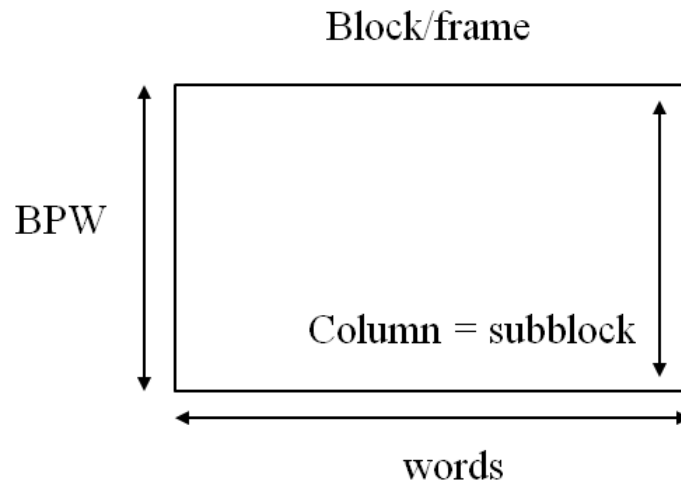


Fig. 8.12: General matrix representation of a basic transmission unit – block.

The BPW in Fig. 8.12 stands for Bits per Word and will be slightly different in each of the simulations. The number of columns of the block could be named subblocks or words. The total number of bits in the block (or frame) we denote BPF (bits per Frame) and calculate easily:

$$\text{BPF} = \text{BFW} \times \text{words}$$

We structure the simulations to blocks not just to simulate the link-layer frame transmission, but also to provide some level of throughput to our simulation: It could be easy to simulate the transmission of each bit (or symbol) completely separately in a loop, but the simulation would be prohibitively slow. Using a matrix orientation we utilize the built-in MATLAB optimizations to accelerate our simulation. All our simulations will be signal space simulations, meaning that we will represent each modulation symbol by a single scalar (real or complex) value – a point in the IQ plane. We will not go deeper into modelling neither pulse shaping nor matched filtering, or any other related topics. These are out of scope of this

short book. Fig. 8.13 shows the generic simulation model – a model that we will gradually develop over the course of several exercises.

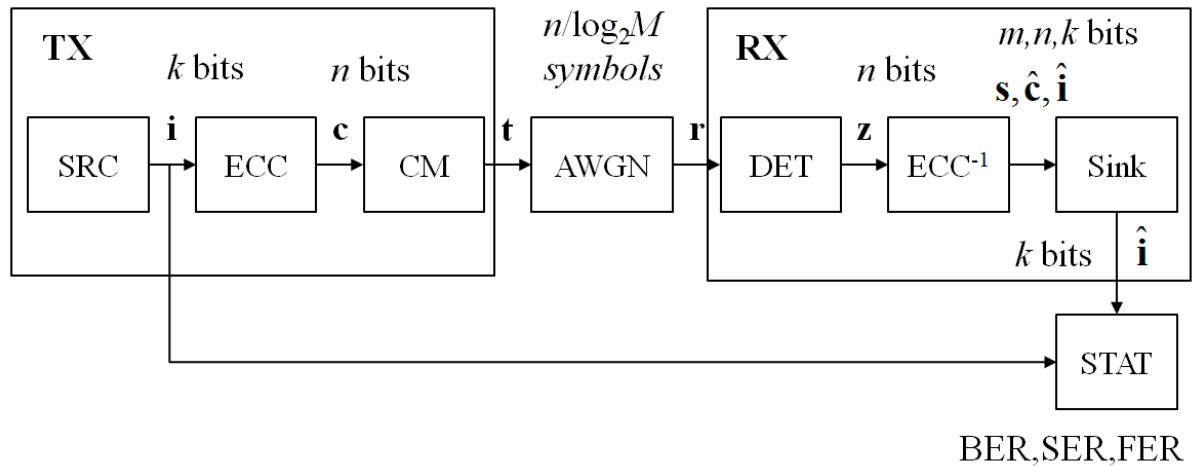


Fig. 8.13: Generic simulation model containing all the optional blocks. The number of bits at each interface specifies the BPW (or number of rows of the matrix representing the frame).

We need to implement simple software models for each of the blocks: TX is the transmitter, that will have to consist from at minimum one block: the SRC or pseudo-random binary source block. The ECC block in the transmitter performs encoding algorithm of a selected Error Control Code, introducing redundancy and thus increasing the number of rows of the matrix to process. The CM stands for Constellation Mapping and will eventually model the process of mapping bits to modulation symbols from the IQ plane. This means that matrices $\mathbf{t}$ and $\mathbf{r}$ will be matrices of real or complex numbers. The AWGN channel model will add a pseudo-randomly generated noise samples to the transmission, introducing errors. The RX, or receiver will then implement models of a detector (DET), that will perform decision regarding each of the received IQ sample and return a corresponding binary value. The erroneous codewords $\mathbf{z}$ will then enter the decoder block ECC⁻¹, implementing a decoding algorithm. This will be by far the most computationally intensive block in the cascade. The resulting estimates of the codeword, and data word, would then be forwarded to upper layers in a real communications system, as modelled by a stub-block Sink. In our simulations, we will have the luxury to have the originally transmitted bits in TX available for comparison with the received bits in RX, so we will be able to simply calculate various error statistics.

MATLAB Exercise 8.1.1: Simple BSC channel

Fig 8.14. shows the ultimate simplification of the system model shown in Fig. 8.13, with most of the blocks omitted. The Binary Symmetric Channel (BSC) channel model can be derived as by joining together a more detailed model of the system, but we don't need to go into more details. Since we already implemented the basic routines for Hamming ECC in Chapter 3 Lab 3.1, and we will reuse our implementations in the exercises below.

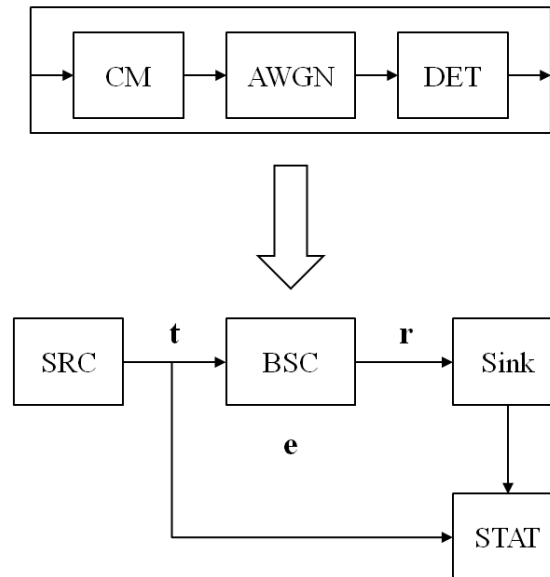


Fig. 8.14: Simulation for a simple BSC channel model.

What will be different this time will be the matrix-orientation of the transmission, and error statistics evaluation.

To implement this exercise, you need to open the basic file `SimSimpleBSC.m`, form the `assignments` subfolder.

MATLAB Exercise 8.1.1a: Uniform binary source

Implement the function given the prototype: `function data = binarySource( rows, columns )` that generates random binary matrix with uniform distribution - that is ones must occur with the same probability as zeros. Use the `rand()` built-in in your implementation. Do not implement the function by generating a precise number of ones and permuting their positions.

MATLAB Exercise 8.1.1b: BSC channel model

Implement the function given the prototype:

`function errors = binaryErrorMatrix( pb, rows, columns )`

that generates random binary matrix. The number of ones in this matrix must follow the probability specified by the parameter `pb`. Use the `rand()` built-in in your implementation. Do not implement the function by generating a precise number of ones and permuting their positions.

MATLAB Exercise 8.1.1c: Evaluate BER - Bit Error Ratio

Implement the function given the prototype: `function dHamm = computeHammingDistance(U,V)` that computes Hamming distance between two matrices.

After implementing the above exercises, run the script for several times with the default value of the `words` variable (10). You should observe ever changing and relatively high difference between the desired error probability `pb`, and its statistically calculated counterpart BER. Change `words` to larger values (100, 1000, ...) and observe the convergence of the statistically calculated BER towards the theoretical requested `pb`.

MATLAB Exercise 8.1.2: BSC channel with Hamming ECC

To implement this exercise, you need to open the basic file `SimBSCECC.m`, from the `assignments` subfolder and you will have to modify the provided code. Also you need to copy your working implementations of the Hamming code routines from chapter 3 Lab 3.1 to a subfolder named `hamming` first. Fig. 8.15 shows the system model used:

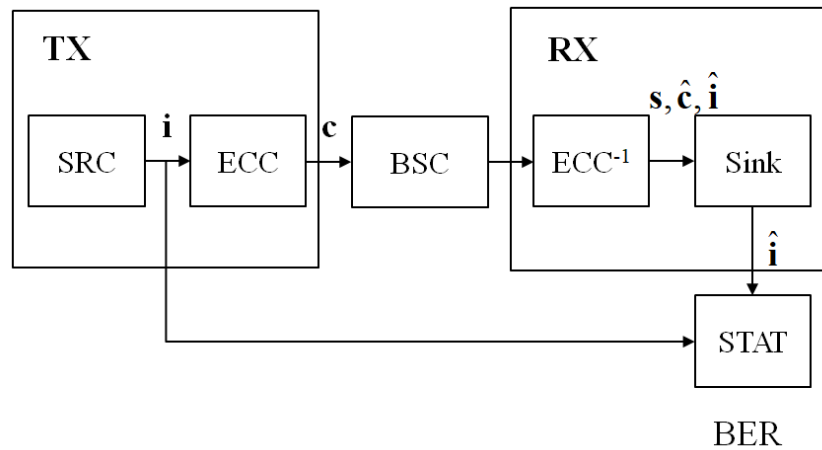


Fig. 8.15: Simulation model for a BSC channel and coded transmission.

MATLAB Exercise 8.1.2a: Add Hamming code encoding

Modify the simulation code to include Hamming code encoding procedure in the transmitter.

MATLAB Exercise 8.1.2b: Add Hamming code decoding

Modify the simulation code to include syndrome method decoding procedure of the Hamming code in the receiver. Also write a wrapper function given the prototype:

```
function CodewordEstimates = blockDecodeSyndromeMethod(H, SyndromeTable, Received)
```

that allows for using the existing row-vector oriented `decodeSyndromeMethod()` function in a column block-oriented simulation.

MATLAB Exercise 8.1.2c: Evaluate error ratios

Modify the simulation code to include error ratio evaluation with and without the ECC decoding.

MATLAB Exercise 8.1.2d: Find uncorrectable errors

Modify the simulation code to include analysis of multiple errors that cannot be decoded by the specified code. Find how many error words cannot be corrected.

MATLAB Exercise 8.1.2e: Find decoder induced errors

The uncorrectable error words result in inducing further errors in the decoding process.

Modify the simulation and write a function given the prototype:

```
function InducedErrors = findInducedErrors(RealChannelErrors, ErrorsAfterDecoding)
```

that finds the number of errors induced by the receiver ECC decoder.

After implementing the above exercises, run the script for several times with the default value of the `pb` variable (0.002). You should observe ever changing BER values, where the BER without ECC should almost always be higher than the BER with ECC (after decoding). Many times the BER after decoding will be zero. Now change the value of `pb` to 0.3, and run the script several times: this time the channel BER is so high, that the decoder introduces further errors and the BER with ECC will be usually higher than the BER without.

MATLAB Exercise 8.1.3: Uncoded AWGN channel with real-valued BPSK modulation

To implement this exercise, you need to open the basic file `SimAWGN_BPSK_R.m`, from the `assignments` subfolder and you will have to modify the provided code. Because we only use the Binary Phase Shift Keying (BPSK) modulation, and we choose the simplest form where the two prototypes take on real values +1 and -1, we may simplify the simulation by not using a 2D complex plane for the constellation, but using just the one-dimensional real axis. This has implications for calculations of the precise AWGN parameters, especially if we need to convert between noise variance and noise power. For a binary modulation, the number of symbols is equal to the number of logical bits, so the BER and SER ratios are the same.

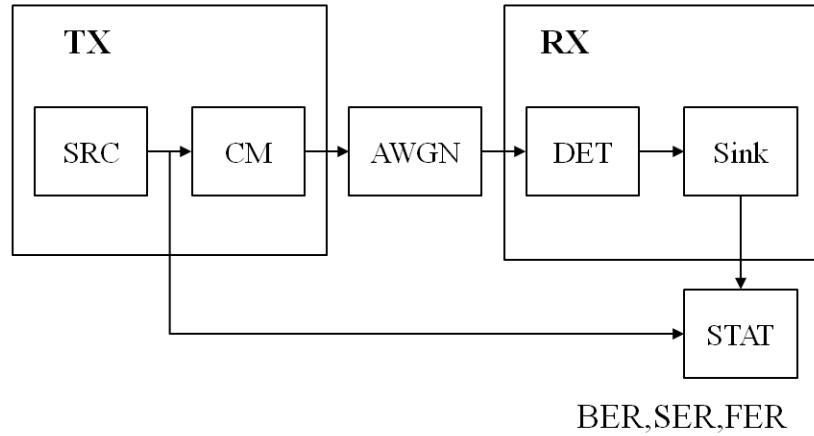


Fig. 8.16: Simulation model of an uncoded transmission over an AWGN channel.

The goal of the simulation is to obtain the waterfall curve: the dependence of the statistically evaluated BER depending on the key systemic parameter: (E_b/N_0) measured in decibels. Let us now first recap some important necessary equations. First we need to establish the relationship between two systemic parameters: Signal to Noise Ratio (SNR) and (E_b/N_0) . The following equation describes it:

$$SNR = \frac{S}{N} = \frac{\frac{E_s}{T_s}}{N_0 W} = \frac{\frac{E_s}{T_s}}{N_0 \frac{1}{T_s}} = \frac{E_s}{N_0} = \frac{k \cdot E_b}{N_0} = k \frac{E_b}{N_0} \quad (8.18)$$

Where the signal power is denoted S and the noise power denoted N . Since we deal with digital systems only, we deal with the mean power of the constellation prototype and only the in-band AWGN noise that actually affects the system. The prototype power can be rewritten as a ratio of energy per symbol E_s and characteristic interval T_s . Similarly the in-band noise power is a product of noise Power Spectral Density (PSD, denoted N_0) and system bandwidth W . For practical null-to-null bandwidth definition we can use the identity $W = 1/T_s$. Similarly, we can express the energy per symbol E_s as a product of energy per bit E_b and number of bits per symbol k : $E_s = k \cdot E_b$. Many times in simulations we may safely assume normalized energy $E_b = 1[\text{J}]$. The number of bits transmitted by each prototype can be calculated as $k = \log_2(M)$, where M is the number of prototypes (Note: symbol k here has nothing to do with the data word length defined in Chapter 3 for a LBC with parameters $[n, k, d_{min}]$. Usage of the same symbol for two different variables is unfortunate, but established practice).

As we see from the equation (8.18), it is customary to call (E_b/N_0) also SNR per bit. This equation is however in linear scale, and what is often needed is the logarithmic scale expressed in decibels. To

convert any quantity from linear scale to dB, we use the well known eq. (8.19). The inverse conversion is left to the reader. For energies or powers, which are proportional to squared current or voltage, the bottom definition applies.

$$\begin{aligned} X_{dB} &= 20 \log_{10}(X) \text{ if } X \propto Y \\ X_{dB} &= 10 \log_{10}(X) \text{ if } X \propto Y^2 \end{aligned} \quad (8.19)$$

We can rewrite eq. (8.18) in decibels:

$$SNR_{dB} = 10 \log_{10} \left(k \frac{E_b}{N_0} \right) = 10 \log_{10}(k) + 10 \log_{10} \left(\frac{E_b}{N_0} \right) = k_{dB} + \left(\frac{E_b}{N_0} \right)_{dB} \quad (8.20)$$

The waterfall curve is a dependency of the Bit Error Ratio (BER) expressed in linear scale, but plotted using logarithmic axis scaling, on the (E_b/N_0) expressed in dB, but plotted using linear axis scaling. The example of such a plot is given in Fig. 8.20. The convenient MATLAB built-in `semilogy()` provides the described axis scaling for us, so we need not perform any extra steps.

Within these exercises we will deal with various forms of a binary BPSK modulation, all transmitting one bit per symbol, so SNR will be equal to (E_b/N_0) . We will denote BER as a value obtained statistically by simulation, while p_b will mean a very similar thing – the probability of error, but calculated from theoretical formulas provided in the previous subchapters. If our simulation is constructed correctly, we may intuitively expect, that the statistically obtained BER will converge towards p_b . The size of the dataset processed by the simulation - the number of bit (or block/frame) transfers we perform will directly impact the succes of this convergence.

The BPSK modulation consist only of two antipodal prototypes, it is considered a one dimensional (1D) modulation. The quaternary QPSK consists of four prototypes placed as corners of a square in the 2D complex (or IQ) plane, aka the signal space.

Howeverm, we can design three main forms of the BPSK constellation: a real-one, and imaginary one, and an actual two-dimensional one. Fig. 8.17 illustrates:

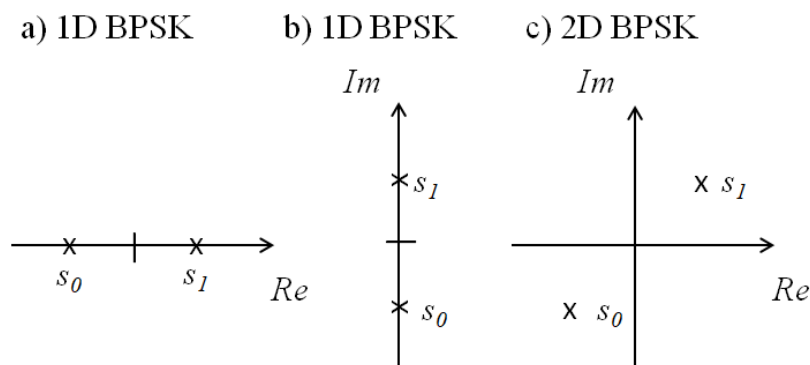


Fig. 8.17: Three forms of BPSK: first are considered 1D (a)b)).
The third one c) may be sometimes considered 2D.

While the first two constellations are very simple, the third, truly complex form of BPSK seems to be unnecessserili complicated. Especially when it comes to detection in the receiver: while the detector for cases a) and b) only needs to compare the received sample with zero and can immediately return the bit

estimate (for instance by returning the sign bit), the source code for determining, whether a received complex sample z is closer to s_1 or to s_0 involves squaring of x and y component differences.

The 1D BPSK is also usefull when thinking about the 2D QPSK modulation. This is illustrated in Fig. 8.18:

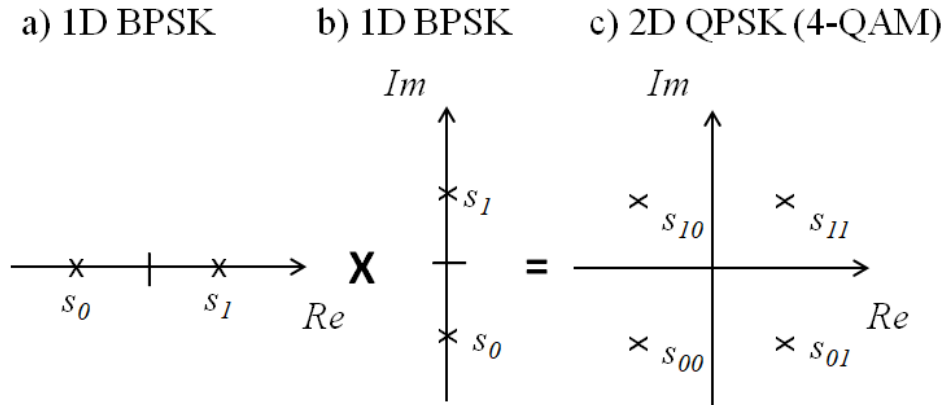


Fig. 8.18: A 2D QPSK can be understood as a Cartesian products of two 1D BPSK modulations.

There are many similar forms of QPSK which will differ only by their rotation

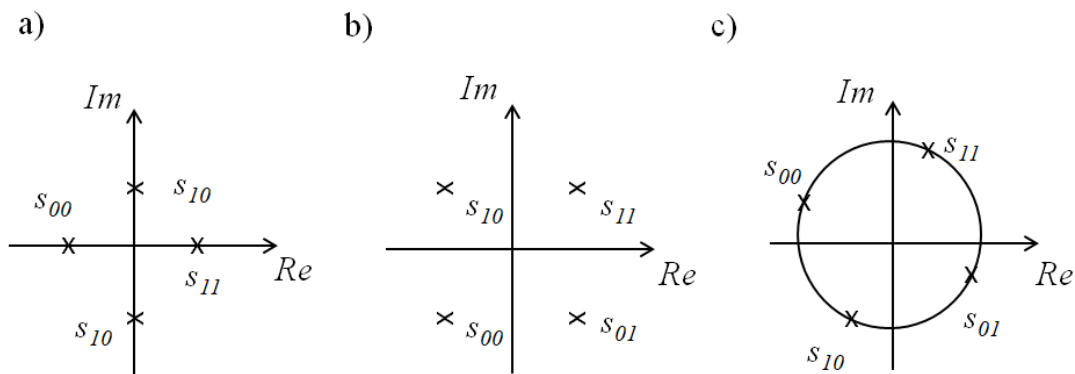


Fig. 8.19: Various forms of QPSK.

Since we like to simplify our simulations as much as possible, we will assume normalized transmitter energy per bit $E_b = 1$ and zero channel attenuation. This means, the magnitudes of the prototypes will be equal to 1 for all the modulations in question. This determines the exact values of the prototypes:

Real-valued BPSK: $s_0 = -1, s_1 = 1$

Imaginary-valued BPSK: $s_0 = -j, s_1 = j$

Complex-BPSK rotated $\pi/4$: $s_0 = -\left(\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j\right), s_1 = \left(\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j\right)$

QPSK: $s_{11} = 1, s_{10} = j, s_{00} = -1, s_{01} = -j$

$$\text{4-QAM: } s_{00} = -\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}j, s_{01} = \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}j, s_{11} = \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j, s_{10} = -\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j$$

The last piece of the puzzle we need to implement our simulations is the relationship between the AWGN noise PSD and a more practical metric – its variance σ^2 .

As derived in literature, the noise variance at the output of the matched filter, for a 1D modulation is numerically equal to:

$$\sigma^2 = \frac{N_0}{2} \quad (8.21)$$

MATLAB Exercise 8.1.3a: Real-valued BPSK modulator

Implement a function given the prototype:

```
function SignalSpaceSymbols = modulateBPSK( LogicalSymbols )
```

that performs all digital BPSK modulation. The output should be signal space (IQ plane) prototypes.

