Benjamin Hackl

Concatenated Error Correcting Codes: Galois and Binary Concatenation

Bachelorarbeit

zur Erlangung des akademischen Grades
BSc.

Studium
Technische Mathematik

Alpen-Adria-Universität Klagenfurt
Fakultät für Technische Wissenschaften

Betreuer
Univ.-Prof. Dr. Clemens Heuberger

Institut für Mathematik

Klagenfurt, 11. Juni 2014

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende wissenschaftliche Arbeit selbstständig angefertigt und die mit ihr unmittelbar verbundenen Tätigkeiten selbst erbracht habe. Ich erkläre weiters, dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle aus gedruckten, ungedruckten oder dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte sind gemäß den Regeln für wissenschaftliche Arbeiten zitiert und durch Fußnoten bzw. durch andere genaue Quellenangaben gekennzeichnet.

Die während des Arbeitsvorganges gewährte Unterstützung einschließlich signifikanter Betreuungshinweise ist vollständig angegeben.

Die wissenschaftliche Arbeit ist noch keiner anderen Prüfungsbehörde vorgelegt worden. Diese Arbeit wurde in gedruckter und elektronischer Form abgegeben. Ich bestätige, dass der Inhalt der digitalen Version vollständig mit dem der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Benjamin Hackl

Klagenfurt, 11. Juni 2014

Acknowledgements

It would not have been possible to write this thesis without the help and support from many of the people around me. In these few lines I want to thank everyone involved in this process.

Firstly, I want to thank my advisor Dr. Clemens Heuberger, not only for the broad spectrum of support he provided throughout the whole process, but also for bringing many exciting challenges to my attention and keeping me busy. I am very much looking forward to future collaborations.

Secondly, I want to thank my colleagues at the department of mathematics for making my work- and study-environment a very interesting and enjoyable place.

My work at the department of mathematics – and therefore also this thesis – is funded by the Austrian Research Promotion Agency (FFG) as a part of the project "CODES – Algorithmic extraction and error correction codes for lightweight security anchors with reconfigurable PUFs", so I also want to thank my colleagues within the project; especially Michela Mazzoli and Dr. Winfried Müller at the *Alpen-Adria Universität Klagenfurt*, Verena Brunner and Martin Deutschmann at *Technikon Forschungs- und Planungsgesellschaft mbH*, and Dr. Ingrid Schaumüller-Bichl and Andrea Kolberger from the University of Applied Sciences in Upper Austria.

Furthermore, I want to express my gratitude to my friends and colleagues from university for many memorable discussions and moments, as well as for the many (late) night shifts in their company.

And last but not least I want to thank my family for encouraging me to choose my own path in life as well as for their never ending support regarding my choice.

Thank you!

Abstract

The class of concatenated linear codes as investigated by David Forney in 1965 consists of some very powerful codes regarding error detection and correction. Additionally, another big advantage of this class of codes is the possibility to use very simple and efficient decoding algorithms, compared to other codes of similar length.

In this thesis we want to give an introduction to the topic of algebraic coding theory, investigate some properties of an alternative transmission channel model influenced by studying the behavior of uninitialized SRAM-cells, and analyze the relation between the aforementioned class of concatenated codes (which we will call Galois concatenated codes) and another class of codes (binary concatenated codes).

We will deduce that the binary concatenation is a generalization of the Galois concatenation and conduct some simulations in order to compare the performance of these two code constructions for specific examples. Furthermore, we investigate under which conditions square matrices over a finite field behave like elements from a larger finite field.

Contents

1 Algebraic preliminaries

In the following sections we give a short introduction on the algebraic constructions occurring repeatedly in the following chapters.

We will be following [3] and [7] for the theoretical overview on the algebraic structures used in the chapters afterwards (i.e. primarily fields and especially finite fields). Most results here will be presented without proof, they can be found in the literature. At first, we introduce some basic structures.

1.1 Basic structures

Definition 1.1 (Group, Abelian group).
A set G together with a binary operation $*\colon G \times G \to G$, $(a, b) \mapsto a * b$ is called a *group* if the following requirements are met:

(G1) The binary operation $*$ is *associative*, meaning that for all a, b, $c \in G$ the relation

$$(a * b) * c = a * (b * c)$$

holds.

(G2) There is an element $e \in G$ which has the following properties:

1. The element e is a *left identity*, i.e. for all $a \in G$ we have $e * a = a$.
2. For each $a \in G$ there is an element $b \in G$, such that $b * a = e$ holds. The element b is called *left inverse* of a.

If additionally the property $a * b = b * a$ holds for all a, $b \in G$, then G is called an *abelian group*. If H is a subset of G and $(H, *)$ also is a group, then H is called a *subgroup* of G, we write $H \leq G$.

Definition 1.2 (Group homomorphisms).
Given two groups $(G, *)$ and $(G', \star)$, a map $\varphi\colon G \to G'$ is called *(group) homomorphism* if

$$\varphi(a * b) = \varphi(a) \star \varphi(b)$$

holds for all a, $b \in G$. A homomorphism φ is called *monomorphism* if φ is injective, *epimorphism* if φ is surjective, and *isomorphism* if φ is bijective.

Two groups are called *isomorphic*, in symbols $G \cong G'$, if there is an isomorphism $\varphi\colon G \to G'$. In the special case of $G = G'$, a homomorphism is also called *endomorphism* and an isomorphism is also called *automorphism*.

Definition 1.3 (Cosets and normal subgroups).
Let H be a subgroup of a group G and let $a \in G$. The sets $aH := \{ax \mid x \in H\}$ and $Ha := \{xa \mid x \in H\}$ are called left coset of H in G with respect to a and right coset of H in G with respect to a, respectively.

Furthermore, H is called normal subgroup of G, if $aH = Ha$ holds for all $a \in G$, i.e. the left cosets and right cosets coincide for all $a \in G$. If H is a normal subgroup of G, we also write $H \trianglelefteq G$.

Theorem 1.1 (Factor groups).
Given a group G and a normal subgroup $N \trianglelefteq G$. Then there is exactly one binary operation $*$ on the set of cosets G/N with the following properties:

(a) $(G/N, *)$ is a group.

(b) The canonical surjective map

$$\rho : G \to G/N,\ a \mapsto aN = Na$$

is a homomorphism.

The identity in $(G/N, *)$ is $N \in G/N$ and the inverse of aN is $a^{-1}N$. The group $(G/N, *)$ is called *factor group* of G modulo N.

Definition 1.4 (Ring).
A *ring* is a set R together with two binary operations $+$ and $\cdot$ for which the following properties hold:

(R1) $(R, +)$ is an abelian group.

(R2) The multiplication $\cdot$ is associative.

(R3) The distributive laws hold, we have

$$a \cdot (b + c) = a \cdot b + a \cdot c \quad \text{and} \quad (b + c) \cdot a = b \cdot a + c \cdot a$$

for all $a,\ b,\ c \in R$.

The additive identity $0 \in R$ is called zero element of R. If R has an multiplicative identity $1 \in R$, the element 1 is called unit element and R is called unitary ring. If R is commutative with respect to the multiplication, i.e. $a \cdot b = b \cdot a$ for all $a, b \in R$, then the ring is called *commutative ring*.

Definition 1.5 (Ring homomorphisms).
Let $(R, +, \cdot)$ and $(R', +', \cdot')$ be two rings. A map $\varphi : R \to R'$ is called *ring homomorphism* if

$$\varphi(a + b) = \varphi(a) +' \varphi(b) \quad \text{and} \quad \varphi(a \cdot b) = \varphi(a) \cdot' \varphi(b)$$

hold for all $a,b \in R$. The terms *monomorphism*, *epimorphism*, *isomorphism*, *endomorphism* and *automorphism* are defined in analogy to group homomorphisms.

Furthermore, if $\varphi : R \to R'$ is a ring homomorphism, the set $\ker \varphi = \{a \in R \mid \varphi(a) = 0\} \subseteq R$ is called *kernel* of φ and $\operatorname{Im} \varphi = \varphi(R) \subseteq R'$ is called *image* of φ.

Definition 1.6 (Polynomial ring).
Given a commutative ring with unit element 1. Then the *polynomial ring* $R[X]$ is defined as the set of all sequences $(a_0, a_1, \ldots, a_k, \ldots)$ with $a_j \in R$ and $a_j = 0$ for almost all $j \in \mathbb{N}_0$.

The addition in $R[X]$ is defined componentwise, whereas the multiplication is defined through a convolution, i.e.

$$(a_0, a_1, \ldots, a_k, \ldots) \cdot (b_0, b_1, \ldots, b_k, \ldots) = (c_0, c_1, \ldots, c_k, \ldots), \text{ where}$$

$$c_k := \sum_{j=0}^{k} a_j b_{k-j}.$$

The unit element in $R[X]$ is $(1, 0, 0, \ldots)$, and R is a subring of $R[X]$. We set $X := (0, 1, 0, \ldots)$, and because of the definition of multiplication in $R[X]$ we have

$$X^k = (0, \ldots, 0, 1, 0, \ldots), \quad k \in \mathbb{N}_0,$$

where the 1 is in the $(k+1)$-st component. Overall, we obtain

$$f = (a_0, a_1, \ldots, a_n, 0, 0, \ldots) = a_0 + a_1 X + \cdots + a_n X^n,$$

which justifies the name *polynomial ring*.

The *degree* of a polynomial $f \in R[X]$ is defined through

$$f = a_0 + a_1 X + \cdots + a_n X^n, \ a_n \neq 0 \Rightarrow \deg f = n.$$

Furthermore, if $a_n = 1$, then f is called a *monic polynomial*.

Definition 1.7 (Ideal).
A subset $A \subseteq R$ of a ring $(R, +, \cdot)$ is called *ideal*, if the following properties hold:

(I1) $A \subseteq R$ is an additive subgroup.

(I2) For all $a \in A$ and all $x \in R$ we have $a \cdot x \in A$ as well as $x \cdot a \in A$.

If the ring R is a commutative unitary ring, then for $a \in R$ the set

$$(a) := R \cdot a = \{x \cdot a \mid x \in R\}$$

is called *principal ideal* generated by a.

Definition 1.8 (Integral domain and Euclidean domain).
A commutative unitary ring without *zero divisors* (i.e. for all elements $a, b \neq 0$ we have $a \cdot b \neq 0$) is called *integral domain*.

An integral domain R is said to be a *Euclidean domain* if there is a map $\delta : R \setminus \{0\} \to \mathbb{N}$ such that for all $a, b \in R$ with $b \neq 0$ there are elements $q, r \in R$ so that

$$a = q \cdot b + r \quad \text{and} \quad \delta(r) < \delta(b) \text{ if } r \neq 0$$

holds. δ is called *degree function* or *norm*.

Lemma 1.2.
A polynomial ring $K[x]$ over a field K is a Euclidean domain.

Theorem 1.3.
A Euclidean domain is a principal ideal domain, that is every ideal in a Euclidean domain is a principal ideal.

Theorem 1.4 (Factor ring, quotient ring, residue class ring).
Given a ring R, an ideal A and $\rho : R \to R/A$ the canonical group homomorphism (cf. Theorem 1.1). Then there is exactly one multiplication $\cdot$ in R/A so that R/A becomes a ring and ρ becomes a ring epimorphism with $\ker \rho = A$.

$(R/A, +, \cdot)$ is called factor ring of R modulo A. The operations in R/A can be expressed through *congruences*, we define

$$x \equiv x' \mod A :\iff x + A = x' + A \iff x - x' \in A.$$

These definitions and theorems are fundamental for the following results regarding fields and field extensions. Especially the knowledge about factor rings and polynomial rings will help with the construction of finite fields at a later point.

1.2 Fields and field extensions

Definition 1.9 (Fields).
Let $(F, +, \cdot)$ be a commutative unitary ring. If every $a \in F \setminus \{0\}$ has an inverse element $a^{-1} \in F$ such that $a \cdot a^{-1} = a^{-1} \cdot a = 1$ holds, then F is called *field*.

If L and K are both fields with $L \subseteq K$, then L is called a *subfield* if L inherits the operations from K. The field K is then also called *extension field* of L and $L \subseteq K$ is called a *field extension*. If L is a subfield of K and $L \neq K$, then L is called *proper subfield*. A field containing no proper subfields is called *prime field*.

Finally, a field F is called *finite field* or *Galois field* if F has cardinality $q \in \mathbb{N}$. A field with q elements is denoted as $\mathbb{F}_q$ or $\mathrm{GF}(q)$. This notation is based on the property that two finite fields with q elements are isomorphic (cf. Theorem 1.16).

Definition 1.10 (Characteristic).
Given a ring with unit element 1, there is always the homomorphism

$$\varphi : \mathbb{Z} \to R,\ n \mapsto n \cdot 1 = \underbrace{1 + 1 + \cdots + 1}_{n \text{ times}}.$$

The kernel of φ is a subgroup of $\mathbb{Z}$, and all subgroups of $\mathbb{Z}$ are given by the groups $m\mathbb{Z}$ where $m \in \mathbb{N}_0$. Thus, there has to be an $m \in \mathbb{N}_0$ such that $\ker \varphi = m\mathbb{Z}$. In that case, the characteristic of the ring R is defined to be m, in signs $\chi(R) = m$.

A direct consequence of the definition above is that the characteristic of a field F either has to be 0 or a prime number – otherwise, F would contain zero divisors, which contradicts the field property.

Theorem 1.5.
Let F be a field. If $\chi(F)$ is a prime number p, then the prime subfield of F is isomorphic to $\mathrm{GF}(p) = \mathbb{Z}/p\mathbb{Z}$. Otherwise, $\chi(F)$ has to equal 0 – then, the prime subfield is isomorphic to $\mathbb{Q}$.

Definition 1.11 (Degree of a field extension).
Let $K \subseteq F$ be a field extension. Then F can be considered a vector space over K, where the scalar

multiplication $\cdot : K \times F \to F$ exactly is the multiplication in the field restricted in the first operand. Then the *degree of the field extension* $K \subseteq F$ is defined as

$$[F : K] := \dim_K(F).$$

Theorem 1.6 (Degree formula).
If $F \subseteq K$ is a field extension and $K \subseteq L$ also is a field extension, the field K is called *intermediate field* and the formula

$$[L : F] = [L : K] \cdot [K : F]$$

applies.

Definition 1.12 (Field extension by adjoining elements).
Let K be a subfield of a field F and let M be a fixed subset of F. Then the field $K(M)$ is defined as the intersection of all subfields of F containing both K and M,

$$K(M) := \bigcap_{\substack{K \cup M \subseteq L \subseteq F \\ L \text{ field}}} L.$$

The field $K(M)$ is called the extension field of K obtained by adjoining the elements in M. If M consists of a single element $\alpha \in F$, then $L = K(\alpha)$ is said to be a *simple extension* of K and α is called a *defining element*, *generating element* or *primitive element* of L over K.

Definition 1.13 (Algebraic elements and algebraic extensions).
Let K be a subfield of F and let $\alpha \in F$. If there is a nonzero polynomial $f \in K[X]$ such that α is a root of f, then α is said to be *algebraic* over F. Otherwise it is called *transcendental*.

An extension L of K is said to be an *algebraic extension* of K if every element of L is algebraic over K. If $[L : K] < \infty$, the extension is called *finite extension*.

Lemma 1.7 (Finite extensions are algebraic).
Every finite extension of a field K is algebraic over K.

Definition 1.14 (Minimal polynomial).
Let F be an extension field of K. If $\alpha \in F$ is algebraic over K, then the uniquely determined monic polynomial $g \in K[x]$ generating the ideal $\{f \in K[x] \mid f(\alpha) = 0\}$ is called the *minimal polynomial* of α over K.

Theorem 1.8 (Properties of the minimal polynomial).
Let F be an extension field of K. If $\alpha \in F$ is algebraic over K, then its minimal polynomial $g \in K[x]$ has the following properties:

1. The polynomial g is *irreducible* in $K[x]$, meaning that it cannot be factored into the product of two non-constant polynomials in $K[x]$.
2. For $f \in K[x]$ we have $f(\alpha) = 0$ if and only if g divides f.
3. The polynomial g is the monic polynomial in $K[x]$ of least degree having α as a root.

The following theorems yield the theoretical basis on how to construct field extensions.

Theorem 1.9.
Let F be an extension field of K, let $\alpha \in F$ be algebraic over K, and let g be the minimal polynomial of α with $\deg g = n$. Then the following holds:

1. $K(\alpha)$ is isomorphic to $K[x]/(g)$, where (g) denotes the principle ideal generated by g.
2. $[K(\alpha) : K] = n$ and $\{1, \alpha, \ldots, \alpha^{n-1}\}$ is a basis of $K(\alpha)$ over K.
3. Every $\omega \in K(\alpha)$ is algebraic over K and the degree of its minimal polynomial is a divisor of n.

Definition 1.15 (Splitting field).
Let $f \in K[x]$ be of positive degree and F an extension field of K. Then f is said to *split* in F if f can be written as a product of linear factors in $F[x]$, i.e. if there exist elements $\alpha_1, \ldots, \alpha_n \in F$ such that

$$f(x) = a \cdot (x - \alpha_1)(x - \alpha_2) \ldots (x - \alpha_n),$$

where a is the leading coefficient of f. The field F is a *splitting field* of f over K if f splits in F and if, moreover, $F = K(\alpha_1, \alpha_2, \ldots, \alpha_n)$, that is, F is the smallest field containing all zeros of f.

Theorem 1.10 (Existence and Uniqueness of Splitting Fields).
If K is a field and f is any polynomial of positive degree in $K[x]$, then there exists a splitting field of f over K. Any two splitting fields of f over K are isomorphic under an isomorphism which keeps the elements of K fixed and maps roots of f into each other.

Before we start considering finite fields only, there is one more result that will be very useful at a later point.

Theorem 1.11.
If a field K has characteristic 0 or K is a finite field, any non-trivial irreducible polynomial $f \in K[x]$ does not have multiple zeros in the splitting field F of f over K, that is, there are $\deg f$ distinct zeros of f in F.

Example 1.1.
The polynomial $g(x) = x^2 + 2 \in \mathbb{R}[x]$ is certainly irreducible. The splitting field of g is $\mathbb{C}$, as the zeros are $x_{1,2} = \pm i\sqrt{2}$. Simultaneously, g is the minimal polynomial of $i\sqrt{2}$, as g is a monic polynomial and there is no polynomial with real coefficients and degree 1 which has $i\sqrt{2}$ as a root.

Also, the zeros are *complex conjugates*. This situation with *conjugate elements* can be further generalized, as we will see in the following section.

1.3 Finite fields

The following results aim for a characterization of finite fields.

Lemma 1.12.
Let F be a finite field containing a subfield K with q elements. Then F has q^m elements where $m = [F : K]$.

Theorem 1.13 (Number of elements in a finite field).
Let F be a finite field. Then F has p^n elements where the prime p is the characteristic of F and n is the degree of F over its prime subfield.

In essence, the construction of finite fields always follows the same procedure: Starting from $\mathrm{GF}(p) = \mathbb{Z}/p\mathbb{Z}$, where p is some prime number, we can adjoin roots of irreducible polynomials to $\mathrm{GF}(p)$ and thus obtain another finite field. Before further characterizing this construction process, we want to give some more properties of finite fields itself.

Lemma 1.14.
If F is a finite field with q elements, then every $a \in F$ satisfies $a^q = a$.

Due to this lemma we can give another possibility to (formally) obtain the finite field with q elements:

Corollary 1.15.
If F is a finite field with q elements and K is a subfield of F, then the polynomial $x^q - x$ in $K[x]$ factors in $F[x]$ as

$$x^q - x = \prod_{a \in F} (x - a)$$

and F is a splitting field of $x^q - x$ over K.

Theorem 1.16 (Existence and Uniqueness of Finite Fields).
For every prime p and every positive integer n there exists a finite field with p^n elements. Any finite field with $q = p^n$ elements is isomorphic to the splitting field of $x^q - x$ over $\mathrm{GF}(p)$.

Therefore, we may speak of *the* finite field or *the* Galois field with q elements.

Theorem 1.17 (Subfield Criterion).
Let $\mathrm{GF}(q)$ be the finite field with $q = p^n$ elements. Then every subfield of $\mathrm{GF}(q)$ has order p^m, where m is a positive divisor of n. Conversely, if m is a positive divisor of n, then there is exactly one subfield of $\mathrm{GF}(q)$ with p^m elements.

The following results will specifically deal with the construction process of finite fields and therefore will give some properties of irreducible polynomials over $\mathrm{GF}(q)$.

Theorem 1.18.
If f is an irreducible polynomial in $\mathrm{GF}(q)[x]$ of degree m, then f has a root α in $\mathrm{GF}(q^m)$. Furthermore, all the roots of f are simple and are given by the m distinct elements α, α^q, α^{q^2}, ..., $\alpha^{q^{m-1}}$ of $\mathrm{GF}(q^m)$.

Corollary 1.19.
Let f be an irreducible polynomial in $\mathrm{GF}(q)[x]$ of degree m. Then the splitting field of f over $\mathrm{GF}(q)$ is given by $\mathrm{GF}(q^m)$.

Corollary 1.20.
Any two irreducible polynomials in $\mathrm{GF}(q)[x]$ of the same degree have isomorphic splitting fields.

The zeros of irreducible polynomials have quite some interesting properties – which is why we will give them a distinct name in the following definition:

Definition 1.16 (Conjugate elements).
Let $\mathrm{GF}(q^m)$ be an extension field of $\mathrm{GF}(q)$ and let $\alpha \in \mathrm{GF}(q^m)$. Then the elements α, α^q, α^{q^2}, …, $\alpha^{q^{m-1}}$ are called the *conjugates* of α with respect to $\mathrm{GF}(q)$.

