A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

EFOP-3.4.3-16-2016-00009

Applied Information Theory

Course notes

University of Pannonia

by

István Vassányi, Benedek Szakonyi

2019

H-8200 Veszprém, Egyetem u.10.
H-8201 Veszprém, Pf. 158.
Telefon: (+36 88) 623-515
Internet: www.uni-pannon.hu

Magyarország Kormánya
Európai Unió
Európai Strukturális és Beruházási Alapok
BEFEKTETÉS A JÖVŐBE

Contents

FOREWORD

This text attempts to present an application-oriented introduction to information theory, with less theorems and proofs than usual, and more engineering considerations. The approach and notation is very similar to that of the excellent textbook [1], with some simplifications and extensions that seemed necessary for a single-semester university course. For those students who have an interest in the more detailed mathematical background of the material, the books [2] and [3] are recommended for further reading.

[1] Richard B. Wells: *Applied Coding and Information Theory for Engineers*, Prentice Hall, 1999.
[2] Steven Roman: *Introduction to Coding and Information Theory*, Springer, 1997.
[3] R.B. Ash: *Information Theory*, Dover Publications, 1990.

I SOURCE CODING

1. Entropy and Mutual Information

1.1. The Cocept of Information (Everyday, Philosophical, Engineering)

The philosophical definition of information, the basis of epistemology[1] [2].

- Plato: **Phaedo** (5th century B.C.). *"If we truly wish to know something, we must free ourselves from the body and contemplate through only the soul itself"*. Based on this, the soul with its spiritual vision can see the true meaning of things, the inner **form,** idea, eidos (form, essence) but only after it has freed itself from the body. This theory of pure, direct intellectual understanding probably had great influence on Aristotle.
- Aristotle: **On the Soul**. Plato's student, Aristotle, with this work of his has laid the foundations of how ancient Christian, Jewish and Arabic philosophies described the meaning of the soul itself. In his radical epistemological thesis, he stated that "*the soul, in a certain way, is unified with every living thing*". During the process of comprehension, the deeper part of the human intellect, the so-called passive intellect, alters itself to the subject's **inner form**, to its substance. This alteration is the inner development of the soul (which itself is a form, the inner form of the body), this is **information**.
- Eriugena: **The Division of Nature** (Neoplatonic Christian mystic). By the Christian theological-anthropological aspect **form** is the inner form of humans, which Eriugena considers to be nothing else but Christ himself. Thus mankind's task is to **conform** to this holy inner form.
- Ficino: **Five Questions Concerning the Mind** (end of 15th century). Marsilio Ficino was one of the most influential humanist philosophers of the early Italian Renaissance in the 15th century. His most important work is the *Theologia platonica de immortalitate animorum* (Platonic Theology). Following in the footsteps of Aristotle, Ficino discusses how the soul sees God through a "vision of intellect". This mixture of Aristotelian philosophy of soul and Neoplatonic mysticism states that with this vision the soul

[1] Epistemology: a branch of philosophy that investigates the origin, nature, methods, and limits of human knowledge.
[2] The original and Hungarian versions of the quoted works are available at the lecturer of the course

transforms into God: God as a **form** joins itself to the soul ("tamquam forma … animae sese iungit")

- John Locke: **An Essay Concerning Human Understanding** (1960). His take on metaphysics, based on Descartes' opinion, is dualism. Locke rejects the Aristotelian philosophy of soul and the Platonic idealism. Despite stating that things have no inner form, the word *information* appears several times in Locke's work, however, not in the modern meaning of manipulable data but rather as someone's inner, moral education or development.

News, Data, Information, Knowledge. The everyday definition of information.
Data and news are carriers of information and the amount of conveyed information is said to be "the amount of knowledge we gained by acquiring it". It is also commonly said that information is "processed data". As an intuitive approach to information, we should take a look at the methods of those who trade in information (e.g. newspapers). We can see that the value of news is based on two related elements:

- The surprise factor (how amazing/shocking it is)
- The relevance factor (how much the news affects the recipient)

An example for a news article concerning everyone would be *"Tomorrow the sun will not rise"*. A less relevant information would be that *"John Doe has finally passed his exams."* (of course for his relatives this is a great achievement worthy of being on the cover page of the family paper, would such exist).
Unexpected components increase the value of information: a statement such as *"The student's mess hall will be closed tomorrow due to pest control"* might go as far as the university news, but one like *"The student's mess hall will be closed tomorrow due to a rattlesnake infestation"* could appear in the nationwide media.

From these two factors of information's everyday meaning, the engineering definition and *information theory* apply only the first, the "surprise factor". The fundamentals of this field of study were laid by C.E. Shannon (1916-2001) in 1948.

1.2. Characterising Discrete Sources, Entropy

Basically our aim is to capture and transmit the information through space (e.g. broadcasting) and/or time (e.g. a book, DVD) from a source to a destination (often called sink). The carriers of the information can be diverse, e.g. radio waves, wires, smoke signals, etc. These carriers are usually referred to as the "**channel**". Unfortunately, in practical applications we always have to take the presence of **noise** into account in this channel.

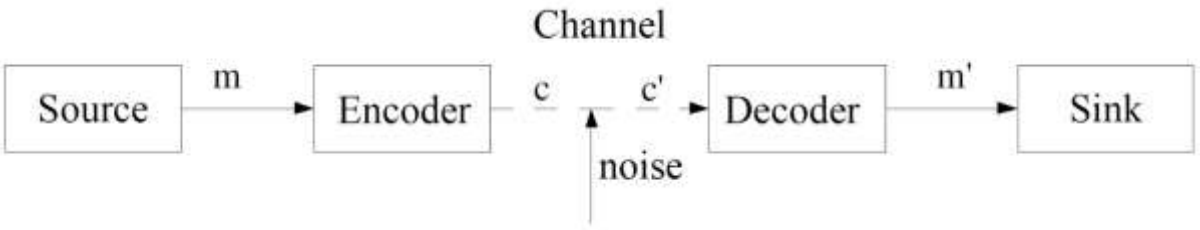


Figure 1.1: *Basic model of information transmission*

In Figure 1.1 *m* is the message to be transferred, *c* is the encoded message sent via the channel, *c'* is the received message, and *m'* is an estimate of the original message.
The two most important goals of information transmission:

- get the message to the sink with as little distortion as possible (where the error is introduced by the channel's noise)

- use the channel as efficiently as possible (as a price has to be paid for the usage)

To achieve high quality information transmission, we introduce a **decoder** and an **encoder**. The former at the beginning of the channel, the latter at the end. Both are designed according to the previous two goals.

A message is interpreted as a series of symbols emitted by a source. We can distinguish between two types of symbol sets: the **discrete** (e.g. letters) and the **continuous** (e.g. vocal chords) ones. Based on how the symbols are distributed in time we can also identify two groups: **discrete** (e.g. written text) and **continuous** (e.g. speech) cases.
For both continuous categories we can produce the discrete equivalents by applying the proper sampling and quantizing methods (e.g. writing down a conversation to a piece of paper). Because of this property the field of information theory deals primarily with sources of discrete-time and discrete symbols sets, and so does this syllabus from now on.

A source can be characterised by the symbols set it generates (**source alphabet**) and the corresponding probabilities of how frequently each symbol is emitted (**source distribution**):

$$A = \{a_0, a_1, \dots, a_{M-1}\},$$
$$P(A) = \{p_0, p_1, \dots, p_{M-1}\}, \sum_i p_i = 1\,, p_i \neq 0$$

where *M* is the cardinality of the symbol set A (the number of its elements, also written as |A|), and the sum of the probabilities is 1 (i.e. the source always emits a member of its alphabet). Apart this, the sources mentioned in this syllabus are (in most cases) expected to also be

- **stationary** – the probabilities are independent of time
- **memoryless** – the symbols (and thus their probabilities) are statistically independent from each other

The "amount of surprise", i.e. the information content of the occurrence of the i-th symbol can be defined as

$$I(a_i) = \log\frac{1}{p_i}\ [\text{bit}]$$

The unit of information is **bit** (binary unit)[3], because a logarithm of base 2 is used. It can be seen that the information contents of statistically independent events are to be summed.
The **entropy** of a source is *the average amount of information conveyed per source symbol*:

$$H(A) = \sum_i p_i \log\frac{1}{p_i}\ [\text{bit}]$$

For example, a source P(A) = {0.5, 0.5} has an entropy of 1 bit. If we allow the values of symbol probability to be 0 or 1, a source P(A) = {0, 1} can be constructed that has the entropy of 0 bit[4] (but it is quite a boring source as it emits the same symbol over and over again). It can be shown that equal distribution maximises entropy and this maximum value is log(*M)* (see Appendix A1). From this we can conclude that for any source with M number of symbols $0 \leq H(A) \leq \log(M)$.

An example: for source P(A) = {0.4, 0.6}, H(A) = 0.970 bits. From now on, instead of P(A) = {…}, a shorter notation A = {…} will be used if the specific symbols are not that important in the problem.

[3] Not to be mistaken as the digits of binary numbers, which are called the same. The connection between these two is that the information content of a digit of a binary number can be *at most* 1 bit.

[4] because $\lim_{x\to 0} x\ \log\frac{1}{x}=0$

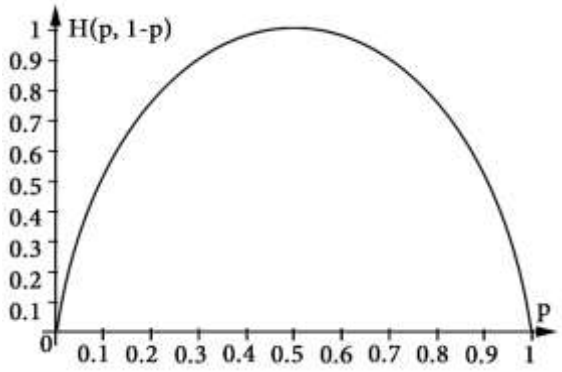


Figure 1.2: *The connection between a source's entropy and distribution*

1.3. Conditional Entropy and Mutual Information

These two concepts are used to describe the mutual, average information content of two or more sources. They will also appear in the coding methods related section.

Suppose we have two sources A and B, e.g. the current images displayed on two separate TV screens. Let $p_{i,j}$ denote the probability of source A emitting the i-th and B the j-th symbol. Then the **mutual (joint) entropy** and the **conditional entropy** can be defined as

$$H(A,B) = \sum_{i,j} p_{i,j} \log \frac{1}{p_{i,j}}$$

$$H(B|A) = \sum_{i} p_i \sum_{j} p_{j|i} \log \frac{1}{p_{j|i}}$$

where $p_{j|i}$ might be 0, for this case see footnote #4. The joint entropy H(A, B) is the average amount of information conveyed by both sources together. It can be seen that if the sources are independent, then H(A, B) is the sum of the individual entropies.
If we expand the definition of H(A ,B) and we apply $p_{i,j} = p_i p_{j|i}$, we get the equation

$$H(A,B) = H(A) + H(B|A)$$

where H(B|A) is the entropy of B given A, the conditional entropy[5]. This shows how much *additional* information we gain from B on average, if we already know source A. From the equation it can be seen that H(B|A) conditional entropy is nothing else but the weighted average of the entropies computed form the conditional source distributions that belong to events (symbols) in A. Of course, it is also true that H(A, B) = H(B) + H(A|B).

If source B is **completely independent** of A (e.g. one TV shows football and the other displays romantic series) than we expect H(A, B) joint entropy to be equal to the sum of the individual entropies of the sources (our knowledge is increased from both sources individually). Indeed, based on the statistical independence of the symbol probabilities the conditional probabilities become "unconditional", thus H(B|A) turns into H(B) and we get H(A, B) = H(A) + H(B).

On the other hand, if B is **completely dependent** on A then we expect no additional information from B (if both TVs show the same then we would know exactly the same if we had only one device). Which means H(A, B) = H(A). Evidently, because B is completely dependent, all conditional probabilities $p_{j/i}$ are either 0 or 1, so H(B|A) = 0.
Therefore, we can state that

[5] It is easier to memorise this equation if we notice its similarity to the connection between conditional and joint probabilities, except that because of the logarithmic definition of information, here we have addition instead of multiplication.

$$0 \leq H(B|A) \leq H(B),$$

i.e. *side information does not increase entropy*. An example is the parity bit, a completely defined source that has less entropy (Example 1.1).

The **mutual information I(B, A)** of sources A and B can be defined as that part of the average information content of B which is provided by source A. To get this we need to subtract from H(B) the part that is *not* conveyed in A, i.e. the average "surprise" caused by B compared to A, which is nothing else but H(B|A):

$$I(B,A) = H(B) - H(B|A)$$

It can be easily seen that because of the two possible interpretation of H(A, B)

$$I(B,A) = I(A,B)$$

If the two sources are **completely independent**[6] of each other, then conditional entropy H(B|A) turns into H(B), thus the mutual information is 0, what it is expected to be. However, if B is completely dependent on A then H(B|A) = 0, so I(B, A) = H(B), i.e. the complete H(B) is contained in source A. But this does not imply that H(A) = H(B). For example, if A has more symbols than B (|A| > |B|), then H(A) > H(B), so B has less average information content. So in the case when such an A is a result of an observation at the beginning of a communication process and B is generated at the end, then we can conclude that some information is lost during the transmission.

The same is true for every situation in which H(B) is not smaller than H(A) yet I(B, A) < H(A) because some part of the information in A "did not pass through". A classic example for this is the *town gossip*. The story we are told is usually more interesting (it's more surprising, than the source) than the original/true version, but important parts of the truth (the original information content) are usually lost in the transmission… So the question arises: where the additional information is coming from? The answer: from the channel.

[6] The independence of two sources can be deduced from the problem's description: e.g. the evening prayer of the imam in Istanbul (source A) and a traffic light with faulty controlling sequences in Veszprém (source B) can be considered to be independent.

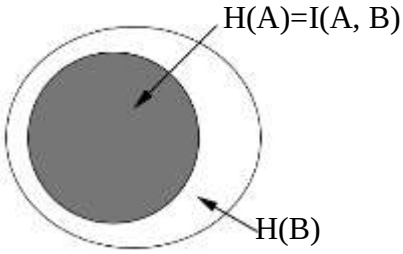
Example 1.1

Some basic examples for information transmission. We suppose that the information goes from A to B. The mutual information is greyed in. Examples 1. and 3. have discrete, 2., 4. and 5. have continuous values.

1. The case of the telegraphist with Parkinson's disease, whose hand is shaking (no information is lost, but additional symbols are added that are typical to the telegraphist, not to the message):

Source A: the message given by the captain, in Morse code

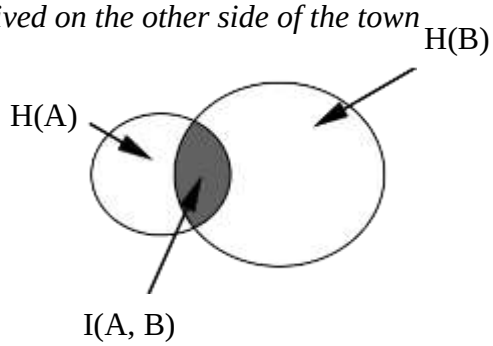
Source B: the message received on the other ship



2. Town gossip (information is lost, and new components are added):

Source A: the original news

Source B: the news received on the other side of the town

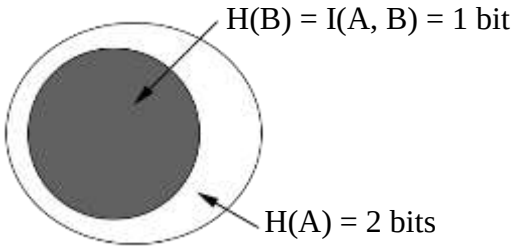


H(A) < H(B)

3. 2+1 parity bit (completely defined source, some information is lost):

Source A: A = {a0, a1, a2, a3} = {0.25, 0.25, 0.25, 0.25}

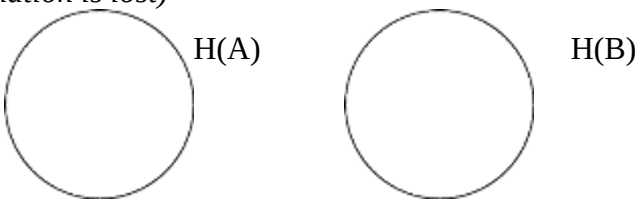
Source B: B = {b0, b1}. B *emits* b0 *if* A *produces* a0 *or* a2, *otherwise it sends* b1.



