University of Trento

Department of Mathematics

Ph.D. in Mathematics
XXVI Cycle

Computational techniques for nonlinear codes and Boolean functions

Emanuele Bellini

Supervisor: Prof. Massimiliano Sala

Head of PhD School: Prof. Francesco Serra Cassano

December, 2014

University of Trento

Department of Mathematics

Ph.D. in Mathematics
XXVI Cycle

Computational techniques for nonlinear codes and Boolean functions

Ph.D.Thesis of: Emanuele Bellini

Supervisor: Prof. Massimiliano Sala

Head of PhD School: Prof. Francesco Serra Cassano

December, 2014

Contents

Introduction

The theory of error correcting codes allows to encode data by adding redundancy information to it, in order to be able to correct possible errors arose during the transmission of this encoded data through a noisy channel. The majority of modern communications use the field of bits as alphabet, and messages can be thought as bit sequences of equal length k. Once the number of messages is fixed, an analysis of the noise of the channel provides statistical information on the number and on the kind of errors that may occur during the transmission. Based on this information, to each message, we want to add the minimum possible redundancy allowing us to detect and possibly correct all occurring errors. Clearly a larger redundancy could correct at least the same number of errors but would overload the channel.

From a mathematical point of view, a code can be seen as the image of an injective function f from a subset of $\{0,1\}^k$ of size M to a subset of $\{0,1\}^n$ of the same size, with $n \geq k$. Thus f is also invertible.
One trivial example of f could be a function which simply concatenates a word w to itself 3 times. Once sent through a channel which flips one bit with probability $1/3$, when the encoded word $f(w)$ is received, it will be very likely that if the bits in position $i, 2i, 3i$, for $1 \leq i \leq k$, disagree, then the transmitted bit was the one occurring more often.
Applying f, i.e. *encoding*, and f^{-1}, i.e. *decoding*, should be an "efficient" task, and, given f, it should be "easy" to derive how many errors the code can correct. Furthermore, we do not wish n to be much larger than k, as in the trivial example we described, but at the same time n should be large enough to permit us to correct as many errors as possible.
An efficient encoding/decoding function for which we can efficiently derive the number of correctable errors, can be constructed by imposing some specific algebraic constraints, yielding what are usually known as linear codes. This algebraic constraints, though, severely limit the possible choices of f. In fact, there exist codes which are not linear codes, i.e. do not embed a useful algebraic structure, but which can encode more messages than any linear code and correct the same number of errors. On the other hand, these codes do not have efficient encoding and decoding functions, and

once the encoding function is given it is not easy to determine how many errors these codes can correct.
In this thesis we deal with such "non-linear" codes, and we focus our research into two main problems.
In Part II we deal with the problem of determining acceptable code parameters. In other words, given the number of errors we want to correct and the length n of the encoded message, we want to determine which is the largest number of messages M that can be encoded. Usually it is not possible to give a precise value for M, but only upper or lower limits. In particular, one of our main results is a closed formula for an upper limit (*bound*) of M improving some previous estimates.
In Part III we provide a deterministic method, in some cases faster than a brute force search, to find the number of correctable errors for any code, provided that the code is represented in a particular efficient form. All methods we are aware of solving the same problem are either brute force methods or probabilistic methods. Probabilistic methods are very efficient, but can only be applyed to linear codes. Our result on codes has also an applications in cryptography, since it allows to compute a particular parameter called the *nonlinearity* of Boolean functions. These are functions used in many cryptographic primitives to spread the entropy during encryption.

In more details, this thesis is structured as follows.

Part I is devoted to preliminaries, essential to understand the rest of the manuscript. In particular we start in Chapter 1 with an overview on polynomial system solving, with focus on systems with a finite number of solutions, then we provide an overview on codes in Chapter 2, and an overview on Boolean functions in Chapter 3.

Part II begins with an overview on classical known bounds (Chapter 4), in particular focusing on upper bounds for nonlinear codes. Chapter 5 and 6 contain original results. The first chapter generalizes an upper bound for linear codes, i.e. the Griesmer bound, to an infinite subset of a larger family of codes, called *systematic*. The second chapter presents a new upper bound on the size of nonlinear codes, which improves in many cases the most important classical upper bounds.

Part III faces two important related problems: finding the *minimum weight* (a quantity related to the number of correctable errors) of a nonlinear code (Chapter 7) and finding the nonlinearity of a Boolean function (Chapter 8). We provide original efficient algorithms for both problems, applying similar techniques based on polynomial system solving methods and fast Fourier transforms. In the first case we find

a deterministic algorithm which, in some cases, is faster than brute force, provided the code is represented in a certain form which we prove to be as efficient as the classical representation. In the second case we find a deterministic algorithm of the same complexity of the best known algorithms which solve the same problem.

Part IV lists the Magma code which implement the new bound of Chapter 6, and the algorithms of Chapter 7 and 8.

Part I

Preliminaries

A brief introduction to polynomial system solving

In this chapter we introduce some basic notions and known results from [CLO07] and [ST09]. Some material comes from the lecture notes of the course *Coding Theory* lectured by M. Sala and written by E. Bellini, D. Frapporti, O. Geil, M. Piva, M. Sala.
In particular, we introduce some important tools to solve a generic polynomial system of equations with a finite number of solutions.

We denote by $\mathbb{F}_q$ the field with q elements, where q is a power of a prime. Let $n \geq 1$ be a natural number and let $(\mathbb{F}_q)^n$ be the vector space of dimension n over $\mathbb{F}_q$. We denote by $\mathbb{K}$ any (not necessarily finite) field and by $\overline{\mathbb{K}}$ its algebraic closure.

1.1 Monomial ordering

A *monomial* in $x_1, \ldots, x_r$ is a product of the form

$$x_1^{\alpha_1} \cdot \ldots \cdot x_r^{\alpha_r}$$

where all of the exponents α_j are non negative integers. The sum $\alpha_1 + \ldots + \alpha_r$ is defined to be the *total degree* of this monomial. We denote by $\mathcal{M}(X) = \mathcal{M}$ the set of all monomials in the variables $x_1, \ldots, x_r$.

A *polynomial* f in $x_1, \ldots, x_r$ with coefficients in $\mathbb{K}$ is a finite linear combination of monomials. That is,

$$f = \sum_{\alpha} a_\alpha x^\alpha, \qquad a_\alpha \in \mathbb{K},$$

where $x^\alpha = x_1^{\alpha_1} \cdot \ldots \cdot x_r^{\alpha_r}$ and the sum is over a finite number of m-uples $\alpha = (\alpha_1, \ldots, \alpha_r)$. Then we call a_α the *coefficient* of the monomial x^α, we call $a_\alpha x^\alpha$ a *term*, and we denote by $\deg(f)$ the *total degree* of f which is the maximum $|\alpha| = \alpha_1 + \ldots + \alpha_r$ such that the coefficient a_α is nonzero.

Note that the sum and product of two polynomials is again a polynomial. It is simple to prove that under addition and multiplication, $\mathbb{K}[x_1, \ldots, x_r] = \mathbb{K}[X]$ satisfies all field axioms except for the existence of multiplicative inverses (since, for

example, $1/x$ is not a polynomial). For this reason $\mathbb{K}[X]$, the set of all polynomials in $x_1, \ldots, x_r$ with coefficients in $\mathbb{K}$, is called a *polynomial ring.*

As in the case of univariate polynomials, we would like to be able to arrange the terms of a multivariate polynomial unambiguously, in descending (or ascending) "order". To do this, we have to define a *monomial ordering* $\prec$.

Definition 1.1.1. *A* ***monomial ordering*** $\prec$ *is a binary relation on* $\mathcal{M}$ *such that:*

1. $\forall\ m_1 \neq m_2 \in \mathcal{M}$, *either* $m_1 \prec m_2$ *or* $m_2 \prec m_1$.
   $\forall\ m_1, m_2, m_3 \in \mathcal{M}$, *if* $m_1 \prec m_2$ *and* $m_2 \prec m_3$, *then* $m_1 \prec m_3$.
2. $\forall\ m_1, m_2, m \in \mathcal{M}$ *if* $m_1 \prec m_2$ *then* $m_1 \cdot m \prec m_2 \cdot m$.
3. $1 \prec m, \forall m \in \mathcal{M}, m \neq 1$.

It can be proved that $\prec$ is a well-ordering, i.e. every non-empty subset of $\mathcal{M}$ has a least element.
Now that we have defined monomial ordering, we report some examples. We can suppose that $x_1 \succ \ldots \succ x_r$ and let $m_1, m_2 \in \mathcal{M}$ such that $m_1 = x_1^{\alpha_1} \cdot \ldots \cdot x_r^{\alpha_r}$ and $m_2 = x_1^{\beta_1} \cdot \ldots \cdot x_r^{\beta_r}$.

Lex: *lexicographic order.* We say that $m_1 \prec_{lex} m_2$ if there exists j such that $\alpha_j < \beta_j$ and $\alpha_i = \beta_i$ for $1 \leq i < j \leq r$.

Example 1.1.2. Let $\mathcal{M} = \mathcal{M}[x, y, z]$ and $x \succ y \succ z$. Then

$$x^2 \succ y^4 \text{ and } x^2yz^3 \succ xy^4z.$$

GrLex: *graded lexicographic order* and it is also call *total lexicographic order.* We say that $m_1 \prec_{GrL} m_2$ if $|\alpha| < |\beta|$ or if $|\alpha| = |\beta|$ and $m_1 \prec_{lex} m_2$.

Example 1.1.3. Let $\mathcal{M} = \mathcal{M}[x, y, z]$ and $x \succ y \succ z$. Then

$$x^2 \prec y^4 \text{ and } x^2yz^3 \succ xy^4z.$$

DegRevLex: *graded reverse lexicographic order.* To say that $m_1 \prec_{DRL} m_2$, first of all we compare their total degrees: if $|\alpha| < |\beta|$ then $m_1 \prec_{DRL} m_2$, otherwise we have to compare the total degree of $n_1 = x_1^{\alpha_1} \cdot \ldots \cdot x_{r-1}^{\alpha_{r-1}}$ and $n_2 = x_1^{\beta_1} \cdot \ldots \cdot x_{r-1}^{\beta_{r-1}}$, and so on.

Example 1.1.4. Let $\mathcal{M} = \mathcal{M}[x, y, z]$ and $x \succ y \succ z$. Then

$$x^2 \prec y^4 \text{ and } x^2yz^3 \prec xy^4z \text{ since } x^2y \prec xy^4.$$

Note that DegRevLex is the same to reverse the lexicographic order, that is, $m_1 \prec_{DRL} m_2$ if there exists j that $\alpha_j > \beta_j$ and $\alpha_j = \beta_j$ for $1 \leq j < i \leq r$.

Weighted Degree. We assign a weight $w_i \in \mathbb{N}^*$ to each variable x_i and we denote by $\mathrm{w}(m_1) = \sum_i \alpha_i w_i$ and by $\mathrm{w}(m_2) = \sum_i \beta_i w_i$. We say that $m_1 \prec_w m_2$ if either $\mathrm{w}(m_1) < \mathrm{w}(m_2)$ or $\mathrm{w}(m_1) = \mathrm{w}(m_2)$ and $m_1 \prec_{lex} m_2$.

Example 1.1.5. Let $\mathcal{M} = \mathcal{M}[x, y, z]$ and $x \succ y \succ z$. We assign the weight to each variables $w_x = 2$, $w_y = 1$ $w_z = 3$. Then

$$x^2 \prec y^4 \text{ and } x^2yz^3 \succ xy^4z.$$

We will use the following terminology.

Definition 1.1.6. *Let $\Omega \in \mathbb{N}^r$. Let $f = \sum_{\alpha \in \Omega} a_\alpha x^\alpha$ be a non zero polynomial in $\mathbb{K}[X]$ and let $\prec$ be a monomial ordering. We say that x^β is the **leading monomial** of f if $x^\beta \succ x^\alpha$ for all $\alpha \neq \beta$ such that $\alpha \in \Omega$ and it is denoted by $\mathrm{lm}(f) = x^\beta$. We denote by $\mathbf{T}(f) = a_\beta x^\beta$ the **leading term** of f and by $lc(f) = a_\beta$ the **leading coefficient** of f.*

Given a monomial ordering, it can be proven that the leading monomial, the leading term and the leading coefficient of f are well defined and unique.

Example 1.1.7. Let $f = 4x^2y + xy^3z + 5z$ in $\mathbb{R}[x, y, z]$ and let $\succ_{lex}$ be a lex order. Then $\mathrm{lm}(f) = x^2y$, $lc(f) = 4$ and $\mathbf{T}(f) = 4x^2y$.

1.2 Basic notions and properties of Gröbner bases

In this section we introduce ideals and Gröbner bases.

Definition 1.2.1. *A subset $I \subset \mathbb{K}[X]$ is an **ideal** if*

1. $0 \in I$.
2. *If $f, g \in I$ then $f + g \in I$.*
3. *If $f \in I$ and $h \in \mathbb{K}[X]$ then $fh \in I$.*

Let $f_1, \ldots, f_s$ be polynomials in $\mathbb{K}[X]$. If

$$I = \Big\{ \sum_{i=1}^{s} \lambda_i f_i \mid \lambda_i \in \mathbb{K}[X] \Big\}$$

then I is *finitely generated* by $f_1, \ldots, f_s$ and it is denoted by $I = \langle f_1, \ldots, f_s \rangle$.

An ideal generated by one element is called a *principal ideal.*
A commutative ring A is a *Noetherian* ring if any ideal $I \subset A$ is finitely generated.

Definition 1.2.2. *We define a* ***semigroup ideal*** T *as a subset of* $\mathcal{M}$ *such that for all* $t \in T$, $m \in \mathcal{M}$ *we have* $t \cdot m \in T$.

Let $t_1, \ldots, t_k \in \mathcal{M}$ and set:

$$T = \bigcup_{i=1}^{k} \{\lambda t_i \mid \lambda \in \mathcal{M}\}.$$

Then T is a semigroup ideal of $\mathcal{M}$. We say that T is *generated* by $\{t_1, \ldots, t_k\}$ and we write $T = \langle t_1, \ldots, t_k \rangle$.

Lemma 1.2.3. *Let* $M \subset \mathcal{M}$ *and* $I = \langle m_i \mid m_i \in M \rangle$ *be an ideal. Then a monomial* m *lies in* I *if and only if* m *is divisible by* m_i *for some* $m_i \in M$.

Proof. See Lemma 2 of chapter 2 of [CLO07, §4]. □

Theorem 1.2.4 (Dickson's Lemma)**.** *Every semigroup ideal is generated by a finite set.*

Proof. See Theorem 5 of chapter 2 of [CLO07, §4]. □

In the previous section, we defined the leading term of $f \in I$. For any ideal I, we can define its *ideal of leading terms* $\mathbf{T}(I)$ as the set of leading terms of elements of I. That is,

$$\mathbf{T}(I) = \{\lambda m \mid \text{ there exists } f \in I \text{ with } \mathbf{T}(f) = \lambda m\}.$$

And we denote by $\langle \mathbf{T}(I) \rangle$ the ideal generated by the elements of $\mathbf{T}(I)$.
In a similar way we can define the *ideal of leading monomials* of I, that is,

$$\mathrm{lm}(I) = \{\mathrm{lm}(f) \mid f \in I\} \subset \mathcal{M}.$$

It is clear that $\mathrm{lm}(I)$ is a semigroup ideal.

Note that, if $I = \langle f_1, \ldots, f_k \rangle$, then $\langle \mathbf{T}(f_1), \ldots, \mathbf{T}(f_k) \rangle \subseteq \langle \mathbf{T}(I) \rangle$, but these two ideals may be different and it is the same for $\mathrm{lm}(I)$.

Example 1.2.5. Let $I = \langle f_1, f_2 \rangle$ where $f_1 = x^2 - x$ and $f_2 = xy - y + 1$. We use lexicographic ordering on the monomials in $\mathbb{K}[x, y]$. Then $xf_2 - yf_1 = x$, so $x \in I$. Thus $x = \mathbf{T}(x) \in \langle \mathbf{T}(I) \rangle$ but x is not divisible by $\mathbf{T}(f_1) = x^2$ or $\mathbf{T}(f_2) = xy$. Hence, by Lemma 1.2.3, $x \notin \langle \mathbf{T}(f_1), \mathbf{T}(f_2) \rangle$.

Proposition 1.2.6. *Let* $I \subset \mathbb{K}[X]$ *be an ideal. Then* $\langle \mathbf{T}(I) \rangle$ *is a monomial ideal and there are* $g_1, \ldots, g_k \in I$ *such that* $\langle \mathbf{T}(I) \rangle = \langle \mathbf{T}(g_1), \ldots, \mathbf{T}(g_k) \rangle$.

Proof. See Proposition 3 of chapter 2 of [CLO07, §5]. □

Theorem 1.2.7 (Hilbert Basis Theorem)**.** *Any ideal $I \subset \mathbb{K}[X]$ has a finite generating set.*

Proof. See Theorem 4 of chapter 2 of [CLO07, §5]. □

We just noted, in Example 1.2.5, that not all bases $\{f_1, \ldots, f_k\}$ of an ideal I have the special property that $\langle \mathbf{T}(I) \rangle = \langle \mathbf{T}(f_1), \ldots, \mathbf{T}(f_k) \rangle$. Those bases for which the equality holds give rise to the following definition.

Definition 1.2.8. *Let I be an ideal and $\prec$ be a monomial ordering. We say that $\mathcal{G} = \{g_1, \ldots, g_k\}$ is a **Gröbner basis** for I if $\langle \mathbf{T}(I) \rangle = \langle \mathbf{T}(g_1), \ldots, \mathbf{T}(g_k) \rangle$. We denote by* $\mathrm{GB}(I)$.

Equivalently, $\mathcal{G}$ is a Gröbner basis of I if $\mathcal{G} \subseteq I$ and if for all $f \in I$ there exist $g_i \in \mathcal{G}$ such that $\mathrm{lm}(g_i)$ divides $\mathrm{lm}(f)$.

Theorem 1.2.9 (Buchberger Theorem)**.** *For every ideal $I \subseteq \mathbb{K}[X]$ and for every monomial ordering $\prec$ on $\mathcal{M}$, there exist a Gröbner basis $\mathcal{G}$ for I.*

Proof. See Corollary 6 of chapter 2 of [CLO07, §5]. □

Moreover, there exists an algorithm, that is, Buchberger algorithm [Buc06, Buc98] [CLO07, 2§7] that transforms any finite set of generators for I into a Gröbner basis.

Actually, Gröbner bases computed using the Buchberger algorithm are often larger than necessary. We can eliminate some unneeded generators by using the following lemma.

Lemma 1.2.10. *Let $\mathcal{G}$ be a Gröbner basis for the polynomial ideal I. Let $g \in \mathcal{G}$ be a polynomial such that $\mathbf{T}(g) \in \langle \mathbf{T}(\mathcal{G} \backslash \{g\}) \rangle$. Then $\mathcal{G} \backslash \{g\}$ is also a Gröbner basis for I.*

Proof. See Lemma 3 of chapter 2 of [CLO07, §7]. □

Because of Lemma 1.2.10, we can define a *minimal Gröbner basis* for $I \subseteq \mathbb{K}[X]$ as a Gröbner basis $\mathcal{G}$ for I such that for all $g \in \mathcal{G}$ we have that $lc(g) = 1$ and $\mathbf{T}(g) \notin \langle \mathbf{T}(\mathcal{G} \backslash \{g\}) \rangle$.
Unfortunately, a given ideal I may have many minimal Gröbner bases. But we can define a *special* minimal basis, that we call a *reduced basis*. In this way to any ideal we can associate a unique basis.

Definition 1.2.11. *Let $\mathcal{G} = \{g_1, \ldots, g_k\}$ be a Gröbner basis for I. We say that $\mathcal{G}$ is **reduced** if for all $g \in \mathcal{G}$, $lc(g) = 1$ and no monomial of g divides $\mathbf{T}(g_i)$ where $g_i \neq g$ and $g_i \in \mathcal{G}$.*

Proposition 1.2.12. *Let $I \neq \{0\}$ be a polynomial ideal. Then, for a given monomial ordering, I has a unique reduced Gröbner basis.*

Proof. See Proposition 6 of chapter 2 of [CLO07, §7]. □

It can be proved the following

Proposition 1.2.13. *Let I be an ideal in $\mathbb{K}[X]$ and let $\{g_1, \ldots, g_k\}$ be a reduced Gröbner basis of I with respect to some monomial order. For any $f \in \mathbb{K}[X]$ there exists a unique* remainder *$r \in \mathbb{K}[X]$ such that no term of r is divisible by the leading term of any g_i and such that $f - r$ belongs to I.*
This unique polynomial r, that we indicate with $\mathrm{Nf}(f, I)$, is sometimes called the Normal Form *of f w.r.t I.*

For any ideal I in a polynomial ring $\mathbb{K}[X]$, $X = \{x_1, \ldots, x_r\}$, we denote by $\mathcal{V}(I)$ the *variety of I* in $\overline{\mathbb{K}}$, that is the set of all zeros of I in $\overline{\mathbb{K}}$

$$\mathcal{V}(I) = \{P \in \overline{\mathbb{K}}^r \mid f(P) = 0 \quad \forall\, f \in I\}.$$

Theorem 1.2.14. *Let $I = \langle f_1, \ldots, f_k \rangle$ be an ideal in $\mathbb{K}[X]$ and let $P \in \overline{\mathbb{K}}^r$. Then*

$$f_1(P) = \ldots = f_k(P) = 0 \iff g(P) = 0 \quad \forall\, g \in I.$$

Proof. See Proposition 9 of chapter 2 of [CLO07, §5]. □

Definition 1.2.15. *Let I be an ideal. If the cardinality of $\mathcal{V}(I)$ is finite, then I is called a* 0-***dimensional ideal***.

Theorem 1.2.16 (The Weak Nullstellensatz)**.** *Let $\overline{\mathbb{K}}$ be an algebraically closed field and let $I \subseteq \mathbb{K}[X]$ be an ideal satisfying $\mathcal{V}(I) = \emptyset$. Then $I = \mathbb{K}[X]$.*

Proof. See Theorem 1 of chapter 4 of [CLO07, §2]. □

Definition 1.2.17. *For any $Z \subset \overline{\mathbb{K}}^r$ a set of points, we denote by $\mathcal{I}(Z)$ the* ***vanishing ideal*** *of Z, $\mathcal{I}(Z) \subset \mathbb{K}[X]$, that is, $\mathcal{I}(Z) = \{f \in \mathbb{K}[X] \mid f(P) = 0\ \forall P \in Z\}$.*

Definition 1.2.18. *Let I be an ideal in a polynomial ring $\mathbb{K}[X]$, the* ***radical of*** *I, denote by $\sqrt{I}$ is the set $\sqrt{I} = \{f \in \mathbb{K}[X] \mid f^n \in I$ for some $n \geq 1\}$.*

Note that $I \subseteq \sqrt{I}$. If $I = \sqrt{I}$, then I is *radical*, that is, $f^n \in I$ implies that $f \in I$, for some $n \geq 1$.
It is easy to prove that $\mathcal{I}(Z)$ is radical (Corollary 3 of chapter 4 of [CLO07, §2]).

Theorem 1.2.19 (Hilbert Nullstellensatz)**.** *Let $\mathbb{K}$ be an algebraically closed field. If $I \subseteq \mathbb{K}[X]$ is an ideal, then*

$$\sqrt{I} = \mathcal{I}(\mathcal{V}(I))$$

Proof. See Theorem 6 of chapter 4 of [CLO07, §2]. □

Theorem 1.2.20 (The Ideal-Variety Correspondence)**.** *Let $\mathbb{K}$ be an arbitrary field. If $I_1 \subset I_2$ are ideals, then $\mathcal{V}(I_2) \subset \mathcal{V}(I_1)$ and, similarly, if $\mathcal{V}(I_2) \subset \mathcal{V}(I_1)$ are varieties, then $\mathcal{I}(\mathcal{V}(I_1)) \subset \mathcal{I}(\mathcal{V}(I_2))$*

Proof. See Theorem 7 of chapter 4 of [CLO07, §2]. □

Theorem 1.2.21. *Let $I \subset \mathbb{F}_q[X]$ be an ideal such that $\{x_i^q - x_i \mid 1 \leq i \leq r\} \subseteq I$, then I is 0-dimensional and radical.*

Proof. If $\{x_i^q - x_i \mid 1 \leq i \leq r\} \subseteq I$ it means that $\mathcal{V}(I) \subset \mathbb{F}_q^r$ and then $\#\mathcal{V}(I) \leq |\mathbb{F}_q^r| = q^r$. Thus I is 0−dimensional.
Since $I \subseteq \sqrt{I}$, to prove that I is radical it is sufficient to show that $\sqrt{I} \subseteq I$.
Let $f = a_1 m_1 + \ldots a_n m_n$ where $a_i \in \mathbb{K}$, $m_i \in \mathcal{M}$ such that $m_i = x_1^{\alpha_{1,i}} \cdot \ldots \cdot x_r^{\alpha_{r,i}}$ with $1 \leq i \leq n$. First of all note that $f^q = f \mod I$. In fact, since $a \in \mathbb{F}_q$ we have $a^q = a$ and $m_i^q = m_i \mod I$ since the field equations are in the ideal and so

$$m_i^q = (x_1^{\alpha_{1,i}} \cdot \ldots \cdot x_r^{\alpha_{r,i}})^q = (x_1^q)^{\alpha_{1,i}} \cdot \ldots \cdot (x_r^q)^{\alpha_{r,i}} = x_1^{\alpha_{1,i}} \cdot \ldots \cdot x_r^{\alpha_{r,i}} = m_i$$

If $f \in \sqrt{I}$ then $f^s \in I$ by definition of radical of I, $f^s \in I$ is equivalent to say that $f^s = 0 \mod I$. We can always consider that $s < q$ since, otherwise, we reduce s module q. So $f^s \in I \implies f^s \cdot f^{q-s} \in I$, that is, $f^q = 0 \mod I$ but $f^q = f \mod I$ and so we can conclude that $f \in I$ and $\sqrt{I} \subseteq I$. □

We now define the *escalier* $\mathrm{N}(I)$, which is the set of all the monomials that are not leading monomial of any polynomial in I:

Definition 1.2.22. *The set $\mathrm{N}(I) = \mathcal{M} \backslash \mathrm{lm}(I)$ is called the **Hilbert staircase** or **footprint** or **escalier** of I.*

Let $I \subset \mathbb{K}[X]$ there is a nice and natural connection between the number of zeros of I and the number of points in its footprint w.r.t. any ordering.

Theorem 1.2.23. *Let I be a 0-dimensional radical ideal in $\mathbb{F}_q$. For any monomial ordering we have: $\#\mathcal{V}(I) = \#\mathrm{N}(I)$.*
Moreover, the set

$$\mathcal{B} = \{m + I \mid m \in \mathrm{N}(I)\}$$

constitutes a basis for R as a vector space over $\mathbb{K}$

Proof. See [CLO07], Propotiotion 3 p. 219, Proposition 1 p. 227, Proposition 4 p. 229. □

We consider $I \subset \mathbb{K}[X]$ an ideal such that $\{x_i^q - x_i \mid 1 \leq i \leq r\} \subset I$ and let $R = \mathbb{K}[X]/I$.

Theorem 1.2.24. *Let I be an ideal in $\mathbb{K}[X]$ and let $\prec$ a monomial ordering. The set*

$$\mathcal{B} = \{m + I \mid m \in \mathrm{N}(I)\}$$

constitutes a basis for R as a vector space over $\mathbb{K}$

Proof. See Theorem 5 of [Gei09]. □

1.3 Solving systems of polynomials equations

Solving multivariate polynomial system of equations is a very important issue in applied mathematics, with many applications in area such as coding theory and cryptology.
The Polynomial System Solving problem, sometimes referred to with the acronymous *PoSSo*, is a NP-Hard problem in computer algebra.
One way to solve a system of polynomial equations it to find the corresponding Gröbner basis. The historical algorithm to compute Gröbner bases is the Buchberger algoritm ([Buc06, Buc98], [Mor05], [CLO07, 2§7]). Other algorithms (FGLM [FGLM93], F4 [Fau99], F5 [Fau02], fast FGLM [FM11]) have been proposed to compute Gröbner bases, often more efficiently.
A new trend in the field is to propose dedicated tools to solve structured polynomial systems (for using the symmetries induced by a finite group [FR09] or the bilinear structure [FEDS11] or determinantal ideals [FEDS10]).
Dedicated methods for finite fields have also been proposed [BFP09], [BFP12], or for 0-dimensional ideals [MCD+10], [Mor05] (Algorithm 29.3.1).

In Chapters 7 and 8 we deal with three types of systems of polynomial equations, all with a finite number of or no solutions:

TYPE 1 Multivariate polynomial system

- over the binary field $\mathbb{F}_2$,
- in k variables $x_1, \ldots, x_k$, and
- with r squarefree-polynomials of degree $\leq k$.

TYPE 2 Multivariate polynomial system

- over the rational field $\mathbb{Q}$ (or a prime field $\mathbb{F}_p$ with $p \sim 2^k$),

- in k variables $x_1, \ldots, x_k$, and
- with one dense ($\sim 2^k$ terms) squarefree-polynomial $g(x_1, \ldots, x_k)$ of degree k, plus k "$\mathbb{F}_2$-field equations" $x_1^2 - x_1, \ldots, x_k^2 - x_k$.

TYPE 3 Multivariate polynomial system

- over the rational field $\mathbb{Q}$ (or a prime field $\mathbb{F}_p$ with $p \sim 2^k$),
- in $k+1$ variables $x_1, \ldots, x_k, t$, and
- with one dense ($\sim 2^k$ terms) squarefree-polynomial $g(x_1, \ldots, x_k) - t$ of degree k, plus k "$\mathbb{F}_2$-field equations" $x_1^2 - x_1, \ldots, x_k^2 - x_k$.

In particular, in our case, we know that TYPE 1 and TYPE 2 systems either have no solutions or have a finite number of solutions in $\{0,1\}^k$. Regarding TYPE 3 systems, they always have a finite number of solutions such that $(x_1, \ldots, x_k, t) \in \{0,1\}^k \oplus \mathbb{Z}_n$ for a certain $n \geq k$.
We write one example for each type of system we need to solve.