Theorem 1.21 (Properties of conjugate elements).
Let $\mathrm{GF}(q^m)$ be an extension field of $\mathrm{GF}(q)$ and let $\alpha \in \mathrm{GF}(q^m)$. Denote the conjugates of α by $\alpha^{(j)} := \alpha^{p^{j-1}}$. Then the following statements hold:

(a) The conjugates of α with respect to $\mathrm{GF}(q)$ are distinct if and only if the minimal polynomial of α over $\mathrm{GF}(q)$ has degree m. Otherwise, the degree d of this minimal polynomial is a proper divisor of m, and then the conjugates are the distinct elements α, α^q, …, $\alpha^{q^{d-1}}$, each repeated m/d times.

(b) The distinct automorphisms of $\mathrm{GF}(q^m)$ over $\mathrm{GF}(q)$ are exactly the mappings σ_1, …, σ_m defined by $\sigma_j(\alpha) = \alpha^{(j)}$.

Due to the results above, the construction of $\mathrm{GF}(p^n)$ is clear. Firstly, we find an irreducible polynomial of degree n (which is not that easy, as it requires us to know whether a given polynomial can be factored). Then, we formally adjoin a zero α of this polynomial to $\mathrm{GF}(p)$ and we obtain $\mathrm{GF}(p)(\alpha) = \mathrm{GF}(p^n)$.

1.4 Frobenius normal form

In this section we will introduce the Frobenius normal form of a matrix – which is essentially a generalized Jordan normal form, but with the improvement that we stay in the original field. That is, if we have a matrix $P \in \mathrm{GF}(q)^{k\times k}$, the Frobenius normal form also is in $\mathrm{GF}(q)^{k\times k}$ – which is not the case for the Jordan normal form, as the characteristic polynomial of P is from $\mathrm{GF}(q)[x]$ and might be irreducible over $\mathrm{GF}(q)$, i.e. the eigenvalues of the matrix P could be from an extension field.

Before we give the definition of the Frobenius normal form, we will introduce *companion matrices*.

Lemma 1.22 (Companion matrices).
Let F be a field. Given a monic polynomial $g(x) = x^n + c_1 x^{n-1} + \cdots + c_{n-1}x + c_n \in F[x]$. Then the matrix $C(g) \in F^{n\times n}$ given by

$$C(g) := \begin{bmatrix} 0 & 1 & & & \\ 0 & & 1 & & \\ \vdots & & & \ddots & \\ 0 & & & & 1 \\ -c_n & -c_{n-1} & -c_{n-2} & \cdots & -c_1 \end{bmatrix}$$

is called (left sided) *companion matrix* of g. The *right sided companion matrix* of g is given by the transposed matrix $C(g)^t$.

The characteristic polynomial $\det(x \cdot I_n - C(g))$ of $C(g)$ is g.

Remark.
Assuming the investigated polynomial g is irreducible, companion matrices are matrix representations of the linear maps corresponding to the multiplication with a generating element of an extension field regarding the respective *polynomial basis*, i.e. $(1, \alpha, \alpha^2, \ldots, \alpha^{n-1})$, where α is a root of g.

Proof of Lemma 1.22. We use induction on the degree of the monic polynomial g. If $\deg g = 1$, we have $g(x) = x - a$ for some $a \in F$. The companion matrix of g is the (1×1)-matrix (a) whose characteristic polynomial obviously is $x - a = g(x)$.

Assume that the statement above holds for some arbitrary but fixed $n \in \mathbb{N}$ where $\deg g = n$.

Now consider a monic polynomial g with $\deg g = n + 1$ with companion matrix $C(g)$. Then we have

$$\chi_{C(g)}(x) = \det(x \cdot I_{n+1} - C(g)) = \det \begin{bmatrix} x & -1 & & & \\ 0 & x & -1 & & \\ \vdots & & \ddots & \ddots & \\ 0 & & & x & -1 \\ c_{n+1} & c_n & \cdots & c_2 & x + c_1 \end{bmatrix}$$

Using Laplace's formula for the first column, we find

$$\chi_{C(g)}(x) = x \cdot \det \begin{bmatrix} x & -1 & & \\ 0 & \ddots & \ddots & \\ & & x & -1 \\ c_n & \cdots & c_2 & x + c_1 \end{bmatrix} + (-1)^n \cdot c_{n+1} \cdot (-1)^n,$$

where the remaining determinant is the characteristic polynomial of the companion matrix of the polynomial $\tilde{g}(x) = x^n + c_1 x^{n-1} + \cdots + c_n$, and due to the assumption above equals to $\tilde{g}(x)$. Therefore, we have

$$\chi_{C(g)}(x) = x \cdot \tilde{g}(x) + c_{n+1} = x^{n+1} + x^n c_1 + \cdots + c_n x + c_{n+1} = g(x),$$

which completes the proof. □

For the following statements we follow [2, p. 472 ff.].

Definition 1.17 (Frobenius normal form, rational canonical form).
Let F be a field. A quadratic matrix $T \in F^{n \times n}$ is said to be in *Frobenius normal form* or *rational canonical form*, if T is a block diagonal matrix with companion matrices for monic polynomials $a_1(x)$, $a_2(x)$, …, $a_m(x) \in F[x]$ with degree at least 1 and $a_1 \mid a_2 \mid \cdots \mid a_m$. The polynomials $a_i(x)$ are called the *invariant factors* of the matrix.

Theorem 1.23 (Frobenius normal form – existence and properties).
Let V be a finite dimensional vector space over the field F and let T and S be linear maps T, $S \colon V \to V$. Then the following holds:

1. There is a basis for V with respect to which the matrix representation of T is in Frobenius normal form.

2. The Frobenius normal form for T is unique.
3. Two linear maps S and T are similar if and only if their Frobenius normal forms coincide.

It is quite unsatisfactory that we do not know much about the *invariant factors*. The following lemma will give us some more properties. For more information on invariant factors and algorithms on how to calculate the Frobenius normal form see [2].

Lemma 1.24 (Properties of invariant factors).
For the invariant factors of some quadratic matrix T over the field F, the following statements hold:

1. The product of the invariant factors is the characteristic polynomial.
2. The "largest" invariant factor (i.e. a_m if we have $a_1 \mid a_2 \mid \cdots \mid a_m$) is the minimal polynomial of the matrix T, i.e. the monic polynomial m_T of least degree dividing the characteristic polynomial and satisfying $m_T(T) = 0$.

Remark.
In a more general setting, the invariant factors generate so-called cyclic $F[x]$-modules, i.e. $F[x]/(a_j(x))$. The direct sum of these modules is isomorphic to the underlying vector space V over F. The a_j with $a_1 \mid a_2 \mid \cdots \mid a_m$ and satisfying the property above are uniquely determined.

Furthermore, a direct consequence of Lemma 1.24 is that for diagonalizable matrices whose characteristic polynomial is the power of an irreducible polynomial g, the Frobenius normal form is the block diagonal matrix where the companion matrix of g is stacked on the main diagonal. This fact will be useful at a later point.

2 Coding theory

Before we discuss the construction strategies for concatenated codes, we will give a short introduction on some coding theoretic basics. We will follow [6], [7] and [8]. Again, proofs will be omitted and can be found in the literature.

2.1 Elements of coding theory

The subject of coding theory is the accurate and efficient transfer of data from a sender to a receiver. The importance of accuracy and efficiency can be emphasized by thinking about a satellite traveling in deep space sending data (e.g. environmental measurements, images, ...) by radio transmission to a basis station on earth. During the transmission there are all kinds of interferences with the signal, possibly changing transmitted zeros to ones and vice versa. If there are neither error correction nor error detection mechanisms implemented, the operators in the basis station on earth will not be able to extract very much data from the received data stream.

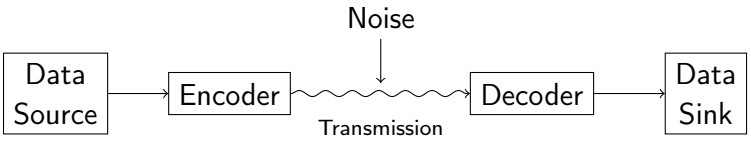

Figure 2.1: Data transmission

Error correcting codes are such mechanisms that improve accuracy and efficiency of data transmitted. They do so by adding *redundancy* to the code, enabling us to reconstruct missing parts of the data transmitted as well as the detection and possibly also the correction of errors. This section intends to give the most important definitions and results regarding coding theory, while specific examples of error correcting codes will be presented in Section 2.2.

The following definitions will introduce codes in a rather general setting.

Definition 2.1 (Alphabet, words and Kleene closure).
An *alphabet* A is a finite, non-empty set of *symbols*, e.g. digits. A word w over the alphabet A is a finite sequence of symbols of A, meaning that if w is a word over A, then there is a $n \in \mathbb{N}_0$ such that $w = a_1 a_2 \dots a_n$, where a_j are symbols from A for all j.

The *length* of a word is the number of symbols in A. If ε denotes the *empty word*, i.e. the word of length 0, and if we define $A^0 := \{\varepsilon\}$, $A^1 := A$ and $A^j := \{ab \mid a \in A,\ b \in A^{j-1}\}$, then the set of all words over the alphabet A is defined by

$$A^* := \bigcup_{j \in \mathbb{N}_0} A^j.$$

A^* is called the *Kleene closure* of A and the unary operator $*$ is called *Kleene star*.

Definition 2.2 (Block codes and related terms).
Given an alphabet A and a injective map $c : A^k \to A^n$, $k \leq n$. Then the image $\mathrm{Im}(c) \subseteq A^n$ is called *code* of length n and dimension k, the elements in $\mathrm{Im}(c)$ are called *code words*, A^k is called *message space* and its elements are the *messages*.
The map c is called *encoding scheme* and the procedure of applying c to a message is called *encoding*. If a map $d : A^n \to A^k$ satisfies $d(c(m)) = m$ for all messages $m \in A^k$, then d is called a *decoding scheme*.

If the map c satisfies

$$c(a_1a_2\dots a_k) = a_1a_2\dots a_ka_{k+1}\dots a_n \quad \text{or} \quad c(a_1a_2\dots a_k) = a_{k+1}\dots a_na_1a_2\dots a_k$$

then the encoding scheme is said to be in *standard form*[1].

Remark.
For various reasons it is very comfortable to assume that the alphabet A equals some finite field. Codes over $\mathrm{GF}(q)$ are called q-ary codes.

Binary codes, i.e. codes over the alphabet $\{0,1\} = \mathrm{GF}(2)$, have a very special role in computer science. The investigation of q-ary codes is theoretically interesting, but we may not forget that in general zeros and ones will be transmitted between sender and receiver. In order to ascertain the quality of the codes we will investigate in Section 2.2, we need a probabilistic model of the transmission channel. The following definition introduces the simplest channel model. We will introduce another channel model in Section 2.3.

Definition 2.3 (Binary symmetric channel).
A *transmission channel* – the medium over which data gets transmitted between sender and receiver – is called *binary symmetric channel* if it satisfies the following conditions:

BSC1 The transmitted data is in binary form, i.e. only zeros and ones get transmitted.

BSC2 The probability that a zero changes to a one equals the probability that a one changes to a zero. This probability is called *bit error probability*.

BSC3 Transmission errors are independent from each other.

A visualization of a binary symmetric channel with bit error probability p is given in Figure 2.2.

The transmission quality of binary symmetric channels can be measured by the bit error probability. In the worst case, we have $\mathbb{P}(\text{bit error}) = 0.5$, because this implies complete chaos: the bits arriving at the receiver are completely random. In general, we may assume that the bit error probability is at most 0.5, otherwise we can just "relabel" the incoming bits at the receiver and then obtain a channel with error rate less then 0.5.

Due to the symmetry and the independence in these binary channels, the bits behave like Bernoulli random variables in terms of incorrect transmission. This means that if a bit b gets transmitted correctly, the respective random variable of interest X_b is zero. Otherwise, if b changes its value

[1] Essentially, an encoding scheme in standard form adds a whole "redundancy block" to the processed messages.

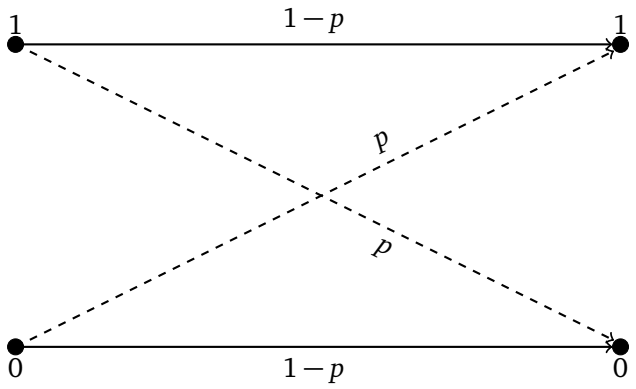

Figure 2.2: Binary symmetric channel

during the transmission, X_b equals to one. Constructing these "error indicating" random variables can be further exploited.

Let us assume that we have not only one, but n bits being transmitted in a block. The random variable X modeling the number of errors in these n bits is the sum of all error indicating random variables, which are stochastically independent due to **BSC3**. This means that X follows a binomial distribution with parameters n and p. We summarize these considerations in the following lemma.

Lemma 2.1 (Error probability in binary symmetric channels).
If a binary word of length n gets transmitted over a BSC with bit error probability p, then the number of bit errors within these n symbols follows a binomial distribution and we have

$$\mathbb{P}(r \text{ of } n \text{ bits incorrect}) = \binom{n}{r} \cdot p^r \cdot (1-p)^{n-r}.$$

In the following definition we will introduce a very broad class of codes which are the main focus of our further investigations.

Definition 2.4 (Linear code).
Given a q-ary code C of length n and dimension k, that is a code with encoding scheme $c : \mathrm{GF}(q)^k \to C \subseteq \mathrm{GF}(q)^n$. C is called *linear code* or *group code*, if the encoding scheme satisfies

$$c(m_1 + m_2) = c(m_1) + c(m_2) \quad \text{and} \quad c(\lambda \cdot m_1) = \lambda \cdot m_1$$

for all m_1, $m_2 \in \mathrm{GF}(q)^k$ and for all scalars $\lambda \in \mathrm{GF}(q)$.

Remark.
The additions of the vectors above are always componentwise in $\mathrm{GF}(q)$. The name *linear code* comes from the fact that the encoding scheme is a linear function – and the name *group code* comes from the fact that $(C, +)$ – the set of code words together with the code word addition – forms a group.

Finally, using a linear code, the zero message always gets encoded to the zero code word.

Before we go on and specify generation algorithms for linear codes we want to further characterize linear codes and begin analyzing error detection and error correction capabilities of given codes. An important tool for that task is the *Hamming distance*.

Definition 2.5 (Hamming distance, Hamming weight, minimum Hamming distance).
Let C be a q-ary code of length n and dimension k. Let $a,\ b \in C \subseteq \mathrm{GF}(q)^n$ be two code words. Then the Hamming distance $d(a,b)$ is defined as the sum of positions of a and b with different symbols from $\mathrm{GF}(q)$, i.e.

$$d(a,b) := \sum_{j=1}^{n} d(a_j, b_j), \text{ where } d(a_j,b_j) := \begin{cases} 0 & \text{if } a_i = b_i, \\ 1 & \text{otherwise.} \end{cases}$$

As can be checked easily, $d : \mathrm{GF}(q)^n \times \mathrm{GF}(q)^n \to \mathbb{N}_0$ is a *metric* on $\mathrm{GF}(q)^n$.

The *Hamming weight* of a code word is defined as the number of non-zero symbols in the code word. The Hamming weight is equivalent to the Hamming distance of the code word to the zero word, $h(a) := d(a,0)$.

The *minimum Hamming distance* $d_{\min}$ of a code C is the smallest distance two different code words have in C.

$$d_{\min} := \min_{\substack{a,b \in C \\ a \neq b}} d(a,b).$$

The Hamming distance plays an important part when trying to decode words and reconstruct the messages sent. The following theorem introduces a very central idea in coding theory.

Theorem 2.2 (Maximum likelihood decoding).
Given a q-ary code C of length n and dimension k over a BSC with bit error probability p. Let $c \to c'$ denote the event that the code word c gets transmitted to some word c' of length n. Assume that a received word has Hamming distance d_1 from the code word c_1 and d_2 from the code word c_2 with $d_1 \leq d_2$. Then

$$\mathbb{P}(c \to c_1) = p^{d_1} \cdot (1-p)^{n-d_1} \geq p^{d_2} \cdot (1-p)^{n-d_2} = \mathbb{P}(c \to c_2).$$

This means that it is more likely that *less* errors occurred during transmission. If we always try to decode an arbitrary word received to the nearest code word, then we follow the principle of *maximum likelihood decoding (MLD)*.

Remark.
As the code word with the least distance to the received word does not have to be uniquely determined, a strategy to resolve such a conflict is needed. Some approaches are simply guessing and randomly decoding to one of the candidates, or deleting the received word and requesting a new transmission. One might also delete the word if the distance to the next code word is higher than a predefined threshold.

These deletion methods lead (amongst others) to *soft decoding*. Simply trying to decode without any further precautions can be described as *hard decoding*. In the examples presented we generally use hard decoding algorithms, with the exception of Section 2.3, where we discuss an alternative decoding strategy.

Theorem 2.3 (Properties of the minimum Hamming distance).
Assume that a given linear q-ary code of length n and dimension k has minimum distance d. In

that case, C can detect every constellation of up to $d-1$ errors and correct every constellation of $t := \lfloor \frac{d-1}{2} \rfloor$ errors[2].

In Figure 2.3 the decoding situation is visualized. There are code words (blue dots) and non-code words (gray dots) in $\mathrm{GF}(q)^k$. Around every code word, a *covering radius* is drawn. Words inside this radius get decoded to the code word in the center by MLD. In general, words being in between two covering circles cannot be decoded uniquely by MLD. Additionally, a threshold θ can be defined, such that a received word gets deleted if it is too far away from any code word.

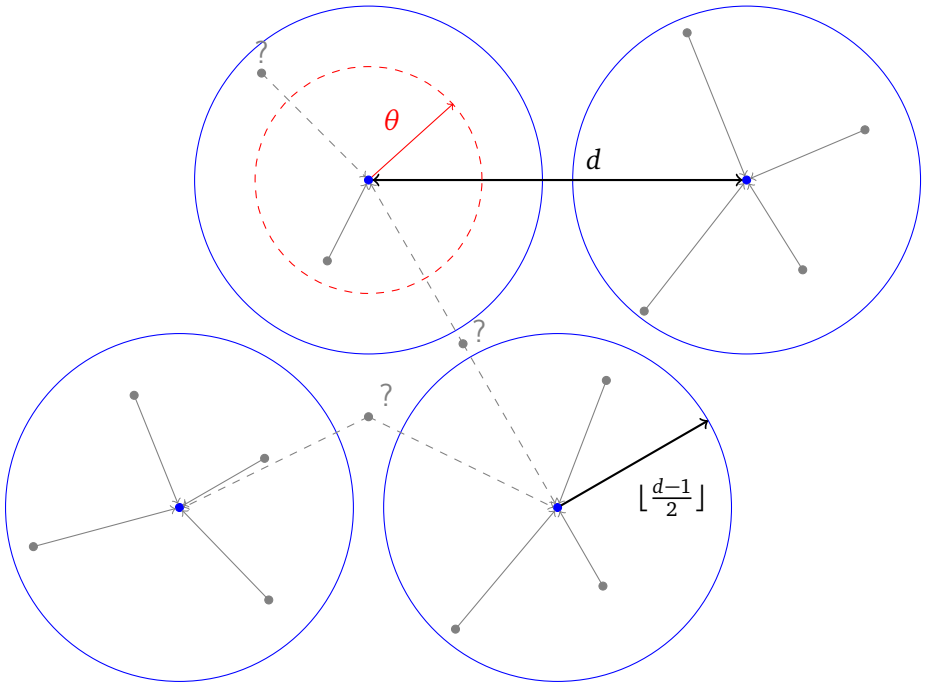

Figure 2.3: Error detection and correction – covering radius

Notation.
The abbreviations "q-ary $[n,k,d]$-code" or "$[n,k,d]_q$-code" stand for a linear q-ary code of length n, dimension k and minimum Hamming distance d.

If we are working with a linear code there are very efficient methods for characterizing and generating a linear code, as well as algorithms which check whether a given word is a code word. As might have been expected, it is rather inefficient for large codes to compare the received word to each word in the list and then decide whether we have received a code word or not.

One class of linear codes with efficient generation and check algorithm is the class of *polynomial codes*. It will be necessary to represent words over $\mathrm{GF}(q)$ as polynomials in $\mathrm{GF}(q)[x]$, i.e. the equivalency

$$m = a_{r-1}a_{r-2}\dots a_1a_0 \triangleq m(x) = a_{r-1}x^{r-1} + a_{r-2}x^{r-2} + \dots + a_1x + a_0$$

[2]This means that there are constellations of d errors which C does not detect and constellations of $t+1$ errors which C does not decode correctly.

as noted in the definition of polynomial rings, Definition 1.6, will be used frequently. If w is a word over $\mathrm{GF}(q)$, $w(x)$ denotes the related polynomial in $\mathrm{GF}(q)[x]$.

Definition 2.6 (Polynomial code).
A *polynomial code* or *polynomially generated code* is a q-ary block code of length n and dimension k encoding words from $\mathrm{GF}(q)^k$ with the *generator polynomial* $g \in \mathrm{GF}(q)[x]$ of degree $n-k$ when the encoding process is realized as in Algorithm 1.