Use the following mapping of the logical binary signal to signal space prototype: 0 > -1, 1 > +1.

The input and output may be a scalar, vector or a matrix.

MATLAB Exercise 8.1.3b: Channel noise power

Modify this simulation to use different representations of channel noise power. Use the built-in `randn()` to generate pseudo-random AWGN noise approximation. Rescale its output to proper variance. Use the scaling properties of random variable moments:

For X a random variable with moments $\mu_x = E[X]$ and $\sigma_x^2 = \text{var}(X)$, the moments of a transformed random variable $Y = aX$ are: $\mu_y = a\mu_x$ and $\sigma_y^2 = a^2\sigma_x^2$.

MATLAB Exercise 8.1.3c: Real-valued BPSK detection

Implement a BPSK detector function

MATLAB Exercise 8.1.3d: Theoretical and simulational results comparison

Modify this simulation to include a theoretical p_b value for the given E_b/N_0 vector to be plotted together with the simulation results.

After implementing the above exercises, run the script for several times with the default value of the `words` variable (1) and observe the resulting waterfall curve. The statistically obtained curve should be slightly different each time you run the simulation. As demonstrated by Fig. 8.20, it should also not match the theoretical curve completely. Then raise the value of `words` to 100 and observe the results. Fig. 8.21 demonstrates the almost total overlap of the resulting curves.

The disadvantage of this type of simulation lies in the `for` cycle defining a pre-set number of iterations and therefore the dataset size. What would be a proper implementation would be a `while` cycle set to terminate after a certain number of errors is collected. The reader is encouraged to rewrite the simulation in this way.

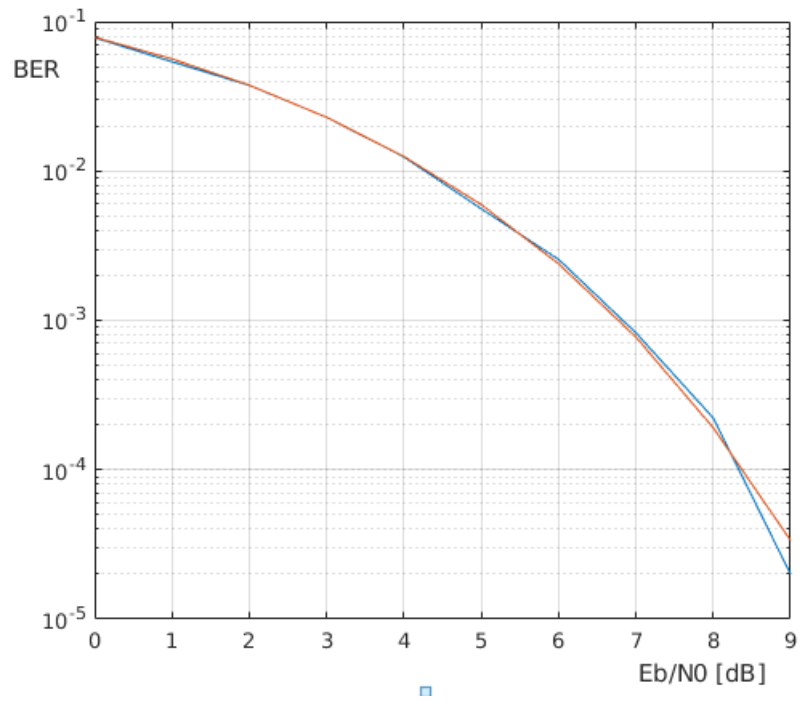


Fig. 8.20: Waterfall curve for BPSK modulation: Theoretical value (red) and simulation results (blue). The curves differ because of too small dataset size (the number of words in each block was set to 1).

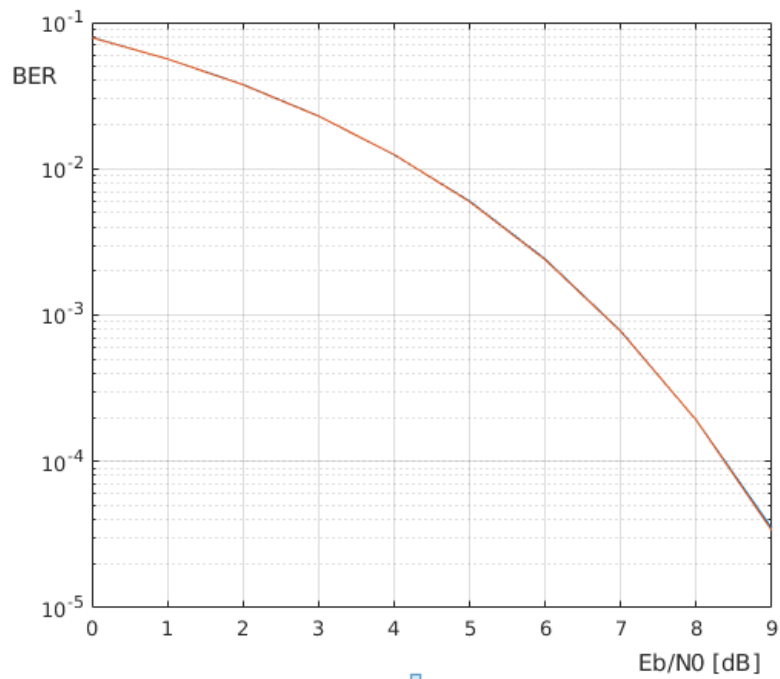


Fig. 8.21: Waterfall curve for BPSK modulation: Theoretical value (red) and simulation results (blue). The curves completely overlap, indicating correct modelling and large-enough data set (the number of words processed in each block was set to 100).

Chapter 9: Reed-Solomon Codes

Galois fields

Finite fields, also known as Galois fields and denoted $GF(q)$ are widely used in computing and digital communications. The q parameter defines the number of field elements. These can often be numbered $0, 1, 2, \dots, q-1$. For $GF(q)$ to exist, the q parameter must be a prime number or a power of prime. For example the fields $GF(2)$ and $GF(4)$ exist, while $GF(6)$ doesn't. The fields most widely used in computing are based on powers of two: $GF(2^r)$ where for $r = 1$ the elements of the field are binary numbers 0 and 1, and for $r > 1$ the elements of these fields are binary vectors of length r . So the elements of $GF(4)$ are binary vectors $\{00, 01, 10, 11\}$. There are two binary operations defined on a $GF(q)$: addition and multiplication. For $GF(2)$ these are defined as the well known logical operations XOR and AND. For a prime-based fields $GF(q)$ where q is a prime (and not a power of prime), we can define these operations simply:

$$\begin{aligned} \text{addition: } x \oplus y &= (x + y) \bmod q = \begin{cases} x + y & \text{if } 0 \leq x + y \leq q - 1 \\ x + y - q & \text{if } q - 1 < x + y \end{cases} \\ \text{multiplication: } x \otimes y &= (x \cdot y) \bmod q = \begin{cases} x \cdot y & \text{if } 0 \leq x \cdot y \leq q - 1 \\ x \cdot y - \left\lfloor \frac{x \cdot y}{q} \right\rfloor \cdot q & \text{if } q - 1 < x \cdot y \end{cases} \end{aligned}$$

Example 1: Construct a table of addition and multiplication for $GF(5)$.

Solution: The elements of $GF(5)$ will be integer numbers $\{0,1,2,3,4\}$. We use the definitions of addition and multiplication defined above. We can write the tables easily:

$\oplus$	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

$\otimes$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

■

Polynomial form of Galois field

For the practically used $GF(q)$ where q is a power of 2, $q = 2^r$, it is useful to define a field generator polynomial $p(x)$ of degree r , and exponential form of the primitive element: α . Field generator polynomial can generate all non-zero elements of the given finite field: $\{\alpha^0, \alpha^1, \alpha^2, \dots, \alpha^{q-2}\}$.

$$\alpha^i = x^i = x^i \bmod p(x); \quad i = 0, 1, \dots, q-2 \quad (9.1)$$

For a given polynomial to be a generator polynomial it has to be a primitive polynomial.

Exercise 9.1:

Generate the elements of a finite field GF(4) if the generator polynomial is $p(x) = x^2 + x + 1$.

Solution:

binary form	exponential form	polynomial form
00	$\alpha^{-\infty} = 0$	0
01	$\alpha^0 = 1$	x^0
10	α^1	x^1
11	α^2	$x+1$

Note: $\alpha^3 = x^3 \rightarrow x^3 \bmod p(x) = 1 \rightarrow \alpha^3 = 1$

■

Exercise 9.2:

Generate the elements of a finite field GF(8) if the generator polynomial is $p(x) = x^3 + x + 1$. Compute the results of the following equations:

$$\begin{aligned} \alpha^6 + \alpha^4 = ? & \quad \alpha^8 \cdot \alpha^3 = ? \\ \alpha^3 - \alpha^{11} = ? & \quad \alpha^3 : \alpha^{18} = ? \end{aligned}$$

Solution:

binary form	exponential form	polynomial form
000	0	0
001	α^0	x^0
010	α^1	x^1
100	α^2	x^2
011	α^3	$x + 1$
110	α^4	$x^2 + x$
111	α^5	$x^2 + x + 1$
101	α^6	$x^2 + 1$

$$\begin{aligned} \alpha^6 + \alpha^4 &= (x^2 + 1) + (x^2 + x) = x + 1 = \alpha^3 \\ &= (101) + (110) = (011) = \alpha^3 \\ \alpha^3 - \alpha^{11} &= \alpha^3 - \alpha^4 = (x + 1) - (x^2 + x) = x^2 + 1 = \alpha^6 \\ &= (011) + (110) = (101) = \alpha^6 \\ \alpha^8 \cdot \alpha^3 &= \alpha^{8+3} = \alpha^{11} = \alpha^4 \\ &= (x) \cdot (x + 1) = x^2 + x = \alpha^4 \\ \alpha^3 : \alpha^{18} &= \alpha^{3-18} = \alpha^{-15} = \alpha^6 \end{aligned}$$

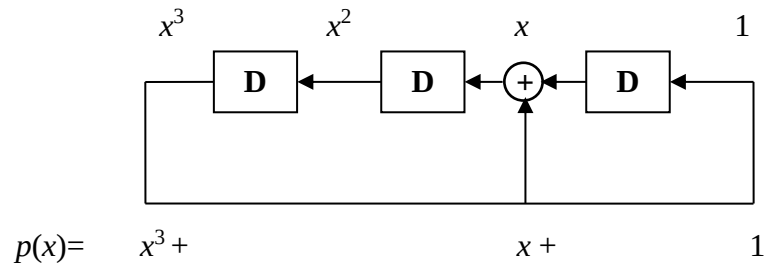
■

Exercise 9.3:

Sketch the hardware circuit for generating the elements of a finite field GF(8) if the generating polynomial is $p(x) = x^3 + x + 1$.

Solution:

One of the possible realizations is the following:



Note:

The **D** denotes the memory element. Each delay, i.e., after each memory term the order of the input signal increases, therefore we can mark the circuit points from 1 to x^3 . If we want to construct a circuit to generate the array of elements of GF(8) with generating polynomial $p(x) = x^3 + x + 1$, we need to do feedback on the point corresponding to x and 1 (if $p(x) = 0 \rightarrow x^3 + x + 1 = 0 \rightarrow x^3 = x + 1$). After filling the memory elements with the primitive element: $\alpha = x$ (010), we will successively generate (after clocking) all the elements of the array. In the next clock cycle, the combination 100 will be written to the memory circuits, in the next 001, and so on.

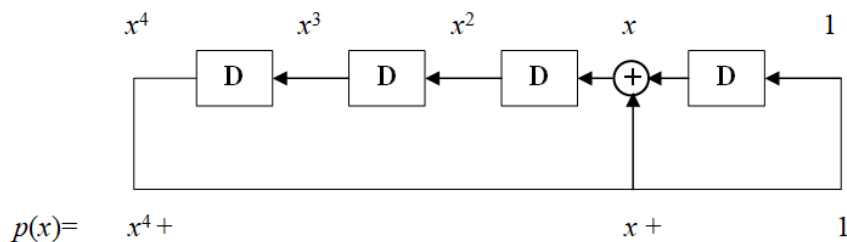
■

Exercise 9.3:

Sketch the hardware circuit to generate the elements of a finite field GF(16) if the generating polynomial is $p(x) = x^4 + x + 1$. Generate all the elements of the field.

Solution:

One of the possible realizations is the following:



binary form	exponential form	polynomial form
0000	0	0
0001	α^0	x^0
0010	α^1	x^1
0100	α^2	x^2
1000	α^3	x^3
0011	α^4	$x + 1$
0110	α^5	$x^2 + x$
1100	α^6	$x^3 + x^2$
1011	α^7	$x^3 + x + 1$
0101	α^8	$x^2 + 1$
1010	α^9	$x^3 + x$
0111	α^{10}	$x^2 + x + 1$
1110	α^{11}	$x^3 + x^2 + x$
1111	α^{12}	$x^3 + x^2 + x + 1$
1101	α^{13}	$x^3 + x^2 + 1$
1001	α^{14}	$x^3 + 1$
0001	α^{15}	1

■

Reed Solomon codes

Despite their long history of usage, and well know competition from more recent ECCs such as turbo-codes or LDPC codes, the Reed Solomon codes (denoted further as RS codes) do not loose their appeal and are still widely used. While their usage in storage for the implementation of RAID arrays is notorious, lesser known is their utilization in the most recent multi-gigabit 200Gbps and 400 Gbps IEEE 802.3 - 2018 standard. Their current usefullnes in the ultra-high speed and ultra low-latency context arises from existing fast decoding implementations, much faster than the iteratively decoded turbo and LDPC codes. Furthermore, their ability to correct bursts of errors makes them unique among the practically used ECC schemes.

RS codes encoding

For the extension of binary field field: $GF(q) = GF(2^r)$, where $r \in \mathbb{N}$, $r > 1$ we can construct RS codes with the parameters $[n, k, d_{\min}]$, where:

$n = q - 1$ - is the codeword length

k - variable: $k \in \langle 1, q - 3 \rangle$ - is the length of the information word

$d_{\min} = (n - k) + 1$ - is the minimum Hamming distance (RS are of MDS type)

$t = \left\lfloor \frac{n - k}{2} \right\rfloor$ - is the number of errors that RS code is able to correct

Generator polynomial of RS codes has to have $2t$ consecutive roots. Therefore:

$$g(x) = (x - \alpha^j)(x - \alpha^{j+1})(x - \alpha^{j+2}) \dots (x - \alpha^{j+2t-1}) \quad (9.2)$$

Usually: $j = 0$.

The encoding can be:

1) nonsystematic: $c(x) = i(x).g(x)$

2) systematic:

$$c(x) = i(x).x^m + r(x) \quad r(x) = i(x).x^m \bmod g(x) \quad m = \deg\{g(x)\} = n - k$$

Exercise 9.4:

Let assume RS code $[7, 3, 5]$ constructed over $GF(8)$, defined by primitive polynomial:

$p(x) = x^3 + x + 1$. Encode the following information: $[0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1]$:

a) nonsystematic way

b) systematic way

Solution:

Finding $g(x)$:

This particular finite field was generated in Exercise 9.2.

$q = 8 \Rightarrow n = 7$, $d_{\min} = 5 \Rightarrow t = 2 \Rightarrow g(x)$ will have 4 roots

$$k = n + 1 - d_{\min} = 7 + 1 - 5 = 3$$

$$\begin{aligned} g(x) &= (x + \alpha^0)(x + \alpha^1)(x + \alpha^2)(x + \alpha^3) = (x^2 + (\alpha^0 + \alpha^1)x + \alpha^0.\alpha^1) \cdot (x^2 + (\alpha^2 + \alpha^3)x + \alpha^2.\alpha^3) = \\ &= (x^2 + \alpha^3x + \alpha^1).(x^2 + \alpha^5x + \alpha^5) = x^4 + \alpha^2x^3 + \alpha^5x^2 + \alpha^5x^1 + \alpha^6 \end{aligned}$$

Verification:

Substitute all its roots $\{\alpha^0, \alpha^1, \alpha^2, \alpha^3\}$ into the resulting $g(x)$ and the result must always be 0.

$$g(\alpha^0) = (\alpha^0)^4 + \alpha^2(\alpha^0)^3 + \alpha^5(\alpha^0)^2 + \alpha^5(\alpha^0)^1 + \alpha^6 = \alpha^0 + \alpha^2 + \alpha^5 + \alpha^5 + \alpha^6 = 0$$

$$g(\alpha^1) = (\alpha^1)^4 + \alpha^2(\alpha^1)^3 + \alpha^5(\alpha^1)^2 + \alpha^5(\alpha^1)^1 + \alpha^6 = \alpha^4 + \alpha^5 + \alpha^0 + \alpha^6 + \alpha^6 = 0$$

$$g(\alpha^2) = (\alpha^2)^4 + \alpha^2(\alpha^2)^3 + \alpha^5(\alpha^2)^2 + \alpha^5(\alpha^2)^1 + \alpha^6 = \alpha^1 + \alpha^1 + \alpha^2 + \alpha^0 + \alpha^6 = 0$$

$$g(\alpha^3) = (\alpha^3)^4 + \alpha^2(\alpha^3)^3 + \alpha^5(\alpha^3)^2 + \alpha^5(\alpha^3)^1 + \alpha^6 = \alpha^5 + \alpha^4 + \alpha^4 + \alpha^1 + \alpha^6 = 0$$

a) Nonsystematic encoding:

We partition the input information: information: $[0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1]$ by r bits, in our case $r = 3$,

since we are using finite field: $GF(2^3)$, then: $\mathbf{i} = [011111001] = [\alpha^3, \alpha^5, \alpha^0]$ and

$$i(x) = \alpha^3x^2 + \alpha^5x + \alpha^0.$$

$$\begin{aligned}
c(x) &= i(x) g(x) \\
&= (\alpha^3 x^2 + \alpha^5 x + \alpha^0) \cdot (x^4 + \alpha^2 x^3 + \alpha^5 x^2 + \alpha^5 x^1 + \alpha^6) \\
&= \alpha^3 x^6 + \alpha^5 x^5 + \alpha^1 x^4 + \alpha^1 x^3 + \alpha^2 x^2 + \alpha^5 x^5 + \alpha^0 x^4 + \alpha^3 x^3 + \alpha^3 x^2 + \alpha^4 x + \\
&\quad + \alpha^0 x^4 + \alpha^2 x^3 + \alpha^5 x^2 + \alpha^5 x + \alpha^6 \\
&= \alpha^3 x^6 + 0 \cdot x^5 + \alpha^1 x^4 + \alpha^6 x^3 + 0 \cdot x^2 + \alpha^0 x + \alpha^6
\end{aligned}$$

Verification:

Substitute all roots of $g(x)$: $\{\alpha^0, \alpha^1, \alpha^2, \alpha^3\}$ into the resulting form of $c(x)$ and the result must always be 0:

$$\begin{aligned}
c(\alpha^0) &= 0 \\
c(\alpha^1) &= 0 \\
c(\alpha^2) &= 0 \\
c(\alpha^3) &= 0
\end{aligned}$$

b) Systematic encoding:

$$m = \deg \{g(x)\} = 4$$

$$i(x) \cdot x^4 = (\alpha^3 x^6 + \alpha^5 x^5 + \alpha^0 x^4)$$

$$r(x) = i(x) \cdot x^m \bmod g(x) \rightarrow \alpha^3 x^6 + \alpha^5 x^5 + \alpha^0 x^4 : x^4 + \alpha^2 x^3 + \alpha^5 x^2 + \alpha^5 x^1 + \alpha^6 = \alpha^3 x^2 + \alpha^3$$

$$\underline{\alpha^3 x^6 + \alpha^5 x^5 + \alpha^1 x^4 + \alpha^1 x^3 + \alpha^2 x^2}$$

$$\alpha^3 x^4 + \alpha^1 x^3 + \alpha^2 x^2$$

$$\underline{\alpha^3 x^4 + \alpha^5 x^3 + \alpha^1 x^2 + \alpha^1 x + \alpha^2}$$

$$\alpha^6 x^3 + \alpha^4 x^2 + \alpha^1 x + \alpha^2 = r(x)$$

$$c(x) = i(x) \cdot x^m + r(x)$$

$$c(x) = \alpha^3 x^6 + \alpha^5 x^5 + \alpha^0 x^4 + \alpha^6 x^3 + \alpha^4 x^2 + \alpha^1 x + \alpha^2$$

$$\text{binary: } \underbrace{011 \ 111 \ 001110}_{i(x)x^4} \underbrace{010 \ 100}_{r(x)}$$

Verification:

Substitute all roots of $g(x)$: $\{\alpha^0, \alpha^1, \alpha^2, \alpha^3\}$ into the resulting form of $c(x)$ and the result must always be 0.

■

Exercise 9.5:

Primitive polynomial $p(x) = x^4 + x + 1$ defines a finite field of the form $GF(2^r)$. Over this field we want to construct an RS code with parameters $[n, 11, d_{\min}]$.

- Find the unknown code parameters: n and $d_{\min}$
- Find the generating polynomial of this code: $g(x)$ in conjunctive form.