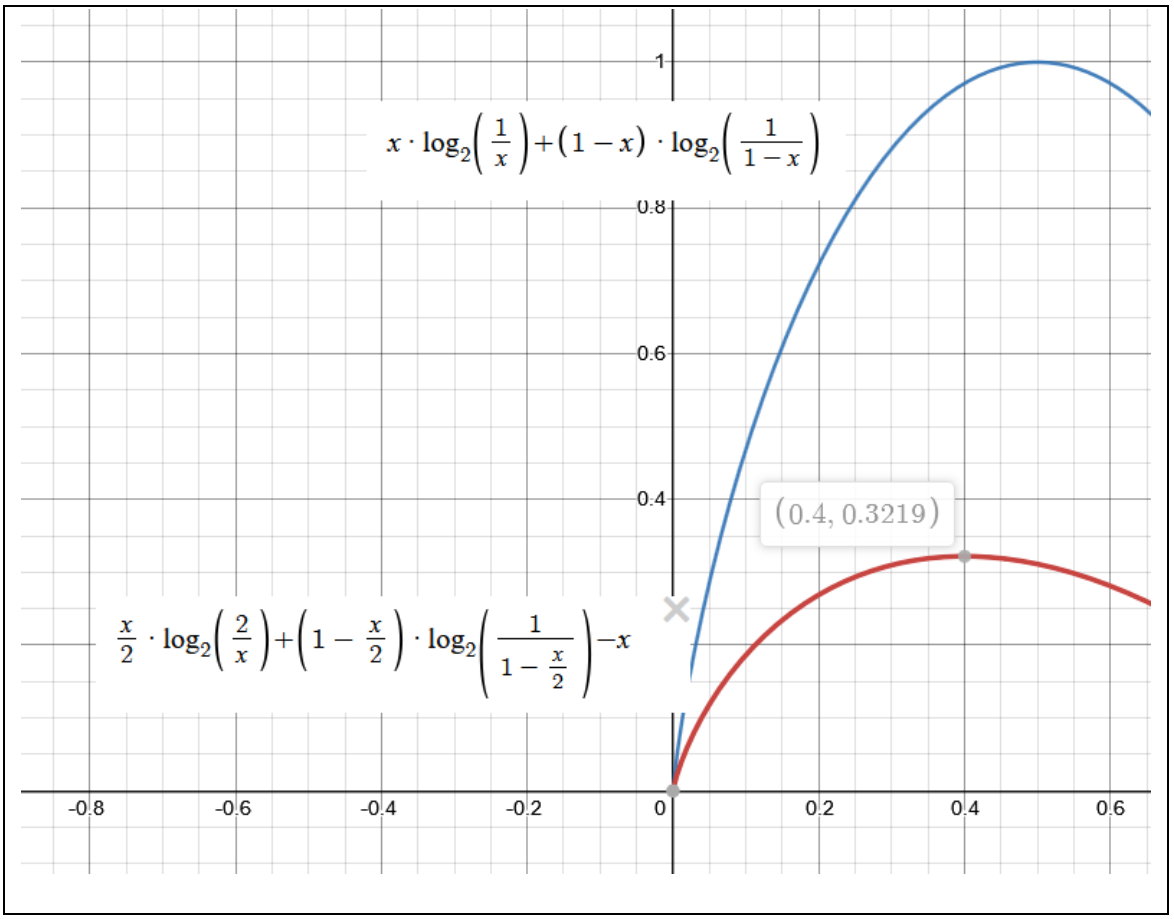
4. Perfect translation (completely defined source, no information loss):
Source A: this syllabus in Hungarian
Source B: this syllabus in English

H(A) = I(A, B) = H(B)

5. Independent sources:
Source A: the prayer of an imam in Istanbul
Source B: a traffic light with faulty controlling sequences in Veszprém
(*independent sources – if they describe the same process, but at different time moments, then it means all information is lost)*



6. The information content of the fast antigen (AG) Covid test:
We know that:
--The AG test can detect the Covid virus in 50% of the cases.
--If the antigen test is positive, then the patient is for sure Covid positive (100%)
So if the probability of the patient being Covid-positive is p, then we have C = {p, 1-p} *as the source that we want to predict based on the AG result.*
How much information can we gain from the AG about C, in the function of p?
AG = {p/2, 1-p/2}
I(C, AG) = H(AG) - H(AG|C). *The maximum is about 0.32 bit, when p=0.4 (see below).*
In other words, about ½..1/3 of the information content of C is accessible by observing AG, depending on p, for p < ½.


$x \cdot \log_2\left(\frac{1}{x}\right)+(1-x) \cdot \log_2\left(\frac{1}{1-x}\right)$
$\frac{x}{2} \cdot \log_2\left(\frac{2}{x}\right)+\left(1-\frac{x}{2}\right) \cdot \log_2\left(\frac{1}{1-\frac{x}{2}}\right)-x$
(0.4, 0.3219)
1
0.8
0.6
0.4
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6

2. Source Coding

2.1. The Basics

The design of encoders and decoders – such as in Figure 1.1 – is performed in two steps (both in theory and in practice) in order to accomplish the two goals mentioned before. The purpose of **source coding** is to improve the efficiency of the channel usage, to **compress** the content of the source without any loss of information. Transmitting information through noisy channels (with as less distortion as possible) is the field of **channel coding**. As a result, when source coding algorithms are mentioned, it is presumed that the channels have **no noise**. Thus Figure 1.1 is modified accordingly and becomes Figure 2.1.

In most practical situations **binary** channels are used – which means only "0" and "1" symbols are available for storing and transmitting – so the following part discusses coding methods for such binary channels. It is also presumed that only one binary symbol can be transmitted at a time, and that the **cost** of transmission is proportional to the amount of symbols sent (i.e. to the number of "channel uses"). In the process of encoding a **code word** (a string of "0" and "1" symbols) is assigned to every source symbol. This assignment (or mapping) is referred to as **code**, for short.

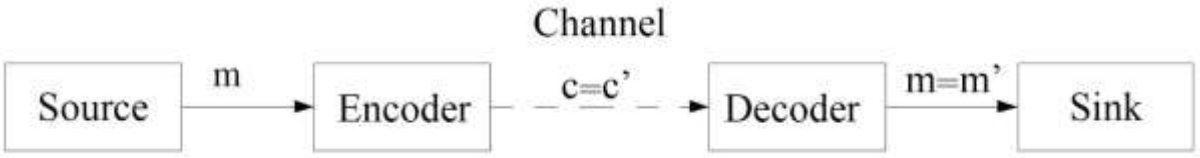


Figure 2.1: *Basic model of source coding*

The **main idea** behind source coding is that shorter code words are used for frequent source symbols and the longer ones are assigned to less frequent symbols. With this we can decrease the cost paid for the channel usage (for transmission sessions "long enough"). A code that uses code words of different lengths is called a **variable-length code**. For such codes we define L(A), **the average code word length** as

$$L(A) = \sum_i p_i l_i$$

where l_i is the length of the code word for the i-th symbol, and p_i is its probability. Sending one code word through the channel has an average cost of L(A), so our task will be to minimize this L(A).

A code is said to be **decodable** if a unique source symbol sequence can be found for every code word sequence generated. This means we cannot have any doubts when decoding. A sufficient[7] condition for a decodable code is that it is a **prefix** code, i.e.

- for every source symbol there is a unique code word, and
- no code word is a prefix of any other code word.

Example 2.1
Coding a source of 3 symbols

Symbol	*Prefix code words*	*Non-prefix code words*
"a"	"0"	"1"
"b"	"10"	"10"
"c"	"11"	"11"

[7] It is not a necessary condition, e.g. *postfix* codes are also decodable.

When using a non-prefix code, the decoder at the recipient's side is in trouble if it receives a symbol sequence such as "111111". It is caused by the code not being decodable.

It is sufficient because the boundaries of the code words can be specified clearly by knowing every valid code word at the recipient's side, and the original source symbol sequence can be decoded as they are all distinct.

A code is **undecodable** if decoding a message is ambiguous, i.e. it is not possible to clearly separate the code words of the message. In some cases, however, this property can be found to be useful: such an example can be found in the Hungarian history, in the plot against queen Gertrudis during the 13th century (a.k.a. the story of Bánk bán). When the plotting nobles have asked János Merániai, the archbishop of Esztergom, for his opinion and approval for their plans, they received a response message containing no punctuation:

"REGINAM OCCIDERE NOLITE TIMERE BONUM EST SI OMNES CONSENTIUNT EGO NON CONTRADICO"

This can be (roughly) translated as:

FEAR NOT TO KILL THE QUEEN IF OTHERS CONSENT I DO NOT OPPOSE

As a result, this message can be read in two ways, having two totally different and contradicting meanings. If we separate it into 3 "code words" we get a consent:

Fear not to kill the Queen. | *If others consent,* | *I do not oppose.*

But if we separate it into 5 parts, then it reads:

(have) *Fear.* | *Not to kill the Queen.* | *If others consent,* | *I do not.* | (I) *oppose.*

This way, if the plot had been successful, he could have appeared to be on their side, but as the plot has failed (the queen was killed, but the king returning to the country executed most of the rebels), he was able to state that he had no intent of helping them, quite the contrary.

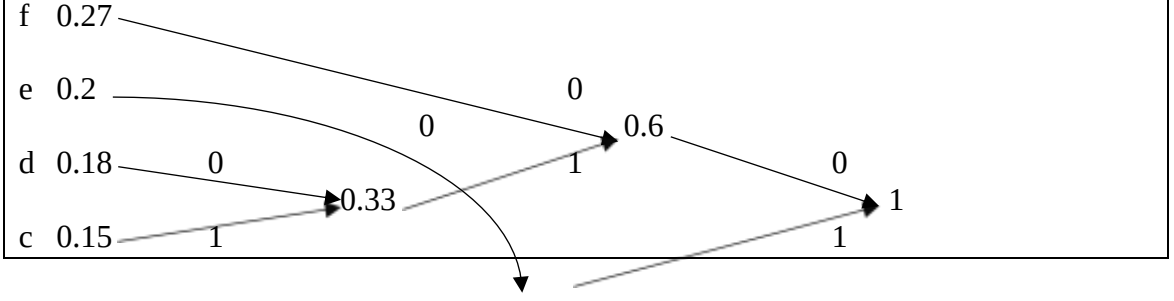
2.2. Huffman Coding

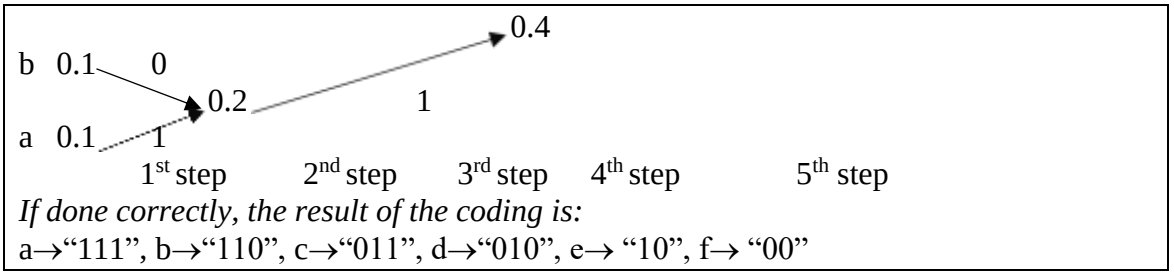
The Huffman code is a prefix, variable-length code originating from 1952. To build such a code we need to know (e.g. from statistical measurements) the probabilities of the source symbols. The code words are constructed in the following procedure:

- Sort the symbols based on their probabilities into decreasing order.
- Combine the two lowest probability (p_1, p_2) symbols into a new "compound" symbol, which has the probability $p_3 = p_1 + p_2$.

We continue this until only one symbol is left with a probability of 1. Then we have a binary tree for which each branching is where we combined two symbols, and its leaves are the original symbols. Code words are then assigned by reading the tree from right to left, adding a "1" to the end of the code word if we go "down", a "0" for going "up".

Example 2.2
Source of five symbols A = {a, b, c, d, e, f} = {0.1, 0.1, 0.15, 0.18, 0.2, 0.27}





If done correctly, the result of the coding is:

a→"111", b→"110", c→"011", d→"010", e→ "10", f→ "00"

It is easy to see that – because of the "binary tree" building rule – the resulting code is prefix, so it is also decodable.

In **hardware** implementations of Huffman coders (e.g. telefax), the encoder stores both the code words and the binary numbers representing their lengths in a ROM (e.g. on the first 10 bits we store the length, and on the next 22 bits the code word itself). The code words are sent to the channel by a shift register managed by a controller, based on the length of the current word. In Figure 2.2 an outline of one possible implementation is shown.

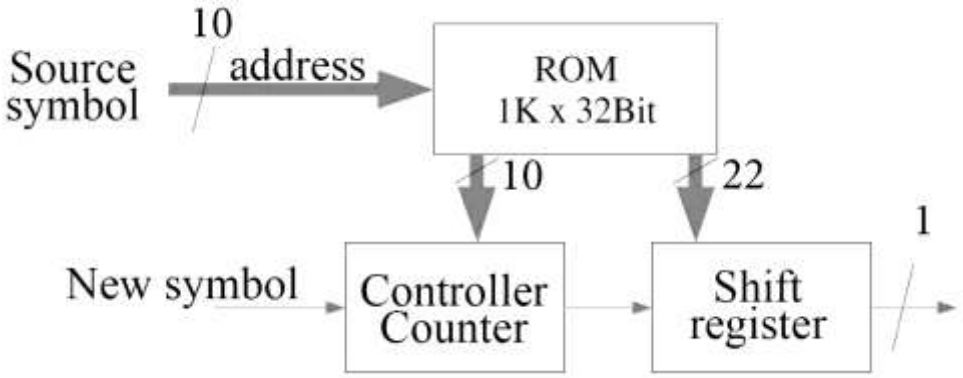


Figure 2.2: *Hardware outline of a Huffman coder*

The decoder uses the prefix property: if the received segment fits only one valid code word, then we have reached the end of the word. This can be achieved using an associative memory. One specific practical application of Huffman coding is the "traditional" (Group 3) black-and-white telefax standard.

2.3. Source Coding Theorems

Shannon's first theorem, also known as the **source coding theorem** (or noiseless-coding theorem) states the theoretical possibilities and limits of source coding.

Let A be a stationary discrete memoryless source, and to every a_i symbol assign a binary code word of length l_i. Then there *exists* a decodable source code such that

$$L(A) < H(A) + 1$$

but there *exists none* that satisfies

$$L(A) < H(A)$$

Hence,

$$H(A) \leq L(A) < H(A) + 1$$

can be achieved, where the left hand side is the "negative", the right hand side is the "positive" statement of the theorem. We can notice that source entropy serves as a basic lower bound on source compression, but it can be approached within 1-bit distance with proper coding. (The previously mentioned Huffman coding has this property.) The proof of this statement and of the theorem can be found in corresponding literature.

From Shannon's theorem we can define the **efficiency** *h* of a code as

$$h = H(A)/L(A)\ [\%]$$

Efficiency varies between 0 and 100%, in ideal cases it can reach 100%. It can be noticed that for Huffman coding such a case is when all symbol probabilities are a negative power of 2. A code of 100% efficiency is called an **optimal code**.

For sources that contain only a few symbols or that have an uneven source distribution, even the average code word length *H(A)* + 1 might prove to be highly inefficient, e.g. for A = {0.001, 0.999}, $h \approx 0.011 / 1 = 1.1\%$. In such situations we can take *N* consecutive symbols and unite them into a single symbol, extending the source *N* times. The number of symbols in this **extended source** will be $|A_{ext}| = |A|^N$ and the probabilities for these symbols will be the products of the original symbol probabilities (e.g. $P(a_1, a_3, a_1) = p_1^2 \cdot p_3$ etc.) for memoryless sources. It can be shown that the entropy of an *N*-fold extended memoryless source is N times the source entropy[8]. If we apply the source coding theorem on the extended source and then divide by block size *N*, then it can be noticed that the theoretical limit can be approached to any arbitrary degree of closeness by increasing the value of *N* due to source extension:

$$H(A_{ext}) \leq L(A_{ext}) < H(A_{ext}) + 1$$

$$N \cdot H(A_{orig}) \leq L(A_{ext}) < N \cdot H(A_{orig}) + 1$$

$$H(A_{orig}) \leq L(A_{orig}) < H(A_{orig}) + \frac{1}{N}$$

where $H(A_{orig})$ is the entropy and $L(A_{orig})$ is the average code word length for the original source. This result is the **source extension theorem**.

Example 2.3
Given the A = {a_0, a_1} = {0.2, 0.8} *source, H(A) = 0.721 bits.*
Using two code words of length 1 i.e. "0" and "1":
L(A) = 1, **h = 72.1**%.
Using a Huffman-code on the twofold extended source:
e.g. P(a_0 a_0) = 0.04, P(a_0 a_1) = 0.16 etc.
L(A×A) = 1.56, H(A×A) = 1.44 bits, so **h = 92.4**%

A drawback of the source extension method is that the cardinality of the source grows exponentially (which results in an expensive and slow hardware), moreover, in cases when the sources are not memoryless[9], it is troublesome to create the source distribution statistics. So when we extend a source or if we do not have full knowledge of the source distribution, it might be necessary to observe the source during runtime (when encoding) and modify the basic source probabilities used by the Huffman code accordingly.

It can be shown that Huffman coding is not sensitive to minor errors in source probabilities, i.e. using estimated values will not result in a considerable loss of code efficiency, compared to using the accurate probabilities.

2.4. Lempel-Ziv Coding

Lempel-Ziv coding (a.k.a. LZ code or dictionary code) uses a completely different approach from Huffman coding. It serves as a basis for numerous **compression** software (e.g. UNIX compress). Its greatest advantage is that it requires no preliminary knowledge of the source

[8] Just like in the case of H(A, B) = H(A) + H(B) for independent sources A, B. If A is memoryless, then it means that two consecutive symbols (that form a composite symbol) are independent: H(A×A) = H(A) + H(A) = 2H(A).
[9] e.g. natural language texts

distribution. It only needs to know the number of source symbols used. It can be shown that for *long enough messages* the average code word length approaches the source entropy, i.e. the code is optimal.

When encoding, LZ coding constructs a linked list structure that contains trees rooted in the initial symbols (forming a "forest"). This forest is the dictionary referred to by the code words. The main steps of the algorithm:

1. Initialize the dictionary, pointer *n* and *m* address registers
   - The columns of the dictionary are the address, the pointer and the symbol. At start it has |A| + 1 rows, the content of the *i*-th row is (i, 0, a_{i-1}). The 0^{th} row is (0, 0, nil).
   - At start n = 0, m = |A| + 1, i.e. the address register points to the dictionary's end.
2. Read the next symbol *x*, exit if no such symbol exists
3. *If* (n, x) is already an element of the dictionary at any address,
   then n = address(n, x)
   else SEND(n)
   store (n, x) at address m
   m = m+1
   n = address(0, x)
4. Return to step 2.