Algorithm 1 Polynomial codes – encoding

1: **procedure** encode($m \in \mathrm{GF}(q)^k,\ g \in \mathrm{GF}(q)[x]$)
2: Transform m to polynomial form $m(x)$.
3: Shift $m(x)$ by multiplying x^{n-k}.
4: Find the remainder $r(x)$ of the shifted message divided by the generator polynomial, $m(x) \cdot x^{n-k}/g(x)$.
5: $c(x) \leftarrow m(x) \cdot x^{n-k} - r(x)$ and transform $c(x)$ to its vector form $c \in \mathrm{GF}(q)^n$.
6: **return** c.
7: **end procedure**

Remark.
Polynomial codes are in standard form, the encoding scheme adds redundancy bits on the right side of the message words.

Theorem 2.4 (Properties of polynomial codes).
Let C be a q-ary polynomial code of length n, dimension k and with generator polynomial $g \in \mathrm{GF}(q)[x]$. Then the following statements hold:

- C is a linear code.
- $c(x)$ is the polynomial of a code word in C if and only if $c(x)$ is a multiple of the generator polynomial $g(x)$.

Another class of practically relevant linear codes very much related to the class of polynomial codes are *cyclic codes*. They have very good parameters in terms of error detection and error correction and furthermore, they are relatively easy to realize on hardware.

Definition 2.7 (Cyclic linear codes).
A q-ary linear $[n,k,d]$-code is called *cyclic code* if for all code words $w = a_{n-1}a_{n-2}\ldots a_1a_0$, the cyclically permuted word $w' = a_{n-2}\ldots a_1a_0a_{n-1}$ is again a code word.

For the following results on cyclic linear codes we have to assume an additional restriction regarding the code length. Let $\gcd(n,q) = 1$.

Let (x^n-1) be the principal ideal generated by the polynomial x^n-1. All elements of the factor ring $\mathrm{GF}(q)[x]/(x^n-1)$ can be represented by polynomials of degree less than n. Every vector with n components from $\mathrm{GF}(q)$ corresponds to one of these polynomials. Therefore, we can identify the set of code words C with a set of code polynomials in $\mathrm{GF}(q)[x]/(x^n-1)$. The structure of this set yields a criterion in order to decide whether C is a cyclic code or not.

Theorem 2.5 (Characterisation of cyclic codes).
Given a linear q-ary $[n,k,d]$-code with $\gcd(n,q)=1$. Then C is a cyclic code if and only if the respective set of polynomials in $\mathrm{GF}(q)[x]/(x^n-1)$ is an ideal.

Proof. If the set of polynomials is an ideal and $a_{n-1}a_{n-2}\dots a_0 \in C$, then also

$$\begin{aligned} x\cdot(a_{n-1}\cdot x^{n-1}+\dots+a_1\cdot x+a_0) &= a_{n-1}\cdot x^n+\dots+a_1\cdot x^2+a_0\cdot x \\ &= a_{n-2}\cdot x^{n-1}+\dots+a_0\cdot x+a_{n-1} \triangleq a_{n-2}a_{n-2}\dots a_1a_0a_{n-1}\in C. \end{aligned}$$

On the other hand, if the cyclic permutations of a code word are in C, then this implies that for every code polynomial $a(x)$ the words $x\cdot a(x)$, $x^2\cdot a(x)$, ... are also code polynomials. Finally, due to linearity we know that $b(x)\cdot a(x)$ is a code polynomial for any polynomial b – which makes the set of code polynomials an ideal. □

Lemma 2.6.
Every ideal of $\mathrm{GF}(q)[x]/(x^n-1)$ is a principal ideal generated by the unique monic polynomial $g(x)$ of lowest degree in the ideal and $g(x)$ also divides x^n-1.

This means that a q-ary polynomial code of length n and dimension k is cyclic if and only if its generator polynomial g is a divisor of x^n-1.

We will come back to special polynomial codes like the BCH-code or the RS-code in Section 2.2. Before we discuss a practical, generally applicable decoding algorithm for linear codes, we will investigate another possibility to characterize linear codes: *generator matrices*. The idea behind the representation of linear codes by generator matrices is to write down the matrix representation of the linear encoding scheme with respect to the standard bases in $\mathrm{GF}(q)^k$ and $\mathrm{GF}(q)^n$.

Definition 2.8 (Generator matrix).
Given a linear q-ary $[n,k,d]$-code C with encoding scheme c. Let $e_i \in \mathrm{GF}(q)^k$ denote the message where all components are zero except for the i-th entry[3], which is 1. Then, the generator matrix $G \in \mathrm{GF}(q)^{k\times n}$ is the matrix

$$G = \begin{bmatrix} c(e_1) \\ \vdots \\ c(e_k) \end{bmatrix}.$$

If the code C is given in standard form, the the generator matrix has the form $G=[I_k \mid P]$, where $P \in \mathrm{GF}(q)^{k\times n-k}$.

Multiplying a message $m \in \mathrm{GF}(q)^k$ from the left side with the generator matrix yields the corresponding code word $c = c(m) \in \mathrm{GF}(q)^n$. Thus, the code is the row space of the generator matrix.

By exploiting the structure of the generator matrix, we can construct a simple criterion for checking whether a given word is a code word or not. To do so, we temporarily restrict ourselves to codes in standard form.

[3] In this context, the e_i are called *unit messages*.

Definition 2.9 (Check matrix).
Given a linear q-ary $[n,k,d]$-code C in standard form with generator matrix $G = [I_k \mid P] \in \mathrm{GF}(q)^{k\times n}$. Then the *check matrix* H of the code C is the $(n \times (n-k))$-matrix defined by

$$H := \begin{bmatrix} -P \\ I_{n-k} \end{bmatrix}.$$

Lemma 2.7.
If H is a check matrix for some q-ary linear code of length n and dimension k, then the code C consists of all words $v \in \mathrm{GF}(q)^n$ for which $v \cdot H = 0$ holds. Also, the product of the generator matrix and the check matrix is the zero matrix.

A check matrix can also be constructed if the respective code is not in standard form, as we know that the code C has to be the kernel of the matrix H by Lemma 2.7.

Definition 2.10 (Equivalent codes).
Two q-ary linear codes C and C' of length n and dimension k are called equivalent if the generator matrix G' of C' can be formed out of the generator matrix G of C by using the following elementary operations:

(a) swapping two columns,

(b) multiplying a column with a scalar $\lambda \neq 0$,

(c) swapping two rows,

(d) multiplying a row with a scalar $\lambda \neq 0$,

(e) adding a scaled row (i.e. a row multiplied with a scalar) to another row.

Due to the fact that the encoding scheme is an injective function, we know that the respective generator matrix G has full row rank. Then, using another result from linear algebra we can prove the following theorem.

Theorem 2.8.
Each linear $[n,k,d]_q$-code C is equivalent to a linear $[n,k,d]_q$-code C' in standard form.

An alternative to the construction of the check matrix via the known kernel is by using the above stated results on equivalent codes. By modifying the check matrix of the respective equivalent code in standard form, the check matrix of the given code can be obtained.

Sometimes a linear code generated by certain parameters (cf. Section 2.2, e.g. BCH and RS codes) has inconvenient length. Removing some symbols from the code words leads to the concept of *shortened codes*.

Definition 2.11 (Shortened linear codes).
Let C be a linear $[n,k,d]_q$-code with generator matrix $G \in \mathrm{GF}(q)^{k\times n}$. If the matrix $G' \in \mathrm{GF}(q)^{k\times n'}$ – obtained from G by deleting some columns – has full row rank, then the linear $[n',k,d']_q$-code C' generated by G' is called *shortened code*.

The minimum Hamming distance of a shortened code with $n' = n-1$, i.e. the generator matrix

of the shortened code can be obtained by deleting one column in the original generator matrix, is usually $d-1$. The minimum Hamming distance does not change if, for example, the column deleted was a zero column.

Shortened codes are used in many applications like for example on compact discs, where shortened Reed Solomon codes are applied.

For the remainder of this section, we will concentrate on general decoding algorithms for linear codes.

Let C be a linear q-ary code of length n and dimension k. Assume that a message $m \in \mathrm{GF}(q)^k$ gets encoded by C to a code word c. Let the (erroneous) word we get by transmitting c over some transmission channel be denoted by c'. Then the *error pattern* is defined as $e := c' - c$. This means that we model transmission by adding an error pattern to the code word, $c' = c + e$.

By *brute force decoding* we decode c' to $\operatorname{argmin}_{u \in C} d(c', u)$. Brute force decoding is *very* inefficient for large k, as we have to compute the Hamming distance to all q^k code words.

In order to efficiently decode a received word, we want to evaluate e from c'. Note that the code words form – by the definition of linear codes – a group regarding code word addition. Due to properties of the group C, we know that $C = -C$, and therefore we can investigate the cosets $c' + C$, because $e = c' - c \in c' - C = c' + C$, meaning that the error vector e always is within the coset $c' + C$. By maximum likelihood decoding we are looking for the word in $c' + C$ with minimum weight – which is called the *coset leader*. Algorithm 2 describes the procedure from this paragraph – the so-called *coset search* – in detail. Coset search still is rather inefficient for large codes.

Algorithm 2 Coset search decoding

```
1:  procedure decode(linear [n,k,d]_q-code C, w ∈ GF(q)^n)
2:      M ← w + C
3:      min ← n,  e ← (0,0,...,0)
4:      while M ≠ ∅ do
5:          w ∈ M
6:          if h(w) ≤ min then
7:              min ← h(w),  e ← w
8:          end if
9:          M ← M \ {w}
10:     end while
11:     return w − e
12: end procedure
```

However, coset search can easily be modified to be more efficient. At this point we will make use of the *check matrix* again.

Definition 2.12 (Syndrome).
Let C be a linear q-ary code of length n, dimension k and check matrix H. Then the *syndrome* or *C-syndrome* of a word $w \in \mathrm{GF}(q)^n$ is given by $w \cdot H$.

Due to Lemma 2.7, a word is a code word if and only if its syndrome is 0.

If $w \in c' + C$, we have $w = c' + u$ for some code word u. Therefore,

$$w \cdot H = (c' + u) \cdot H = c' \cdot H = (e + c) \cdot H = e \cdot H$$

holds, that is words within the same coset have the same syndrome. Furthermore, if two code words have the same syndrome, they have to be from the same coset. This leads us to the *syndrome decoding* algorithm, as described in Algorithm 3. Coset search and syndrome decoding are very much related, thus syndrome decoding also belongs to the family of maximum likelihood decoding algorithms.

Algorithm 3 Syndrome decoding

```
1:  procedure initialization(linear [n,k,d]_q-code C, check matrix H, w ∈ GF(q)^n)
2:      cosetLeader ← ∅                ▷ cosetLeader is a function, whose images will be set below.
3:      j ← 0, M_j ← {w ∈ GF(q)^n | h(w) = j}
4:      while |im(cosetLeader)| < q^{n−k} do                ▷ Coset leader generation.
5:          if M_j = ∅ then
6:              j ← j + 1, M_j ← {w ∈ GF(q)^n | h(w) = j}
7:          end if
8:          Choose e ∈ M_j
9:          if cosetLeader(e · H) is undefined then
10:             cosetLeader(e · H) := e          ▷ If e · H has no value assigned yet, assign e.
11:         end if
12:         M_j ← M_j \ {e}
13:     end while
14:     return cosetLeader
15: end procedure
16: procedure decode(cosetLeader, check matrix H, w ∈ GF(q)^n)
17:     return w − cosetLeader(w · H)
18: end procedure
```

Remark.
Note that even though syndrome decoding is more efficient than simple hard decoding using brute force or coset search, q^{n-k} syndromes have to be computed and stored. This number obviously grows exponentially with the number of redundancy symbols added. For special types of codes there are special decoding algorithms, like the *Berlekamp-Massey* algorithm for Reed-Solomon and BCH codes. We refer to [6] for detailed information on some specialized decoding algorithms.

2.2 Selected linear codes and their properties

The first two classes of linear codes we want to introduce are very simple codes: the class of *repetition codes* and the class of *parity check codes*.

Definition 2.13 (Repetition codes).
Repetition codes of order t are linear q-ary $[t \cdot k, k, t]$-codes whose generator matrix G consists of the identity matrix I_k repeated t times, i.e. $G = [I_k \mid I_k \mid \cdots \mid I_k]$.

In terms of error correction capability, repetition codes are not very good. Their *information rate*,

i.e. the code dimension divided by the code length, is $1/t$, which is quite low for a code with minimum Hamming distance t.

Definition 2.14 (Parity check codes).
Parity check codes are linear q-ary $[k+1, k, 2]$-codes where the encoding scheme is given by

$$c(m_1, m_2, \ldots, m_k) = \left(m_1, m_2, \ldots, m_k, \sum_{j=1}^{k} m_j \right)$$

Parity check codes cannot correct errors. However, they are very easy to implement and they do have some nice properties. For example, binary parity check codes are able to detect whether an odd number of errors occurred.

Definition 2.15.
A linear binary code C is called *Hamming code* of order m, if its length is $n = 2^m - 1$, its dimension is $2^m - m - 1$ and the rows of the $(2^m - 1) \times m$ check matrix are the binary representations of the integers $1, 2, \ldots, 2^m - 1$. Adding a parity check column to the generator matrix of a Hamming code results in the generator matrix for an *extended Hamming code*.

Example 2.1.
The generator matrices for the binary Hamming code of order 3 and the extended binary Hamming code of order 3 are given below:

$$G = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right], \quad G' = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}\right].$$

Lemma 2.9 (Minimum distance of Hamming codes).
The minimum distance for any linear binary Hamming code is 3. The minimum distance for any extended linear binary Hamming code is 4.

Therefore, Hamming codes and extended Hamming codes can correct one error and detect 2 and 3 errors, respectively.

Remark.
Linear binary Hamming codes are *perfect codes*, i.e. codes which satisfy the *Hamming bound* for linear q-ary $[n, k, d]$-codes C capable of correcting t errors,

$$|C| \cdot \sum_{j=0}^{t} \binom{n}{j} (q-1)^j \leq q^n$$

with equality.

Perfect codes have the very nice property that for each word w there is exactly one code word with Hamming distance at most d from w. Therefore, every word can be uniquely decoded by maximum likelihood decoding.

The next code we want to introduce – the so-called (extended) binary *Golay code* – was used in the Voyager spacecraft program in the early 1980's.

Definition 2.16 (Extended linear binary Golay code).
Let $P \in \mathrm{GF}(2)^{12\times 12}$ be the matrix

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

Then the binary linear $[24, 12, 8]$-code with generator matrix $G = [I_{12} \mid P]$ is called *extended linear binary Golay code*.

The extended linear binary Golay code has a minimum Hamming distance of 8, and thus is able to detect and correct up to 7 and up to 3 errors, respectively.

The linear binary Golay code can be defined as a shortened version of the extended linear binary Golay code.

Definition 2.17 (Linear binary Golay code).
The *linear binary Golay code* is a $[23, 12, 7]$-code where the generator matrix can be obtained by deleting the last column of the extended linear binary Golay code.

Remark.
Both versions of the Golay code have some interesting properties. The linear binary Golay code is a *perfect code*, being able to correct 3 errors and detecting up to 6 errors.

The extended linear binary Golay code has some deeper algebraic background which we will not get into detail here. However, let it be said that a very efficient decoding algorithm of this code can be constructed, see [6] for details.

The next class of codes we want to introduce is the class of *Reed-Muller codes*. Actually, Reed-Muller codes are a generalization of extended Hamming-Codes, as these are included in the class of Reed-Muller codes.

Definition 2.18 (Reed-Muller codes).
The Reed-Muller code with length parameter m and order r is a binary $[2^m, k, 2^{m-r}]$-code with $k := \sum_{j=0}^{r} \binom{m}{j}$. Let us define the *wedge product* of two vectors $a,\ b \in \mathrm{GF}(2)^n$ as

$$a \wedge b := (a_1 \cdot b_1, a_2 \cdot b_2, \ldots, a_n \cdot b_n).$$

This way, a generator matrix for the Reed-Muller code can be obtained from Algorithm 4.

Algorithm 4 Reed-Muller code generator matrix

procedure getGeneratorMatrix($r, m \in \mathbb{N}$)
  $n \leftarrow 2^m$
  $v_0 \leftarrow (1, 1, \ldots, 1) \in \mathrm{GF}(2)^n$ ▷ First row of generator matrix G.
  Let M_1 be a $m \times 2^m$ matrix with the binary representations of $0, 1, \ldots, 2^m - 1$ as columns.
  Let $v_1, v_2, \ldots, v_m$ be the rows of M_1. ▷ Rows 1 to m from generator matrix G.
  $S \leftarrow \{1, 2, \ldots, m\}$
  $j \leftarrow 2$
  while $j \leq r$ **do**
    Add the vector $\bigwedge_{k \in S_j} v_k$ for all $S_j \subseteq S$, $|S_j| = j$ to G.
  end while
  return G
end procedure

Example 2.2.
The Reed-Muller code with length parameter $m = 4$ and order $r = 2$ is a binary $[16, 11, 4]$-code with generator matrix

$$G = \left[\begin{array}{cccccccccccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right].$$

Remark.
Note that for Reed-Muller codes of order $r = 1$, there is a fast and efficient decoding algorithm based on the *Fast Hadamard Transform* (cf. [6, p. 88 ff.]).

There are two more codes we want to introduce: *Bose-Chaudhuri-Hocquenghem codes*, i.e. *BCH codes*, and *Reed-Solomon codes*. Reed-Solomon codes were investigated before the BCH codes, and later it was found that Reed-Solomon codes actually are included in the class of BCH codes. Note that for these codes there are very specialized decoding algorithms which we will not go into detail here (cf. [7, p. 328 ff.], [6, p. 135 ff.]).

Definition 2.19 (BCH codes, RS codes).
Let b and n be nonnegative integers and let n and q be coprime. Furthermore, let $\alpha \in \mathrm{GF}(q^m)$ be a primitive n-th root of unity, where m is the multiplicative order of q modulo n. A *BCH code*

over $\mathrm{GF}(q)$ of length n and *designed distance* d where $2 \leq d \leq n$ is a cyclic code defined by the roots

$$\alpha^b, \alpha^{b+1}, \ldots, \alpha^{b+d-2}$$

of the generator polynomial.

If $m_\alpha^{(i)}(x) \in \mathrm{GF}(q)[x]$ denotes the minimal polynomial of α^i over $\mathrm{GF}(q)$, then the generator polynomial $g(x)$ of a BCH code is of the form

$$g(x) := \mathrm{lcm}(m_\alpha^{(b)}(x), m_\alpha^{(b+1)}(x), \ldots, m_\alpha^{(b+d-2)}(x)).$$

Also, there are some special cases:

- When choosing $b = 1$, then the constructed BCH codes are called *narrow-sense* BCH codes.
- If $n = q^m - 1$, the corresponding BCH codes are called *primitive*.
- For $n = q - 1$, BCH codes of length n over $\mathrm{GF}(q)$ are called *Reed-Solomon codes*.

After all parameters are set and the generator polynomial is found, encoding takes place as described in Algorithm 1 and the paragraphs about cyclic codes. The following theorem justifies the term *designed distance*.

Theorem 2.10.
The minimum distance of a BCH code of designed distance d is at least d.

We will give two examples, an ordinary BCH code and a RS code to visualize the construction process.

Example 2.3.
We want to construct a primitive, narrow-sense BCH code with $q = 2$, $n = 15$ and designed distance $d = 7$. Obviously, we also have $m = 4$, thus we are looking for a generating element α in $\mathrm{GF}(2^4)$. As $x^4 + x + 1$ is irreducible over $\mathrm{GF}(2)$ and has degree 4, the element α in $\mathrm{GF}(2^4)$ with $\alpha^4 + \alpha + 1 = 0$ is such an element.

Due to characteristic 2 we know that $p(x)^2 = p(x^2)$, and thus we only need to compute the least common multiple of the minimal polynomials for the odd powers of α, i.e. $\alpha, \alpha^3, \alpha^5$.

As the polynomial $x^4 + x + 1$ is irreducible, we already know $m_\alpha^{(1)}(x) = x^4 + x + 1$.

The minimal polynomial of α^3 also needs to have the roots

$$(\alpha^3)^2 = \alpha^6,\ (\alpha^6)^2 = \alpha^{12},\ (\alpha^{12})^2 = \alpha^{24} = \alpha^{15} \cdot \alpha^9 = \alpha^9,\ (\alpha^9)^2 = \alpha^{18} = \alpha^3.$$

Therefore,

$$m_\alpha^{(3)}(x) = (x - \alpha^3)(x - \alpha^6)(x - \alpha^{12})(x - \alpha^9) = x^4 + x^3 + x^2 + x + 1.$$

Finally, $m_\alpha^{(5)}$ needs to have the roots

$$(\alpha^5)^2 = \alpha^{10},\ (\alpha^{10})^2 = \alpha^{20} = \alpha^5$$

we have $m_\alpha^{(5)}(x) = (x - \alpha^5)(x - \alpha^{10}) = x^2 + x + 1$.

Then, the generator polynomial of the BCH code is given by

$$g(x) = \mathrm{lcm}(m_\alpha^{(1)}(x), m_\alpha^{(3)}(x), m_\alpha^{(5)}(x)) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1.$$

Example 2.4.
Again, we want to stay within $\mathrm{GF}(2^4)$ and construct a Reed-Solomon code, meaning that $q = 16$. This forces the length of the code to be 15. Furthermore, we want our code to be able to correct 3 errors, thus we choose the designed distance to be 7. Also, we choose $b = 1$ in Definition 2.19.

Furthermore, we assume β to be a generating element of $\mathrm{GF}(2^4)$, meaning that the field is given by $\mathrm{GF}(2^4) := \{0, 1, \beta, \beta^2, \ldots, \beta^{14}\}$. Also, let us assume that β is a root of the irreducible polynomial $x^4 + x + 1$.