Example 1.3.1 (TYPE 1)**.** In this case we have $k = 4$ and $r = 11$ with an ideal $I \in \mathbb{F}_2[x_1, \ldots, x_4]/\langle x_1^2 + x_1, x_2^2 + x_2, x_3^2 + x_3, x_4^2 + x_4 \rangle$ such that

$$
\begin{aligned}
I = \{ & x_1x_2x_3x_4 + x_2x_3x_4, \\
& x_1x_2x_3x_4 + x_1x_3x_4, \\
& x_1x_3x_4 + x_2x_3x_4, \\
& x_1x_2x_3x_4 + x_1x_2x_3 + x_1x_2x_4 + x_1x_2, \\
& x_1x_2x_3x_4 + x_1x_2x_3 + x_2x_3x_4 + x_2x_3, \\
& x_1x_2x_3x_4 + x_1x_2x_3 \\
& x_1x_2x_3x_4, \\
& x_1x_2x_3 + x_1x_3, \\
& x_1x_2x_3x_4 + x_1x_2x_4 + x_2x_3x_4 + x_2x_4, \\
& x_1x_2x_3 + x_1x_3x_4, \\
& x_1x_2x_3x_4 + x_1x_2x_3 + x_1x_3x_4 + x_1x_3 + x_2x_3x_4 + x_2x_3 + x_3x_4 + x_3 \} .
\end{aligned}
$$

Note that here the equations $x_1^2 + x_1, x_2^2 + x_2, x_3^2 + x_3, x_4^2 + x_4$ are implicit, since we are working over the affine algebra $\mathbb{F}_2[x_1, \ldots, x_4]/\langle x_1^2 + x_1, x_2^2 + x_2, x_3^2 + x_3, x_4^2 + x_4 \rangle$. The solutions of the systems are

$$
\mathcal{V} = \{(0,0,0,0), (0,0,0,1), (0,0,1,1), (0,1,0,0), (1,0,0,0), (1,0,0,1), (1,1,0,1)\} .
$$

Example 1.3.2 (TYPE 2)**.** In this case we have $k = 4$ and an ideal $I \in \mathbb{Q}[x_1, \ldots, x_4]$ such that

$$\begin{aligned} I = \{&x_1^2 - x_1, x_2^2 - x_2, x_3^2 - x_3, x_4^2 - x_4, \\ &- 4x_1x_2x_3x_4 + 4x_1x_2x_3 - 2x_1x_2x_4 - 3x_1x_2 + 8x_1x_3x_4 + \\ &- 4x_1x_3 - 4x_1x_4 + 6x_1 + 2x_2x_3x_4 - 4x_2x_3 + 4x_2 - 5x_3x_4 + 7x_3 + 4x_4 - 7\}. \end{aligned}$$

The solutions of this system are

$$\mathcal{V} = \{(0,0,1,0), (0,1,1,0), (1,1,0,0)\}.$$

Example 1.3.3 (TYPE 3)**.** In this case we have $k = 4$ and an ideal $I \in \mathbb{Q}[x_1, \ldots, x_4]$ such that

$$\begin{aligned} I = \{&x_1^2 - x_1, x_2^2 - x_2, x_3^2 - x_3, x_4^2 - x_4, \\ &- 4x_1x_2x_3x_4 + 4x_1x_2x_3 - 2x_1x_2x_4 - 3x_1x_2 + 8x_1x_3x_4 + \\ &- 4x_1x_3 - 4x_1x_4 + 6x_1 + 2x_2x_3x_4 - 4x_2x_3 + 4x_2 - 5x_3x_4 + 7x_3 + 4x_4 - t\}. \end{aligned}$$

The solutions of this system are

$$\begin{aligned} \mathcal{V} = \{&(0,0,0,0,0), (0,0,0,1,4), (0,0,1,0,7), (0,0,1,1,6), \\ &(0,1,0,0,4), (0,1,0,1,8), (0,1,1,0,7), (0,1,1,1,8), \\ &(1,0,0,0,6), (1,0,0,1,6), (1,0,1,0,9), (1,0,1,1,12), \\ &(1,1,0,0,7), (1,1,0,1,5), (1,1,1,0,10), (1,1,1,1,9)\}. \end{aligned}$$

1.4 The 0-dimensional case

From a practical point of view, it is much faster to compute a Gröbner basis for a degree ordering such as the degree reverse lexicographic (DegRevLex) order than for a lexicographic order. For 0-dimensional systems, it is usually less costly to first compute a DegRevLex-Gröbner basis, and then to compute the Lex-Gröbner basis using a change ordering algorithm such as FGLM [FGLM93]. This strategy, called *zero-dim solving*, is performed blindly in modern computer algebra softwares such as MAGMA or MAPLE. This is convenient for the user, but can be an issue for advanced users. In general, a polynomial system of equations with a finite number of solutions may yield more efficient algorithms to solve it. In particular, this is the case when the solutions of the system lie in a finite field, as is our case.
The polynomial system solving problem over finite fields is sometimes referred to as PoSSo_q.
From a complexity-theoretical point of view, PoSSo_q is NP-Hard independently of

the size q [GJ79]. Thus, any algorithm for PoSSo_q should be exponential in the worst case. However, this does not exclude that large family of PoSSo_q instances can be solved in sub-exponential or polynomial complexity. In addition, the exact exponent occurring in algorithms of exponential complexity is often a critical question in applications.
We now present some approaches which, according to the author, deserve consideration when trying to solve a system of polynomial equations with a finite number of solutions.

1.4.1 Representation of 0-dim. ideals

The following notions can be found in [Mor05].
Let $X = x_1, \ldots, x_k$. Let $\mathsf{J} \subset \mathbb{K}[X]$ be a zero-dimensional ideal, $\deg(\mathsf{J}) = s$, and denote $\mathsf{A} := \mathbb{K}[X]/\mathsf{J}$ the corresponding quotient algebra, which satisfies $\dim_{\mathbb{K}}(\mathsf{A}) = s$.

For any $f \in \mathbb{K}[X]$, we will denote $[f] \in \mathsf{A}$ its residue class modulo J and Φ_f the endomorphism $\Phi_f : \mathsf{A} \to \mathsf{A}$ defined by

$$\Phi_f([g]) = [fg] \forall [g] \in \mathsf{A}.$$

Natural representation

If we fix any $\mathbb{K}$-basis $\mathbf{b} = \{[b_1], \ldots, [b_s]\}$ of A so that $\mathsf{A} = \text{span}_{\mathbb{K}}(\mathbf{b})$, then for each $g \in \mathbb{K}[X]$, there is a unique (row) vector, the *Gröbner description of* g,

$$\mathbf{Rep}(g, \mathbf{b}) := (\gamma(g, b_1, \mathbf{b}), \ldots, \gamma(g, b_s, \mathbf{b})) \in \mathbb{K}^s$$

which satisfies

$$[g] = \sum_j \gamma(g, b_j, \mathbf{b})[b_j]$$

and the endomorphism Φ_f is naturally represented by the square matrix

$$M([f], \mathbf{b}) = \big(\gamma(fb_i, b_j, \mathbf{b})\big) : \Phi_f(b_i) = [fb_i] = \sum_j \gamma(fb_i, b_j, \mathbf{b})[b_j].$$

Definition 1.4.1. *A* natural representation *of* J *is the assignement of*

- *a* $\mathbb{K}$*-basis* $\mathbf{b} = \{[b_1], \ldots, [b_s]\} \subset \mathsf{A}$ *and*
- *the square matrices* $A_h := \left(a_{ij}^{(h)}\right) = M([x_h], \mathbf{b})$ *for each* $h, 1 \leq h \leq k$.

Remark that, for each $f(x_1, \ldots, x_k) \in \mathbb{K}[X]$, $M([f], \mathbf{b}) = f(A_1, \ldots, A_k)$.
An equivalent (via the remark above) definition of natural representation can require the further assignement of

- s^3 values $\gamma_{ij}^{(l)} \in \mathbb{K}$ such that

$$[q_i q_j] = \sum_l \gamma_{ij}^{(l)} [q_l]$$

for each $i, j, l, 1 \leq i, j, l \leq s$.

This notion was introduced in [Tra92b, Tra92a] and reconsidered in [AMM03], [Mor05, Definition 29.3.3] under the name of *Gröbner representation.*
The endomorphism Φ_f and its represention $M([f], \mathbf{b})$ were introduced, with f a linear form, in [AS88] as a tool for efficient solving 0-dimensional ideals.
If J is given by its Gröbner basis wrt a term-ordering $<$ its natural (actually: "linear" with the definition below) representation can be obtained via [FGLM93, Procedure 3.1].
If J is an affine complete intersection defined by r polynomials a natural representation of it can be efficiently computed via Cardinal-Mourren Algorithm [J.P93, Mou05]. We will assume that both the input and the output ideals of the algorithm are given via a natural representation.

A Gröbner-free approach to natural representation

Recalling that a set $\mathrm{N} \subset \mathcal{M}$ is called an *escalier* if it is an *order ideal*, *i.e.* if for each $\lambda, \tau \in \mathcal{M}$, $\lambda\tau \in \mathrm{N} \implies \tau \in \mathrm{N}$ and properly extending [Mor05, Definition 29.3.3] we set

Definition 1.4.2. *A natural representation is called a* linear representation *iff* $\mathbf{q} = \mathrm{N}$ *is an* escalier.

If $\mathrm{N} = \{v_1, \ldots, v_s\}$ is an *escalier* then [Mor09] $\mathsf{T} := \mathcal{M} \setminus \mathrm{N}$ is a *semigroup ordering*, *i.e.* $\tau \in \mathsf{T} \implies \tau\lambda \in \mathsf{T}$ for each $\lambda, \tau \in \mathcal{M}$; we set $\mathsf{G} := \{\tau_1, \ldots, \tau_u\} \subset \mathsf{T}$ the minimal basis of T.

1.4.2 Traverso's Algorithm

Traverso introduced his algorithm in a talk at MEGA-1992, [Tra92b] and in [Tra92a], in a scenario related to Gröbner bases computation of a zero-dimensional ideal I. The assumption is that, in the course of the computation, one produces an escalier $\mathrm{N} \supset \mathrm{N}_<(\mathsf{J})$ and a finite list $g_1, \ldots, g_r$ of S-polynomials to be reduced.
The setting was reformulated in [AMM03], [Mor05, Algorithm 29.3.8] as follows: given a zero-dimensional ideal $\mathsf{I} \subset \mathbb{K}[X]$ via its natural representation

$$\mathbf{b} = \{b_1, \ldots, b_s\}, b_1 = 1, \mathbf{M} := \left\{ \left(a_{lj}^{(h)} \right), 1 \leq h \leq k \right\},$$

and a finite set of elements $F := \{g_1, \ldots, g_r\} \subset \mathbb{K}[X]$, given via their Gröbner descriptions

$$\mathsf{c}^{(i)} = (c_1^{(i)}, \ldots, c_s^{(i)}), c_j^{(i)} = \gamma(g_i, b_j, \mathbf{b}) \forall i, j, 1 \leq i \leq r, 1 \leq j \leq s,$$

so that $g_i - \sum_{j=1}^{s} c_j^{(i)} b_j \in \mathsf{I}$, for each i, compute with good complexity the linear representation of the ideal $\mathsf{J} := \mathsf{I} \cup \langle F \rangle$.

The basic idea of the algorithm (Algorithm 1) is the following: if we consider an element $g \in F$, having the Gröbner description

$$g - \sum_{j=1}^{\iota} c_j b_j \in \mathsf{I}, \quad c_\iota \neq 0,$$

and we enlarge I by adding g to it, then we obtain the relation

$$b_\iota \equiv -\sum_{j=1}^{\iota-1} c_\iota^{-1} c_j b_j \bmod \mathsf{I} \cup \{g\};$$

the decomposition $\mathbb{K}[X] = \mathsf{I} \oplus \mathrm{span}_{\mathbb{K}}(\mathbf{b})$ of $\mathbb{K}[X]$ into disjoint $\mathbb{K}$-vectorspaces is then transformed into

$$\mathbb{K}[X] = (\mathsf{I} \cup \{g\}) \oplus \mathrm{span}_{\mathbb{K}}(\mathbf{q} \setminus \{q_\iota\}),$$

and we only have to substitute, in each Gröbner description $\sum_{j=1}^{s} d_j b_j$ of the polynomials g_i and $x_h b_l$ — which are respectively encoded in the vectors $\mathsf{c}^{(i)}$ and in the rows $\left(a_{l1}^{(h)}, \ldots, a_{ls}^{(h)}\right)$ of the matrices of $\mathbf{M}$ — the instances of b_ι with $-\sum_{j=1}^{\iota-1} c_\iota^{-1} c_j b_j$ thus getting $\sum_j (d_j - c_\iota^{-1} c_j d_\iota) b_j$.
Since J is an ideal, the inclusion in it of g implies that J necessarily contains also the polynomials $x_h g$; note that, if the current natural representation is

$$(\mathbf{b}', \mathbf{M}') : \mathbf{b}' := \{b'_1, \ldots, b'_\sigma\}, \mathbf{M}' = \mathbf{M}(\mathbf{b}') := \left\{ \left(d_{lj}^{(h)} \right) \right\}$$

and $g = \sum_{l=1}^{s} c_l b'_l$ then

$$x_h g = \sum_{l=1}^{s} c_l x_h b'_l = \sum_{j=1}^{s} \left(\sum_{l=1}^{s} c_l d_{lj}^{(h)} \right) b_j$$

which must be inserted in the list F in order to be treated in the same way.
At termination, if $I \subset \{1, \ldots, n\}$ denotes the set of indices of the elements b_j which have not being removed from $\mathbf{b}$ in this procedure, then J is described by the natural representation

$$\mathbf{b}' = \{b_j, i \in I\}, \mathbf{M}' = \{ \left(a_{lj}^{(h)} \right), l, j \in I, 1 \leq h \leq n \}.$$

Note that Traverso's Algorithm needs to perform at most s **While**-loops, each costing $\mathcal{O}(ns^2)$.

Algorithm 1 (Traverso) To compute the natural representation of $\mathsf{J} := \mathsf{I} \cup \langle g_1, \ldots, g_r \rangle \subset \mathbb{K}[X]$ from the natural representation of $\mathsf{I} \subset \mathbb{K}[X]$.

Input: $\mathsf{I} \subset \mathbb{K}[X]$, a zero-dimensional ideal, $\deg(\mathsf{I}) = s$
$\mathbf{b} := \{[b_1], \ldots, [b_s]\} \subset \mathsf{A} := \mathbb{K}[X]/\mathsf{I}$, $b_1, \ldots, b_s \in \mathbb{K}[X]$
$\mathbf{M} := \left\{ \left(a_{ij}^{(h)} \right) = M([x_h], \mathbf{b}) \in \mathbb{K}^{s^2}, 1 \leq h \leq k \right\}$
$(\mathbf{b}, \mathbf{M})$, natural representation of I
$\mathsf{J} := \mathsf{I} \cup \langle g_1, \ldots, g_r \rangle \subset \mathbb{K}[X], \sigma := \deg(\mathsf{J})$, $[g_i] := \sum_{j=1}^{s} c_j^i [b_j]$, for each $i, 1 \leq i \leq r$
$\mathsf{B} := \{\mathsf{c}^{(1)}, \ldots, \mathsf{c}^{(r)}\}, \mathsf{c}^{(i)} := (c_1^i, \ldots, c_s^i) \in \mathbb{K}^s$

Output: $\mathbf{b}, \mathbf{M}$ natural representation of J

while $\mathsf{B} \neq \emptyset$ **do**
  Choose $\mathsf{c} := (c_1, \ldots, c_s) \in \mathsf{B}$
  $\mathsf{B} := \mathsf{B} \setminus \{\mathsf{c}\}$
  $\mathsf{B} := \mathsf{B} \cup \{\mathsf{cM} : \mathsf{M} \in \mathbf{M} \wedge \mathsf{cM} \neq (0, \ldots, 0)\}$
  $\iota := \max\{j \in \{1, \ldots, s\} : c_j \neq 0\}$
  Remove $[b_\iota]$ from $\mathbf{b}$;
  $s := s - 1$
  for $h = 1..k$ **do**
    $\hat{\mathsf{a}}^{(h)} := (a_{\iota 1}, \ldots, a_{\iota,\iota-1}, a_{\iota,\iota+1}, \ldots, a_{\iota,s})$
    Remove ι-th column and row from $A_h \in \mathbf{M}$
  $\hat{c} := c_\iota$
  Remove ι-th component from c
  for $h = 1..k, j = 1..s, l = 1..s$ **do**
    $a_{lj}^{(h)} := a_{lj}^{(h)} - \hat{c}^{-1} c_j \hat{\mathsf{a}}_l^{(h)}$
  $\mathsf{B}' := \mathsf{B}, \mathsf{B} := \emptyset$
  for $\mathsf{d} := (d_1, \ldots, d_{s+1}) \in \mathsf{B}'$ **do**
    $\hat{d} := d_\iota$
    Remove ι-th component from d
    for $j = 1..s$ **do**
      $d_j := d_j - \hat{c}^{-1} c_j \hat{d}$
    if $\mathsf{d} \neq (0, \ldots, 0)$ **then**
      $\mathsf{B} := \mathsf{B} \cup \{\mathsf{d}\}$
return $\mathbf{b}, \mathbf{M}$

Example 1.4.3. Let $k = 2, \mathbb{K} = \mathbb{F}_2, \mathsf{I} = \langle x_1^2 + x_1, x_2^2 + x_2 \rangle, x_1 > x_2$ with graded reverse lexicographical monomial ordering.

The basis $\mathbf{b} = \mathrm{N}(\mathsf{I})$ of $\mathbb{K}[x_1, x_2]/\mathsf{I}$ is

$$\mathbf{b} = \{q_1, q_2, q_3, q_4\} = \{1, x_2, x_1, x_1x_2\}\,.$$

Suppose we want to find the basis $\mathbf{b}'$ of the algebra $\mathbb{K}[x_1, x_2]/(\mathsf{I} \cup \mathsf{J})$, where $\mathsf{J} = \langle g_1, g_2\rangle = \langle x_2 + 1, x_1x_2 + x_1 + x_2 + 1\rangle$.
Consider $g_1 = x_2 + 1 = 0$ as a new relation, i.e. $\mathsf{I} = \mathsf{I} \cup \langle g_1\rangle$. This means that from now on, whenever we find x_2 in elements of $\mathbf{b}$ and J we can apply the substitution $x_2 = 1$.
We remove g_1 from I, i.e.

$$\mathsf{J} = \mathsf{J} \setminus \{g\} = \langle x_1x_2 + x_1 + x_2 + 1\rangle\,.$$

We update the base $\mathbf{b}$

$$\begin{aligned} q_1 &= 1 \\ q_2 &= x_2 \text{ reduces to } 1 \\ q_3 &= x_1 \\ q_4 &= x_1x_2 \text{ reduces to } x_1\,. \end{aligned}$$

Thus now $\mathbf{b} = \{q_1, q_3\} = \{1, x_1\}$.
We update the polynomial in $\mathsf{J} = \langle x_1x_2 + x_1 + x_2 + 1\rangle$

$$x_1x_2 + x_1 + x_2 + 1 \text{ reduces to } x_1 + x_1 + 1 + 1 = 0\,.$$

Thus now $\mathsf{J} = \emptyset$, i.e. we can stop the computation and return $\mathbf{b} = \{1, x_1\}$, which shows that the polynomial system

$$\begin{cases} x_1^2 + x_1 = 0 \\ x_2^2 + x_2 = 0 \\ x_2 + 1 = 0 \\ x_1x_2 + x_1 + x_2 + 1 = 0 \end{cases}$$

has only two solutions over $(\mathbb{F}_2)^2$, which happen to be $(0, 1), (1, 1)$.

We now report the same example with vectorial notation.

Example 1.4.4. Let $\mathsf{I} = \langle x_1^2 + x_1, x_2^2 + x_2\rangle \in \mathbb{F}[x_1, x_2], x_1 > x_2$ with respect to DegRevLex order.
The linear representation of I is given by

$$\mathbf{b} = \{1, x_2, x_1, x_1x_2\}$$

$$\mathbf{M} = \{x_1\mathbf{b}, x_2\mathbf{b}\} = \left\{ \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right\}.$$

Let $g_1 = x_2 + 1, g_2 = x_1x_2 + x_1 + x_2 + 1$ and let us compute the linear representation of $\mathsf{J} = \mathsf{I} \cup \langle g_1, g_2 \rangle$ using Traverso's algorithm.
The linear descriptions of g_1, g_2 are

$$\mathsf{B} = \{(1,1,0,0),(1,1,1,1)\}\,.$$

Since $\mathsf{B} \neq \emptyset$ we choose $c = (1,1,0,0)$. We have

$$\begin{aligned}\mathsf{B} &= \mathsf{B} \setminus \{\mathsf{c}\} \cup \{\mathsf{cM} : \mathsf{M} \in \mathbf{M}\} = \{(1,1,1,1),(0,0,1,1)\}\\ \iota &= \max\{j \in \{1,\ldots,s\} : c_j \neq 0\} = 2\,.\end{aligned}$$

We can remove the second element x_2 from $\mathbf{b}$, and update $\mathbf{M}$:

$$\begin{aligned}\mathbf{b} &= \{1, x_1, x_1x_2\}\\ \mathbf{M} &= \left\{\begin{pmatrix}0&1&0\\0&1&0\\0&0&1\end{pmatrix}, \begin{pmatrix}1&0&0\\0&0&1\\0&0&1\end{pmatrix}\right\}\\ \mathsf{c} &= (1,0,0)\,.\end{aligned}$$

Since we have that

$$\begin{aligned}(1,1,1,1) &\text{ reduces to } (0,1,1)\\ (0,0,1,1) &\text{ reduces to } (0,1,1)\end{aligned}$$

then $\mathsf{B} = \{(0,1,1)\}$, which is still not empty.
We choose $c = (0,1,1)$.
We have

$$\begin{aligned}\mathsf{B} &= \mathsf{B} \setminus \{\mathsf{c}\} \cup \{\mathsf{cM} : \mathsf{M} \in \mathbf{M}\} = \{(0,1,1)\}\\ \iota &= \max\{j \in \{1,\ldots,s\} : c_j \neq 0\} = 3\,.\end{aligned}$$

We can remove the third element x_1x_2 from $\mathbf{b}$, and update $\mathbf{M}$:

$$\begin{aligned}\mathbf{b} &= \{1, x_1\}\\ \mathbf{M} &= \left\{\begin{pmatrix}0&1\\0&1\end{pmatrix}, \begin{pmatrix}1&0\\0&1\end{pmatrix}\right\}\\ \mathsf{c} &= (1,0)\,.\end{aligned}$$

Since we have that

$$(0,1,1) \text{ reduces to } (0,0)$$

then $\mathsf{B} = \emptyset$, and we are done, returning $\mathbf{b}, \mathbf{M}$ as linear representation of $\mathsf{J} = \mathsf{I} \cup \langle x_2 + 1, x_1x_2 + x_1 + x_2 + 1\rangle$.

Determining if a solution exists

In Chapter 7 and 8 we need to know if a system of polynomial equations has a solution or not, rather than finding all the solutions. This is somehow a simpler problem and can be solved with a simpler version of Traverso's algorithm. The idea is briefly described in Algorithm 2, where $\mathbf{T}(g)$ denotes the leading term of g and $q(g) = g - \mathbf{T}(g)$.

Algorithm 2 To know if $1 \in \mathsf{J} := \langle x_1^q - x_1, \ldots, x_k^q - x_k, g_1, \ldots, g_r \rangle \subseteq \mathbb{K}[X]$

Input: $S := \{g_1, \ldots, g_r\}, g_i \in \mathbb{K}[X]$
Output: TRUE if $1 \in \mathsf{J}$, FALSE otherwise

```
1:  N := N(J)
2:  R := ∅
3:  for g ∈ S do
4:     Remove g from S
5:     Add (T(g), q(g)) to R
6:     for f ∈ S do
7:        Replace T(g) with q(g) in f
8:     for (t, q) ∈ R do
9:        Replace T(g) with q(g) in q
10:    for x ∈ X do
11:       Compute h := xg  mod ⟨x_1^q − x_1, ..., x_k^q − x_k⟩
12:       if h contains a term t such that t ∉ N then
13:          Find (t, q) in R // such pair must exists in R
14:          Replace t with q in h
15:    Remove T(g) from N
16: return  TRUE if N = ∅, FALSE otherwise
```

1.4.3 Hybrid approach

In [BFP09], the authors present a hybrid approach which can improve the way of solving zero-dimensional multivariate systems over finite fields with at least 2^2 elements. This approach uses Gröbner bases techniques and exhaustive search. A more accurate analysis of the hybrid approach is given in [BFP12] by the same authors.
In general, when we want to solve a system which has coefficients over a finite field $\mathbb{F}_q$, we can always find all the solutions in the ground field by exhaustive search. The complete search should take $O(q^k)$ operations if k is the number of variables. The idea of the hybrid approach is to mix exhaustive search with Gröbner bases computations. Instead of computing one single Gröbner basis of the whole system, we compute the

Gröbner bases of q^r subsystems obtained by fixing r variables. The intuition is that the gain obtained by working on systems with less variables may overcome the loss due to the exhaustive search on the fixed variables.
The main problem is to realize if such a trade-off may exists and in that case to choose the best one. That is to choose properly the value of r making the complexity of the hybrid approach minimal. If $\mathcal{C}_{GB}(A)$ is the complexity of solving a Gröbner basis using algorithm A, then the complexity of the hybrid approach is clearly

$$O(q^r \mathcal{C}_{GB}(A)) .$$

In [BFP09] an algorithm to find the best trade-off when using F_5 algorithm to solve each Gröbner basis is given.

1.4.4 XL family of algorithms

One particular algorithm has received considerable attention from the cryptographic community: the XL algorithm [CKPS00] (and its several variants, e.g., [Cou04], [CP03], [CP02]) was originally proposed by cryptographers to tackle problems arising specifically from cryptology. In particular XL was introduced as an efficient algorithm for solving polynomial equations in case a single solution exist.
Other more general variants of XL algorithm, such as MutantXL [BDMM09],[BCDM10], MXL_2 [MMDB08], MXL_3 [MCD+10], have been proposed.
Though, in [ACFP12], it is claimed that the XL family of algorithms can be simulated using redundant variants of F_4 algorithm.

1.4.5 Boolean polynomial systems

In [BD09], the authors introduce a specialized data structure for Boolean polynomials based on zero-suppressed binary decision diagrams(ZDDs), which are capable of handling these polynomials more efficiently with respect to memory consumption and also computational speed. Furthermore, they concentrate on high-level algorithmic aspects, taking into account the new data structures as well as structural properties of Boolean polynomials. For example, a new useless-pair criterion for Gröbner basis computations in Boolean rings is introduced.
The authors provide an entire framework, i.e. a C++ library called *PolyBoRi* (*Poly*nomials over *Bo*olean *Ri*ngs), to efficiently compute Gröbner basis over Boolean polynomials.
The authors point out that the advantage of PolyBoRi grows with the number of variables.

1.4.6 Magma approach

The software Magma implements various optimized versions of Buchberger algorithm and F_4 algorithm, see [CBFS13] for details. Visit also [Ste13] for some practical timings.
In particular, since Version 2.15, a special type of polynomial ring is available: the boolean polynomial ring in k variables. Such a ring is a multivariate polynomial ring defined over $\mathbb{F}_2$ but such that all monomials are reduced modulo the field relations $x_i^2 = x_i$ for each $1 \leq i \leq k$ (so a bit vector representation can be used for monomials). As we have already mentioned, the ring is the quotient algebra $\mathbb{F}_2[x_1, \ldots, x_k]/\langle x_1^2 + x_1, \ldots, x_k^2 + x_k \rangle$. This particular structure allows very fast computations, though it is not declared which particular algorithms are used in this case. Furthermore, since Version 2.20, Magma includes a new *dense variant* of the F_4 algorithm.
Quoting from Magma documentation [CBFS13]:

> The dense variant is currently only practically applicable to input systems over a finite field where the input polynomials are considered "dense"; that is, if the input polynomials are written as a matrix with columns labeled by the monomials, then the input matrix should be dense. Equivalently: if the field size is q and the set of all monomials occurring in the input has size m, then the number of monomials in each input polynomial should be reasonably close to (1 - 1/q)m.
> Also, according to Magma documentation, a new experimental optional heuristic which can be selected for the algorithm when solving systems of equations over GF(2), called the **Reduction Heuristic**, which can give an even greater reduction in time and memory usage for some large examples.
> The Reduction Heuristic is a new experimental heuristic which can be selected in the dense variant of F4 when attempting to solve certain kinds of systems of equations over GF(2) where it is assumed that there is a very small number of solutions, so the Groebner basis will consist of mostly linear polynomials or collapse to 1 when there is no solution. The heuristic attempts to reduce the size of the matrices involved in the linear algebra phase of each F4 step. When the heuristic is selected, the run may simply fail, but when it succeeds, it often saves significant time and memory usage. The kinds of systems for which the saving in time and memory usage tends to be greatest are those such that in the F4 steps of maximal degree D, the number of S-polynomials is relatively small compared to the total number of monomials of degree D.
> Currently the Reduction Heuristic depends on a manual choice by the user of

a numerical parameter M. Thus if B is the sequence of input polynomials, to select the Reduction Heuristic with parameter M, one should currently invoke the algorithm with something like the following:

```
GroebnerBasis(B, D: ReductionHeuristic := M) ;
```

where M is the expected maximal degree reached in the computation.

To understand how to correctly choose M, in particular for the binary case, please see [Ste13] or [CBFS13].

A brief introduction to nonlinear and systematic codes

2.1 Basic notion and notation

We first recall a few definitions. A good introduction to coding theory can be found in [PBH98].
Let $\mathbb{F}_q$ be the finite field with q elements, where q is any power of any prime.
Let $n \geq k \geq 1$ be integers. Let $C \subseteq \mathbb{F}_q^n, C \neq \emptyset$. We say that C is an $(n, |C|)_q$-**code**. Any $c \in C$ is a **word**. Note that here and afterwards a "code" denotes what is called a "non-linear code" in literature.

Definition 2.1.1. *Let $\phi : (\mathbb{F}_q)^k \to (\mathbb{F}_q)^n$ be an injective function and let $C = \mathrm{Im}(\phi)$. We say that C is an $(n, q^k)_q$-**systematic code** if $\phi(v)_i = v_i$ for any $v \in (\mathbb{F}_q)^k$ and any $1 \leq i \leq k$.*

If C is a vector subspace of $(\mathbb{F}_q)^n$, then C is a **linear** code. Clearly any non-zero linear code is equivalent to a systematic code.

We denote by $\mathcal{C}(n, k, q)$ the class of all systematic codes, by $\mathcal{C}_0(n, k, q)$ the subset of $\mathcal{C}(n, k, q)$ of codes with the zero-vector as a word. In case $q = 2$, we will often write $\mathcal{C}(n, k)$ instead of $\mathcal{C}(n, k, 2)$ and $\mathcal{C}_0(n, k)$ instead of $\mathcal{C}_0(n, k, 2)$.

Definition 2.1.2. *Let C be an (n, k, q) code. We call C^* a* **punctured** *code of C, the code obtained from C deleting the same coordinate i, $1 \leq i \leq n$, in each word.*

Remark 2.1.3. If C is a systematic code, then a punctured code C^* obtained by deleting a non-systematic component is still systematic. So, that, if $C \in \mathcal{C}(n, k, q)$, then $C^* \in \mathcal{C}(n-1, k, q)$.
Moreover if C is linear then any punctured code C^* of C is linear.

From now on, $\mathbb{F}$ will denote $\mathbb{F}_q$ and q is understood.

From the definition of systematic code it follows that, given a systematic code C, to any vector $a \in \mathbb{F}^k$ we can associate only one vector $b \in \mathbb{F}^{n-k}$ such that $(a, b) \in C$, where (a, b) is the concatenation of a and b. From now on, given a systematic code C, we use F to denote this association, $a \xrightarrow{F} b$. In particular any $c \in C$ can be seen as $c = (a, F(a))$ for (exactly) one $a \in \mathbb{F}^k$.