Solution:

a)

$$\deg\{g(x)\} = r \Rightarrow GF(2^4) \Rightarrow n = q - 1 = 16 - 1 = 15$$

$$d_{\min} = (n - k) + 1 = 15 - 11 + 1 = 5$$

$$t = \left\lfloor \frac{n-k}{2} \right\rfloor = \left\lfloor \frac{15-11}{2} \right\rfloor = 2$$

b)

$t = 2 \Rightarrow g(x)$ must have 4 consecutive roots

$$g(x) = (x + \alpha^0)(x + \alpha^1)(x + \alpha^2)(x + \alpha^3)$$

■

Exercise 9.6:

We have a $[7, 5, 3]$ RS code constructed over the field $GF(2^r)$ defined by the primitive polynomial $p(x) = x^3 + x + 1$. Encode the information: $i = (\alpha^4, \alpha^1, \alpha^5, \alpha^0, \alpha^2)$ systematic way using:

- polynomial form
- equations

Solution:

$$t = 1 \Rightarrow 2 \text{ roots: } g(x) = (x + \alpha^0)(x + \alpha^1) = x^2 + \alpha^3 x + \alpha$$

a)

$$c(x) = i(x) \cdot x^m + r(x) \quad r(x) = i(x) \cdot x^m \bmod g(x) \quad m = \deg\{g(x)\} = 2$$

$$i(x) \cdot x^2 = (\alpha^4 x^4 + \alpha^1 x^3 + \alpha^5 x^2 + \alpha^0 x^1 + \alpha^2). x^2 = \alpha^4 x^6 + \alpha^1 x^5 + \alpha^5 x^4 + \alpha^0 x^3 + \alpha^2 x^2$$

$$\begin{array}{r}
i(x) \cdot x^2 : g(x) = \alpha^4 x^6 + \alpha^1 x^5 + \alpha^5 x^4 + \alpha^0 x^3 + \alpha^2 x^2 : x^2 + \alpha^3 x + \alpha \\
\underline{\alpha^4 x^6 + \quad x^5 + \alpha^5 x^4} \\
\alpha^3 x^5 + \quad x^3 \\
\underline{\alpha^3 x^5 + \alpha^6 x^4 + \alpha^4 x^3} \\
\alpha^6 x^4 + \alpha^5 x^3 + \alpha^2 x^2 \\
\underline{\alpha^6 x^4 + \alpha^2 x^3 + \quad x^2} \\
\alpha^3 x^3 + \alpha^6 x^2 \\
\underline{\alpha^3 x^3 + \alpha^6 x^2 + \alpha^4 x} \\
\alpha^4 x = r(x)
\end{array}$$

$$c(x) = \alpha^4 x^6 + \alpha^1 x^5 + \alpha^5 x^4 + \alpha^0 x^3 + \alpha^2 x^2 + \alpha^4 x$$

Verification:

$$\begin{aligned}
c(\alpha^0) &= 0 \\
c(\alpha^1) &= 0
\end{aligned}$$

b)

$$i(x) = \alpha^4 x^4 + \alpha^1 x^3 + \alpha^5 x^2 + \alpha^0 x + \alpha^2$$

$$c(x) = i(x) \cdot x^2 + r(x) = \alpha^4 x^6 + \alpha^1 x^5 + \alpha^5 x^4 + \alpha^0 x^3 + \alpha^2 x^2 + r(x)$$

$$c(x) = c_6 x^6 + c_5 x^5 + \dots + c_1 x + c_0 \Rightarrow c_6 = \alpha^4, c_5 = \alpha^1, c_4 = \alpha^5, c_3 = \alpha^0, c_2 = \alpha^2$$

We set up the equations by plugging all the roots of $g(x)$ into $c(x)$:

$$c(\alpha^0) = \alpha^4 + \alpha^1 + \alpha^5 + \alpha^0 + \alpha^2 + c_1 \alpha^0 + c_0 = 0$$

$$c(\alpha^1) = \alpha^4 \alpha^6 + \alpha^1 \alpha^5 + \alpha^5 \alpha^4 + \alpha^0 \alpha^3 + \alpha^2 \alpha^2 + c_1 \alpha^1 + c_0 = 0$$

$$c_1 \alpha^0 + c_0 = \alpha^4$$

$$c_1 \alpha^1 + c_0 = \alpha^5 \Rightarrow c_0 = c_1 \alpha^1 + \alpha^4$$

$$\Rightarrow c_0 = 0, c_1 = \alpha^4$$

■

RS codes decoding

We assume an additive error channel, so the received word = codeword + error word:

$v(x) = c(x) + e(x)$, then: $\hat{c}(x) = v(x) + e(x) \rightarrow$ estimation of the transmitted codeword
(corrected received vector)

$c(x) = c_{n-1}x^{n-1} + \dots + c_1x + c_0$. Coefficients: $c_{n-1}, \dots, c_0$ are the elements of $\text{GF}(p^n)$ and they are in the form of α^i .

To correct the received vector for RS codes, we need to know two pieces of information - the error location and the error value.

If t errors occur, then:

$$\bar{s}_k = \sum_{i=1}^t Y_i X_i^k, \quad k = 0, 1, \dots, 2t - 1 \quad (9.3)$$

X_i denotes the locator of the i -th error; Y_i denotes the value of the i -th error in the received vector. The error locator is an element of the field $GF(q)$ such that its exponent determines the location of the error.

To correct t errors, $2t$ roots of $g(x)$ are needed from which $2t$ syndromes are computed $\Rightarrow$ for each root of $g(x)$ there is one syndrome.

In decoding, the syndrome elements: s_k are computed by plugging all roots of $g(x)$ into the received vector:

$$s_k = v(\alpha^k) = c(\alpha^k) + e(\alpha^k), \quad k = 0, 1, \dots, 2t - 1 \quad (9.4)$$

For cyclic codes if $s = 0 \Rightarrow v(x) = c(x)$, if $s \neq 0$ the error is at the corresponding position of x^i .

For RS codes, if one syndrome element = 0, $v(x)$ may or may not be $c(x)$. We have to plug all roots of $g(x)$ into $s_k = v(\alpha^k)$. If all syndromes = 0, then: $v(x) = c(x)$. If $s_k \neq 0$ we know that an error has occurred, but we don't know where the errors are or how many errors there are.

Procedure for decoding RS codes:

1. syndrome calculation

number of syndrome elements = number of roots of $g(x)$ = double the number of correctable errors ($2t$)

$$\bar{s}_k = v(\alpha^k), \quad k = 0, 1, \dots, 2t - 1$$

we plug all roots of $g(x)$ to $v(x) \Rightarrow$ we obtain all elements of syndrome vector

2. error location detection

we need to:

a) determine the locator polynomial:

$$\sigma(x) = x^t + \sigma_1 x^{t-1} + \dots + \sigma_{t-1} x^1 + \sigma_t \quad (9.5)$$

b) construct the locator polynomial using a system of syndrome equations:

$$\bar{s}_{t+i} + \sum_{j=1}^t \bar{s}_{t+i-j} \cdot \sigma_j = 0; \quad i = 0, 1, \dots, t-1 \quad j = 1, 2, \dots, t \quad (9.6)$$

t = number of correctable errors = number of equations

$$\begin{aligned}
\bar{S}_t + \bar{S}_{t-1}\sigma_1 + \bar{S}_{t-2}\sigma_2 + \cdots + \bar{S}_0\sigma_t &= 0 \\
\bar{S}_{t+1} + \bar{S}_t\sigma_1 + \bar{S}_{t-1}\sigma_2 + \cdots + \bar{S}_1\sigma_{t+1} &= 0 \\
&\vdots \\
\bar{S}_{2t-1} + \bar{S}_{2t-2}\sigma_1 + \bar{S}_{2t-3}\sigma_2 + \cdots + \bar{S}_{t-1}\sigma_t &= 0
\end{aligned} \tag{9.7}$$

From (9.7) it is possible to construct the locator polynomial $\sigma(x)$. Alternatively we can use matrix notation:

$$\begin{bmatrix} s_0 & s_1 & \cdots & s_{t-1} \\ s_1 & s_2 & & s_t \\ \vdots & & & \\ s_{t-1} & s_t & \cdots & s_{2t-2} \end{bmatrix} \cdot \begin{bmatrix} \sigma_0 \\ \sigma_1 \\ \vdots \\ \sigma_{t-1} \end{bmatrix} = \begin{bmatrix} s_t \\ s_{t+1} \\ \vdots \\ s_{2t-1} \end{bmatrix} \tag{9.8}$$

3. finding locators

To find roots of locator polynomial we use Chien algorithm: to the locator polynomial: $\sigma(x)$ we substitute for x all non-zero elements of the given finite field. If $\sigma(\alpha^k) = 0$ then α^k is the root of $\sigma(x) \Rightarrow X_i^k$ the locator of the i -th error will be an element of the field $\text{GF}(q)$ such that its exponent determines the location of the error.

4. calculation of the error values

Error values Y_i can be calculated from the system of equations (9.3).

Note: (9.3) hold only if the first root of $g(x)$ is α^0 .

5. correction of the received vector

add Y_i in place of X_i :

$$s_i = v(\alpha^i) = c(\alpha^i) + e(\alpha^i) = e(\alpha^i) \tag{9.9}$$

corrected message:

$$\hat{c}(x) = v(x) + e(x) \tag{9.10}$$

Exercise 9.7:

$\text{GF}(8)$ is defined by the primitive polynomial $p(x) = x^3 + x + 1$ and $p(\alpha^0) = 0$ (the first root is α^0). Let $t = 2$. The transmitted message is encoded with an RS code. Decode the received message: $v(x) = \alpha^5x^6 + \alpha^2x^5 + \alpha^4x^4 + \alpha x^3 + \alpha^0x^2 + \alpha^0x + \alpha^5$.

Solution:

First we have to find code parameters:

$$\left. \begin{aligned} n &= q - 1 = 8 - 1 = 7 \\ t &= \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor \Rightarrow d_{\min} = 2t - 1 = 5 \end{aligned} \right\} [7, 3, 5]_{\text{GF}(8)}$$

$$2t = n - k \Rightarrow k = n - 2t = 7 - 4 = 3$$

If $t = 2 \Rightarrow g(x)$ has 4 consecutive roots: $\alpha^0, \alpha^1, \alpha^2, \alpha^3$

Elements of given GF(8):

$\alpha^{-\infty}$	0	000
α^0	1	001
α^1	x	010
α^2	x^2	100
α^3	$x + 1$	011
α^4	$x^2 + x$	110
α^5	$x^2 + x + 1$	111
α^6	$x^2 + 1$	101

Syndrome calculation:

$$s_k = v(\alpha^k) = c(\alpha^k) + e(\alpha^k) = e(\alpha^k) \quad k = 0, 1, \dots, 2t - 1$$

$$v(x) = c(x) + e(x)$$

If all syndromes = 0 $\Rightarrow v(x) = c(x)$ (i.e. the transmission was without error, or the error changed the transmitted codeword to another codeword and was not detected). If at least 1 syndrome $\neq 0 \Rightarrow$ an error occurred, but we don't know how many of them.

$\Rightarrow$ to $v(x)$ we plug all roots of $g(x) \Rightarrow 4$ syndromes

$$x = \alpha^0 \rightarrow s_0 = v(\alpha^0) = \alpha^5 + \alpha^2 + \alpha^4 + \alpha + \alpha^0 + \alpha^0 + \alpha^5 = 0$$

$$x = \alpha^1 \rightarrow s_1 = v(\alpha^1) = \alpha^5 \alpha^6 + \alpha^2 \alpha^5 + \alpha^4 \alpha^4 + \alpha \alpha^3 + \alpha^0 \alpha^2 + \alpha^0 \alpha^1 + \alpha^5 = \alpha$$

$$x = \alpha^2 \rightarrow s_2 = v(\alpha^2) = \alpha^5 \alpha^{12} + \alpha^2 \alpha^{10} + \alpha^4 \alpha^8 + \alpha \alpha^6 + \alpha^0 \alpha^4 + \alpha^0 \alpha^2 + \alpha^5 = \alpha^5$$

$$x = \alpha^3 \rightarrow s_3 = v(\alpha^3) = \alpha^5 \alpha^{18} + \alpha^2 \alpha^{15} + \alpha^4 \alpha^{12} + \alpha \alpha^9 + \alpha^0 \alpha^6 + \alpha^0 \alpha^3 + \alpha^5 = \alpha^0$$

$$\bar{S} = [0, \alpha, \alpha^5, \alpha^0]$$

Finding locator polynomial:

$$\text{General form of the locator polynomial for } t = 2 \Rightarrow \sigma(x) = x^2 + \sigma_1 x + \sigma_2$$

The unknown coefficients σ_1, σ_2 must be determined using a system of syndrome equations:

I. $s_2 + \sigma_1 s_1 + \sigma_2 s_0 = 0$ ($\rightarrow$ starts from s_2 , since $t = 2$)

II. $s_3 + \sigma_1 s_2 + \sigma_2 s_1 = 0$ ($\rightarrow$ starts from s_3 since $t + 1 = 3$)

$$\left. \begin{array}{l} \alpha^5 + \sigma_1 \alpha + \sigma_2 0 = 0 \\ \alpha^0 + \sigma_1 \alpha^5 + \sigma_2 \alpha = 0 \end{array} \right\} \begin{array}{l} \text{for } s_i \text{ we input calculated elements} \\ \text{of } \bar{S} = [0, \alpha, \alpha^5, \alpha^0] \end{array}$$

$$\alpha^5 + \sigma_1 \alpha = 0 \quad \Rightarrow \quad \sigma_1 = \alpha^5 / \alpha = \alpha^4$$

$$\alpha^0 + \sigma_1 \alpha^5 + \sigma_2 \alpha = 0$$

$$\text{in II: } \alpha^0 + \alpha^4 \alpha^5 + \sigma_2 \alpha = 0$$

$$\alpha^6 + \sigma_2 \alpha = 0 \quad \Rightarrow \quad \sigma_2 = \alpha^6 / \alpha = \alpha^5$$

$$\underline{\underline{\sigma(x) = x^2 + \alpha^4 x + \alpha^5}} \quad \leftarrow \text{locator polynomial}$$

↓

$$\deg\{g(x)\} = 2 \Rightarrow 2 \text{ roots} = 2 \text{ errors}$$

Possible solutions:

- 1) system of linear equations
- 2) Chien algorithm (more simple)

Finding locators:

Chien algorithm:

$$\left. \begin{array}{l} \sigma(\alpha^0) = (\alpha^0)^2 + \alpha^4 \alpha^0 + \alpha^5 = \alpha^0 + \alpha^4 + \alpha^5 = 0 \\ \sigma(\alpha^1) = \alpha^2 + \alpha^4 \alpha^1 + \alpha^5 = \alpha^2 + \alpha^5 + \alpha^5 = \alpha^2 \\ \sigma(\alpha^2) = \alpha^4 + \alpha^4 \alpha^2 + \alpha^5 = \alpha^4 + \alpha^6 + \alpha^5 = \alpha^2 \\ \sigma(\alpha^3) = \alpha^6 + \alpha^4 \alpha^3 + \alpha^5 = \alpha^6 + \alpha^0 + \alpha^5 = \alpha^3 \\ \sigma(\alpha^4) = \alpha^8 + \alpha^4 \alpha^4 + \alpha^5 = \alpha^1 + \alpha^1 + \alpha^5 = \alpha^5 \\ \sigma(\alpha^5) = \alpha^{10} + \alpha^4 \alpha^5 + \alpha^5 = \alpha^3 + \alpha^2 + \alpha^5 = 0 \\ \sigma(\alpha^6) = \alpha^{12} + \alpha^4 \alpha^6 + \alpha^5 = \alpha^5 + \alpha^3 + \alpha^5 = \alpha^3 \end{array} \right\} \begin{array}{l} \Rightarrow 1^{\text{st}} \text{ error is at the position of } x^0 \\ \text{locators: } x_1 = \alpha^0, x_2 = \alpha^5 \\ \Rightarrow 2^{\text{nd}} \text{ error is at the position of } x^5 \end{array}$$

Powers of α are the locators of the error position for x .

Calculation of the error values:

$$s_k = \sum_{i=1}^t Y_i X_i^k \quad k = 0, 1, \dots, 2t-1 \Rightarrow 4 \text{ syndromes} \Rightarrow 4 \text{ equations with 4 unknowns}$$

We use locators: $x_1, x_2 \Rightarrow$ we have to solve 2 equations:

$$s_0 = Y_1 X_1^0 + Y_2 X_2^0$$

$$s_1 = Y_1 X_1^1 + Y_2 X_2^1$$

$$0 = Y_1 + Y_2 \Rightarrow Y_1 = Y_2$$

$$\alpha = Y_1 (\alpha^0)^1 + Y_1 (\alpha^5)^1$$

$$\alpha = Y_1 \alpha^0 + Y_1 \alpha^5 = Y_1 (\alpha^0 + \alpha^5) = Y_1 \alpha^4$$

$$Y_1 = \frac{\alpha}{\alpha^4} = \alpha^4$$

$$Y_1 = Y_2 = \alpha^4$$

$X_1 = \alpha^0, Y_1 = \alpha^4 \rightarrow$ at the position of x^5 the error value is α^4

$X_2 = \alpha^5, Y_2 = \alpha^4 \rightarrow$ at the position of x^0 the error value is α^4

Correction of the received message:

$$e(x) = \alpha^4 x^0 + \alpha^4 x^5$$

$$\hat{c}(x) = v(x) + e(x) =$$

$$\begin{aligned} \hat{c}(x) &= v(x) + \alpha^4 x^0 + \alpha^4 x^5 = \alpha^5 x^6 + (\alpha^2 + \alpha^4) x^5 + \alpha^4 x^4 + \alpha x^3 + \alpha^0 x^2 + \alpha^0 x + (\alpha^5 + \alpha^4) \\ &= \alpha^5 x^6 + \alpha x^5 + \alpha^4 x^4 + \alpha x^3 + \alpha^0 x^2 + \alpha^0 x + \alpha^0 \end{aligned}$$

Verification:

$$\hat{c}(\alpha^0) = \alpha^5 + \alpha^1 + \alpha^4 + \alpha^1 + \alpha^0 + \alpha^0 + \alpha^0 = 0$$

$$\hat{c}(\alpha^1) = \alpha^5 \alpha^6 + \alpha^1 \alpha^5 + \alpha^4 \alpha^4 + \alpha^1 \alpha^3 + \alpha^0 \alpha^2 + \alpha^0 \alpha^1 + \alpha^0 = 0$$

$$\hat{c}(\alpha^2) = \alpha^5 \alpha^{12} + \alpha^1 \alpha^{10} + \alpha^4 \alpha^8 + \alpha^1 \alpha^6 + \alpha^0 \alpha^4 + \alpha^0 \alpha^2 + \alpha^0 = 0$$

$$\hat{c}(\alpha^3) = \alpha^5 \alpha^{18} + \alpha^1 \alpha^{15} + \alpha^4 \alpha^{12} + \alpha^1 \alpha^9 + \alpha^0 \alpha^6 + \alpha^0 \alpha^3 + \alpha^0 = 0$$

$c(x) = \hat{c}(x) \Rightarrow$ we decoded correctly, which is confirmed by the verification.

■

Exercise 9.8:

RS code [7, 5, 3] is constructed over GF(8) which is defined by the primitive polynomial $p(x) = x^3 + x + 1$ and $p(\alpha^0) = 0$ (the first root is α^0). Decode the received message: $v(x) = \alpha^5 x^6 + \alpha^1 x^5 + \alpha^4 x^4 + \alpha^0 x^3 + \alpha^0 x^2 + \alpha^0 x + \alpha^0$.

Solution:

$$n - k = 2t \Rightarrow 2 = 2t \Rightarrow t = 1 \Rightarrow 2t \text{ roots} \Rightarrow 2t \text{ syndromes}$$

$$g(x) = (x - \alpha^0)(x - \alpha^1)$$

Syndrome calculation:

$$s_0 = v(\alpha^0) = \alpha^5 + \alpha^1 + \alpha^4 + \alpha^0 + \alpha^0 + \alpha^0 + \alpha^0 = \alpha^3$$

$$\begin{aligned} s_1 = v(\alpha^1) &= \alpha^5 \alpha^6 + \alpha^1 \alpha^5 + \alpha^4 \alpha^4 + \alpha^0 \alpha^3 + \alpha^0 \alpha^2 + \alpha^0 \alpha^1 + \alpha^0 \\ &= \alpha^4 + \alpha^6 + \alpha^1 + \alpha^3 + \alpha^2 + \alpha^1 + \alpha^0 = \alpha^6 \end{aligned}$$

Locator polynomial:

$$\sigma(x) = x^1 + \sigma_1$$

$$s_1 + \sigma_1 s_0 = 0$$

$$\alpha^6 + \sigma_1 \alpha^3 = 0 \Rightarrow \sigma_1 = \alpha^3$$

$$\sigma(x) = x + \alpha^3$$

Finding locators:

$$\left. \begin{aligned} \sigma(\alpha^0) &= \alpha^0 + \alpha^3 = \alpha^1 \\ \sigma(\alpha^1) &= \alpha^1 + \alpha^3 = \alpha^0 \\ \sigma(\alpha^2) &= \alpha^2 + \alpha^3 = \alpha^5 \\ \sigma(\alpha^3) &= \alpha^3 + \alpha^3 = 0 \\ \sigma(\alpha^4) &= \alpha^4 + \alpha^3 = \alpha^6 \\ \sigma(\alpha^5) &= \alpha^5 + \alpha^3 = \alpha^2 \end{aligned} \right\} \begin{aligned} X_1 &= \alpha^3 \\ \text{error is at the position } x^3 \end{aligned}$$

Calculation of error value:

$$s_0 = Y_1 X_1^0$$

$$\alpha^3 = Y_1 (\alpha^3)^0 = Y_1 \cdot 1$$

$$\alpha^3 = Y_1$$

$$X_1 = \alpha^3, Y_1 = \alpha^3 \rightarrow \text{error is at the position } x^3, \text{ error value is: } \alpha^3$$

$$e(x) = \alpha^3 x^3$$

Correction of the received message:

$$\begin{aligned} \hat{c}(x) &= v(x) + e(x) = v(x) + \alpha^3 x^3 \\ &= \alpha^5 x^6 + \alpha^1 x^5 + \alpha^4 x^4 + \alpha^0 x^3 + \alpha^0 x^2 + \alpha^0 x + \alpha^0 + \alpha^3 x^3 \\ &= \alpha^5 x^6 + \alpha^1 x^5 + \alpha^4 x^4 + (\alpha^0 + \alpha^3) x^3 + \alpha^0 x^2 + \alpha^0 x + \alpha^0 \\ &= \alpha^5 x^6 + \alpha^1 x^5 + \alpha^4 x^4 + \alpha^1 x^3 + \alpha^0 x^2 + \alpha^0 x + \alpha^0 \end{aligned}$$

Verification:

$$\hat{c}(\alpha^0) = \alpha^5 + \alpha^1 + \alpha^4 + \alpha^1 + \alpha^0 + \alpha^0 + \alpha^0 = 0$$

$$\hat{c}(\alpha^1) = \alpha^5 \alpha^6 + \alpha^6 + \alpha^1 + \alpha^4 + \alpha^2 + \alpha^1 + \alpha^0 = 0$$



Chapter 10: LDPC codes in modern communications standards

Low Density Parity Check (LDPC) codes belong to a class of Linear Block Codes (LBC) which specifies a sparse binary parity check matrix, usually quite large. This means, all the mathematical formulas describing the encoding and orthogonality that apply to the general LBC apply also for all LDPC codes. The practical differences arise from the extremely large size of the code generator matrix $\mathbf{G}$ (not necessarily sparse), an sparse parity check matrix $\mathbf{H}$. Standard notation assigns the two basic parameters: $[N, K]$ - the length of the codeword and the length of the data word in bits. The $\mathbf{G}$ and matrix are then defined with sizes: $\mathbf{G}^{K \times N}$ and $\mathbf{H}^{M \times N}$, where: $M = N - K$ and code rate $R = K / N$. Regardless of the algorithm, the decoding process than utilizes the basic property pf all LBC: the orthogonality of the $\mathbf{G}$ and $\mathbf{H}$ matrices:

$$\mathbf{G}^{K \times N} \cdot \mathbf{H}^{T(N \times M)} = \mathbf{0}^{K \times M} \quad (10.1)$$

The encoding process is in theory described easily by multiplying of the data word $\mathbf{i}$ by matrix $\mathbf{G}$ over GF(2), and obtaining the codeword $\mathbf{c}$. Most practical applications use a specific form of systematic $\mathbf{G}$ in order to implement systematic encoding in which the data word $\mathbf{i}$ is a subvector of $\mathbf{c}$. The rest of the codeword consists of redundant or parity bits and may be denoted as parity vector $\mathbf{p}$ of size M bits:

$$\mathbf{c}^{1 \times N} = [\mathbf{i}^{1 \times K} \quad \mathbf{p}^{1 \times M}] = \mathbf{i}^{1 \times K} \cdot \mathbf{G}^{K \times N} \quad (10.2)$$

In practical communications standards, such as WiFi-6 and WiMAX [4,5], a subclass of LDPC codes called Quasi-Cyclic codes is actually used. The QC-LDPC codes have the desired property that they allow for an efficient, near linear encoding, on top of their excellent error properties. According to Shannon, one attribute facilitating good error correcting properties of linear block code is a large codeword length N . Practically, values of N start at 576 bits for WiMAX and 648 for Wi-Fi and continue up to 2304 and 1944. Other standards, such as CCSDS or DVB-S2 utilize even much longer codewords. This then implies large code generator matrix $\mathbf{G}$, and also parity check matrix $\mathbf{H}$. Matrices for several code parameter combinations are defined. In order to be able even to fit the matrices specifications into the standards, a carefully constructed subclass of QC-LDPC allows for efficient storage of the $\mathbf{H}$ matrices in a compressed form. Furthermore, the codes defined for WiMAX and Wi-Fi also allow for a direct encoding from the $\mathbf{H}$ matrix, without actually knowing the $\mathbf{G}$ matrix. The QC-LDPC will be in the focus of this chapter.