Note that this code actually is a code over $\mathrm{GF}(2^4)$. This means that if we want to use it in standard applications, we have to think about how to represent these elements in binary form – as the standard transmission channels are generally symmetric binary channels. We will use the fact that every element in $\mathrm{GF}(2^4)$ can be represented as a polynomial with degree up to 3 in $\mathrm{GF}(2)[x]$. Thus, we obtain the following "translation table"[4]:

0	$\triangleq 0000$	β^3	$\triangleq 1000$	$\beta^7 = \beta^3 + \beta + 1$	$\triangleq 1011$	$\beta^{11} = \beta^3 + \beta^2 + \beta$	$\triangleq 1110$
1	$\triangleq 0001$	$\beta^4 = \beta + 1$	$\triangleq 0011$	$\beta^8 = \beta^2 + 1$	$\triangleq 0101$	$\beta^{12} = \beta^3 + \beta^2 + \beta + 1$	$\triangleq 1111$
β	$\triangleq 0010$	$\beta^5 = \beta^2 + \beta$	$\triangleq 0110$	$\beta^9 = \beta^3 + \beta$	$\triangleq 1010$	$\beta^{13} = \beta^3 + \beta^2 + 1$	$\triangleq 1101$
β^2	$\triangleq 0100$	$\beta^6 = \beta^3 + \beta^2$	$\triangleq 1100$	$\beta^{10} = \beta^3 + \beta + 1$	$\triangleq 1011$	$\beta^{14} = \beta^3 + 1$	$\triangleq 1001$

Minimal polynomials of elements β^j in $\mathrm{GF}(2^4)$ are linear factors, they have the form $(x - \beta^j)$. Therefore, computing the generating polynomial of the desired Reed-Solomon code is very easy, we have

$$\begin{aligned} g(x) &= (x-\beta)\cdot(x-\beta^2)\cdot(x-\beta^3)\cdot(x-\beta^4)\cdot(x-\beta^5)\cdot(x-\beta^6) \\ &= x^6 + \beta^{10}\cdot x^5 + \beta^{14}\cdot x^4 + \beta^4\cdot x^3 + \beta^6\cdot x^2 + \beta^9\cdot x + \beta^6. \end{aligned}$$

The encoding process then is the same as with BCH codes described above.

2.3 Soft decoding over a different channel model

In this section we want to discuss an alternative channel model. The *binary symmetric parallel channel* is a generalization of the symmetric parallel channel from Definition 2.3.

Definition 2.20 (Binary symmetric parallel channel).
Assume there are n labelled and ordered binary symmetric channels with respective error probabilities p_j, $j \in \{1, 2, \ldots, n\}$ between a sender A and a receiver B, such that if A sends a bit over channel j, B knows that the information received belongs to position j.

Then we call the transmission channel obtained by the (ordered) union of these n binary symmetric channels a *binary symmetric parallel channel* of capacity n.

Remark.
In Figure 2.4, the Binary symmetric parallel channel is visualized. Note that it is a characteristic property of this channel type to have several different error probabilities. Assuming that $p_1 = p_2 = \cdots = p_n$ leads back to the binary symmetric channel model, because there is no difference

[4] Reed-Solomon codes are frequently used when so-called *burst errors* are likely to appear over a communication channel, as they treat a group of bits as one symbol. Correcting one symbol therefore corrects a whole bit group.

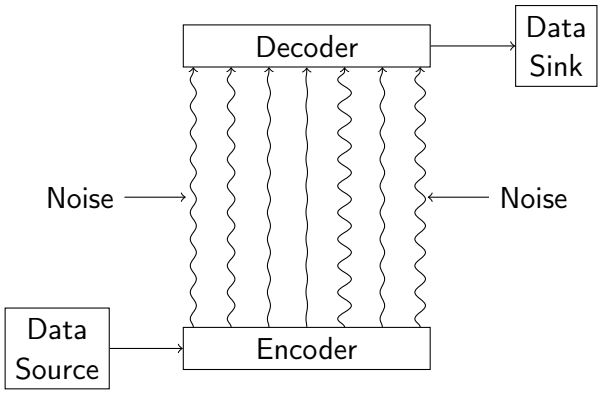

Figure 2.4: Binary symmetric parallel channel

whether a word gets transmitted bitwise over different channels with the same error probability or whether the whole word just gets transmitted over one binary symmetric channel.

Remark (Motivation).
The reason for us investigating this rather strange transmission channel model is that it appears when analyzing the behavior of uninitialized SRAM-cells with respect to their stability (meaning whether they tend to assume state 0 or 1 on initialization). This is one of the problems we were confronted with in the *CODES*-project.

Before we begin our investigations and derive some elementary properties of this channel model, we want to specify how we will model the error probabilities p_j.

Imagine that before using the channel, sender A and receiver B agree to perform some sort of *enrollment phase*. First, they specify an error correcting code to be used during the transmission procedure. Then, A sends the zero word to B repeatedly[5]. Thus, B is able to estimate the probability $p_j = \mathbb{P}(j\text{-th position is a } 1)$. Then B *shortens* the error correcting code by ignoring or deleting the r bits transmitted over the binary channels with the r highest estimated error probabilities and thus (ideally) decreases the decoding error probability.

A suitable distribution model for the error probabilities p_j is a modified Beta distribution. In general, the density of a Beta distribution with parameters α and β is

$$f(x) = \frac{\mathbb{1}_{[0,1]}(x)}{B(\alpha,\beta)} \cdot x^{\alpha-1} \cdot (1-x)^{\beta-1},$$

where $B(\alpha,\beta)$ is the Beta function. There are two modifications we will apply to this model: firstly, we want the density to be concentrated on $[0, 0.5]$ which we can achieve by the transformation $x \mapsto 2x$. And secondly, we want to eliminate the influence the term $(1-x)^{\beta-1}$ has on the density, which we will do by setting $\beta = 1$. If $\beta < 1$, the density increases again around 0.5, which increases the probability of having a channel with very high error probability. We want to exclude this case in our investigations. By setting $\beta > 1$, the behavior of the density near 0.5 can be adjusted further – which we will refrain from in our deliberations.

Also, we want to choose α between 0 and 1. Otherwise, the density near 0 is small – which we do not want.

[5] In this context, the number of zero words sent in the enrollment phase will be called *degree of enrollment*.

These modifications lead to the following distribution[6]: $X \sim \mathrm{Be}(\alpha, 1; 0, 0.5)$ with density

$$f_X(x) = 2^\alpha \cdot \frac{\mathbb{1}_{[0,0.5]}(x)}{B(\alpha,1)} \cdot x^{\alpha-1}.$$

This can be simplified further. Note that $B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$ and $\Gamma(\alpha+1) = \Gamma(\alpha)\cdot\alpha$. Thus, we can rewrite the density to

$$f_X(x) = \mathbb{1}_{[0,0.5]}(x) \cdot 2^\alpha \cdot \alpha \cdot x^{\alpha-1}. \tag{2.1}$$

Integration yields the cumulative density function,

$$\mathbb{P}(X \le x) = F_X(x) = \begin{cases} 0 & \text{for } x < 0 \\ (2x)^\alpha & \text{for } 0 \le x \le 0.5 \\ 1 & \text{for } x > 0.5 \end{cases} \tag{2.2}$$

Now, assume we have a binary symmetric parallel channel of capacity n with p_1, p_2, …, p_n being realizations of a random variable $P \sim \mathrm{Be}(\alpha, 1; 0, 0.5)$. We want to investigate how the mean error rate $p := \frac{1}{n} \cdot \sum_{j=1}^{n} p_j$ is related to the parameter α of the modified Beta distribution.

By the method of moments, p is an estimator for the mean of P. We can use this knowledge for two purposes: on the one hand, we can estimate the expected bit error rate for known α, and on the other hand, we can construct an estimator $\hat{\alpha}$ for known bit error rate p. The expected value of P is given by

$$\mathbb{E}P = \int_{-\infty}^{\infty} x \cdot f(x)\, dx = \frac{\alpha}{2\cdot(\alpha+1)}.$$

By setting $\mathbb{E}P = p$, we can construct the following estimator:

$$\hat{\alpha} = \frac{2p}{1-2p}. \tag{2.3}$$

The enrollment phase of the soft decoding algorithm we propose is described in Algorithm 5. After completing the enrollment of the transmission channel, sender and receiver shall use standard maximum likelihood decoding over the unmarked channels. With our modified Beta distribution, we are able to do some simulations to compare the performance of this soft decoding strategy against standard maximum likelihood decoding over binary symmetric parallel channels.

Algorithm 5 Soft decoding over binary symmetric parallel channels – enrollment.

1: **procedure** enrollment($[n,k,d]$-code C, $l \in \mathbb{N}$, $1 \le r < d$)
2: Sender A transmits the zero word $0 \in \mathrm{GF}(2)^n$ l times to receiver B.
3: B counts the number of ones (which is the number of errors) c_j received over channel j.
4: B marks the r transmission channels $j_1, \ldots, j_r$ where the most errors occurred.
5: **return** List of marked transmission channels $(j_1, \ldots, j_r)$ to be ignored during decoding.
6: **end procedure**

Remark.
The degree of enrollment l from Algorithm 5 controls the precision with which the transmission

[6] $\mathrm{Be}(\alpha, 1; 0, 0.5)$ denotes a truncated Beta distribution with unspecified parameter α and fixed parameter $\beta = 1$ which is concentrated on the interval $[0, 0.5]$.

channels with high error probability are identified. Due to the law of large numbers, c_j/l converges to p_j almost surely for $l \to \infty$.
Furthermore, as shortening a linear code by one bit also possibly reduces the minimum Hamming distance of the code by one, we will refrain from choosing r too large, which would strongly reduce the error correction capabilities of the original code.

The following example shall demonstrate the performance of the described soft decoding algorithm. The simulations are done with Sage using the code snippets included in the appendix in Section A.3.

Example 2.5.
Consider the Reed-Muller code of length 2^4 and order 1, which is a $[16, 5, 8]$ linear binary code. The code words are transmitted between sender and receiver via a binary symmetric parallel channel. We will analyze two cases where the average bit error rate over all channels is given by 0.08 and 0.12, respectively. The degree of enrollment is fixed at 300 evaluations, and $r \in \{0, 1, 2, 3\}$ (number of ignored channels).

By simulation (10000 evaluations per simulated parallel channel, 1000 simulated parallel channels – overall: 10^7 evaluations for each of the 8 cases, $r = 0$ marks simple hard decoding) on basis of the Beta distribution as described above we find the following results:

p_b	r	% inc. dec.	% inc. dec. e_3	% inc. dec. e_4	% inc. dec. e_5	% inc. dec. e_6	% inc. dec. e_7
0.08	0	6.47	0	52.32	89.99	99.68	100
	1	3.90	0	18.83	75.65	99.33	100
	2	3.33	2.24	14.76	49.23	93.12	100
	3	2.43	1.66	10.36	34.50	72.46	99.31
0.12	0	19.88	0	51.40	89.34	99.64	100
	1	14.74	0	25.53	78.16	99.29	100
	2	13.16	3.85	21.23	57.78	93.64	100
	3	11.31	3.85	18.13	46.47	79.93	99.25

In this table, "% inc. dec." denotes the percentage of incorrectly decoded words, and with the suffix e_j the percentage of words is denoted which were decoded incorrectly after j errors were applied to a code word. After applying more than 7 errors to a code word, all words are incorrectly decoded by every decoder investigated. From our simulation it is clear that the algorithm definitely does improve the data correction capabilities of the underlying Reed-Muller code. Nevertheless, the numbers show that not too many binary channels should be ignored, as for $r = 2$ and $r = 3$ decoding errors may occur, even though "only" three errors were applied to the code word. In the case of $p_b = 0.08$, we are able to reduce the overall decoding error rate from 6.47 %, which may be observed when using hard decoding, to 2.43 % when ignoring $r = 3$ binary symmetric channels. For $p_b = 0.12$, we can reduce the error rate from 19.88 % (hard decoding) to 11.31 % ($r = 3$).

3 Code concatenations

3.1 Definition and properties

When David Forney was looking for a class of (linear) codes with polynomial decoding time complexity and exponentially decreasing error probability (for increasing code length) in 1965, his search lead him to the concatenation of linear codes as described in [4]. The general idea is to have one binary[1] code (inner code) of dimension k and length n, and a linear code over $\mathrm{GF}(2^k)$, the "outer code", with dimension K and length N.

If we want to encode some bitstring, we accumulate $k \cdot K$ bits and transform them into K symbols in $\mathrm{GF}(2^k)$. Then, applying the outer code yields N symbols, transferable into $k \cdot N$ bits. And finally, we apply the inner code to each group of k bits, yielding n bits each and $n \cdot N$ bits overall. Formally this leads to the following definition:

Definition 3.1 (Galois concatenation).
Given a linear binary $[n,k,d]$-code C_{in} and a linear $[N,K,D]$-code C_{out} over $\mathrm{GF}(2^k)$. Then, the code $C := C_{\text{out}} \circ C_{\text{in}}$ is called (Galois) concatenated code with length $n \cdot N$ and dimension $k \cdot K$, if C maps messages to code words as described in Algorithm 6.

Algorithm 6 Galois code concatenation – encoding

1: **procedure** encode($m \in \mathrm{GF}(2)^{k \cdot K}$, enc_{in}, $\mathrm{enc}_{\text{out}}$)
2: Form K symbols $a_j \in \mathrm{GF}(2^k)$ from the $k \cdot K$ input bits
3: $\mathrm{GF}(2^k)^N \ni b = (b_1, \ldots, b_N) \leftarrow \mathrm{enc}_{\text{out}}(a_1, a_2, \ldots, a_K)$ ▷ Encode with C_{out}.
4: Form $(\tilde{b}_1, \ldots, \tilde{b}_N) = \tilde{b} \in \mathrm{GF}(2)^{k \cdot N}$ from b ▷ $\tilde{b}_j$ is b_j in binary representation
5: $c \leftarrow (\mathrm{enc}_{\text{in}}(\tilde{b}_1), \ldots, \mathrm{enc}_{\text{in}}(\tilde{b}_N)) \in \mathrm{GF}(2)^{n \cdot N}$ ▷ Encode with C_{in} and concatenate
6: **return** c
7: **end procedure**

However, in this section we want to consider another possibility of concatenation, which does not force us to work over an extension field like $\mathrm{GF}(2^k)$. Results for the "classical" Galois concatenation can be found in [4] and [9].

Definition 3.2 (Binary Concatenation).
Given a linear binary $[n,k,d]$-code C_{in} (*inner code*) and a linear binary $[N,K,D]$-code C_{out} (*outer code*). Let r denote the least common multiple of N and k and let $r = t \cdot k$. Then, we will call

[1] However, the inner code does not have to be binary; the idea can be generalized to a code over an arbitrary alphabet A.

the code $C := C_{\text{out}} \tilde{\circ} C_{\text{in}}$ with length $t \cdot n$ and dimension $\frac{r}{N} \cdot K$ *(binary) concatenation* of C_{out} and C_{in}, if C maps messages to code words as described in Algorithm 7.

Algorithm 7 Binary code concatenation – encoding

1: **procedure** encode($m \in \mathrm{GF}(2)^{\frac{r}{N} \cdot K}$, enc_{in}, $\mathrm{enc}_{\text{out}}$)
2: $\quad \mathrm{GF}(2)^r \ni \tilde{c} \leftarrow (\mathrm{enc}_{\text{out}}(m^{(1)}), \ldots, \mathrm{enc}_{\text{out}}(m^{(\frac{r}{N})}))$ ▷ Encode each group of K bits with C_{out}
3: $\quad \mathrm{GF}(2)^{t \cdot n} \ni c \leftarrow (\mathrm{enc}_{\text{in}}(\tilde{c}^{(1)}), \ldots, \mathrm{enc}_{\text{in}}(\tilde{c}^{(t)}))$ ▷ Encode each group of k bits with C_{in}
4: $\quad$ **return** c
5: **end procedure**

As can be seen from the definitions, the main difference between the *Galois concatenation* and the *binary concatenation* is that the first option forces us to work over an extension field. Otherwise the algorithms are quite similar.

On the following pages, we want to give some theoretical results regarding the binary concatenation of linear codes.

Lemma 3.1 (Linearity and generator matrix – binary concatenation).
Given a binary concatenated code $C = C_{\text{out}} \tilde{\circ} C_{\text{in}}$ with the same notation as in Definition 3.2. Assume $\mathrm{lcm}(N, k) = N = t \cdot k$, let $G_{\text{in}} \in \mathrm{GF}(2)^{k \times n}$ be the generator matrix of the inner code and let

$$\mathrm{GF}(2)^{K \times N} = \mathrm{GF}(2)^{K \times t \cdot k} \ni G_{\text{out}} = (G_{\text{out}}^{(1)} \mid G_{\text{out}}^{(2)} \mid \cdots \mid G_{\text{out}}^{(t)}), \quad G_{\text{out}}^{(j)} \in \mathrm{GF}(2)^{K \times k},$$

with $j \in \{1, 2, \ldots, t\}$, be the generator matrix of the outer code. Then C is a linear binary code and its generator matrix G is given by

$$G = (G_{\text{out}}^{(1)} \cdot G_{\text{in}} \mid G_{\text{out}}^{(2)} \cdot G_{\text{in}} \mid \cdots \mid G_{\text{out}}^{(t)} \cdot G_{\text{in}}).$$

Remark.
The assumption $\mathrm{lcm}(N, k) = N$ is *not* restrictive:

Recall the encoding algorithm for such a concatenation as described in Algorithm 7. Without loss of generality we can assume that $\mathrm{lcm}(N, k) = N$, otherwise we can design a new outer code C'_{out} with generator matrix

$$G'_{\text{out}} = \mathrm{diag}(G_{\text{out}})_{j=1}^{r/N},$$

which is the "old" generator matrix stacked r/N times on the main diagonal. Then we have $G'_{\text{out}} \in \mathrm{GF}(2)^{r/N \cdot K \times r}$ and the "new" outer code is a linear binary code of length r and dimension $r/N \cdot K$. This procedure formally corresponds to the process of accumulating "sufficiently" many bits before decoding.

Proof of Lemma 3.1. We will show that there is a generator matrix for the concatenated code. Then the linearity is obvious.

Now, under the assumption above we encode K bits into $N = t \cdot k$ bits and then each block of k bits into a block of n bits, resulting in $t \cdot n$ bits overall. Obviously, we receive the j-th block of k bits we need for the second encoding procedure by selecting columns $(j-1) \cdot k + 1$ to $j \cdot k$ from the generator matrix of the outer code. These submatrices are exactly the $G_{\text{out}}^{(j)}$. As we encode

$$m \mapsto m \cdot G_{\text{out}}^{(j)} = \tilde{c}_j \mapsto \tilde{c}_j \cdot G_{\text{in}} = c_j$$

we have

$$c_j = m \cdot G_{\text{out}}^{(j)} \cdot G_{\text{in}}. \tag{3.1}$$

To obtain the code word of m with respect to the concatenated code C we concatenate the words c_j. Finally, we have

$$\begin{aligned} m \mapsto (c_1, c_2, \ldots, c_t) &= \left(m \cdot G_{\text{out}}^{(1)} \cdot G_{\text{in}}, m \cdot G_{\text{out}}^{(2)} \cdot G_{\text{in}}, \ldots, m \cdot G_{\text{out}}^{(t)} \cdot G_{\text{in}}\right) \\ &= m \cdot \left[G_{\text{out}}^{(1)} \cdot G_{\text{in}} \mid G_{\text{out}}^{(2)} \cdot G_{\text{in}} \mid \cdots \mid G_{\text{out}}^{(t)} \cdot G_{\text{in}}\right] = m \cdot G. \end{aligned}$$

G is now a generator matrix of C by construction – which also makes C a linear code. □

Lemma 3.2 (Minimum Hamming distance – binary concatenation).
Given a linear binary concatenated code $C = C_{\text{out}} \tilde{\circ} C_{\text{in}}$ with the same notation as in Definition 3.2 and $\operatorname{lcm}(N, k) = N = t \cdot k$. Let $G_{\text{in}} \in \text{GF}(2)^{k \times n}$ be the generator matrix of the inner code and

$$\text{GF}(2)^{K \times N} = \text{GF}(2)^{K \times t \cdot k} \ni G_{\text{out}} = (G_{\text{out}}^{(1)} \mid G_{\text{out}}^{(2)} \mid \cdots \mid G_{\text{out}}^{(t)}), \quad G_{\text{out}}^{(j)} \in \text{GF}(2)^{K \times k},\ j \in \{1, 2, \ldots, t\}$$

be the generator matrix of the outer code.

Let s denote the number of matrices $G_{\text{out}}^{(j)}$ with full row rank ($\operatorname{rank}(G_{\text{out}}^{(j)}) = K$). Then the minimum Hamming distance of C is at least equal to $s \cdot d$, where d denotes the minimum Hamming distance of the inner code C_{in}.

Proof. Due to Lemma 3.1, the concatenation C is a linear code – and thus the minimum Hamming distance equals the minimum Hamming weight. From (3.1) we know how the j-th of the code blocks of length n looks like. Note that because of

$$c_j = m \cdot G_{\text{out}}^{(j)} \cdot G_{\text{in}} = m^{(j)} \cdot G_{\text{in}},$$

c_j is an inner code word. Therefore, if $m^{(j)} \neq 0$ we have $h(c_j) \geq d$. Furthermore, due to

$$h(c) = h(c_1, c_2, \cdots, c_t) = \sum_{j=1}^{t} h(c_j)$$

we just need to check whether the case $c_j = 0$ is possible for $m \neq 0$. The kernel of the linear map where we multiply a message from the left side to $G_{\text{out}}^{(j)}$ only consists of the zero message if and only if the matrix has full row rank, $\operatorname{rank}(G_{\text{out}}^{(j)}) = K$. Therefore, if $m \neq 0$ we also have $m^{(j)} \neq 0$ for these matrices – and therefore also $c_j \neq 0$, as C_{in} is a linear code.