We denote with $\mathrm{d}(c, c')$ the **(Hamming) distance** of two words $c, c' \in C$, which is the number of different components between c and c'. We denote with d a number such that $1 \leq d \leq n$ to indicate the **minimum (Hamming) distance of a code**, which is

$$d = \min_{c,c' \in C, c \neq c'} \{\mathrm{d}(c, c')\}.$$

Note that a code with only one word has, by convention, distance equal to infinity. The whole $\mathbb{F}^n$ has distance 1, and $d = n$ in a systematic code is possible only if $k = 1$. From now on, n, k are understood.

Definition 2.1.4. *Let l, $m \in \mathbb{N}$ such that $l \leq m$. In $\mathbb{F}^m$, we denote by $B_l^m(x)$ the set of vectors with distance from the word x less than or equal to l, and we call it the* ***ball*** *centered in x of radius l.*
For conciseness, B_l^m denotes the ball centered in the zero vector.

Obviously, B_l^m is the set of vectors of weight less than or equal to l and

$$|B_l^m| \;=\; \sum_{j=0}^{l} \binom{m}{j} (q-1)^j.$$

We also note that any two balls having the same radius over the same field contain the same number of vectors.

Definition 2.1.5. *The number $A_q(n, d)$ denotes the maximum number of words in a code over $\mathbb{F}_q$ of length n and distance d.*

Definition 2.1.6. *Let $C \in \mathcal{C}(n, k, q)$. We denote by $W_i = W_i(C)$ the number of words in C with weight i. Integer set $\{W_0, \ldots, W_n\}$ is called the* **weight distribution** *of C.*

Definition 2.1.7. *Let $C \in \mathcal{C}(n, k, q)$. We denote by $D_i = D_i(C)$ the number of (unordered) word pairs having distance i, i.e.:*

$$D_i = |\{(c_q, c_2) \mid c_1, c_2 \in C, \mathrm{d}(c_1, c_2) = i\}|$$

Integer set $\{D_0, \ldots, D_n\}$ is called the **distance distribution** *of C.*

Definition 2.1.8. *A code C is* **distance-invariant** *if for any $1 \leq i \leq n$ and any $c, c' \in C$, we have*

$$|\{y \in C \mid \mathrm{d}(c, y) = i\}| = |\{y \in C \mid \mathrm{d}(c', y) = i\}|$$

Clearly linear codes are distance-invariant. The distance distribution of distance-invariant codes (containing the zero vector) can be immediately obtained from their weight distribution.

Two important parameters are the *information rate* and the *relative distance* of a code.

Definition 2.1.9. *For a (possibly) nonlinear code over $\mathbb{F}_q$ with M codewords and length n, we call* information rate*, or simply* rate*, of the code, the value* $\log_q M/n$. *If the code has minimum distance d, we call* relative distance *the value d/n.*

If a code is a $(n, q^k, d)_q$-linear code, the information rate is trivially k/n. In the linear case the rate of a code is a measure of the number of information coordinates relative to the total number of coordinates. The higher the rate, the higher the proportion of coordinates in a codeword that actually contain information rather than redundancy.
The relative distance is a measure of the error-correcting capability of the code relative to its length.
Once the relative distance is fixed, a code is considered to be good if its information rate is the highest possible. Actually it is not easy to determine this relation, and coding theorist often can only formulate upper and lower bounds on this values.

The last fundamental and largly studied parameter we want to mention is the *covering radius* of a code.

Definition 2.1.10. *We define the* covering radius $\rho = \rho(C)$ *to be the smallest integer s such that $\mathbb{F}_q^n$ is the union of the spheres of radius s centered at the codewords of C. Equivalently,*

$$\rho(C) = \max_{x \in \mathbb{F}_q^n} \min_{c \in C} \mathrm{d}(x, c)\,.$$

When $s = \lfloor \frac{d-1}{2} \rfloor$ we say that the code is *perfect*. Such codes satisfy the so called Sphere Packing bound (Section 4.2.1).

2.2 Equivalence of codes

Definition 2.2.1. *Two binary codes C_1 and C_2 of length n are said to be* permutation equivalent *if there exists a coordinate permutation π such that $C_2 = \{\pi(c) \mid c \in C_1\}$. They are said to be* equivalent *if there exists a vector $a \in (\mathbb{F}_2)^n$ and a coordinate permutation π such that $C_2 = \{a + \pi(c) \mid c \in C_1\}$.*

Note that two equivalent codes have the same minimum distance.

Two structural properties of binary codes are the rank and the dimension of the kernel.

Definition 2.2.2. *The* rank r *of a binary code* C*, is the dimension of the linear span* $\langle C \rangle$ *of* C*, i.e.*

$$r = \text{rank}(C) = \dim(\langle C \rangle)$$

The kernel $\ker(C)$ *of a binary code* C *is defined as*

$$K = \ker(C) = \{x \in (\mathbb{F}_2)^n \mid x + C = C\}\,.$$

We will often assume $0 \in C$. Note that if C is linear, then $0 \in C$, but if C is nonlinear, then 0 does not need to belong to C. In this case, we can always consider a new binary code $C' = C + c$ for any $c \in C$, which is equivalent to C, such that $0 \in C'$.

Since $0 \in C$, $\ker(C)$ is a binary linear subcode of C.

We denote with d_K the dimension of the kernel of C. In general, C can be written as the union of cosets of K, and K is the largest such linear code for which this is true [BGH83]. Therefore

$$C = \bigcup_{i=0}^{t} (K + c_i)\,,$$

where $c_0 = (0, \ldots, 0), t + 1 = |C|/2^{d_K}$.

Example 2.2.3. Consider the code C

$$\begin{aligned} C = \{&(1,1,0,0),(0,1,1,0)(0,0,1,0),(0,0,1,1),\\ &(0,0,0,0),(1,1,1,0),(1,0,1,0),(1,1,1,1),\} \end{aligned}$$

We have that

$$\begin{aligned} K = \ker(C) &= \{(1,1,0,0),(0,0,0,0)\}\\ d_K = \dim(K) &= 1\,. \end{aligned}$$

The $|C|/2^{d_K} = 4$ cosets of C are

$$\begin{aligned} K + (0,0,0,0) &= \{(0,0,0,0),(1,1,0,0)\}\\ K + (0,0,1,1) &= \{(0,0,1,1),(1,1,1,1)\}\\ K + (1,0,1,0) &= \{(1,0,1,0),(0,1,1,0)\}\\ K + (0,0,1,0) &= \{(0,0,1,0),(1,1,1,0)\} \end{aligned}$$

The parameters r, d_K can be used to distinguish between non-equivalent binary codes, since equivalent ones have the same r, d_K.

A brief introduction to Boolean function

In this chapter we summarize some definitions and known results from [Car10] and [MS77], concerning Boolean functions and the classical techniques to determine their nonlinearity.

We denote by $\mathbb{F}$ the field $\mathbb{F}_2$. The set $\mathbb{F}^n$ is the set of all binary vectors of length n, viewed as an $\mathbb{F}$-vector space.
Let $v \in \mathbb{F}^n$. The *Hamming weight* $\mathrm{w}(v)$ of the vector v is the number of its nonzero coordinates. For any two vectors $v_1, v_2 \in \mathbb{F}^n$, the *Hamming distance* between v_1 and v_2, denoted by $\mathrm{d}(v_1, v_2)$, is the number of coordinates in which the two vectors differ.
A *Boolean function* is a function $f : \mathbb{F}^n \to \mathbb{F}$. The set of all Boolean functions from $\mathbb{F}^n$ to $\mathbb{F}$ will be denoted by $\mathcal{B}_n$.

3.1 Representations of Boolean functions

3.1.1 Evaluation vector

We assume implicitly to have ordered $\mathbb{F}^n$, so that $\mathbb{F}^n = \{\mathsf{p}_1, \ldots, \mathsf{p}_{2^n}\}$.
A Boolean function f can be specified by a *truth table*, which gives the evaluation of f at all p_i's.

Example 3.1.1. A Boolean function in $\mathcal{B}_3$ is specified by the truth table:

x_1	x_2	x_3	$f(X)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Definition 3.1.2. *We consider the evaluation map:*

$$\mathcal{B}_n \longrightarrow \mathbb{F}^{2^n} \qquad f \longmapsto \underline{f} = (f(\mathsf{p}_1), \ldots, f(\mathsf{p}_{2^n}))\,.$$

The vector $\underline{f}$ *is called the* evaluation vector *of* f*.*

Example 3.1.3. Let $f \in \mathcal{B}_3$, $f(x_1, x_2, x_3) = x_1x_2 + x_1x_3 + x_2$. We order the vectors in $\mathbb{F}^3$ as follows:

$$\begin{array}{llll} v_1 = (0,0,0), & v_2 = (0,0,1), & v_3 = (0,1,0), & v_4 = (1,0,0), \\ v_5 = (0,1,1), & v_6 = (1,0,1), & v_7 = (1,1,0), & v_8 = (1,1,1). \end{array}$$

So we have:

$$\underline{f} = (0,0,1,0,0,1,1,1)\,.$$

Once the order on $\mathbb{F}^n$ is chosen, i.e. the p_i's are fixed, it is clear that the evaluation vector of f uniquely identifies f.

3.1.2 Algebraic normal form

A Boolean function $f \in \mathcal{B}_n$ can be expressed in a unique way as a square free polynomial in $\mathbb{F}[X] = \mathbb{F}[x_1, \ldots, x_n]$, i.e.

$$f = \sum_{v \in \mathbb{F}^n} b_v X^v\,,$$

where $X^v = x^{v_1} \cdots x^{v_n}$.
This representation is called the *Algebraic Normal Form* (ANF).

Example 3.1.4. Let $f \in \mathcal{B}_3$ be the function in the previous example. This function is equal to one if and only if $(x_1+1)(x_2+1)x_3 = 1$ or $x_1(x_2+1)x_3 = 1$ or $x_1x_2x_3 = 1$. Then the ANF is:

$$f(X) = (x_1+1)(x_2+1)x_3 + x_1(x_2+1)x_3 + x_1x_2x_3 = x_1x_2x_3 + x_2x_3 + x_3\,.$$

Definition 3.1.5. *The degree of the ANF of a Boolean function* f *is called the* algebraic degree *of* f*, denoted by* $\deg f$*, and it is equal to* $\max\{\mathrm{w}(v) \mid v \in \mathbb{F}^n, b_v \neq 0\}$.

Let $\mathcal{A}_n$ be the set of all affine functions from $\mathbb{F}^n$ to $\mathbb{F}$, i.e. the set of all Boolean functions in $\mathcal{B}_n$ with algebraic degree 0 or 1. If $\alpha \in \mathcal{A}_n$ then its ANF can be written as

$$\alpha(X) = a_0 + \sum_{i=1}^{n} a_i x_i\,.$$

It is interesting to find properties of Boolean functions which are invariant under an affine coordinate change (we say that they are *affine invariants*).

Remark 3.1.6. Let $\mathrm{AGL}(2,n)$ be the general affine group acting over $\mathbb{F}^n$. By changing the coordinates $\mathrm{AGL}(2,n)$ acts on $\mathcal{B}_n$. In other words, any orbit of $\mathrm{AGL}(2,n)$ shares the same affine invariants.

Proposition 3.1.7. *The algebraic degree of $f \in \mathcal{B}_n$ is an affine invariant.*

There exists a simple divide-and-conquer butterfly algorithm ([Car10], p.10) to compute the ANF from the truth-table (or vice-versa) of a Boolean function, which requires $O(n2^n)$ bit sums (with big O constant 1/2), while $O(2^n)$ bits must be stored. This algorithm is known as the *fast Möbius transform.*

3.1.3 Numerical normal form

In [CG99] a useful representation of Boolean functions for characterizing several cryptographic criteria (see also [CG01], [Car02]) is introduced.
Boolean functions can be represented as elements of $\mathbb{K}[X]/\langle X^2 - X\rangle$, where $\langle X^2 - X\rangle$ is the ideal generated by the polynomials $x_1^2 - x_1, \ldots, x_n^2 - x_n$, and $\mathbb{K}$ is $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, or $\mathbb{C}$.

Definition 3.1.8. *Let f be a function on $\mathbb{F}^n$ taking values in a field $\mathbb{K}$. We call the* numerical normal form (NNF) *of f the following expression of f as a polynomial:*

$$f(x_1, \ldots, x_n) = \sum_{u \in \mathbb{F}^n} \lambda_u \Big(\prod_{i=1}^{n} x_i^{u_i}\Big) = \sum_{u \in \mathbb{F}^n} \lambda_u X^u \,,$$

with $\lambda_u \in \mathbb{K}$ and $u = (u_1, \ldots, u_n)$.

It can be proved ([CG99], Proposition 1) that any Boolean function f admits a unique numerical normal form. As for the ANF, it is possible to compute the NNF of a Boolean function from its truth table by mean of an algorithm similar to a fast Fourier transform, thus requiring $O(n2^n)$ additions over $\mathbb{K}$ and storing $O(2^n)$ elements of $\mathbb{K}$.

From now on let $\mathbb{K} = \mathbb{Q}$.
The truth table of f can be recovered from its NNF by the formula

$$f(u) = \sum_{a \preceq u} \lambda_a, \forall u \in \mathbb{F}^n \,,$$

where $a \preceq u \iff \forall i \in \{1, \ldots, n\}\ a_i \leq u_i$. Conversely, as shown in [CG99] (Section 3.1), it is possible to derive an explicit formula for the coefficients of the NNF by means of the truth table of f.

Proposition 3.1.9. *Let f be any integer-valued function on $\mathbb{F}^n$. For every $u \in \mathbb{F}^n$, the coefficient λ_u of the monomial X^u in the NNF of f is:*

$$\lambda_u = (-1)^{\mathrm{w}(u)} \sum_{a \in \mathbb{F}^n | a \preceq u} (-1)^{\mathrm{w}(a)} f(a)\,. \tag{3.1}$$

It is possible to convert a Boolean function from NNF to ANF simply by reducing its coefficients modulo 2.
The inverse process is less trivial. One can either apply Proposition 3.1.9 to the evaluation vector of f, or can apply recursively the fact that

$$a +_{\mathbb{F}} b = a +_{\mathbb{Z}} b +_{\mathbb{Z}} (-2ab)\,, \tag{3.2}$$

and the fact that each variable has to be square-free (we are working over the affine algebra $\mathbb{K}[x_1, \cdots, x_n]/\langle x_1^2 - x_1, \cdots, x_n^2 - x_n\rangle$).

Example 3.1.10. Let

$$f^{(\mathbb{N})}(x_1, x_2, x_3) = 2x_1x_2x_3 - x_1x_2 - x_2x_3 + 1$$

be a Boolean function in NNF. Reducing its coefficients modulo 2 we obtain

$$f^{(\mathbb{F})}(x_1, x_2, x_3) = x_1x_2 + x_2x_3 + 1\,.$$

Notice that the ANF has only three monomials, while the NNF has four monomials. This is due from the fact that passing from ANF to NNF each sum in $\mathbb{F}$ involving two terms is equivalent to a sum in $\mathbb{Z}$ involving three terms, as shown in Equation (3.2). The inverse process can be done recursively, first converting the sum of $x_1x_2 + x_2x_3$ over $\mathbb{F}$ and then the sum over $\mathbb{F}$ of its result with the term 1, precisely (indicating with $\oplus$ the sum in $\mathbb{F}$ and as usual the sum in $\mathbb{Z}$):

$$\begin{aligned}(x_1x_2 \oplus x_2x_3) \oplus 1 &= (x_1x_2 + x_2x_3 - 2(x_1x_2)(x_2x_3)) \oplus 1 = \\ &= (x_1x_2 + x_2x_3 - 2(x_1x_2)(x_2x_3)) + 1 + \\ &- 2(x_1x_2 + x_2x_3 - 2(x_1x_2)(x_2x_3)) = \\ &= x_1x_2 + x_2x_3 - 2x_1x_2^2x_3 + 1 - 2x_1x_2 - 2x_2x_3 + 4x_1x_2^2x_3 = \\ &= 2x_1x_2x_3 - x_1x_2 - x_2x_3 + 1\end{aligned}$$

3.2 Nonlinearity of a Boolean function

Definition 3.2.1. *Let $v \in \mathbb{F}^n$. The* Hamming weight $\mathrm{w}(v)$ *of the vector v is the number of its nonzero coordinates. For any two vectors $v_1, v_2 \in \mathbb{F}^n$, the* Hamming distance, $\mathrm{d}(v_1, v_2)$, *between v_1 and v_2 is the number of coordinates in which the two vectors differ.*

Definition 3.2.2. *Let $f, g \in \mathcal{B}_n$. The distance $\mathrm{d}(f,g)$ between f and g is the number of $v \in \mathbb{F}^n$ such that $f(v) \neq g(v)$.*

The following lemma is obvious:

Lemma 3.2.3. *Let f, g be two Boolean functions. Then*

$$\mathrm{d}(f,g) = \mathrm{d}(\underline{f}, \underline{g}) = \mathrm{w}(\underline{f} + \underline{g})\,.$$

Definition 3.2.4. *Let $f \in \mathcal{B}_n$. The* nonlinearity *of f is the minimum of the distances between f and any affine function*

$$\mathrm{N}(f) = \min_{\alpha \in \mathcal{A}_n} \mathrm{d}(f, \alpha)\,.$$

We denote by $\nu(n)$ the following natural number

$$\nu(n) = \max_{f \in \mathcal{B}_n} \mathrm{N}(f)\,.$$

Proposition 3.2.5. *The non-linearity of $f \in \mathcal{B}_n$ is an affine invariant.*

The maximum nonlinearity for a Boolean function f is bounded by:

$$\max\{\mathrm{N}(f) \mid f \in \mathcal{B}_n\} \leq 2^{n-1} - 2^{\frac{n}{2}-1}\,. \tag{3.3}$$

3.3 Walsh transform of a Boolean function

Definition 3.3.1. *The* Walsh transform *of a Boolean function $f \in \mathcal{B}_n$ is the following function:*

$$\hat{F} : \mathbb{F}^n \longrightarrow \mathbb{Z} \qquad x \longmapsto \sum_{y \in \mathbb{F}^n} (-1)^{x \cdot y + f(y)}\,.$$

where $x \cdot y$ is the scalar product of x and y.

We have the following fact: ([Car10], p.42)

Fact 3.3.2.

$$\mathrm{N}(f) = \min_{v \in \mathbb{F}^n}\{2^{n-1} - \frac{1}{2}\hat{F}(v)\} = 2^{n-1} - \frac{1}{2}\max_{v \in \mathbb{F}^n}\{\hat{F}(v)\}$$

Definition 3.3.3. *The set of integers $\{\hat{F}(v) \mid v \in \mathbb{F}^n\}$ is called the* Walsh spectrum *of the Boolean function f.*

It is possible ([Car10], p.18) to compute the Walsh spectrum of f from its evaluation vector in $O(n2^n)$ integer operations (with big O constant 1), while storing $O(2^n)$ integers, by means of the *fast Walsh transform* (the Walsh transform is the Fourier transform of the sign function of f). Thus the computation of the nonlinearity of a Boolean function f, when this is given either in its ANF or in its evaluation vector, requires $O(n2^n)$ integer operations and a memory of $O(2^n)$.

Faster methods are known in particular cases, for example when the ANF is a sparse polynomial [Çal13a], [Çal13b].

The Walsh transform of a Boolean function f satisfies the following relation

Fact 3.3.4. *Let $f \in \mathcal{B}_n$ and let $\hat{F}$ be the Walsh transform of f. Then*

$$\sum_{x \in \mathbb{F}^n} \hat{F}(x)\hat{F}(x+y) = \begin{cases} 2^{2n} & \text{if } y = 0 \\ 0 & \text{if } y \neq 0 \end{cases}$$

Corollary 3.3.5 (Parseval's equation)**.**

$$\sum_{x \in \mathbb{F}^n} \hat{F}(x) = 2^{2n}\,.$$

Let $\alpha = \sum_{i=1}^{n} a_i x_i$ (so $\alpha \in \mathcal{A}_n$ with $a_0 = 0$). Then

$$\hat{F}(a) = \sum_{x \in \mathbb{F}^n} (-1)^{x \cdot a + f(x)}\,.$$

We observe that $\hat{F}(a)$ is equal to the number of 0's minus the number of 1's in the vector $\underline{f} + \underline{\alpha}$. Then

$$\hat{F}(a) = \mathrm{w}(\underline{f} + \underline{\alpha} + 1) - \mathrm{w}(\underline{f} + \underline{\alpha}) = 2^n - 2\mathrm{d}(f, \alpha)\,.$$

So we have

$$\mathrm{d}(f, \alpha) = 2^{n-1} - \frac{1}{2}\hat{F}(a)\,.$$

In the same way, we obtain that

$$\mathrm{d}(f, \alpha + 1) = 2^{n-1} + \frac{1}{2}\hat{F}(a)\,.$$

3.4 Non-linearity and Walsh transform

Let m be a monomial in $\mathbb{F}[x_1, \ldots, x_n]$ with $\deg(m) = k \geq 2$, then we will see that $\mathrm{N}(m) = \mathrm{w}(m)$ [1]. Let $\alpha \in \mathcal{A}_n$, $\alpha \neq 0, 1$:

$$\mathrm{d}(0, \alpha) \leq \mathrm{d}(0, m) + \mathrm{d}(\alpha, m)\,,$$

[1] It is well-known that $\mathrm{w}(m) = 2^{n-k}$.

then

$$\mathrm{d}(\alpha, m) \geq \mathrm{w}(\alpha) - \mathrm{w}(m) = 2^{n-1} - 2^{n-k}\,.$$

As $\mathrm{w}(m) = 2^{n-k} \leq 2^{n-1} - 2^{n-k}$ for any $k \geq 2$, then we have that $\mathrm{N}(m) = 2^{n-k}$.

From the relation between the distance and the Walsh transform, we have that

$$\mathrm{N}(f) = \min_{a\in\mathbb{F}^n}(2^{n-1} - \frac{1}{2}\hat{F}(a)) = 2^{n-1} - \frac{1}{2}\max_{a\in\mathbb{F}^n}|\hat{F}(a)|\,.$$

From Parseval's equation we can deduce that $\max_{a\in\mathbb{F}^n}|\hat{F}(a)| \geq \sqrt{2^n} = 2^{\frac{n}{2}}$. Then

$$\mathrm{N}(f) \leq \nu(n) \leq 2^{n-1} - 2^{\frac{n}{2}-1}\,. \tag{3.4}$$

This bound, valid for every Boolean function, is called the *universal non-linearity bound*. In this bound the equality occurs if and only if $|\hat{F}(a)| = 2^{\frac{n}{2}}$ for every $a \in \mathbb{F}^n$. The corresponding functions are called *bent functions*. They can exist only for even values [2] of n, because $2^{n-1} - 2^{\frac{n}{2}-1}$ must be an integer. For n odd, inequality $\nu(n) \leq 2^{n-1} - 2^{\frac{n}{2}-1}$ cannot be tight. If $n = 2d+1$, then

$$\nu(n) = \nu(2d+1) \leq \lfloor 2^{2d} - 2^{\frac{2d+1}{2}-1}\rfloor = \lfloor 2^{2d} - \sqrt{2}\,2^{d-1}\rfloor\,.$$

As regards lower bounds, it is well-known that for any $n = 2d+1$ there exist some quadratic functions with non-linearity $2^{2d} - 2^d$. So we have the following result

Theorem 3.4.1. *Let $n = 2d+1$ be an odd integer. Then*

$$2^{2d} - 2^d \leq \nu(n) \leq \lfloor 2^{2d} - \sqrt{2}\,2^{d-1}\rfloor\,.$$

For $n = 2d+1$, it has been shown that

$$\nu(n) = 2^{2d} - 2^d \text{ for } n = 1, 3, 5, 7\,,$$

and

$$\nu(n) > 2^{2d} - 2^d \text{ for } n \geq 15\,.$$

Moreover, $\nu(15) \geq 16276$. While for $n = 9, 11, 13$ nothing is known, apart from Theorem 3.4.1. We summarize the situation for n odd in the following table:

[2] actually, we will see in the following section, that they exist for every n even.

n	Maximum non-linearity
3	$\nu(3) = 2$
5	$\nu(5) = 12$
7	$\nu(7) = 56$
9	$240 \leq \nu(9) \leq 244$
11	$992 \leq \nu(11) \leq 1001$
13	$4032 \leq \nu(13) \leq 4050$
15	$16276 \leq \nu(15) \leq 16293$
17	$65280 < \nu(17) \leq 65354$
$\vdots$	$\vdots$

Table 3.1: Maximum non-linearity for n odd

3.5 Bent functions

In this section n is an even integer.

Definition 3.5.1. *A Boolean function $f \in \mathcal{B}_n$ is called* bent *if $|\hat{F}(a)| = 2^{\frac{n}{2}}$, for every $a \in \mathbb{F}^n$.*

Example 3.5.2. Let $f \in \mathcal{B}_4$, $f(x_1, x_2, x_3, x_4) = x_1x_2 + x_3x_4$. If we compute the Walsh transform of f for any $a \in \mathbb{F}^4$, we see that $\hat{F}(a) = \pm 4$, so f is bent.

a	$\hat{F}(a)$	a	$\hat{F}(a)$	a	$\hat{F}(a)$	a	$\hat{F}(a)$
$(0,0,0,0)$	4	$(1,0,0,0)$	4	$(0,1,1,0)$	4	$(1,0,1,1)$	-4
$(0,0,0,1)$	4	$(0,0,1,1)$	-4	$(1,0,1,0)$	4	$(1,1,0,1)$	-4
$(0,0,1,0)$	4	$(0,1,0,1)$	4	$(1,1,0,0)$	-4	$(1,1,1,0)$	-4
$(0,1,0,0)$	4	$(1,0,0,1)$	4	$(0,1,1,1)$	-4	$(1,1,1,1)$	4

A bent function $f \in \mathcal{B}_n$ is further away from any affine function $\alpha \in \mathcal{A}_n$. More precisely, we have the following proposition:

Proposition 3.5.3. *A Boolean function $f \in \mathcal{B}_n$ is bent if and only if its distance between any affine function is equal to $2^{n-1} \pm 2^{\frac{n}{2}-1}$.*

We have a bound on the algebraic degree of a bent function:

Proposition 3.5.4. *If $f \in \mathcal{B}_n$ is a bent function and $n > 2$ then* $\deg f \leq \frac{n}{2}$.

For any even n and an even integer $m < n$ we can construct bent function as the sum of a bent function in m variables and a bent function in $n - m$ variables as follows

Proposition 3.5.5. *Let $f \in \mathcal{B}_n$ such that $f(x_1, \ldots, x_n) = g(x_1, \ldots, x_m) + h(x_{m+1}, \ldots, x_n)$, with $g \in \mathcal{B}_m$ and $h \in \mathcal{B}_{n-m}$. Then f is a bent function if and only if g and h are bent functions.*

Corollary 3.5.6. *For any even $n \geq 2$ the Boolean function $f = x_1x_2 + x_3x_4 + \cdots + x_{n-1}x_n$ is bent.*

Since $\mathrm{N}(f) = 2^{n-1} - 2^{\frac{n}{2}-1}$ for any bent function f (see equation 3.4), and at least one exists for every n even, then we have

Theorem 3.5.7. *For n even, $\nu(n) = 2^{n-1} - 2^{\frac{n}{2}-1}$.*

n	Maximum non-linearity
2	$\nu(2) = 1$
4	$\nu(4) = 6$
6	$\nu(6) = 28$
8	$\nu(8) = 120$
10	$\nu(10) = 496$
12	$\nu(12) = 2016$
14	$\nu(14) = 8128$
16	$\nu(16) = 32640$
$\vdots$	$\vdots$

Table 3.2: Maximum non-linearity for n even

Part II

On code bounds

Introduction

The problem of bounding the size of a code depends heavily on the code family that we are considering. In this part we are interested in three types of codes: linear codes, systematic codes and non-linear codes. Referring to the subsequent section for rigorous definitions, with **linear codes** we mean linear subspaces of $(\mathbb{F}_q)^n$, while with **non-linear codes** we mean (following consolidated tradition) codes that are not necessarily linear. In this sense, a linear code is always a non-linear code, while a non-linear code may be a linear code, although it is unlikely. Systematic codes form a less-studied family of codes, whose definition is given in the next section. Modulo code equivalence all (non-zero) linear codes are systematic and all systematic codes are non-linear. In some sense, systematic codes stand in the middle between linear codes and non-linear codes. The size of a systematic code is directly comparable with that of a linear code, since it is a power of the field size.
In this part we propose some **theoretical bounds**, that is, bounds on the size of a code that can be obtained by a closed-formula expression. Algorithmic bounds exist, and actually one of these (the Linear Programming bound [Del73]) is considered in general the most powerful known bound.
Any upper bound for non-linear codes is also an upper bound for both systematic codes and linear codes, while an upper bound for systematic codes is also an upper bound for linear codes. Given the constraint on the size of systematic codes, when we consider an upper bound on the size of non-linear codes, we will consider the largest power of q which is less than or equal to the upper bound.
The algebraic structure of linear codes would suggest the knowledge of a high number of bounds strictly for linear codes, and only a few bounds for the other case. Rather surprisingly, the literature reports only one bound for linear codes, the Griesmer bound ([Gri60]), no bounds for systematic codes and many bounds for non-linear codes. Among those, we recall some theoretical bounds: the Johnson bound ([Joh62],[Joh71],[HP03]), the Elias-Bassalygo bound ([Bas65],[HP03]), the Levenshtein bound ([Lev98]), the Hamming (Sphere Packing) bound and the Singleton bound ([PBH98]), and the Plotkin bound ([Plo60], [HP03]).
Since the Griesmer bound is specialized for linear codes, we would expect it to beat the other bounds, but even this does not happen, except in some cases. So we have an

unexpected situation where the bounds holding for the more general case are numerous and beat bounds holding for the specialized case. Actually, as far as it concern the Griesmer bound, it seems to hold also in the more general systematic case. We investigate this fact in Chapter 5, and we prove the bound holds also for an infinite family of systematic codes.
Chapter 4 is an overview of known bounds, with a special focus on upper bounds.
In Chapter 6 we present an original upper bound which holds for all codes containing a systematic code, and we compare it with other well known upper bounds.