QC-LDPC codes parity check matrix

As shown in eq. (10.3), the large and sparse binary parity check matrix $\mathbf{H}$ of a QC-LDPC code can be partitioned into a grid of square submatrices $\mathbf{P}_{i,j}$, each of size $z \times z$, while the lengths of the data word and codeword must obviously be divisible by the subblock size z .

$$\mathbf{H} = \begin{bmatrix} \mathbf{P}_{0,0} & \mathbf{P}_{0,1} & \cdots & \mathbf{P}_{0,N_b-1} \\ \mathbf{P}_{1,0} & \mathbf{P}_{1,1} & & \mathbf{P}_{1,N_b-1} \\ \vdots & & & \\ \mathbf{P}_{M_b-1,0} & \mathbf{P}_{M_b-1,1} & \cdots & \mathbf{P}_{M_b-1,N_b-1} \end{bmatrix} \quad (10.3)$$

Each of the $\mathbf{P}$ submatrices may be one of the following:

- a zero matrix
- an identity matrix
- a special case of circulant matrix – a cyclically shifted identity matrix.

This allows for a very efficient compressed representation of the $\mathbf{H}$ matrix using a single integer value to represent each of the $\mathbf{P}_{i,j}$ matrices. Instead of storing the large binary matrix $\mathbf{H}$ of size $M \times N$, where N and M values may reach to thousands, a much smaller *model matrix* $\mathbf{H}_{bm}$ of size $M_b \times N_b$ needs to be stored. Each element of $\mathbf{H}_{bm}$ then may be:

- a) a -1 , representing a zero submatrix $\mathbf{P}$
- b) a 0 , representing an identity submatrix $\mathbf{P}$
- c) a positive integer between 1 and $z - 1$ (including these values)

For the expansion of scalar $\mathbf{H}_{bm(i,j)}$ to matrix $\mathbf{P}_{i,j}$ it is necessary to understand the relation between cyclic shift operation and it's matrix equivalent first.

The vector shifts and rotations operations are very common operations present in almost any Instruction Set Architecture (ISA) and are available as assembler instructions. The most common operations are: zero filling shift left and right, denoted further by SHL and SHR, and in the context of LDPC encoding their more useful cyclic counterparts, denoted ROL and ROR. Fig. 10.1 demonstrates these operations:

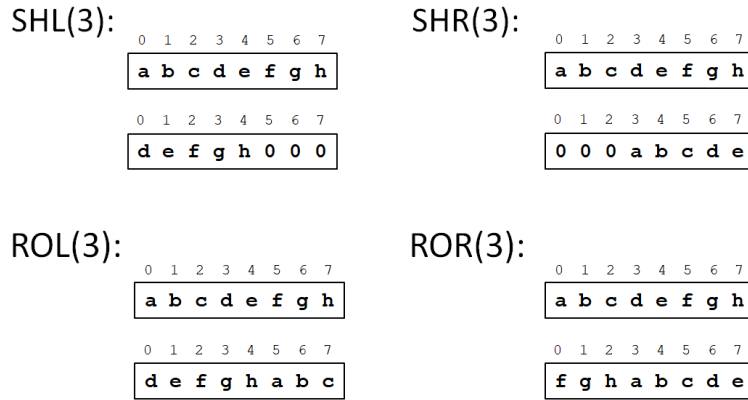


Fig.10.1: Comparison of SHL, SHR, ROL and ROR operations, shifting an 8 bit register by 3 positions.

The cyclic shift can be expressed in matrix form by multiplication of a vector by a special kind of permutation matrix. A *circulant* is a square matrix, where each column / row is defined as a circular shift of the previous column / row and the first column / row is a circular shift of the last one. For the purpose of LDPC codes description a special case of binary circulant is sufficient, where each column / row is shifted by exactly one position from the previous one, and each row / column contains exactly one nonzero value - value of 1.

Example 10.1: Cyclical shift of a length-4column vector by one position down by multiplying it with a cyclic shift matrix $\mathbf{R}$.

The matrix $\mathbf{R}$ implementing the cyclic shift is itself a circularly shifted identity matrix:

$$\mathbf{R} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The matrix-vector multiplication of the operation then implements the cyclic shift:

$$\bar{w}_C = \mathbf{R} \cdot \bar{v} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} d \\ a \\ b \\ c \end{bmatrix}$$

Given the matrix vector multiplication rules, we can reformulate the example for a column vector by transposing the multiplication:

$$(\mathbf{R} \cdot \bar{v})^T = \bar{v}^T \cdot \mathbf{R}^T$$

$$\bar{w}_R = \bar{v}^T \cdot \mathbf{R}^T \equiv \bar{v}^T \mathbf{R}^T = [a \quad b \quad c \quad d] \cdot \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} = [d \quad a \quad b \quad c]$$

clearly: $\bar{w}_R = \bar{w}_C^T$.

Now when we have some insight into the circular shifts and circulant matrices, it is time to demonstrate the model matrix $\mathbf{H}_{bm}$ expansion to a large binary $\mathbf{H}$, using all-zero, identity, and cyclic shift $\mathbf{R}$ matrices.

Exercise 10.1:

For a given submatrix of size $z = 3$, expand the given model matrix $\mathbf{H}_{bm}$ to a binary quasi-cyclic form of the matrix $\mathbf{H}$.

$$\mathbf{H}_{bm} = \begin{bmatrix} 1 & 0 & -1 & 2 & 1 & 0 & -1 & -1 \\ 2 & -1 & 0 & 1 & 0 & 0 & 0 & -1 \\ -1 & 2 & 1 & 0 & -1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 2 & 1 & -1 & -1 & 0 \end{bmatrix}$$

Solution:

Matrix $\mathbf{H}_{bm}$ is of size $M_b \times N_b = 4 \times 8$ and each element of $\mathbf{H}_{bm}$ expands to a 3×3 matrix according to its integer value:

$$\mathbf{H}_{bm(i,j)} == -1 \Rightarrow \mathbf{P}_{i,j} := \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{H}_{bm(i,j)} == 0 \Rightarrow \mathbf{P}_{i,j} := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{H}_{bm(i,j)} == 1 \Rightarrow \mathbf{P}_{i,j} := \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\mathbf{H}_{bm(i,j)} == 2 \Rightarrow \mathbf{P}_{i,j} := \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

The resulting matrix $\mathbf{H}$ will be of size $M = z \times M_b$ and $N = z \times N_b$: $M = 3 \times 4 = 12$, $N = 3 \times 8 = 24$, see Figure 10.2.

$$\mathbf{H} = \begin{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{pmatrix}$$

Fig. 10.2: Partitioning of a large matrix $\mathbf{H}$ into a grid of square submatrices.

For the convenience of the reader, a partitioning of the $\mathbf{H}$ matrix to square submatrices $\mathbf{P}_{i,j}$ is shown. This doesn't, however impose any restrictions on $\mathbf{H}$ – it is just an ordinary matrix of size 12×24 that can be further used in any matrix operations. We can even define different partitioning as we see fit.

In practical implementation of matrix partitioning and handling calculations using a grid of submatrices, care must be taken to correct mapping of indices. We may define three sets of indices:

1. The indices (i, j) of scalar elements in the large matrix ($\mathbf{H}$ or $\mathbf{G}$) that is being partitioned. Let us also recall the size of $\mathbf{H}$: $M \times N$ and let's call these indices *absolute indices*.
2. The indices (ib, jb) of the submatrices (denote $\mathbf{P}_{ib,jb}$ or $\mathbf{P}(ib,jb)$) within the matrix grid. Let us also denote the size of the grid $M_b \times N_b$ and call these indices *block indices*.

3. The indices (r, c) of scalar elements inside the submatrix $\mathbf{P}_{ib, jb}$ and let's call these indices *relative indices*.

In a general case, each block-row and each block-column can have a different size. Let us focus on a practical special case, where the large matrix is partitioned into a grid of square submatrices $\mathbf{P}$, each of size: $z \times z$. This will simplify our calculations. We can now formulate exact relationship between the indices inside a submatrix at block-position (ib, jb) with the absolute indices of elements within the original large matrix $\mathbf{H}$:

$$i = (ib - 1) \cdot z + r, j = (jb - 1) \cdot z + c \quad (10.4)$$

This simple formula implements conversion from the relative indices to absolute, while also the opposite direction is often useful:

$$r = i \bmod z, c = j \bmod z \quad (10.5)$$

Luckily, the same formula for this direction applies to MATLAB, where array indexing starts at one, just as the common theoretical indexing in any math textbook, and also for languages like C and Python, where array indexing starts at zero. However, the formula for calculating of absolute indices from relative ones must be updated for C-style array indexing:

$$i = ib \cdot z + r, j = jb \cdot z + c \quad (10.6)$$

■

Let us illustrate the index mapping using a simple example:

Exercise 10.2:

Given the matrix $\mathbf{H}$ of size 6×12 , and size of the submatrix $z = 3$, find the partitioning of the matrix $\mathbf{H}$ into a grid of square submatrices $\mathbf{P}(ib, jb)$ if it exists. Explicitly calculate the relationship between indices of all the elements of $\mathbf{P}(2,3)$ for MATLAB indexing, and the equivalent for the same submatrix, denoted as $\mathbf{P}(1,2)$ using C indexing. Let's denote this submatrix $\mathbf{P}$.

$$\mathbf{H} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Solution:

The number of row of $\mathbf{H}$ is equal 6 and is divisible by z , while the number of columns 12 is divisible by z , so the partitioning of $\mathbf{H}$ to a grid of submatrices of size 3×3 actually makes sense. The size of the grid will be $M_b \times N_b = 2 \times 4$ and we may group the bits for better understanding:

$$\mathbf{H} = \begin{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{pmatrix}$$

When using theoretical or MATLAB indexing starting at one, the $\mathbf{P} = \mathbf{P}(2,3)$ matrix is of size 3x3 and its elements may be denoted by relative indices:

$$\mathbf{P} = \begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

We may view the $\mathbf{H}$ matrix as a grid of submatrices $\mathbf{P}$:

$$\mathbf{H} = (\mathbf{P}_{ib,jb}) = \begin{pmatrix} \mathbf{P}_{1,1} & \mathbf{P}_{1,2} & \mathbf{P}_{1,3} & \mathbf{P}_{1,4} \\ \mathbf{P}_{2,1} & \mathbf{P}_{2,2} & \mathbf{P}_{2,3} & \mathbf{P}_{2,4} \end{pmatrix}$$

where indices (r, c) inside each $\mathbf{P}_{ib,jb}$ start at one, which further translates to a mapping between (r, c) (ib, jb) and (i, j) as shown in Fig. 10.3:

$$\mathbf{H} = \begin{array}{c} j \\ \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \\ i \end{array} \begin{array}{l} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}$$

$(ib, jb) = (2, 3)$ 

$$\mathbf{P}_{2,3} = \begin{array}{c} c \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ r \end{array} \begin{array}{l} 1 \\ 2 \\ 3 \end{array}$$

Fig. 10.3: Mapping between relative, block and absolute indices for MATLAB array indexing.

Explicitly evaluating (10.4), using MATLAB indexing the elements of $\mathbf{P}(2,3)$ can be calculated as:

$$\begin{aligned}
p_{11} &\equiv p_{1,1} \equiv p(1,1) = h(1 \cdot z + 1, 2 \cdot z + 1) = h(4, 7) \equiv h_{4,7} \\
p_{12} &= h(z + 1, 2z + 2) = h_{4,8} \\
p_{13} &= h(z + 1, 2z + 3) = h_{4,9} \\
p_{21} &= h(z + 2, 2z + 1) = h_{5,7} \\
p_{22} &= h(z + 2, 2z + 2) = h_{5,8} \\
p_{23} &= h(z + 2, 2z + 3) = h_{5,9} \\
p_{31} &= h(z + 3, 2z + 1) = h_{6,7} \\
p_{32} &= h(z + 3, 2z + 2) = h_{6,8} \\
p_{33} &= h(z + 3, 2z + 3) = h_{6,9}
\end{aligned}$$

When using C style indexing that starts at zero the same $\mathbf{P}$ matrix is actually $\mathbf{P}(1,2)$ and its element indices again start at zero:

$$\mathbf{P} = \begin{pmatrix} p_{00} & p_{01} & p_{02} \\ p_{10} & p_{11} & p_{12} \\ p_{20} & p_{21} & p_{22} \end{pmatrix}$$

Again we may view the $\mathbf{H}$ matrix as a grid of submatrices $\mathbf{P}$:

$$\mathbf{H} = (\mathbf{P}_{ib,jb}) = \begin{pmatrix} \mathbf{P}_{0,0} & \mathbf{P}_{0,1} & \mathbf{P}_{0,2} & \mathbf{P}_{0,3} \\ \mathbf{P}_{1,0} & \mathbf{P}_{1,1} & \mathbf{P}_{1,2} & \mathbf{P}_{1,3} \end{pmatrix}$$

And the mapping of indices changes slightly as can be seen in fig. 10.4.

$$\begin{array}{c}
\begin{array}{c} j \\ \mathbf{H} = \end{array} \begin{array}{c|cccc|cccc|cccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \hline \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} & \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} \end{array} \begin{array}{c} i \end{array} \\
\\
\begin{array}{c} (ib, jb) = (1, 2) \\ \Downarrow \\ \begin{array}{c} c \\ \mathbf{P}_{2,3} = \end{array} \begin{array}{c|cc} 0 & 1 & 2 \\ \hline \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{matrix} 0 \\ 1 \\ 2 \end{matrix} \end{array} \begin{array}{c} r \end{array}
\end{array}$$

Fig. 10.4: Mapping between relative, block and absolute indices for C array indexing.

Simply put, all the indexing must be shifted by one position: submatrix $P(2,3)$ now becomes $\mathbf{P}(1,2)$:

$$\begin{aligned}
p_{00} &\equiv p_{0,0} \equiv p(0,0) = h(1 \cdot z + 0, 2 \cdot z + 0) = h(3,6) \equiv h_{3,6} \\
p_{01} &= h(z + 0, 2z + 1) = h_{3,7} \\
p_{02} &= h(z + 0, 2z + 2) = h_{3,8} \\
p_{10} &= h(z + 1, 2z + 0) = h_{4,6} \\
p_{11} &= h(z + 1, 2z + 1) = h_{4,7} \\
p_{12} &= h(z + 1, 2z + 2) = h_{4,8} \\
p_{20} &= h(z + 2, 2z + 0) = h_{5,6} \\
p_{21} &= h(z + 2, 2z + 1) = h_{5,7} \\
p_{22} &= h(z + 2, 2z + 2) = h_{5,8}
\end{aligned}$$

CQ-LDPC codes in modern communications standards

With mathematical prerequisites out of the way, we can now look how practical LDPC codes are defined in modern IEEE communications standards. Both WiFi-6 and WiMAX define several codes with different codeword lengths and different code rates, thus specifying several model matrices $\mathbf{H}_{bm}$. Both standards define the same set of supported code rates $R = \{1/2, 2/3, 3/4, 5/6\}$. All of the model matrices share the same number of columns $N_b = 24$, but differ in the number of rows M_b which ranges from 4 for the highest-rate code $R = 5/6$, to 12 for the lowest-rate $R = 1/2$. The subblock size z is then directly related to the codeword length $z = N / N_b = N / 24$. All of the model matrices defined share a similar QC-LDPC structure, but the actual cyclic shift matrices $\mathbf{P}$ are different. The standards also specify different codeword (and subblock) sizes. Table 10.1 summarizes the supported codeword sizes for the Wi-Fi standard, while Table 10.2, and provides an overview of WiMAX [4, 5].

$N(\text{bit})$	$z(\text{bit})$	$K(\text{bit})$			
		$R = 1/2$	$R = 2/3$	$R = 3/4$	$R = 5/6$
648	27	324	432	486	540
1296	54	648	864	972	1080
1944	81	972	1296	1458	1620

Table 10.1: Supported QC-LDPC code parameters defined by the IEEE 802.11x [5] standard.

N(bit)	Z(bit)	K (Byte)			
		R = 1/2	R = 2/3	R = 3/4	R = 5/6
576	24	36	48	54	60
672	28	42	56	63	70
768	32	48	64	72	80
864	36	54	72	81	90
960	40	60	80	90	100
1056	44	66	88	99	110
1152	48	72	96	108	120
1248	52	78	104	117	130
1344	56	84	112	126	140
1440	60	90	120	135	150
1536	64	96	128	144	160
1632	68	102	136	153	170
1728	72	108	144	162	180
1824	76	114	152	171	190
1920	80	120	160	180	200
2016	84	126	168	189	210
2112	88	132	176	198	220
2208	92	138	184	207	230
2304	96	144	192	216	240

Table 10.2: Supported QC-LDPC code parameters defined by the IEEE 802.16-2017 [4] standard.

The Wi-Fi standard defines a separate matrix for each of the subblock size z (or equivalently codeword size N), giving 12 model matrices total. The WiMAX standard defines one model matrix for the longest supported codeword size $N = 2304$ and $z_0 = 96$, and provides a simple formula to rescale the values of $\mathbf{H}_{bm}$ for a smaller N . Furthermore, the WiMAX standard defines two variants: A and B for the $R=2/3$ and $R=3/4$ codes that define not only a different $\mathbf{H}_{bm}$, but also a different scaling formula. For simplicity we will focus on the B code option with scaling factor given by the following equation:

$$\mathbf{H}_{\text{bmS}}(i, j) = \left| \frac{z}{z_0} \cdot \mathbf{H}_{\text{bm}}(i, j) \right|, \mathbf{H}_{\text{bm}}(i, j) > 0 \quad (10.7)$$

WiFi-6 uses a slightly different notation for the model matrix, shown in Fig.10.5.

57	-	-	-	50	-	11	-	50	-	79	-	1	0	-	-	-	-	-	-	-
3	-	28	-	0	-	-	-	55	7	-	-	-	0	0	-	-	-	-	-	-
30	-	-	-	24	37	-	-	56	14	-	-	-	-	0	0	-	-	-	-	-
62	53	-	-	53	-	-	3	35	-	-	-	-	-	-	0	0	-	-	-	-
40	-	-	20	66	-	-	22	28	-	-	-	-	-	-	0	0	-	-	-	-
0	-	-	-	8	-	42	-	50	-	-	8	-	-	-	-	0	0	-	-	-
69	79	79	-	-	-	56	-	52	-	-	-	0	-	-	-	-	0	0	-	-
65	-	-	-	38	57	-	-	72	-	27	-	-	-	-	-	-	-	0	0	-
64	-	-	-	14	52	-	-	30	-	-	32	-	-	-	-	-	-	-	0	0
-	45	-	70	0	-	-	-	77	9	-	-	-	-	-	-	-	-	-	0	0
2	56	-	57	35	-	-	-	-	-	12	-	-	-	-	-	-	-	-	-	0
24	-	61	-	60	-	-	27	51	-	-	16	1	-	-	-	-	-	-	-	0

Fig. 10.5: $\mathbf{H}_{\text{bm}}$ for a $R = 1/2$ code with $N = 1944$ and $z = 81$ bits as defined in IEEE Std. 802.11-2020 [5].

Where each – symbol denotes a zero matrix and can easily be preplaced by a -1 (Fig. 10.6):

```

57 -1 -1 -1 50 -1 11 -1 50 -1 79 -1 1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1
3 -1 28 -1 0 -1 -1 -1 55 7 -1 -1 -1 0 0 -1 -1 -1 -1 -1 -1 -1
30 -1 -1 -1 24 37 -1 -1 56 14 -1 -1 -1 -1 0 0 -1 -1 -1 -1 -1 -1
62 53 -1 -1 53 -1 -1 3 35 -1 -1 -1 -1 -1 -1 0 0 -1 -1 -1 -1 -1
40 -1 -1 20 66 -1 -1 22 28 -1 -1 -1 -1 -1 -1 -1 0 0 -1 -1 -1 -1
0 -1 -1 -1 8 -1 42 -1 50 -1 -1 8 -1 -1 -1 -1 -1 0 0 -1 -1 -1 -1
69 79 79 -1 -1 -1 56 -1 52 -1 -1 -1 0 -1 -1 -1 -1 -1 0 0 -1 -1 -1
65 -1 -1 -1 38 57 -1 -1 72 -1 27 -1 -1 -1 -1 -1 -1 -1 0 0 -1 -1
64 -1 -1 -1 14 52 -1 -1 30 -1 -1 32 -1 -1 -1 -1 -1 -1 -1 0 0 -1
-1 45 -1 70 0 -1 -1 -1 77 9 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0 -1
2 56 -1 57 35 -1 -1 -1 -1 -1 12 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0
24 -1 61 -1 60 -1 -1 27 51 -1 -1 16 1 -1 -1 -1 -1 -1 -1 -1 -1 -1

```

Fig. 10.6: $\mathbf{H}_{bm}$ for a $R = 1/2$ code with $N = 1944$ and $z = 81$ bits as defined in IEEE Std. 802.11-2020 [5].