Overall, if c is the code word of $m \neq 0$ regarding the concatenation C, the inequality

$$h(c) = \sum_{j=1}^{t} h(c_j) \geq \sum_{j=1}^{s} d = s \cdot d$$

holds. Thus, the minimum Hamming distance of C is at least $s \cdot d$. □

Remark.
Note that the minimum Hamming distance of binary concatenated codes does not depend directly on the Hamming distance of the outer code. Using the notation of Lemma 3.2, the inequality $D \geq s$ holds. However, when using two-step decoding as described in Algorithm 8, the minimum distance of the outer code definitely has an influence on the decoding capabilities, as can be seen in Section 3.2.

Algorithm 8 Binary code concatenation – decoding

1: **procedure** decode($c \in \mathrm{GF}(2)^{t \cdot n}$, $\mathrm{dec}_{\mathrm{in}}$, $\mathrm{dec}_{\mathrm{out}}$)
2: Split c into t code words of length n: $(c_1, \ldots, c_t) \leftarrow c$
3: Decode these inner code words: $v_j \leftarrow \mathrm{dec}_{\mathrm{in}}(c_j)$ ▷ $v_j \in \mathrm{GF}(2)^k$
4: Concatenate: $v \leftarrow (v_1, \ldots, v_t)$ ▷ $v \in \mathrm{GF}(2)^{t \cdot k} = \mathrm{GF}(2)^N$
5: Decode this outer code word: $m \leftarrow \mathrm{dec}_{\mathrm{out}}(v)$ ▷ $m \in \mathrm{GF}(2)^K$
6: **return** m
7: **end procedure**

Example 3.1.
Consider the following binary concatenation: Let $C_{\mathrm{out}} = \mathrm{Ham}[8,4,4]$ be the extended binary Hamming code and let C_{in} be constructed by a generator matrix G_{in}, where

$$G_{\mathrm{in}} := \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}\right], \qquad G_{\mathrm{out}} = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}\right].$$

A simple calculation with Sage shows that the minimum distance of C_{in} is 2. Using Lemma 3.1 we can compute the generator matrix of the binary concatenated code C, which is

$$G = \left[\begin{array}{cccccccc|cccccccc} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{array}\right].$$

In the process of evaluating G we had to split G_{out} into two blocks, thus we have $t = 2$. Finally, with the help of Sage, we can calculate the minimum distance of C, which is 7. The inequality

$$s \cdot d \leq d_{min}, \text{ that is } 2 \cdot 2 \leq 7$$

holds.

This is an example for the fact that the minimum distance often can be greatly improved, compared to the lower bound proved in Lemma 3.2.

3.1.1 Equivalency of binary concatenation and Galois concatenation

We are interested in investigating the relation between binary concatenated codes and Galois concatenated codes. On the one hand, we want to know whether there are any codes for which binary concatenation and Galois concatenation yield the same code, meaning that the encoding algorithms coincide. On the other hand, we want to investigate whether we can construct a binary code to a code over an extension field (or vice versa), such that the encoding algorithms yield the same bit strings.

Full equivalency
The point with binary concatenated codes is that we do not need to leave $\mathrm{GF}(2)$, whereas Galois concatenated codes – or at least the respective outer code – work over $\mathrm{GF}(2^k)$, where k is the dimension of the inner code.

However, there is a case where Galois concatenated codes also stay in $\mathrm{GF}(2)$: if $k = 1$, i.e. the inner code only encodes single bits. There is only one type of linear codes taking only one bit as input and having maximum Hamming weight: *repetition codes*.

Example 3.2.
Consider the following two codes:

$$C_{\text{out}} = \mathrm{Gol}[24, 12, 8], \quad C_{\text{in}} = \mathrm{Rep}[7, 1, 7].$$

No matter which concatenation we perform on these two codes, the result stays the same: First, we accumulate 12 bits and apply the outer code, receiving 24 bits. Then we apply the inner code on each bit, gaining $24 \cdot 7 = 168$ bits. The resulting code is a code of dimension 12 and length 168. Note that this code is nothing else than the Golay code repeated seven times – and thus has a minimum distance of $7 \cdot 8 = 56$. Therefore, the code is able to correct up to 27 errors.

Also note that G_{out} is a 12×24-matrix and the $G_{\text{out}}^{(j)}$ are the columns of G_{out}, so $G_{\text{out}}^{(j)} \in \mathrm{GF}(2)^{12\times 1}$. As none of the $G_{\text{out}}^{(j)}$ has full row rank the inequality from Lemma 3.2 does not yield an interesting lower bound.

Structural equivalency
At first we need a concept which allows us to decide whether two given codes are equivalent. The following definition will take care of that.

Definition 3.3 (Structural equivalency).
Given two linear binary codes, C_{out} and C_{in} in the sense of the definition of binary concatenation. Let C'_{out} be a linear code over $\mathrm{GF}(2^k)$. If the binary concatenation of C_{out} and C_{in} and the Galois concatenation of C'_{out} and C_{in} yield the same linear code, then C_{out} and C'_{out} are called *structurally equivalent*.

That is, two binary codes C_{out} and C'_{out} are called *structurally equivalent* if they generate the same bit strings although being defined over different fields.

Now we want to investigate whether every binary concatenation is structurally equivalent to a Galois concatenation – and vice versa.

Example 3.3.
We consider the binary concatenation of $\mathrm{Ham}[8, 4, 4]$ with $\mathrm{Ham}[8, 4, 4]$. We have

$$G_{\text{out}} = G_{\text{in}} = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}\right].$$

The result of this binary concatenation is a linear binary code of length 16 and dimension 4 which is generated by the following matrix:

$$G = \left[\begin{array}{cccccccc|cccccccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right].$$

If there is an outer code which generates, when Galois-concatenated with the $[8,4,4]$ extended Hamming code as the inner code, the same linear code as the binary concatenation above, we would call these outer codes *structurally equivalent*.

Such a code would have to be over the extension field $\mathrm{GF}(2^4)$, as the inner code has dimension 4. Also, as we are looking for a concatenated code of length 16 and dimension 4, the outer code would have to be of length 2, as one symbol in $GF(2^4)$ translates into 4 bits. So, we know that we are looking for a code of length 2 and dimension 1 over $\mathrm{GF}(2^4)$ – therefore, and because the given outer code is in standard form, the generator matrix has to be

$$G'_{\text{out}} = \left[\begin{array}{c|c} 1 & \omega \end{array}\right],$$

where $\omega \in \mathrm{GF}(2^4)$. At first, let us try to solve this by constructing $\mathrm{GF}(2^4)$ through a factor ring. Let

$$\mathrm{GF}(2^4) := \mathrm{GF}(2)[x]/(x^4+x+1),$$

where x^4+x+1 is an irreducible polynomial. Assuming we use the standard basis, i.e.

$$1 \triangleq 0001, \quad x \triangleq 0010, \quad x^2 \triangleq 0100, \quad x^3 \triangleq 1000,$$

we can try to solve the equations above for ω and hope that the solution is consistent. As 0001 is mapped to 0001|1110, 1 has to be mapped to $1 \mid x^3+x^2+x$. Therefore we have $\omega = x^3+x^2+x$. Trying to compute the code word of $x \triangleq 0010$ we find

$$x \cdot G'_{\text{out}} = x \cdot \left[\begin{array}{c|c} 1 & x^3+x^2+x \end{array}\right] = (x,\ x^3+x^2+x+1),$$

due to $x^4 = x+1$ which would mean that 0010 would get mapped to 0010|1111 – contradiction.

In the second attempt, we still want to use the standard basis, but we want to investigate whether another irreducible polynomial generates the field in a way such that G_{out} can be modeled. Using the standard basis directly implies that ω has to be x^3+x^2+x, as 1 is mapped to $(1,\ \omega)$. Furthermore, we want x to be mapped to $(x,\ x^3+x^2+1)$, that is

$$x \cdot \left[\begin{array}{c|c} 1 & x^3+x^2+x \end{array}\right] \stackrel{!}{=} (x,\ x^3+x^2+1),$$

which leads to the equation

$$x^4+x^3+x^2 \stackrel{!}{=} x^3+x^2+1 \iff x^4 = 1.$$

This equation would hold if we were to use the polynomial x^4+1 for generation of $\mathrm{GF}(2^4)$ – but this polynomial is not irreducible, $x^4+1 = (x+1)^4$. Therefore, the polynomial does not generate

$\mathrm{GF}(2^4)$, thus there is no configuration using the standard base.

The last attempt in trying to repair the ability of modeling G_{out} with a matrix G'_{out} over $\mathrm{GF}(2^4)$ is to consider arbitrary bases. That is, we have the following translations from $\mathrm{GF}(2)^4$ to $\mathrm{GF}(2^4)$:

$$1000 \triangleq \alpha, \quad 0100 \triangleq \beta, \quad 0010 \triangleq \gamma, \quad 0001 \triangleq \delta, \qquad \alpha, \beta, \gamma, \delta \in \mathrm{GF}(2^4).$$

The problem in modeling G_{out} is not the identity matrix, but rather the matrix on the right side of G_{out}. And if we could model this matrix with the basis above, the following equation has to hold (cf. Lemma 3.3):

$$\omega \cdot \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix},$$

which means that $(\alpha, \beta, \gamma, \delta)^t$ is an eigenvector to the eigenvalue ω of the matrix P above. Note that $P \in \mathrm{GF}(2^4)^{4\times 4}$. The characteristic polynomial is $\chi_P(x) = x^4 + 1 = (x+1)^4$. Therefore, $\omega = 1$ is an eigenvalue with algebraic multiplicity 4.
But now the first equation in the system of equations above is $\alpha = \beta + \gamma + \delta$, thus α, β, γ and δ are linearly dependent over $\mathrm{GF}(2)$ and therefore cannot form a basis.

This example shows that in general an arbitrary binary concatenation does not have to be structurally equivalent to a Galois concatenation. However, there are cases where binary concatenation can be expressed through an adequate Galois concatenation, as the following example will show.

Example 3.4.
We consider the binary concatenation of the extended Hamming $[8,4,4]$-code as the inner code with a $[8,4,3]$-code given by

$$G_{\text{out}} = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}\right].$$

Again we search for a linear code over $\mathrm{GF}(2^4)$ which models the behavior of C_{out} – which would have to be a code of length 2 and dimension 1. Therefore,

$$G'_{\text{out}} = \left[\begin{array}{c|c} 1 & \omega \end{array}\right]$$

if such a code exists. We will follow the same approach as in the example above: let $\mathrm{GF}(2^4) := \mathrm{GF}(2)[x]/(x^4 + x + 1)$ and we use the standard basis. The element ω can also be calculated just by interpretation of the last row of the matrix above: As 0001 is mapped to 0001|0110,

$$1 \mapsto \left(\begin{array}{c|c} 1 & x^2 + x \end{array}\right),$$

which means that $\omega = x^2 + x$. Fortunately, the other equations do not contradict the choice of ω as

$$\omega \cdot x = x^3 + x^2, \quad \omega \cdot x^2 = x^3 + x + 1, \quad \omega \cdot x^3 = x^2 + 1.$$

Therefore the outer code can be modeled through C'_{out}, which is a linear code over $\text{GF}(2^4)$ of length 2 and dimension 1 with generator matrix

$$G'_{\text{out}} = \left[\begin{array}{c|c} 1 & x^2 + x \end{array}\right].$$

Another issue we have to keep in mind in order to be able to construct a structurally equivalent code over $\text{GF}(2^k)$ is that the parameters of the binary outer code are not to be chosen arbitrarily. The code dimension k has to be a divisor of the code length n, otherwise the generator matrix cannot be split into $k \times k$ blocks which then may or may not be associated to an element in $\text{GF}(2^k)$. Before going on to answer the second question asked above (whether each Galois concatenation is structurally equivalent to a binary concatenation), we want to examine this correspondence between matrices over $\text{GF}(2)^{k\times k}$ and elements in $\text{GF}(2^k)$.

Lemma 3.3.
Let $\alpha \in \text{GF}(2^k)$, where $\text{GF}(2^k)$ is generated by ζ. Let $\varphi\colon \text{GF}(2^k) \to \text{GF}(2)^k$ denote the map which maps elements from $\text{GF}(2^k)$ to their binary representation in $\text{GF}(2)^k$, i.e. φ is the natural isomorphism mapping *vectors* from $\text{GF}(2^k)$ to their representation regarding the basis $\{\zeta^{k-1}, \ldots, \zeta, 1\}$ in $\text{GF}(2)^k$, i.e.

$$\varphi(a_{k-1}\zeta^{k-1} + \cdots + a_1\zeta + a_0) = (a_{k-1}, \ldots, a_1, a_0).$$

Then we can find a matrix $P_\alpha \in \text{GF}(2)^{k\times k}$ such that

$$\varphi(\omega \cdot \alpha) = \varphi(\omega) \cdot P_\alpha$$

holds for all $\omega \in \text{GF}(2^k)$. In that case we say that P_α is associated to α with respect to the basis $\{\zeta^{k-1}, \ldots, \zeta, 1\}$.

Proof. We examine the linear map $g : \text{GF}(2^k) \to \text{GF}(2^k),\ \omega \mapsto \omega \cdot \alpha$. As described in Section 1.2, $\text{GF}(2^k)$ can be considered as a vector space over $\text{GF}(2)$. Because of linear algebra we know that each linear map – in this case, even a linear operator, as g stays in the same vector space – has a matrix representation with respect to a chosen basis of the vector space. Furthermore, because we multiply the vectors from the left side, the rows of the matrix representation are the images of the basis vectors.

Choosing the basis $\{\zeta^{k-1}, \ldots, \zeta, 1\}$ and constructing the corresponding matrix representation yields a matrix P_α with the properties stated above. □

It is quite easy to construct an associated matrix to an element of an extension field. Algorithm 9 describes the construction of the corresponding matrix. The other problem – checking whether a given binary square matrix is the matrix representation of such a linear map with respect to some fixed basis – is harder and will be investigated in Section 3.3.

Now we want to answer the previous question regarding the existence of structurally equivalent linear binary codes.

Lemma 3.4.
Given a linear code C of length N and dimension K over the field $\text{GF}(2^k)$. Then there always is a structurally equivalent linear binary code C' of length $N \cdot k$ and dimension $K \cdot k$.

Algorithm 9 Construction of an associated matrix

procedure construct($\alpha \in \mathrm{GF}(2^k)$, generating element $\zeta \in \mathrm{GF}(2^k)$)
  $j \leftarrow 0$
  for $j < k$ **do**
    $v_j \leftarrow \varphi(\alpha \cdot \zeta^j)$ ▷ Find binary representations
    $j \leftarrow j + 1$
  end for
  $P_\alpha \leftarrow \begin{bmatrix} v_{k-1} \\ \vdots \\ v_0 \end{bmatrix}$ ▷ P_α is the matrix with v_j as rows
  return P_α
end procedure

Proof. Let G be the generator matrix of C, so $G \in \mathrm{GF}(2^k)^{K \times N}$. Due to Lemma 3.3, each element in $\mathrm{GF}(2^k)$ is associated to a matrix in $\mathrm{GF}(2)^{k \times k}$. By replacing each component of G by the respective associated matrix, we obtain a matrix $G' \in \mathrm{GF}(2)^{K \cdot k \times N \cdot k}$. Due to the properties of these associated matrices, the linear binary code C' generated by G' is structurally equivalent to C – which means that for each linear code over an extension field there is a structurally equivalent linear binary code. □

Remark.
Note that although structurally equivalent codes are essentially the same, the error correction efficiency is better for Galois-concatenated codes due to specialized decoding algorithms over extension fields which cause an improved decoding performance. We have very efficient decoding algorithms for these special codes over extension fields which we simply do not have for binary codes.

Before we start analyzing some concrete binary concatenation examples with respect to error decoding capabilities etc., we summarize the theoretical results in this section in the following theorem.

Theorem 3.5 (Relation between Galois and binary concatenation).
The binary concatenation as described in Definition 3.2 is a generalization of the Galois concatenation as described in Definition 3.1 in terms of structural equivalency.

Remark.
Note that by Example 3.3 we know that there are binary codes which do not have a structural equivalent code over the according extension field. Therefore, the two concatenation types are **not** equivalent.

3.2 Examples

Before we jump right into some examples for the binary concatenation there is a question regarding the error correction capabilities of these codes we should answer first. If too many errors in the "inner code words" happen, they lead to an incorrect code word – and thus also the decoded

message is wrong as different code words must have different messages. But how wrong can these messages be?

This means we want to calculate $d(m, m')$ if we know $d(mG, m'G)$. Instead of brute force computing all these distances, we can make some improvements:

$$d(mG, m'G) = d(mG - m'G, 0) = d(\tilde{m}G, 0) = h(\tilde{m}G), \quad d(m, m') = h(\tilde{m}).$$

Considering these equations we can compute the number of code words with specific Hamming weight (i.e. the number of errors in the case of considering 0 as the "true" message) and how many how erroneous messages we can extract from them. We will include this analysis for the inner code in each of the following examples.

3.2.1 Ham$[8,4,4]$ ◦̃ Ham$[8,4,4]$

In this example we use the same code for the inner and the outer code: the extended Hamming $[8,4,4]$-code. Note that the generator matrix $G_{\text{in}} = G_{\text{out}}$ is given by

$$G_{\text{in}} = G_{\text{out}} = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}\right].$$

Practically, the binary concatenation of these two codes can be implemented as in Algorithm 10.

Algorithm 10 Hamming-Hamming binary concatenation – encoding

1: **procedure** encode($m \in \text{GF}(2)^4,\ G$)
2: $(\tilde{c}_1, \tilde{c}_2, \ldots \tilde{c}_8) = \tilde{c} \leftarrow mG_{\text{out}}$ ▷ Standard encoding
3: $m_1 \leftarrow (\tilde{c}_1, \tilde{c}_2, \tilde{c}_3, \tilde{c}_4)$ ▷ First new message from first code block
4: $m_2 \leftarrow (\tilde{c}_5, \tilde{c}_6, \tilde{c}_7, \tilde{c}_8)$ ▷ Second new message from second code block
5: $c_1 \leftarrow m_1 G_{\text{in}},\ c_2 \leftarrow m_2 G_{\text{in}}$ ▷ Encode these new messages again
6: **return** $c \leftarrow (c_1, c_2)$ ▷ Concatenate the new code words and return the result
7: **end procedure**

As described in Lemma 3.1, we can find the generator matrix for the resulting concatenated code, which is

$$G = \left[\begin{array}{cccc|cccccccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right].$$

We now analyze the resulting code in two ways: first, we decode with the decoding algorithm generally described in Algorithm 8 – which degenerates to Algorithm 11 in this case. And second, we want to apply syndrome decoding on the linear binary code generated by the large matrix G above.

Algorithm 11 Hamming-Hamming binary concatenation – decoding

1: **procedure** decode($c \in \mathrm{GF}(2)^{16}$)
2: Split the code word c in half and retrieve two smaller code words $c_1,\ c_2$
3: Decode: $m_1 \leftarrow \mathrm{dec}(c_1),\ m_2 \leftarrow \mathrm{dec}(c_2)$
4: Combine these messages to another code word, $\tilde{c} = (m_1, m_2)$
5: Decode: $m \leftarrow \mathrm{dec}(\tilde{c})$
6: **return** m
7: **end procedure**

The minimum distance for the code generated by G is 8, which can be computed easily with the help of Sage. Therefore, 3 errors can be corrected using syndrome decoding.

Running some calculations on the extended Hamming code we find that

- there is 1 code word with Hamming weight 0 where we can extract one message of weight 0.
- there are 14 code words of weight 4 where we can extract 4, 6 and 4 messages of weights 1, 2 and 3, respectively.
- there is 1 code word of weight 8 which comes from a message of weight 4.

Taking into account that we are only able to correct one error per word with the extended Hamming code, we find that the messages gained from the two simultaneous decoding procedures must not contain more than one error overall in order to decode correctly.

Now, let us assume we encode the zero message with the method above and want to decode the (erroneous) transmitted word. Brute-force computation shows that from $2^{16} = 65536$ possible transmitted words, 61440 are decoded incorrectly (using two step syndrome decoding, see Algorithm 11). The weight distribution for the words decoded incorrectly (i.e. in our case the number of errors in each word) is as follows:

Errors	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
incorr. dec. words	0	0	27	284	1250	3516	7114	10768	12489	11244	7899	4332	1820	560	120	16	1
Number of words	1	16	120	560	1820	4368	8008	11440	12870	11440	8008	4368	1820	560	120	16	1
dec. failure ratio	0	0	0.225	0.507	0.687	0.805	0.888	0.941	0.970	0.983	0.986	0.992	1	1	1	1	1

If p_b denotes the single bit error probability and w_i denotes the number of words with weight i being decoded incorrectly (taken from the table above), then the probability that decoding does not lead back to the zero message is given by

$$\mathbb{P}(\text{transmitted word not decoded back to 0}) = \sum_{j=0}^{16} w_j \cdot p_b^j \cdot (1-p_b)^{16-j}.$$

Due to symmetry this is already the probability that an arbitrary word is not decoded correctly. Overall:

$$\mathbb{P}(\text{decoding error}) = \sum_{j=0}^{16} w_j \cdot p_b^j \cdot (1-p_b)^{16-j} = 27p^2 - 94p^3 + 15p^4 + 840p^5 - 3259p^6 + O(p^7)$$

Assuming $p_b = 0.1$, we find $\mathbb{P}(\text{decoding error}) = 0.183250860198400$ – the error probability is substantially worse than the respective bit error probability.