Of course, instead of SEND, the command STORE can also be used, if that is our purpose. When the end of the symbol sequence is reached, the content of register *n* has to be sent too.

It can be seen that the algorithm tries to find a subsequence in the incoming symbol sequence that is already stored in the dictionary. If an incoming symbol does not match the end of the previously found substring or the string ends, only then the address of the end of the substring is sent (which, having the dictionary, will identify the string). Then the pointer is set to the root: n = address(0, x). So next time, when the same symbols sequence appears (e.g. it is a frequent expression in a text) the algorithm will be able to match it for *one more symbol longer* without extending the dictionary or sending additional information. Given an encoding process long enough, the whole content of The Tragedy of Man[10] could be stored with a single *n*.

How can we describe this algorithm using the notions of Section 2.1? As it generates only one code word from an arbitrary long input symbol sequence, we can say that it extends the source to an arbitrary level and the missing source distribution is resolved by the adaptive modification applied when encoding.

Decoding

From the received code word sequence (address/pointer sequence) and the known source alphabet it is possible to reconstruct both the dictionary and the encoded symbol sequence.

1. Initialize the dictionary, pointer *n* and *m* address registers
   - The columns of the dictionary are the address, the pointer and the symbol. At start it has |A| + 1 rows, the content of the *i*-th row is (i, 0, a_{i-1}). The 0^{th} row is (0, 0, nil).
   - At start n = 0, m = |A| + 1, i.e. the address register points to the dictionary's end.
2. Read the next pointer *n*, exit if no such symbol exists
3. Identify the root symbol (a_x) by a backward search through the pointers stored in the dictionary registries.
4. At the address *m* the value of the pointer is set to *n*, leaving the symbol column empty for now.
5. At the m - 1-th position the symbol is set to a_x. This step is skipped when the first pointer is received.

[10] Imre Madách's The Tragedy of Man (Az ember tragédiája)

6. Decode the symbols sequence starting at the address pointed by *n*.
7. m = m + 1
8. Return to step 2.

Example 2.4

LZ code for a source of 3 symbols

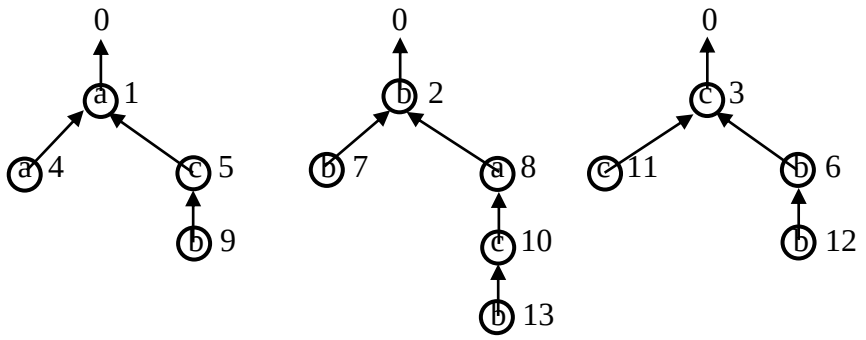
*Let the source be A = {a, b, c} and encode "*a a c b b a c b a c c b b a c b*" sequence.*

The code word sequence sent is 1 1 3 2 2 5 8 3 6 10, n = 2 (the underlined code words are encoding multiple symbols).

The dictionary built when encoding:

Address (m)	*pointer (n)*	*symbol*
0	0	nil
1	0	a
2	0	b
3	0	c
4	1	a
5	1	c
6	3	b
7	2	b
8	2	a
9	5	b
10	8	c
11	3	c
12	6	b
13	10	b, *and n = 2*

The graphical representation of the dictionary's content (the "forest"):



Practical considerations

If addresses are stored on b bits (length of register m is b) then the maximum length of the dictionary is 2^b. So a "too small" b might cause problems during encoding, yet using a big one results in unnecessary high usage of sending/storing capacity, as the message sent/stored is made of the addresses in the dictionary.

2.5. Arithmetic Coding

This coding method is similar to Huffman coding as it requires a priori knowledge of the source distribution, and is similar to LZ coding because it assigns a single code word (a real number) to a sequence of source symbols. Its practical applications include image and video compression.

The basic idea behind the algorithm is that parts of the [0, 1) interval are assigned to the source symbols, based on their probabilities. For example, if A = {a, b, c} and P(A) = {0.4, 0.4, 0.2}, then we get 3 intervals: [0, 0.4), [0.4, 0.8) and [0.8, 1). The "first" symbol (e.g. b) is matched to the proper subinterval (in case of b it is [0.4, 0.8)). Then this subinterval serves as the basis interval and it is divided into more subintervals in the same manner as in the previous step (so we get [0.4, 0.56), [0.56, 0.72) and [0.72, 0.8)). From these three, the second symbol (e.g. c) is matched to its interval ([0.72, 0.8)), and so on. As the number of processed symbols increases, the size of the actual subinterval decreases. Finally, if no symbol is left to be encoded, an arbitrary real number is chosen from the last interval to represent the whole string which is being encoded.

Example 2.5
Let A = {a, b, c} = {0.5, 0.25, 0.25}
Then the subinterval for the symbol sequence "b a a" *is* [0.5, 0.5625), and any number from this interval could be selected.

When selecting a final number from the subintervals, it is advisable to choose one that can be transmitted via binary channels or storage devices at a low cost. So we should choose the "shortest" (least number of digits) binary fraction. From the example above, for [0.72, 0.8), it would be 0.75, which can be transmitted as "0.11" (or by saving the starting "0", just as "11").

Example 2.6
A = {b, a} = {1/3, 2/3}. *Identify the interval boundaries for symbol sequences of length 3!*

Symbol sequence	*interval*	*chosen number*	*binary fraction's length*
bbb	[0, 0.0366)	0.03125	5
bba	[0.0366, 0.11)	0.06125	4
bab	[0.11, 0.183)	0.125	3
baa	[0.183, 0.33)	0.25	2
abb	[0.33, 0.4033)	0.375	3
aba	[0.4033, 0.55)	0.5	1
aab	[0.55, 0.7)	0.625	3
aaa	[0.7, 1)	0.75	2

The example tries to demonstrate that the more probable a symbol sequence is, the wider the assigned interval and shorter the generated code word is.

Naturally, the decoding procedure is then started by identifying the symbol encoded first. It can be noticed that the effect of encoding a symbol is adding a number to the interval bounds, which might cause a change in more than one digits of the chosen binary fraction. This way arithmetic coding distributes the code bits between the symbols, so it is unaffected by the quantizing effect Huffman coding has.

Practical considerations
When decoding, a problem might arise: if symbol sequences of a given length are used every time in encoding, then what guarantees that the resulting variable-length code is decodable? Or if we use code words of the same length instead (i.e. we stop decoding, when a specified binary fraction is reached) then how does the decoder know when to stop? The solution is applying a

special STOP symbol: it has to be treated in the same manner as other elements of the source alphabet, i.e. a probability has to be assigned to it.

Another problem could be caused by the finite numerical precision (arithmetic overflow when calculating interval bounds), that is, we cannot encode a symbol sequence of truly arbitrary length into one code word. The "infinite-precision arithmetic" can be simulated by computer science techniques. Suppose that somehow we know that the first few digits of the interval bounds will not change. For example, if we are between 0.2 and 0.4 – so both are smaller than 0.5 – then the first digit is surely 0. So we can already send this fixed digit, and can continue the calculation with the rest.

2.6. Evaluating Source Coding Methods

The main purpose of the three methods mentioned before was to "**compress**" the sources without any loss, i.e. identifying such binary code words which have their average code length as close to the theoretical minimum (source entropy) as possible. In an optimal case, when this minimum is achieved, every symbol of the compressed message contains 1 bit information on *average*.

The oldest and simplest method is **Huffman coding**. Its advantage is that it can be implemented with simple and fast hardware (code words have to be read from a table) and that by every source extension we get closer to achieving optimal code. A drawback is that for small alphabets or those which have "extreme" source distributions, the generated codes might have low efficiency. It is because after constructing the dictionary, *every time* a given symbol appears, the same code word made of the same *integer* number of binary symbols is sent to the channel (quantizing effect). So the efficiency is not improved as the length of the message is increased. Yet, if the source is memoryless, by source extension the code can approach optimality, but only at the price of increasing the cost and operation time of the encoder used.

As an alternative to Huffman coding, in case of "too small" alphabets, there is **arithmetic coding**, which also uses a priori knowledge of source distribution. Its advantage is that by dividing code bits the quantizing effect is eliminated and it generates an asymptotically optimal code. But a trade-off is its relatively higher complexity.

Finally, we have **LZ-coding** with its completely different approach that needs no knowledge of the source distribution. However, in order to achieve high efficiency, a symbol sequence of adequate length is required.

It is important to note that *there is no* lossless compression algorithm that could *always* encode a binary source's symbol sequence of arbitrary length N by using fewer symbols than N (not even by 1 less, i.e. with at most N-1 binary symbols). This can be easily understood with the help of the pigeonhole principle (see Appendix A2). The statements on code efficiency are only meant in ***general***, as an *average* evaluation.

Example 2.7
In example 2.3 using naïve coding L(A) = 2, with Huffman coding L(A) = 1.7. But if the source emits only a_2 and a_3 for a long time period, then the Huffman coding results in 3 channel uses for a symbol to transmit, while naïve needs only 2.
How is this possible? It is simple: we design codes for general *(probable) cases, not for special and rare symbol sequences.*

As already mentioned, in source coding we suppose that the channel is not affected by noise and that no information is lost during the transmission process (the code is decodable). The main application areas of **lossy** encoding are image, audio and video compression, where the special features of human perception are exploited. (These, however, are not mentioned in this syllabus.)

In case we want to take into account the channel noise, we need to move on to the field of **channel coding**.

II CHANNEL CODING

3. Channel Coding

3.1. The Basics

To model information transmission over a noisy channel we use the following figure:

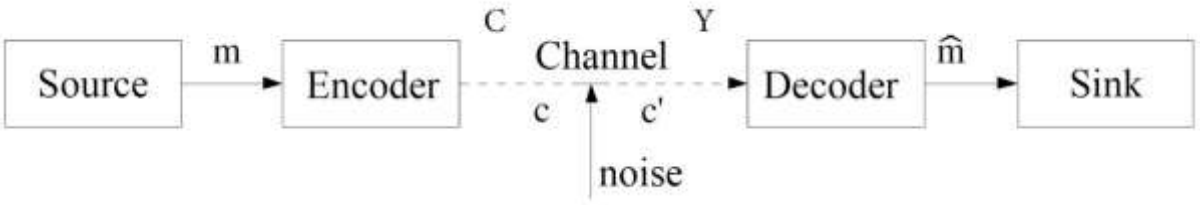


Figure 3.1: *Basic model of information transmission*

In channel coding we suppose that the source and its message m is **already encoded** (compressed). We also suppose that message m is constructed of **binary** symbols. Our task is to transfer this message to the sink (which embeds a source decoder) with as less noise related distortion as possible.

Our basic method to overcome the effects of noise is making m "redundant enough": with **redundancy** we try to make it possible to fix errors with the decoder, i.e. finding the most probable message estimate $\hat{m}$. So in source coding we *reduced* redundancy (in optimal cases and “in general” we even eliminated it), and now we want to *increase* it, based on the properties of the channel and the problem itself. This simple principle has been used with success intuitively in many fields of communication, as an example, consider the code words used in navigation and aviation:

1	UNAONE	A	ALPHA
2	BISSOTWO	B	BRAVO
3	TERRATHREE	C	CHARLIE
4	KARTEFOUR	D	DELTA
5	PANTAFIVE	E	ECHO
6	SOXISIX etc.	F	FOXTROT etc.

The way of increasing redundancy for the most commonly used *block codes* is dividing the continuous input bitstream (consecutive code words of a decodable code) into blocks of length K, and from them computing channel code blocks of length N (N > K).

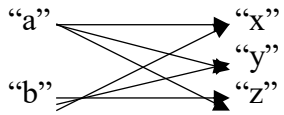
Numbers N and K are called **code parameters**, and a code described by them is said to be a "code of parameters (N, K)” or "(N, K) code", for short.

3.2. Characterizing Channels

We define the **discrete memoryless channel** (DMC), the medium of information transmission, in the following way:

- in every time segment, one input is received at the beginning and one is emitted at the end of the channel (**synchronous behaviour**)
- the input and output symbol sets are not necessarily identical, but both contain a fixed and finite number of elements (**discrete property**)
- if input symbols are statistically independent, then output symbols are also independent (**memoryless source**)

Example 3.1
A channel (transition) graph



The channel has 2 input and 3 output symbols

Let c_i denote the i-th input symbol and y_j the j-th output symbol of the channel. Then, in case of a real channel, we can get the probability of the channel emitting y_j for input c_i (i.e. the conditional probability $p(y_j|c_i)$) by using empirical measurements or with other methods. If we produce $(y_j|c_i)$ for every i and j, then from the information theory point of view we can characterize the channel completely. The probability of receiving y_j in a time instant:

$$p(y_j) = \sum_i p(y_j|c_i)p(c_i)$$

From the $p(y_j|c_i)$ conditional probabilities we can construct the so-called **transition probability matrix P**(Y|C), which contains $p(y_j|c_i)$ in its j-th row and i-th column. The i-th column of P gives us the statistical distribution of the outputs when the input is c_i, thus each column of the matrix must sum up to 1. We can indicate these probabilities on the edges of the channel graph too.

If the channel has no noise, then the input symbol defines uniquely the corresponding output symbol and every $p(y_j|c_i)$ is either 0 or 1. As the amount of noise increases, the probabilities change as well, and it becomes harder and harder to figure out the original input symbol for a given output symbol. The quality of a channel is measured by the average amount of information that can be transferred from the input to the output. If we consider the input and the output as two sources, then this is nothing else but their **mutual information**.

Example 3.2
Let the previous channel’s transition probability matrix be

$$\mathbf{P}(\mathrm{Y}|\mathrm{C}) = \begin{bmatrix} 0.7 & 0.1 \\ 0.2 & 0.2 \\ 0.1 & 0.7 \end{bmatrix}$$

Suppose that P(C) = {0.5, 0.5}. *What is the entropy of the output, of the input and what is their mutual information?*
From the definition: H(C) = 1 bit, H(Y) = 1.51 bit, I(C, Y) = 0.36 bits.
Note that because of $p(y_1|c_0) = p(y_1|c_1)$ *the output symbol “*b*” contains no information about* C*, regardless of* C*’s probability distribution.*

The example shows a case when only 0.36 bits of the initial average information content of 1 bit at the input has passed through to the output, as a result of the noise in the channel. Yet at

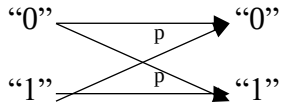
the output we measure an average information content of 1.51 bits (this excess information – like in the case of town gossip – conveys information about the channel itself).

If we consider not just one specific input distribution, we can define the **channel capacity C** as *the maximum average amount of information that can be sent per channel use* for any input distribution, i.e.

$$C = \max_{P(C)} I(C,Y)$$

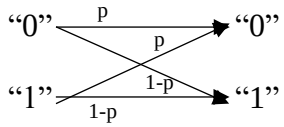
Finding the channel capacity through theoretical methods is a difficult task, however, there exists a numerical algorithm that can provide a result of arbitrary accuracy (this is the Arimoto-Blahut algorithm). Further information can be found in the corresponding literature.

Example 3.3
Some important channels and their capacity
Binary symmetric channel*, BSC: the probability of error is p*



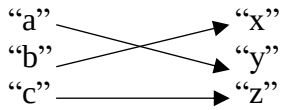
C = 1-H(p 1-p)

0-transmission channel

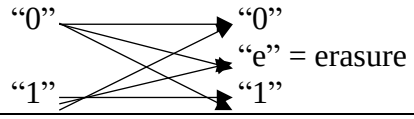


C = 1-H(p, 1-p)

Noiseless channel



Binary erasure channel



We should take a look at the capacity of BSC (see Figure 3.2). We can see that a channel with p = 0 is just as noiseless as one with p = 1. The latter is also often called as the *inverter* channel. Just like for people who *always* give us wrong advice, if we do the opposite of what they say, we can be sure to do the right thing.

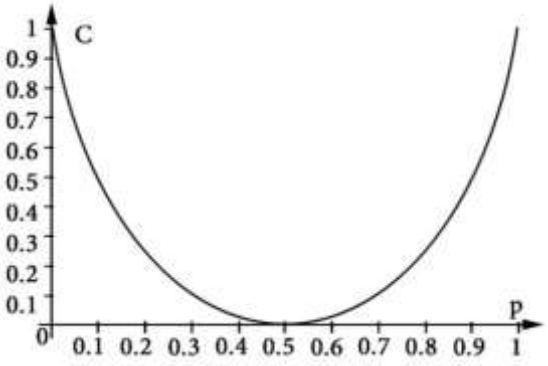


Figure 3.2: *The channel capacity of a binary symmetric channel, as a function of error probability p*

Channel capacity is similar to the width of a road: it depends on technical construction, not on the information transmitted – just like as a road's width is independent of the traffic it has. As

we defined capacity with the maximal mutual information, it is clear that in *general*, information of higher amounts than the capacity cannot be transmitted through the channel. However, for some specific cases this might not be true, e.g. it can happen that an original message – without any channel coding – passes through with no distortion, despite the presence of noise.