Overview of known classical bounds

4.1 Preliminaries

For ease of reading, we first recall a few definitions.
Let $\mathbb{F}_q$ be the finite field with q elements, where q is any positive power of any prime. Let $n \geq k \geq 1$ be integers. Let $C \subseteq (\mathbb{F}_q)^n$, $C \neq \emptyset$. We say that C is an $(n, |C|)_q$-**code**. Any $c \in C$ is a **codeword**. Note that here and afterwards a "code" denotes what is called a "non-linear code" in the introduction.
Let $\phi : (\mathbb{F}_q)^k \to (\mathbb{F}_q)^n$ be an injective function and let $C = \mathrm{Im}(\phi)$. We say that C is an $(n, q^k)_q$-**systematic code** of **dimension** k if $(\phi(v))_i = v_i$ for any $v \in (\mathbb{F}_q)^k$ and any[1] component $1 \leq i \leq k$. If C is a vector subspace of $(\mathbb{F}_q)^n$, then C is a **linear code**. Clearly any non-zero linear code is equivalent to a systematic code.
From now on, $\mathbb{F}$ will denote $\mathbb{F}_q$ and q is fixed.
We denote with $\mathrm{d}(c, c')$ the **(Hamming) distance** of two words $c, c' \in C$, which is the number of different components between c and c'. We denote with d a number such that $1 \leq d \leq n$ to indicate the **minimum distance** of a code, which is $d = \min_{c,c' \in C, c \neq c'}\{\mathrm{d}(c, c')\}$. If C is an $(n, M)_q$-code with distance d then we can write that C is an $(n, M, d)_q$-code. We will omit q when clear from the context. Note that a code with only one codeword has, by convention, minimum distance equal to infinity. The whole $\mathbb{F}^n$ has minimum distance 1, and $d = n$ in a systematic code is possible only if $k = 1$.
From now on, n, k are fixed.

Definition 4.1.1. *Let l, $m \in \mathbb{N}$ be such that $l \leq m$. In $\mathbb{F}^m$, we denote by $B_l^m(x)$ the set of vectors with distance from the word x less than or equal to l, and we call it the* ***ball*** *centered in x of radius l.*
For conciseness, B_l^m denotes the ball centered in the zero vector.

Obviously, B_l^m is the set of vectors of weight less than or equal to l and

$$|B_l^m| = \sum_{j=0}^{l} \binom{m}{j} (q-1)^j.$$

[1] Subscript i indicates the i-th component of a vector.

We also note that any two balls having the same radius over the same field contain the same number of vectors.

Definition 4.1.2. *The number $A_q(n,d)$ denotes the maximum number of codewords in a code over $\mathbb{F}_q$ of length n and minimum distance d.*

We now recall some results regarding the quantity $A_q(n,d)$, whose proofs can be found in [HP03] or in [Rom92].

Theorem 4.1.3. *It holds that*

- *If $d > 1$ then $A_q(n,d) \leq A_q(n-1,d-1)$.*
- *If $n \geq 1$ then $A_q(n,1) = q^n$ and $A_q(n,n) = q$.*
- *If $q = 2$ and d is even, then $A_2(n,d) = A_2(n-1,d-1)$.*

The last property holds since if d is odd, then C is an (n,M,d) code if and only if the code C obtained by adding a *parity check bit* (i.e. the sum of all the bits of a codeword) to each codeword in C is an $(n+1,M,d+1)$ code. So in order to understand the behavior of $A_2(n,d)$ it is sufficient to understand its behavior for d even.

Theorem 4.1.4. *Let C be a code with distance d and length n on $\mathbb{F}_q$. Then:*

$$A_q(n,d) \leq qA_q(n-1,d)$$

Proof. Let C be an (n,M,d) code on $\mathbb{F}_q$ and let $M = A_q(n,d)$. Given $v \in \mathbb{F}_q$ we denote by C_v the subset of C of elements with v in the n-th position. For some v, the set C_v is an $\mathbb{F}_q$ code with at least M/q codewords. Erasing from the words in C_v the n-th component, a code B of length $n-1$ and distance d is obtained. So that

$$\frac{M}{q} \leq |C_v| \leq A_q(n-1,d)$$

which implies

$$A_q(n,d) \leq qAq(n-1,d)$$

□

4.2 Upper bounds

4.2.1 The Hamming upper bound

From the fact that the spheres of radius $t = \left\lfloor \frac{d-1}{2} \right\rfloor$ are pairwise disjoint, the *sphere packing bound* (or *Hamming bound)* immediately follows:

Theorem 4.2.1 (Hamming bound)**.**

$$A_q(n,d) \leq \frac{q^n}{|B_t^n|}$$

Proof. Let M be the total number of codewords in a code C. The union of the balls of radius t around all codewords is contained in $(\mathbb{F}_q)^n$. Then, since each ball is non-intersecting, summing the number of elements in each, we obtain:

$$M|B_t^n| \leq q^n$$

Since last formula holds for any code, we have:

$$A_q(n,d) \leq \frac{q^n}{|B_t^n|}$$

□

Codes that meet the Sphere Packing bound are called *perfect*.

4.2.2 The Plotkin upper bound

First, we provide a binary version of the Plotkin bound and its general version in the case of any alphabet.

Theorem 4.2.2 (Plotkin bound - binary case [Plo60])**.** *We have two different cases:*

1. *If d is even and $2d > n$ then*

$$A_2(n,q) \leq 2\left\lfloor \frac{d}{2d-n} \right\rfloor$$

2. *If d is odd and $2d+1 > n$, then*

$$A_2(n,d) \leq 2\left\lfloor \frac{d+1}{2d+1-n} \right\rfloor$$

We report the proof of the first inequality.

Proof. Let C be a generic binary (M,n) code (in particular the theorem will hold when $M = A_2(n,q)$, where M is the number of codewords and n the length of the code. The bound is proved by bounding the quantity $\sum_{(x,y)\in C^2, x\neq y} \mathrm{d}(x,y)$ in two different ways.

On the one hand, there are M choices for x and for each such choice, there are $M-1$ choices for y. Since by definition $\mathrm{d}(x,y) \geq d$ for all x and y ($x \neq y$), it follows that

$$\sum_{(x,y)\in C^2, x\neq y} \mathrm{d}(x,y) \geq M(M-1)d.$$

On the other hand, let A be an $M \times n$ matrix whose rows are the elements of C. Let s_i be the number of zeros contained in the i'th column of A. This means that the i'th column contains $M - s_i$ ones. Each choice of a zero and a one in the same column contributes exactly 2 (because $\mathrm{d}(x,y) = \mathrm{d}(y,x)$) to the sum $\sum_{x,y\in C} \mathrm{d}(x,y)$ and therefore

$$\sum_{x,y\in C} \mathrm{d}(x,y) = \sum_{i=1}^{n} 2s_i(M-s_i).$$

If M is even, then the quantity on the right is maximized if and only if $s_i = M/2$ holds for all i, then

$$\sum_{x,y\in C} \mathrm{d}(x,y) \leq \frac{1}{2}nM^2.$$

Combining the upper and lower bounds for $\sum_{x,y\in C} \mathrm{d}(x,y)$ that we have just derived,

$$M(M-1)d \leq \frac{1}{2}nM^2$$

which given that $2d > n$ is equivalent to

$$M \leq \frac{2d}{2d-n}.$$

Since M is even, it follows that

$$M \leq 2\left\lfloor \frac{d}{2d-n} \right\rfloor.$$

On the other hand, if M is odd, then $\sum_{i=1}^{n} 2s_i(M-s_i)$ is maximized when $s_i = \frac{M\pm 1}{2}$ which implies that

$$\sum_{x,y\in C} \mathrm{d}(x,y) \leq \frac{1}{2}n(M^2-1).$$

Combining the upper and lower bounds for $\sum_{x,y\in C} \mathrm{d}(x,y)$, this means that

$$M(M-1)d \leq \frac{1}{2}n(M^2-1)$$

or, using that $2d > n$,

$$M \leq \frac{2d}{2d-n} - 1.$$

Since M is an integer,

$$M \leq \left\lfloor \frac{2d}{2d-n} - 1 \right\rfloor = \left\lfloor \frac{2d}{2d-n} \right\rfloor - 1 \leq 2\left\lfloor \frac{d}{2d-n} \right\rfloor.$$

This completes the proof of the bound. □

Plotkin bound is a very strong bound and it is experimentally known to be very tight in the tiny range where it can be applied. In fact, in the binary case Levenshtein proved that if Hadamard's conjecture is true then Plotkin's bound is sharp. A discussion and analysis of this facts can be found in [Rom92].
Though Hadamard's conjecture is probably true, its resolution remains a difficult open question. Let us indicate with $P(n,d)$ the Plotkin bound in the binary case. In [dLG01] de Launey and Gordon consider the ratio $R(n,d) = A_2(n,d)/P(n,d)$. They present an efficient heuristic for constructing for any $d \geq n/2$, a binary code which has at least $0.495P(n,d)$ codewords. Their result is confirmed by a computer calculation, which shows that $R(n,d) > 0.495$ for d up to one trillion.
Plotkin bound says that a good binary code (meeting this bound) must have about the same number of ones and zeros on each column of the $M \times n$ matrix of all codewords.
Plotkin bound has been generalized to any alphabet by Blake and Mullin (p. 84 of [BM76]):

Theorem 4.2.3 (Plotkin bound - q-ary case)**.** *Let q be an integer and $dq > n(q-1)$, then:*

$$A_q(n,d) \leq \frac{dq}{dq - n(q-1)}$$

Elias gave another refinement of the bound which is stated in Section 4.2.5.
Using theorem 4.1.4 and the Plotkin bound, we can derive the following properties:

1. if d is even, then $A_2(2d,d) \leq 4d$,
   since $A_2(2d,d) \leq_{Thm} 2A_2(2d-1,d) \leq_{Plotkin} 2 \cdot 2 \left\lfloor \frac{d}{2d-2d+1} \right\rfloor = 4d$;
2. if d is even, then $A_2(2d+1,d) \leq 8d$;
3. if d is odd, then $A_2(2d,d) \leq 2d+2$;
4. if d is odd, then $A_2(2d+1,d) \leq 4d+4$.

We leave to the reader the proof of (2), (3) and (4).

4.2.3 The Johnson upper bounds

A nonlinear (n,M,d) code C over $\mathbb{F}_q$ is a *constant weight code* provided every codeword has the same weight w.
Define $A_q(n,d,w)$ to be the maximum number of codewords in a constant weight (n,M) code over $\mathbb{F}_q$ of length n and minimum distance at least d whose codewords have weight w. Obviously $A_q(n,d,w) \leq A_q(n,d)$.

Theorem 4.2.4 (Restricted Johnson bound for $A_q(n,d,w)$)**.**

$$A_q(n,d,w) \leq \left\lfloor \frac{nd(q-1)}{qw^2 - 2(q-1)nw + nd(q-1)} \right\rfloor$$

provided $qw^2 - 2(q-1)nw + nd(q-1) > 0$

Removing the restriction of $qw^2 - 2(q-1)nw + nd(q-1) > 0$ Johnson obtained:

Theorem 4.2.5 (Unrestricted Johnson bound for $A_q(n,d,w)$)**.** *The following cases hold:*

1. *If* $2w < d$*, then* $A_q(n,d,w) = 1$.
2. *if* $2w \geq d$ *and* $d \in \{2e-1, 2e\}$*, then, setting* $q^* = q-1$*,*

$$A_q(n,d,w) \leq \left\lfloor \frac{nq^*}{w} \left\lfloor \frac{(n-1)q^*}{w-1} \left\lfloor \cdots \left\lfloor \frac{(n-w+e)q^*}{e} \right\rfloor \cdots \right\rfloor \right\rfloor \right\rfloor$$

3. *If* $w < e$*, then* $A_2(n, 2e-1, w) = A_2(n, 2e, w) = 1$.
4. *if* $w \geq e$ *then,*

$$A_2(n,2e-1,w) = A_2(n,2e-1,w) \leq \left\lfloor \frac{n}{w} \left\lfloor \frac{n-1}{w-1} \left\lfloor \cdots \left\lfloor \frac{n-w+e}{e} \right\rfloor \cdots \right\rfloor \right\rfloor \right\rfloor$$

The bounds on $A_q(n,d,w)$ can be used to give upper bounds on $A_q(n,d)$ also due to Johnson ([Joh62],[Joh71]). These bounds strengthen the Sphere Packing bound. The idea of the proof is to count not only the vectors in $\mathbb{F}_q$ that are within distance $t = (d-1)/2$ of all codewords (that is, the disjoint spheres of radius t centered at codewords) but also the vectors at distance $t+1$ from codewords that are not within these spheres.

Theorem 4.2.6 (Johnson bound for $A_q(n,d)$)**.** *Let* $t = \lfloor (d-1)/2 \rfloor$

1. *If* d *is odd, then*

$$A_q(n,d) \leq \frac{q^n}{\sum_{i=0}^{t} \binom{n}{i}(q-1)^i + \frac{\binom{n}{t+1}(q-1)^{t+1} - \binom{d}{t}A_q(n,d,d)}{A_q(n,d,t+1)}}$$

2. *If* d *is even, then*

$$A_q(n,d) \leq \frac{q^n}{\sum_{i=0}^{t} \binom{n}{i}(q-1)^i + \frac{\binom{n}{t+1}(q-1)^{t+1}}{A_q(n,d,t+1)}}$$

3. *If d is odd, then*

$$A_2(n,d) \leq \frac{2^n}{\sum_{i=0}^{t}\binom{n}{i} + \frac{\binom{n}{t+1}-\binom{d}{t}A_2(n,d,d)}{\left\lfloor\frac{n}{t+1}\right\rfloor}}$$

4. *If d is even, then*

$$A_2(n,d) \leq \frac{2^n}{\sum_{i=0}^{t}\binom{n}{i} + \frac{\binom{n}{t+1}}{\left\lfloor\frac{n}{t+1}\right\rfloor}}$$

5. *If d is odd, then*

$$A_2(n,d) \leq \frac{2^n}{\sum_{i=0}^{t}\binom{n}{i} + \frac{\binom{n}{t}(\frac{n-t}{t+1}-\left\lfloor\frac{n-t}{t+1}\right\rfloor)}{\left\lfloor\frac{n}{t+1}\right\rfloor}} \tag{4.1}$$

Bound 4.1 strengthens the Sphere Packing bound and the two bounds in fact agree precisely when $(t+1)|(n-t)$. Recall that codes that meet the Sphere Packing bound are called perfect. An $(n, M, 2t+1)$ binary code with $M = A_2(n, 2t+1)$ which attains the Johnson bound 4.1 is called *nearly perfect*. The classification of such codes is known. One example of nearly perfect code is the $[256, 16, 6]$-*Nordstrom-Robinson code*, of which we provide a Magma construction in Section 9.1.

4.2.4 The Singleton upper bound and MDS codes

The next bound is a rather weak bound in general.

Theorem 4.2.7 (Singleton bound)**.**

$$A_q(n,d) \leq q^{n-d+1}$$

Proof. First observe that there are q^n many q-ary words of length n, since each letter in such a word may take one of q different values, independently of the remaining letters.
Now let C be an arbitrary q-ary block code of minimum distance d. Clearly, all codewords $c \in C$ are distinct. If we delete the first $d-1$ letters of each codeword, then all resulting codewords must still be pairwise different, since all original codewords in C have Hamming distance at least d from each other. Thus the size of the code remains unchanged.
The newly obtained codewords each have length

$$n - (d-1) = n - d + 1$$

and thus there can be at most

$$q^{n-d+1}$$

of them. Hence the original code C shares the same bound on its size $|C|$:

$$|C| \leq A_q(n,d) \leq q^{n-d+1}.$$

□

Codes reaching the Singleton bound are called *Maximum Distance Separable (MDS)* codes. This class of codes contains the very important family of codes known as *Reed-Solomon* codes.

4.2.5 The Elias upper bound

Extending the ideas of Plotkin, in 1960 Elias discovered a new bound without publishing it. The same bound was published by Bassylago in 1965 [Bas65].
Despite this bound is rather weak, its importance lies in the fact that the asymptotic form of this bound is superior to many classical bounds.

Theorem 4.2.8 (Elias bound)**.** *Let $r = 1 - q^{-1}$. Suppose that $w \leq rn$ and $w^2 - 2rnw + rnd > 0$. Then*

$$A_q(n,d) \leq \frac{rnd}{w^2 - 2rnw + rnd} \cdot \frac{q^n}{|B_w^n|} \tag{4.2}$$

4.2.6 The Linear Programming upper bound

This bound, discovered by Delsarte in 1975 [Del73], is in general the most powerful of the classical bounds, but it requires the use of linear programming. Before stating the bound we need to introduce a couple of new definitions.

Definition 4.2.9. *The* (Hamming) distance distribution *of a code C of length n is the list $B_i = B_i(C)$ for $0 \leq i \leq n$, where*

$$B_i(C) = \sum_{c \in C} |\{v \in C \mid \mathrm{d}(v,c) = i\}|.$$

Definition 4.2.10. *The* Krawtchouck polynomial $K_k^{n,q}(x)$ *of degree k is defined by*

$$K_k^{n,q}(x) = \sum_{j=0}^{k} (-1)^j (q-1)^{k-j} \binom{x}{j} \binom{n-x}{k-j}$$

for $0 \leq k \leq n$.

Theorem 4.2.11 (Linear Programming bound)**.** *The following hold:*

1. *When $q \geq 2$, $A_q(n,d) \leq \max\{\sum_{w=0}^{n} B_w\}$, where the maximum is taken over all B_w subject to the following conditions:*

   (a) *$B_0 = 1$ and $B_w = 0$ for $1 \leq w \leq d-1$,*

   (b) *$B_w \geq 0$ for $d \leq w \leq n$, and*

   (c) *$\sum_{w=0}^{n} B_w K_k^{n,q}(w) \geq 0$ for $1 \leq k \leq n$.*

2. *When d is even and $q = 2$, $A_q(n,d) \leq \max\{\sum_{w=0}^{n} B_w\}$, where the maximum is taken over all B_w subject to the following conditions:*

   (a) *$B_0 = 1$ and $B_w = 0$ for $1 \leq w \leq d-1$ and all odd w,*

   (b) *$B_w \geq 0$ for $d \leq w \leq n$, and $B_n \leq 1$, and*

   (c) *$\sum_{w=0}^{n} B_w K_k^{n,2}(w) \geq 0$ for $1 \leq k \leq \lfloor n/2 \rfloor$.*

Solving the inequalities of this theorem is accomplished by linear programming, hence the name. At times other inequalities can be added to the list which add more constraints to the linear program and reduce the size of $\sum_{w=0}^{n} B_w$. In specific cases other variations to the Linear Programming bound can be performed to achieve a smaller upper bound.

4.2.7 The Levenshtein upper bound

In 1978, Levenshtein proved a bound in the setting of systems of orthogonal polynomials. The article was written in Russian. The first English version of this bound, stated using the language of coding theory, can be found in [Lev95], published in 1995. In [Lev98] the whole theory regarding this bound is exposed in more than one hundred pages.
Here we only provide the basic definitions to state the bound.

Definition 4.2.12. *Let $d_k(n,q) = d_k(n)$ be the smallest root of the equation $K_k^{n,q}(z) = 0$.*
Define the function

$$L^{n,q}(z) = \begin{cases} L_k^{n,q}(z) & \text{if} \quad d_k(n-1)+1 < z \leq d_{k-1}(n-2)+1 \\ qL_k^{n-1,q}(z) & \text{if} \quad d_k(n-2)+1 < z \leq d_k(n-1)+1 \end{cases} \tag{4.3}$$

where

$$L_k^{n,q}(z) = |B_{k-1}^n| - \binom{n}{k}(q-1)^k \frac{K_{k-1}^{n-1,q}(z-1)}{K_k^{n,q}(z)} \tag{4.4}$$

Theorem 4.2.13 (Levenshtein bound)**.**

$$A_q(n,d) \leq L^{n,q}(d) \tag{4.5}$$

Levenshtein bound is one of the strongest bound, especially for small values of q. An intuition of its behavior can be grasp from Table 6.1 and 6.2.
We notice that the computation of this bound requires the computation of the roots of a Krawtchouck polynomial, which can be very long compared to other closed formula upper bounds.

4.2.8 The Zinoviev-Litsyn-Laihonen upper bound

In 1984, Zinoviev and Litsyn [ZL84] prove a bound for non-linear codes, in a Russian written article.
In 1998 Litsyn and Laihonen prove the same bound, Theorem 1 of [LL98], and apply it to show some results on asymptotic bounds.
In the work of 1984, the authors obtain some new bounds for the dual distance of generalized concatenated codes and BCH codes. The use of these bounds in the existing code-shortening arrangements lead to a number of codes with optimal parameters. A construction was proposed for shortening arbitrary (linear and nonlinear) codes. Application of this construction to existing codes yields a large number of codes with optimal known parameters.
In the paper of 1998, they consider upper bounds on minimum distance and covering radius of a code, generalizing techniques from [LT96] and combining them with the mentioned upper bound on the asymptotic information rate of non-binary codes. The upper bound on the information rate is an application of a shortening method of a code. These results were aimed to improve on the best currently known asymptotic upper bounds on minimum distance and covering radius of non-binary codes in certain intervals.
We write the bound with our notation as follows.

Theorem 4.2.14 (Zinoviev-Litsyn-Laihonen (ZLL) bound)**.** *Let* $1 \leq d \leq n$*. Let* $t \in \mathbb{N}$ *be such that* $t \leq n-d$*. Let* $r \in \mathbb{N}$ *be such that* $0 \leq r \leq t$ *and* $0 \leq r \leq \frac{1}{2}d$*. Then*

$$A_q(n,d) \leq \frac{q^t}{|B_r^t|} A_q(n-t, d-2r).$$

Note that $t \leq n-d$ implies $d-2r \leq n-t$ so that the value $A_q(n-t,d-2r)$ is meaningful.
We present an improvement of this bound in Section 6.

4.2.9 The Griesmer upper bound for linear codes

Thanks to their strong algebraic structure, linear codes enjoy more specific bounds. We show here an important bound due to Griesmer [Gri60], which generalizes the Singleton bound.

Theorem 4.2.15 (Griesmer bound)**.** *Let n be the smallest integer such that there exists an (n, q^k) binary linear code with minimum distance at least d. Then*

$$n \geq \sum_{i=0}^{k-1} \left\lceil \frac{d}{2^i} \right\rceil .$$

Proof. Let $N(k, d)$ denote the minimum length of a binary code of dimension k and distance d. Let C be such a code. We want to show that

$$N(k, d) \geq \sum_{i=0}^{k-1} \left\lceil \frac{d}{2^i} \right\rceil$$

Let G be a generator matrix of C. We can always suppose that the first row of G is of the form $r = (1, ..., 1, 0, ..., 0)$ with weight d:

$$G = \begin{bmatrix} 1 & ... & 1 & 0 & ... & 0 \\ * & * & * & & G' & \end{bmatrix}.$$

The matrix G' generates a code C', which is called the residual code of C. C' has obviously dimension $k' = k - 1$ and length $n' = N(k, d) - d$. C' has a distance d', but we don't know it.
Let $u \in C'$ s.t. $w(u) = d'$. There exists a vector $v \in (\mathbb{F}_2)^d$ s.t. the concatenation $(v|u) \in C$. Then

$$w(v) + w(u) = w(v|u) \geq d.$$

On the other hand, also $(v|u) + r \in C$, since $r \in C$ and C is linear, so

$$w((v|u) + r) \geq d.$$

But $w((v|u) + r) = w(((1, 1, ..., 1) + v)|u) = d - w(v) + w(u)$, so this becomes

$$d - w(v) + w(u) \geq d.$$

By summing this with $w(v) + w(u) \geq d$, we obtain

$$d + 2w(u) \geq 2d.$$

But $w(u) = d'$, so we get

$$d' \geq d/2.$$

This implies $n' \geq N(k-1, d/2)$, therefore $n' \geq \lceil N(k-1, d/2) \rceil$ (due to the integrality of n'), so that

$$N(k,d) \geq \lceil N(k-1, d/2) \rceil + d.$$

By induction over k we will eventually get

$$N(k,d) \geq \sum_{i=0}^{k-1} \left\lceil \frac{d}{2^i} \right\rceil$$

(note that at any step the dimension decreases by 1 and the distance is halved, and we use the identity $\left\lceil \frac{\lceil a/2^{k-1} \rceil}{2} \right\rceil = \lceil \frac{a}{2^k} \rceil$ for any integer a and positive integer k). □

Since $\lceil d/q^0 \rceil = d$ and $\lceil d/q^i \rceil \geq 1$ for $i = 1, \ldots, k-1$, we have that $n \geq \sum_{i=0}^{k-1} \lceil \frac{d}{2^i} \rceil \geq d + \sum_{i=1}^{k-1} 1 = d + k - 1$, which is the Singleton bound.
Griesmer also showed that for certain values of k and d the equality holds. In 1965 Solomon and Stiffler [SS65] simplified Griesmer's proof and at the same time generalized it to linear codes over an arbitrary finite field $\mathbb{F}_q$, where it takes the form

$$n \geq \sum_{i=0}^{k-1} \left\lceil \frac{d}{q^i} \right\rceil .$$

More important, however, Solomon and Stiffler introduced the notion of puncturing a $(q^k - 1, k)$ maximal-length shift-register code and showed that for many more values of k and d equality holds. It can be also shown that the Griesmer bound is implied by the Plotkin bound in case 2^{k-1} divides d.
There exists nonlinear and systematic codes with q^k codewords overpassing the Griesmer bound. Though the Griesmer bound holds for some families of systematic codes and we investigate this fact in Section 5.

Example 4.2.16. The following is a $(19, 2^4, 10)_2$-nonlinear code.
Furthermore $\sum_{i=0}^{3} \lceil \frac{10}{2^i} \rceil = 20$.

$C = \{(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0), (1,1,0,0,1,1,1,1,0,1,0,1,0,0,0,0,1,1,0),$
$(1,0,0,1,1,1,1,0,1,0,1,0,0,0,0,1,1,0,1), (0,0,1,1,1,1,0,1,0,1,0,0,0,0,1,1,0,1,1),$
$(0,1,1,1,1,0,1,0,1,0,0,0,0,1,1,0,1,1,0), (1,1,1,1,0,1,0,1,0,0,0,0,1,1,0,1,1,0,0),$
$(1,1,1,0,1,0,1,0,0,0,0,1,1,0,1,1,0,0,1), (1,1,0,1,0,1,0,0,0,0,1,1,0,1,1,0,0,1,1),$
$(1,0,1,0,1,0,0,0,0,1,1,0,1,1,0,0,1,1,1), (0,1,0,1,0,0,0,0,1,1,0,1,1,0,0,1,1,1,1),$
$(1,0,1,0,0,0,0,1,1,0,1,1,0,0,1,1,1,1,0), (0,1,0,0,0,0,1,1,0,1,1,0,0,1,1,1,1,0,1),$
$(1,0,0,0,0,1,1,0,1,1,0,0,1,1,1,1,0,1,0), (0,0,0,0,1,1,0,1,1,0,0,1,1,1,1,0,1,0,1),$
$(0,0,0,1,1,0,1,1,0,0,1,1,1,1,0,1,0,1,0), (0,0,1,1,0,1,1,0,0,1,1,1,1,0,1,0,1,0,0)\}$

4.3 Lower bounds

There are fewer lower bounds presented in literature, as lower bounds are often tied to particular constructions of codes. For example, if a code with a given length n and minimum distance d is produced, its size becomes a lower bound on the code size.

4.3.1 The Gilbert-Varshamov lower bound

Theorem 4.3.1 (Gilbert-Varshanov bound)**.**

$$A_q(n,d) \geq \frac{q^n}{|B^n_{d-1}|}$$

Proof. Let C be a code of length n and minimum Hamming distance d having maximal size:

$$|C| = A_q(n,d).$$

Let $y \in \mathbb{F}_q^n$ be arbitrary. If y in is not in $B^n_{d-1}(x)$ for all $x \in C$ then $\mathrm{d}(x,y) \geq d$ for every $x \in C$. Thus $C \cup \{y\}$ is a code C' of distance d, length n and $|C'| = A_q(n,d)+1$, which is impossible. Thus $y \in B^n_{d-1}(x)$ for some $x \in C$.
Therefore, the union of all balls of radius $d-1$ centered in all codewords of C must cover all $\mathbb{F}_q^n$. Hence we deduce:

$$|\mathbb{F}_q^n| = |\cup_{x \in C} B^n_{d-1}(x)|$$

$$\leq \sum_{x \in C} |B^n_{d-1}(x)|$$

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

That is:

$$A_q(n,d) \geq \frac{q^n}{\sum_{j=0}^{d-1} \binom{n}{j}(q-1)^j}$$

(using the fact $|\mathbb{F}_q^n| = q^n$). □

A generalization of the Griesmer bound to systematic codes

In this section we want to prove that the Griesmer bound applies also to systematic codes in the following cases:

- $q \geq d$
- $d = 1, 2, 3, 4$
- $q = 2, d = 5, 6$

5.1 The Griesmer bound

From now on let q be the power of a prime number, and n, k, d three integers such that a $[n, q^k, d]_q$ systematic code exists.

Theorem 5.1.1 (Griesmer bound)**.** *Let n be such that there exists an $[n, q^k]_q$ linear code with distance at least d. Then*

$$n \geq \sum_{j=0}^{k-1} \left\lceil \frac{d}{q^j} \right\rceil .$$

Lemma 5.1.2. *If $k = 1$, then for each q, n, d such that a systematic code C exists, then*

$$n \geq \sum_{j=0}^{k-1} \left\lceil \frac{d}{q^j} \right\rceil .$$

Proof. For $k = 1$ we have $\sum_{j=0}^{k-1} \left\lceil \frac{d}{q^j} \right\rceil = d$, and clearly $n \geq d$. □

In the following sections let C be a $[n, q^k, d]_q$ systematic code, with $k \geq 2$, such that $0 \in C$, and let us indicate a word of C as $c = (\bar{c}, \tilde{c})$, where $\bar{c}$ is the systematic part of c and $\tilde{c}$ is the nonsystematic part of c.

5.2 The case $q \geq d$

Theorem 5.2.1. *If $q \geq d$, for all $k \geq 2$, there exists no q_ary systematic code such that $n < \sum_{i=0}^{k-1} \lceil \frac{d}{q^i} \rceil$.*

Proof. If $q \geq d$ we have that $\lceil \frac{d}{q^i} \rceil = 1$ for all $i \geq 1$, and so:

$$\sum_{i=0}^{k-1} \lceil \frac{d}{q^i} \rceil = d + \lceil \frac{d}{q} \rceil + \cdots + \lceil \frac{d}{q^{k-1}} \rceil = d + k - 1$$

But we also have, by the Singleton bound, that $n \geq d + k - 1$. □

5.3 The case $d = 1, 2, 3, 4$

Theorem 5.3.1. *If $d = 1, 2$, than for all $k \geq 2$, there exists no q_ary systematic code such that $n < \sum_{i=0}^{k-1} \lceil \frac{d}{q^i} \rceil$.*

Proof. If $d = 1, 2$ we have $q \geq d$ an so we are in the hypothesis of Theorem 5.2.1. □

Theorem 5.3.2. *If $d = 3, 4$, then for all $k \geq 2$, there exists no q_ary systematic code such that $n < \sum_{i=0}^{k-1} \lceil \frac{d}{q^i} \rceil$.*

Proof. If $d = 3$ and $q \geq 3$ or $d = 4$ and $q \geq 4$ then we are in the hypothesis of Theorem 5.2.1.