If we would decode this code by ordinary syndrome decoding we have the following weight distribution for incorrectly decoded words (55467 from 65536 were decoded incorrectly):

Errors	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
incorr. dec. words	0	0	0	0	0	784	5320	10416	12614	11440	8008	4368	1820	560	120	16	1
Number of words	1	16	120	560	1820	4368	8008	11440	12870	11440	8008	4368	1820	560	120	16	1
dec. failure ratio	0	0	0	0	0	0.179	0.664	0.910	0.980	1	1	1	1	1	1	1	1

This yields the following expansion:

$$\mathbb{P}(\text{decoding error}) = 784p^5 - 3304p^6 + 336p^7 + 28910p^8 - 94176p^9 + O(p^{10}),$$

and therefore, assuming $p_b = 0.1$ we find $\mathbb{P}(\text{decoding error}) = 0.00477900501289390$, which is a way better result.

In Figure 3.1 we plotted the error probability curves for varying p_b. The normal blue line is for the decoding algorithm as described in Algorithm 11, the stroked line is for syndrome decoding.

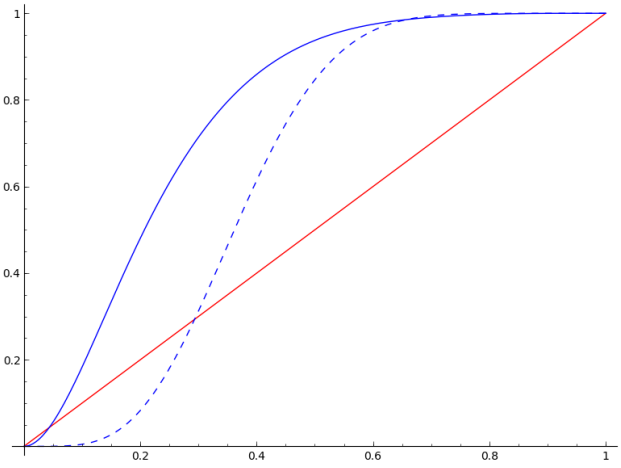

Figure 3.1: Error probability curves

3.2.2 MatrixCode$[12,4,5]$ ∘ Ham$[8,4,4]$

In this example we use a linear binary $[12,4,5]$-code defined by some generator matrix G_{out} as the outer code and the extended Hamming $[8,4,4]$-code as the inner code. The respective generator matrices are given by

$$G_{\text{out}} = \left[\begin{array}{cccc|cccccccc} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{array}\right], \quad G_{\text{in}} = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}\right].$$

Algorithm 12 Matrix Code-Hamming binary concatenation – encoding

procedure encode($m \in \mathrm{GF}(2)^4,\ G_{\text{in}},\ G_{\text{out}}$)
  $\tilde{c} \leftarrow mG_{\text{out}}$ ▷ Standard encoding
  $m_1 \leftarrow (\tilde{c}_1, \tilde{c}_2, \tilde{c}_3, \tilde{c}_4)$ ▷ First new message from first code block
  $m_2 \leftarrow (\tilde{c}_5, \tilde{c}_6, \tilde{c}_7, \tilde{c}_8)$ ▷ Second new message from second code block
  $m_3 \leftarrow (\tilde{c}_9, \tilde{c}_{10}, \tilde{c}_{11}, \tilde{c}_{12})$ ▷ Third new message from third code block
  $c_1 \leftarrow m_1 G_{\text{in}},\ c_2 \leftarrow m_2 G_{\text{in}},\ c_3 \leftarrow m_3 G_{\text{in}}$ ▷ Encode these messages again
  return $c \leftarrow (c_1, c_2, c_3)$ ▷ Concatenate the new code words and return the result
end procedure

Practically, the concatenation of these two codes can be implemented as in Algorithm 12.

As described in Lemma 3.1, we can find the generator matrix for the resulting concatenated code, which is

$$G = \left[\begin{array}{cccc|cccccccccccccccccccc} 1&0&0&0&0&1&1&1&1&0&1&1&0&1&0&0&0&0&1&0&1&1&0&1\\ 0&1&0&0&1&0&1&1&0&1&1&1&1&0&0&0&1&0&0&0&0&1&1&1\\ 0&0&1&0&1&1&0&1&1&1&1&1&1&1&1&1&0&1&1&1&1&0&0&0\\ 0&0&0&1&1&1&1&0&1&1&1&0&0&0&0&1&1&1&1&0&0&0&0&1 \end{array}\right]$$

The minimum distance for the code generated by G is 12, which can be computed easily with the help of Sage. Therefore, 5 errors can be corrected using syndrome decoding.

Algorithm 13 Matrix Code-Hamming binary concatenation – decoding

procedure decode($c \in \mathrm{GF}(2)^{24}, \mathrm{dec}_{\text{in}},\ \mathrm{dec}_{\text{out}}$)
  Split the code word c in three equal pieces and obtain the smaller code words c_1, c_2, c_3
  Decode these code words: $m_1 \leftarrow \mathrm{dec}_{\text{in}}(c_1),\ m_2 \leftarrow \mathrm{dec}_{\text{in}}(c_2),\ m_3 \leftarrow \mathrm{dec}_{\text{in}}(c_3)$
  Combine the messages to another code word, $\tilde{c} = (m_1, m_2, m_3)$
  Try to decode $\tilde{c}$ and receive a message m
  return $m \leftarrow \mathrm{dec}_{\text{out}}(c)$
end procedure

Running some calculations for the extended Hamming code we find that

- there is 1 code word with Hamming weight 0 where we can extract one message of weight 0.
- there are 14 code words of weight 4 where we can extract 4, 6 and 4 messages of weight 1, 2 and 3, respectively.
- there is 1 code word of weight 8 which comes from a message of weight 4.

In order to guarantee that decoding works correctly, the three messages obtained in the first step of decoding must not contain more than two errors, as the outer code can only correct two errors.

Now, let us assume we encode the zero message with the method above and want to decode the (erroneous) transmitted word. Brute-force computation shows that from $2^{24} = 16777216$ possible transmitted words, 15728640 are decoded incorrectly (using two step syndrome decoding). The

weight distribution for the words decoded incorrectly (i.e. in our case the number of errors in each word) is as follows:

Weight	1	2	3	4	5	6	7	8	9	10	11	12
incorr. dec. words	0	6	108	1764	16152	78220	248940	595815	1139196	1790346	2351532	2601654
Number of words	24	276	2024	10626	42504	134596	346104	735471	1307504	1961256	2496144	2704156
dec. failure ratio	0	0.022	0.053	0.166	0.380	0.581	0.719	0.810	0.871	0.913	0.942	0.962
Weight	13	14	15	16	17	18	19	20	21	22	23	24
incorr. dec. words	2432520	1924872	1288060	726684	343260	134056	42504	10626	2024	276	24	1
Number of words	2496144	1961256	1307504	735471	346104	134596	42504	10626	2024	276	24	1
dec. failure ratio	0.974	0.981	0.985	0.988	0.992	0.996	1	1	1	1	1	1

If p_b denotes the single bit error probability and w_i denotes the number of words with weight i being decoded incorrectly (taken from the table above), then the probability that decoding does not lead back to the zero message is given by

$$\mathbb{P}(\text{transmitted word not decoded back to 0}) = \sum_{j=0}^{24} w_j \cdot p_b^j \cdot (1-p_b)^{24-j}.$$

Note that $w_0 = 0$, obviously. Due to symmetry this is already the probability that an arbitrary word is not decoded correctly. Overall:

$$\mathbb{P}(\text{decoding error}) = \sum_{j=0}^{24} w_j \cdot p_b^j \cdot (1-p_b)^{24-j} = 6p^2 - 24p^3 + 882p^4 - 5688p^5 + 6742p^6 + O(p^7)$$

Assuming $p_b = 0.1$, we find $\mathbb{P}(\text{decoding error}) = 0.0782693356314399$. If we would decode this concatenated code by ordinary syndrome decoding we have the following weight distribution for incorrectly decoded words (14693704 out of 16777216 words were decoded incorrectly):

Errors	1	2	3	4	5	6	7	8	9	10	11	12
incorr. dec. words	0	0	0	0	0	0	9504	118152	642564	1711134	2471664	2704156
Number of words	24	276	2024	10626	42504	134596	346104	735471	1307504	1961256	2496144	2704156
dec. failure ratio	0	0	0	0	0	0	0.028	0.161	0.491	0.872	0.990	1
Errors	13	14	15	16	17	18	19	20	21	22	23	24
incorr. dec. words	2496144	1961256	1307504	735471	346104	134506	42504	10626	2024	276	24	1
Number of words	2496144	1961256	1307504	735471	346104	134596	42504	10626	2024	276	24	1
dec. failure ratio	1	1	1	1	1	1	1	1	1	1	1	1

Following this approach, we obtain the following expansion:

$$\mathbb{P}(\text{decoding error}) = 9504p^7 - 43416p^8 + 44676p^9 - 211806p^{10} + 2439408p^{11} + O(p^{12})$$

Again, assuming $p_b = 0.1$ we find $\mathbb{P}(\text{decoding error}) = 0.000556013544860564$, which is a better result.

In Figure 3.2 we plotted the error probability curves for varying p_b. The normal blue curve is for the decoding algorithm as described in Algorithm 13, the stroked line is the curve for "standard" syndrome decoding.

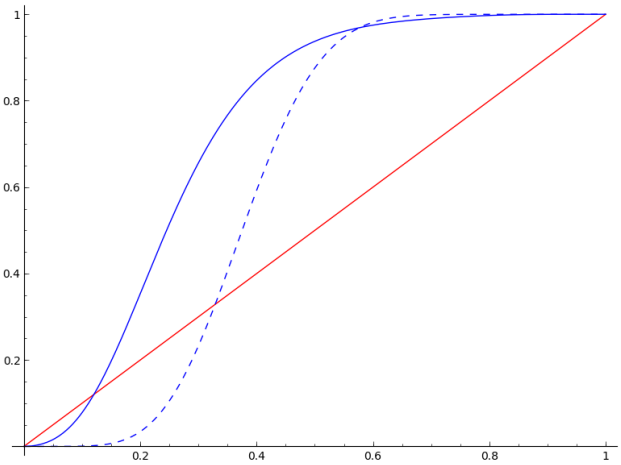

Figure 3.2: Error probability curves

3.2.3 MatrixCode$[12,4,4]$ ∘̃ Ham$[8,4,4]$

In this example we want to use a linear binary $[12,4,4]$-code defined by the following generator matrix G_{out} as the outer code and the extended Hamming $[8,4,4]$-code as the inner code. The respective generator matrices are given by

$$G_{\text{out}} = \left[\begin{array}{cccc|cccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{array}\right], \quad G_{\text{in}} = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}\right].$$

This example differs from the one before only through the fact, that the outer code now has minimum distance 4 instead of 5 and thus only is able to correct 1 error. We will see, in how far this change effects the decoding failure probability. Practically, the concatenation of these two codes can be implemented as in Algorithm 14.

Algorithm 14 Matrix Code 2-Hamming binary concatenation – encoding

1: **procedure** encode($m \in \mathrm{GF}(2)^4,\ G_{\text{in}},\ G_{\text{out}}$)
2: $\tilde{c} \leftarrow mG_{\text{out}}$ ▷ Standard encoding
3: $m_1 \leftarrow (\tilde{c}_1, \tilde{c}_2, \tilde{c}_3, \tilde{c}_4)$ ▷ First new message from first code block
4: $m_2 \leftarrow (\tilde{c}_5, \tilde{c}_6, \tilde{c}_7, \tilde{c}_8)$ ▷ Second new message from second code block
5: $m_3 \leftarrow (\tilde{c}_9, \tilde{c}_{10}, \tilde{c}_{11}, \tilde{c}_{12})$ ▷ Third new message from third code block
6: $c_1 \leftarrow m_1 G_{\text{in}},\ c_2 \leftarrow m_2 G_{\text{in}},\ c_3 \leftarrow m_3 G_{\text{in}}$ ▷ Encode these messages again
7: **return** $c \leftarrow (c_1, c_2, c_3)$ ▷ Concatenate the new code words and return the result
8: **end procedure**

As described in Lemma 3.1, we can find the generator matrix for the resulting concatenated code,

which is

$$G = \left[\begin{array}{cccc|cccccccccccccccccccc} 1&0&0&0&0&1&1&1&0&0&1&1&0&0&1&1&0&1&1&0&0&1&1&0 \\ 0&1&0&0&1&0&1&1&1&0&0&0&0&1&1&1&1&0&0&1&1&0&0&1 \\ 0&0&1&0&1&1&0&1&1&1&0&0&1&1&0&0&1&0&0&0&0&1&1&1 \\ 0&0&0&1&1&1&1&0&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1 \end{array}\right]$$

The minimum distance for the code generated by G is 8, which can be computed easily with the help of Sage. Therefore, 3 errors can be corrected using syndrome decoding.

Algorithm 15 Matrix Code 2-Hamming binary concatenation – decoding

1: **procedure** decode($c \in \mathrm{GF}(2)^{24}, \mathrm{dec}_{\mathrm{in}}, \mathrm{dec}_{\mathrm{out}}$)
2: Split the code word c in three equal pieces and obtain the smaller code words c_1, c_2, c_3
3: Decode these code words: $m_1 \leftarrow \mathrm{dec}_{\mathrm{in}}(c_1),\ m_2 \leftarrow \mathrm{dec}_{\mathrm{in}}(c_2),\ m_3 \leftarrow \mathrm{dec}_{\mathrm{in}}(c_3)$
4: Combine the messages to another code word, $\tilde{c} = (m_1, m_2, m_3)$
5: Try to decode $\tilde{c}$ and receive a message m
6: **return** $m \leftarrow \mathrm{dec}_{\mathrm{out}}(c)$
7: **end procedure**

Running some calculations for the extended Hamming code we find that

- there is 1 code word with Hamming weight 0 where we can extract one message of weight 0.
- there are 14 code words of weight 4 where we can extract 4, 6 and 4 messages of weight 1, 2 and 3, respectively.
- there is 1 code word of weight 8 which comes from a message of weight 4.

In order to guarantee that decoding works correctly, the three messages obtained in the first step of decoding must not contain more than one error overall, as the outer code can only correct one error.

Now, let us assume we encode the zero message with the method above and want to decode the (erroneous) transmitted word. Brute-force computation shows that from $2^{24} = 16777216$ possible transmitted words, 15728640 are decoded incorrectly (using two step syndrome decoding). It is quite interesting that the total number of incorrectly decoded words actually is the same as in the last example! The weight distribution for the words decoded incorrectly (i.e. in our case the number of errors in each word) is as follows:

Errors	1	2	3	4	5	6	7	8	9	10	11	12
incorr. dec. words	0	18	324	3060	19200	81843	252180	598200	1140380	1791180	2350752	2598016
Number of words	24	276	2024	10626	42504	134596	346104	735471	1307504	1961256	2496144	2704156
dec. failure ratio	0	0.0652	0.160	0.288	0.452	0.608	0.729	0.813	0.872	0.913	0.942	0.961

Errors	13	14	15	16	17	18	19	20	21	22	23	24
incorr. dec. words	2427468	1920459	1285880	726369	343584	134272	42504	10626	2024	276	24	1
Number of words	2496144	1961256	1307504	735471	346104	134596	42504	10626	2024	276	24	1
dec. failure ratio	0.972	0.979	0.983	0.988	0.993	0.998	1	1	1	1	1	1

If p_b denotes the single bit error probability and w_i denotes the number of words with weight i

being decoded incorrectly (taken from the table above), then the probability that decoding does not lead back to the zero message is given by

$$\mathbb{P}(\text{transmitted word not decoded back to 0}) = \sum_{j=0}^{24} w_j \cdot p_b^j \cdot (1-p_b)^{24-j}.$$

Note that $w_0 = 0$, obviously. Due to symmetry this is already the probability that an arbitrary word is not decoded correctly. Overall:

$$\mathbb{P}(\text{decoding error}) = \sum_{j=0}^{24} w_j \cdot p_b^j \cdot (1-p_b)^{24-j} = 18p^2 - 72p^3 + 414p^4 - 1680p^5 - 807p^6 + O(p^7)$$

Assuming $p_b = 0.1$, we find $\mathbb{P}(\text{decoding error}) = 0.134197324635424$.

If we would decode this concatenated code by ordinary syndrome decoding we have the following weight distribution for incorrectly decoded words (14693704 from 16777216 words were decoded incorrectly):

Errors	1	2	3	4	5	6	7	8	9	10	11	12
incorr. dec. words	0	0	0	0	56	924	15064	129907	632184	1664832	2465040	2704156
Number of words	24	276	2024	10626	42504	134596	346104	735471	1307504	1961256	2496144	2704156
dec. failure ratio	0	0	0	0	0.001	0.007	0.044	0.177	0.483	0.849	0.988	1

Errors	13	14	15	16	17	18	19	20	21	22	23	24
incorr. dec. words	2496144	1961256	1307504	735471	346104	134596	42504	10626	2024	276	24	1
Number of words	2496144	1961256	1307504	735471	346104	134596	42504	10626	2024	276	24	1
dec. failure ratio	1	1	1	1	1	1	1	1	1	1	1	1

Thus we obtain the following expansion:

$$\mathbb{P}(\text{decoding error}) = 56p^5 - 140p^6 + 8008p^7 - 39073p^8 + 65448p^9 + O(p^{10})$$

Again, assuming $p_b = 0.1$ we find $\mathbb{P}(\text{decoding error}) = 0.000881642838120140$, which is a better result.

In Figure 3.3 we plotted the error probability curves for varying p_b. The normal blue curve is for the decoding algorithm as described in Algorithm 15, the stroked line is the curve for "standard" syndrome decoding.

Remark.
Comparing the examples in Sections 3.2.2 and 3.2.3 we see that the hamming distance on the outer code clearly has an impact of the decoding capability of both decoding algorithms investigated!

3.2.4 MatrixCode$[12,4,5]$ ∘ Golay$[24,12,8]$

In this example we want to analyze a very simple concatenation by using the same linear $[12,4,5]$-code as above as the outer code and the binary extended $[24,12,8]$-Golay code as the inner code. The respective generator matrices are given by

$$G_{\text{out}} = \left[\begin{array}{cccc|cccccccc} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{array}\right],$$

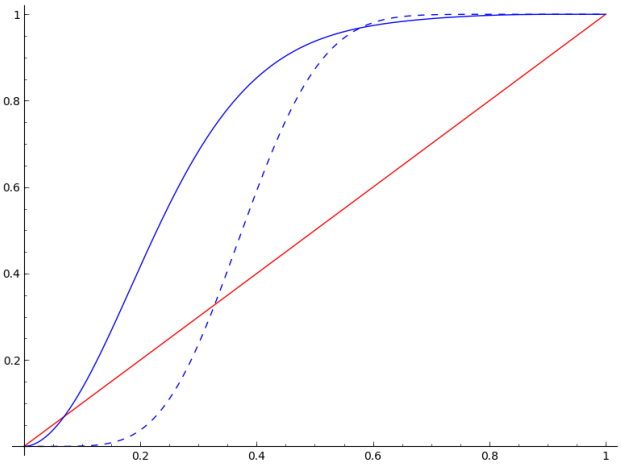

Figure 3.3: Error probability curves

and

$$G_{\text{in}} = \left[\begin{array}{cccccccccccc|cccccccccccc} 1&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&1&1&1&0&0&0&1&1\\ 0&1&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&0&0&1&0&0&1&0\\ 0&0&1&0&0&0&0&0&0&0&0&0&1&1&0&1&0&0&1&0&1&0&1&1\\ 0&0&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&1&1&1&0&1&1&0\\ 0&0&0&0&1&0&0&0&0&0&0&0&1&1&0&0&1&1&0&1&1&0&0&1\\ 0&0&0&0&0&1&0&0&0&0&0&0&0&1&1&0&0&1&1&0&1&1&0&1\\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&1&0&0&1&1&0&1&1&1\\ 0&0&0&0&0&0&0&1&0&0&0&0&1&0&1&1&0&1&1&1&1&0&0&0\\ 0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&1&1&0&1&1&1&1&0&0\\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&1&1&0&1&1&1&1&0\\ 0&0&0&0&0&0&0&0&0&0&1&0&1&0&1&1&1&0&0&0&1&1&0&1\\ 0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&1&1&1&0&0&0&1&1&1 \end{array}\right].$$

Practically, the implementation of the encoding algorithm is very easy this time and simply consists of the calculation of $c \leftarrow m \cdot G_{\text{out}} \cdot G_{\text{in}}$. As described in Lemma 3.1, we can find the generator matrix for the resulting concatenated code, which is

$$G_{\text{out}} \cdot G_{\text{in}} = G = \left[\begin{array}{cccc|cccccccccccccccccccc} 1&0&0&0&1&0&1&1&0&0&1&0&0&1&0&1&1&1&1&1&1&0&0&0\\ 0&1&0&0&0&1&1&1&1&0&0&0&0&1&0&0&0&0&0&0&1&1&0&0\\ 0&0&1&0&1&1&1&1&0&1&1&1&0&0&1&1&0&1&0&0&0&1&0&0\\ 0&0&0&1&1&1&1&0&1&1&1&0&1&0&0&1&0&0&0&1&1&0&1&0 \end{array}\right]$$

The minimum distance for the code generated by G is 8, which can be computed easily with the help of Sage. Therefore, 3 errors can be corrected using syndrome decoding.