3.3. The Binary Communication Problem

Let us return to our channel model and take a look at the physical properties of the channel. Figure 3.3 shows the inner model of a discrete noisy channel, between the encoder and the decoder.

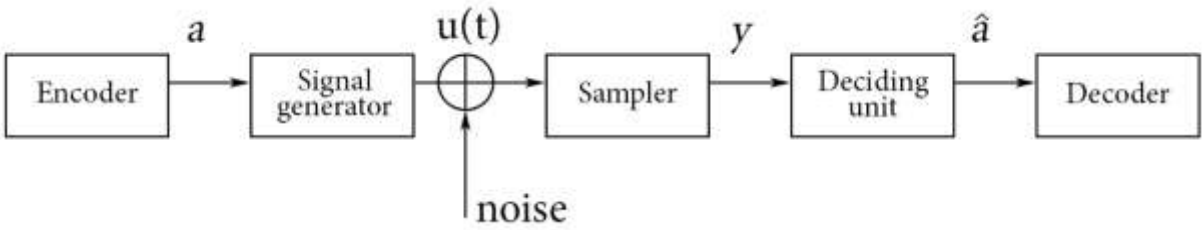


Figure 3.3: *The model of a noisy discrete channel*

In the figure above *a* is one of the channel symbols – "0" or "1" if the channel is binary – and $\hat{a}$ is its estimate at the recipient's side. The communication is time scheduled, the sampler takes a sample from the channel's u(t) physical signal (which is a continuous time value function of voltage) at the end of every t_0 time interval. The signal generator assigns a specific waveform to the channel symbol in the given time segment, based on *a*. For binary channels we need two base signals (a **base signal pair**).
Some base signal pairs:

- On-off keying, OOK. We assign a constant high voltage value to "1" and a low one to "0".
- Phase-shift keying, PSK. We assign a sine signal with 0 starting phase to "1", while "0" gets one with a $\pi/2$ starting phase.
- Frequency-shift keying, FSK. We assign a sine wave signal of lower frequency to "1" and a higher to "0".

The additive noise present in the channel distorts the waveform of the base signals. The decision maker has to identify $\hat{a}$ for a given time segment, based on a given point (e.g. from the middle) of the sample taken.

The Standard Decision Rule
The decision made is based on partitioning the range of the received y, where the A_i regions are identified for the corresponding a_i symbols (the partitioning is the rule itself). The decision rule is proper (for rational reasons), if

$$P(a_i|y) \geq P(a_j|y), \qquad \forall i,j \ \ where\ i \neq j\ and\ y \in A_i$$

The probability $P(a_i|y)$ is called a posteriori probability. Since we want to maximize it with our decision, the decision rule is called **maximum a posteriori** (MAP) or **standard** decision. The probability of a wrong decision:

$$P_{error} = P(y \in A_i, a \neq a_i)$$

It can be proven, that if we partition the range of y based on the standard decision rule, then the probability of false decision is minimal.
The decision problem is of **Bayes-type**, if

- the wrong decisions can be treated together, i.e. for $\hat{a} = a_i, a = a_j, i \neq j$ the cost of error is independent of *i* and *j* and
- there exists and can be given the **a priori** probability distribution *f(a)*.

If at least one of these conditions is not met, then the problem can be solved with general (statistical) hypothesis testing procedures. Such an example is the detection of enemy aircrafts and sounding the alarm: the cost of a false alarm and that of a missed one are not comparable.

The binary communication problem

Our task is to detect a channel symbol encoded via OOK (a_0 = "0" or a_1 = "1") sent over a binary symmetric channel that is affected by noise that has normal distribution. Based on the previously mentioned conditions, it is a Bayesian problem (it is the same, if we receive "1" instead of "0" or if we get "0" instead of "1"). To make our decision, first we modify the expression from the standard decision rule by inserting the conditional probabilities (in order to take into account the discrete distribution of the channel symbols and the continuous valued y):

$$f(a_i|y) = \frac{P(a_i)f(y|a_i)}{f(y)}$$

In this expression $P(a_i)$ is the a priori probability distribution, which is measured at the output of the encoder. $f(y|a_i)$ is the characteristics of the signal generator and the noise, which can be measured or calculated at the output of the receiver, when a continuous transmission of a_i is provided. To get our decision rule we need only to compare the numerators for the $i = 0$ and $i = 1$ cases, as the denominators are the same for every *i*. The intersection of curves $P(a_0)f(y|a_0)$ and $P(a_1)f(y|a_1)$ will give us the decision threshold (see Figure 3.4.) On the side where $P(a_0)f(y|a_0) < P(a_1)f(y|a_1)$ the decision will be "1", while on the other side it will be "0".

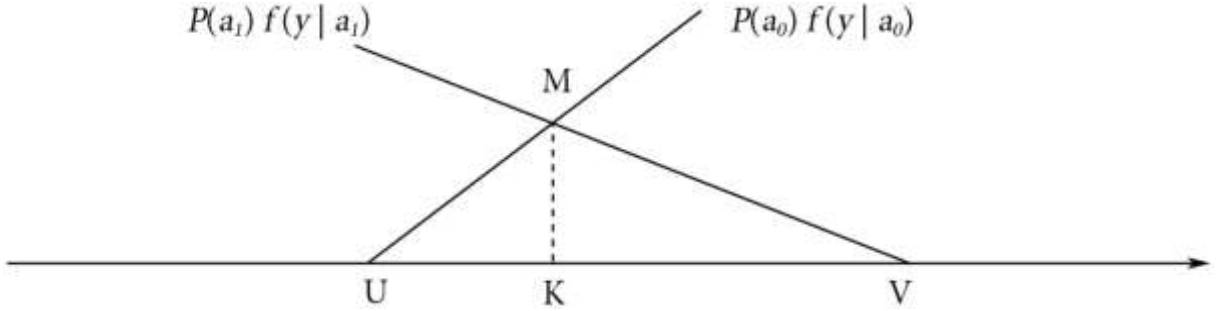


Figure 3.4: *The probability of error*

Limiting error probability, the erasure range

The p error probability of the channel is equal to the sum of the areas of triangles KMV and KMU in Figure 3.4. This error probability is an important parameter in channel design. If in our application this probability is higher then what we can accept, we can introduce an erasure range and thus modify our channel to act as a binary erasure channel (Figure 3.5).

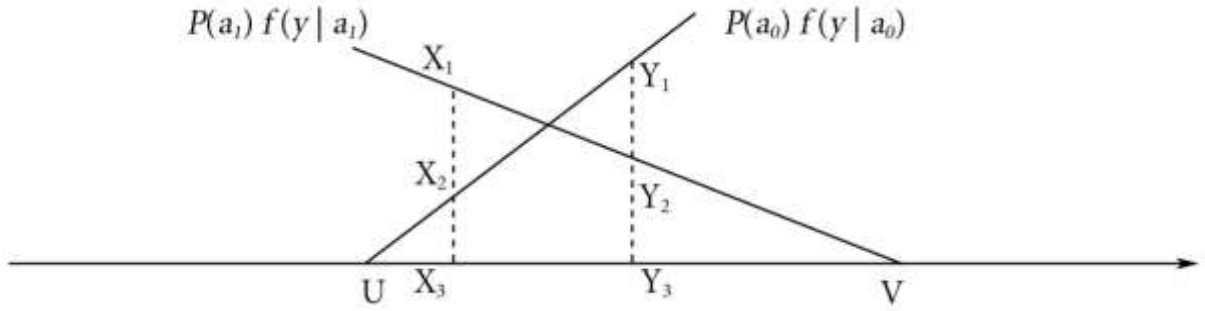


Figure 3.5: *Introducing the erasure range*

With this erasure range (between X_3 and Y_3) the probability of error is only the sum of the areas of triangles X_2X_3U and Y_2Y_3V. The deciding unit will detect an erasure symbol if y is between X_3 and Y_3. In such cases we need to resend the lost symbol (but this means extra cost due to channel reuse) or by using an error-correcting code we can fix the error on the recipient's side, if the number of erasures is less than the code length (these concepts will be detailed later on). The sum of the areas of trapezoids $X_1Y_2Y_3X_3$ and $X_2Y_1Y_3X_3$ is the probability of erasure error, which is also an important aspect in channel design.

The question is if we can find a method that **achieves maximum capacity usage** and provides **lossless** information transmission in general.

3.4. The Channel-Coding Theorem

The answer to the previous question is given by **Shannon's second Theorem** (a.k.a. Shannon's main theorem or the channel-coding theorem):

> Let us introduce redundant symbols in order to change a binary source's symbol blocks of length K to be of length N, and transmit the blocks through a noisy channel with capacity C. Then,
>
> - Then in case of K/N<C there *exists* an extension function (i.e. a channel encoding and decoding algorithm) where given a K large enough, the probability of block decoding error is smaller than any ε>0.
> - And in case of K/N>C **there is no** code that could achieve an arbitrary block decoding error probability.

What is a **block decoding error**? If the block size is increased due to redundant symbols (e.g. repeating the code words), then these "bigger" blocks are sent via the channel. This channel has a probability that it makes an error during the transmission of any "0" or "1" symbol (the output is different from what it should be). With the help of redundancy, we have a chance to recognize this error at the recipient's side, and to correct this error. Then $\hat{m} = m$. However, if we fail to notice the presence of the error, e.g. because the distortion moves the input into another valid code word, then we have no way of fixing this. And even if we notice and try to fix an error, it might happen that we end up with a valid code word that is different from the original. If **any of these** happen, then it is considered to be a block decoding error at the recipient's side.

Besides the theorem's **positive** statement

- it does not provide the channel coding method to use,
- decreasing ε results in a rapidly increasing block size K,
- lossless and errorless decoding is guaranteed only in general (not for specific cases)

The **negative** statement can be interpreted as "If we want to decrease the probability of block decoding errors, then we need to decrease the "density" of the information to be sent: by introducing redundant symbols in the blocks, we have to "dilute" it until it is below the channel capacity". And in case we *cannot* limit the presence of block decoding errors, then *we have no way of ensuring the usability of our communication system, **not even in the general case***.

Entropy rate

We introduce entropy rate R (a.k.a. code rate) as

$$R = \lim_{n\to\infty} \frac{H(c_0, c_1, \dots, c_{n-1})}{n}$$

where $H(c_0, c_1, \dots, c_{n-1})$ is the entropy of the n-times extended source C (on source C we mean the channel symbols sent to the channel, i.e. the output of the channel encoder. The 0, 1, … n-1 indexes represent a time sequence). In accordance with Section 2.3, $R \leq H(C)$, and equality means that C is a memoryless source: the consecutive symbols are statistically independent. Thus R is the measure of *the average information (bits) per channel use*.

We can state the channel coding theorem using the entropy rate, if we substitute K/N with R. Thus the theorem is applicable not just for block codes. In the special case of block codes, every block of length N will convey just the same amount of information as the message block of length K. That is because the extension does not add any extra information. In case of an optimally compressed source, the message block conveys at most K bit information. As this situation is most vulnerable when transmitting through a noisy channel, in channel coding we

suppose that at the encoder's input this K bit is the information content of a block of length K. This way, for the N symbols sent to the channel, the average information is K bit, while for one symbol the average information (entropy rate R) is K/N.

Example 3.4
Channel coding using parity bit 2 + 1, the 3 cases of Example 1.1
Code parameters: (3, 2)
The code: (for odd number of 1s the redundant part is 1, otherwise it is 0)

m_i	c_i
00	000
01	011
10	101
11	110

It can be noticed that the extension did not increase the entropy:

$$H(C) = H(M) = 2 \text{ bits}, \quad R = 2/3 \text{ bits}$$

3.5. Error Detection and Correction

In the decoder we try to detect and correct errors with the help of the redundancy that we introduced into the code word of length N. This redundancy means that of 2^N possible code words only 2^K are valid, because the original encoded message blocks are of length K. Naturally, the decoder knows the valid code words, and when the received word is invalid, it tries to find and forward the "most probable" legal code word. If it selects the correct one, then the **error correction** is successful.

It is also possible that when an invalid code word is found, the decoder does not try to correct it, instead it **notifies** the sender and requests a **resend** until a valid code word is received. However, by retransmitting the same code word over and over again, the efficiency of the channel (entropy rate) will decrease accordingly.

In some cases, it might occur that the channel transforms the input code word into another valid code word. When this happens, there is no way of detecting or correcting this error. This is why $\varepsilon = 0$ is not achievable in the channel coding theorem.

To help us in selecting the "correct" word, we introduce the **Hamming distance** d_H which is the number of bit positions that are different for two code words.

Example 3.5
For code words $c_1 = [011010]$ *and* $c_2 = [010110]$ $d_H(c_1, c_2) = 2$

Hamming distance is a **metric,** as for a and b binary code words of length N:

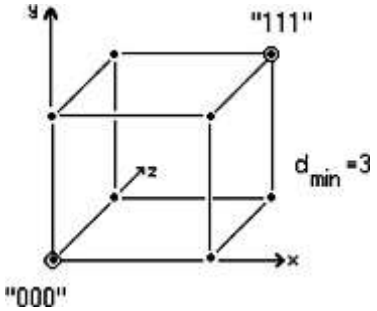
1. if $d_H(a, b) = 0$, then $a = b$
2. $d_H(a, b) = d_H(b, a)$
3. $d_H(a, c) \leq d_H(a, b) + d_H(b, c)$

Suppose that when correcting, the decoder selects the valid code word which is **closest** to the received word, and if there are more than one candidates, it selects from them randomly. It can be shown that this strategy selects the code word which is the most probable from the source statistics aspect (maximum likelihood decision), if the following two conditions hold:

1. The BSC used has an error probability $p < 0.5$
2. At the channel's input every code word (i.e. every message of length K) has the same probability of occurrence.

We can suppose the latter, as our main assumption in channel coding was that the source is already near-optimally encoded (compressed).

Example 3.6
*The "**Hamming cube"** for demonstrating error correction*



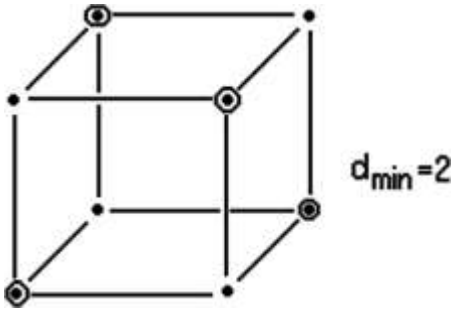
The cube shows a (3, 1) *code, where from the 8 possible code words only 2 are valid. We can correct errors most successfully if these 2 words are on the ends of a space diagonal (circled vertices). For any invalid code word, the closest valid code word can be found uniquely, which will also be the corrected word.*

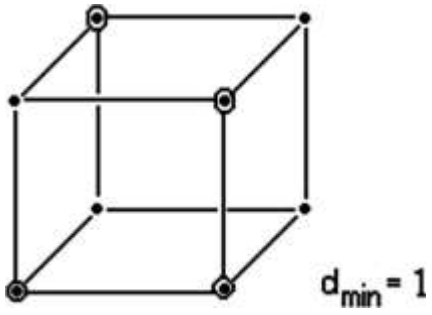
*If we select the code shown above, we get the so-called **(3, 1) repetition code**.*

The Compromise of Correction and Detection

Our ability of correcting and/or detecting errors depends greatly on the **code distance** d_{min}, which is the minimum Hamming distance of the code. It can be noticed (or seen on the Hamming cube) that the bigger this distance, the more errors we can correct.

Example 3.7
For the code set of code words [0 1 1 1], [1 0 0 0], [0 1 1 0], [1 0 1 1] d_{min}=1

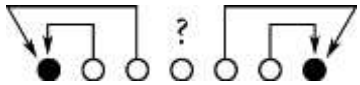




Parity code of Example 3.4 *and a less efficient code*

Denote the number of errors to be corrected as t_c. It is easy to see that in this case every code word should be of at least $2t_c$ distance from all other words ($d_{min} \geq 2t_c + 1$), otherwise our correction will be wrong. Its importance can be seen in Example 3.8, where empty circles represent invalid words and the filled are the valid ones. In the "middle" we do not know which code word should be chosen.

Example 3.8
Error correction when $d_{min} = 6$:



It can also be noticed that for *detection only*, a distance $d_{min} \geq 2t_d + 1$ is required. In Example 3.8 if we correct 2 bit errors, then for the middle one – when the distance is 3 for both words – we only detect the error. So we say we detect 3 errors. Without correction we could detect 5 errors, and 6 errors would result in a valid code word.

In most cases, we correct until a given number of errors is present, beyond that we just detect them. Of course, this is true only when $t_c < t_d$. But if we correct an error, then we cannot also detect it at the same time: for example, if we correct one error in the example above, we can detect only 4 errors. For a code word with 5 errors, we will "repair" it incorrectly into another valid word. So the relations are:

$$\begin{aligned} \textit{correction:} \quad & d_{min} \geq 2t_c + 1 \\ \textit{detection:} \quad & d_{min} \geq t_d + 1 \\ \textit{correction + detection:} \quad & d_{min} \geq t_c + t_d + 1 \textit{ and } d_{min} \geq 2t_c + 1 \end{aligned}$$

Example 3.9
Let $d_{min} = 6$. What are the code correction and detection possibilities?
Detect 5 errors OR
Correct 1 error *and* detect 4 OR
Correct 2 errors *and* detect 3

If our channel is a binary erasure channel, as the one in Example 3.3, and our received code word contains t_{erase} number of **erasure** symbols, i.e. we have erasure errors (but no other), then we can definitely find the original code word, if $d_{min} \geq t_{erase} + 1$ is satisfied, as we know the "positions" of these errors.