Let us denote this model matrix $\mathbf{H}_{bmFI}$. From IEEE Std. 802.16-2017 we may select the matrix for a rate $R = 1/2$ and observe a similar structure (Fig.10.7):

```

-1 94 73 -1 -1 -1 -1 55 83 -1 -1 7 0 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 27 -1 -1 -1 22 79 9 -1 -1 -1 12 -1 0 0 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 24 22 81 -1 33 -1 -1 -1 0 -1 -1 0 0 -1 -1 -1 -1 -1 -1
61 -1 47 -1 -1 -1 -1 65 25 -1 -1 -1 -1 -1 0 0 -1 -1 -1 -1 -1
-1 -1 39 -1 -1 -1 84 -1 -1 41 72 -1 -1 -1 -1 0 0 -1 -1 -1 -1 -1
-1 -1 -1 -1 46 40 -1 82 -1 -1 -1 79 0 -1 -1 -1 -1 0 0 -1 -1 -1 -1
-1 -1 95 53 -1 -1 -1 -1 -1 14 18 -1 -1 -1 -1 -1 0 0 -1 -1 -1 -1
-1 11 73 -1 -1 -1 2 -1 -1 47 -1 -1 -1 -1 -1 -1 -1 0 0 -1 -1 -1
12 -1 -1 -1 83 24 -1 43 -1 -1 -1 51 -1 -1 -1 -1 -1 -1 0 0 -1 -1
-1 -1 -1 -1 -1 94 -1 59 -1 -1 70 72 -1 -1 -1 -1 -1 -1 -1 0 0 -1
-1 -1 7 65 -1 -1 -1 -1 39 49 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0
43 -1 -1 -1 -1 66 -1 41 -1 -1 -1 26 7 -1 -1 -1 -1 -1 -1 -1 -1 -1

```

Fig. 7: $\mathbf{H}_{bm}$ for a $R = 1/2$ code with $N = 2304$ and $z = 96$ bits as defined in IEEE Std. 802.16-2017 [4].

For the highest rate $R = 5/6$ the WiMAX standard defines the following model matrix (Fig. 10.8):

```

1 25 55 -1 47 4 -1 91 84 8 86 52 82 33 5 0 36 20 4 77 80 0 -1 -1
-1 6 -1 36 40 47 12 79 47 -1 41 21 12 71 14 72 0 44 49 0 0 0 0 -1
51 81 83 4 67 -1 21 -1 31 24 91 61 81 9 86 78 60 88 67 15 -1 -1 0 0
68 -1 50 15 -1 36 13 10 11 20 53 90 29 92 57 30 84 92 11 66 80 -1 -1 0

```

Fig. 10.8: $\mathbf{H}_{bm}$ for a $R = 5/6$ code with $N = 2304$ and $z = 96$ bits as defined in IEEE Std. 802.16-2017 [4].

Wi-Fi defines three different matrices, one for each N , the one for the larges N is in Fig. 10.9.

```

13 48 80 66 4 74 7 30 76 52 37 60 -1 49 73 31 74 73 23 -1 1 0 -1 -1
69 63 74 56 64 77 57 65 6 16 51 -1 64 -1 68 9 48 62 54 27 -1 0 0 -1
51 15 0 80 24 25 42 54 44 71 71 9 67 35 -1 58 -1 29 -1 53 0 -1 0 0
16 29 36 41 44 56 59 37 50 24 -1 65 4 65 52 -1 4 -1 73 52 1 -1 -1 0

```

Fig. 10.9: $\mathbf{H}_{bm}$ for a $R = 5/6$ code with $N = 1944$ and $z = 81$ bits as defined in IEEE Std. 802.11-2020 [5].

As demonstrated by the examples above, the model matrices in Wi-Fi and WiMAX share further structural similarities which will be explained in later sections. All of the matrices $\mathbf{H}_{bm}$ defined above expand to a very large sparse binary matrix $\mathbf{H}$. In fact so large, that it cannot simply be printed out: even the smallest $\mathbf{H}$ defined in the WiMAX standard for $N = 567$ and $R = 5/6$ has 96 rows and 576 columns.

In order to be able at least to visualize the matrix, a special MATLAB script `displayBinaryMatrix.m` was designed that converts the matrix to a picture: each matrix element will be represented by one pixel: a one will be represented by a black pixel, while a zero by a white pixel. To better visualize the partitioning to square submatrices a grey horizontal line is drawn every z columns and a grey vertical line every z rows. The binary matrix $\mathbf{H}$ for the smallest $R = 5/6$ WiMAX code is then shown in Fig.10.10.

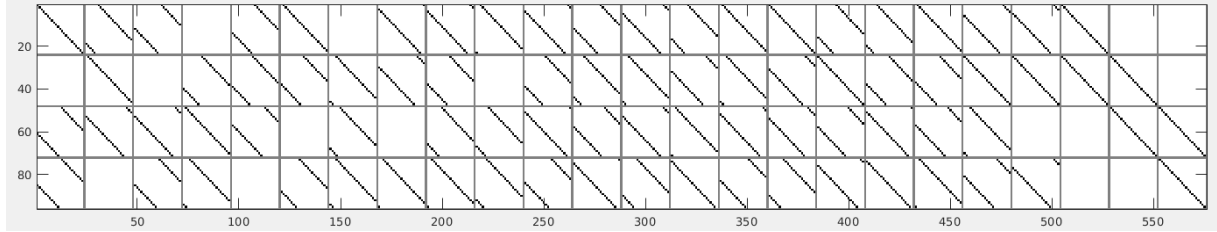


Fig. 10.10: Graphical representation of the $\mathbf{H}$ matrix for the smallest codeword size $N = 576$ of the $R = 5/6$ WiMAX-defined LDPC code. Each submatrix size is 24×24 elements and the grid size is $M_b = 4$ block-columns and $N_b = 24$ block-rows, which is exactly the size of the corresponding model matrix $\mathbf{H}_{bm}$.

Lab 10a – MATLAB exercises: uncompressing $\mathbf{H}$ matrix

Note: Solutions to exercises are placed on GitHub.

MATLAB Exercise 10.3 cyclic shift matrix - right shift:

For a vector length of 4 find the cyclic shift matrices $\mathbf{R}_2$ and $\mathbf{R}_3$, implementing a cyclic shift by 2 and 3 positions down and test their effect by multiplying the chosen column vector $\mathbf{v}$. Also find a matrix $\mathbf{R}$ implementing a cyclic shift by 4 positions.

MATLAB Exercise 10.4 cyclic shift matrix powers - right shift:

For an arbitrary length N of vector $\mathbf{v}$, let $\mathbf{R}$ implement a cyclic shift by 1 position down, multiply $\mathbf{R}$ by itself to obtain its powers $\mathbf{R}^2$, $\mathbf{R}^3$, $\mathbf{R}^4$. What operations are implemented by a multiplication by these matrices? Try in MATLAB with a value of $N = 5$.

MATLAB Exercise 10.5 cyclic shift matrix - left shift:

For a column vector length of 4 find a cyclic shift matrix $\mathbf{R}$, that implement a cyclic shift of the vector by one position up. This corresponds to cyclic shift by -1 position down and, when using a transposed form, results to cyclic shift of a row-vector to the left. Make sure by calculating all by hand and compare with simple MATLAB code.

Note: Sometimes, the precise mathematical relationship between the down shift of a column vector and its transposed equivalent of right shift of a row vector is ignored by literature. This can be useful in some situations, but generally creates more problems than it solves.

In the following two exercises, do not use any loops (`for`, `while`, etc....) in your implementation of the functions (but you may use loops for testing). This exercise is to be implemented using vector / matrix indexing and with core MATLAB commands ONLY. The use of any toolbox functions, such as `circshift()`, is explicitly prohibited (but you may use it to check your solutions).

MATLAB Exercise 10.6 universal cyclic shift of a vector or matrix:

Implement the body of the function with a given prototype:

```
function rotated = cyclicShiftRight( vector, shift )
```

that performs cyclic shift of vectors positions - the ROR operation on a row-vector stored in the first argument. The scalar parameter `shift` specifies how many positions to rotate. If negative, a ROL operation should be performed. If the `shift` is zero, the original vector should be returned. If `shift` is larger than the length of the input vector, use `mod()` function to implement an equivalent shorter shift. If a matrix is supplied as the first input (that means the variable `vector` actually contains a matrix), rotate all rows. If a column vector is supplied as the first argument, perform a rotation down, as shown in the **Example 10.1**. Also write a unit test script, that will test the implementation by shifting some vector or matrix there and back.

MATLAB Exercise 10.7 Circular shift matrix:

Implement the body of the function with a given prototype:

```
function M = cyclicShiftMatrix( Size, shift )
```

that generates and returns a square permutation matrix **M** of a given `Size` (The `Size` is a scalar value that does not shadow the built-in function `size()`. This choice of parameter name is deliberate and serves as an example of wrong design decision:). The permutation is a right cyclic shift (aka rotation) - the ROR operation. That means the right-multiplication of a row vector by this matrix performs a cyclic shift of vectors positions - the ROR operation. The scalar `shift` parameter specifies by how many positions to rotate. If negative, a matrix representing the ROL operation should be returned. If the `shift` is zero, an identity matrix should be returned. If `shift` is larger than `Size`, use `mod()` function to implement an equivalent shorter shift. The resulting matrix should be of the default MATLAB type – `double`, but it can only contain values zero and one. This exercise is may easily be implemented using the working implementation of the function `cyclicShiftRight()` from the previous exercise. Also write a unit test script, that will test the implementation by left-multiplication of some column-vector or matrix there and back. Also try right-multiplication of a row-vector and also a matrix. All unit test should be successful.

MATLAB Exercise 10.8: find cyclic shift value

Implement the body of the function with a given prototype in MATLAB:

```
function s = findCyclicShift( U, V )
```

that performs a test whether or not the vector **v** is a cyclically shifted vector **u**. If it is, the return value gives the number of positions by which we need to shift **u** to the right in order to get **v**. Using this definition, the return value will be nonnegative for any shift. Use the value -1 to indicate that vector **v** is not a cyclic shift of vector **u**. The function should work both for column and row vectors.

MATLAB Exercise 10.9: Model matrix expansion for WiFi-6

Implement the body of the function with a given prototype in MATLAB:

```
function Hs = LDPCExpandH( Hbm, z )
```

that expands the integer model matrix **H_{bm}** to a large binary sparse matrix **Hs**. Use `sparse()` to convert a regular matrix to a sparse form. This exercise is to be implemented using the working implementation of the function `cyclicShiftMatrix()`, from the Exercise 10.6. The use of `for` loops and `if` conditions is permitted only for iterating over the elements of **H_{bm}**. Use the provided function `loadWiFi6_LDPC( R, n )` to obtain the correct **H_{bm}** matrix for a given combination of *R* and *N*. Try all combinations of valid code parameters to test your expansion implementation with various model matrices. For the WiFi-6 standard, the valid values for *N* are: {648, 1296, 1944} and the valid values for

$R = \{1/2, 2/3, 3/4, 5/6\}$ try all combinations. Use the provided function `displayBinaryMatrix( H, z )` to visualize the large binary $\mathbf{H}$ matrices.

Direct encoding of QC-LDPC codes

The special structure of the $\mathbf{H}$ matrix introduced in the previous section serves an important purpose: the ability to implement the encoding process without the necessity of using (or knowing) the code generator matrix $\mathbf{G}$. This is actually a substantial advantage of the structure of LDPC codes utilized in WiMAX and WiFi-6. There are two complementary algorithms that utilize the special structure of $\mathbf{H}$ for direct encoding: the first one a matrix-oriented algorithm, suitable for MATLAB implementation, while the second is a cycle-oriented algorithm suitable for C implementation. Both take advantage of another partitioning of the $\mathbf{H}$ matrix, defined as follows [4]:

$$\mathbf{H}_{\text{bm}}^{M_b \times N_b} = \begin{bmatrix} \mathbf{H}_{\text{b1}}^{M_b \times K_b} & \mathbf{H}_{\text{b2}}^{M_b \times M_b} \end{bmatrix} = \begin{bmatrix} \mathbf{H}_{\text{b1}}^{M_b \times K_b} & \mathbf{h}_b^{M_b \times 1} & \mathbf{H}_{\text{b2}'}^{M_b \times (K_b - 1)} \end{bmatrix} \quad (10.8)$$

Where the submatrices may be precisely defined as [6]:

$$\mathbf{H}_{\text{b1}} = \mathbf{H}_{\text{bm}}[0, 1, \dots, M_b - 1; 0, 1, \dots, K_b - 1] \quad (10.9)$$

$$\mathbf{h}_b = \mathbf{H}_{\text{bm}}[0, 1, \dots, M_b - 1; K_b] \quad (10.10)$$

$$\mathbf{H}_{\text{b2}'} = \mathbf{H}_{\text{bm}}[0, 1, \dots, M_b - 1; K_b + 1, \dots, N_b - 1] \quad (10.11)$$

In other words, $\mathbf{h}_b$ is the first column of matrix $\mathbf{H}_{\text{b2}}$. We use a slightly mathematically imprecise indexing starting at zero instead of one. We do this in order to be compatible with the indexing defined in the standard [4].

As summarized in [6]: the direct encoding algorithm further requires that the column vector $\mathbf{h}_b$ has a special structure containing two paired (equal) values in the first and last position and a single non-negative value among the remaining positions. All other elements must be set to -1 indicating later expansion to zero matrices. Also required is the structure of the submatrix $\mathbf{H}_{\text{b2}'}$, where two zero diagonals within an all -1 matrix in $\mathbf{H}_{\text{b2}'}$ translate to double diagonal expanded submatrix of $\mathbf{H}$. This is clearly visible in Fig. 10. Equation (12) shows the example structure of $\mathbf{h}_b$ and $\mathbf{H}_{\text{b2}'}$ for $N_b = 24$, $K_b = 18$ for the $R = 3/4$ QC-LDPC code defined in [4]:

$$\mathbf{H}_{\text{b2}}^{6 \times 6} = \begin{bmatrix} \mathbf{h}_b^{6 \times 1} & \mathbf{H}_{\text{b2}'}^{6 \times 5} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} \mathbf{0} \\ -1 \\ \mathbf{80} \\ -1 \\ -1 \\ \mathbf{0} \end{bmatrix} & \begin{bmatrix} 0 & -1 & -1 & -1 & -1 \\ 0 & 0 & -1 & -1 & -1 \\ -1 & 0 & 0 & -1 & -1 \\ -1 & -1 & 0 & 0 & -1 \\ -1 & -1 & -1 & 0 & 0 \\ -1 & -1 & -1 & -1 & 0 \end{bmatrix} \end{bmatrix} \quad (10.12)$$

Similarly, the structure of the $\mathbf{h}_b$ column of the model matrix is also shown in bold in figures 6 – 9, where the paired values and the non-negative one are also underlined.

Let us start by describing the matrix-oriented and thus MATLAB-friendly LDPC encoding approach:

Direct encoding Method I: matrix algorithm

The $\mathbf{H}$ matrix was designed with a special structure so that it can be partitioned into a grid of (2, 3) submatrices defined as:

$$\mathbf{H}^{M \times N} = \begin{pmatrix} \mathbf{A}^{(M-z) \times K} & \mathbf{B}^{(M-z) \times z} & \mathbf{T}^{(M-z) \times (M-z)} \\ \mathbf{C}^{z \times K} & \mathbf{D}^{z \times z} & \mathbf{E}^{z \times (M-z)} \end{pmatrix} \quad (10.13)$$

An equivalent structure can be also imposed on the model matrix:

$$\mathbf{H}_{bm}^{M_b \times N_b} = \begin{pmatrix} \mathbf{A}^{(M_b-1) \times K_b} & \mathbf{B}^{(M_b-1) \times 1} & \mathbf{T}^{(M_b-1) \times (M_b-1)} \\ \mathbf{C}^{1 \times K_b} & \mathbf{D}^{1 \times 1} & \mathbf{E}^{1 \times (M_b-1)} \end{pmatrix} \quad (10.14)$$

For the $\mathbf{H}_{bm}$ of a code rate $R = 5/6$ of the WiMAX standard, this structure is visualized in Fig. 10.11, along with the equivalent expansion.

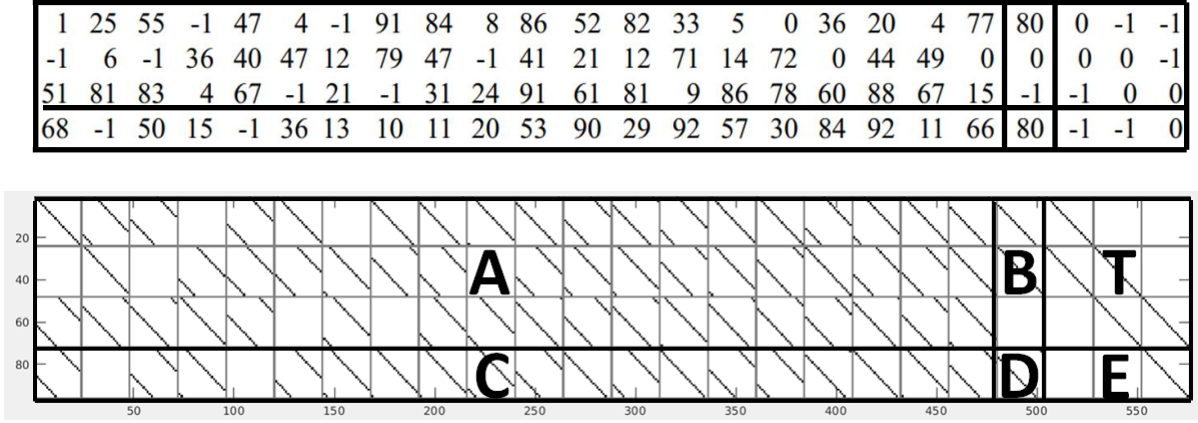


Fig. 10.11: Partitioning of $\mathbf{H}_{bm}$ (above) and equivalent partitioning of $\mathbf{H}$ (below).

This partitioning to submatrices is then utilized for direct encoding in the following way:

Let $\mathbf{i}$ denote the information vector of size K bits to be systematically encoded. Let the unknown parity vector $\mathbf{p}$ be divided into two subvectors $\mathbf{p}_1$ and $\mathbf{p}_2$ of lengths z and $M - z$. The systematic codeword is then divided into three subvectors:

$$\mathbf{c}^{1 \times N} = [\mathbf{i}^{1 \times K} \quad \mathbf{p}_1^{1 \times z} \quad \mathbf{p}_2^{1 \times (M-z)}] \quad (10.15)$$

Since LDPC is a linear block code, for every codeword, every codeword must be orthogonal to the $\mathbf{H}$ matrix, so that the equation:

$$\mathbf{H}^{M \times N} \cdot \mathbf{c}^{T(N \times 1)} = \mathbf{0}^{M \times 1} \quad (10.16)$$

must hold, where $\mathbf{0}$ is an all-zero vector M positions long and T indicates transpose operation. This can be further represented as a block matrix multiply:

$$\mathbf{H}^{M \times N} \cdot \mathbf{c}^{T(N \times 1)} = \mathbf{0}^{M \times 1} = \begin{pmatrix} \mathbf{A}^{(M-z) \times K} & \mathbf{B}^{(M-z) \times z} & \mathbf{T}^{(M-z) \times (M-z)} \\ \mathbf{C}^{z \times K} & \mathbf{D}^{z \times z} & \mathbf{E}^{z \times (M-z)} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{i}^{K \times 1} \\ \mathbf{p}_1^{z \times 1} \\ \mathbf{p}_2^{(M-z) \times 1} \end{pmatrix} \quad (10.17)$$

For the following text, let us denote the $\mathbf{i}$, $\mathbf{p}_1$ and $\mathbf{p}_2$ vectors as column vectors. Since we are dealing with binary vectors and matrices, all operations in the following are Boolean operations. The matrix equation can then be rewritten as two equations:

$$\begin{aligned}\mathbf{A}\mathbf{i} + \mathbf{B}\mathbf{p}_1 + \mathbf{T}\mathbf{p}_2 &= \mathbf{0} \\ \mathbf{C}\mathbf{i} + \mathbf{D}\mathbf{p}_1 + \mathbf{E}\mathbf{p}_2 &= \mathbf{0}\end{aligned}\tag{10.18}$$

Which can further be transformed over GF(2) where $-$ is the same operation as $+$, sometimes called a bitwise XOR operation :

$$\begin{aligned}\mathbf{T}^{-1}\mathbf{A}\mathbf{i} + \mathbf{T}^{-1}\mathbf{B}\mathbf{p}_1 &= \mathbf{p}_2 \\ \mathbf{C}\mathbf{i} + \mathbf{D}\mathbf{p}_1 &= \mathbf{E}\mathbf{p}_2\end{aligned}\tag{10.19}$$

Further multiplying the first equation in (10.19) by $\mathbf{E}$ and equating with the second we get:

$$\begin{aligned}\mathbf{E}\mathbf{p}_2 &= \mathbf{E}\mathbf{T}^{-1}\mathbf{A}\mathbf{i} + \mathbf{E}\mathbf{T}^{-1}\mathbf{B}\mathbf{p}_1 \\ \mathbf{E}\mathbf{p}_2 &= \mathbf{C}\mathbf{i} + \mathbf{D}\mathbf{p}_1\end{aligned}\tag{10.20}$$

Now we subtract the equations over GF(2) to get:

$$(\mathbf{E}\mathbf{T}^{-1}\mathbf{A} + \mathbf{C})\mathbf{i} + (\mathbf{E}\mathbf{T}^{-1}\mathbf{B} + \mathbf{D})\mathbf{p}_1 = \mathbf{0}\tag{10.21}$$

The structure of the $\mathbf{H}$ matrix is deliberately designed so that the expression in the second term is an identity matrix (We will show this is true shortly):

$$(\mathbf{E}\mathbf{T}^{-1}\mathbf{B} + \mathbf{D}) = \mathbf{I}\tag{10.22}$$

From this we derive the first parity subvector $\mathbf{p}_1$:

$$\mathbf{p}_1 = (\mathbf{E}\mathbf{T}^{-1}\mathbf{A} + \mathbf{C})\mathbf{i}\tag{10.23}$$

and finally the second parity subvector $\mathbf{p}_2$:

$$\mathbf{p}_2 = \mathbf{T}^{-1}\mathbf{A}\mathbf{i} + \mathbf{T}^{-1}\mathbf{B}\mathbf{p}_1 = \mathbf{T}^{-1}(\mathbf{A}\mathbf{i} + \mathbf{B}\mathbf{p}_1)\tag{10.24}$$

Once the two parity subvectors are calculated, the systematic codeword $\mathbf{c}$ is put together simply by vector concatenation as shown in (10.15). In order to check the correctness of the implementation, it is useful to check orthogonality of the codeword with the $\mathbf{H}$ matrix – equation (10.16) must hold.