Otherwise, if $d = 3$ and $q = 2$ or $d = 4$ and $q = 2, 3$ then we have that $\lceil \frac{d}{q} \rceil = 2$ and $\lceil \frac{d}{q^i} \rceil = 1$ for all $i \geq 2$, and so:

$$\sum_{i=0}^{k-1} \lceil \frac{d}{q^i} \rceil = d + \lceil \frac{d}{q} \rceil + \cdots + \lceil \frac{d}{q^{k-1}} \rceil = d + k$$

Suppose by contradiction that $n < d+k$. It is enough to prove the case $n = d+k-1$. Then $n-k = d-1$. Since $0 \in C$, if we consider two different words $c_1 = (\bar{c_1}, \tilde{c_1}), c_2 = (\bar{c_2}, \tilde{c_2})$ such that $\mathrm{w}(\bar{c_1}) = \mathrm{w}(\bar{c_2}) = 1$, then $\mathrm{w}(\tilde{c_1}) = \mathrm{w}(\tilde{c_2}) = d - 1$ and, if $q = 2$ then $\tilde{c_1} = \tilde{c_2} = (1, \ldots, 1)$, and so $\mathrm{d}(c_1, c_2) \leq 2$, contradiction. There is only one case left, which is the case $q = 3$ and $d = 4$. In this case, since $k \geq 2$, we have at least 9 words in C. Consider the following four words:

$$\begin{aligned} c_0 = (\bar{c_0}, \tilde{c_0}) &= (0 \ldots 0000, 0 \ldots 0) \\ c_1 = (\bar{c_1}, \tilde{c_1}) &= (0 \ldots 0001, c_{11} c_{12} c_{13}) \\ c_2 = (\bar{c_2}, \tilde{c_2}) &= (0 \ldots 0002, c_{21} c_{22} c_{23}) \\ c_3 = (\bar{c_3}, \tilde{c_3}) &= (0 \ldots 0010, c_{31} c_{32} c_{33}) \end{aligned}$$

Since the distance between c_1, c_2, c_3 from c_0 must be greater than d, then $\tilde{c_1}, \tilde{c_2}, \tilde{c_3}$ must have weigth 3. c_{11}, c_{12}, c_{13} can be any combination of 1 and 2, let us suppose $(c_{11}, c_{12}, c_{13}) = (111)$. Then, to have $\mathrm{d}(c_1, c_2) \geq 4$, we must have $(c_{21}, c_{22}, c_{23}) = (222)$. And for the same reason $(c_{31}c_{32}c_{33})$ must differ from $(c_{11}c_{12}c_{13})$ and from $(c_{21}c_{22}c_{23})$ in at least two positions at the same time, but this is not possible. □

5.4 The case $q = 2$ and $d = 5, 6$

Theorem 5.4.1. *For $k = 2$, there exists no binary systematic code such that $n < \sum_{i=0}^{k-1} \lceil \frac{d}{2^i} \rceil$ for $d = 5, 6$.*

Proof. If $k = 2$ then $\sum_{i=0}^{1} \lceil \frac{d}{2^i} \rceil = d + \lceil \frac{d}{2} \rceil = d + 3$. Suppose by contradiction that $n < d + 3$. It is enough to prove the case $n = d + 2$. Consider two different words $c_1 = (\bar{c_1}, \tilde{c_1}), c_2 = (\bar{c_2}, \tilde{c_2})$ such that $\mathrm{w}(\bar{c_1}) = \mathrm{w}(\bar{c_2}) = 1$, then $\mathrm{w}(\tilde{c_1}) = \mathrm{w}(\tilde{c_2}) \geq d - 1$. Since $n - k = d$, then $\mathrm{d}(\tilde{c_1}, \tilde{c_2}) \leq 2$ and thus $\mathrm{d}(c_1, c_2) \leq 4$. □

Theorem 5.4.2. *For all $k \geq 3$, there exists no binary systematic code such that $n < \sum_{i=0}^{k-1} \lceil \frac{d}{2^i} \rceil$ for $d = 5, 6$.*

Proof. If $d = 5, 6$, we have that $\lceil \frac{d}{2} \rceil = 3$, $\lceil \frac{d}{4} \rceil = 2$ and $\lceil \frac{d}{2^i} \rceil = 1$ for all $i \geq 3$, and so:

$$\sum_{i=0}^{k-1} \lceil \frac{d}{2^i} \rceil = d + \lceil \frac{d}{2} \rceil + \lceil \frac{d}{4} \rceil + \cdots + \lceil \frac{d}{2^{k-1}} \rceil = d + 5 + k - 3 = d + k + 2$$

Suppose by contradiction that $n < d+k+2$. It is enough to prove the case $n = d+k+1$, so that $n - k = d + 1$. Let us consider the following five words:

$$\begin{aligned}
c_0 &= (\bar{c_0}, \tilde{c_0}) = (0 \ldots 0000, 0 \ldots 0) \\
c_1 &= (\bar{c_1}, \tilde{c_1}) = (0 \ldots 0001, c_{11} \ldots c_{1,d+1}) \\
c_2 &= (\bar{c_2}, \tilde{c_2}) = (0 \ldots 0010, c_{21} \ldots c_{2,d+1}) \\
c_3 &= (\bar{c_3}, \tilde{c_3}) = (0 \ldots 0011, c_{31} \ldots c_{3,d+1}) \\
c_5 &= (\bar{c_5}, \tilde{c_5}) = (0 \ldots 0101, c_{51} \ldots c_{5,d+1})
\end{aligned}$$

We want to show that there is no way to assign 0 or 1 to the c_{ij} to obtain distance d between these five words. Let us do the following considerations:

1. to have $\mathrm{d}(c_1, c_0) = \mathrm{w}(c_1) \geq d$ and $\mathrm{d}(c_2, c_0) = \mathrm{w}(c_2) \geq d$, it must be that $\mathrm{w}(\tilde{c_1}), \mathrm{w}(\tilde{c_2}) \geq d - 1$. Clearly it is not possible that $\mathrm{w}(\tilde{c_1}), \mathrm{w}(\tilde{c_2}) \geq d$, otherwise $\mathrm{d}(c_1, c_2) \leq 4$. So, wlog, we have only one of the two following cases:

   (a) either $\mathrm{w}(\tilde{c_1}) = d$ and $\mathrm{w}(\tilde{c_2}) = d - 1$,

(b) or $\mathrm{w}(\tilde{c_1}) = \mathrm{w}(\tilde{c_2}) = d - 1$.

2. to have $\mathrm{w}(c_3), \mathrm{w}(c_5) \geq d$, it must be that $\mathrm{w}(\tilde{c_3}), \mathrm{w}(\tilde{c_5}) \geq d - 2$.

Consider the case (1.a). Since $\mathrm{d}(\bar{c_1}, \bar{c_3}) = 1$ and $\mathrm{w}(\tilde{c_1}) = d$, the only way to have distance at least d between c_0, c_1, c_3 (modulo the permutation of the colums) is to assign the following values to c_1, c_3:

$$\begin{aligned}
c_0 &= (\bar{c_0}, \tilde{c_0}) = (0 \ldots 0000, 000000) \\
c_1 &= (\bar{c_1}, \tilde{c_1}) = (0 \ldots 0001, 011111) \\
c_3 &= (\bar{c_3}, \tilde{c_3}) = (0 \ldots 0011, 111000) \\
c_2 &= (\bar{c_2}, \tilde{c_2}) = (0 \ldots 0010, c_{21} \ldots c_{2,d+1}) \\
c_5 &= (\bar{c_5}, \tilde{c_5}) = (0 \ldots 0101, c_{51} \ldots c_{5,d+1})
\end{aligned}$$

in case $d = 5$ and:

$$\begin{aligned}
c_0 &= (\bar{c_0}, \tilde{c_0}) = (0 \ldots 0000, 0000000) \\
c_1 &= (\bar{c_1}, \tilde{c_1}) = (0 \ldots 0001, 0111111) \\
c_3 &= (\bar{c_3}, \tilde{c_3}) = (0 \ldots 0011, 1111000) \\
c_2 &= (\bar{c_2}, \tilde{c_2}) = (0 \ldots 0010, c_{21} \ldots c_{2,d+1}) \\
c_5 &= (\bar{c_5}, \tilde{c_5}) = (0 \ldots 0101, c_{51} \ldots c_{5,d+1})
\end{aligned}$$

in the case $d = 6$.
This allows to have $\mathrm{d}(\tilde{c_1}, \tilde{c_3}) = d - 1$, which is the only we can reach in our conditions. Now consider c_2. $\tilde{c_1}$ has only a zero and d ones, and $\tilde{c_2}$ has $d - 1$ ones, which are either in the same postitions of the ones in $\tilde{c_1}$ (this case is impossible because otherwise $\mathrm{d}(\tilde{c_1}, \tilde{c_2}) = 1 \implies \mathrm{d}(c_1, c_2) = 3$) or $c_{21} = 1$. Since $\mathrm{w}(c_2) = d$, there remain d bits to be filled in $\tilde{c_2}$, and $d - 2$ of this bit must be ones and the other 2 zeros. Since $c_{31} = c_{21} = 1$, to have $\mathrm{d}(\tilde{c_3}, \tilde{c_2}) \geq d - 1$, at least $d - 1$ of the d rightmost bits must differ. Thus we have the following situation in case $d = 5$:

$$\begin{aligned}
c_0 &= (\bar{c_0}, \tilde{c_0}) = (0 \ldots 0000, 000000) \\
c_1 &= (\bar{c_1}, \tilde{c_1}) = (0 \ldots 0001, 011111) \\
c_3 &= (\bar{c_3}, \tilde{c_3}) = (0 \ldots 0011, 111000) \\
c_2 &= (\bar{c_2}, \tilde{c_2}) = (0 \ldots 0010, 100111) \\
c_5 &= (\bar{c_5}, \tilde{c_5}) = (0 \ldots 0101, c_{51} \ldots c_{56})
\end{aligned}$$

and in the following situation in case $d = 6$:

$$\begin{aligned} c_0 = (\bar{c_0}, \tilde{c_0}) &= (0 \dots 0000, 0000000) \\ c_1 = (\bar{c_1}, \tilde{c_1}) &= (0 \dots 0001, 0111111) \\ c_3 = (\bar{c_3}, \tilde{c_3}) &= (0 \dots 0011, 1111000) \\ c_2 = (\bar{c_2}, \tilde{c_2}) &= (0 \dots 0010, 1001111) \\ c_5 = (\bar{c_5}, \tilde{c_5}) &= (0 \dots 0101, c_{51} \dots c_{57}) \end{aligned}$$

Now, $\tilde{c_5}$ must be such that $\mathrm{w}(\tilde{c_5}) \geq d-2$ and $\mathrm{d}(\tilde{c_5}, \tilde{c_1}) \geq d-1$. Thus in $\tilde{c_5}$ there must be at most $d+1-(d-2) = 3$ zero components, which is a contradiction because, in the case $d = 5$, if:

$$\begin{aligned} c_0 = (\bar{c_0}, \tilde{c_0}) &= (0 \dots 0000, 000000) \\ c_1 = (\bar{c_1}, \tilde{c_1}) &= (0 \dots 0001, 011111) \\ c_3 = (\bar{c_3}, \tilde{c_3}) &= (0 \dots 0011, 111000) \\ c_2 = (\bar{c_2}, \tilde{c_2}) &= (0 \dots 0010, 100111) \\ c_5 = (\bar{c_5}, \tilde{c_5}) &= (0 \dots 0101, 100011) \end{aligned}$$

or, in the case $d = 6$, if:

$$\begin{aligned} c_0 = (\bar{c_0}, \tilde{c_0}) &= (0 \dots 0000, 0000000) \\ c_1 = (\bar{c_1}, \tilde{c_1}) &= (0 \dots 0001, 0111111) \\ c_3 = (\bar{c_3}, \tilde{c_3}) &= (0 \dots 0011, 1111000) \\ c_2 = (\bar{c_2}, \tilde{c_2}) &= (0 \dots 0010, 1001111) \\ c_5 = (\bar{c_5}, \tilde{c_5}) &= (0 \dots 0101, 1000111) \end{aligned}$$

then $\mathrm{d}(c_2, c_5) = 4$.
Let us try now with case (1.b), so that we know $\mathrm{w}(\tilde{c_1}) = \mathrm{w}(\tilde{c_2}) = d-1$. We also have that $\mathrm{d}(\tilde{c_1}, \tilde{c_2}) \geq d-2$, and at the same time $\mathrm{d}(\tilde{c_1}, \tilde{c_2})$ can only be $0, 2, 4$, since there are only two zeros components both in $\tilde{c_1}$ and in $\tilde{c_2}$, and c_1 and c_2 have the same parity. Since $d-2 > 2$, then $\mathrm{d}(\tilde{c_1}, \tilde{c_2})$ must be 4 and the only choice (modulo permutation of the columns) for $\tilde{c_1}, \tilde{c_2}$ is:

$$\begin{aligned} c_0 = (\bar{c_0}, \tilde{c_0}) &= (0 \dots 0000, 000000|0) \\ c_1 = (\bar{c_1}, \tilde{c_1}) &= (0 \dots 0001, 001111|1) \\ c_2 = (\bar{c_2}, \tilde{c_2}) &= (0 \dots 0010, 111100|1) \\ c_4 = (\bar{c_4}, \tilde{c_4}) &= (0 \dots 0100, c_{41} \dots c_{4,d+1}) \\ c_3 = (\bar{c_3}, \tilde{c_3}) &= (0 \dots 0011, c_{31} \dots c_{3,d+1}) \end{aligned}$$

where the rightmost component exists only in the case $d = 6$.
Now consider c_4, which must be such that $\mathrm{w}(\tilde{c_4}) = d - 1$ (it can not be d or $d + 1$, otherwise we would be in a similar case to (1.a)), so that it has two zero components which, using a reasoning similar to that for $\tilde{c_1}$ and $\tilde{c_2}$, to have $\mathrm{d}(\tilde{c_1}, \tilde{c_4}) = \mathrm{d}(\tilde{c_4}, \tilde{c_2}) = 4$, must be positioned as follows:

$$\begin{aligned}
c_0 &= (\bar{c_0}, \tilde{c_0}) = (0 \ldots 0000, 000000|0) \\
c_1 &= (\bar{c_1}, \tilde{c_1}) = (0 \ldots 0001, 001111|1) \\
c_2 &= (\bar{c_2}, \tilde{c_2}) = (0 \ldots 0010, 111100|1) \\
c_4 &= (\bar{c_4}, \tilde{c_4}) = (0 \ldots 0100, 110011|1) \\
c_3 &= (\bar{c_3}, \tilde{c_3}) = (0 \ldots 0011, c_{31} \ldots c_{3,d+1})
\end{aligned}$$

Now, c_3 is such that $\mathrm{w}(\tilde{c_3}) \geq d - 2$.
$\mathrm{w}(\tilde{c_3}) = d + 1$ or $\mathrm{w}(\tilde{c_3}) = d$ is not possible, otherwise we would have $\mathrm{d}(c_3, c_1) \leq 4$.
$\mathrm{w}(\tilde{c_3}) = d - 1$ is not possible in the case $d = 6$, because, having only two zeros the value $\mathrm{d}(c_3, c_1)$ can be at most 5.
In the case $d = 5$, if $\mathrm{w}(\tilde{c_3}) = d - 1$, to have $\mathrm{d}(c_3, c_1) \geq 5$ the two leftmost component of $\tilde{c_3}$ must be the same as the two leftmost component of $\tilde{c_2}$ and of $\tilde{c_4}$, which are ones, giving either $\mathrm{d}(c_3, c_2) = 5$ and $\mathrm{d}(c_3, c_4) = 3$, or $\mathrm{d}(c_3, c_2) < 5$, which is a contradiction.
It remains to prove that $\mathrm{w}(\tilde{c_3}) \neq d - 2$. In this case, again, to have $\mathrm{d}(c_3, c_1) \geq d$ the two leftmost component of $\tilde{c_3}$ must be the same as the two leftmost component of $\tilde{c_2}$ and of $\tilde{c_4}$, which are ones, obtaining actually $\mathrm{d}(c_3, c_1) = 6$. In the remaining components of $\tilde{c_3}$ there must be three zeros. If these three zeros are in the same positions where the leftmost ones of $\tilde{c_2}$ are, then $\mathrm{d}(c_3, c_2) = d$ and $\mathrm{d}(c_3, c_4) \leq 4$. Otherwise $\mathrm{d}(c_3, c_2) < d$.
This completes our proof. □

A new bound on the size of codes

Partial results in this chapter have been presented at WCC 2013, Bergen, while a full updated manuscript can be found in [BGS14].
In this chapter we present one (closed-formula) bound (Bound $\mathcal{A}$) for a large part of non-linear codes (including all systematic codes), which is an improvement of a bound first introduced by Zinoviev and Litsyn ([ZL84]), and then applied by Litsyn and Laihonen ([LL98]). The crux of our improvement is a preliminary result presented in Section 6.1, while in Section 6.2 we are able to prove Bound $\mathcal{A}$. Then we restrict Bound $\mathcal{A}$ to the systematic/linear case and compare it with many classical upper bounds by computing their values for a large set of parameters (corresponding to about one week of computations with our computers). Our findings are in favour of Bound $\mathcal{A}$ and are reported in Section 6.3. For large values of q, our bound provides the best value in the majority of cases.
The only closed-formula bound we never beat is Plotkin's, but its range is very small (it must be $d > n(1 - 1/q)$), which it becomes barely applicable for large q's.

For standard definitions and known bounds, the reader is directed to the original articles or to any recent good book, e.g. [HP03] or [PBH98].

6.1 A first result for a special family of codes

Recall that $A_q(n, d)$ denotes the maximum number of codewords in a code over $\mathbb{F}_q$ of length n and minimum distance d.
If we have extra constraints on the weight of the codewords, then the maximum number of codewords can be smaller than $A_q(n, d)$. The following result is an example and it will be instrumental for the proof of Bound $\mathcal{A}$.

Proposition 6.1.1. *Let C be an $(n, |C|, d)_q$-code. Let $\epsilon \in \mathbb{N}$ and $\epsilon \geq 1$ be such that for any $c \in C$ we have $\mathrm{w}(c) \geq d + \epsilon$. Then*

$$|C| \leq A_q(n, d) - \frac{|B^n_\epsilon|}{|B^n_{d-1}|}.$$

Proof. Let C be the code satisfying our hypothesis. C belongs to the set of all codes with distance d that are contained in the ball exterior $\mathbb{F}^n \setminus B^n_{d+\epsilon-1}(0)$. Let D be any

code of the largest size in this set, then $|C| \leq |D|$.
Clearly, any codeword c of D has weight $\mathrm{w}(c) \geq d + \epsilon$. Consider also $\bar{D}$, the largest code in $\mathbb{F}^n$ of distance d and such that $D \subseteq \bar{D}$. By definition, the only codewords of $\bar{D}$ of weight greater than $d + \epsilon - 1$ are those of D, while all other codewords of $\bar{D}$ are confined to the ball $B^n_{d+\epsilon-1}(0)$. Thus:

$$|C| \leq |D| \leq |\bar{D}| \leq A_q(n, d)$$

and

$$\bar{D} \setminus D \subseteq B^n_{d+\epsilon-1}(0).$$

Let $\rho = d - 1$ and $r = d + \epsilon - 1$, so that $r - \rho = \epsilon$, and let $N = \bar{D} \cap B^n_r(0)$. We have: $D = \bar{D} \setminus N, |D| = |\bar{D}| - |N|$. We are searching for a lower bound on $|N|$, in order to have an upper bound on $|D|$.
We start with proving $B^n_{r-\rho}(0) \subseteq \bigcup_{x \in N} B^n_\rho(x)$. Consider $y \in B^n_{r-\rho}(0)$. If for all $x \in N$ we have that $y \notin B^n_\rho(x)$, then y is a vector whose distance from N is at least $\rho + 1$. Since $y \in B^n_{r-\rho}(0)$, also its distance from $\bar{D} \setminus N$ is at least $\rho + 1$. Therefore, the distance of y from the whole $\bar{D}$ is at least $\rho + 1 = d$ and so we can obtain a new code $\bar{D} \cup \{y\}$ containing D and with distance d, contradicting the fact that $|\bar{D}|$ is the largest size for such a code in $\mathbb{F}^n$.
So, our claim must hold, and its consequence is:

$$|N| \cdot |B^n_\rho(x)| \geq |B^n_{r-\rho}(0)|,$$

which gives:

$$|N| \geq \frac{|B^n_{r-\rho}(0)|}{|B^n_\rho(x)|} = \frac{|B^n_\epsilon(0)|}{|B^n_{d-1}(x)|}.$$

Using previous observations, we obtain the desired bound:

$$\begin{aligned} |C| \leq |D| &= |\bar{D}| - |\bar{D} \cap B^n_{d+\epsilon-1}(0)| \\ &\leq A_q(n, d) - \frac{|B^n_\epsilon(0)|}{|B^n_{d-1}(x)|}. \end{aligned}$$

□

Note that if C is a linear code then Proposition 6.1.1 is not directly applicable, since we would have $\epsilon = 0$ (but it might still be applicable by translating the code with a suitable vector). Note also that the theorem is specially interesting when $|B^n_\epsilon|/|B^n_{d-1}|$ is large, although any positive value of ϵ would give (in the worst case) an upper bound of $A_q(n, d) - 1$, since we can obviously write

$$|C| \leq A_q(n, d) - \left\lceil \frac{|B^n_\epsilon|}{|B^n_{d-1}|} \right\rceil.$$

6.2 An improvement of the ZLL bound

In 1998 Litsyn and Laihonen prove a bound for non-linear codes: Theorem 1 of [LL98], which we write with our notation as follows.

Theorem 6.2.1 (Zinoviev-Litsyn-Laihonen bound)**.** *Let $1 \leq d \leq n$. Let $t \in \mathbb{N}$ be such that $t \leq n-d$. Let $r \in \mathbb{N}$ be such that $0 \leq r \leq t$ and $0 \leq r \leq \frac{1}{2}d$. Then*

$$A_q(n,d) \leq \frac{q^t}{|B_r^t|} A_q(n-t, d-2r).$$

Note that $t \leq n-d$ implies $d-2r \leq n-t$ so that the value $A_q(n-t,d-2r)$ is meaningful.
Let C be an $(n,|C|,d)_q$-code, let $k = \lfloor \log_q(|C|) \rfloor$. We say that C is *systematic-embedding* if C contains a systematic code D with size $|D| = q^k$. Obviously, a systematic code is systematic-embedding with $D = C$. Moreover, if the code is linear then k is the dimension of C.
All known families of maximal codes are either systematic codes or systematic-embedding codes (see e.g., [Pre68], [Ker72] and [BvLW83]).

Definition 6.2.2. *We denote with $A_q^*(n,d)$ the maximum number of codewords that an $(n,|C|,d)$-systematic-embedding code can contain.*

Although we can only say that $A_q^*(n,d) \leq A_q(n,d)$, there are no known counterexamples to $A_q^*(n,d) = A_q(n,d)$.
We are ready to show a strengthening of Theorem 6.2.1 restricted to systematic-embedding codes: Bound $\mathcal{A}$. In the proof we follow initially the outline of the proof of [LL98][Theorem 1] and then we apply Proposition 6.1.1.

Theorem 6.2.3 (Bound $\mathcal{A}$)**.** *Let $2 \leq d \leq n$. Let $t \in \mathbb{N}$ be such that $t \leq n-d$. Let $r \in \mathbb{N}$ be such that $0 \leq r \leq t$ and $0 \leq r \leq \frac{1}{2}d$. Let C be any $(n,|C|,d)_q$-systematic-embedding code such that $|C| = A_q^*(n,d)$. Let $t \leq k = \lfloor \log_q(|C|) \rfloor$. Then*

$$A_q^*(n,d) \leq \\ \frac{q^t}{|B_r^t|}\left(A_q(n-t,d-2r) - \frac{|B_r^{n-t}|}{|B_{d-2r-1}^{n-t}|} + 1\right).$$

Proof. We consider an $(n,|C|,d)$-systematic-embedding code C s.t. $|C| = A_q^*(n,d)$. We number all words in C in any order:

$$C = \{c_i \mid 1 \leq\ i \leq A_q^*(n,d)\}.$$

We indicate the i-th codeword with $c_i = (c_{i,1}, \ldots, c_{i,n})$. We puncture C as follows:

(i) we choose any t columns among the k columns of the systematic part of C, $1 \leq j_1, \ldots, j_t \leq n$; since two codes are equivalent w.r.t. column permutations we can suppose $j_1 = 1, \ldots, j_t = t$.
Let us split each codeword $c_i \in C$ in two parts, $c_i = (\tilde{c_i}, \bar{c_i})$, with:
$\tilde{c_i} = (c_{i,1}, \ldots, c_{i,t}), \quad \bar{c_i} = (c_{i,t+1}, \ldots, c_{i,n})$.

(ii) We choose a $z \in \mathbb{F}^t$.

(iii) We collect in I all i's s.t. $\mathrm{d}(z, \tilde{c_i}) \leq r$;

(iv) We delete the first t components of $\{c_i \mid i \in I\}$.

Then the punctured code $\bar{C}_z$ obtained by (i),(ii),(iii), (iv) is:

$$\bar{C}_z = \{\bar{c_i} \mid i \in I\} = \{\bar{c_i} \mid 1 \leq i \leq A_q^*(n,d), \mathrm{d}(z, \tilde{c_i}) \leq r\}.$$

We claim that we can choose z in such a way that $\bar{C}_z$ is equivalent to a code with the following properties:

$$\bar{n} = n - t \tag{6.1}$$

$$\bar{d} \geq d - 2r \tag{6.2}$$

$$|\bar{C}_z| \geq \frac{|C|}{q^t}|B_r^t| \tag{6.3}$$

$$\mathrm{w}(\bar{c_i}) \geq d - r \text{ for all } \bar{c_i} \neq 0. \tag{6.4}$$

(6.1) is obvious.
As regards (6.2), note that $\mathrm{d}(c_i, c_j) = \mathrm{d}(\tilde{c_i}, \tilde{c_j}) + \mathrm{d}(\bar{c_i}, \bar{c_j}) \geq d$ and also that $\tilde{c_i}, \tilde{c_j} \in B_r^t(z)$ implies $\mathrm{d}(\tilde{c_i}, \tilde{c_j}) \leq 2r$. Therefore

$$2r + \mathrm{d}(\bar{c_i}, \bar{c_j}) \geq \mathrm{d}(\tilde{c_i}, \tilde{c_j}) + \mathrm{d}(\bar{c_i}, \bar{c_j}) \geq d$$

for any $i \neq j$. The proof of (6.3) is more involved and we need to consider the average number M of the i's such that $\tilde{c_i}$ happens to be in a sphere of radius r centered at a fixed word in $\mathbb{F}^t$. The average is taken over all sphere centers, that is, all vectors x's in $\mathbb{F}^t$, so that

$$M = \frac{1}{|\mathbb{F}^t|} \sum_{x \in \mathbb{F}^t} |\{i \mid 1 \leq i \leq A_q^*(n,d), \tilde{c_i} \in B_r^t(x)\}|.$$

Let us define a function:

$$\psi : \mathbb{F}^t \times \mathbb{F}^t \longrightarrow \{0,1\}, \psi(x,y) = \begin{cases} 1, & \mathrm{d}(x,y) \leq r \\ 0, & \text{otherwise} \end{cases}.$$

Then we can write M and $|B_r^t(y)|$ $(\forall y \in \mathbb{F}^t)$ as

$$M = \frac{1}{q^t} \sum_{x \in \mathbb{F}^t} \sum_{i=1}^{A_q^*(n,d)} \psi(x, \tilde{c}_i) \text{ and } |B_r^t(y)| = \sum_{x \in \mathbb{F}^t} \psi(x, y)\,.$$

By swapping variables we get:

$$M = \frac{1}{q^t} \sum_{x \in \mathbb{F}^t} \sum_{i=1}^{A_q^*(n,d)} \psi(x, \tilde{c}_i) =$$

$$= \frac{1}{q^t} \sum_{i=1}^{A_q^*(n,d)} \sum_{x \in \mathbb{F}^t} \psi(x, \tilde{c}_i) \qquad = \frac{A_q^*(n,d)}{q^t} |B_r^t(\tilde{c}_i)|\,.$$

This means that there exists $\hat{x} \in \mathbb{F}^t$ such that:

$$|\{i \mid 1 \leq i \leq A_q^*(n,d), \tilde{c}_i \in B_r^t(\hat{x})\}| \geq M \geq \frac{A_q^*(n,d)}{q^t} |B_r^t|\,.$$

In other words, there are at least $\frac{|C|}{q^t}|B_r^t|$ c_i's such that their $\tilde{c}_i$'s are contained in $B_r^t(\hat{x})$. Distinct c_i's may well give rise to the same $\tilde{c}_i$'s, but they always correspond to distinct $\bar{c}_i$'s (see the proof of (6.2)), so there are at least $\frac{|C|}{q^t}|B_r^t|$ (distinct) $\bar{c}_i$'s such that their corresponding $\tilde{c}_i$'s fall in $B_r^t(\hat{x})$. By choosing $z = \hat{x}$ we then have at least $\frac{|C|}{q^t}|B_r^t|$ (distinct) codewords of $\bar{C}_z$ and so (6.3) follows.

We claim that (6.4) holds if $0 \in C$ and $z = 0$. In fact:
$\mathrm{w}(c) = \mathrm{d}(0, c) \geq d$, $\forall c \in C$ such that $c \neq 0$, and
$z = 0 \implies y \in B_r^t(z) \iff \mathrm{w}(y) \leq r$.
As a consequence, any nonzero codeword $c_i = (\tilde{c}_i, \bar{c}_i)$ of weight at most r in $\tilde{c}_i$ has weight at least $d - r$ in the other $n - t$ components.
If $0 \notin C$ or $z \neq 0$ we consider a code $C + v$ equivalent to C, by choosing the translation v in the following way. By hypothesis of systematic-embedding there exists $\hat{c} \in C$ such that its first t coordinates form the vector $\hat{x}$. By considering $v = \hat{c}$ we obtain the desired code, thus (6.4) is proved.
Now we call X the largest $(\bar{n}, |X|, d-2r)$-code containing the zero codeword and such that $\mathrm{w}(\bar{x}) \geq d - r = (d - 2r) + r$, $\forall \bar{x} \in X$. Observe that X satisfies (6.1), (6.2), (6.3), (6.4) and so $|X| \geq |\bar{C}_z|$. Then we can apply Proposition 6.1.1 to $X \setminus \{0\}$ and $\epsilon = r$, and obtain the following chain of inequalities:

$$\frac{|C|}{q^t}|B_r^t| \leq |\bar{C}_z| \leq |X| \leq A_q(\bar{n}, d - 2r) - \frac{|B_r^{\bar{n}}|}{|B_{d-2r-1}^{\bar{n}}|} + 1,$$

and since $|C| = A_q^*(n,d)$ we have the bound:

$$A_q^*(n,d) \leq \frac{q^t}{|B_r^t|} \left(A_q(\bar{n}, d - 2r) - \frac{|B_r^{\bar{n}}|}{|B_{d-2r-1}^{\bar{n}}|} + 1 \right).$$

□

Note that Bound $\mathcal{A}$ is trivial if $r \leq d-2r-1$. Note also that it may be generalized to any alphabet.