Based on which strategy we choose we end up with different probabilities of correcting into the wrong code word, i.e. executing a block decoding error at the recipient's side. Of course, the exact number of errors in a code word have a notable impact on this probability. From the user's point of view this probability is one of the most important properties of our

coding method. It can be seen that the expected value of the **number of errors**, for a BSC of error probability p and with block size N, is N · p, while the **probability of exactly t errors occurring** is:

$$P(t,p,N) = \binom{N}{t} p^t (1-p)^{N-t}$$

In practice, we usually do not correct all errors that the code could correct. If the distance of a code word from a valid code word is too high, we have it resent and we only correct code words that are close to a valid one. As a result, we get a slight decrease in the entropy rate due to the resending, but a rather significant decrease in the block decoding error probability.

3.6. Filling the Code Space

The Singleton Bound

We attach redundant symbols to a block to increase code distance. In the optimal case every additional symbol increases it with one. For example, for the left cube in Example 3.7 the parity bit increased the code distance to 2, while for the right one, the extension has no useful result.

Denote the number of redundant symbols with r=N-K. Then

$$d_{min} \leq r + 1$$

Rearranging the expression and exponentiating both sides with a base 2, we get the **Singleton bound**:

$$2^K \leq 2^{N-d_{min}+1}$$

where on the left side we get the maximal number of possible code words for a given d_{min}. When equality is achieved, i.e. all redundant symbols successfully increased the code distance, we have created a **maximum distance separable** (MDS) code.

Example 3.11
The (N, 1) *repetition code and the* (K+1, K) *parity code are MDS codes.*
Explanation: for the repetition code it is trivial. For the parity code, we only need to show that if the distance between two messages (both of length K) is 1, then the inserted parity bits will also be different, thus the distance will be 2. This is true, because if the distance is 1, it means that one of the words contains exactly one less "1" than the other, so if it has a parity bit "1", then the other gets "0" (and vice-versa).

The Sphere Packing Bound

The basic concept of correction and detection is that we have an N dimensional space, where we have our valid code words (as vectors), and for each we have a sphere with radius t_c (suppose that they do not overlap). Any (invalid) word inside a sphere is corrected into the valid code word in the centre. The way these spheres fill the N dimensional space shows us how "cost-efficient" our code is.

As an error can occur on any bit position, the number of invalid words with t errors for a given valid code word is

$$\binom{N}{t}$$

The number of code words within a sphere of radius t_c (including the valid word):

$$\sum_{i=0}^{t_c} \binom{N}{i}$$

If we sum all points inside the spheres of the possible 2^K valid code words, then this sum cannot exceed the total number of code words. This way we get the **Hamming bound** (a.k.a. **sphere packing bound**):

$$\sum_{i=0}^{t_c} \binom{N}{i} \leq 2^{N-K}$$

When the equality is satisfied – the spheres fill the space completely– then we have a **perfect code**. Such known perfect codes are repetition codes, Hamming codes (see Section 4.4) and the Golay code (further information in the corresponding literature). The corresponding values of N and K:

$t_c = 1$ ($d_{min} = 3$)	$t_c = 2$ ($d_{min} = 5$)	code type
(3, 1)	(5, 1)	repetition code
(7, 4)		Hamming code
(15, 11)		Hamming code
…		

The fact that parameters N and K satisfy the above mentioned equation with equality, does not necessarily mean that there exists a code with t_c code distance that could be constructed. For example, for $t_c = 2$ a (90, 78) code would seem acceptable, however, it can be proved that such code (for which $t_c = 2$ would require $d_{min} \geq 5$) does not exist.

Up to this point, we have introduced two simple channel codes, the parity and the repetition codes. Yet, the small block sizes of repetition codes and the small number of redundant symbols in parity codes mean that they have a limited significance in practical applications.

4. Binary Linear Block Codes

The simplest way to give a general block code is using a table that assigns a unique code word of length N to every (2^K) possible messages. Decoding can be done using the same **code table**. If the received code word is not in this table, then an error has occurred during transmission, so the closest valid code word will be selected[11].

In practical applications, for most cases, in order to achieve our two most important goals – decreasing the probability of block decoding errors and approaching capacity C – the block size has to be increased. However, increasing K results in greater table sizes (2^K) and rapidly increasing encoder/decoder complexity, moreover, finding the closest code word will also be harder.

A great advantage of **linear** block codes is that code generation and correction can be achieved **more easily** and at a lower cost (by using matrix-vector multiplication). Before inspecting linear block codes, we have to introduce the concept of N dimensional vector spaces generated by code words interpreted as vectors.

4.1. Code Words as Vectors

For the mathematical representation of channel symbols we introduce **field GF[2]**[12] with operations *addition*$(+)$ and *multiplication*$(\cdot)$, elements $\{0, 1\}$, and with the following properties:

- *Closure:*
  The two operations "stay" in the set of elements
  $$\forall a, b \in GF[2], \qquad a + b \in GF[2], \qquad a \cdot b \in GF[2]$$
- *Associative property of the operations:*
  $$\forall a, b \in GF[2] \quad \begin{matrix} a + b + c = (a + b) + c = a + (b + c) \\ a \cdot b \cdot c = (a \cdot b) \cdot c = a \cdot (b \cdot c) \end{matrix}$$
- *Identity elements exist:*
  additive identity element: $a + 0 = 0 + a = a$
  multiplicative identity element: $a \cdot 1 = 1 \cdot a = a$
- *Additive inverses exist:*
  For any element *a* exists element x such that $a + (x) = 0$. Such an x is written as $-a$.
- *Multiplicative inverses exist:*
  For any not null element *a* exists element *y* such that $a \cdot y = 1$. This is noted as a^{-1}.
- *Addition is commutative:*
  $$a + b = b + a$$
- *Distributive property of the operations:*
  $$a \cdot (b + c) = a \cdot b + a \cdot c$$
  $$(a + b) \cdot c = a \cdot c + b \cdot c$$

It can be shown that these properties hold for field GF[2], if we use addition and multiplication with their "general algebraic properties", except for the case 1 + 1 = 0 (1 is its own additive inverse).

To represent our **code words**, we create **vectors** (N-tuples) from the field's elements:

$$\boldsymbol{a} = [a_0 \;\; a_1 \;\; \ldots \;\; a_{N-1}], \qquad a_i \in GF[2]$$

For these vectors we define two operations:

- Scalar multiplication:

[11] Supposing that we want to fix all errors correctable by the code, i.e. we use no error signalling/resending.
[12] Symbol GF[2] stands for "Galois Field with two elements".

$$c \cdot \boldsymbol{a} = [c \cdot a_0 \;\; c \cdot a_1 \;\; \ldots \;\; c \cdot a_{N-1}], \qquad c \in GF[2]$$

- Vector addition:

$$\boldsymbol{a} + \boldsymbol{b} = [a_0 + b_0 \quad a_1 + b_1 \quad \ldots \quad a_{N-1} + b_{N-1}]$$

For simplifying notations, on code word vectors we will mean line vectors, for example:

$$\boldsymbol{e} = [1\ 0\ 1\ 0\ 0\ 0\ 0]$$

The vectors are denoted by lowercase bold letters, while the matrices generated from them are noted by uppercase bold letters.

Using the two aforementioned operations, from the vectors and elements of GF[2], we generate **vector space** $GF[2]^N$ that has the following properties:

1. Closure: addition "stays" in the set: $\forall \boldsymbol{a}, \boldsymbol{b} \in GF[2]^N$, $\boldsymbol{a} + \boldsymbol{b} \in GF[2]^N$.
2. Addition is commutative.
3. Addition and multiplication are associative.
4. There exists null element **0** such that for any $\boldsymbol{a} \in GF[2]^N$, $\boldsymbol{a} + 0 = \boldsymbol{a}$.
5. For any vector $\boldsymbol{a} \in GF[2]^N$ there is an additive inverse $-\boldsymbol{a}$ such that $\boldsymbol{a} + (-\boldsymbol{a}) = 0$.
6. For any $a \in GF[2]$ and $\boldsymbol{b} \in GF[2]^N$ **,** $a \cdot \boldsymbol{b} \in GF[2]^N$.
7. Scalar multiplication and vector addition are distributive.
8. As the multiplicative identity of GF[2] is 1, $1 \cdot \boldsymbol{a} = \boldsymbol{a}$ for any vector $\boldsymbol{a} \in GF[2]^N$.

4.2. Properties of Linear Block Codes

An (N, K) liner block code is defined as a K-dimensional subspace of the N-dimensional vector space, which is defined by K number of linearly independent basis vectors ($\mathbf{g}_i$). With these we generate the code words. Let the message be:

$$\boldsymbol{m} = [m_0\, m_1 \;\; \ldots \;\; m_{K-1}], \text{ where } m_i \text{ is an element of GF[2].}$$

Then the code word for $\boldsymbol{m}$ is created as the linear combination of the basis vectors and $\boldsymbol{m}$:

$$\boldsymbol{c} = m_0\boldsymbol{g}_0 + m_1\boldsymbol{g}_1 + \cdots + m_{K-1}\boldsymbol{g}_{K-1}$$

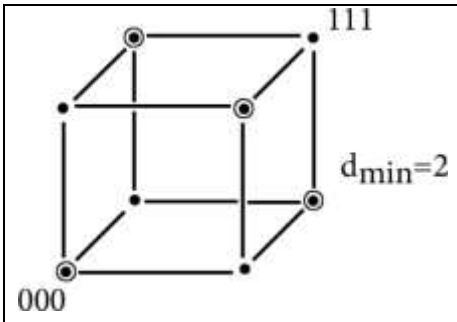
The basis vectors are moved into a **generator matrix G**, where the rows will be the $\mathbf{g}_i$ vectors. This way encoding becomes a matrix-vector multiplication:

$$\boldsymbol{c} = \boldsymbol{mG}$$

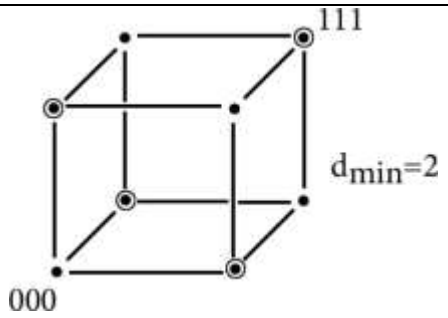
As the rows of **G** form a K-dimensional basis, it is easy to see, that

1. For the 2^K different **m** messages, we get 2^K different **c** code words (if two code words would be the same, then the basis could not be K-dimensional).
2. The sum of two valid code words is a valid code word.
3. Every basis vector is a valid code word.
4. **0** cannot be a basis vector.
5. **0** is a valid code word (this causes that only one of the two Hamming-cubes for (3, 2) codes below is linear).
6. The valid code words are in a K-dimensional subspace of the N-dimensional code space. The **shape** of the subspace defines the code distance, what is important for error correction. Because of this, for *error correction* it is of no importance *which* K number of non **0** linearly independent code word vectors are selected as basis. Of course, the message-code word assignment does depend on it.

Example 4.1
These are two (3, 2) codes that have the same shape and code distance, but only one is linear, as the other does not contain the **0** *as a valid code word.*



linear code



non-linear code

The (3, 1) *repetition code as the generator matrix of a linear block code:*

$$\boldsymbol{G} = [1\ 1\ 1]$$

The K=1 dimensional subspace of this code is the body diagonal that contains [0 0 0] *point.*

Two possible generator matrices for (3, 2) parity code:

$$\boldsymbol{G}_1 = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}, \boldsymbol{G}_2 = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

The subspace of this code can be seen on the left side of the figure above. Because 1 + 1 = 0, it is a "backwards bending" space.

We call a code **systematic** if the original message appears at the end of every generated code word (on its last K positions). Then the r = N – K redundant symbols can be easily identified, and decoding (but not error correcting) is trivial. At the end of the generator matrix of such systematic codes the unit matrix $E_{K\times K}$ is present.

As the rows of **G** are linearly independent, by using row-column transformations any **G** can be transformed into a systematic form. This means that the code words for the messages will change, but the **shape** of the subspace, the code word set and code distance will not.

Example 4.2
The (3, 1) *repetition code for the* ***G*** *matrix above and the* $\boldsymbol{G}_1$ *matrix of* (3, 2) *parity code define systematic codes.*

We define the **Hamming weight** of a vector as the number of nonzero elements in it. As **0** is always a valid code word in linear block codes, and as the sum of two valid code words is also valid, it can be shown (indirectly) for linear block codes, that the weight of the code word with lowest Hamming weight is the same as the code's code distance, i.e.:

$$d_{min} = \min\left(w_H(\boldsymbol{c}_i)\right), \boldsymbol{c}_i \neq \boldsymbol{0}$$

Knowing this, we can get the code distance of a linear code with given generator matrix simply by finding the lowest weight nonzero code word (after generating all of the code words), whose weight will be d_{min}. (a.k.a. **the code distance theorem**)

Example 4.3
If we list the four valid code words of the previous (3, 2) parity code from Example 4.1 (the rows of matrices $\boldsymbol{G}_1$, $\boldsymbol{G}_2$ *and the* ***0****), we can notice that all except the* ***0*** *have 2 ones. Thus the code distance* $d_{min} = 2$.

Example 4.4
Consider the following code word set:
$\mathbf{c}_0$ = [0 0 0], $\mathbf{c}_1$ = [1 0 0], $\mathbf{c}_2$ = [1 1 1], $\mathbf{c}_3$ = [0 1 1]
What are the code parameters? Is the code linear? What its code distance is?

Based on the number and length of code words, the code parameters are (3, 2). The code is linear, as it contains the ***0*** *code word and the sum of any two code words is also valid. As the weight of* $\mathbf{c}_1$ *is 1, the code distance* $d_{min} = 1$*. This means that with this code we cannot even detect a single error. Not all linear block codes make sense.*

4.3. The Parity-Check Theorem

From the previous section, you might already have a grasp on how code word generation in encoders can be efficiently implemented with simple and fast hardware: by using **matrix-vector multiplication**. Moreover, we only need to store matrix **G** and not a table of 2^K rows.

The basis for achieving effective error detection and correction at the recipient's side is the **parity-check theorem**. Before stating this theorem, first we need to create a **parity-check matrix**

$$\boldsymbol{H}^T = \left[\begin{array}{c} \boldsymbol{E}_{r\times r} \\ \hline \boldsymbol{P}_{K\times r} \end{array}\right]$$

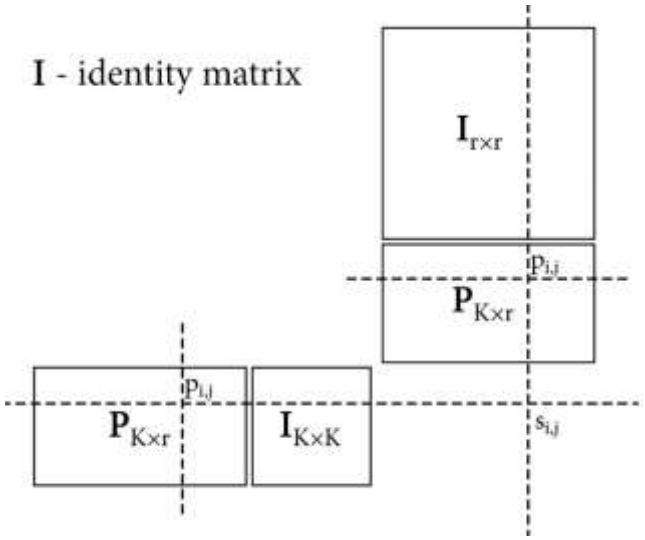
to our systematic generator matrix

$$\boldsymbol{G} = \left[\begin{array}{c|c} \boldsymbol{P}_{K\times r} & \boldsymbol{E}_{K\times K} \end{array}\right]$$

At the recipient's side, every received code word **c**' is multiplied by the matrix $\mathbf{H}^T$ for error detection, and the result is denoted as **syndrome s** = $\mathbf{c}'\mathbf{H}^T$.

Then the **parity-check theorem** states that
1. the syndrome of valid code words is **0**,
2. for invalid code words the syndrome is not **0**,
3. not **0** syndromes uniquely define the errors **correctable by the code**.

The *first* statement can be proved by taking a look at an arbitrary $s_{i,j}$ element of matrix $\mathbf{S} = \mathbf{GH}^T$, in $\mathbf{s} = \mathbf{cH}^T = (\mathbf{mG})\mathbf{H}^T = \mathbf{m}(\mathbf{GH}^T)$:



which is $s_{i,j} = p_{i,j} \cdot 1 + 1 \cdot p_{i,j} = 0$.

It can be noticed, that as a result of the structure of matrix $\mathbf{H}^T$, the rows of **G** (the basis vectors) and the columns of $\mathbf{H}^T$ are orthogonal. Thus the syndrome for every code word that is valid (can be written as **c** = **mG),** is **0**.

To demonstrate the *second* statement, we should consider what $\mathbf{cH}^T = \mathbf{0}$ means for a code word **c**: it tells us that **c** is orthogonal to every column vector of $\mathbf{H}^T$, which, at the same time, define an (N – K)-dimensional subspace that is orthogonal to the K dimensional subspace defined by **G** (as stated above). Thus **c** must be a part of the subspace defined by **G**, i.e. it cannot be an invalid code word.