Direct encoding Method II: Low-level algorithm

The special structure of the $\mathbf{H}$ matrix was designed to allow for the direct encoding algorithm where the encoding of a systematic code uses the $\mathbf{H}$ matrix instead of the code generator matrix $\mathbf{G}$. For a practical C encoder implementation the model matrix $\mathbf{H}_{bm}$ is actually used, so the large sparse binary $\mathbf{H}$ matrix, or even permutation submatrices $\mathbf{P}_{i,j}$ are never actually stored anywhere in memory. Since the multiplication of some subvectors of $\mathbf{i}$ of length Z by matrix $\mathbf{P}_{i,j}$ is identical in result to cyclic shift, the actual operation to implement as a building block of the encoder is the cyclic shift of subvectors of size Z bits by $\mathbf{H}_{bm}(i, j)$ positions. The direct encoder algorithm originally introduced in [38] and described in sufficient detail also in [4] can be summarized as follows:

The information block $\mathbf{i}$ (also known as systematic vector $\mathbf{s}$) is divided into K_b blocks of Z bits (we will use z and Z interchangeably). The grouped $\mathbf{i}$ can be denoted as a matrix $\mathbf{u}$, where each element u_i is a column vector:

$$\mathbf{u}^{Z \times K_b} = [u_0 \quad u_1 \quad \dots \quad u_{K_b-1}] \quad (10.25)$$

$$u_i = [s_{iZ} \quad s_{iZ+1} \quad \dots \quad s_{(i+1)Z-1}]^T \quad (10.26)$$

The parity sequence $\mathbf{p}$ is calculated in blocks of Z bits directly using the matrix $\mathbf{H}_{bm}$. Again, the parity sequence $\mathbf{p}$ can be expressed as a matrix in terms of blocks, where each element v_i is a column vector Z bits long:

$$\mathbf{p}^{Z \times M_b} = [v_0 \quad v_1 \quad \dots \quad v_{M_b-1}] \quad (10.27)$$

$$v_i = [p_{iZ} \quad p_{iZ+1} \quad \dots \quad p_{(i+1)Z-1}]^T \quad (10.28)$$

slightly nonstandard notation for vectors u_i and v_i is used here in order to preserve notation in [3]. The encoding is divided into initialization and recursion. First the initialization is defined as follows:

$$v_0 = \mathbf{P}^{-1}_{\mathbf{H}_{bm}(y, K_b)} \cdot \left(\sum_{i=0}^{M_b-1} \sum_{j=0}^{K_b-1} \mathbf{P}_{\mathbf{H}_{bm}(i, j)} u_j \right) \quad (10.29)$$

Where $\mathbf{H}_{bm}(i, j)$ is an element of the (potentially scaled) model matrix $\mathbf{H}_{bm}$, $\mathbf{P}_{\mathbf{H}_{bm}(i, j)}$ represents the circulant permutation matrix, and multiplication by $\mathbf{P}_{\mathbf{H}_{bm}(i, j)}$ implements the cyclic shift operation on subvector u_j . The element $\mathbf{H}_{bm}(y, K_b)$ is the only nonnegative element of the subvector $\mathbf{h}_b$ outside of the paired values, as shown in bold in equation (10.12) with $\mathbf{H}_{bm}(y, K_b) = \mathbf{h}_b(2) = 80$. Note that the value of y is different for each $\mathbf{H}_{bm}$ and must be found on-the-fly by the encoder (or stored). The inverse permutation $\mathbf{P}^{-1}$ is just a cyclic shift in the opposite direction. All addition/sum operations are performed over GF(2).

After initialization and successful calculation of v_0 , the encoder proceeds with iteration/recursion, finding the remaining parity subblocks v_i :

$$v_1 = \mathbf{P}_{\mathbf{H}_{bm}(0, K_b)} v_0 + \sum_{j=0}^{K_b-1} \mathbf{P}_{\mathbf{H}_{bm}(0, j)} u_j, i = 0 \quad (10.30)$$

$$v_{i+1} = v_i + \mathbf{P}_{\mathbf{H}_{bm}(i, K_b)} v_0 + \sum_{j=0}^{K_b-1} \mathbf{P}_{\mathbf{H}_{bm}(i, j)} u_j, i > 0 \quad (10.31)$$

Since the sums terms in (12):

$$\mathbf{S}_i \equiv \sum_{j=0}^{K_b-1} \mathbf{P}_{\mathbf{H}_{bm}(i, j)} u_j, i = 0 \dots M_b - 1 \quad (10.32)$$

are calculated during the initialization and reused during the iteration, it is advantageous to store their values for reuse. Since the elements of $\mathbf{H}_{bm}$ equal to -1 expand to all zero $\mathbf{P}$ submatrix, a multiplication by such a $\mathbf{P}_{\mathbf{H}_{bm}}$ results to a zero term in sums and may be safely skipped in the implementation.

Example 10.2:

Let us continue with our Exercise 10.1: the large $\mathbf{H}$ matrix shown in Fig. 10.2 has 12 rows and 24 columns and can be partitioned into a grid of 3x4 square submatrices of size $z = 3$. Furthermore, it can also be partitioned into a grid of 2 x 3 matrices A, B, T, C, D, E as defined by (10.13). The resulting partitioning is show in Fig. 10.12.

$$\mathbf{H} = \left(\begin{array}{ccc|ccc|ccc} \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \right)$$

Fig. 10.12: Partitioning of the $\mathbf{H}$ matrix to submatrices A, B, T, C, D, E.

The assignment here is to proof, that the identity in (10.22) actually holds.

Solution:

Note, that the special double-diagonal structure of $\mathbf{T}$ determines also a special structure of its GF(2) inverse. This is easily calculated; even by hand (we leave this step for the next example):

$$\mathbf{T}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{I}_z & \mathbf{0} & \mathbf{0} \\ \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} \\ \mathbf{I}_z & \mathbf{I}_z & \mathbf{I}_z \end{pmatrix}$$

Furthermore, the multiplication of another special-structure matrix $\mathbf{E}$ by $\mathbf{T}^{-1}$ has then the effect of repeating its last submatrix of M_b-1 times:

$$\begin{aligned}
\mathbf{E}\mathbf{T}^{-1} &= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \mathbf{T}^{-1} = (\mathbf{E}_{11} \quad \mathbf{E}_{12} \quad \mathbf{E}_{13}) \begin{pmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{I} & \mathbf{I} & \mathbf{0} \\ \mathbf{I} & \mathbf{I} & \mathbf{I} \end{pmatrix} = \\
&= (\mathbf{E}_{11} + \mathbf{E}_{12} + \mathbf{E}_{13} \quad \mathbf{E}_{12} + \mathbf{E}_{13} \quad \mathbf{E}_{13}) = (\mathbf{0} + \mathbf{0} + \mathbf{E}_{13} \quad \mathbf{0} + \mathbf{E}_{13} \quad \mathbf{E}_{13}) = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}
\end{aligned}$$

We can double-check the definitions of all standard-define $\mathbf{H}_{bm}$ matrices in Figures 10.5. - 10.9 to see that this last submatrix of $\mathbf{E}$ will by definition always be an identity matrix, so that the term $\mathbf{E}\mathbf{T}^{-1}$ will always be a one block-row matrix consisting of all identity submatrices.

We can now continue further by multiplying the $\mathbf{E}\mathbf{T}^{-1}$ term by the $\mathbf{B}$ submatrix, as defined in (10.22):

$$\mathbf{E}\mathbf{T}^{-1}\mathbf{B} = (\mathbf{I}_z \quad \mathbf{I}_z \quad \mathbf{I}_z) \begin{pmatrix} \mathbf{P}_z \\ \mathbf{I}_z \\ \mathbf{0}_z \end{pmatrix} = \mathbf{P} + \mathbf{I} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

From the definition of the block column $\mathbf{h}_b$ shown in (10.12) we know, that it must always consist of two paired values at the first and last position, exactly one zero position, and all other values of -1 . This translates to the equality of the permutation matrix $\mathbf{P}$ above with the $\mathbf{D}$ matrix defined by the partitioning in Fig 12. Since $\mathbf{P} = \mathbf{D}$ we may confirm that the expression (10.22) in the direct encoding algorithm definition truly evaluates to an identity matrix over GF(2):

$$\mathbf{E}\mathbf{T}^{-1}\mathbf{B} + \mathbf{D} = \mathbf{P} + \mathbf{I} + \mathbf{D} = \mathbf{D} + \mathbf{I} + \mathbf{D} = \mathbf{I}$$

■

Example 10.3:

GF(2) inversion of a special-structure matrix $\mathbf{T}$: We can see from the previous example, that a double-diagonal matrix $\mathbf{T}$ defined in Fig. 10.12 also determines a special structure of its GF(2) inversion. Let us show, that this is also true for any size of $\mathbf{T}$, given its double-diagonal structure.

Solution:

The $\mathbf{T}$ matrix of any size may be expressed in terms of a grid of square submatrices of size $z \times z$ in the following way:

$$\mathbf{T}^{Mb-1 \times Mb-1} = \begin{pmatrix} \mathbf{I}_z & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \dots & \mathbf{0} \\ \vdots & \vdots & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z \end{pmatrix}$$

Regardless of the submatrix size z , we can perform the classic elimination method using row-equivalent transformations by hand, performed on matrix-rows and matrix-columns. We start by adding the first

matrix-row to the second, and we simply continue by adding the previous matrix-row to the next, until we reach the last matrix row:

$$\begin{aligned}
& \left(\begin{array}{ccccc|ccccc} \mathbf{I}_z & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{I}_z & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{I}_z & \mathbf{0} & & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I}_z & & \vdots \\ \vdots & \vdots & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & \vdots & & & \mathbf{I}_z & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{I}_z \end{array} \right) R_2 := R_2 + R_1 / \approx \\
& \left(\begin{array}{ccccc|ccccc} \mathbf{I}_z & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{I}_z & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_z & \mathbf{0} & \dots & \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I}_z & & \vdots \\ \vdots & \vdots & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & & & & \mathbf{I}_z & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{I}_z \end{array} \right) R_3 := R_3 + R_2 / \approx \\
& \left(\begin{array}{ccccc|ccccc} \mathbf{I}_z & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{I}_z & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_z & \mathbf{0} & \dots & \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I}_z & \dots & \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \mathbf{I}_z & & \vdots \\ \vdots & \vdots & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & & & & \mathbf{I}_z & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{I}_z \end{array} \right) R_4 := R_4 + R_3 / \approx \dots \\
& \left(\begin{array}{ccccc|ccccc} \mathbf{I}_z & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{I}_z & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_z & \mathbf{0} & \dots & \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & & \mathbf{0} \\ \vdots & \mathbf{0} & \mathbf{I}_z & \dots & \mathbf{0} & \vdots & \mathbf{I}_z & \mathbf{I}_z & & \vdots \\ \mathbf{0} & \dots & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & \mathbf{I}_z & \dots & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} \\ \mathbf{0} & \dots & \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{I}_z \end{array} \right) R_{Mb-1} := R_{Mb-1} + R_{Mb-2} / \approx \\
& \left(\begin{array}{ccccc|ccccc} \mathbf{I}_z & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{I}_z & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_z & \mathbf{0} & \dots & \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} & & \mathbf{0} \\ \vdots & \mathbf{0} & \mathbf{I}_z & \dots & \mathbf{0} & \vdots & \mathbf{I}_z & \mathbf{I}_z & & \vdots \\ \mathbf{0} & \dots & \mathbf{0} & \mathbf{I}_z & \mathbf{0} & \mathbf{I}_z & \dots & \mathbf{I}_z & \mathbf{I}_z & \mathbf{0} \\ \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{I}_z & \mathbf{I}_z & \dots & \mathbf{I}_z & \mathbf{I}_z & \mathbf{I}_z \end{array} \right)
\end{aligned}$$

From the last form of the inverse matrix on the right we can see that $\mathbf{T}^{-1}$ has a lower-triangular block-matrix structure where each block-element at and below the main diagonal is an identity matrix of size $z \times z$. For the special case of a double-diagonal structure of $\mathbf{T}$ the Jordan elimination method is greatly simplified: it consists only of repeated addition of previous to current block-row (over GF(2)) and iterating over all block rows. This can be used in an optimized implementation.

Lab 10b MATLAB exercises: QC-LDPC Encoding

Note: The MATLAB solutions for the exercises of this chapter are divided into several source files. Open `lab10b.m` for the following exercises solutions:

MATLAB Exercise 10.10: Paired submatrices extraction

Implement the body of the function with prototype:

```
function P = submatrix( H, z, ib, jb )
```

that extracts the square submatrix $\mathbf{P}(ib, jb)$ of size $z \times z$ from the large matrix $\mathbf{H}$. The function should throw an error if some of the indices is out of bounds or if the number of rows or columns of $\mathbf{H}$ is not divisible by z . Use the MATLAB built-in `size()` function to determine the size of $\mathbf{H}$. Write a unit test using the implementations of MATLAB Exercise 10.9: use `LDPCExpand()` to obtain a large $\mathbf{H}$ matrix with block-matrix structure and then extract the paired submatrices corresponding to the first and last element of the $\mathbf{h}_b$ column of the model matrix as defined in (10.12). These two matrices should be equal.

MATLAB Exercise 10.11: Circulant structure of $\mathbf{H}$ submatrices

Implement the body of the function with prototype:

```
function cir = isCirculant( P )
```

that checks, whether or not the $\mathbf{P}$ matrix is a special case of a circulant in the sense that each row is a cyclic shift by one position of the previous row and the first row is a cyclic shift by one position of the last row. The result should be a scalar of type `logical` (Boolean), true if matrix is circulant. The function should return true regardless of the cyclic shift direction (left or right). Write a unit test based on the `u2nit` test of the previous exercise: Test that each of the submatrices aligned to the grid on $M_b \times N_b$ submatrices of size $z \times z$ in the expanded block-matrix $\mathbf{H}$ is in fact a circulant. Note that also an all-zero and identity matrix is in fact a circulant matrix.

MATLAB Exercise 10.12: Reconstructing $\mathbf{H}_{bm}$ from $\mathbf{H}$

Implement the body of the function with prototype:

```
function [ ok, Hbm ] = isW6_QCLDPC( H, z )
```

that will test whether the matrix $\mathbf{H}$ is the QC-LDPC structure: that it consists of a grid of square submatrices of size $z \times z$, each of which may only be a zero matrix, an identity matrix, or a circulant. Use the built-in function `isequal()` to compare submatrices along with the function `isCirculant()`. The function should throw an error if the size of the $\mathbf{H}$ is not a multiple of z . As a second output parameter, the function should return a reconstructed $\mathbf{H}_{bm}$: for each all-zero submatrix it should return a -1, a zero output will be given for each identity submatrix, and the value of the circular shift from an identity matrix should be returned for the rest of the submatrices – the circulants. The Even if some submatrices are not circulants, the function should attempt to reconstruct as much of the structure as possible – indicate the submatrices that cannot be classified into any of the three categories above by outputting a NAN at their positions.

MATLAB Exercise 10.13: Partition the $\mathbf{H}$ matrix into submatrices

Implement the body of the function with prototype:

```
function [ A, B, T, C, D, E ] = partitionH( H, z )
```

where $\mathbf{H}$ is a large binary matrix with structure defined by eq. (10.13). The function should return an error if any of the following conditions is met:

- The $\mathbf{H}$ matrix doesn't have the basic desired QC-LDPC structure.
- $\mathbf{T}$ submatrix doesn't have the required double diagonal structure.

Use the provided `displayBinaryMatrix()` function to visualize the structure of submatrices of $\mathbf{H}$.

MATLAB Exercise 10.14: special GF(2) matrix inversion

Implement the body of the function with prototype:

```
function MI = invGF2dd( M, z )
```

that inverts a special-structure double-diagonal square binary matrix $\mathbf{M}$, divisible into a grid of square matrices of size $z \times z$. Assume the matrix $\mathbf{M}$ is binary stored as double: that means it contains values of the default MATLAB double precision floating point type, but limited only to values 0 and 1. Do not use any functions or special data types from the MATLAB Galois toolkit. The function should throw an error if the input $\mathbf{M}$ is not a matrix of the special required structure. Use Example 10.3 as a blueprint for your implementation – implement a simplified Jordan method for block-matrices with special double diagonal structure. Use the construct `mod( X + Y, 2 )` for a simple implementation of GF(2) addition. Also implement helper function `R = getBlockRow( M, z, ib )` and that extracts a single block-row $\mathbf{R}$ of z rows from a block-matrix $\mathbf{M}$, given its block-index ib and `MA = addBlockRow( M, R, z, ib )` that adds the block-row $\mathbf{R}$ to the block-matrix $\mathbf{M}$ at its block-row index r and returns the resulting matrix $\mathbf{M}_A$ with block-row added. These helper functions must be GF(2) agnostic, meaning that they operate on any values, not just GF(2).

The resulting inverse matrix $\mathbf{M}^{-1}$ must be binary, stored as double and the MATLAB correctness check: `mod( M * MI, 2 )` must hold true.

Use the combination of MATLAB built-in functions `mod( inv( M ), 2 )` to compare your implementation and built-in matrix inversion results. Will the results be the same? If yes, can we use this trick in general? Write a unit test, that loads the WiFi-6 LDPC codes defined for all parameters R , n combinations and test your GF(2) matrix inversion for all of them.

MATLAB Exercise 10.15: QC-LDPC direct encoding – matrix encoder

Implement the body of the function with prototype

```
function C = encodeLDPCmatrix( I, H, Z )
```

that encodes a binary vector $\mathbf{i}$ of length k bits to a binary vector $\mathbf{c}$ of length n bits using the procedure described by equations (10.13), (10.23), (10.24). First check, that the supplied $\mathbf{H}$ matrix has the proper structure and throw an error if it doesn't (use your `isW6_QCLDPC()` function). Also check the validity of (10.22) and throw an error if invalid. Also check the orthogonality of the codeword – eq. (10.16) over GF(2) and throw an error if codeword is not orthogonal to the $\mathbf{H}$ matrix. The third input parameter is optional. All output parameters must be a binary vectors stored as double.

Test your implementation first using a small $\mathbf{H}$ matrix from the Exercise10.1, and if everything works test it using actual WiFi-6 LDPC code parameters R , n . Use the orthogonality between each codeword $\mathbf{c}$ and parity check matrix $\mathbf{H}$ for testing the correctness of your encoder implementation.

MATLAB Exercise 10.16: Non-matrix QC-LDPC encoder

Section Direct Encoding Method II described in the Wi-MAX standard [4] is valid also for the LDPC codes defined for Wi-Fi and provides a more practical encoder algorithm, closer to actual low-level implementation. Such non-matrix algorithm is much closer to a low-level C/C++ code that could be found in device drivers. Equations (10.25) to (10.32) are indeed defined using matrix multiplications, but these deal exclusively with small $\mathbf{P}$ matrices, each of which can only be an all zero matrix, identity matrix, or a cyclic-shift matrix. Multiplication of a vector by a zero or identity matrix in a low-level language without vector constructs is trivial: just returning zeros or the original input vector. Low-level implementation of a cyclic-shift matrix-vector multiplication translated to simple outputting of a cyclic-shift of the vector. This can be implemented by a simple cyclic-shift function (such as the already

implemented `cyclicShiftRight()` that would further translate directly to CPU ROR and ROL instructions (or equivalent). In this exercise we will simulate this low-level implementation in MATLAB by implementing this algorithm (eq. 10.25 to 10.32) first by explicitly using matrix-vector multiplications of $\mathbf{P}_{ij}\mathbf{u}_j$, then by replacing these with cyclic shifts of $\mathbf{u}_j$. Again, we will use the construct `mod( X + Y, 2 )` for a simple implementation of GF(2) addition without the necessity of the Galois toolkit:

Implement the body of the function with prototype:

```
function v = encodeG1M1aM( Hbm, u, z )
```

that implements the algorithm given by equations (10.25) to (10.32) and encodes a binary input vector directly using the model matrix $\mathbf{H}_{bm}$. This implementation should be a straightforward expression of these equations in MATLAB.

Also implement the body of the function with prototype:

```
function v = encodeG1M1aR( Hbm, u, z )
```

that implements the algorithm described by equations (10.25) to (10.32) and encodes a binary input vector directly using the model matrix $\mathbf{H}_{bm}$. This implementation should replace any matrix multiplication by a call to `cyclicShiftRight()` as described above.

Both implementations should produce exactly the same output, which should also be equal to the output `encodeLDPCmatrix()` function. Write a unit test to confirm for all WiFi-6 parameters R, n combinations.

Decoding LDPC codes

For the decoding of LDPC codes, several algorithms have been designed over the years. The computationally simpler are the hard-decision algorithms that focus on bit flipping. Since these algorithms do not really achieve the best performance in terms of decoding error ratio, more advanced, soft-decoding algorithms were designed. All of these are defined using a special graph representation of the LDPC code parity check matrix $\mathbf{H}$ – the *Tanner graph* described in more detail below. The soft-decision decoding algorithms are all based on the maximum posterior probability criterion, expressed using a chosen metric. Intuitively the most straightforward metric would be directly the posterior probability. A formulation of the decoder using this metric does exist [17] and is in fact the original algorithm. However, the necessity of high-throughput implementation of the decoder requires to reduce its complexity. A logarithm of the probability turns out to be a good practical metric that facilitates decoding algorithm simplification. Before we get into details, the Tanner graph should be explained.

Tanner graph

The Tanner graph is a bipartite graph: a graph that contains two types of nodes: the first one – variable node (denoted as v) represents a position in a codeword, while the second one – a check node (denoted as ch), and represents a parity equation as defined by matrix $\mathbf{H}$. The nonzero elements of $\mathbf{H}$ then define the connection between the corresponding variable and check node. If we index the variable nodes using n , and check nodes using m , an edge will be drawn between each variable and check node where $H(m, n) = 1$. There are no edges between two variable nodes or two check nodes. In other words, a Tanner graph can be understood as a handy visual representation of any parity check matrix.