6.2.1 Restriction to the systematic case

When we restrict ourselves to the systematic/linear case, then the maximum number of codewords of a code of length n and distance d can only be a power of q, and if the dimension of the code C is k, then the value $A_q^*(n,d)$ is replaced by q^k. By choosing $t = k$ in Theorem 6.2.3 we have the following:

Corollary 6.2.4 (Bound $\mathcal{B}$)**.** *Let $k, d, r \in \mathbb{N}, d \geq 2, k \geq 1$. Let n be such that there exists an $(n, q^k)_q$-systematic code C with distance at least d.*
If $0 \leq r \leq \frac{d}{2}$, then:

$$|B_r^k| \leq A_q(n-k, d-2r) - \frac{|B_r^{n-k}|}{|B_{d-2r-1}^{n-k}|} + 1.$$

6.2.2 Theoretical comparison with the ZLL bound

In the systematic/linear case the Zinoviev-Litsyn-Laihonen bound becomes:

$$|B_r^k| \leq A_q(n-k, d-2r).$$

The case $d \leq 2$ is trivial.
The first interesting computations can be done in the case $d = 3$, since in this case r can take the value 1, so that:

- $|B_1^k| = (q-1)k + 1$,
- $A_q(n-k, d-2r) = A_q(n-k, 1) = q^{n-k}$,
- $|B_1^{n-k}| = (q-1)(n-k) + 1$,
- $|B_{d-2r-1}^{n-k}| = |B_0^{n-k}| = 1$.

Our bound then reduces to:

$$0 \leq q^{n-k} - (q-1)n - 1\,,$$

and so it is stronger than the Zinoviev-Litsyn-Laihonen bound, which reduces to:

$$0 \leq q^{n-k} - (q-1)k - 1\,.$$

For $d > 3$ and when restricted to the linear/systematic case, Bound $\mathcal{B}$ and the Zinoviev-Litsyn-Laihonen bound are very close. This happens because

$$\lfloor \log_q(A - \frac{|B_r^{n-k}|}{|B_{d-2r-1}^{n-k}|} + 1) \rfloor \approx \lfloor \log_q(A) \rfloor$$

where $A = A_q(n-k, d-2r)$, since the floor function cuts off the difference between the two bounds.
To have a fair comparison, the two bounds should be studied in the nonlinear case. Here it is clear that Bound $\mathcal{B}$ beats the Zinoviev-Litsyn-Laihonen bound if and only if:

$$\frac{q^t}{|B_r^t|}\left(\frac{|B_r^{n-k}|}{|B_{d-2r-1}^{n-k}|}-1\right)\geq 1.$$

Since $\frac{q^t}{|B_r^t|}\geq 1$ we only need that $|B_r^{n-k}|\geq 2|B_{d-2r-1}^{n-k}|$, which is:

$$\sum_{j=0}^{r}\binom{n-t}{j}(q-1)^j \geq 2\sum_{j=0}^{d-2r-1}\binom{n-t}{j}(q-1)^j.$$

For this inequality to hold it is sufficient that $r\sim(d-1)/2$ (since $0\leq r\leq \frac{d}{2}$) with d large enough.

6.3 Experimental comparisons: linear case

We have analyzed the case of linear codes, implementing Bound $\mathcal{B}$. The algorithm to compute the bound takes as inputs n, d, and returns the largest k (checks are done until $k = n-d+1$) such that the inequality of the bound holds. If the inequality always holds in this range, $n-d+1$ is returned. Then we compared our upper bound on k with other bounds, restricting those holding in the general non-linear case to the systematic case. In particular, they provide a bound on $A_q(n,d)$ instead of a bound on k. As a consequence, for example, if the Johnson bound returns the value $A_q(n,d)$ for a certain pair (n,d), then we compare our bound with the value $\lfloor \log_q(A_q(n,d))\rfloor$, which is the largest power s of q such that $q^s\leq A_q(n,d)$.
The inequality in Theorem 6.2.4 involves the value $A_q(n-k, d-2r)$, which is the maximum number of codewords that we can have in a *non-linear* code of length $n-k$ and distance $d-2r$. To implement Bound $\mathcal{B}$ it is necessary to compute $A_q(n-k, d-2r)$; when this value is unknown (we use known values only in the binary case for $n=3,\ldots,28, d=3,\ldots,16$), we return instead an upper bound on it, choosing the best among the Hamming (Sphere Packing), Singleton, Johnson, and Elias bound (the Plotkin bound is used when possible). Even though the Levenshtein bound is a very strong bound, we do not use it because it performs very slow as n grows, and neither we use the Linear Programming bound. This means that if better values of $A_q(n-k, d-2r)$ can be found, then Bound $\mathcal{B}$ could return even tighter results.
Table 6.1 and 6.2 show a comparison between all bounds' performance, except for Plotkin's, due to its restricted range. For each bound and for each q power of a

prime in the range $\{2, \ldots, 29\}$ we have computed, for all values $n = 3, \ldots, 100$ and $d = 3, \ldots, n-1$, the percentage of cases where Bound $\mathcal{B}$ is the "best" known bound among the Griesmer, Johnson, Levenshtein, Elias, Hamming, Singleton bound, and Bound $\mathcal{B}$. Both wins and draws are counted in the percentage, since more than one bound may reach the best known bound, and in this case we increased the percentage of each best bound. Up to $q = 7$ the Levenshtein bound is the most performing bound. From $9 \leq q \leq 29$ we have that Bound $\mathcal{B}$ is the most performing bound, and in the case $q = 29$ it is the best known bound almost 91% of the times.
It can be shown that there are some cases where Bound $\mathcal{B}$ is tight, as for the parameters $(17, 7)_9$, for which there exists a code with dimension 10.
The first line of Tables 6.3, and 6.4 give emphasis to the percentage of times Bound $\mathcal{B}$ improves the best known bound (thus the cases where it beats all other bounds). In the considered range, Bound $\mathcal{B}$ starts to beat all other bounds from $q = 7$. The second line represents the percentage of the ties.
The third row of Tables 6.3 and 6.4 shows how many times (percentage over the number of draws and wins) the value $\delta = \lfloor \frac{|B(r,n-k)|}{|B(d-2r-1,n-k)|} \rfloor$ is different from zero. Informally, we can view δ as the probability to randomly pick up a codeword of weight less than r from a ball of radius $d-2r-1$. We can notice that this percentage is very high, which means that a weaker version of Bound $\mathcal{B}$, which is similar to the Zinoviev-Litsyn-Laihonen bound for systematic codes, could be used, by simply searching the largest k satisfying:

$$|B_r^k| \leq A_q(n-k, d-2r) + 1$$

It is curious to notice that in all the wins we have $\delta = 0$, and that $\delta = 0$ also 38094 times over the 46967 ties and wins. This means that the weaker version of Bound $\mathcal{B}$ is sufficient to obtain most of the wins and ties in the investigated cases.
Comparisons have been made using inner MAGMA ([MAG]) implementations of known upper bounds, except for the Johnson bound. For this bound we noted that the inner MAGMA implementation could be improved.

6.4 Experimental comparisons: nonlinear case

Since systematic-embedding codes are a subset of nonlinear codes, an upper bound on a systematic-embedding code implies an upper bound on a nonlinear code. Clearly nothing can be said in the opposite case. So we can compare Bound $\mathcal{A}$ with other bounds on nonlinear codes, such as the Linear Programming bound. In Table 6.5 some of these comparison are reported. In the first and the third rows it can be seen that Bound $\mathcal{A}$ ties with the Linear Programming bound, which is beaten in all the other rows. Also a bound from Schrijver ([Sch05]) is beaten for $A_2(20, 8)$, even though

q	2	3	4	5	7	8	9	11
$\mathcal{B}$	38.0	31.2	31.2	32.0	40.7	48.6	**55.3**	**66.4**
J	40.6	31.1	33.5	35.1	35.7	35.5	35.1	33.3
H	18.1	15.6	16.4	16.4	16.0	15.9	15.6	14.7
G	56.3	39.8	32.3	29.1	30.9	37.0	43.3	55.2
L	**72.6**	**69.7**	**66.3**	**64.0**	**60.8**	**58.2**	54.5	46.3
E	6.9	32.2	38.2	40.0	40.8	40.1	37.2	31.4
S	0.0	0.02	0.08	0.19	0.61	0.93	1.24	3.62

Table 6.1: When each bound is the best for $2 \leq q \leq 11$.

q	13	16	17	19	23	25	27	29
$\mathcal{B}$	**76.4**	**81.6**	**82.8**	**85.4**	**88.1**	**88.7**	**89.4**	**90.8**
J	30.8	26.6	24.9	21.9	17.1	15.5	14.4	13.3
H	13.6	11.9	11.3	10.1	8.27	7.55	7.05	6.61
G	63.4	71.9	72.3	71.9	69.8	69.4	68.7	67.9
L	40.0	32.9	30.7	27.5	22.6	20.7	19.4	18.4
E	27.0	21.8	20.0	17.6	12.5	10.8	9.66	8.69
S	4.44	4.63	6.99	6.71	10.1	12.0	14.1	18.0

Table 6.2: When each bound is the best for $13 \leq q \leq 29$.

for these values it is known that the best known bound is 256 from Brouwer's tables ([Broa]). Finally an improvement in the ternary case of [Brob] is given in the last row.

6.5 Tables

The following tables show the results computed in the range $n = 3, \ldots, 100$, $d = 3, \ldots, n-1$.
In Tables 6.1 and 6.2 the following letters have the following meaning: J for Johnson, H for Hamming, G for Griesmer, L for Levenshtein, E for Elias, S for Singleton, and $\mathcal{B}$ for our bound. In Table 6.5 the following letters have the following meaning: S for Schrijver bound [Sch05], LP for the Linear Programming bound, and BR for the bound in Brouwer's tables ([Broa], [Brob]).

q	2	3	4	5	7	8	9	11
W	0	0	0	0	0.19	1.05	1.83	3.96
D	38.0	31.2	31.2	32.0	39.9	47.6	53.5	62.4
$\delta =$ 0	44.7	71.1	61.8	59.8	68.8	74.2	79.7	85.4

Table 6.3: Statistics for Bound $\mathcal{B}$ for $2 \leq q \leq 11$.

q	13	16	17	19	23	25	27	29
W	3.51	4.21	5.11	7.57	14.7	17.6	19.8	21.2
D	72.9	77.4	77.7	77.8	73.4	71.1	69.6	69.6
$\delta =$ 0	88.0	88.5	88.5	88.3	85.0	83.0	80.9	78.6

Table 6.4: Statistics for Bound $\mathcal{B}$ for $13 \leq q \leq 29$.

q	n	d	$\mathcal{A}$	S	LP	BR
2	19	8	145	142	145	135
2	20	8	271	274	290	256
2	22	10	95	87	95	84
2	25	10	537	503	551	466
2	26	10	933	886	1040	836
3	16	3	**1240029**	-	-	1304424

Table 6.5: Some relevant nonlinear comparison.

Part III

Polynomials techniques for minimum weight problems

Introduction

In this chapter we analyse two algorithms from [Gue09] (also in [GS07], [GOS10], [GOS09]) and [Sim07] (also in [Sim09], [SS07b], [SS07a]). The first one is used to compute the minimum weight of a systematic code (and can be easily extended to compute the distance of a systematic code), the second to compute the nonlinearity of a Boolean function. They are basically the same algorithm, which reduce both problems to the problem of solving a polynomial system of equations over a finite field to find all the codewords with weight less than or equal to a certain quantity. We first generalize the first method to work for any nonlinear code and then make some consideration on the complexity of the algorithm.
We also provide two different and new algorithms to compute the minimum weight of a binary code and the nonlinearity of a B.f. . The first of these algorithms reduces again the two problems to the problem of solving a polynomial system of equations, though defined over the rationals or over big prime fields instead of the finite field $\mathbb{F}_2$, and with a very different structure from the previous one. The second method takes advantage of fast Fourier techniques, yielding an easy analysis of its complexity and turning out to be a very efficient solution to solve the two problems, even compared to known results.

Computing the minimum weight of a code

The computation of the minimum weight and of the minimum distance of a code are necessary in order to establish the error-correction capability of the code.
If C is linear, it is easy to show that the minimum weight coincides with the minimum distance and the Brouwer-Zimmerman minimum weight probabilistic algorithm for linear codes over finite fields [Zim96] can be used (or any of its variations, such as [CC98]).
Algorithms to solve the decoding problem for a random linear code which are faster than brute-force are known, see for example [BJMM12], [Pet10], and [BLP11]. These randomized Las Vegas type algorithms, are known as *Information Set Decoding* algorithms. The most performing one is [BJMM12], which has an asymptotic running time of $2^{0.04934n} \sim 2^{n/20}$.
In the nonlinear case the minimum weight and the minimum distance may be different. For nonlinear codes with large kernel some algorithms are known which perform better than brute force ([PVZ12]), but in general, we are not aware of any efficient algorithm to compute the two parameters. In particular, to compute the minimum weight of a generic binary $(n, 2^k)$-nonlinear code with brute force we need to perform $O(n2^k)$ bit operations and to store $O(n2^k)$ bits.
The main result of this chapter is a deterministic algorithm to compute the minimum weight of any random binary code represented as a set of B.f. in numerical normal form (NNF).
In Section 7.2 and 7.3, we first show that this representation does not have any particular drawback with respect to the classical representation (code as a set of binary vectors).
In Section 7.4 we generalize an algorithm, from [GOS06] and [Gue05] to find all codewords of weight less than t for any nonlinear code (in the previous work the algorithm was designed to work only for systematic codes). Their algorithm reduces the computation of the minimum weight of a nonlinear code to the problem of solving a system of polynomial equations over $\mathbb{F}_q$.
In Section 7.5, for the binary case, we reduce the computation to the problem of solving a polynomial system of equations over $\mathbb{Q}$ or $\mathbb{Z}_p$ with p prime, containing only the "field equations" and one single dense polynomial of which we have to find the

zeros. Then we show how to find such solutions applying fast Fourier techniques. Finally, in Section 7.6 we develop a new method: we show that, using fast Fourier techniques to compute the minimum weight starting from the NNF representation of a binary nonlinear code has a complexity of $O((n/h+k)2^k)$, where n/h is the average number of nonzero monomials of the Boolean functions representing the code. In particular, there are many important cases where our method is faster than brute-force (e.g. in the linear case and in the nonlinear case when the NNF representation of the code is sparse), and cases where it is faster than the Brouwer-Zimmerman method.

7.1 Polynomials and vector weights

Here we introduce some common notation between the two problems we are going to analyze, recalling some definitions and results about the weight of vectors in $\mathbb{F}^n$, taken from [GOS06] and [Gue05].

We denote by $E_q[X]$ the set of field equations, i.e. the following set of polynomials in $\mathbb{F}[X] = \mathbb{F}[x_1, \ldots, x_s]$: $E_q[X] = \{x_1^q - x_1, \ldots, x_s^q - x_s\}$, where $s \geq 1$ is an integer, understood from now on.

Definition 7.1.1. *Let $1 \leq t \leq s$ and $\mathsf{m} \in \mathbb{F}[X]$. We say that m is a* **square free monomial** *of degree t (or a* **simple t-monomial***) if:*

$$\mathsf{m} = x_{h_1} \cdots x_{h_t}, \text{ where } h_1, \ldots, h_t \in \{1, \ldots, s\} \text{ and } h_\ell \neq h_j, \forall \ell \neq j\,,$$

i.e. a monomial in $\mathbb{F}[X]$ such that $\deg_{x_{h_i}}(\mathsf{m}) = 1$ for any $1 \leq i \leq t$. We denote by $\mathcal{M}_{s,t}$ the set of all square free monomials of degree t in $\mathbb{F}[X]$.

Let $t \in \mathbb{N}$, with $1 \leq t \leq s$ and let $I_{s,t} \subset \mathbb{F}[X]$ be the following ideal

$$I_{s,t} = \langle \{\sigma_t, \ldots, \sigma_s\} \cup E_q[X] \rangle\,,$$

where σ_i are the elementary symmetric functions:

$$\begin{aligned}
\sigma_1 &= x_1 + x_2 + \cdots + x_s,\\
\sigma_2 &= x_1x_2 + x_1x_3 + \cdots + x_1x_s + x_2x_3 + \cdots + x_{s-1}x_s,\\
&\cdots\\
\sigma_{s-1} &= x_1x_2x_3\cdots x_{s-2}x_{s-1} + \cdots + x_2x_3\cdots x_{s-1}y_s,\\
\sigma_s &= x_1x_2\cdots x_{s-1}x_s.
\end{aligned}$$

We also denote by $I_{s,s+1}$ the ideal $\langle E_q[X] \rangle$.

For any $1 \leq i \leq s$, let P_i be the set which contains all vectors in $\mathbb{F}^n$ of weight i, $P_i = \{v \in \mathbb{F}^n \mid w(v) = i\}$, and let Q_i be the set which contains all vectors of weight up to i, $Q_i = \sqcup_{0 \leq j \leq i} P_j$.

Theorem 7.1.2. *Let t be an integer such that $1 \leq t \leq s$. Then the vanishing ideal $\mathcal{I}(Q_t)$ of Q_t is*

$$\mathcal{I}(Q_t) = I_{s,t+1}\,,$$

and its reduced Gröbner basis G is

$$\begin{aligned} G &= E_q[X] \cup \mathcal{M}_{s,t}\,, \qquad \text{for } t \geq 2\,,\\ G &= \{x_1, \ldots, x_s\}\,, \qquad \text{for } t = 1\,. \end{aligned}$$

Let $I \subset \mathbb{F}[X]$ be an ideal and let X' be a subset of X. We denote by $I_{X'}$ the elimination ideal of I, i.e. $I_{X'} = I \cap \mathbb{F}[X']$.

Let $\mathbb{F}[Z]$ be a polynomial ring over $\mathbb{F}$. Let $\mathsf{m} \in \mathcal{M}_{s,t}$, $\mathsf{m} = z_{h_1} \cdots z_{h_t}$. For any $W \in (\mathbb{F}[Z])^n$, $W = (W_1, \ldots, W_n)$, we denote by $\mathsf{m}(W)$ the following polynomial in $\mathbb{F}[Z]$:

$$\mathsf{m}(W) = W_{h_1} \cdots W_{h_t}\,.$$

Example 7.1.3. Let $n = s = 3$ and $W = (x_1x_2 + x_3, x_2, x_2x_3) \in (\mathbb{F}[x_1, x_2, x_3])^3$ and $\mathsf{m} = z_1z_3$. Then

$$\mathsf{m}(W) = (x_1x_2 + x_3)(x_2x_3)\,.$$

7.2 Representing a code as a set of Boolean functions

Now we show that any binary $(n, 2^k)$-code C with 2^k codewords can be represented in a unique way as a set of n Boolean functions $f_1, \ldots, f_n : (\mathbb{F}_2)^k \to \mathbb{F}_2$. It is sufficient to consider the matrix whose rows are all the codewords of C. Then we can consider each column i, with $1 \leq i \leq n$, as truth table of a certain Boolean function f_i, for a fixed order of the vectors $v \in (\mathbb{F}_2)^k$.
Then, each f_i can be represented as a square free polynomial in the variables $x_1, \ldots, x_k$ either with coefficients in $\mathbb{F}_2$ (Algebraic Normal Form, see Section 3.1.2) or with integer coefficients (Numerical Normal Form, see Section 3.1.3).
Let us remark that the NNF coefficients require more memory space when stored with respect to the ANF, since they are integers instead of bits, and furthermore, the NNF is usually much denser than the ANF, as shown in Example 3.1.10, Equation 3.2, and proved in Proposition 7.3.1.
We indicate with $f^{(\mathbb{F})}$ a Boolean function represented in algebraic normal form, and with $f^{(\mathbb{Z})}$ a Boolean function represented in numerical normal form.

Definition 7.2.1. *Given a binary $(n, 2^k)$-code C, consider a fixed order of the codewords of C and of the vectors of $(\mathbb{F}_2)^k$. Then consider the matrix M whose rows are the codewords of C. We call the* defining polynomials *of the code C the set $\mathcal{F}_C = \{f_1, \ldots, f_n\}$ of the uniquely determined Boolean functions whose truth table are*

the columns of M. We also indicate with $F = (f_1, \ldots, f_n)$ the vector whose components are the defining polynomials of C. With abuse of notation, we sometimes write

$$\mathcal{F}_C = \{f_1^{(\mathbb{F})}, \ldots, f_n^{(\mathbb{F})}\} \text{ or } \mathcal{F}_C = \{f_1^{(\mathbb{Z})}, \ldots, f_n^{(\mathbb{Z})}\}$$

Notice that F is an encoding function, since $F : (\mathbb{F}_2)^k \to (\mathbb{F}_2)^n$.

Example 7.2.2. Consider the code

$$C = \{c_1, c_2, c_3, c_4\} = \{(0,1,0,0,1), (1,1,1,0,1), (1,0,0,0,0), (1,0,0,1,1)\}.$$

Consider the vectors of $(\mathbb{F}_2)^2$ ordered as follows

$$\mathrm{v}_1 = (0,0), \quad \mathrm{v}_2 = (1,0), \quad \mathrm{v}_3 = (0,1), \quad \mathrm{v}_4 = (1,1).$$

Each column is the truth table of the following Boolean functions represented in ANF

$$\begin{aligned}
f_1^{(\mathbb{F})}(x_1, x_2) &= x_1x_2 + x_1 + x_2 \\
f_2^{(\mathbb{F})}(x_1, x_2) &= x_2 + 1 \\
f_3^{(\mathbb{F})}(x_1, x_2) &= x_1x_2 + x_1 \\
f_4^{(\mathbb{F})}(x_1, x_2) &= x_1x_2 \\
f_5^{(\mathbb{F})}(x_1, x_2) &= x_1x_2 + x_2 + 1,
\end{aligned}$$

whose corresponding NNF is

$$\begin{aligned}
f_1^{(\mathbb{Z})} &= \mathrm{NNF}(f_1) = -x_1x_2 + x_1 + x_2 \\
f_2^{(\mathbb{Z})} &= \mathrm{NNF}(f_2) = -x_2 + 1 \\
f_3^{(\mathbb{Z})} &= \mathrm{NNF}(f_3) = -x_1x_2 + x_1 \\
f_4^{(\mathbb{Z})} &= \mathrm{NNF}(f_4) = x_1x_2 \\
f_5^{(\mathbb{Z})} &= \mathrm{NNF}(f_5) = x_1x_2 - x_2 + 1.
\end{aligned}$$

Thus the defining polynomials of C are

$$\mathcal{F}_C = \{f_1, \ldots, f_n\},$$

where each f_i can be represented as a truth table, as polynomials in algebraic or numerical normal form.
Furthermore we have that for each $1 \leq i \leq 4$

$$c_i = F(v_i) = (f_1(v_i), f_2(v_i), f_3(v_i), f_4(v_i)).$$

7.2.1 Memory cost of representing a code

Let us call *vectorial* the representation of a code as a list of vectors over $\mathbb{F}_2$, and *Boolean* the representation of the same code as a list of Boolean functions.
For a random code, in terms of memory cost, the two representations are equivalent. In the vectorial representation we need to store all the components of each codeword, which are n times 2^k codewords. In the Boolean representation we need to store the 2^k coefficients of the n defining polynomials. In both cases we need a memory space of order $O(n2^k)$.
If the code C is linear it can be represented with a binary generator matrix of size $k \times n$. In this case the defining polynomials are linear Boolean functions, i.e. any is of the form

$$\sum_{i=1}^{k} \lambda_i x_i, \lambda_i \in \mathbb{F}_2 \,,$$

which means that to represent them it is sufficient to store kn elements of $\mathbb{F}_2$, yielding again an equivalent representation.
As shown in [PVZ12], if C is a binary code of length n with kernel K of dimension d_K and t coset leaders given by the set $S = \{c_1, \ldots, c_t\}$, we can represent it as the kernel K plus the coset leaders S (see Example 2.2.3). Since the kernel needs a memory space of order $O(nk)$, then the kernel plus the t coset leaders takes up a memory space of order $O(n(k+t))$. When C is linear then $C = \ker(C)$, so the generator matrix is used to represent C. On the other hand, when $t+1 = |C|$, then representing the code as the kernel plus the coset leaders requires a memory of $O(n|C|) = O(n2^k)$ (since we are supposing the code has 2^k codewords. In the latter case, a Boolean representation could be more convenient, as shown in the following example.

Example 7.2.3. Consider the code

$$\begin{aligned} C = \{&(0,0,0,0),(1,0,0,1),(0,1,0,0),(1,1,0,1),\\ &(0,0,1,1),(1,0,1,0),(0,1,1,1),(1,1,1,1)\}\,. \end{aligned}$$

We have that $\ker(C) = \{(0,0,0,0\}$, thus we have 8 coset leaders. On the other hand, the defining polynomials of C are

$$\mathcal{F}_C = \{x_1, x_2, x_3, x_1x_2x_3 + x_1 + x_3\}$$

which is a much more compact representation.
Unfortunately the code in this example has distance 1.

Another situation in which a Boolean representation is more convenient is the case where the dimension k of the code is much less than the length n, i.e. when certain

components have to be repeated. As shown in [Gue09] (see Appendix), many optimal codes have this form.

Example 7.2.4. The code

$$C = \{(0,0,0,0,0,0,0,0,0,0,0,0), (1,1,1,1,1,1,1,1,0,0,0,0)$$
$$(1,1,1,1,0,0,0,0,1,1,1,1), (0,0,0,0,1,1,1,1,1,1,1,1)\}$$

is linear, with distance 8, dimension 2 and is optimal since it reaches the Plotkin bound. Its generator matrix is

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

The same code can be represented with the 3 Boolean polynomials

$$x_1 + x_2, x_1, x_2\,,$$

each repeated 4 times.
Instead, the code

$$C = \{(0,0,0,0,0,0,0,0,0,0,0,0), (1,1,1,1,1,1,1,1,0,0,0,0)$$
$$(1,1,1,1,0,0,0,0,1,1,1,0), (0,0,0,0,1,1,1,1,1,1,1,1)\}$$

is nonlinear, with distance 7, dimension 2 and has kernel

$$K = (0,0,0,0,0,0,0,0,0,0,0,0)$$

with 4 coset leaders.
The same code can be represented with the 4 Boolean polynomials

$$x_1 + x_2, x_1, x_2, x_1x_2\,.$$

where the first two are repeated 4 times, the third 3 times, and the last only once.

Example 7.2.5. We show now a larger example.
Consider the binary $(16, 2^4)$-nonlinear code

$$C = \{(0,0,0,1,0,1,0,0,1,0,1,1,0,0,1,0), (1,0,0,1,0,1,1,0,1,0,1,1,1,0,1,1),$$
$$(1,0,0,0,0,1,0,0,1,1,1,0,1,0,0,0), (1,0,0,0,0,1,1,0,1,0,1,0,1,0,0,1),$$
$$(0,1,1,1,0,1,0,0,0,0,1,1,0,0,1,0), (1,1,1,1,0,1,1,1,0,0,1,1,1,0,1,1),$$
$$(1,0,1,1,0,0,0,0,0,1,1,0,1,1,0,0), (1,0,0,1,1,0,1,0,0,0,1,0,1,1,0,0),$$
$$(0,0,0,1,1,1,0,0,1,0,0,1,0,0,1,1), (1,0,0,1,1,1,0,0,1,1,0,1,0,1,1,1),$$
$$(1,0,0,0,1,1,0,0,1,1,0,0,1,0,0,1), (1,0,0,0,1,1,0,0,1,0,0,0,0,1,0,1),$$
$$(0,0,1,1,0,0,0,0,0,0,0,1,0,0,1,1), (0,0,1,0,0,0,1,1,1,1,0,0,0,0,1,1),$$
$$(1,1,0,1,0,1,0,0,0,1,1,0,1,1,1,1), (0,1,1,0,0,1,1,0,0,0,0,1,0,1,0,0)\}\,.$$

This code has average weight 7.5 and 120 ones over 256 components. Its minimum distance is 3 and its minimum weight is 5.
The same code represented as a set of B.f. 's in NNF is

$$\begin{aligned} F = \{ & -x_1x_2 - x_1x_3x_4 + x_1 + x_2, & 2x_2x_3x_4 - x_2x_3 - x_3x_4 + x_3, \\ & 2x_1x_2x_3x_4 - x_1x_2x_3 - x_2x_3x_4 + x_3, & -x_1x_3x_4 + x_2x_3 - x_2 + 1, \\ & -x_1x_2x_3x_4 + x_1x_2x_3 - x_3x_4 + x_4, & 2x_2x_3x_4 - x_2x_3 - x_3x_4 + 1, \\ & x_1x_3x_4 - x_1x_4 + x_1, & -x_1x_2x_3 + x_1x_3, \\ & -x_1x_2x_3x_4 + x_1x_3x_4 - x_3 + 1, & -x_1x_2x_4 - x_1x_2 + x_1x_4 + x_2, \\ & -x_1x_2x_3x_4 + x_2x_3x_4 - x_4 + 1, & 2x_1x_2x_3x_4 - x_1x_3x_4 - x_2 + 1, \\ & -x_1x_2 - x_1x_4 + x_1 + x_2, & -x_1x_3x_4 + x_1x_4 + x_2x_3, \\ & -x_1x_2x_3x_4 + x_2x_3x_4 - x_2 + 1, & -x_1x_2x_3 - x_1x_4 + x_1 + x_4 \}, \end{aligned}$$

which is a set of $n = 16$ B.f. with $2, 3$ or 4 coefficients, for a total of 16 coefficients varying in the set $\{-1, 1, 2\}$.

It is worth noticing that a linear structure of a nonlinear binary code can be found over a different ring. For example there are binary codes which have a $\mathbb{Z}_4$-linear or $\mathbb{Z}_2\mathbb{Z}_4$-linear structure and, therefore, they can also be compactly represented using quaternary generator matrix, as shown in [HKC+94] and [BFCP+10].

7.3 Number of coefficients of the NNF

In order to prove that representing a code with practical parameters and using NNF B.f. 's is as convenient as the usual representation of the code, in this section we want to study the distribution of the number of nonzero coefficients of a B.f. represented in NNF, i.e., once the number of variables k is fixed we want to know how many B.f. 's have only 1 nonzero coefficient, how many have 2, and so on.
We are also interested in finding a relation between this distribution and the distribution of the number of nonzero coefficients of a B.f. represented in ANF.
In Table 7.1 we report the distribution of the nonzero coefficients of B.f. 's represented in ANF and NNF with $k = 1, 2, 3, 4$ variables. As one may expect, the ANF follows a binomial distribution. This means that choosing a random B.f. its ANF is likely to have half of the coefficients equal to 0 and half equal to 1. This does not happen for the NNF, eventhough for k small the two distributions are close. This means that, when k is small, a random binary $(n, 2^k)$-nonlinear code can be represented with a set of B.f. 's in NNF with half of the coefficients equal to 0 with high probability, while sparse NNF representations are more rare as k grows.

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A: 1	1	**2**	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-
N: 1	1	**2**	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-
A: 2	1	4	**6**	4	1	-	-	-	-	-	-	-	-	-	-	-	-
N: 2	1	4	**5**	4	2	-	-	-	-	-	-	-	-	-	-	-	-
A: 3	1	8	28	56	**70**	56	28	8	1	-	-	-	-	-	-	-	-
N: 3	1	8	19	42	**59**	50	34	28	15	-	-	-	-	-	-	-	-
A: 4	1	16	120	560	1820	4368	8008	11440	**12870**	11440	8008	4368	1820	560	120	16	1
N: 4	1	16	65	304	840	1768	3250	5458	8077	**9986**	9819	7948	5954	4458	3193	2830	1569

Table 7.1: Distribution of the nonzero coefficients in the ANF and NNF.