For proving the *third* statement, we need to introduce an **error model**. We model the errors appearing in our channel as an **error vector e** that is added to the code words. For example, if N = 7 and errors occurred in the 1st and 3rd channel symbol positions during transmission, then

$$\boldsymbol{e} = [1\ 0\ 1\ 0\ 0\ 0\ 0]$$

As both the error vector and the original code word are vectors in the N dimensional space, for which the distributive property of scalar multiplication holds, during the parity-check, the syndrome is defined by *only the error vector* $\boldsymbol{e}$:

$$\boldsymbol{s} = \boldsymbol{c}'\boldsymbol{H}^T = (\boldsymbol{c} + \boldsymbol{e})\boldsymbol{H}^T = \boldsymbol{c}\boldsymbol{H}^T + \boldsymbol{e}\boldsymbol{H}^T = \mathbf{0} + \boldsymbol{e}\boldsymbol{H}^T$$

We will use this important property later on, in the practical implementation.

To prove the 3rd statement, we use an indirect approach: suppose there exist two different error vectors that produce the same syndrome (i.e. they can be used for correction). We only need to show that at least one of them is not correctable, i.e. its weight is greater than the number of correctable errors based on the code distance (as the weight of an error vector is the same as the errors caused by it). For this, suppose that $\boldsymbol{e}_1 \boldsymbol{H}^T = \boldsymbol{e}_2 \boldsymbol{H}^T = \boldsymbol{s}$, where $\boldsymbol{s} \neq \mathbf{0}$ and $\boldsymbol{e}_1 \neq \boldsymbol{e}_2$. Then $(\boldsymbol{e}_1 + \boldsymbol{e}_2)\boldsymbol{H}^T = 0$, so $\boldsymbol{e}_1 + \boldsymbol{e}_2$ must be a valid code word. But in this case, as we have seen for the

Hamming weight in Section 4.1, we get that $d_{min} \leq w_H(\boldsymbol{e}_1 + \boldsymbol{e}_2) \leq w_H(\boldsymbol{e}_1) + w_H(\boldsymbol{e}_2)$. (And equality is met only when there are no "1"s at the same positions in e_1 and e_2.)

On the other hand, because $d_{min} \geq 2t_c + 1$, we know that $2t_c + 1 \leq w_H(\boldsymbol{e}_1) + w_H(\boldsymbol{e}_2)$, which means either $\mathbf{e}_1$ or $\mathbf{e}_2$ must have a weight greater than t_c. This results that (at least) one of them is not correctable, which is contradicting to our starting statement. ■

Based on the parity-check theorem, for error correction at the receiver, we only need to generate and store the table of syndromes for correctable error vectors, and when needed, use that to look up the error vector of the current syndrome received. The corrected code word is then computed by adding the error vector and the received code word together. Such a syndrome table is far smaller than one containing all valid code words, what would be needed for a non-linear code.

Example 4.5
The parity-check matrix of the (3, 1) repetition code:

$$\boldsymbol{H}^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

The corresponding syndrome table:

s	***e***
10	*100*
01	*010*
11	*001*

If we use a mixed error correcting and detecting strategy, then by default, the syndrome table does not contain the syndromes for cases when we do not want to fix the received code word. So when a computed syndrome for a received code word is not in the table, we notify the sender of the detection of an error, and request a retransmission of the current word.

4.4. The Hamming Code

Hamming codes are single-error-correcting perfect codes. For constructing such codes, we define the structure of parity-check matrix $\mathbf{H}^T$, then its corresponding **G** can be computed easily by applying the parity-check theorem.

We assemble our $\mathbf{H}^T$ matrix for given code parameters N and K (which satisfy the condition of perfect codes, i.e. $2^{N-K} = 1 + N$) in the following way:

- In the "upper part" of $\mathbf{H}^T$, a unity matrix $I_{(N-K)\times(N-K)}$ is placed.
- In the remaining K rows, we list all binary vectors with N – K number of elements, which have a weight greater than 1, in an arbitrary order. The number of such vectors is $2^{N-K} - (N - K) - 1$ (where –1 is because **0** is not listed), what should be equal to K, and thanks to the code being perfect, it is.

The $\mathbf{H}^T$ and its **G** matrix constructed this way together define a systematic code.

Example 4.6
A possible $\boldsymbol{G} - \boldsymbol{H}^T$ pair for the (7, 4) Hamming code:

$$\boldsymbol{G} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{H}^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

When generating a syndrome, error vectors of weight 1 are matched to the rows of $\mathbf{H}^T$, based on the position of the error: e.g. [0 0 0 0 0 0 1] vector's pair will be [1 1 0] (7^{th} bit error – 7^{th}

row). As all rows of $\mathbf{H}^T$ are distinct and not **0**, all 1 weighted error vectors will also create distinct, not **0** syndromes, so correcting 1 error in the code will be possible.

Example 4.7
Using the Hamming code of Example 4.6 for 1 and 3 errors:
Initial data:
Let be given $\boldsymbol{m}$ = [1 0 0 1]. *Then* $\boldsymbol{c} = \boldsymbol{mG}$ = [0 0 1 1 0 0 1], *the syndrome for an errorless code word is* $\boldsymbol{s} = \boldsymbol{c}'\boldsymbol{H}^T = \boldsymbol{cH}^T = \mathbf{0}$, *the code word estimate* $\hat{\boldsymbol{c}} = \boldsymbol{c}'$ *and the message estimate is*
$\hat{\boldsymbol{m}}$ = [1 0 0 1].
Correcting 1 error:
For given $\boldsymbol{e}_1$ = [0 0 0 0 0 0 1].
Then $\boldsymbol{c}' = \boldsymbol{c} + \boldsymbol{e}_1$ = [0 0 1 1 0 0 0], $\boldsymbol{s} = \boldsymbol{c}'\boldsymbol{H}^T$ = [1 1 0], *which is in the* 7^{th} *row of* $\boldsymbol{H}^T$, *so the error estimate is:* $\hat{\boldsymbol{e}}$ = [0 0 0 0 0 0 1] = $\boldsymbol{e}_1$.
$\hat{\boldsymbol{c}} = \boldsymbol{c}' + \hat{\boldsymbol{e}}$ = [0 0 1 1 0 0 1], *so* $\hat{\boldsymbol{m}}$ = [1 0 0 1] = $\boldsymbol{m}$, *i.e. 1 error is corrected properly.*
Correcting 3 errors:
For given $\boldsymbol{e}_2$ = [1 0 1 1 0 0 0].
Then $\boldsymbol{c}'$ = [1 0 0 0 0 0 1], $\boldsymbol{s}$ = [0 1 0], *which is in the* 2^{nd} *row of* $\boldsymbol{H}^T$.
Thus $\hat{\boldsymbol{e}}$ = [0 1 0 0 0 0 0], $\hat{\boldsymbol{c}}$ = [1 1 0 0 0 0 1], *so* $\hat{\boldsymbol{m}}$ = [0 0 0 1] $\neq \boldsymbol{m}$, *we end up with a block decoding error by correcting it to a wrong (though valid) code word.*

As Example 4.7 shows, it is always possible to identify a row in $\mathbf{H}^T$ from the syndrome, and its corresponding error vector estimate, that is summed to the received invalid code word in order to get the message estimate. Naturally, if not just 1, but more errors occur, a block decoding error will appear, as we will correct it into another valid code word, not to the one actually sent.

In real life applications, however, the importance of Hamming codes is limited, as the requirement of being a perfect code serves as a strict constraint on possible N, K parameters[13].

[13] Interesting fact: by rearranging the columns of $\mathbf{H}^T$ and the rows of **G**, parity-checking can be made simpler, but the result will not be systematic.

5. Cyclic Codes

Cyclic codes are widely used linear block codes. We define them as such linear block codes in which the rotated version $\mathbf{c}^s$ of every valid code word $\mathbf{c}$ is also a valid code word:

$$\boldsymbol{c} = [c_0 \quad c_1 \quad \dots \quad c_{N-2} \quad c_{N-1}], \qquad \boldsymbol{c}^s = [c_{N-1} \quad c_0 \quad c_1 \quad \dots \quad c_{N-2}]$$

In cyclic codes, there can always be found a valid code word of the form $[g_0\, g_1 \dots g_{N-K}\, 0 \dots 0]$, so the generator matrix of these codes can always be written in a band matrix form:

$$\boldsymbol{G} = \begin{bmatrix} g_0 & g_1 & \dots & g_{N-K} & 0 & \dots & \dots & 0 \\ 0 & g_0 & g_1 & & \ddots & & & \vdots \\ 0 & & g_0 & g_1 & & \ddots & & \vdots \\ \vdots & & & \ddots & \ddots & & \ddots & 0 \\ 0 & \dots & \dots & 0 & g_0 & g_1 & \dots & g_{N-K} \end{bmatrix}$$

Example 5.1
An example for a simple (4,2) cyclic code:
$\boldsymbol{c}_0 = [1\ 0\ 1\ 0]$
$\boldsymbol{c}_1 = [0\ 1\ 0\ 1]$
$\boldsymbol{c}_2 = [0\ 0\ 0\ 0]$
$\boldsymbol{c}_3 = [1\ 1\ 1\ 1]$

It can be seen, that as a result of the band matrix structure, the vector components of **c** = **mG** are the convolution sum of series (m_0, m_1, …, m_{K-1}) and (g_0, g_1, …, g_{N-K}). (For that, note that series (g_0, g_1, …, g_{N-K}) also appears vertically in the matrix rows, but in a reversed index order.)

Using this property, we introduce the bit position operator *x*, where the exponent of *x* marks the position of the symbol, and instead of vectors, we will work with polynomials of *x*. Numbers in the exponents are integers, while the coefficients are elements of GF[2]. Since the convolution product of the polynomial coefficient sequences consists of coefficients of polynomial multiplication, the application of the *x* operator turns the matrix-vector multiplication into polynomial multiplication. The motivation for this transform is that binary polynomial multiplication and division can be implemented in hardware even simpler than matrix-vector multiplication. In summary:

$$\boldsymbol{c} = [c_0 \quad c_1 \quad c_2 \quad \dots \quad c_{N-1}] \xrightarrow{x} c(x) = c_0 + c_1 x + c_2 x^2 + \dots + c_{N-1} x^{N-1}$$
$$\boldsymbol{c} = \boldsymbol{mG} \xrightarrow{x} m(x)g(x)$$

where $g(x) = g_0 + g_1 x + g_2 x^2 + \dots + g_{N-K} x^{N-K}$ is the *generator polynomial*. We write the members of the polynomials in an increasing order of the exponent *x*.

It is easy to show that applying the normal arithmetic polynomial division with a remainder

$$x \cdot c(x) = c^s(x) \bmod (1 + x^N)$$

This means that multiplication by *x*, the bit position operator, is equivalent to rotation (cyclical shifting), but only modulo $1 + x^N$. So we introduce the ring $GF(2)[x] \mid 1 + x^N$, which is commutative and has identity elements, and whose elements are polynomials of degrees less than N. In this ring we define two operations, polynomial addition and multiplication (denoted by + and ·) in the usual way. In cases when multiplication would "lead outside" of the ring, the result is divided by the basic polynomial ($1 + x^N$) and the remainder is kept.

Later on, we will utilize the easily proven fact that the remainder for the sum of two polynomials is the sum of the individual remainders.

5.1. Construction of Cyclic Codes

Choose such a *g(x)* that $h(x) \cdot g(x) = 1 + x^N$ is satisfied, and generate the valid code words, based on *c(x) = m(x)g(x)*. Then for the valid code words in the polynomial ring:

$$s(x) = c(x) \cdot h(x) = m(x) \cdot g(x) \cdot h(x) = 0 \bmod x^N + 1$$

Using a reasoning similar to the one is Section 4.3, it can be proved that the statements of the parity-check theorem are also true for the generated cyclic codes. These codes can be directly implemented by using polynomial multiplier circuits (both at the sender and at the receiver) which consist of a shift register and a combination logic containing the polynomial coefficients (for detailed information see Figure 6.1 in the convolutional codes section). The error correction for correctable errors can be executed by using the syndrome table. However, in decoding, it causes some trouble that the code is not systematic, thus the message estimate cannot be computed from the corrected code word.

Systematic Cyclic Codes
To make the code systematic, we start again with $h(x) \cdot g(x) = 1 + x^N$, but the valid code words are generated differently, in the following way:

$$c(x) = x^r m(x) + d(x), \text{ where}$$
$$d(x) = x^r m(x) \mid g(x)$$

As *deg(d(x))* < N – K, the code is indeed systematic, thus its first K positions are given only by $x^r m(x)$, and the last N – K by *d(x)*. At the receiver, the syndrome is given by the remainder modulo *g(x)*:

$$s(x) = c'(x) \mid g(x)$$

Then, for the valid code words

$$s(x) = \left(x^r m(x) + d(x)\right) \mid g(x) = x^r m(x) \mid g(x) + x^r m(x) \mid g(x) = 0$$

as the polynomial coefficients are elements of GF[2] (so 1 + 1 = 0).

The code generated is also cyclic, as it is equivalent to a code created by *g(x)* via polynomial multiplication, as every code word is a multiple of *g(x)*. To see this, introduce *b(x)* in the following way:

$$x^r m(x) = b(x)g(x) + x^r m(x) \mid g(x)$$

Then the systematic code word *c(x)* can be written as

$$c(x) = b(x)g(x) + x^r m(x) \mid g(x) + x^r m(x) \mid g(x) = b(x)g(x)$$

so indeed, every valid code word is a multiple of *g(x)*.

It can also be shown, that condition $h(x) \cdot g(x) = 1 + x^N$ is not just sufficient, but also necessary for the generated code word to be cyclic. Another necessary condition is that in *g(x)*, the coefficients of smallest and highest degree are both 1.

Implementing Systematic Cyclic Codes with Polynomial Divider Circuit
As already mentioned, for the sender's cyclic code generation, and for the receiver's syndrome calculation, the same device – a divider by *g(x)* circuit – is sufficient. The main idea behind the circuit is that for arbitrary *w(x)*:

$$w(x) \mid g(x) = w_0 \mid g(x) + w_1 x \mid g(x) + w_2 x^2 \mid g(x) + \cdots + w_{N-1} x^{N-1} \mid g(x)$$

So the remainders are calculated member wise. The structure of such circuit at the sender's side can be seen in Figure 5.1

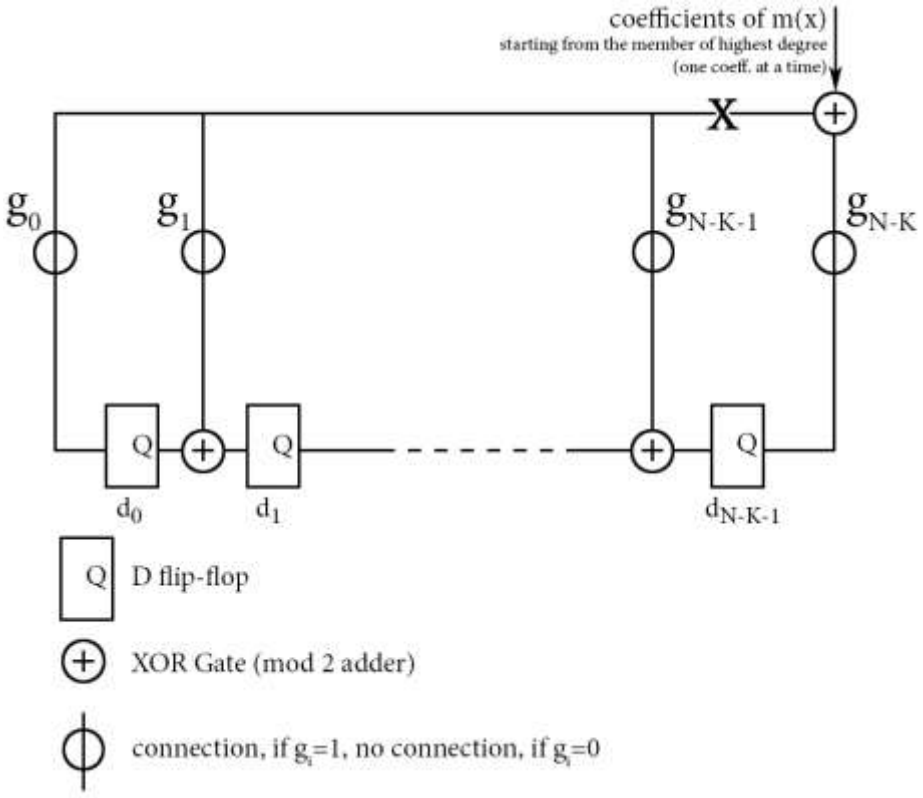


Figure 5.1: *Structure of a binary polynomial divider. The feedback link marked by an X is referred to in Example 5.1*

The content of the D latches at the beginning is 0. After K cycles, the latches store the coefficients (d_0, …, d_{N-K-1}) of the remainder of the division by *g(x)*. To produce a systematic code, the divider circuit is used in two modes:

1. First, for K cycles we load the coefficients of *m(x)*, while at the same time, we send these to the channel too (so the code is systematic). After K "ticks", the remainder of *d(x)* will be in the latches.
2. We disable the feedback marked by the X in Figure 5.1, and instead of the output of the XOR gate, the feedback line gets 0's. Then, for N – K clock cycles, we send the content of the rightmost latch to the channel, that is, we empty the shift register. As a "side effect", the circuit gets back to its initial state, and will be ready for producing the next code word.

The **decoder** executes the division for N cycles with the same circuit[14], then the resulting syndrome is used to address the syndrome table (ROM), in order to identify the error vector estimate, which is then added to the code word received. For proper scheduling, the incoming code word has to be buffered for N cycles in a shift register.