Running some calculations for the extended binary Golay code we find that

- there is 1 code word with Hamming weight 0 where we can extract one message of weight 0.
- there are 759 code words of weight 8 where we can extract 12, 60, 180, 255, 180, 60 and 12 messages of weight 1, 2, 3, 4, 5, 6 and 7, respectively.
- there are 2576 code words of weight 12 where we can extract 6, 40, 240, 600, 804, 600, 240, 40, 6 and 6 words of weight 2, 3, 4, 5, 6, 7, 8, 9 and 10, respectively.
- there are 759 code words of weight 16 where we can extract 12, 60, 180, 255, 180, 60 and 12 words of weight 5, 6, 7, 8, 9, 10 and 11, respectively.
- there is 1 code word of weight 24 which comes from a message of weight 12.

In order to guarantee that decoding works correctly, the message obtained in the first step of decoding must not contain more than 2 errors, as the outer code can only correct two errors.

Now, let us assume we encode the zero message with the method above and want to decode the (erroneous) transmitted word. Brute-force computation shows that from $2^{24} = 16777216$ possible transmitted words, 15728640 are decoded incorrectly (using two step syndrome decoding). Once again the same exact number as before! The weight distribution for the words decoded incorrectly (i.e. in our case the number of errors in each word) is as follows:

Errors	1	2	3	4	5	6	7	8	9	10	11	12
incorr. dec. words	0	0	0	7035	31192	103068	253992	599268	1098692	1757498	2340412	2589743
Number of words	24	276	2024	10626	42504	134596	346104	735471	1307504	1961256	2496144	2704156
dec. failure ratio	0	0	0	0.662	0.734	0.766	0.734	0.815	0.840	0.896	0.938	0.958
Errors	13	14	15	16	17	18	19	20	21	22	23	24
incorr. dec. words	2453532	1932018	1295844	730191	346104	134596	42504	10626	2024	276	24	1
Number of words	2496144	1961256	1307504	735471	346104	134596	42504	10626	2024	276	24	1
dec. failure ratio	0.983	0.985	0.991	0.993	1	1	1	1	1	1	1	1

If p_b denotes the single bit error probability and w_i denotes the number of words with weight i being decoded incorrectly (taken from the table above), then the probability that decoding does not lead back to the zero message is given by

$$\mathbb{P}(\text{transmitted word not decoded back to } 0) = \sum_{j=0}^{24} w_j \cdot p_b^j \cdot (1-p_b)^{24-j}.$$

Note that $w_0 = 0$, obviously. Due to symmetry this is already the probability that an arbitrary word is not decoded correctly. Overall:

$$\mathbb{P}(\text{decoding error}) = \sum_{j=0}^{24} w_j \cdot p_b^j \cdot (1-p_b)^{24-j} = 7035p^4 - 109508p^5 + 847070p^6 + O(p^7).$$

Assuming $p_b = 0.1$, we find $\mathbb{P}(\text{decoding error}) = 0.148754405145761$.

If we would decode this "concatenated" code by ordinary syndrome decoding we have the following

weight distribution for incorrectly decoded words (15728640 out of 16777216 words got decoded incorrectly):

Errors	1	2	3	4	5	6	7	8	9	10	11	12
incorr. dec. words	0	0	0	35	616	10484	90952	413684	1086732	1893816	2491716	2704075
Number of words	24	276	2024	10626	42504	134596	346104	735471	1307504	1961256	2496144	2704156
dec. failure ratio	0	0	0	0.003	0.015	0.078	0.263	0.562	0.831	0.966	0.998	1

Errors	13	14	15	16	17	18	19	20	21	22	23	24
incorr. dec. words	2496144	1961256	1307504	735471	346104	134596	42504	10626	2024	276	24	1
Number of words	2496144	1961256	1307504	735471	346104	134596	42504	10626	2024	276	24	1
dec. failure ratio	1	1	1	1	1	1	1	1	1	1	1	1

This yields the following expansion:

$$\mathbb{P}(\text{decoding error}) = 35p^4 - 84p^5 + 543p^6 - 32324p^7 + 44223p^8 + O(p^9).$$

Therefore, assuming $p_b = 0.1$ we find $\mathbb{P}(\text{decoding error}) = 0.00538887888517034$, which is a better result.

In Figure 3.4 we plotted the error probability curves for varying p_b. On the left side is the curve for the two-step decoding algorithm, on the right side is the curve for "standard" syndrome decoding.

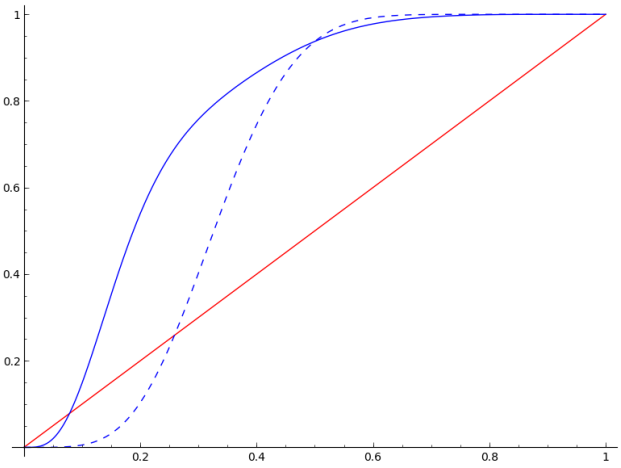

Figure 3.4: Error probability curves

3.3 Associated matrices

For the following considerations let p be a prime number. In this section we want to investigate how we can determine whether a given matrix $P \in \mathrm{GF}(p)^{k\times k}$ is an associated matrix to some $\omega \in \mathrm{GF}(p^k)$ with respect to some basis of $\mathrm{GF}(p^k)$. At first we want to prove that this problem is related to finding eigenvectors and eigenvalues.

Lemma 3.6.
Let $\zeta \in \mathrm{GF}(p^k)$ be a primitive element with minimal polynomial $f \in \mathrm{GF}(p)[x]$. If $P_\omega \in \mathrm{GF}(p)^{k\times k}$ is the associated matrix to some $\omega \in \mathrm{GF}(p^k)$ with respect to the basis $B = \{\alpha_1, \ldots, \alpha_k\}$ of $\mathrm{GF}(p^k)$, then the equation

$$\omega \cdot (\alpha_1, \ldots, \alpha_k)^t = P_\omega \cdot (\alpha_1, \ldots, \alpha_k)^t$$

holds, so $(\alpha_1, \ldots, \alpha_k)$ is a right eigenvector of P_ω to the eigenvalue ω.

Proof. As noted in the proof of Lemma 3.3, the rows of P_ω are the images of the basis vectors. Therefore, if

$$\omega \cdot \alpha_j = \sum_{i=1}^{k} a_{ji} \cdot \alpha_i \quad \Rightarrow \quad [\omega \cdot \alpha_j]_B = (a_{j1}, \ldots, a_{jk}),$$

and the matrix P_ω is given by

$$P_\omega = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{k1} & \cdots & a_{kk} \end{bmatrix}.$$

Finally, we have

$$\begin{aligned} P_\omega \cdot (\alpha_1, \ldots, \alpha_k)^t &= \sum_{i=1}^{k} (a_{1i}, \ldots, a_{ki})^t \cdot \alpha_i = \left(\sum_{i=1}^{k} a_{1i}\alpha_i, \ldots, \sum_{i=1}^{k} a_{ki}\alpha_i \right)^t \\ &= (\omega\alpha_1, \ldots, \omega\alpha_k)^t = \omega \cdot (\alpha_1, \ldots, \alpha_k)^t. \end{aligned}$$

□

Remark.
Note that the converse of Lemma 3.6 is not true in general. This is because the components of an arbitrary eigenvector to some eigenvalue ω of a matrix P are not necessarily linearly independent and thus may not form a basis. However, if the linear independency holds, the statement can be reversed.

Next we want investigate to what extend the observed structure can be exploited in order to yield a criterion for checking whether such a matrix has an element associated to it in the extension field with respect to some basis or not.

The first criterion we want to give is rather weak (primarily because it is a necessary statement), but in some situations it might be useful to check which eigenvalues of a matrix are uninteresting for the purpose above.

Lemma 3.7.
Let $P \in \mathrm{GF}(p)^{k\times k}$ and ζ be a generating element of $\mathrm{GF}(p^k)$ with minimal polynomial $f \in \mathrm{GF}(p)[x]$.

Let $\varphi : \mathrm{GF}(p^k) \to GF(p)^k$ denote the function which maps vectors $\alpha \in \mathrm{GF}(p^k)$ to their representation regarding the basis $B = \{\zeta^{k-1}, \zeta^{k-2}, \ldots, \zeta, 1\}$, i.e.

$$\varphi\left(\sum_{j=0}^{k-1} a_j \cdot \zeta^j\right) = (a_{k-1}, a_{k-2}, \ldots, a_1, a_0).$$

Also let $\Phi : \mathrm{GF}(p^k)^k \to \mathrm{GF}(p)^{k\times k}$ be defined through

$$\Phi(\alpha_1, \ldots, \alpha_k) = \begin{bmatrix} \varphi(\alpha_1) \\ \vdots \\ \varphi(\alpha_k) \end{bmatrix},$$

so Φ maps k elements from $\mathrm{GF}(p^k)$ to a matrix where each row is given by the representation of α_j regarding B. If P is associated to an eigenvalue $\omega \in \mathrm{GF}(p^k)$ with respect to some basis of $\mathrm{GF}(p^k)$, and if $\{v_1, \ldots, v_r\}$ is a basis of the eigenspace $\mathcal{E}_\omega^P$, then the inequality

$$\sum_{j=1}^{r} \mathrm{rank}(\Phi(v_j)) \geq k$$

holds.

Proof. From Lemma 3.6 we know that P is the associated matrix to ω if and only if there is an eigenvector $w \in \mathcal{E}_\omega^P$ where the components of w, i.e. $w_1, w_2, \ldots, w_k$ are linearly independent over $\mathrm{GF}(p^k)$. Obviously, this is equivalent to $\det(\Phi(w)) \neq 0$, which is the case if and only if $\mathrm{rank}(\Phi(w)) = k$.

Furthermore, the following chain of equations holds because of $v_1, \ldots, v_r$ being a basis of $\mathcal{E}_\omega^P$, Φ being a homomorphism with respect to the addition, as well because of Lemma 3.3:

$$\Phi(w) = \Phi\left(\sum_{j=1}^{r} c_j \cdot v_j\right) = \sum_{j=1}^{r} \Phi(c_j \cdot v_j) = \sum_{j=1}^{r} \begin{bmatrix} \varphi(c_j \cdot v_j^{(1)}) \\ \vdots \\ \varphi(c_j \cdot v_j^{(k)}) \end{bmatrix} = \sum_{j=1}^{r} \begin{bmatrix} \varphi(v_j^{(1)}) \cdot C_j \\ \vdots \\ \varphi(v_j^{(k)}) \cdot C_j \end{bmatrix}$$
$$= \sum_{j=1}^{r} \begin{bmatrix} \varphi(v_j^{(1)}) \\ \vdots \\ \varphi(v_j^{(k)}) \end{bmatrix} \cdot C_j = \sum_{j=1}^{r} \Phi(v_j) \cdot C_j,$$

where C_j denotes the matrix associated to c_j with respect to the basis B. Finally, as $\mathrm{rank}(A+\tilde{A}) \leq \mathrm{rank}(A) + \mathrm{rank}(\tilde{A})$, we find

$$k = \mathrm{rank}(\Phi(w)) = \mathrm{rank}\left(\sum_{j=1}^{r} \Phi(v_j) \cdot C_j\right) \leq \sum_{j=1}^{r} \mathrm{rank}(\Phi(v_j) \cdot C_j) \leq \sum_{j=1}^{r} \mathrm{rank}(\Phi(v_j)).$$

□

The following theorem is the outcome of some discussions with Prof. Heuberger and characterizes associated matrices. The first part of the proof follows an approach in [1, p. 424, Theorem 6].

Theorem 3.8 (Criterion for associated matrices).
Let $P \in \mathrm{GF}(p)^{k\times k}$ be a quadratic matrix and let $\chi_P(x) := \det(x \cdot I_k - P) \in \mathrm{GF}(p)[x]$ denote its characteristic polynomial.

Then P is the associated matrix to $\omega \in \mathrm{GF}(p^k)$ with respect to some basis if and only if the minimal polynomial of ω is the only prime factor of χ_P over $\mathrm{GF}(p)$ and P is diagonalizable.

Proof. At first, let P be the matrix associated to $\omega \in \mathrm{GF}(p^k)$ with respect to the basis $\{\alpha_1, \ldots, \alpha_k\}$. Let the minimal polynomial of ω be given by the monic polynomial $g(x) = x^\ell + c_1 x^{\ell-1} + \cdots + c_{\ell-1}x + c_\ell$. Due to Theorem 1.9 we know that the powers of ω, i.e. 1, ω, ω^2, ..., $\omega^{\ell-1}$ form a basis of the vector space $\mathrm{GF}(p)(\omega)$ over the field $\mathrm{GF}(p)$. Furthermore, let $\vartheta_1, \ldots, \vartheta_r$ be a basis of $\mathrm{GF}(p^k)$ over $\mathrm{GF}(p)(\omega)$.

Theorem 1.6 – the degree multiplication theorem – states that for the vector space $\mathrm{GF}(p^k)$ over $\mathrm{GF}(p)$ we need $k = r \cdot \ell$ basis vectors. Such a basis is given by

$$B = \{\vartheta_1, \omega\vartheta_1, \omega^2\vartheta_1, \ldots, \omega^{\ell-1}\vartheta_1, \vartheta_2, \ldots, \omega^{\ell-1}\vartheta_2, \ldots, \omega^{\ell-1}\vartheta_r\},$$

which is shown in the proof of Theorem 1.6.

We denote the matrix representation of the linear map $t \mapsto t \cdot \omega$ with respect to the basis B with Q. As can be easily seen, Q is a block diagonal matrix with the $\ell \times \ell$ matrices

$$Q_1 = \cdots = Q_r = \begin{bmatrix} 0 & 1 & & & \\ 0 & & 1 & & \\ \vdots & & & \ddots & \\ 0 & & & & 1 \\ -c_\ell & -c_{\ell-1} & -c_{\ell-2} & \cdots & -c_1 \end{bmatrix}$$

on the main diagonal, as for $t = \omega^i \cdot \vartheta_j$, we have $t \cdot \omega = \omega^{i+1}\vartheta_j$ for $0 \le i < \ell - 1$ and especially

$$t = \omega^{\ell-1}\vartheta_j \text{ implies that } t \cdot \omega = \omega^\ell \vartheta_j = (-c_\ell - c_{\ell-1} \cdot \omega - \ldots - c_1 \cdot \omega^{\ell-1})\vartheta_j.$$

These matrices are *Frobenius companion matrices*, and due to Lemma 1.22 their characteristic polynomial equals g, the minimal polynomial of ω. Furthermore, Q is in Frobenius normal form as described in Theorem 1.23.

Finally, the characteristic polynomial of Q obviously is g^r – and as P is similar to Q, they have the same characteristic polynomial. Therefore, g is the only prime factor of χ_P. Regarding the diagonalizability of Q it suffices to show that the block matrices Q_j are diagonalizable. This holds because the eigenvalues of the companion matrices Q_j of the minimal polynomial of ω are the ℓ conjugates of ω – which are pairwise distinct due to Lemma 1.21.

Now let us assume that $\chi_P = g^r$ for some irreducible g and let P be diagonalizable. We will show that P is associated to a root ω of g with respect to some basis by proving that P has an eigenvector to the eigenvalue ω which has linearly independent components – this directly implies that P is associated to ω.

Due to the given properties of P, we already know the Frobenius normal form of P, which is shaped like the matrix Q from the first part of the proof. This is due to Lemma 1.24 and the remark afterwards. Analogous to above, we construct a basis B of $\mathrm{GF}(p^k)$ over $\mathrm{GF}(p)$ with

$$B = \{\vartheta_1, \omega\vartheta_1, \omega^2\vartheta_1, \ldots, \omega^{\ell-1}\vartheta_1, \vartheta_2, \ldots, \omega^{\ell-1}\vartheta_2, \ldots, \omega^{\ell-1}\vartheta_r\}.$$

This basis consists of $\ell \cdot r = k$ elements. Now consider the vector $v \in \mathrm{GF}(p^k)^k$, whose entries are the elements of B:

$$v = (\vartheta_1, \omega\vartheta_1, \omega^2\vartheta_1, \ldots, \omega^{\ell-2}\vartheta_r, \omega^{\ell-1}\vartheta_r).$$

Some simple computation over $\mathrm{GF}(p^k) = \mathrm{GF}(p)[x]/(g(x))$ shows that v is a eigenvector of Q to the eigenvalue ω. Furthermore, due to construction, v has linearly independent components.

As the rational normal form of P is Q, there is a regular matrix S over $\mathrm{GF}(p)$ such that $Q = S^{-1}PS$. Furthermore, due to our considerations above, we have $Qv = \omega \cdot v$. Therefore,

$$S^{-1}PSv = \omega \cdot v \quad \Rightarrow \quad P(Sv) = \omega \cdot (Sv) \quad \Rightarrow \quad P\tilde{v} = \omega \cdot \tilde{v},$$

where $\tilde{v} = Sv$. As v had linearly independent components, and S is regular, $\tilde{v}$ has to have linearly independent components, too.

Overall, we have shown that under the given assumptions, P has an eigenvector to the eigenvalue ω, whose components are linearly independent over $\mathrm{GF}(p)$ – and therefore, P is associated to ω with respect to the basis consisting of the components of the vector Sv. □

Example 3.5.
We consider the matrix

$$P = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

from Section 3.1.1. We already know that P is structurally equivalent to $\omega = x^2 + x \in \mathrm{GF}(2)[x]/(x^4 + x + 1) \simeq \mathrm{GF}(2^4)$. As ω satisfies

$$\omega^2 + \omega + 1 = (x^2 + x)^2 + (x^2 + x) + 1 = x^4 + x + 1 = 0,$$

and the minimal polynomial of ω surely cannot be of degree 1 (as $\omega \notin \mathrm{GF}(2)$), we have found the minimal polynomial $g_\omega(t) = t^2 + t + 1$ of ω. The characteristic polynomial χ_P has to have degree 4, and its only irreducible factor may be $g_\omega(t)$ – therefore, we find

$$\chi_P(t) = (g_\omega(t))^2 = (t^2 + t + 1)^2 = t^4 + t^2 + 1,$$

which can also be verified easily by computing the characteristic polynomial of P directly.

Example 3.6.
We want to investigate whether the matrix

$$P = \begin{bmatrix} 4 & 2 & 2 & 0 & 4 & 0 & 3 & 0 \\ 1 & 1 & 1 & 2 & 3 & 2 & 2 & 1 \\ 1 & 2 & 0 & 1 & 3 & 1 & 3 & 2 \\ 1 & 3 & 4 & 2 & 1 & 3 & 3 & 3 \\ 1 & 0 & 0 & 3 & 3 & 3 & 2 & 1 \\ 3 & 0 & 0 & 1 & 1 & 3 & 3 & 2 \\ 4 & 2 & 4 & 3 & 2 & 1 & 1 & 3 \\ 0 & 0 & 1 & 4 & 4 & 4 & 1 & 0 \end{bmatrix}$$

is an associated matrix to some $\omega \in \mathrm{GF}(5^8)$. To do so, we compute the characteristic polynomial of P and find its irreducible factors. We find

$$\chi_P(t) = t^8 + t^7 + t^6 + t^4 + 3t^3 + 2t^2 + t + 1 = (t^4 + 3t^3 + t^2 + 2t + 4)^2 = (t^4 - 2t^3 - 4t^2 - 3t - 1)^2$$

where $t^4 + 3t^3 + t^2 + 2t + 4$ is irreducible, which means that Theorem 3.8 can be applied, if P additionally is diagonalizable. This is also the case, as can be verified with Sage by `matrix.rational_form(format="bottom")`, which delivers the rational canonical form of P. In our case, it has the form

$$P_F = \left[\begin{array}{cccc|cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 3 & 4 & 2 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 3 & 4 & 2 \end{array}\right],$$

which is a block diagonal matrix with Frobenius companion matrices on the main diagonal. Therefore we know that P is diagonalizable and due to Theorem 3.8, P is an associated matrix to a root ω of $g_\omega(t) = t^4 - 2t^3 - 4t^2 - 3t - 1$ in $\mathrm{GF}(5^8)$.