Example 10.4:

Let the $\mathbf{H}$ matrix is a parity check matrix of a systematic Hamming [7, 4, 3] code.

$$\mathbf{H} = \begin{pmatrix} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{matrix} ch_1 \\ ch_2 \\ ch_3 \end{matrix}$$

Let us denote the positions of a codeword $\mathbf{c} = (c_1, c_2, \dots, c_N)$ where $N = 7$ and the three parity equations ($M = N - K = 3$) defined by the $\mathbf{H}$ matrix ch_1, ch_2, ch_3 . We can rewrite the $\mathbf{H}$ matrix as a system of GF(2) equations:

$$\begin{aligned} c_2 + c_3 + c_4 + c_5 &= 0 \\ c_1 + c_3 + c_4 + c_6 &= 0 \\ c_1 + c_2 + c_4 + c_7 &= 0 \end{aligned} \tag{10.33}$$

Each variable v_n node will correspond to a given codeword position c_n . The difference between c_n and v_n lies in the fact that c_n is a binary value and is the result of the decoding process; while the v_n takes on a real-value of a selected metric and is used during the iterative decoding process (It may be even a vector value for some algorithm formulations). We can draw a Tanner graph (Fig. 10.12):

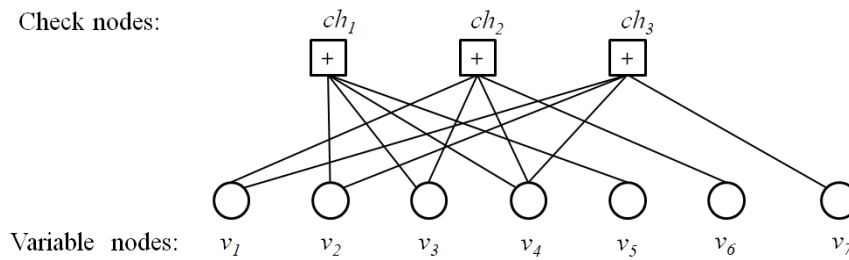


Fig. 10.12: A Tanner graph for a [7, 4, 3] Hamming code.

Since all of the decoding algorithms conceptually use the Tanner graph as basis for their operation, it is necessary to define other important terms that will be used further. For every variable node index $n \in \{1, 2, \dots, N\}$ we may define a set of incident check nodes $M(n)$. Simultaneously, for every check node index $m \in \{1, 2, \dots, M\}$ we may define a set of incident variable nodes indices $N(m)$. We call these sets neighborhoods of n and m and we sometimes omit the word index, so when we speak of variable node n , we actually mean the variable node with index n . Furthermore, for a given $N(m)$ and n , we may define a set of variable nodes incident with check node m , except the specific variable node n , we denote by a set difference operator: $N(m) \setminus n$. Similarly, for a given $M(n)$ and m , we may define a set of check nodes incident with variable node n , except for a specific check node m , and denote $M(n) \setminus m$. There are N neighborhoods - sets of check node indices $M(n)$ - one for each variable node n . Together, these completely define the Tanner graph and therefore also the matrix $\mathbf{H}$. Similarly, there are M sets of neighborhoods $N(m)$ - one for each check node m . ■

Example 10.5:

For a Hamming [7, 4, 3] code from the previous example, let us find all the variable and check nodes neighborhoods: set of indices of incident nodes. This can be done directly from the **H** matrix or from the Tanner graph:

For all the variable nodes (or column indices) $n \in \{1, 2, \dots, 7\}$ we note the row indices of ones in column n :

$$\begin{aligned} M(1) &= \{2, 3\}, M(2) = \{1, 3\}, M(3) = \{1, 2\}, M(4) = \{1, 2, 3\}, \\ M(5) &= \{1\}, M(6) = \{2\}, M(7) = \{3\}. \end{aligned}$$

Similarly, for all the check nodes $m \in \{1, 2, 3\}$ we note the row indices of ones in column m :

$$\begin{aligned} N(1) &= \{2, 3, 4, 5\}, \\ N(2) &= \{1, 3, 4, 6\}, \\ N(3) &= \{1, 2, 4, 7\}. \end{aligned}$$

■

LLR – the Log-Likelihood ratio

The LLR metric is a basis for a computationally feasible LDPC decoding. For a binary random variable X with alternative probability distribution, we can define LLR simply:

$$LLR(x) = \lambda(x) = \ln \left(\frac{p(x=1)}{p(x=0)} \right) \quad (10.34)$$

or alternatively with reversed values:

$$LLR(x) = \lambda(x) = \ln \left(\frac{p(x=0)}{p(x=1)} \right) \quad (10.35)$$

Both of these definitions are equally useful, and since both can be found in different sources, we present them here.

Using the definition provided by (10.34), we can also express the probabilities of X taking on the values 0 and 1 as:

$$p(x=1) = \frac{e^{\lambda(x)}}{1 + e^{\lambda(x)}} \quad (10.36)$$

and:

$$p(x=0) = 1 - p(x=1) = \frac{1}{1 + e^{\lambda(x)}} \quad (10.37)$$

The sign of $\lambda(x)$ determines the hard decision (0 or 1) regarding the variable X while its magnitude determines its reliability. We can see how the probability $p(x)$ translates to the LLR metric $\lambda(x)$ defined by (10.34) on Fig. 10.13.

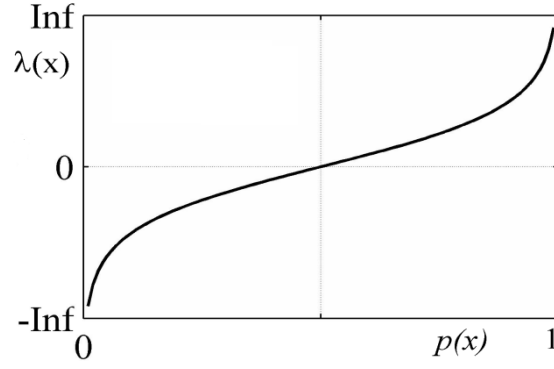


Fig. 10.13: The graph of the LLR metric $\lambda(x)$ as a function of $p(x)$ for the definition (10.34).

For an AWGN channel and BPSK modulation modelled by (10.39), this LLR metric (defined by (10.35)) is easily calculated by inserting the channel probability distribution from Chapter 8:

$$LLR(z) = \ln \left(\frac{p(z/0)}{p(z/1)} \right) = \ln \left(\frac{\frac{1}{\sigma_0 \sqrt{2\pi}} \exp \left[-\frac{1}{2} \left(\frac{z - \sqrt{E}}{\sigma_0} \right)^2 \right]}{\frac{1}{\sigma_0 \sqrt{2\pi}} \exp \left[-\frac{1}{2} \left(\frac{z + \sqrt{E}}{\sigma_0} \right)^2 \right]} \right) = \frac{2\sqrt{E}}{\sigma_0^2} z \quad (10.38)$$

where E is the energy per bit that can be safely assumed equal to 1 in simulations.

The Min-Sum decoding algorithm

For an AWGN channel and binary (BPSK) modulation, let x_n be the n -th transmitted bit, n_n the sample of Additive White Gaussian Noise (AWGN), and y_n the sample in the receiver. The modulation and channel transfer can be then modelled simply by (10.39). Note that with this choice for BPSK mapping, a logical zero bit is mapped to an $s = +1$ prototype in the signal space, while a logical one value is mapped to $s = -1$.

$$y_n = (-2x_n + 1) + n_n \quad (10.39)$$

Let us choose the posterior-probability derived LLR metric to be used with the soft-decoding Min-Sum algorithm, denoted as Z_n and defined by (10.40). We choose the second variant of the LLR metric definition shown previously in (10.35) in order to be compatible with the most sources.

$$Z_n^{(0)} = \ln \frac{p(x_n = 0 / y_n)}{p(x_n = 1 / y_n)} = 2y_n / \sigma^2 \quad (10.40)$$

where the σ^2 is the channel variance (may be omitted in the Min-Sum implementation) and the superscript (k) indicates the iteration number.

Before the iteration starts, we must first define two important buffers: the Z_{mn} matrix of messages send out by the variable nodes towards the check nodes, and the L_{mn} matrix of messages send out of the check nodes towards variable nodes. Both matrices will have the same size as $\mathbf{H}$ and will only contain nonzero values at positions where $\mathbf{H}$ has ones. We only need to initialize the values of Z_{mn} and we do it in the following way: For each row m of Z_{mn} we copy the input metrics Z_n at indices $N(m)$ to Z_{mn} , using the indices $N(m)$ also as the target column indices. This way we multiply some values and from the memory usage perspective this representation is in fact sub-optimal. It is, however, convenient for algorithm

understanding. The L_{mn} metrics need no special initialization, but setting all values to zero is a good tip to avoid potential bugs.

After initialization of important buffers, the iteration process may begin. Each of the iteration implements the following steps:

1. The horizontal scan – the check node update:
For each m and each $N(m)$ calculate the check node message L_{mn} , based on the variable node messages Z_{mn} coming from all incident variable nodes except the output one (with index n).

$$L_{mn}^{(k)} = \min_{n' \in N(m) \setminus n} |Z_{mn'}^{(k)}| \cdot \prod_{n' \in N(m) \setminus n} \text{sgn}(Z_{mn'}^{(k)}) \quad (10.41)$$

2. The vertical scan - the variable node update:
For each n and each $M(n)$ calculate the variable node message Z_{mn} coming from all incident check nodes, except the output one (with index m).

$$Z_{mn}^{(k)} = Z_n^{(0)} + \sum_{m' \in M(n) \setminus m} L_{m'n}^{(k)} \quad (10.42)$$

3. For each n and each $M(n)$ calculate the posterior LLR estimate $Z_n^{(k)}$:

$$Z_n^{(k+1)} = Z_n^{(0)} + \sum_{m \in M(n)} L_{mn}^{(k)} \quad (10.43)$$

4. Hard decision and termination:
For each codeword position n calculate the bit estimate $\hat{x}_n$ using (10.44) and if the orthogonality with the $\mathbf{H}$ matrix condition is satisfied: $\mathbf{H}\hat{\mathbf{x}} = \mathbf{0}$ (or $\hat{\mathbf{x}}\mathbf{H}^T = \mathbf{0}$) over GF(2), terminate the decoding.

$$\hat{x}_n^{(k)} = \begin{cases} 0, & \text{if } Z_n^{(k)} > 0; \\ 1, & \text{otherwise} \end{cases} \quad (10.44)$$

Note: The values of Z_{mn} are initialized by received channel samples: $Z_{mn}^{(0)} := Z_n^{(0)}$ and equations (10.42) and (10.43) can be easily reordered and (10.41) replaced with:

$$Z_{mn}^{(k+1)} = Z_n^{(k+1)} - L_{mn}^{(k)} \quad (10.45)$$

This replacement will be done in actual implementations since evaluating (10.42) is much more complicated than just (10.45). ■

For many students, the equations above make little sense as they see them for the first time. It is therefore beneficial to visualize the equations above using an example:

Example 10.6: Passing of messages during the horizontal a) and vertical b) step during a belief propagation decoding. The Tanner graph used represents a Hamming [7, 4, 3] code, see Fig. 10.14 and Fig. 10.15.

- a) Fig. 10.14: horizontal step for check node index $m = 2$ and output variable node index $n = 6$

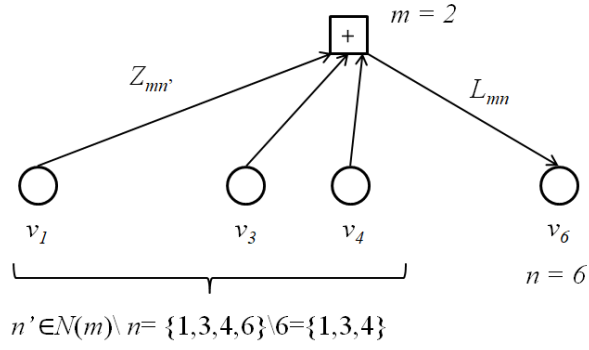


Fig. 10.14: Similar message passing will need to take place $|N(m)| = 4$ times. Each time with a different output node index n . In this case four different pictures could be drawn.

b) Fig. 10.15: Vertical step for variable node index $n = 4$ and output check node index $m = 3$.

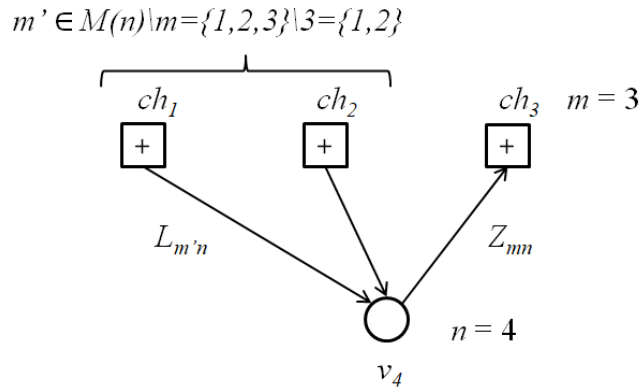


Fig. 10.15: Similar message passing will need to take place $|M(n)| = 3$ times. Each time with a different output node index m . In this case three different pictures could be drawn.

With basic insights on the message passing, let us dig a little deeper into the indexing: we need to keep several indices in mind:

1. the absolute index of the variable node n , which is the same thing as the index of the column in the $\mathbf{H}$ matrix.
2. The absolute index of the check node m , which is the same as the index of the row of $\mathbf{H}$.
3. The index of the iteration (k) written in superscript.
4. The relative index n' , used in the horizontal step for iteration over the set of variable node indices $N(m) \setminus n$.
5. The relative index m' , used in the vertical step for iteration over the set of check node indices $M(n) \setminus m$.

■

Example 10.7: For the example [7, 4, 3] Hamming code with the $\mathbf{H}$ matrix and variable and check nodes indices defined as:

$$\mathbf{H} = \begin{pmatrix} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{matrix} ch_1 \\ ch_2 \\ ch_3 \end{matrix}$$

Let the receiver obtain the following LLR metrics: $Z_n^{(0)} = (1, -2, 3, -4, 5, 6, -7)$. Let us observe the Min-Sum algorithm in operation:

For a good visual insight, it is useful to map the input metric vector $Z_n^{(0)}$ to a matrix $Z_{mn}^{(0)}$, we do this by replacing each 1 in the $\mathbf{H}$ matrix by the value of the $Z_n^{(0)}$ vector at the same column index:

$$Z_{mn}^{(0)} = \begin{pmatrix} 0 & -2 & 3 & -4 & 5 & 0 & 0 \\ 1 & 0 & 3 & -4 & 0 & 6 & 0 \\ 1 & -2 & 0 & -4 & 0 & 0 & -7 \end{pmatrix}$$

Now we have initialized the Z_{mn} matrix and we proceed to the horizontal step. For each of the row index m , we calculate the equation (40). The trick here is to realize, that (40) actually defines several equations: each one for a different output variable node index n , out of the set of indices $N(m)$, thus also defining several different subsets of input nodes $N(m) \setminus n$. Let us calculate for the rows of the example $\mathbf{H}$ matrix:

For the first row, the index (of the check node) $m = 1$, and the set of incident variable node indices is $N(1) = \{2, 3, 4, 5\}$. We use each index n as an output index:

If $n = 2$ is the output index, the set $N(m) \setminus n$ will be $\{2, 3, 4, 5\} \setminus \{2\} = \{3, 4, 5\}$.

Now we can calculate (10.41):

$$\begin{aligned} L_{1,2}^{(0)} &= \min_{n' \in \{3,4,5\}} |Z_{1,n'}^{(0)}| \cdot \prod_{n' \in \{3,4,5\}} \text{sgn}(Z_{1,n'}^{(0)}) = \\ &= \min(|3|, |-4|, |5|) \cdot (\text{sgn}(3) \cdot \text{sgn}(-4) \cdot \text{sgn}(5)) = -3 \end{aligned}$$

If $n = 3$ is the output index, the set $N(m) \setminus n$ will be $\{2, 3, 4, 5\} \setminus \{3\} = \{2, 4, 5\}$.

$$\begin{aligned} L_{1,3}^{(0)} &= \min_{n' \in \{2,4,5\}} |Z_{1,n'}^{(0)}| \cdot \prod_{n' \in \{2,4,5\}} \text{sgn}(Z_{1,n'}^{(0)}) = \\ &= \min(|-2|, |-4|, |5|) \cdot (\text{sgn}(-2) \cdot \text{sgn}(-4) \cdot \text{sgn}(5)) = 2 \end{aligned}$$

If $n = 4$ is the output index, the set $N(m) \setminus n$ will be $\{2, 3, 4, 5\} \setminus \{4\} = \{2, 3, 5\}$.

$$\begin{aligned} L_{1,4}^{(0)} &= \min_{n' \in \{2,3,5\}} |Z_{1,n'}^{(0)}| \cdot \prod_{n' \in \{2,3,5\}} \text{sgn}(Z_{1,n'}^{(0)}) = \\ &= \min(|-2|, |3|, |5|) \cdot (\text{sgn}(-2) \cdot \text{sgn}(3) \cdot \text{sgn}(5)) = -2 \end{aligned}$$

If $n = 5$ is the output index, the set $N(m) \setminus n$ will be $\{2, 3, 4, 5\} \setminus \{5\} = \{2, 3, 4\}$.

$$\begin{aligned}
L_{1,5}^{(0)} &= \min_{n' \in \{2,3,4\}} |Z_{1,n'}^{(0)}| \cdot \prod_{n' \in \{2,3,4\}} \text{sgn}(Z_{1,n'}^{(0)}) = \\
&= \min(|-2|, |3|, |-4|) \cdot (\text{sgn}(-2) \cdot \text{sgn}(3) \cdot \text{sgn}(-4)) = 2
\end{aligned}$$

For the second row index $m = 2$, the set $N(2) = \{1, 3, 4, 6\}$, so we would again need to actually calculate (10.40) four times. We do it here just for the $n = 1$:

$$\begin{aligned}
L_{2,1}^{(0)} &= \min_{n' \in \{3,4,6\}} |Z_{2,n'}^{(0)}| \cdot \prod_{n' \in \{3,4,6\}} \text{sgn}(Z_{2,n'}^{(0)}) = \\
&= \min(|3|, |-4|, |6|) \cdot (\text{sgn}(3) \cdot \text{sgn}(-4) \cdot \text{sgn}(6)) = -3
\end{aligned}$$

For the third row index $m = 3$, the set $N(3) = \{1, 2, 4, 7\}$ and we would again need to calculate (10.40) four times. The reader can perform these calculations as an exercise and should obtain the following resulting matrix $L_{mn}^{(0)}$ of metrics as the horizontal step output:

$$L_{mn}^{(0)} = \begin{pmatrix} 0 & -3 & 2 & -2 & 2 & 0 & 0 \\ -3 & 0 & -1 & 1 & 0 & -1 & 0 \\ -2 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

The vertical step is very similar, except we process the $L_{mn}^{(k)}$ matrix on a per-column basis. Eq. (10.42) defines the same approach using the $M(n) \setminus m$ sets, but we can use the simplification: just calculate the sums of each of the columns of $L_{mn}^{(k)}$ in (10.43) to obtain $Z_n^{(k+1)}$ and then subtract as in (10.45):

$$\begin{aligned}
Z_n^{(1)} &= Z_n^{(0)} + \sum_{m \in M(n)} L_{mn}^{(0)} = \\
&= (1 \quad -2 \quad 3 \quad -4 \quad 5 \quad 6 \quad -7) + \text{sum}_C \begin{pmatrix} 0 & -3 & 2 & -2 & 2 & 0 & 0 \\ -3 & 0 & -1 & 1 & 0 & -1 & 0 \\ -2 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} = \\
&= (1 \quad -2 \quad 3 \quad -4 \quad 5 \quad 6 \quad -7) + (-5 \quad -2 \quad 1 \quad 0 \quad 2 \quad -1 \quad 1) = \\
&= (-4 \quad -4 \quad 4 \quad -4 \quad 7 \quad 5 \quad -6)
\end{aligned}$$

Now to calculate the Z_{mn} metrics for the input to the new iteration, we subtract the value of L_{mn} for each matrix position. We must be careful to correctly interpret (10.45), since the buffers in question are not all of the same size: Z_{mn} and L_{mn} are matrices of the size $M \times N$, while Z_n is a vector of length N . In order to correctly implement (10.45) we may again “extend” the vector Z_n to a temporary matrix form, similarly to the algorithm initialization step as shown below:

$$\begin{aligned}
Z_{mn}^{(1)} &= Z_n^{(1)} - L_{mn}^{(0)} = \\
&= \begin{pmatrix} -4 & -4 & 4 & -4 & 7 & 5 & -6 \end{pmatrix} - \begin{pmatrix} 0 & -3 & 2 & -2 & 2 & 0 & 0 \\ -3 & 0 & -1 & 1 & 0 & -1 & 0 \\ -2 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \approx \\
&\approx \begin{pmatrix} 0 & -4 & 4 & -4 & 7 & 0 & 0 \\ -4 & 0 & 4 & -4 & 0 & 5 & 0 \\ -4 & -4 & 0 & -4 & 0 & 0 & -6 \end{pmatrix} - \begin{pmatrix} 0 & -3 & 2 & -2 & 2 & 0 & 0 \\ -3 & 0 & -1 & 1 & 0 & -1 & 0 \\ -2 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} = \\
&= \begin{pmatrix} 0 & -1 & 2 & -2 & 5 & 0 & 0 \\ -1 & 0 & 5 & -5 & 0 & 6 & 0 \\ -2 & -5 & 0 & -5 & 0 & 0 & -7 \end{pmatrix}
\end{aligned}$$

Note that this expansion of the vector to matrix isn't really necessary in actual implementation; it is just convenient for the demonstration purpose here.

We have not demonstrated the first iteration: the alternative execution of the horizontal and vertical steps. Before we would continuous with the next iteration, we must check the stopping criterion (the decoding may already be successful): We perform the hard decision as defined by (10.44) on each of the metrics to obtain transmitted codeword estimate:

We get: $\hat{\mathbf{x}} = (1, 1, 0, 1, 0, 0, 1)$ and we must perform the test or orthogonality with the $\mathbf{H}$ matrix:

$$\hat{\mathbf{x}}\mathbf{H}^T = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$$

Since the orthogonality condition is satisfied, we terminate the decoding process after exactly one iteration. In actual implementations, several iterations will be necessary, while sometimes several tens of iterations may be needed. However, the convergence of the algorithm is not guaranteed, so there must be a hard limit placed on the number of iterations.