Example 5.1
Generating a cyclic code and parity-checking for N = 7
Factoring the initial polynomial we get:

$$1 + x^7 = (1 + x)(1 + x^2 + x^3)(1 + x + x^3)$$

[14] The circuit also automatically executes multiplication by x^r, so we do not really generate the syndrome in the same way as originally planned. However, it can be easily shown, that the statements of the parity-check theorem still hold, and the syndrome table is usable.

Let

$g(x) = 1 + x^2 + x^3$, *which means K = 4, r = 3*

$m(x) = 1 + x + x^3$

Then

$c(x) = x^3 m(x) + x^3 m(x) \mid g(x) = x^3 + x^4 + x^6 + x^2 = [0\ \ 0\ \ 1\ \ 1\ \ 1\ \ 0\ \ 1]$

Achieving the same with the polynomial divider:

Tick	$d_0 d_1 d_2$	*Point "X"*	*Input*
0	*0 0 0*	*1*	*1*
1	*1 0 1*	*1*	*0*
2	*1 1 1*	*0*	*1*
3	*0 1 1*	*0*	*1*
4	*0 0 1 =* x^2 *is the remainder*		

From this, the code word is: c(x) = $x^2 + x^3 + x^4 + x^6$*, that is checked at the receiver:*

Tick	$d_0 d_1 d_2$	*Point "X"*	*Input*
0	*0 0 0*	*1*	*1*
1	*1 0 1*	*1*	*0*
2	*1 1 1*	*0*	*1*
3	*0 1 1*	*0*	*1*
4	*0 0 1*	*0*	*1*
5	*0 0 0*	*0*	*1*
6	*0 0 0*	*0*	*0*
7	*0 0 0 the remainder is 0 (the code word is valid)*		

By generating all 16 valid code words of the codes it can be seen, that

a) the code is indeed cyclic

b) code distance is 3, so we can correct 1 error

In a similar way, it can be computed that the syndrome for the wrong code word c'(x) = $x^2 + x^3 + x^4$*, affected by the 1 weighted e(x) =* x^6 *error vector, is* x^2*. We get the same syndrome if we compute the remainder for* x^6*, as the syndrome depends only on the error. Note that the previous code is a (7, 4) Hamming code, as it is perfect and can correct 1 error. Hamming codes form a subclass of cyclic codes.*

Example 5.2

(K+1, K) parity code, as a linear block code, and as a cyclic code

Linear block code:

The redundant part is 1 bit, which is the mod 2 sum of the digits of the message:

$$\boldsymbol{G} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & 0 & 1 & 0 & 0 & & \\ 1 & 0 & 0 & 1 & 0 & \dots & 0 \\ \vdots & & & & 1 & & \\ & & & & & \ddots & \\ 0 & 0 & & \dots & & 1 & 1 \end{bmatrix}$$

Implementation: XOR gate with K inputs, addition in parallel time

Cyclic code:

By using addition for the rows of the previous ***G*** *matrix, a cyclic code is created easily:*

$$\boldsymbol{G} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 1 & 0 & 0 & & \\ 0 & 0 & 1 & 1 & 0 & \dots & 0 \\ \vdots & & & 1 & 1 & & \\ & & & & \ddots & \ddots & \\ 0 & 0 & & \dots & & 1 & 1 \end{bmatrix}$$

From this g(x) = 1 + x. Indeed, this g(x) is a divisor of 1 + x^N for every N.
Implementation: The polynomial divider, based on Figure 5.1, is a single D-FF with feedback from a XOR gate. This executes the same addition operation in bit serial mode (in K ticks, by storing the partial result) as a XOR gate with K inputs in transient time.

Cyclic Codes Used in Practice

Only a part of cyclic codes can be designed (in a way not detailed here) for given N, K and d_{min} parameters. The most important are:

- Hamming codes
- BCH codes (Bose–Chaudhuri–Hocquenghem). As the number of correctable errors increase, so does the complexity of encoders and decoders. For example, the code generated by the polynomial 3551[15] has parameters (31, 21), and it can correct 2 errors.

CRC Codes

Cyclic Redundancy Check codes are able to only detect errors, therefore, they are typically used together with some kind of error correcting coding technology. CRC codes are special, shortened cyclic codes, that create short (e.g. r = 16) check sums for big block sizes (e.g. K = 32000). If anything changes in the content of the block or in the check sum, it has a great probability that the CRC sum calculated at the receiver will not match the one at the end of the block sent. CRC codes and the syndrome can be generated by polynomial division just like cyclic codes. The main steps of using CRC:

1. After source encoding, but before error correcting channel encoding, we generate CRC for each blocks and place them to their ends.
2. Divide the message into blocks again, based on the size of the error correcting code. Generate the error correcting code.
3. At the receiver, after parity-check/correction, for every CRC block the CRC sum is checked. If the received and the calculated one differ, then there is an error in the block, so the error correction was not successful. So we send a notification of the detected error and request the resending of the CRC block.

By using CRC, it is possible to detect the cases of block decoding errors at the receiver (correcting into another valid code word due to too many errors; the error changing the message into another valid code word) with relatively small loss in entropy rate. This is true, because block sizes for error correction are typically smaller than in case of CRC, so one CRC block covers more error correcting blocks.

Naturally, using CRC is unsuccessful, if an error occurs in the CRC block itself, but we get no CRC errors. This can happen in the two following cases:

1. The checksum did not change, but the block changed in a way that the result provides the same sum as before.
2. Both the block and the sum changes, but in such a way, that the new sum will match the new block.

[15] If g(x) is given in this way, by octal (base-8) numbers, then we can get the coefficients of g(x) by transforming every digit into binary (base-2) form. In this case, every digit encodes 3 coefficients, e.g. 4 → 100. On the left side, stands the coeff. of highest degree. An example: $g(x) = 57 = x^5 + x^3 + x^2 + x + 1$.

Therefore, the ability to detect errors does not depend only on the number of errors, but on their positions (pattern) too. The quality related properties of CRC codes are the following:

- **Error detection capability**: this can be determined by exact analysis of the specific code. It can be shown, that the undetected error probability approaches 2^{-N} as K increases.
- **Error pattern coverage**: the probability that a randomly chosen CRC-block and a check-sum does not match (it is not a valid CRC block). As the number of such pairs that *do not* produce CRC errors is 2^{N-K}, the coverage is $\lambda = 1 - \frac{1}{2^{N-K}}$, that depends only on the degree of *g(x)*.
- **Burst error detection capability**: Due to some physical effects on the channel (e.g. smudges on a disk, power outages etc.) errors in most cases tend to appear not in a random distribution, but close to each other, in so-called *bursts*[16]. For CRC codes, if the length of a burst is *b*, then the code detects every error burst, if $b \leq N - K$, and $1 - \frac{1}{2^{N-K}}$ part of them if $b > N - K + 1$.

5.2. Reed-Solomon Codes

Reed-Solomon codes are non-binary cyclic codes defined over a finite field GF[q], where q is either a prime number or a power of a prime number p. It can be shown that a finite field always has a *primitive* element a such that $a^{q-1} = 1$ and all the other powers of a are different, which means that the set of these powers is identical to the set of all elements in GF[q].

A typical setup of Reed-Solomon codes in information technology is when we choose $q = 2^8$ such that we have field of 256 elements and an element (a symbol of the code) will be equivalent to a byte. This yields an efficient hardware and software implementation.
For an (N, K) linear block code we choose q such that $q - 1 > N - 1$ and we define the generator matrix **G** of the code as follows.

$$\mathbf{G} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & a & a^2 & a^3 & \cdots & a^{N-1} \\ 1 & a^2 & a^4 & a^6 & \cdots & a^{2(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a^{K-1} & a^{2(K-1)} & a^{3(K-1)} & \cdots & a^{(K-1)(N-1)} \end{bmatrix}$$

When we multiply the message vector $\mathbf{m} = [m_0 \quad m_1 \quad m_2 \quad \cdots \quad m_{K-1}]$ with this generator matrix we get the following components of the code vector **c**:

$$\begin{gathered} c_0 = m_0 + m_1 + m_2 + \cdots + m_{K-1} \\ c_1 = m_0 + m_1 a + m_2 a^2 + \cdots + m_{K-1} a^{K-1} \\ c_2 = m_0 + m_1 a^2 + m_2 a^4 + \cdots + m_{K-1} a^{2(K-1)} \\ \text{etc.} \end{gathered}$$

We now make use of the fact that the above components are indeed the values of the polynomial *m(x)* at $a^0, a^1, a^2, \cdots, a^{N-1}$. Note that $a^0, a^1, a^2, \cdots, a^{N-1}$ are all different because a is a primitive element of the field and $N < q$. We also know from the Fundamental Theorem of Algebra. that an *n*-degree polynomial cannot have more than *n* different roots. From these it follows that there will be at least $N - (K - 1) = r + 1$ non-zero components in any valid code vector of a Reed-Solomon code, i.e. the weight of any code word is at least $r + 1$. This in turn implies, by the power of the code distance theorem, that the code distance is $r + 1$, so the Reed-Solomon

[16] For block codes we define a burst as the distance in the positions of the first and last wrong channel symbol in a code word block of length N, regardless of any errors in between. So even if only the first and last symbols are wrong, the length of the burst is still N.

codes are maximum distance separable (MDS) codes. In the specific case when $K = 1$, the Reed-Solomon code turns into the $(N, 1)$ repetition code.

The code can be made systematic by column-row transforms of the generator matrix.

Problem. Assume that $q = 7, a = 3, N = 6, K = 3$. Compute the systematic generator matrix of the (6, 3) Reed-Solomon code. *Do not forget that all calculations must be performed in the finite field, i.e. modulo 7.*

Erasure error correction with Reed-Solomon codes

If we use a channel with erasures, we may make the assumption that the position of all errors are known, so we need to correct only erasure type errors. According to the above, the code distance of the code is $r + 1$, therefore we can correct *r* erasures. If we have r erasures, then we can use the rest K (non-erased) received channel symbols to construct a linear equation system of K equations to determine the K components of the original message vector $\mathbf{m} = [m_0 \quad m_1 \quad m_2 \quad \cdots \quad m_{K-1}]$.

Problem. Assume that with a (6, 3) Reed-Solomon code the received code vector is $\mathbf{c}' = [5 \quad 1 \quad E \quad E \quad 2 \quad E]$ and the systematic generator matrix is

$$\mathbf{G} = \begin{bmatrix} 6 & 1 & 3 & 1 & 0 & 0 \\ 3 & 3 & 6 & 0 & 1 & 0 \\ 6 & 4 & 6 & 0 & 0 & 1 \end{bmatrix}$$

Find the original (corrected) message.

Correction of errors with an unknown location

The correction of general (not erasure type) errors can be performed in the usual way using the parity check matrix computed from G, and a syndrome table in the same way as shown for the Hamming-codes.

Problem. Compute the parity check matrix and (a part of) the syndrome table for the generator matrix above. Correct a single error with the code.

5.3. Interleaving

Interleaving is a method widely used in telecommunications for various purposes, here we will apply it for error burst control.

Using Hardware Redundancy

An obvious defence against error bursts is dividing the to-be-encoded channel symbol stream into two parts, into odd and even time intervals, and then execute the channel encoding in parallel (maybe even by using different parameters, or encoding techniques). The output pairs of these two encoders are then recombined (interleaved) and transmitted. At the receiver, the words received are split into parts and processed separately with the proper decoders. Then their outputs are arranged (de-interleaved) into the original message sequence.

This process requires 2 encoder/decoder pairs (interleave of 2), but

- These run on half of the channel's clock rate, which is useful in case of fast channels and slow coders (complex algorithms).
- An error burst of length *b* appears on the two streams as two error blocks of length *b*/2, as a result of interleaving. So the communication system needs to handle bursts of half the size than as usual.

Naturally, a similar approach can be utilized for more than double redundancy, to apply n-fold hardware redundancy, which results in decreasing the error burst sizes to 1/n.

Block Interleaver

This solution does not require the usage of multiple coders. The symbol stream encoded by channel coding is used to fill a table of size D×D, in *column-wise* order. When the table is full,

the data is read in a *row-wise* order and sent to the channel. At the recipient, a similar table is loaded in a row-wise order, and when it is full, then the columns are read in their order. It can be seen, that if an error burst emerges on the channel, then up to a specific size ($D^2 - D$), after block de-interleaving at the recipient, it appears in D different blocks, broken into D parts.

A drawback is that filling the table requires D^2 time segments to pass, but this delay can be compensated by buffering.

Application: Storing Music with Reed-Solomon Code

In coding of audio CDs to protect against possible error bursts (e.g. scratches), the block interleaving technique is used. The audio is sampled at 44 KHz, on two channels, and every value recorded is quantized into a 2 byte integer. This way 4 bytes represent one sample. The bytes are written into a 24×24 table *column-by-column (*6 samples per column, 144 in total). Then an error correcting code is used to extend the *rows* of the table, to have 28 columns, and afterwards to extend the *columns* in order to have a 28×28 table. This table is then written onto the disk *row-by-row*. The coding used is a (28, 24) Reed-Solomon code with code distance of 5 over GF[256]. This method splits the error bursts into their one twenty-eighth.

At the receiver, in every row of the table, 1-errors are corrected with the error correcting code. The code could also correct 2-errors, but in those cases the two errors are transformed into an erasure error. For more than 2 errors, the whole row is considered as an erasure error. After the rows, the columns are corrected, where only erasure errors can appear, from which 4 could be corrected, as the code is MDS. Yet the code word is discarded and the missing value is estimated based on its context if more than 2 errors are present.

5.4. Error Rates of Block Codes

The following statements are true for all error correcting block codes. Suppose that we use a binary symmetric channel with error probability p in our system, the code can correct t_c errors and we correct all correctable errors. Then the probability of block decoding error (correcting into another word) at the recipient is

$$P_{bde} = 1 - P_{good\ correction} = 1 - \sum_{i=0}^{t_c} \binom{N}{i} p^i (1-p)^{N-1} \xrightarrow[N \gg 1]{} \approx 1 - e^{-Np} \sum_{i=0}^{t_c} \frac{(Np)^i}{i!}$$

So this probability can be approached as a function of Np and t_c, where Np is the expected number of errors inside the blocks. The question is that for a usable communication system, how much bigger the number of correctable errors should be than that of the expected errors. To answer this, we should take a look at function P_{bde} with regard to t_c/Np. If we require that $P_{bde} > 10^{-10}$(which is quite a weak constraint), then for Np = 2 we get that the code should correct about 8 times more errors. In general, t_c should be not just somewhat bigger than Np, but it should be a greater multiple of Np, in order to have a proper and usable system.

In Section 1.2 we defined our two main goals in information transmission: utilizing channel capacity as much as possible (good entropy rate), and providing the smallest achievable block decoding error probability. When selecting code parameters and coding techniques, we can try two ways to fulfil these:

- Choose N to be small, to have a small Np (fewer errors to be corrected). Unfortunately, for the known block codes, K values close to N are available only for big N values (e.g. for BCH codes correcting 2 errors, if N = 15, then R = K/N = 0.46, and if N = 127, R = 0.89). So it is difficult to achieve good entropy rate in this way.
- Choose N to be large, so by selecting a K small enough we can have enough redundancy for being able to correct the required number of errors. However, this way we also end up with a low entropy rate (K/N).

Therefore, it can be easily seen, that it is challenging to achieve both goals simultaneously. Nonetheless, block codes are widely used, mainly because of their simple generation and parity checking. For improved error correcting properties, there are the convolutional codes that do not require dividing into blocks.

6. Convolutional Codes

Convolutional codes have good error correcting capabilities, however, the generated codes are not systematic, and their decoding is relatively complex. They assign a single code word for an input channel symbol sequence of arbitrary length (like the arithmetic code in source coding), so there is no block creating step. Therefore, the already mentioned concepts used in describing the block codes and the corresponding methods are just partially applicable for convolutional codes.

6.1. Generating Convolutional Codes

Generating convolutional codes is done in a similar way how basic cyclic codes are produced: the input is considered as a polynomial; the code is generated by multiplication with generator polynomials, but in this case, more than one generator polynomial is used simultaneously. Thus the encoder device is a so called FIR encoder (finite impulse response encoder; also called as convolver), like the one shown in Figure 6.1.

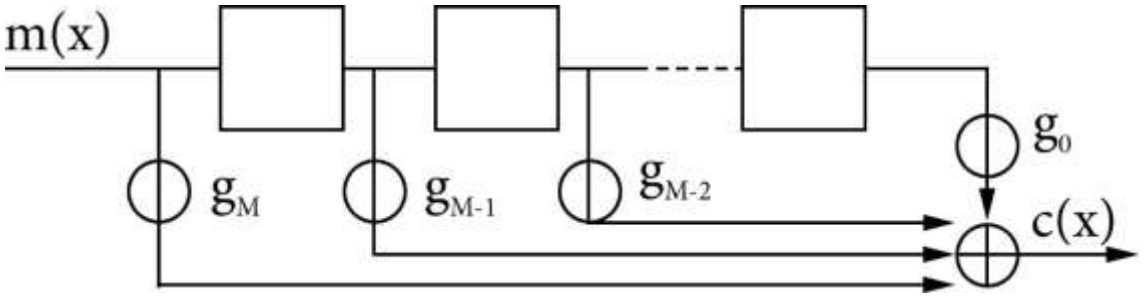


Figure 6.1: *Outline of a polynomial multiplier circuit*
g_i – polynomial coeff.; M – shift register's depth, i.e. number of shift registers; other elements same as in Figure 5.1

The circuit in Figure 6.1 multiplies its input, an m(x) polynomial of arbitrary length, by a g(x) polynomial, and the result is its output c(x). One coefficient is processed in a time segment. The convolutional encoder multiplies the input with different polynomials at the same time by using the same shift register (a multiple input XOR gate and an output for each multiplier polynomial). So the code is defined by the multiplier polynomials.