A Implementations

A.1 Elementary calculations

```
import time
import itertools

def codeProperties(C):
    """ calculates minimum distance and returns a matrix where the
        entry on position (i,j) is the number of code words of weight
        j belonging to a message of weight i. counting starts at 0. """
    if not isinstance(C, LinearCode):
        raise Exception("ValueError")
    ti = time.time()
    d = C.minimum_distance()
    print("Minimum distance: %s (%s sec.)" %(d, time.time()-ti))
    n = C.length(); k = C.dimension()
    Z = zero_matrix(k+1,n+1)
    msg_iter = itertools.product([0,1], repeat=k)

    for msg in msg_iter:
        mw = sum(msg)
        cw = sum(vector(ZZ, vector(msg)*C.gen_mat()))
        Z[mw, cw] += 1

    print("Calculations finished after %s sec." % (time.time()-ti))
    return Z
```

Listing A.1: Minimum distance and message errors

```
import time

def finField(q):
    """ this method returns GF(q) if q is prime or a list [GF(q), a]
        where a generates GF(q). """
    if(not isinstance(q, Integer) or q < 2 or not q.is_prime_power()):
        raise Exception("ValueError")
    if q.is_prime():
        return GF(q)
    else:
        F.<a> = GF(q, 'a')
        return [F, a]

def fieldExt(F, poly):
    """ generates the extension field K  of F by adjoining a root
        of the irreducible polynomial poly to F (i.e. by constructing
```

```
        the factor ring F[x]/(poly))."""
    if not F.is_field():
        raise Exception("ValueError: Field!")
    R.<x> = PolynomialRing(F, 'x')
    poly = poly(x)
    if (poly(x) not in R or not poly(x).is_irreducible()):
        raise Exception("ValueError: Polynomial!")

    K = R.quotient(poly(x), 'a')
    a = K.gen()
    return [K,a]
```

Listing A.2: Construction of finite fields and field extensions

```
import time

def constCorr(baseField, extField, omega):
    """ matrix representation of multiplication with omega
        over K with respect to the polynomial basis (in reversed order) """
    K = baseField
    F = extField
    if omega not in F:
        raise Exception("ValueError")

    deg = F.degree()/K.degree()
    a = F.gen()
    P = zero_matrix(K, deg)
    for k in range(deg):
        v = vector(a^k * omega)
        for j in range(deg):
            P[deg-1-k, deg-1-j] = v[j]
    return P
```

Listing A.3: Construction of an associated matrix – Algorithm 9

```
import time
load ./constCorr.sage

def checkCorr(baseField, extField, A):
    """ returns the corresponding field element and the transformation
        matrix such that P^{-1} * A * P = B, where B is the matrix
        generated by constCorr(K, F, alpha)."""
    K = baseField
    F = extField
    A = matrix(K, A)
    if not A.minpoly().is_irreducible():
        print("Matrix does not correspond to any field element.")
        return []
    else:
        A = matrix(F, A)
        alpha = - A.minpoly().factor()[0][0][0] #first root of min. poly.
        B = matrix(F,constCorr(K, F, alpha))
        T1 = A.jordan_form(transformation=True)[1]
        T2 = B.jordan_form(transformation=True)[1]
        P = T1 * T2.inverse() # rows of P are a basis of extField.
```

```
21         return [alpha, P]
```

Listing A.4: Check for being an associated matrix – Theorem 3.8

A.2 Decoding and error generation

```
1  import time
2  import itertools
3
4  def generate_cosetleaders(C):
5      """ generation of coset leaders used during syndrome decoding.
6       computing intensive if n-k is large."""
7      if not isinstance(C, LinearCode):
8          raise Exception("ValueError")
9      G = C.gen_mat()
10     H = C.check_mat()
11     n = C.length()
12     k = C.dimension()
13     til = time.time()
14     d_csl = {}
15     j = 0
16     errorit = itertools.combinations(range(n), j)
17     while(len(d_csl) < 2^(n-k)):
18         try:
19             error = generror(errorit.next(), n)
20             syn = vectostr(H*error)
21             if (syn not in d_csl):
22                 d_csl[syn] = error
23         except StopIteration:
24             print("Errors of weight %s processed after %s sec." %(j, time.time
       () - til))
25             j = j+1
26             errorit = itertools.combinations(range(n), j)
27             pass
28     return d_csl
29
30 def generror(pos, n):
31     """error generation. returns a n bit error vector with
32        ones on given positions."""
33     vec = zero_vector(GF(2), n)
34     for p in pos:
35         vec[p] = 1
36     return vec
37
38 def vectostr(v):
39     """conversion from vector (space complex!) to string."""
40     return ''.join(str(w) for w in v)
```

Listing A.5: Coset leader generation

```
1  import time
2  import random
3
4  def addbits(b1, b2):
```

```
    bitsum = ''
    for j in range(len(b1)):
        if b1[j] == b2[j]:
            bitsum = bitsum + '0'
        else:
            bitsum = bitsum + '1'
    return bitsum

class parallelChannel:
    def __init__(self, length, alpha=0.08, beta=1):
        self.length = length
        self.alpha = alpha
        self.beta = beta
        self.w = r.rbeta(self.length, self.alpha, self.beta).sage()
        self.prob = [el_w*0.5 for el_w in self.w]
    def eval_sim(self):
        ev = ''
        for j in range(self.length):
            if self.prob[j] <= random.random():
                ev = ev + '0'
            else:
                ev = ev + '1'
        return ev
    def error_sim(self):
        """returns an error string for a parallel channel."""
        ret1 = self.eval_sim()
        ret2 = self.eval_sim()
        return addbits(ret1, ret2)
    def enrolment(self, degree=10):
        """returns estimated instability per bit."""
        p_est = zero_vector(self.length)
        for k in range(degree):
            p_est = p_est + vector([float(s)/degree for s in self.eval_sim()])
        print("Enrolment of degree %s finished. Estimates:" %degree)
        print(p_est)
        return p_est
```

Listing A.6: Parallel channel error model

```
import time
load ./coset-leaders.sage

class decoder:
    def __init__(self, Code_in, concatenation=False, Code_out=None):
        self.Code_in = Code_in
        self.Code_out = Code_out
        self.concatenation = concatenation
        self.d_csl_in = generate_cosetleaders(Code_in)
        self.k_in = Code_in.dimension()
        self.n_in = Code_in.length()
        self.H_in = Code_in.check_mat()
        if concatenation:
            self.k_out = Code_out.dimension()
            self.n_out = Code_out.length()
            self.d_csl_out = generate_cosetleaders(Code_out)
```

```
            self.r = lcm(self.n_out, self.k_in)
            self.t_in = self.r/self.k_in
            self.t_out = self.r/self.n_out
            self.H_out = Code_out.check_mat()
            print("Decoder for a concatenated code initialized.")
        else:
            print("Decoder for a linear code initialized.")

    def synDecode(self, word):
        """if concatenation=False, simply decode a given
           codeword based on its syndrome. the parameter
           d_csl is the dictionary returned by generate_cosetleaders.
           otherwise: binary concatenation, two-step syndrome decoding!
           caution: codes have to be in standard form!"""
        if not self.concatenation:
            syn = vectostr(self.H_in*word)
            return self.d_csl_in[syn] + word
        else:
            blocks_in = [vector(word[j*self.n_in:(j+1)*self.n_in]) \
                         for j in range(self.t_in)]
            for k in range(self.t_in):
                blocks_in[k] = (self.d_csl_in[vectostr(self.H_in*blocks_in[k])
    ] + blocks_in[k])[0:self.k_in]
            cw_out = vector(sum([list(cw) for cw in blocks_in], []))
            blocks_out = [vector(cw_out[j*self.n_out:(j+1)*self.n_out]) \
                          for j in range(self.t_out)]
            for k in range(self.t_out):
                blocks_out[k] = (self.d_csl_out[vectostr(self.H_out*blocks_out
    [k])] + blocks_out[k])[0:self.k_out]
        return vector(sum([list(cw) for cw in blocks_out], []))
```

Listing A.7: Syndrome decoder

```
import time
load ./errorgen.sage
load ./syndromeDec.sage

class softDecoder:
    def __init__(self, Code, enr_degree=50, erasures=1, alpha=0.08):
        self.alpha = alpha
        self.Code = Code
        self.n = Code.length()
        self.k = Code.dimension()
        self.d_csl = generate_cosetleaders(Code)
        self.parCh = parallelChannel(Code.length(), alpha=self.alpha)
        self.p_est = list(self.parCh.enrolment(enr_degree))
        if (not erasures in ZZ or erasures > self.n - self.k):
            raise Exception("ValueError: erasures")
        else:
            p_est_sorted = sorted(self.p_est, reverse=True)
        lst = [self.p_est.index(p_est_sorted[k]) for k in range(erasures)]
        self.bad_bits = sorted(lst, reverse=True)
        print("Deleted columns %s." %self.bad_bits)
        self.redCode = LinearCode(Code.gen_mat().delete_columns(lst))
        self.dec = decoder(self.redCode)
```

```
        print("Soft decoder for a linear code initialized.")
    def softDecode(self, word):
        word = list(word)
        for pos in self.bad_bits:
            del word[pos]
        word = vector(word)
        return self.dec.synDecode(word)
    def getMessage(self, codeword):
        return self.redCode.gen_mat().solve_left(codeword)
    def selfTest(self, evals):
        """tests the decoder's quality by repeatedly challenging
           it with the erroneous zero word (errors according to
           the related parallel channel added)."""
        corr_dec = 0
        errvec = zero_vector(self.n +1)
        wghvec = zero_vector(self.n +1)
        notif_step = min(10000, evals//20)
        print("Beginning decoder self test."); ti = time.time()
        for k in range(evals):
            err = self.parCh.error_sim()
            word = vector([int(s) for s in err])
            wgh = sum(word)
            wghvec[wgh] = wghvec[wgh] + 1
            if self.softDecode(word) == zero_vector(self.redCode.length()):
                corr_dec += 1
            else:
                errvec[wgh] += 1
            if mod(k, notif_step) == 0:
                print("%s evaluations done after %s, %s" %(k, time.time()-ti,
    n(k/evals)))
        return [evals, corr_dec, n(corr_dec/evals), wghvec, errvec]
```

Listing A.8: Soft decoder

```
import time
load ./errorgen.sage
load ./syndromeDec.sage

class softDecoder:
    def __init__(self, Code, enr_degree=50, erasures=1, parCh=None):
        self.Code = Code
        self.n = Code.length()
        self.k = Code.dimension()
        self.d_csl = generate_cosetleaders(Code)
        if isinstance(parCh, parallelChannel):
            self.parCh = parCh
        else:
            raise Exception("ValueError: parCh")
        self.p_est = list(self.parCh.enrolment(enr_degree))
        if (not erasures in ZZ or erasures > self.n - self.k):
            raise Exception("ValueError: erasures")
        else:
            p_est_sorted = sorted(self.p_est, reverse=True)
        lst = [self.p_est.index(p_est_sorted[k]) for k in range(erasures)]
        self.bad_bits = sorted(lst, reverse=True)
```

```
        print("Deleted columns %s." %self.bad_bits)
        self.redCode = LinearCode(Code.gen_mat().delete_columns(lst))
        self.dec = decoder(self.redCode)
        print("Soft decoder for a linear code initialized.")

    def softDecode(self, word):
        word = list(word)
        for pos in self.bad_bits:
            del word[pos]
        word = vector(word)
        return self.dec.synDecode(word)

    def getMessage(self, codeword):
        return self.redCode.gen_mat().solve_left(codeword)
```

Listing A.9: Modified soft decoder (for parallelization)

A.3 Simulations and examples

```
import time
import itertools
load ./syndromeDec.sage

def simulation(Code_in, Code_out=None, evals=100, p_b=0.08, simple=False):
    """random input, number of correctly decoded words
       gets returned.
       simple = True --> only inner code is considered.
       simple = False --> 2-step decoding.
       p_b --> bitwise error rate."""
    corr_dec = 0; ti = time.time()
    notif_step = min(10000, evals//20)
    if simple:
        dec = decoder(Code_in)
        for k in range(evals):
            err = r.rbinom(dec.Code_in.length(), 1, p_b).sage()
            if dec.synDecode(err) == zero_vector(len(err)):
                corr_dec = corr_dec + 1
            if mod(k, notif_step)==0:
                print("%s evaluations done after %s sec., %s" %(k, time.time-
    ti, n(k/evals)))
        return [evals, corr_dec, n(corr_dec/evals)]
    else:
        dec = decoder(Code_in, concatenation=True, Code_out=Code_out)
        t = lcm(Code_out.length(), Code_in.dimension())/Code_in.dimension()
        for k in range(evals):
            err = r.rbinom(dec.Code_in.length()*t, 1, p_b).sage()
            cw = dec.synDecode(err)
            if cw == zero_vector(len(cw)):
                corr_dec = corr_dec + 1
            if mod(k, notif_step)==0:
                print("%s evaluations done after %s sec., %s" %(k, time.time()
    -ti, n(k/evals)))
        return [evals, corr_dec, n(corr_dec/evals)]

```

```
34
35 def evaluation(Code_in, Code_out=None, p_b=0.08, simple=False):
36     """trying to decode all possible codewords back
37        to the zero word. number of incorrectly decoded
38        words and weight distr. of incorr. dec. words get
39        returned."""
40     ti = time.time(); incorr_dec = 0
41     if simple:
42         dec = decoder(Code_in)
43         errvec = zero_vector(Code_in.length()+1)
44         notif_step = min(10000, 2**Code_in.length()//25)
45         li = itertools.product([0,1], repeat=Code_in.length())
46         for k in range(2**Code_in.length()):
47             word = vector(GF(2), li.next())
48             cw = dec.synDecode(word)
49             if cw != zero_vector(len(cw)):
50                 incorr_dec = incorr_dec + 1
51                 wgh = sum(vector(ZZ, word))
52                 errvec[wgh] = errvec[wgh] + 1
53             if mod(k, notif_step) == 0:
54                 print("%s evaluations done after %s sec., %s" %(k, time.time()
    -ti, n(k/2**Code_in.length())))
55         return [incorr_dec, errvec]
56     else:
57         dec = decoder(Code_in, concatenation=True, Code_out=Code_out)
58         r = lcm(Code_out.length(), Code_in.dimension())
59         t = r/Code_in.dimension()
60         conc_len = t*Code_in.length()
61         errvec = zero_vector(conc_len +1)
62         notif_step = min(10000, 2**conc_len//25)
63         li = itertools.product([0,1], repeat=conc_len)
64         for k in range(2**conc_len):
65             word = vector(GF(2), li.next())
66             cw = dec.synDecode(word)
67             if cw != zero_vector(len(cw)):
68                 incorr_dec = incorr_dec + 1
69                 wgh = sum(vector(ZZ, word))
70                 errvec[wgh] = errvec[wgh] + 1
71             if mod(k, notif_step) == 0:
72                 print("%s evaluations done after %s sec., %s" %(k, time.time()
    -ti, n(k/2**conc_len)))
73         return [incorr_dec, errvec]
```

Listing A.10: Simulation and evaluation

```
1 import time
2 load ./simAndEval.sage
3
4 G_out = HammingCode(3, GF(2)).gen_mat().augment(vector(GF(2), [1,1,1,0]))
5 G_in = G_out
6 C_out = LinearCode(G_out); C_in = LinearCode(G_in)
7
8 G = matrix(GF(2), [[1,0,0,0,0,1,1,1,0,1,1,1,1,0,0,0],
    [0,1,0,0,1,0,1,1,1,0,1,1,0,1,0,0], [0,0,1,0,1,1,0,1,1,1,0,1,0,0,1,0],
    [0,0,0,1,1,1,1,0,1,1,1,0,0,0,0,1]])
```

```
C = LinearCode(G)

### choose which simulation is to be performed!
#print("One single decoding step:")
#print(evaluation(Code_in=C, simple=True))
#print("\n")
print("Two decoding steps:")
print(evaluation(C_in, Code_out=C_out))
```

Listing A.11: Hamming ∘ Hamming – Example 3.2.1

```
import time
load ./simAndEval.sage

G_in = HammingCode(3, GF(2)).gen_mat().augment(vector(GF(2), [1,1,1,0]))
G_out = matrix(GF(2), [[1,0,0,0,1,0,1,1,0,0,1,0], [0,1,0,0,0,1,1,1,1,0,0,0],
    [0,0,1,0,1,1,1,1,0,1,1,1], [0,0,0,1,1,1,1,0,1,1,1,0]])
C_in = LinearCode(G_in); C_out = LinearCode(G_out)

G = matrix(GF(2), [[1,0,0,0,0,1,1,1,1,0,1,1,0,1,0,0,0,0,1,0,1,1,0,1],
    [0,1,0,0,1,0,1,1,0,1,1,1,1,0,0,0,1,0,0,0,0,1,1,1],
    [0,0,1,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,0,0,0],
    [0,0,0,1,1,1,1,0,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1]])
C = LinearCode(G)

print("One single decoding step:")
print(evaluation(Code_in=C, simple=True))
print("\n")
print("Two decoding steps:")
print(evaluation(C_in, Code_out=C_out))
```

Listing A.12: Matrix Code ∘ Hamming – Example 3.2.2

```
import time
load "./sim-and-eval.sage"

G_in = HammingCode(3, GF(2)).gen_mat().augment(vector(GF(2), [1,1,1,0]))
G_out = matrix(GF(2), [[1,0,0,0,0,0,1,1,0,1,1,0], [0,1,0,0,1,0,0,0,1,0,0,1],
    [0,0,1,0,1,1,0,0,1,0,0,0], [0,0,0,1,0,0,1,1,0,0,1,1]])
C_in = LinearCode(G_in); C_out = LinearCode(G_out)

G = matrix(GF(2), [[1,0,0,0,0,1,1,1,0,0,1,1,0,0,1,1,0,1,1,0,0,1,1,0],
    [0,1,0,0,1,0,1,1,1,0,0,0,0,1,1,1,1,0,0,1,1,0,0,1],
    [0,0,1,0,1,1,0,1,1,1,0,0,1,1,0,0,1,0,0,0,0,1,1,1],
    [0,0,0,1,1,1,1,0,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1]])
C = LinearCode(G)

print("One single decoding step:")
print(evaluation(Code_in=C, simple=True))
print("\n")
print("Two decoding steps:")
print(evaluation(C_in, Code_out=C_out))
```

Listing A.13: Matrix Code 2 ∘ Hamming – Example 3.2.3

```
import time
load "./sim-and-eval.sage"


G_out = matrix(GF(2), [[1,0,0,0,1,0,1,1,0,0,1,0], [0,1,0,0,0,1,1,1,1,0,0,0],
    [0,0,1,0,1,1,1,1,0,1,1,1], [0,0,0,1,1,1,1,0,1,1,1,0]])
C_in = ExtendedBinaryGolayCode(); C_out = LinearCode(G_out)
G_in = C_in.gen_mat()

G = G1 * G2
C = LinearCode(G)

print("One single decoding step:")
print(evaluation(Code_in=C, simple=True))
print("\n")
print("Two decoding steps:")
print(evaluation(C_in, Code_out=C_out))
```

Listing A.14: Matrix Code õ Golay – Example 3.2.4

```
import time
import itertools
import multiprocessing
load "./modifiedSoftDec.sage"

def ergSum(erg):
    return [sum([erg[j][0] for j in range(len(erg))]), [
        sum([erg[j][1][0] for j in range(len(erg))]),
        sum([erg[j][1][1] for j in range(len(erg))]),
        sum([erg[j][1][2] for j in range(len(erg))]),
        sum([erg[j][1][3] for j in range(len(erg))])]]

def vectorProduct(G, li):
    if(len(li) == 1):
        return G[li[0]]
    else:
        v1 = G[li[0]]
        v2 = vectorProduct(G, li[1:len(li)])
        return vector(GF(2), [v1[k] * v2[k] for k in range(len(v1))])

def genGeneratorMatrix(m, r): #length 2^m, order r
    G = ones_matrix(GF(2), 1, 2**m)
    he = transpose(matrix(GF(2), list(itertools.product([0,1], repeat=m))))
    G = G.stack(he)
    ind_list = range(2, r+1)
    for j in ind_list:
        choose_list = list(itertools.combinations(range(1, m+1), j))
        for perm in choose_list:
            G = G.stack(vectorProduct(G, perm))
    return G

G = genGeneratorMatrix(4,1) #length: 16, dimension: 5
C = LinearCode(G)
H = C.check_mat()

```

```
time_elapsed = time.time()

def mp_eval_all(nr_evals):
    result = []
    while(len(result) < 125):
        try:
            channel = parallelChannel(length=16, alpha=0.24/(1-0.24))
            decoders = [softDecoder(C, enr_degree=300, erasures=j, parCh=channel) for j in range(4)]
            errorCount = zero_vector(C.length() + 1)
            decodingErrorCount = [zero_vector(C.length() + 1) for j in range(4)]
            startTime = time.time()
            print("Beginning to challenge decoders. Already evaluated: %s" % len(result))
            for k in range(nr_evals):
                err = channel.error_sim()
                word = vector([int(s) for s in err])
                wgh = sum(word)
                errorCount[wgh] += 1
                for j in range(len(decoders)):
                    if decoders[j].softDecode(word) != zero_vector(C.length() - j):
                        decodingErrorCount[j][wgh] += 1
            print("Decoder challenge finished after %s sec." %(time.time()-startTime))
            result.append([errorCount, decodingErrorCount])
        except:
            print("Dimension error..... %s evaluations done." %len(result))
    return ergSum(result)

proc = multiprocessing.Pool(8)
overall_result = proc.map(mp_eval_all, [10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000])
proc.close()
proc.join()

time_elapsed = time.time() - time_elapsed
print("Channel simulations done after %s sec.!" %(time_elapsed))
```

Listing A.15: Reed Muller on parallel channel – performance test (parallelized)

Bibliography

[1] Senon I. Borewicz and Igor R. Šafarevič. *Zahlentheorie*. Birkhäuser Verlag Basel und Stuttgart, 1966.

[2] David S. Dummit and Richard M. Foote. *Abstract Algebra*. Wiley, third edition, 2004.

[3] Gerd Fischer. *Lehrbuch der Algebra*. Vieweg, 2008.

[4] G. David Forney. *Concatenated codes*. Massachusetts Institute of Technology, 1965.

[5] John B. Fraleigh. *A First Course In Abstract Algebra*. Addison Wesley, seventh edition, 2003.

[6] D. R. Hankerson, D. G. Hoffman, D. A. Leonard, C. C. Lindner, K. T. Phelps, C. A. Rodger, and J. R. Wall. *Coding Theory And Cryptography – The Essentials*. Pure And Applied Mathematics, second edition, 2000.

[7] Rudolf Lidl and Harald Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, revised edition, 1994.

[8] W. Müller. Lecture: Codierungstheorie, 2012, summer semester. Alpen-Adria Universität Klagenfurt.

[9] John G. Proakis. *Digital Communications*. McGraw-Hill Higher Education, fourth edition, 2001.

[10] W. A. Stein et al. *Sage Mathematics Software (Version 6.0)*. The Sage Development Team, 2013. `http://www.sagemath.org`.