Proposition 7.3.1. *Let f be a B.f. in k variables. Let $f^{(\mathbb{F})}$ and $f^{(\mathbb{Z})}$ be respectively the ANF and the NNF of f. Then if $f^{(\mathbb{F})}$ is a polynomials with $r \leq 2^k$ non zero coefficients, then $f^{(\mathbb{Z})}$ is a polynomial with no more than $\min\{2^k, 2^r - 1\}$ nonzero coefficients.*

Proof. When computing the NNF from the ANF we have again the r initial terms of the ANF, plus $\binom{r}{2}$ terms which are all possible double product of the r initial terms, plus, in general, $\binom{r}{i}$ terms which are all possible i-product of the r initial terms, for each $i \in \{1, \ldots, r\}$. Thus we will have

$$\sum_{i=1}^{r} \binom{r}{i} = 2^r - 1 \tag{7.1}$$

terms to be summed together. If no sum of similar monomials becomes zero than we have $2^r - 1$ nonzero terms. □

By Proposition 7.3.1, if we want a NNF with no more than s terms then we have to choose the ANF with no more than $r = \log_2(s+1)$ terms.

Proposition 7.3.2. *Let f be a linear B.f. in k variables. Let $f^{(\mathbb{F})}$ and $f^{(\mathbb{Z})}$ be respectively the ANF and the NNF of f. We have*

$$f^{(\mathbb{F})} = x_{i_1} + \ldots + x_{i_r}\,,$$

for $r \leq k$. Then $f^{(\mathbb{Z})}$ is a polynomial with exactly $2^r - 1$ nonzero coefficients.
In particular,

$$f^{(\mathbb{Z})} = \sum_{\substack{v \in (\mathbb{F}_2)^r \\ v=(v_1,\ldots,v_h)\neq 0}} (-1)^{\mathrm{w}(v)-1} \binom{r}{\mathrm{w}(v)-1} x_{i_1}^{v_1} \cdots x_{i_r}^{v_r}\,.$$

Proof. Directly from Proposition 3.1.9. □

Proposition 7.3.2 says that linear B.f. 's are always denser when represented in NNF with respect to the ANF representation.

7.4 Finding the codewords with weight $< t$

We are now ready to present a general method to find all the codewords of weight less than t, for some $t \leq n$, using polynomial system solving techniques.

We can define an ideal whose variety is the set of points in $(\mathbb{F}_2)^k$ such that evaluated in the defining polynomials $f_i, i = 1, \ldots, n$ of C give the codewords of C which have weight less than t.

Definition 7.4.1. *We call* **weights ideal** *over* $\mathbb{F}_2$ *the ideal*

$$\mathcal{W}_C^t = \langle E_2[X] \bigcup \{\mathrm{m}(f_1^{(\mathbb{F})}(X), \ldots, f_n^{(\mathbb{F})}(X)) \mid \mathrm{m} \in \mathcal{M}_{n,t}\} \rangle, \mathcal{W}_C^t \in \mathbb{F}_2[X],$$

where $E_2[X] = \{x_1^2 - x_1, \ldots, x_k^2 - x_k\}$ *are the field equations.*

The name comes from the following lemma:

Lemma 7.4.2. *Let* C *be a binary* $(n, 2^k)$*-code. Let* $t \in \mathbb{N}$ *such that* $1 \leq t \leq n$*. Then*

$$\mathcal{V}(\mathcal{W}_C^t) \neq \emptyset \iff \exists c \in C \text{ s.t. } \mathrm{w}(c) \leq t - 1$$

Clearly this variety is empty if there is no codeword of weight less than t.

To find the minimum weight of a binary $(n, 2^k)$-code, supposing the defining polynomials are given in ANF we can use the following algorithm:

Algorithm 3 Basic algorithm to compute the minimum weight of a code C

Input: a binary $(n, 2^k)$-code C defined by $f_1^{(\mathbb{F})}, \ldots, f_n^{(\mathbb{F})}$
Output: the minimum weight of C

1: $j \leftarrow 1$
2: **while** $\mathcal{V}(\mathcal{W}_C^j) = \emptyset$ **do**
3: $\quad j \leftarrow j + 1$
4: **return** $j - 1$

We will see in the upcoming chapters how this algorithm can be used in particular to compute the nonlinearity of a B.f. . The same algorithm could also be used to compute the distance of a code C, by computing the minimum weight of the code C' composed by the difference of all possible pair of codewords of the initial code C. The complexity of computing $\mathcal{V}(\mathcal{W}_C^t)$ is estimated in the following theorem.

Theorem 7.4.3. *To find the variety of the ideal* $\mathcal{W}_C^t$ *is equivalent to solve a multivariate system of* $\binom{n}{t}$ *polynomials of degree* $\leq k$ *in* k *variables.*

Proof. Recall that $\mathcal{W}_C^t = \langle E_2[X] \bigcup \{\mathrm{m}(f_1(X), \ldots, f_n(X)) \mid \mathrm{m} \in \mathcal{M}_{n,t}\} \rangle$., thus the system is composed by the k field equations (which we do not consider as these equations are only saying that the solutions will be binary vectors) plus $|\mathcal{M}_{n,t}| = \binom{n}{t}$ polynomials in the variables $X = x_1, \ldots, x_k$. Since this polynomials are in ANF, their degree is at most k. □

The construction of the system $\mathcal{W}_C^t$ dominates the entire cost of Algorithm 3. This is because $\binom{n}{t}$ monomials have to be evaluated. Clearly one can save computations from previous while cycles. Furthermore, all multiplications of the polynomials $f_i^{(\mathbb{F})}$ should be done considering their normal forms with respect to the ideal being constructed. This saves time when 1 is in the ideal, but when the ideal is not trivial, $\binom{n}{t}$ operations *must* be done to construct it, and the resulting system has to be solved.
To brute force the system requires to evaluate 2^k points.
Even for small values of t with respect to n ($t \ll n$) we have

$$\begin{aligned}\binom{n}{t} &= \frac{n(n-1)\ldots(n-t+1)}{t!} \approx \\ &\approx \frac{(n-t/2)^t}{t^t e^{-t}\sqrt{2\pi t}} = \frac{(n/t-0.5)^t e^t}{\sqrt{2\pi t}} = \\ &= e^{t(\ln(n/t-0.5)+1)-\ln\sqrt{2\pi t}}\end{aligned}$$

and if $t > k$, then $\binom{n}{t} \approx e^{t(\ln(n/t-0.5)+1)-\ln\sqrt{2\pi t}} > 2^k$.
A Gröbner basis of the ideal $\mathcal{W}_C^t$ is a much simpler description of $\mathcal{W}_C^t$. A way to solve this system is thus to find a Gröbner basis with one of the techniques outlined in Section 1.3. We are currently not aware though of which algorithm is best.
Furthermore not all the monomials in $\mathcal{M}_{n,t}$ may be useful to solve the system (as shown when computing the nonlinearity of a Boolean function, see Table 8.3 and 8.4), though it is still unclear which monomials should be chosen to determine it in a few steps.
As a last comment, we point out that the zeros of the ideal $\mathcal{W}_C^t$ reveal those vectors which evaluated in the defining polynomials give the codewords of C of weight less than t. To compute the minimum weight we do not need to actually find such zeros, but we only have to determine if any zero exists, which is somehow a simpler problem, as shown in Section 1.4.2.

7.5 Finding the codewords with weight exactly t

It is possible to construct a polynomial with integer coefficients whose evaluations in $\{0,1\}^k \subseteq \mathbb{Z}^k$ are the weights of the codewords of the code C.

Definition 7.5.1. *Let $X = x_1, \ldots, x_k$, and $X^2 - X = x_1^2 - x_1, \ldots, x_k^2 - x_k$. We call the* **weight polynomial** *of the code C the polynomial*

$$\mathfrak{w}_C(X) = \sum_{i=1}^{n} f_i^{(\mathbb{Z})}(X) \in \mathbb{Z}[X]/\langle X^2 - X \rangle \,,$$

where the $f_i^{(\mathbb{Z})}$'s are the defining polynomials of the code C in NNF.

Theorem 7.5.2. *Let $v \in \{0,1\}^k \subseteq \mathbb{Z}^k$. Then there exist a codeword $c \in C$ such that* $\mathrm{w}(c) = \mathfrak{w}_C(v)$.

Proof. It is sufficient to note that a codeword $c \in C$ is such that $c = (f_1^{(\mathbb{Z})}(P), \ldots, f_n^{(\mathbb{Z})}(P))$ for some $P \in \{0,1\}^k$, and that the sum of all $f_i^{(\mathbb{Z})}$ is over the integers. □

Example 7.5.3. Continuing from Example 7.2.2 we have that

$$\mathfrak{w}_C(x_1, x_2) = f_1^{(\mathbb{Z})} + f_2^{(\mathbb{Z})} + f_3^{(\mathbb{Z})} + f_4^{(\mathbb{Z})} + f_5^{(\mathbb{Z})} = 2x_1 - x_2 + 2 \,.$$

Evaluating $\mathfrak{w}_C$ in $\mathrm{v}_1, \mathrm{v}_2, \mathrm{v}_3, \mathrm{v}_4$ we get

$$\mathfrak{w}_C(\mathrm{v}_1) = 2, \quad \mathfrak{w}_C(\mathrm{v}_2) = 4, \quad \mathfrak{w}_C(\mathrm{v}_3) = 1, \quad \mathfrak{w}_C(\mathrm{v}_4) = 3 \,,$$

which are, respectively, the weights of the codewords

$$(0,1,0,0,1), (1,1,1,0,1), (1,0,0,0,0), (1,0,0,1,1) \,.$$

We can define an ideal whose variety is the set of points in $\{0,1\}^k$ such that evaluated in the sum of the defining polynomials $f_i^{(\mathbb{Z})}, i = 1, \ldots, n$ of C give the codewords of C which have weight exactly t.

Definition 7.5.4. *We call* **weights ideal** *over $\mathbb{Z}$ the ideal*

$$\overline{\mathcal{W}}_C^t = \langle \overline{E}_2[X] \bigcup \{\mathfrak{w}_C(X) - t\} \rangle \,, \overline{\mathcal{W}}_C^t \in \mathbb{Q}[X] \,,$$

where $\overline{E}_2[X] = \{x_1^2 - x_1, \ldots, x_k^2 - x_k\}$.

Lemma 7.5.5. *Let C be a binary $(n, 2^k)$-code. Let $t \in \mathbb{N}$ such that $1 \leq t \leq n$. Then*

$$\mathcal{V}(\overline{\mathcal{W}}_C^t) \neq \emptyset \iff \exists c \in C \text{ s.t. } \mathrm{w}(c) = t$$

Clearly this variety is empty if there is no codeword of weight t.
Algorithm 4 is a method to find the minimum weight of a binary $(n, 2^k)$-code, supposing the defining polynomials are given in ANF.

Algorithm 4 To find the minimum weight of a binary code C

Input: $f_1^{(\mathbb{F})}, \ldots, f_n^{(\mathbb{F})}$

Output: the minimum weight of C

1: $f_i^{(\mathbb{Z})} \leftarrow \mathrm{NNF}(f_i^{(\mathbb{F})})$, for each $i = 1, \ldots, n$
2: $j \leftarrow 0$
3: **while** $\mathcal{V}(\bar{\mathcal{W}}_C^j) = \emptyset$ **do**
4: $\quad j \leftarrow j + 1$
5: **return** j

If we know that the code contains the 0 codeword and we are interested in finding the minimum weight different from 0, then j must be initialized to 1.
Algorithm 4 can be modified to eliminate the while cycle. Instead of checking if a solution of the system

$$\begin{cases} x_1^2 - x_1 = 0 \\ \ldots \\ x_k^2 - x_k = 0 \\ \mathfrak{w}_C(x_1, \ldots, x_k) - j = 0 \end{cases} \tag{7.2}$$

exists in the affine algebra $\mathbb{Q}/\langle x_1^2 - x_1, \ldots, x_k^2 - x_k \rangle$ for each $j \in \{1, \ldots, n\}$, we can add the variable t to the system

$$\begin{cases} x_1^2 - x_1 = 0 \\ \ldots \\ x_k^2 - x_k = 0 \\ \mathfrak{w}_C(x_1, \ldots, x_k) - t = 0 \end{cases} \tag{7.3}$$

and solve it in $\mathbb{Q}[t]/\langle x_1^2 - x_1, \ldots, x_k^2 - x_k \rangle$, with respect to lexicographical monomial ordering, to find as a solution a polynomial $\mathfrak{t}(t)$, whose solutions are integers, representing the weights of the codewords of C. We are interested in the smallest solution of $\mathfrak{t}(t)$.
We did not investigate further which of the two solutions is best.

Example 7.5.6. Consider the code

$$\begin{aligned} C = \{&(0,1,0,0,1) \\ &(1,1,0,0,1) \\ &(1,0,1,0,0) \\ &(1,0,0,1,1)\}. \end{aligned}$$

The defining polynomials of C in ANF and NNF are respectively

$$\begin{aligned} f_1^{(\mathbb{F})} &= x_1x_2 + x_1 + x_2 & f_1^{(\mathbb{Z})} &= -x_1x_2 + x_1 + x_2 \\ f_2^{(\mathbb{F})} &= x_2 + 1 & f_2^{(\mathbb{Z})} &= -x_2 + 1 \\ f_3^{(\mathbb{F})} &= x_1x_2 + x_2 & f_3^{(\mathbb{Z})} &= -x_1x_2 + x_2 \\ f_4^{(\mathbb{F})} &= x_1x_2 & f_4^{(\mathbb{Z})} &= x_1x_2 \\ f_5^{(\mathbb{F})} &= x_1x_2 + x_2 + 1 & f_5^{(\mathbb{Z})} &= x_1x_2 - x_2 + 1\,. \end{aligned}$$

The weight polynomial of C is

$$\mathfrak{w}_C(x_1, x_2) = x_1 + 2\,.$$

If we want to find all the codewords $c \in C$ such that $\mathrm{w}(c) = 3$, we can compute a Gröbner basis of the ideal $\overline{\mathcal{W}}_C^3$, which is

$$\begin{aligned} GB(\{\mathfrak{w}_C(x_1, x_2) - 3, x_1^2 - x_1, x_2^2 - x_2\}) &= GB(\{x_1 - 1, x_1^2 - x_1, x_2^2 - x_2\}) \\ &= \{x_1 - 1, x_2^2 - x_2\}\,. \end{aligned}$$

Its variety is $\mathcal{V}(\overline{\mathcal{W}}_C^3) = \{(1,0),(1,1)\}$.
Then we consider the points $p \in \mathcal{V}(\overline{\mathcal{W}}_C^3)$ and compute

$$c = F(p) = (f_1(p), f_2(p), f_3(p), f_4(p), f_5(p))\,,$$

i.e.:

$$\begin{aligned} \mathcal{V}(\overline{\mathcal{W}}_C^3) &= \{(1,0),(1,1)\} \\ F((1,0)) &= (1,1,0,0,1) \\ F((1,1)) &= (1,0,0,1,1) \end{aligned}$$

Instead of computing the Gröbner basis, which is sometimes a heavy task, we can consider the evaluation vector of $\mathfrak{w}_C$ over the set $\{0,1\}^2$, and consider only those pairs whose evaluation is 3,

$$\begin{aligned} \mathfrak{w}_C((0,0)) &= 2 \\ \mathfrak{w}_C((1,0)) &= 3 \\ \mathfrak{w}_C((0,1)) &= 2 \\ \mathfrak{w}_C((1,1)) &= 3\,, \end{aligned}$$

Which are $(1,0),(1,1)$.

If we wanted to find the minimum weight of C, we could either use Algorithm 4 or to solve the system

$$\begin{cases} x_1^2 - x_1 = 0 \\ x_2^2 - x_2 = 0 \\ \mathfrak{w}(x_1, x_2) = x_1 + 2 - t = 0 \end{cases}$$

with respect to lexicographical order and with respect to the variable t.
We find the solution $\mathfrak{t}(t) = t^2 - 5t + 6$, whose roots are 2 and 3, implying the minimum weight of the code is 2.

As shown in the last example, once we have the weight polynomial $\mathfrak{w}_C$ of the code C, not only we can find the minimum weight of C, but we also find which are the codewords having certain weights by looking at its evaluation vector over the set $\{0,1\}^k$. As we will see in Section 7.6.3, computing this evaluation has a cost of $O(k2^k)$. The complexity maintains the same order if the number of terms of each defining polynomial in NNF is on average $O(\frac{k}{n}2^k)$.

To summarize, we state the algorithm to find the evaluation vector of the weight polynomial $\mathfrak{w}_C$ of a binary $(n, 2^k)$-code C given as a list of 2^k codewords (and thus also the minimum weight of C). We indicate with $C_{i,j}$ the j-th component of the i-th word of C, with $1 \leq j \leq n$ and $1 \leq i \leq 2^k$.

Algorithm 5 To find the evaluation vector of $\mathfrak{w}_C$ from the list of codewords of C.
Input: $c_1, \ldots, c_{2^k} \in C$
Output: the evaluation vector $\underline{\mathfrak{w}}_C$ of $\mathfrak{w}_C$
1: $f_j^{(\mathbb{Z})} \leftarrow$ NNF of the binary vector $(C_{1,j}, \ldots, C_{2^k,j})$ for $1 \leq j \leq n$
2: $\mathfrak{w}_C \leftarrow f_1^{(\mathbb{Z})} + \ldots + f_n^{(\mathbb{Z})}$
3: $\underline{\mathfrak{w}}_C \leftarrow$ Evaluation of $\mathfrak{w}_C$ over $\{0,1\}^k$
4: **return** $\underline{\mathfrak{w}}_C$

7.6 Complexity considerations

First of all let us notice that given a binary $(n, 2^k)$-code as a list of 2^k codewords, to find all the codewords of weight t using brute force requires $n2^k$ bit operations, since we have to check each component of each codeword of C.
We now analyze the complexity of Steps 1, 2, and 3 of Algorithm 5. Then we compare our method to compute the minimum weight of a binary code with brute force and, in the linear case, with the Brouwer-Zimmerman method ([Zim96]).

7.6.1 From list of codewords to defining polynomials in NNF

Proposition 7.6.1. *The overall worst-case complexity of determining the coefficients of the n defining polynomials in NNF of the code C given as a list of vectors is $O(nk2^k)$.*

Proof. We want to find the NNF of the Boolean function whose truth table is given by a column of the binary matrix

$$C = \begin{pmatrix} C_{1,1} & C_{1,2} & \dots & C_{1,j} & \dots & C_{1,n} \\ \vdots & \vdots & & \vdots & & \vdots \\ C_{i,1} & C_{i,2} & \dots & C_{i,j} & \dots & C_{i,n} \\ \vdots & \vdots & & \vdots & & \vdots \\ C_{2^k,1} & C_{2^k,2} & \dots & C_{2^k,j} & \dots & C_{2^k,n} \end{pmatrix}$$

whose rows are the codewords of the code C.
In [CG99] (Proposition 2) it is shown that to compute the NNF of a Boolean function in k variables given its truth table requires $k2^{k-1}$ integer subtractions. Since we have to compute the NNF for n columns the overall complexity is $O(nk2^k)$. □

A similar result holds if we want to determine the coefficients of the n defining polynomials in ANF.

7.6.2 From defining polynomials to weight polynomial

Proposition 7.6.2. *The overall worst-case complexity of summing together all the defining polynomials in NNF is $O(n2^k)$ integer sums.*

Proof. Each monomial m in a defining polynomial is square free, and since $m \in \mathbb{Z}[x_1, \dots, x_k]$, then a defining polynomial can have no more than 2^k monomials. Since the defining polynomials are n, the proposition follows. □

Remark 7.6.3. Clearly, the computational complexity of this steps decreases if the defining polynomials are sparse.

7.6.3 Evaluation of the weight polynomial

Algorithm 6 describes the fast Möbius transform to compute the evaluation vector of a Boolean function f in NNF in k variables.
We use the following notation: the coefficient c_{2^k} is the coefficient of the greatest monomial, i.e. of $x_1 \cdots x_k$, c_{2^k-1} the coefficient of the second greatest monomial, and so on until c_1, which is the costant term. We provide Example 7.6.4 to clarify

notation.
Notice that the sum in Step 6 is over the integer. If it was a sum in $\mathbb{F}_2$ then we would obtain the truth table of f.

Algorithm 6 Fast Möbius transform for fast integer polynomial evaluation.

Input: vector of coefficients $c = (c_1, \ldots, c_{2^k})$
Output: evaluation vector $e = (e_1, \ldots, e_{2^k})$

1: $e \leftarrow c$
2: **for** $i = 0, \ldots, k$ **do**
3: $\quad b \leftarrow 0$
4: $\quad$ **repeat**
5: $\quad\quad$ **for** $x = b, \ldots, b + 2^i - 1$ **do**
6: $\quad\quad\quad e_{x+1+2^i} \leftarrow e_{x+1} + e_{x+1+2^i}$
7: $\quad\quad b \leftarrow b + 2^{i+1}$
8: $\quad$ **until** $b = 2^k$
9: **return** e

Example 7.6.4. Consider $k = 3$ and lexicographical ordering with $x_1 \succ x_2 \succ x_3$. Let $f = 8x_1x_2x_3 + 3x_1 + 2$. Then

$$c = (c_1, \ldots, c_8) = (2, 0, 0, 0, 3, 0, 0, 8)$$
$$e = (e_1, \ldots, e_8) = (2, 2, 2, 2, 5, 5, 5, 13)\,.$$

where the vector e has been obtained following the scheme in Figure 7.1.

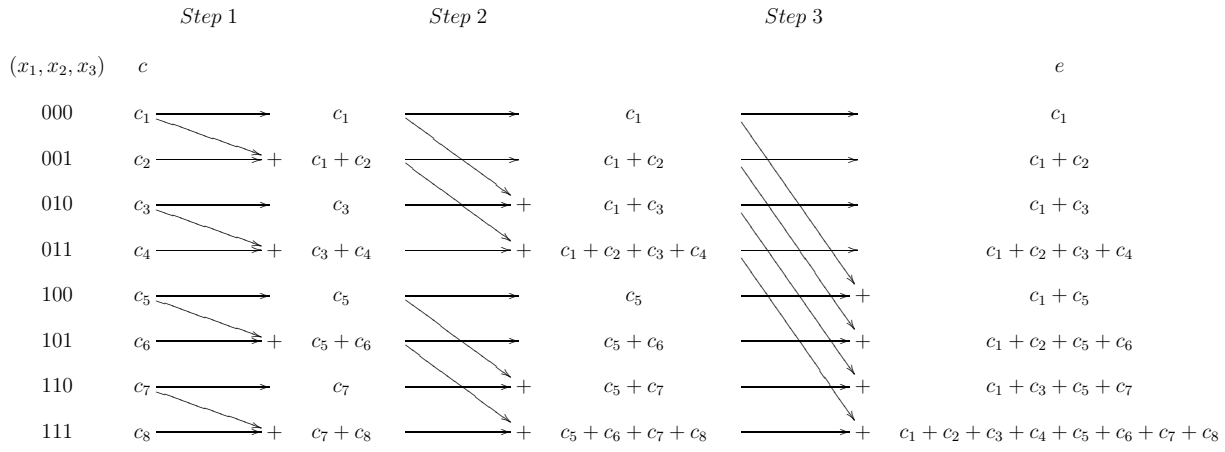

Figure 7.1: Butterfly scheme describing the fast Möbius transform.

Proposition 7.6.5. *Evaluating the weight polynomial over the set* $\{0, 1\}^k$ *has a computational cost of* $O(k2^k)$ *integer operations.*

Proof. This is the cost of Algorithm 6, i.e. $k2^{k-1}$ integer sums. □

7.6.4 Comparison with brute-force method

Theorem 7.6.6. *Let h be a positive integer. If the code C is given as a set of B.f. 's whose NNF have on average $2^k/h$ coefficients different from 0, then computing the minimum weight of C requires at most*

$$\left(\frac{n}{h}+k\right)2^k$$

integer sums.

Proof. By Proposition 7.6.5 computing the evaluation vector of the weight polynomial $\mathfrak{w}_C$ requires $k2^{k-1}$ integer sums using the fast Möbius transform. To compute the weight polynomial we need to sum the n defining polynomials $f_i^{(\mathbb{Z})}, i = 1, \ldots, n$, in NNF. If each of these polynomials has on average $2^k/h$ coefficients then the complexity of computing $\mathfrak{w}_C$ requires $O(n\frac{2^k}{h})$ integer sums. So the final complexity is at most $(n/h)2^k + k2^{k-1}$.

□

The codewords of a random code have on average half 0's and half 1's, thus, if we consider that 0's do not count when summing the defining polynomials, then the computation will require $n2^{k-1}$ integers sums.

Our method is more efficient than brute force when $n/h + k < n$. This is very likely to happen for a random code.

Notice also that if the sets of nonzero monomials of two polynomials in NNF are disjoint, then the sum of the two polynomials is simply their concatenation. So, if the defining polynomials of a code are "disjoint", then the cost of computing the weight polynomial is $O(1)$, and the final cost of finding the minimum weight becomes the cost of computing the evaluation of $\mathfrak{w}_C$, i.e. $O(k2^{k-1})$.

Fact 7.6.7 shows that, for $n \gg k$, when the code is linear our method to compute the minimum nonzero weight (i.e. the distance of the code) given the set of the defining polynomials in NNF is more efficient than the classical method which uses brute force, given the list of the codewords of the code.

Fact 7.6.7 (Comparison with brute force when $n \sim 2^k$)**.** *Consider a random binary (n, k)-linear code C such that $n \sim 2^k$. Then computing the minimum weight of C*

1. *given the list of its codewords and using brute force requires*

$$O(2^{2k})$$

integer operations.

2. *given the list of the defining polynomials in NNF and finding the minimum of* $\mathfrak{w}_C$ *requires*

$$O(2^{\frac{3}{2}k})$$

integer operations.

Proof. The complexity of finding the minimum weight of C in case 1 is $O(n2^k) = O(2^{2k})$.
The complexity of finding the minimum weight of C in case 2 is $O((n/h + k)2^k)$ (by Theorem 7.6.6), where n/h is the average number of nonzero coefficients of the NNF. If the linear code C is random, then so are the random linear defining polynomials. A random linear function in k variables has on average $k/2$ nonzero coefficient in ANF and thus $2^{k/2} - 1$ nonzero coefficients in NNF by proposition 7.3.2, i.e. $n/h \sim 2^{k/2}$, and

$$O((n/h + k)2^k) = O((2^{k/2} + k)2^k) = O(2^{\frac{3}{2}k}) .$$

□

Remark 7.6.8. If the code is non linear and the ANF has on average $k/2$ coefficients then $n/h \leq 2^{k/2}$ and our method is even faster.

In Table 7.2 we show the coefficient of growth of the complexity of our method in three different cases. The first line shows the coefficient of growth of the brute force method applied to a linear code. The second line shows the coefficient of growth of our method applied to a linear code. In the third line our method is applied to a nonlinear code whose representation in ANF is sparse, and in the last line nonlinear codes with dense representaion in ANF are considered.
For the comparison we choose for each k, 10 random $(2^k, k)$-codes and 10 random $(2^{k+1}, k+1)$-codes and compute the average times t_1, t_2 to compute the minimum weight in each case. Then we report the number $\log_2(t_1/t_2)$.

k	$8-9$	$9-10$	$10-11$	$11-12$
Brute-force Linear ANF	1.93	1.98	2.00	1.99
Linear ANF	1.32	1.38	1.53	1.61
Sparse Nonlinear ANF	0.89	1.12	1.32	1.38
Dense Nonlinear ANF	2.09	2.03	2.04	2.08

Table 7.2:

7.6.5 Comparison with Brouwer-Zimmerman method for linear codes

In the linear case the defining polynomials of a code C clearly have a sparse ANF. This is not necessarily so with the NNF. Consider the following example.

Example 7.6.9. Suppose C is a $(10, 2^4)_2$-linear code, i.e. $k = 4, n = 10$. Then it has 10 defining polinomials $f_1, \ldots, f_{10} \in \mathbb{F}[x_1, x_2, x_3, x_4]$. Consider one of them, for example f_5. If $f_5^{(\mathbb{F})} = x_4 + x_3 + x_2 + x_1$ is the ANF of f_5 then the NNF is

$$\begin{aligned} f_5^{(\mathbb{Z})} = &- 8x_1x_2x_3x_4 + 4x_1x_2x_3 + 4x_1x_2x_4 - 2x_1x_2 \\ &+ 4x_1x_3x_4 - 2x_1x_3 - 2x_1x_4 + x_1 + 4x_2x_3x_4 \\ &- 2x_2x_3 - 2x_2x_4 + x_2 - 2x_3x_4 + x_3 + x_4 \end{aligned}$$

i.e. the NNF has all the coefficients different from 0.
Consider now f_6. If $f_6^{(\mathbb{F})} = x_2 + x_4$ is the ANF of f_6 then the NNF is

$$f_6^{(\mathbb{Z})} = -2x_4x_2 + x_4 + x_2\,.$$

As shown in Example 7.6.9 if a defining polynomial in $\mathbb{F}[x_1, \ldots, x_k]$ is linear and with less than k variables, than many coefficients of the NNF are 0, precisely, the coefficients of the monomials containing the missing variable in the ANF. In this case the computation of the minimum weight of C (and thus of the distance of C, since the code is linear) is faster than brute force.
In Table 7.3 we compare the time t_1 needed to compute the minimum weight w of a linear code given as list of codewords with the MAGMA command

`MinimumWeight(C:Method:="Zimmerman")`,

with the time t_2 needed to compute w when the code is given as a list of B.f. 's in NNF using our method. The comparison has been done for 10 random linear codes fixing a pair (k, n), with $n \gg k$. In the column w_{av} the average minimum weight found is shown.
An AMD E2-1800 APU processor with 850 MHz has been used for the computations. We can see that there are cases, i.e. $(k, n) = (8, 1200)$ or $(k, n) = (9, 1800)$, where our method is 10 times faster than the Brouwer-Zimmerman method. This is not surprising, since the it is known that there are cases where brute force performs better than the Brouwer-Zimmerman method, e.g.

```
> C := RandomLinearCode(GF(2),20000,20) ;
> time MinimumWeight(C:Method:="Zimmerman") ;
9665
Time: 4.220
```

k	n	t_1	t_2	t_1/t_2	w_{av}
8	$100k = 800$	0.043	0.007	6.143	360.1
8	$150k = 1200$	0.122	0.012	10.17	554.1
8	$200k = 1600$	0.122	0.015	8.13	745.2
8	$250k = 2000$	0.171	0.011	15.55	935.0
9	$100k = 900$	0.833	0.019	4.368	403.1
9	$150k = 1350$	0.116	0.020	5.800	615.6
9	$200k = 1800$	0.277	0.024	11.54	834.0
9	$250k = 2250$	0.256	0.029	8.828	1050.0
10	$100k = 1000$	0.050	0.031	1.613	448.3
10	$150k = 1500$	0.136	0.041	3.317	687.5
10	$200k = 2000$	0.178	0.050	3.560	922.7
10	$250k = 2500$	0.185	0.056	3.304	1168.3

Table 7.3: Comparison with Brouwer-Zimmerman method

```
> time MinimumWeight(C:Method:="Distribution") ;
9665
Time: 0.520
```

We also recall that Brouwer-Zimmerman method is probabilistic, while our method is deterministic.