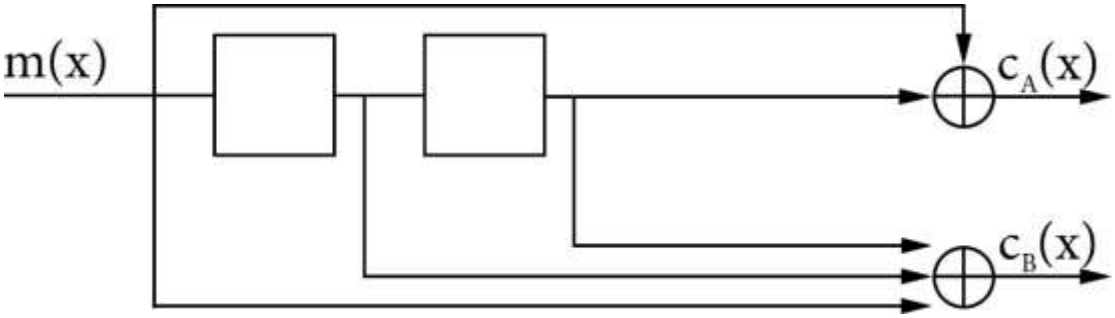


Figure 6.2: *Convolutional encoder with 2 outputs*

The multiplier polynomials of the convolutional code generated in Figure 6.2.:

$$g_A(x) = 1 + x^2$$
$$g_B(x) = 1 + x + x^2$$

In each time segment, a channel symbol enters the encoder, and for every output, one symbol exits. Hence, if we denote the number of outputs with N_0, then its net code rate is

$$R_{net} = \frac{1}{N_0}$$

If the input m(x) channel symbol sequence is not infinite, but is of length K, then the real code rate is even lower, as we need to wait M instances for the register to flush:

$$R_{eff} = \frac{K}{KN_0 + MN_0} = \frac{R_{net}}{1 + \frac{M}{K}} < R_{net}$$

The generated code can also be given in a vector form:

$$c_A(x) = m(x)g_A(x),\ c_B(x) = m(x)g_B(x), \text{ from this } \boldsymbol{c}(x) = m(x)\boldsymbol{g}(x)$$

An important property of convolutional codes is linearity that requires that the sum of two valid code words is also a valid code word. Based on the previous statement, this is also met:

$$\boldsymbol{c}_1(x) = m_1(x)\boldsymbol{g}(x)$$
$$\boldsymbol{c}_2(x) = m_2(x)\boldsymbol{g}(x)$$
$$\boldsymbol{c}_1(x) + \boldsymbol{c}_2(x) = \big(m_1(x) + m_2(x)\big)\boldsymbol{g}(x)$$

6.2. The State Machine Modell

For decoding the generated code, we have to know in which inner state the encoder is for any specific time instant. This can be achieved by considering the encoder as a state machine and using its **state diagram**. For this, the state labels are associated with the contents of the shift register, e.g. for the code in Figure 6.2:

$$S_0 \rightarrow 00$$
$$S_1 \rightarrow 10$$
$$S_2 \rightarrow 01$$
$$S_3 \rightarrow 11$$

A state diagram shows the output and the next state the system will be in, for each input received:

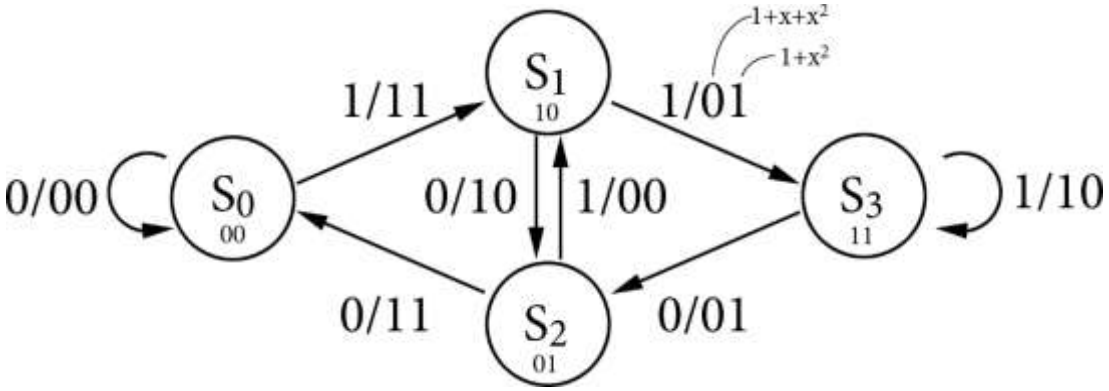


Figure 6.3: *The state diagram of encoder in Figure 6.2*

From these, it is easy to understand the following properties of convolutional codes:

- An all 0 input will create an all 0 output.
- Input 0's after at most M steps lead back to state 0…0.
- To every state, we can get from exactly two states, which have a different LSB.
- From every state, we can get to exactly two states, which have a different MSB.

An *alternative path* or adversary path in a convolutional code is a sequence of state transitions (and the corresponding output) of arbitrary length, that starts in state 0…0, leaves it at least once, and returns to this state. The d_f **free code distance** is the minimum of Hamming distances between the outputs of all possible alternative paths. Similarly to the code distance defined for linear block codes the free distance of a convolutional code is the basis of error correction. As the code is linear, based on Section 4.2, it is enough to find the alterative path with the output of smallest weight to get this free code distance; the weight is the free distance. In many cases, this can be easily found by just taking a look at the state diagram, e.g. in Figure 6.3, it is the path $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_0$, so $d_f = 5$.

The Bit Error Rate

In case of convolutional codes, we cannot talk about block decoding errors in the same way as for linear block codes, as we do not code blocks. Instead, we introduce **bit error rate P_b** as the probability of one binary symbol (bit) being incorrect in a symbol sequence that was decoded and corrected (in a way discussed later on). For the bit error rate, there can be given both a lower and an upper bound, based on transfer function T(N, J, D) and free code distance, as a function of channel error probability. So the transfer function, in fact, fully defines the error correcting capabilities of the code. (More details in the corresponding literature.)

For constructing convolutional codes with good error correcting properties, however, only a few methods are known. In order to find good codes, computers are used. Some examples for such codes can be found in the table below, for others please see [1].

R_{net}	M	Generator polynomials[17]	d_f
1/3	3	(13, 15, 17)	10
1/3	5	(47, 53, 75)	13
1/2	5	(53, 75)	8
1/4	4	(25, 27, 33, 37)	16

6.3. Decoding and Correcting in One Step: the Viterbi Algorithm

This algorithm is used at the receiver to obtain the estimate of message m(x) from a received code word of arbitrary length. As the code is not systematic, this task is far from trivial.

The algorithm uses as many processors as many states are in the state diagram (2^M). These are assigned to the states and are connected accordingly to the graph, i.e. each processor has two input and two output connections (which might, of course, lead into the same unit). In each time segment, each processor receives its input, the N_0 number of channel symbols.

Every processor has a register for storing one integer number, that contains the current value of the **likelihood metric** μ. (There are several types of likelihood metrics, for now, we will use Hamming distance.) Each processor also has a path register *pr* that can store a record of the decoded message bits (0 / 1 / X - empty) for the current path.

The decoder is operating continuously, so we suppose that the code is of infinite length, but at the start, the encoder is considered to be in state S_0. At the beginning, when receiving the first input of N_0 channel symbols, for processor S_0 $\mu = 0$ and for the others $\mu = \infty$, while all path registers contain only empty (X) symbols.

Then in each cycle, the processors execute the following tasks:

1. Send the value of the current path register and the likelihood metric to the output processors, and receive those that arrive from the source processors (pr_1, pr_2, μ_1, μ_2), and read the current input (N_0 channel symbols).

[17] The same octal notation used for cyclic codes

2. For each incoming path, calculate the difference (Hamming distance) between the actual input read and what it should be (based on the state diagram) for the given path leading to the processor. In the previous example, for $S_0 \rightarrow S_1$, transition the appropriate code is 10, but if the received is 11, then this value will be 1.
3. Add the resulting two distances to the corresponding received likelihoods (μ_1, μ_2) as a penalty, and select the smaller one[18]. This also selects which path (/previous state) is to be accepted. If the two are equal, select randomly.
4. Update the (state's) likelihood metric with the selected value, and set the path register to the received path. From the state diagram, write the input (message bit) that corresponds to the selected state to the end of the path register (into the first X from left). In our example, to transition $S_1 \rightarrow S_2$ this message bit is 0.

The controller that uses the decoder compares the contents of the path registers at the end of each time segment/cycle. If for *all processors*, on one or more positions (read from left to right) the same message bits are stored, then these matching bits are considered to be decoded and corrected, and are shifted out (to the left) from each path register.

It can be proved, that the algorithm follows the maximum-likelihood principle, i.e. from the possible inputs it finds the one with highest probability of corresponding to the given code of arbitrary length. The value of the likelihood metric in a given time segment for a specific processor is nothing else, but the total distance between the received code (the input), that may contain errors, and the code that belongs to the path starting in state S_0 and leading to the processor, based on the state diagram. So it shows *how likely is* that the estimate for the given path is the same as the original message sent – this is why the expression "likelihood metric" is used.

6.4. Improvements on Convolutional Codes

In this section, some of the problems of the previously introduced basic algorithm are discussed, with their possible solutions.

The Length of Path Registers
In real life hardware implementations (VLSI), obviously, the path registers are of finite length (e.g. 32 bit registers), so selecting their size properly is an important aspect in design. If all path registers are full, but they differ even in the first bit position, we have no place to write the estimate of the next cycle. In such cases, the processor with the best likelihood value is selected, its register's first bit is removed as the first decoded bit, and all registers are shifted left with one position.

Overflow in the Likelihood Metric
For a code long enough, that has many errors, an arithmetic overflow might occur in the register of the likelihood metric. To avoid this, the worst likelihood is inspected in every cycle. If it approaches the highest representable number, then we decrease every processor's likelihood value with the best (smallest) likelihood, as only their *difference* is important in the algorithm. (Of course, the previously mentioned additional meaning of the likelihood metric is lost this way.)

Improving Communication (the Traceback Method)
Another implementation related problem, for codes of greater depth, is the high number of processors (2^M) and their communication need (sending the path registers, mainly). This can be avoided by using the so called *traceback version* of the algorithm. The concept of this is that the LSBs of the state codes for two processors leading into the same processor, are different.

[18] The larger the Hamming distance is, the further we are from the message possibly sent (so actually, in this case, Hamming distance is more like a sort of an "unlikelihood metric").

So in this version, only the likelihood metric is sent in every cycle, while in the path register the LSB of the selected processor is written. This way, in a given time segment, it is still possible to trace the path that leads to the specific processor by checking the path register. But comparing the processors' results cannot be done by checking the fronts of their registers. Instead, a transition sequence is considered decoded where paths from every processor meet.

Elimination of Ties

As the Viterbi algorithm selects randomly in case of two equal estimates, it might happen that the one it will select is not the better one. This can be avoided by forwarding the sampled signal of the channel's binary symbol to the receiver's decoder without quantization (N_0 real numbers in each cycle). Obviously, for such values Hamming distance cannot be used (in the algorithm's 2nd step), so instead, an Euclidean distance is calculated. For example, if +5V is the ideal "1" on the channel and 0V is the ideal "0", and in a given cycle (of a code with rate 1/2) the code based on the state diagram should be 10, and the received values are 3.4V and 2.9 then the penalty is

$$p = (5 - 3.4)^2 + (0 - 2.9)^2$$

Naturally, in this case the likelihood metric is not an integer, but a real valued number. Because the quantization step is left out, this version is referred as the one that has a **soft-decision decoder**. This method does not only eliminate the possibility of ties (two real values are almost never identical), but it can also provide a better bit error probability by avoiding the information loss caused by quantization.

Increasing code rate

The codes discussed so far had a code rate of 1/2 at most. For less noisy channels, the need for better rates is present, which can be satisfied by a technique called "**puncturing"** a code. Generating these codes is done almost in the same fashion as in the basic version, but the time segments are grouped into periods, and in such a period only a part of the generated code is sent to the channel. This way, good code rates can be achieved. We use the (7, 5) code from the previous example and its punctured version with a puncturing period of 3 as an example. In the 2nd time segment of a period, only the output polynomial of 7 is forwarded, while in the 3rd, only the one that corresponds to 5 is sent.

Time segments	1st	2nd	3rd	4th	5th	6th
Periods	1st period			2nd period		
Message bits	1	1	0	1	0	0
(7, 5) basic code, R=1/2	11	01	01	00	10	11
(7,5), 7,5 punctured code, R=3/4	11	0	1	00	1	0

The state diagram of a punctured code consists of P times more states than the one for its "original" version, if the length of the period is P and the number of processors is still 2^M in the decoder. However, these processors have more complex programs as they have to "know" in which segment of the period they are, to calculate the penalties accordingly.

By puncturing a code, the code rate can be increased, but this results in a loss of error correcting capacity. Some examples of such punctured codes can be found in the table below (it is worth to compare them to the parameters of their non-punctured versions). For more examples please refer to [1].

R_{net}	M	Generator polynomials[19]	d_f
2/3	3	(13, 15) 17	4
3/4	5	(61, 53), 53, 53	5

[19] The same octal notation used for cyclic codes

4/5	5	(61, 53), 47, 47, 53	4
5/6	4	(37, 23), 23, 23, 25, 25	4

Acknowledgement

I thank *Miklós Vassányi* for his guidance in creating the epistemology overview, *Balázs Gaál* and *András Nemetz* for their contribution in editing and for their help with the equations and figures. I also thank *Benedek Szakonyi* for his tremendous help in translating and editing this syllabus.

Appendices

A1 Uniform source distribution maximises source entropy

Proof:

Let A = {p_0, p_1, …} and |A| = M. The proof can be considered complete once we have shown that H(A) – log(M) ≤ 0 and equality is satisfied only if $p_i = \frac{1}{M}$, as log(M) is the entropy of a source with equal distribution.

It can be rewritten as:

$$\sum_i p_i \log \frac{1}{p_i} - \log(M) =$$

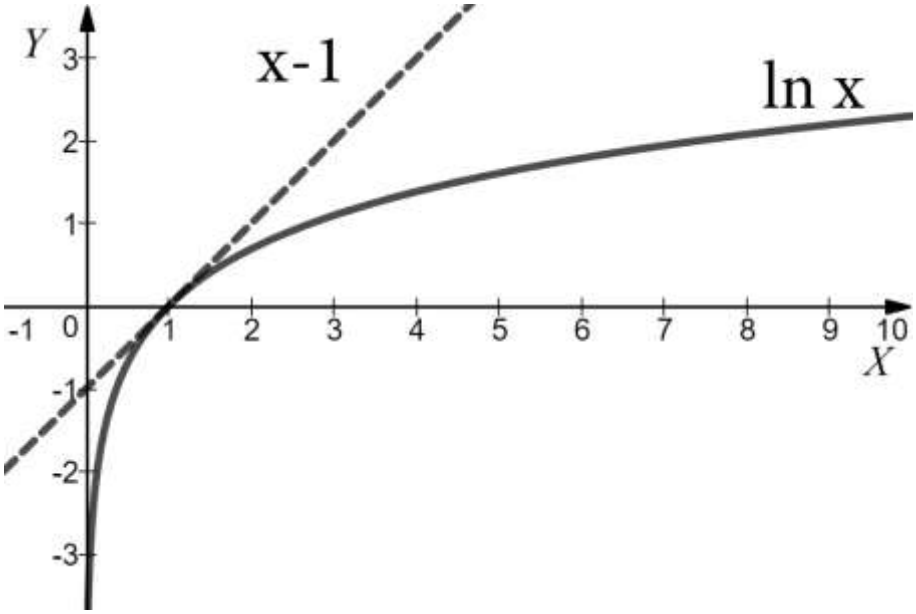
$$= \sum_i p_i \log \frac{1}{p_i M} =$$

$$= \frac{1}{\ln 2} \sum_i p_i \ln \frac{1}{p_i M}$$

Here we can use that by substituting $x = \frac{1}{p_i M}$ into all members of the summation, the following inequality is applicable:

$$\ln x \leq x - 1 \text{ and it is equal only if } x = 1$$

This inequality can be proved by analysing the function graphs



Substituting the inequality into the expression:

$$H(A) - \log M \leq \frac{1}{\ln 2} \sum_i p_i \left(\frac{1}{p_i M} - 1 \right) = 0$$

So indeed, H(A) – log(M) ≤ 0 and equality is satisfied only if $x = 1$, i.e. $p_i = \frac{1}{M}$ for all i, which is the case of equal distribution.

A2 "**There is no** lossless compression algorithm that could *always* encode a binary source's symbol sequence of arbitrary length N by using fewer symbols than N (not even by 1 less, i.e. with at most N-1 binary symbols)."

Proof:

The statement uses the pigeonhole principle. Suppose indirectly that there exists such an encoding machine. It would have 2^N different binary inputs of length N in total, where the number of possible outputs is 2^{N-1} if the output is of length N – 1, 2^{N-2} if its length is N – 2 and so on. Besides that any of these could be used, the total number of possible outputs would be still less than the number of possible inputs:

$$2^{N-1} + 2^{N-2} + \cdots + 2^1 = 2^N - 1 < 2^N$$

This means that there would exist at least two inputs that for the encoder produces the same output, so the code would not be decodable.