Alternatively, and just for the sake of example, we may also try using the definition in eq. (10.42) and we should really arrive to the same numbers:

Let us evaluate the (10.42):

$$Z_{mn}^{(k)} = Z_n^{(0)} + \sum_{m' \in M(n) \setminus m} L_{m'n}^{(k)}$$

For each column index n :

The structure of the $\mathbf{H}$ matrix determines the sets of variable node neighborhoods:

$$M(1) = \{2, 3\}, M(2) = \{1, 3\}, M(3) = \{1, 2\}, M(4) = \{1, 2, 3\},$$

$$M(5) = \{1\}, M(6) = \{2\}, M(7) = \{3\}.$$

If $n = 1$ the set of incident check node indices is $M(1) = \{2, 3\}$ and each one of these will serve as an output:

If $m = 2$ is the output index, the set $M(n) \setminus m$ will be $\{2, 3\} \setminus \{2\} = \{3\}$.

$$Z_{2,1}^{(1)} = Z_1^{(0)} + \sum_{m' \in \{3\}} L_{m'1}^{(0)} = Z_1^{(0)} + L_{3,1}^{(0)} = 1 + (-2) = -1$$

If $m = 3$ is the output index, the set $M(n) \setminus m$ will be $\{2, 3\} \setminus \{3\} = \{2\}$.

$$Z_{3,1}^{(1)} = Z_1^{(0)} + \sum_{m' \in \{2\}} L_{m'1}^{(0)} = Z_1^{(0)} + L_{2,1}^{(0)} = 1 + (-3) = -2$$

If $n = 2$ the set of incident check node indices is $M(2) = \{1, 3\}$ and each one of these will serve as an output:

If $m = 1$ is the output index, the set $M(n) \setminus m$ will be $\{1, 3\} \setminus \{1\} = \{3\}$.

$$Z_{1,2}^{(1)} = Z_2^{(0)} + \sum_{m' \in \{3\}} L_{m'2}^{(0)} = Z_2^{(0)} + L_{3,2}^{(0)} = -2 + (1) = -1$$

If $m = 3$ is the output index, the set $M(n) \setminus m$ will be $\{1, 3\} \setminus \{3\} = \{1\}$.

$$Z_{3,2}^{(1)} = Z_2^{(0)} + \sum_{m' \in \{1\}} L_{m'2}^{(0)} = Z_2^{(0)} + L_{1,2}^{(0)} = -2 + (-3) = -5$$

If $n = 4$ the set of incident check node indices is $M(4) = \{1, 2, 3\}$ and each one of these will serve as an output:

If $m = 1$ is the output index, the set $M(n) \setminus m$ will be $\{1, 2, 3\} \setminus \{1\} = \{2, 3\}$.

$$Z_{1,4}^{(1)} = Z_4^{(0)} + \sum_{m' \in \{2,3\}} L_{m'4}^{(0)} = Z_4^{(0)} + (L_{2,4}^{(0)} + L_{3,4}^{(0)}) = -4 + (1+1) = -2$$

If $m = 2$ is the output index, the set $M(n) \setminus m$ will be $\{1, 2, 3\} \setminus \{2\} = \{1, 3\}$.

$$Z_{2,4}^{(1)} = Z_4^{(0)} + \sum_{m' \in \{1,3\}} L_{m'4}^{(0)} = Z_4^{(0)} + (L_{1,4}^{(0)} + L_{3,4}^{(0)}) = -4 + (-2+1) = -5$$

If $m = 3$ is the output index, the set $M(n) \setminus m$ will be $\{1, 2, 3\} \setminus \{3\} = \{1, 2\}$.

$$Z_{3,4}^{(1)} = Z_4^{(0)} + \sum_{m' \in \{1,2\}} L_{m'4}^{(0)} = Z_4^{(0)} + (L_{1,4}^{(0)} + L_{2,4}^{(0)}) = -4 + (-2+1) = -5$$

If $n = 5$ the set of incident check node indices is $M(5) = \{1\}$ and each one of these will serve as an output:
For $m = 1$:

$$Z_{1,5}^{(1)} = Z_5^{(0)} + \sum_{m' \in \emptyset} L_{m'4}^{(0)} = Z_5^{(0)} = 5$$

■

Lab 10c: MATLAB exercises: Min-Sum decoding of LDPC codes

MATLAB Exercise 10.17 - find $N(m)$ and $M(n)$ sets

For each of the $\mathbf{H}$ matrix defined below, find all the sets of (linear) indices $N(m)$ of columns – one index set (vector of indices) of incident columns for each row, as shown in Example 10.5. Also find all the sets $M(n)$ of rows incident with each column n .

MATLAB Exercise 10.18: calculate LLR for an AWGN channel

Implement the body of the function with prototype:

```
function Zn0 = detectorSamplesLLR( RxBlock , noiseVariance )
```

that calculates the LLR metric (defined in 10.35) for a BPSK modulation (defined in 10.39) and AWGN channel, as defined in (10.38). The function must accept scalars, vectors, or matrices.

MATLAB Exercise 10.19: calculate hard decision for given LLR

Implement the body of the function with prototype:

```
function bits = hardDecision( LLR )
```

that calculates the hard decision (10.44) on an LLR metric input, consistently with the definitions in the previous exercise. The function must accept scalars, vectors, or matrices.

MATLAB Exercise 10.20: the Min-Sum algorithm LDPC decoder

Using the prepared source file `decodeLDPCassignment.m` to finish the implementation of the body of the function with prototype:

```
function Xne = decodeLDPC( H, Zn0, nIter, verbose )
```

that:

- A) Initializes the Z_{mn} buffer as a matrix $Z_{mn}^{(0)} := Z_n^{(0)}$. To be more precise, the Z_{mn} buffer should be a matrix of the same size and structure as the $\mathbf{H}$ matrix, i.e. it should have nonzero values only at positions where $\mathbf{H}$ has ones. The $Z_n^{(0)}$ is a vector of input LLRs of length n . Its values are copied to each row of Z_{mn} so that a binary one in $\mathbf{H}$ at index (m, n) defines the assignment $Z_{mn} := Z_n$. A zero in $\mathbf{H}$ at index (m, n) defines a zero value at Z_{mn} . See the example 10.8 if unclear.
- B) Finds the neighborhood index set $N(m)$ for each row index m of $\mathbf{H}$.
- C) Performs the horizontal step by implementing a private function with a given prototype:

```
function Lmn = horizontal( N, Nm, Zmn, verbose )
```

This function should implement the equations (10.41) for a given row m , and for each of the incident variable nodes $n \in N(m)$ set as output as demonstrated in Example 10.7. The function should take the following inputs parameters:

 - N – a scalar -the length of the codeword
 - Nm – a vector - the set of indices $N(m)$ for current row
 - Zmn – a vector - the input buffer of messages from variable nodes – a vector of size $|N(m)|$.

The function should output a vector of check node messages Lmn of size N , including the redundant zero values. This is suggested for debugging purposes, so that the user can clearly see the connection between the messages and their position in the Tanner graph. See debugging example 10.8 if unclear.
- D) Updates $Z_n(k)$ values by evaluating eq. (10.43) for each column.
- E) Updates $Zmn(k)$ values by eq. (10.45) in a similar way as performed in the initializations step.
- F) Performs hard decision on the resulting posterior LLR estimates $Z_n(k)$ and tests orthogonality of the codeword estimate and $\mathbf{H}$ matrix. If orthogonal, or the maximum number of decoder iterations $nIter$ is reached, return the current codeword estimate under the parameter Xne .

For debugging purposes, use the provided `dbg()` function to be able to easily turn on and off debug output using the `verbose` parameter. Save the resulting source file as a new file `decodeLDPC.m`.

Note: The implementation described here has some potential trivial optimizations, especially regarding the memory usage. The preservation of the buffers Z_{mn} and L_{mn} structure similar to the $\mathbf{H}$ matrix was, however, a priority here.

Debugging

In order to be truly able to correctly implement the Min-Sum algorithm, it is beneficial to demonstrate the algorithm operation by printing a simple debug output. This can be useful to catch bugs in the implementation. The following two examples are given precisely to facilitate debugging of the Min-Sum implementation prescribed by exercises below:

Example10.8:

Debug output for a Min-Sum decoder applied to a [7, 4, 3] Hamming code with the $\mathbf{H}$ matrix in a non-systematic form. The Min-Sum decoder maximum iterations limit was set to 6 and the decoding converged successfully after the 5th iteration. The codeword $\mathbf{c}$ is BPSK modulated and vector $\mathbf{t}$ is transmitted. The AWGN noise takes on the value $\mathbf{n}$, and the input to Min-Sum is not an LLR. Instead the direct receiver sample is used for $\mathbf{Z}_0$. The listing below shows the debug output of a reference MATLAB implementation. $\mathbf{X}_{ne}$ denotes the codeword estimate within each iteration.

```
H = 1 0 1 0 1 0 1
    1 0 1 1 0 1 0
    0 1 1 0 0 1 1

C = 1 0 1 1 0 1 0
T = -1 1 -1 -1 1 -1 1
N = -0.8223 -0.0942 0.3362 -0.9047 -0.2883 0.3501 -1.8359
Z = -1.8223 0.9058 -0.6638 -1.9047 0.7117 -0.6499 -0.8359

ZMN initialized =
-1.8223 0 -0.6638 0 0.7117 0 -0.8359
-1.8223 0 -0.6638 -1.9047 0 -0.6499 0
0 0.9058 -0.6638 0 0 -0.6499 -0.8359

1st iteration:
LMN =
0.6638 0 0.7117 0 -0.6638 0 0.6638
-0.6499 0 -0.6499 -0.6499 0 -0.6638 0
0 -0.6499 0.6499 0 0 0.6638 0.6499
Znk =
-1.8084 0.2559 0.0479 -2.5546 0.0479 -0.6499 0.4778
ZMN =
-2.4722 0 -0.6638 0 0.7117 0 -0.186
-1.1585 0 0.6978 -1.9047 0 0.0139 0
0 0.9058 -0.602 0 0 -1.3137 -0.1721
Xne =
1 0 0 1 0 1 0

2nd iteration:
LMN =
0.186 0 0.186 0 -0.186 0 0.6638
-0.0139 0 0.0139 -0.0139 0 0.6978 0
0 -0.1721 0.1721 0 0 0.1721 0.602
Zn0 =
-1.8223 0.9058 -0.6638 -1.9047 0.7117 -0.6499 -0.8359
```

```

Znk =
-1.6502      0.7337      -0.2918      -1.9186      0.5257      0.22      0.4299
ZMN =
-1.8362      0      -0.4778      0      0.7117      0      -0.2339
-1.6363      0      -0.3057      -1.9047      0      -0.4778      0
0      0.9058      -0.4639      0      0      0.0479      -0.1721

```

```

Xne =
1  0  1  1  0  0  0

```

3rd iteration:

```

LMN =
0.2339      0      0.2339      0      -0.2339      0      0.4778
-0.3057      0      -0.4778      -0.3057      0      -0.3057      0
0      0.0479      -0.0479      0      0      0.1721      -0.0479

```

```

Zn0 =
-1.8223      0.9058      -0.6638      -1.9047      0.7117      -0.6499      -0.8359

```

```

Znk =
-1.8941      0.9537      -0.9556      -2.2104      0.4778      -0.7835      -0.406

```

```

ZMN =
-2.128      0      -1.1895      0      0.7117      0      -0.8838
-1.5884      0      -0.4778      -1.9047      0      -0.4778      0
0      0.9058      -0.9077      0      0      -0.9556      -0.3581

```

```

Xne =
1  0  1  1  0  1  1

```

4th iteration:

```

LMN =
0.7117      0      0.7117      0      -0.8838      0      0.7117
-0.4778      0      -0.4778      -0.4778      0      -0.4778      0
0      -0.3581      0.3581      0      0      0.3581      0.9058

```

```

Zn0 =
-1.8223      0.9058      -0.6638      -1.9047      0.7117      -0.6499      -0.8359

```

```

Znk =
-1.5884      0.5477      -0.0718      -2.3825      -0.1721      -0.7696      0.7816

```

```

ZMN =
-2.3001      0      -0.7835      0      0.7117      0      0.0699
-1.1106      0      0.406      -1.9047      0      -0.2918      0
0      0.9058      -0.4299      0      0      -1.1277      -0.1242

```

```

Xne =
1  0  1  1  1  1  0

```

5th iteration:

```

LMN =
-0.0699      0      -0.0699      0      0.0699      0      0.7117
0.2918      0      -0.2918      0.2918      0      0.406      0
0      -0.1242      0.1242      0      0      0.1242      0.4299

```

```

Zn0 =
-1.8223      0.9058      -0.6638      -1.9047      0.7117      -0.6499      -0.8359

```

```

Znk =
-1.6004      0.7816      -0.9013      -1.6129      0.7816      -0.1197      0.3057

```

```

ZMN =
-1.5305      0      -0.8314      0      0.7117      0      -0.406
-1.8922      0      -0.6095      -1.9047      0      -0.5257      0
0      0.9058      -1.0255      0      0      -0.2439      -0.1242

```

```

Xne =
1  0  1  1  0  1  0

```

MATLAB Exercise 10.21: LDPC encoded transmission in AWGN channel – simulation results

After your program successfully produces the decoding output shown above in the debugging example, run the `SIM_LDPC.m` file that uses the `decodeLDPC()` function to simulate a BPSK transmission in AWGN channel. You should obtain results very similar to the figure below.

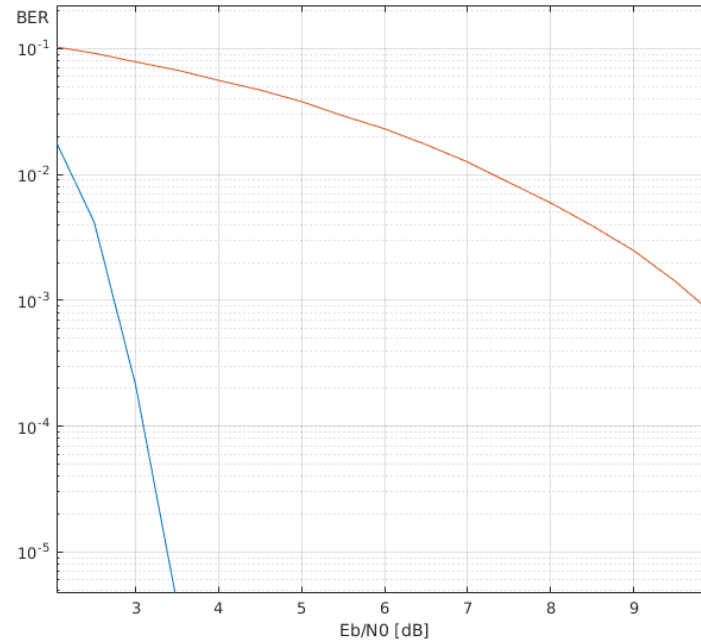


Fig. 10.16: Waterfall curve – the bit error ratio of an LDPC-encoded BPSK transmission before (red) and after decoding (blue). Basic Min-Sum algorithm [6] for 8 decoder iterations and number of block processed at once (block multiplier) $bMul = 10$.

List of acronyms

3G	3 rd Generation
AWGN	Additive White Gaussian Noise
BDPSK	Binary Differentially encoded Phase Shift Keying
BER	Bit Error Rate
BLER	Block Error Rate
BFSK	Binary Frequency Shift Keying
BPNRZ	Bipolar Non-Return to Zero
BPSK	Binary Phase Shift Keying
BPF	Bits Per Frame
BPW	Bits Per Word
BSC	Binary Symmetric Channel
CC	Convolutional Code
CCSDS	Consultative Committee for Space Data Systems
CDMA	Code Division Multiple Access
CM	Constellation Mapping
CR	Communication Resource
CW	Continuous Wave
DET	Detector
DFT	Discrete Fourier Transform
DL	Downlink
DMPSK	Differential M-ary Phase Shift Keying
DS	Direct Sequence
DVB	Digital Video Broadcasting
ECC	Error Control Code
EIRP	Effective Isotropically Radiated Power
ESD	Energy Spectral Density
FDMA	Frequency Division Multiple Access
FER	Frame Error Rate
FFH	Fast Frequency Hopping
FFT	Fast Fourier Transform
FH	Frequency Hopping
FIR	Finite Impulse Response
FPC	Fractional Power Containment
GF	Galois Field
GFT	Generalized Fourier Transform

GNSS	Global Navigational Satellite System
GPS	Global Positioning System
HP	Half Power
HW	Hardware
IEEE	Institute of Electrical and Electronics Engineers
ISA	Instruction Set Architecture
ISI	Inter Symbol Interference
IQ	In phase/Quadrature
LBC	Linear Block Codes
LDPC	Low Density Parity Check
LLR	Log-Likelihood Ratio
LPF	Low Pass Filter
LTE	Long Term Evolution
MA	Multiple Access
MAP	Maximal A'posteriori Probability
MDS	Maximal Distance Separable
MF	Matched Filter
MFSK	M-ary Frequency Shift Keying
ML	Maximum Likelihood
MPSK	M-ary Phase Shift Keying
MQAM	M-ary Quadrature Amplitude Modulation
NE	Noise Equivalent
PAM	Pulse Amplitude Modulation
PDMA	Polarization Division Multiple Access
pdf	probability density function
PN	Pseudo-random
PSD	Power Spectral Density
RAID	Redundant Array of Inexpensive/Independent Disks
RC	Raised Cosine
RCF	Raised Cosine Filter
RRC	Root Raised Cosine
RF	Radio Frequency
ROL	Rotate Left
ROR	Rotate Right
RS	Reed Solomon
RX	Receiver
QAM	Quadrature Amplitude Modulation
QC-LDPC	Quasi-Cyclic LDPC

QPSK	Quadrature Phase Shift Keying
SDMA	Space Division Multiple Access
SER	Symbol Error Rate
SFH	Slow Frequency Hopping
SHL	Shift Left
SHR	Shift Right
SJR	Signal-to-Jammer Ratio
SNR	Signal to Noise Ratio
SRC	Source
SS	Spread Spectrum
STAT	Statistics
TDMA	Time Division Multiple Access
TIMS	Telecommunications Instructional Modelling System
TX	Transmitter
UL	Uplink
UNRZ	Unipolar Non-Return to Zero
WiFi	Wireless Fidelity
WiMAX	Worldwide Interoperability for Microwave Access
WISSCE	Wireless Indoor Spread Spectrum Communication Equipment
WSS	Wide-Sense Stationary (Stochastic Process)

References

1. GALLAGER, R.G. Low-density parity-check codes, MIT Press Cambridge (USA), 1963. Available at: <https://direct.mit.edu/books/book/3867/Low-Density-Parity-Check-Codes> DOI: 10.7551/mitpress/4347.001.0001
2. TANNER, R. M., SRIDHARA, D., SRIDHARAN, A., et al. LDPC block and convolutional codes based on circulant matrices. IEEE Transactions on Information Theory, 2004, vol. 50, no. 12, pp. 2966-2984. DOI: 10.1109/TIT.2004.838370
3. LI, Z., CHEN, L., ZENG, L., et al. Efficient encoding of quasi-cyclic low-density parity-check codes. IEEE Transactions on Communications, 2006, vol. 54, no. 1, pp. 71-81. DOI:10.1109/TCOMM.2005.861667
4. IEEE Std. 802.16-2017, IEEE standard for air interface for broadband wireless access systems - Section 8.4.9.2.5: low density parity check (LDPC) code. IEEE New York (USA), 2018, pp. 1459 – 1463. ISBN 978-1-5044-4474-3
5. IEEE Std 802.11-2020, Part 11: wireless lan MAC and PHY specifications - Annex F: HT LDPC matrix definitions. IEEE New York (USA), 2021. p. 4130–4132. ISBN 978-1-5044-7283-8
6. Palenik T., Sztitkey V. HIGH THROUGHPUT OPEN-SOURCE IMPLEMENTATION OF Wi-Fi 6 AND WiMAX LDPC ENCODER AND DECODER. In: International Journal Information Technology Applications (ITA) Vol 11, Number 2, Dec 2022. ISSN: 2453-7497 (online), ISSN: 1338-6468 (print version) [Available: <https://www.paneurouni.com/veda/vedeckeczasopisy/aplikacie-informacnych-technologii/archiv-5/>]
7. IEEE Std. 802.3-2018, IEEE Standard for Ethernet 802.3-2018 (Revision of IEEE Std 802.3-2015) - Section 8: 200G and 400G Ethernet Chapter: 119.2.4.6. IEEE New York (USA), 2018, p.67–69. DOI: 10.1109/IEEESTD.2018.8457469
8. Hooper, T.: Volume D2 Further & Advanced Digital Experiments, TIMS, Emona Instruments Pty Ltd, Australia, 2004.
9. Radzyner R., Rakus, M.: Volume S1 TIMS Signals & Systems Experiment Manual, Communication Systems Modelling with TIMS, Emona Instruments Pty Ltd, Australia, 2011, ISBN 978-1-921903-02-1.
10. Rakus, M. et al.: Volume D4 Further Advanced Digital Experiments, TIMS, Emona Instruments Pty Ltd, Australia, 2004, ISBN 978-1-921903-00-7.
11. Rakus, M.: Volume D7 Advanced BER Experiments, TIMS, Emona Instruments Pty Ltd, Australia, 2006, ISBN 978-1-921903-10-6.
12. Rakus, M.: EMONA web page: TIMS-300 → Lab Manuals → Explorative Experiments: https://www.emona-tims.com/my-account/file-download/?search_product=3302&swcfpc=1
13. Rakus, M.: Volume D5 Basic Spread Spectrum Techniques, TIMS, Emona Instruments Pty Ltd, Australia, 2003, ISBN 978-1-921903-01-4
14. Rakus, M.: Volume D6 Advanced Spread Spectrum Experiments, TIMS, Emona Instruments Pty Ltd, Australia, 2002, ISBN 978-1-921903-09-0
15. Rakus, M.: Volume D8 Multipath Experiments, TIMS, Emona Instruments Pty Ltd, Australia, 2004, ISBN 978-1-921903-11-3
16. Rakus, M.: TIMS Signals & Systems Advanced Experiments – Volume 2, TIMS, Emona Instruments Pty Ltd, Australia, 2004.
17. MacKay, D. J. C. ; Neal, R. M. Near Shannon Limit Performance of Low Density Parity Check Codes. In Electronics Letters, vol. 32, issue 18, p. 1645, 1996. DOI: 10.1049/el:19961141

Digital communications – collection of solved problems and examples

M. Rakus, T. Palenik

Emona Instruments Pty Ltd ISBN 978-1-921903-15-1