7.7 Binary codes whose cardinality is not a power of 2

Algorithm 5 can be modified to work also with binary codes whose cardinality is not a power of 2. We now present two methods to find the minimum weight of such codes.

7.7.1 Method 1: expanding the code

A first method consist in "expanding" the code until it reaches a size of 2^k. The key observation is that the minimum weight vector of a list of vectors in $(\mathbb{F}_2)^n$ (i.e. the codewords of C) is equal to the minimum weight vector of the same list concatenated to the list of some repeated words of C (eventhough this new list is not a code anymore).

Proposition 7.7.1. *Let C be a binary nonlinear code of length n and with m codewords, where $2^{k-1} < m < 2^k$. Then there exists a set F of n Boolean functions*

$f_1, \ldots, f_n$ such that

$$C = F = \{(f_1(x_1, \ldots, x_k), \ldots, f_n(x_1, \ldots, x_k)) \mid (x_1, \ldots, x_k) \in (\mathbb{F}_2)^k\}$$

Proof. Suppose

$$C = \{c_1, \ldots, c_m\} = \{(C_{1,1}, C_{1,2}, \ldots, C_{1,n}), \ldots, (C_{m,1}, C_{m,2}, \ldots, C_{m,n})\}$$

Then consider the $(2^k \times n)$ matrix M whose first m rows are the codewords of C and the last $2^k - m$ rows are equal the a fixed codeword of C, e.g. c_m:

$$M = \left.\begin{pmatrix} C_{1,1} & C_{1,2} & \ldots & C_{1,n} \\ \ldots & \ldots & \ldots & \ldots \\ C_{m,1} & C_{m,2} & \ldots & C_{m,n} \\ \hline C_{m,1} & C_{m,2} & \ldots & C_{m,n} \\ \ldots & \ldots & \ldots & \ldots \\ C_{m,1} & C_{m,2} & \ldots & C_{m,n} \end{pmatrix}\right\} 2^k - m$$

Then for each $i = 1, \ldots, n$, the i-th column M is the vector $(C_{1,i}, \ldots, C_{m,i}, \overbrace{C_{m,i}, \ldots, C_{m,i}}^{2^k - m})$ of length 2^k and can be seen as the truth table of a B.f. in k variables.
Clearly the minimum weight codeword of C is the same as the minimum weight vector of the list L. □

In the proof of Proposition 7.7.1 we constructed the matrix M concatenating to the matrix composed by the m codewords of C one fixed codeword of C. A different choice of the concatenated $2^k - m$ codewords determines a different set $F = \{f_1, \ldots, f_n\}$. Different choices may yield to more convenient representation, but we did not investigate further.
We report in Algorithm 7 the steps to compute the minimum weight of a nonlinear binary code with size not a power of 2.

Algorithm 7 To find the evaluation vector of $\mathfrak{w}_C$ from the list of codewords of C.
Input: $C = \{c_1, \ldots, c_m\}$
Output: the evaluation vector $\underline{\mathfrak{w}}_C$ of $\mathfrak{w}_C$

1: Construct the matrix M, whose rows are $c_1, \ldots, c_m, \overbrace{c_m, \ldots, c_m}^{2^k - m}$
2: $f_j^{(\mathbb{Z})} \leftarrow$ NNF of the binary vector $(C_{1,j}, \ldots, C_{m,j}, \overbrace{C_{m,j}, \ldots, C_{m,j}}^{2^k - m})$ for $1 \leq j \leq n$
3: $\mathfrak{w}_C \leftarrow f_1^{(\mathbb{Z})} + \ldots + f_n^{(\mathbb{Z})}$
4: $\underline{\mathfrak{w}}_C \leftarrow$ Evaluation of $\mathfrak{w}_C$ over $\{0,1\}^k$
5: **return** $\underline{\mathfrak{w}}_C$

It easy to see that the complexity of Algorithm 7 is again $O(nk2^k)$.

7.7.2 Method 2: dividing into subcodes

A second approach is to divide the code C in subcodes whose cardinality is a power of 2. Then to each of these codes we can apply Algorithm 5 and then take the minimum of all the results, as shown in Algorithm 8.

Algorithm 8 To find the list of the weights of all C codewords.

Input: $C = \{c_1, \ldots, c_m\}$

Output: the evaluation vector $\underline{\mathfrak{w}}_C$ of $\mathfrak{w}_C$

1: Let $(b_r, \ldots, b_2, b_1)$ be the binary expansion of m (with b_1 lsb)
2: $\underline{v}_C = ()$, empty list
3: **for** $i = 1, \ldots, r$ **do**
4: **if** $b_i = 1$ **then**
5: Construct the $(n, 2^{i-1})$-code D taking 2^{i-1} new codewords from C
6: Apply Algorithm 5 to D
i: $f_j^{(\mathbb{Z})} \leftarrow$ NNF of the j-th column of D, with $1 \leq j \leq n$
ii: $\mathfrak{w}_D \leftarrow f_1^{(\mathbb{Z})} + \ldots + f_n^{(\mathbb{Z})}$
iii: $\underline{\mathfrak{w}}_D \leftarrow$ Evaluation of $\mathfrak{w}_D$ over $\{0,1\}^k$
7: $\underline{v}_C = \underline{v}_C || \underline{\mathfrak{w}}_D$
8: **return** $\underline{v}_C$

Remark 7.7.2. The complexity of Algorithm 8 is dominated by the complexity of Algorithm 5 applied to the largest subcode of C having a size which is a power of 2, which is

$$O(n\lfloor \log_2 m \rfloor 2^{\lfloor \log_2 m \rfloor}) .$$

Example 7.7.3. Consider the $(5, 11)$-code

$$\begin{aligned} C = \{&(0,0,0,1,1), (1,0,1,0,1), (0,1,1,1,1), (0,0,1,1,0), \\ &(1,0,0,1,1), (1,0,1,1,0), (1,1,1,0,0), (1,1,0,1,1), \\ &(1,1,1,1,0), (0,0,1,1,1), (1,1,1,1,1)\} . \end{aligned}$$

Since the binary expansion of 11 is 1011, then we can split C in 3 subcodes of size a power of 2.

$$\begin{aligned} D_1 &= \{(0,0,0,1,1)\} \\ D_2 &= \{(1,0,1,0,1), (0,1,1,1,1)\} \\ D_3 &= \{(0,0,1,1,0), (1,0,0,1,1), (1,0,1,1,0), (1,1,1,0,0), \\ &\quad (1,1,0,1,1), (1,1,1,1,0), (0,0,1,1,1), (1,1,1,1,1)\} . \end{aligned}$$

This code can be represented with 5 constant B.f. 's, 5 B.f. 's in one variable and 5 in three variables, as follows

$$\begin{aligned}
D_1 &= \{0,0,0,1,1\} \\
D_2 &= \{-x_1+1, x_1, 1, x_1, 1\} \\
D_3 &= \{2x_1x_2x_3 - x_1x_2 - x_1x_3 + x_1 - 2x_2x_3 + x_2 + x_3, \\
&\qquad x_1x_2 - x_2x_3 + x_3, \\
&\qquad -2x_1x_2x_3 + x_1x_2 + 2x_1x_3 - x_1 + x_2x_3 - x_3 + 1, \\
&\qquad x_1x_2x_3 - x_1x_2 + 1, \\
&\qquad 2x_1x_2x_3 - x_1x_2 - 2x_1x_3 + x_1 + x_3\}\,.
\end{aligned}$$

We can compute the weight polynomial for each D_i, obtaining

$$\begin{aligned}
\mathfrak{w}_{D_1} &= 2 \\
\mathfrak{w}_{D_2} &= x_1 + 3 \\
\mathfrak{w}_{D_3} &= 3x_1x_2x_3 - 2x_1x_2 - x_1x_3 + 2x_1 - x_2x_3 + x_2 + x_3 + 2\,,
\end{aligned}$$

whose evaluations over $\{0,1\}^{i-1}$ are

$$\begin{aligned}
\underline{\mathfrak{w}}_{D_1} &= (2) \\
\underline{\mathfrak{w}}_{D_2} &= (3,4) \\
\underline{\mathfrak{w}}_{D_3} &= (2,3,3,3,4,4,3,5)\,.
\end{aligned}$$

The minimum entry of these evaluation vectors is 2, which is the minimum weight of the code.

Computing the nonlinearity of Boolean function

Any function from $(\mathbb{F}_2)^n$ to $\mathbb{F}_2$ is called a Boolean function. Boolean functions are important in symmetric cryptography, since they are used in the confusion layer of ciphers. An affine Boolean function does not provide an effective confusion. To overcome this, we need functions which are as far as possible from being an affine function. The effectiveness of these functions is measured by several parameters, one of these is called "nonlinearity" ([Car10]).
In this chapter, we provide three methods to compute the nonlinearity of Boolean functions. Moreover, we give an estimate of the complexity of our methods, comparing it with the complexity of the classical method which uses the fast Walsh transform and the fast Möbius transform.
In Sections 3 and 8.1 we recall the basic notions and statements, especially regarding Boolean functions, which are necessary for our methods.
In Section 8.2 and 8.3 we provide two algorithms which reduce the problem of computing the nonlinearity of a Boolean function to that of solving a polynomial system of equations. In particular, in Section 8.3 we associate to each Boolean function in n variables a polynomial whose evaluations represent the distance from all possible affine functions.
In Section 8.4 we show that this polynomial can be used to find the nonlinearity of a Boolean function without solving the previously mentioned polynomial systems. In Section 8.5 we provide some results to express the coefficients of this polynomials, and we show in Section 8.6 that these can be computed also using fast transforms.
Finally, in Section 8.7 we analyze the complexity of the proposed methods, both experimentally and theoretically. In particular, we show that using fast Fourier methods we arrive at a worst-case complexity of $O(n2^n)$ operations over the integers, that is, sums and doublings. This way, with a different approach, we reach the same complexity of established algorithms, such as those based on the fast Walsh transform.
Part of the previously mentioned works can be found in [Bel14a], [Bel14b] and [BSS14].

For definition and notation on B.f. refer to Section 3.

8.1 Polynomials and vector weights

Here we present the main results from [SS07a], [Sim09]. The same techniques are also applied in [GOS06] and [Gue05]. Let $\mathbb{K}$ be a field and $X = \{x_1, \ldots, x_s\}$ be a set of variables. We denote by $\mathbb{K}[X]$ the multivariate polynomial ring in the variables X. If $f_1, \ldots, f_N \in \mathbb{K}[X]$, we denote by $\langle\{f_1, \ldots, f_N\}\rangle$ the ideal in $\mathbb{K}[X]$ generated by $f_1, \ldots, f_N$.
Let q be the power of a prime. We denote by $E_q[X] = \{x_1^q - x_1, \ldots, x_s^q - x_s\}$, the set of field equations in $\mathbb{F}_q[X] = \mathbb{F}_q[x_1, \ldots, x_s]$, where $s \geq 1$ is an integer, understood from now on. We write $E[X]$ when $q = 2$.

Definition 8.1.1. *Let $1 \leq t \leq s$ and $\mathsf{m} \in \mathbb{F}_q[X]$. We say that m is a* **square free monomial** *of degree t (or a* **simple t-monomial***) if:*

$$\mathsf{m} = x_{h_1} \cdots x_{h_t}, \text{ where } h_1, \ldots, h_t \in \{1, \ldots, s\} \text{ and } h_\ell \neq h_j, \forall \ell \neq j\,,$$

i.e. a monomial in $\mathbb{F}_q[X]$ such that $\deg_{x_{h_i}}(\mathsf{m}) = 1$ for any $1 \leq i \leq t$. We denote by $\mathcal{M}_{s,t}$ the set of all square free monomials of degree t in $\mathbb{F}_q[X]$.

Let $t \in \mathbb{N}$, with $1 \leq t \leq s$ and let $I_{s,t} \subset \mathbb{F}_q[X]$ be the following ideal

$$I_{s,t} = \langle\{\sigma_t, \ldots, \sigma_s\} \cup E_q[X]\rangle\,,$$

where σ_i are the elementary symmetric functions:

$$\begin{aligned}
\sigma_1 &= x_1 + x_2 + \cdots + x_s,\\
\sigma_2 &= x_1x_2 + x_1x_3 + \cdots + x_1x_s + x_2x_3 + \cdots + x_{s-1}x_s,\\
&\cdots\\
\sigma_{s-1} &= x_1x_2x_3\cdots x_{s-2}x_{s-1} + \cdots + x_2x_3\cdots x_{s-1}y_s,\\
\sigma_s &= x_1x_2\cdots x_{s-1}x_s.
\end{aligned}$$

We also denote by $I_{s,s+1}$ the ideal $\langle E_q[X]\rangle$. For any $1 \leq i \leq s$, let P_i be the set which contains all vectors in $(\mathbb{F}_q)^n$ of weight i, $P_i = \{v \in \mathbb{F}_q^n \mid \mathrm{w}(v) = i\}$, and let Q_i be the set which contains all vectors of weight up to i, $Q_i = \sqcup_{0 \leq j \leq i} P_j$.

Theorem 8.1.2. *Let t be an integer such that $1 \leq t \leq s$. Then the vanishing ideal $\mathcal{I}(Q_t)$ of Q_t is*

$$\mathcal{I}(Q_t) = I_{s,t+1}\,,$$

and its reduced Gröbner basis G is

$$\begin{aligned}
G &= E_q[X] \cup \mathcal{M}_{s,t}\,, \qquad \text{for } t \geq 2\,,\\
G &= \{x_1, \ldots, x_s\}\,, \qquad \text{for } t = 1\,.
\end{aligned}$$

Let $\mathbb{F}_q[Z]$ be a polynomial ring over $\mathbb{F}_q$. Let $\mathsf{m} \in \mathcal{M}_{s,t}$, $\mathsf{m} = z_{h_1} \cdots z_{h_t}$. For any polynomial vector W in the module $(\mathbb{F}_q[Z])^n$, $W = (W_1, \ldots, W_n)$, we denote by $\mathsf{m}(W)$ the following polynomial in $\mathbb{F}_q[Z]$:

$$\mathsf{m}(W) = W_{h_1} \cdot \ldots \cdot W_{h_t} .$$

Example 8.1.3. Let $n = s = 3, q = 2$ and $W = (x_1x_2 + x_3, x_2, x_2x_3) \in (\mathbb{F}[x_1, x_2, x_3])^3$ and $\mathsf{m} = z_1z_3$. Then

$$\mathsf{m}(W) = (x_1x_2 + x_3)(x_2x_3) .$$

8.2 Nonlinearity and polynomial systems over $\mathbb{F}$

In this section we want to tackle the following problem: to find a method to compute the nonlinearity of a given Boolean function $f \in \mathcal{B}_n$ by constructing a finite number of polynomial systems over $\mathbb{F}_2$ with N variables and such that:

A) N is of the order of n,

B) the nonlinearity is obtained by merely deciding which of these systems have a binary solution.

Since the maximum nonlinearity is of the order of 2^{n-1}, we are satisfied if the number of systems we have to construct does not exceed 2^{n-1} (or even 2^n).

In this section we report the solution of the above problem, given by Simonetti in [SS07a], which depends on Theorem 8.1.2.
The starting idea is to define an ideal such that a point in its variety corresponds to an affine function with distance at most $t - 1$ from f.

Let A be the variable set $A = \{a_i\}_{0 \le i \le n}$. We denote by $\mathfrak{g}_n \in \mathbb{F}[A, X]$ the following polynomial:

$$\mathfrak{g}_n = a_0 + \sum_{i=1}^{n} a_i x_i \ .$$

According to Lemma 3.2.3, determining the nonlinearity of $f \in \mathcal{B}_n$ is the same as finding the minimum weight of the vectors in the set $\{\underline{f} + \underline{g} \mid g \in \mathcal{A}_n\} \subset \mathbb{F}^{2^n}$. We can consider the evaluation vector of the polynomial $\mathfrak{g}_n$ as follows:

$$\underline{\mathfrak{g}_\mathfrak{n}} = (\mathfrak{g}_n(A, \mathsf{p}_1), \ldots, \mathfrak{g}_n(A, \mathsf{p}_{2^n})) \in (\mathbb{F}[A])^{2^n} .$$

Example 8.2.1. We consider the case $n = 3$. Then $\mathfrak{g}_3 = a_1x_1 + a_2x_2 + a_3x_3 + a_0$. We consider vectors in $\mathbb{F}^3$ ordered as follows:

$$\begin{array}{lll} \mathsf{p}_1 = (0,0,0), & \mathsf{p}_2 = (0,0,1), & \mathsf{p}_3 = (0,1,0), \\ \mathsf{p}_4 = (1,0,0), & \mathsf{p}_5 = (0,1,1), & \mathsf{p}_6 = (1,0,1), \\ \mathsf{p}_7 = (1,1,0), & \mathsf{p}_8 = (1,1,1). & \end{array}$$

The evaluation vector of $\mathfrak{g}_3$ is:

$$\begin{aligned} \underline{\mathfrak{g}_3} = (&a_0, a_0 + a_1, a_0 + a_2, a_0 + a_3, a_0 + a_1 + a_2, \\ &a_0 + a_1 + a_3, a_0 + a_2 + a_3, a_0 + a_1 + a_2 + a_3) \,. \end{aligned}$$

Definition 8.2.2. *We denote by $J_t^n(f)$ the ideal in $\mathbb{F}[A]$:*

$$\begin{aligned} J_t^n(f) &= \langle \{ \mathsf{m}\big(\mathfrak{g}_n(A, \mathsf{p}_1) + f(\mathsf{p}_1), \ldots, \mathfrak{g}_n(A, \mathsf{p}_{2^n}) + f(\mathsf{p}_{2^n})\big) \mid \mathsf{m} \in \mathcal{M}_{2^n,t} \} \cup E[A] \rangle \\ &= \langle \{ \mathsf{m}(\underline{\mathfrak{g}_n} + \underline{f}) \mid \mathsf{m} \in \mathcal{M}_{2^n,t} \} \cup E[A] \rangle \,. \end{aligned}$$

Remark 8.2.3. As $E[A] \subset J_t^n(f)$, $J_t^n(f)$ is zero-dimensional and radical (Theorem 1.2.21).

Lemma 8.2.4. *For $1 \leq t \leq 2^n$ the following statements are equivalent:*

1. $\mathcal{V}(J_t^n(f)) \neq \emptyset$,
2. $\exists u \in \{\underline{f} + \underline{g} \mid g \in \mathcal{A}_n\}$ *such that* $\mathrm{w}(u) \leq t - 1$,
3. $\exists \alpha \in \mathcal{A}_n$ *such that* $\mathrm{d}(f, \alpha) \leq t - 1$.

Proof.
(2)⇔(3). Obvious.
(1)⇒(2). Let $\bar{A} = (\bar{a}_0, \bar{a}_1, \ldots, \bar{a}_n) \in \mathcal{V}(J_t^n(f)) \subset \mathbb{F}^{n+1}$ and let
$u = (\mathfrak{g}_n(\bar{A}, v_1) + f(v_1), \ldots, \mathfrak{g}_n(\bar{A}, v_{2^n}) + f(v_{2^n})) \in \mathbb{F}^{2^n}$. We have that $\mathsf{m}(u) = 0$ for all $\mathsf{m} \in \mathcal{M}_{2^n,t}$. So $u \in \mathcal{V}(I_{2^n,t})$ and, thanks to Theorem 8.1.2, $u \in Q_{t-1}$, i.e. $\mathrm{w}(u) \leq t-1$.
(2)⇒(1). It can be proved by reversing the above argument. □

From Lemma 8.2.4 we immediately have the following theorem.

Theorem 8.2.5. *Let $f \in \mathcal{B}_n$. The nonlinearity $\mathrm{N}(f)$ is the minimum t such that $\mathcal{V}(J_{t+1}^n(f)) \neq \emptyset$.*

From this theorem we can derive an algorithm to compute the nonlinearity for a function $f \in \mathcal{B}_n$, by determining if the variety of the ideal $J_t^n(f)$ has a solution or not.

Algorithm 9 Basic algorithm to compute the nonlinearity of a Boolean function by finding if a solution of a polynomial systems over $\mathbb{F}$ exists

Input: a Boolean function f
Output: the nonlinearity of f

1: $j \leftarrow 1$
2: **while** $\mathcal{V}(J_j^n(f)) = \emptyset$ **do**
3: $\quad j \leftarrow j + 1$
4: **return** $j - 1$

Simonetti's systems $J_j^n(f)$ are the solutions of the problem we stated at the beginning of this section: they use only $n+1$ variables and all we want to know from them (in the worst case) is whether they have a solution or not. Observe also that the solution we are interested in does not lie in some big extension field but it must remain in $(\mathbb{F}_2)^{n+1}$.
Moreover, the number of systems we need to check is, in the worst case, the maximum nonlinearity plus one. We claim that with our constraints Simonetti's solution is, in principle, still the best-known.
However, a practical application of Algorithm 9 was missing in Simonetti's work, where she would use straightfoward applications of Gröbner bases. Before proceeding to propose more refined approaches in the next sections, we would like now to provide some examples for Simonetti's original contribution.

Remark 8.2.6. If f is not affine, we can start our check from $J_2^n(f)$.

Example 8.2.7. Let $f : \mathbb{F}^3 \to \mathbb{F}$ be the Boolean function:

$$f(x_1, x_2, x_3) = x_1x_2 + x_1x_3 + x_2 + 1\,.$$

We want to compute $\mathrm{N}(f)$ and clearly f is not affine. We compute vector $\underline{f}$ and we take a general affine function $\mathfrak{g}_3$, so that:

$$\begin{aligned}\underline{f} =&(1,1,0,1,1,0,0,0),\\ \underline{\mathfrak{g}_3} =&(a_0, a_0+a_1, a_0+a_2, a_0+a_3, a_0+a_1+a_2,\\ &a_0+a_1+a_3, a_0+a_2+a_3, a_0+a_1+a_2+a_3).\end{aligned}$$

So

$$\begin{aligned}\underline{f} + \underline{\mathfrak{g}_3} =&(a_0+1, a_0+a_1+1, a_0+a_2, a_0+a_3+1,\\ &a_0+a_1+a_2+1, a_0+a_1+a_3, a_0+a_2+a_3,\\ &a_0+a_1+a_2+a_3) = (p_1, p_2, \ldots, p_8)\,.\end{aligned}$$

Ideal $J_2^3(f)$ is the ideal generated by

$$J_2^3(f) = \langle \{p_1p_2, p_1p_3, \ldots, p_7p_8\} \cup \{a_0^2 + a_0, a_1^2 + a_1, a_2^2 + a_2, a_3^2 + a_3\} \rangle .$$

We can compute any Gröbner basis of this ideal and we obtain that it is trivial, so $\mathcal{V}(J_2^3(f)) = \emptyset$ and $\mathrm{N}(f) > 1$. Now can compute a Gröbner basis for $J_3^3(f)$. We obtain, using degrevlex ordering with $a_1 > a_2 > a_3 > a_0$, that $G(J_3^3(f)) = \{a_2+a_3+1, a_3^2+a_3, a_1a_3+a_0+1, a_0a_3+a_0+a_3+1, a_1^2+a_1, a_0a_1+a_0+a_1+1, a_0^2+a_0\}$. So, $\mathrm{N}(f) = 2$ by Theorem 8.2.5. By inspecting $G(J_3^3(f))$, we also obtain all affine functions having distance 2 from f:

$$\alpha_1 = 1 + x_1 + x_2, \qquad \alpha_2 = 1 + x_2,$$
$$\alpha_3 = 1 + x_3, \qquad \alpha_4 = x_1 + x_3 .$$

Example 8.2.8. Let $f : \mathbb{F}^5 \to \mathbb{F}$ be the Boolean function

$$f = x_1x_3x_4x_5 + x_1x_2x_4 + x_1x_4x_5 + \tag{8.1}$$
$$x_2x_3x_4 + x_2x_4x_5 + x_3x_4x_5 + x_4x_5 . \tag{8.2}$$

We have that

$$\underline{f} = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1, 0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,1) .$$

Then we compute $\underline{f} + \underline{\mathfrak{g}_5}$ and we obtain:

$$\begin{aligned}
\underline{f} + \underline{\mathfrak{g}_5} =& (a_0, a_1 + a_0, a_2 + a_0, a_3 + a_0, a_4 + a_0, a_5 + a_0,\\
& a_1 + a_2 + a_0, a_1 + a_3 + a_0, a_1 + a_4 + a_0,\\
& a_1 + a_5 + a_0, a_2 + a_3 + a_0, a_2 + a_4 + a_0,\\
& a_2 + a_5 + a_0, a_3 + a_4 + a_0, a_3 + a_5 + a_0,\\
& a_4 + a_5 + a_0 + 1, a_1 + a_2 + a_3 + a_0,\\
& a_1 + a_2 + a_4 + a_0 + 1, a_1 + a_2 + a_5 + a_0,\\
& a_1 + a_3 + a_4 + a_0, a_1 + a_3 + a_5 + a_0,\\
& a_1 + a_4 + a_5 + a_0, a_2 + a_3 + a_4 + a_0 + 1,\\
& a_2 + a_3 + a_5 + a_0, a_2 + a_4 + a_5 + a_0,\\
& a_3 + a_4 + a_5 + a_0, a_1 + a_2 + a_3 + a_4 + a_0,\\
& a_1 + a_2 + a_3 + a_5 + a_0, a_1 + a_2 + a_4 + a_5 + a_0,\\
& a_1 + a_3 + a_4 + a_5 + a_0, a_2 + a_3 + a_4 + a_5 + a_0,\\
& a_1 + a_2 + a_3 + a_4 + a_5 + a_0 + 1) =\\
=& (p_1, p_2, \ldots, p_{32}) .
\end{aligned}$$

As it is obvious that f is not affine, we start from the ideal $J_2^5(f)$, which is generated by

$$J_2^5(f) = \langle \{p_1p_2, p_1p_3, \ldots, p_{31}p_{32}\} \cup \{a_0^2 + a_0, a_1^2 + a_1, \\ a_2^2 + a_2, a_3^2 + a_3, a_4^2 + a_4, a_5^2 + a_5\} \rangle .$$

The Gröbner basis of $J_2^5(f)$ with respect to any monomial order is trivial so we compute a Gröbner basis of $J_3^5(f)$. We obtain that the Gröbner basis of $J_t^5(f)$ is trivial with respect to any monomial order for $2 \leq t \leq 4$. For $t = 5$, we obtain the following Gröbner basis with respect to the degrevlex order with $a_1 > a_2 > a_3 > a_4 > a_5 > a_0$:

$$G(J_5^5(f)) = \{a_0, a_5, a_4, a_3, a_2, a_1\} .$$

Then $\mathrm{N}(f) = 4$, that is, there is only one affine function α which has distance equal to 4 from f: $\alpha = 0$.

8.3 Nonlinearity and polynomial systems over $\mathbb{Q}$

Here we present an algorithm to compute the nonlinearity of a Boolean function by solving a polynomial system of equations over $\mathbb{Q}$ rather than over $\mathbb{F}$, which turns out to be much faster than Algorithm 9. The same algorithm can be slightly modified to work over the field $\mathbb{F}_p$, where p is a prime. The complexity of these algorithms will be analyzed in Section 8.7.

As we have seen in Section 8.2, the nonlinearity of a Boolean function can be computed solving polynomial systems over $\mathbb{F}$. It is sufficient to find the minimum j such that the variety of the ideal $J_t^n(f)$ is not empty. Recall that

$$J_t^n(f) = \langle \{\mathsf{m}(\underline{\mathfrak{g}_n} + \underline{f}) \mid \mathsf{m} \in \mathcal{M}_{2^n,t}\} \cup E[A] \rangle .$$

This method becomes impractical even for small values of n, since $\binom{2^n}{t}$ monomials have to be evaluated. A first slight improvement could be achieved by adding to the ideal one monomial evaluation at a time and check if 1 has appeared in the Gröbner basis. Even this way, the algorithm remains very slow.
For each $i = 1, \ldots, 2^n$, let us denote:

$$f_i^{(\mathbb{F})}(A) = \mathfrak{g}_n(A, \mathsf{p}_i) + f(\mathsf{p}_i)$$

the Boolean function where as usual $A = \{a_0, \ldots, a_n\}$ are the $n + 1$ variables representing the coefficient of a generic affine function.
In this case we have that:

$$(f_1^{(\mathbb{F})}(A), \ldots, f_{2^n}^{(\mathbb{F})}(A)) = \underline{\mathfrak{g}_n}(A) + \underline{f} \in (\mathbb{F}[A])^{2^n}$$

Note that the polynomials $f_i^{(\mathbb{F})}$ are affine polynomials.
We also denote by

$$f_i^{(\mathbb{Z})}(A) = \mathrm{NNF}(f_i^{(\mathbb{F})}(A))$$

the NNF of each $f_i^{(\mathbb{F})}(A)$ (obtained as in [CG99], Theorem 1).

Definition 8.3.1. *We call* $\mathfrak{n}_f(A) = f_1^{(\mathbb{Z})}(A) + \cdots + f_n^{(\mathbb{Z})}(A) \in \mathbb{Z}[A]$ *the* **integer nonlinearity polynomial** *(or simply the* nonlinearity polynomial*) of the Boolean function* f.
For any $t \in \mathbb{N}$ *we define the ideal* $\mathcal{N}_f^t \subseteq \mathbb{Q}[A]$ *as follows:*

$$\mathcal{N}_f^t = \langle E[A] \bigcup \{f_1^{(\mathbb{Z})} + \cdots + f_{2^n}^{(\mathbb{Z})} - t\}\rangle = \tag{8.3}$$

$$= \langle E[A] \bigcup \{\mathfrak{n}_f - t\}\rangle \tag{8.4}$$

Note that the evaluation vector $\underline{\mathfrak{n}_f}$ represents all the distances of f from all possible affine functions (in n variables).

Theorem 8.3.2. *The variety of the ideal* $\mathcal{N}_f^t$ *is non-empty if and only if the Boolean function* f *has distance* t *from an affine function. In particular,* $\mathrm{N}(f) = t$*, where* t *is the minimum positive integer such that* $\mathcal{V}(\mathcal{N}_f^t) \neq \emptyset$.

Proof. Note that

$$\mathcal{N}_f^t = \langle E[A]\rangle + \langle\{\mathfrak{n}_f(A) - t\}\rangle$$

and so

$$\mathcal{V}(\mathcal{N}_f^t) = \mathcal{V}(\langle E[A]\rangle) \cap \mathcal{V}(\langle\{\mathfrak{n}_f(A) - t\}\rangle)\,.$$

Therefore $\mathcal{V}(\mathcal{N}_f^t) \neq \emptyset$ if and only if $\exists \bar{a} = (\bar{a}_0, \ldots, \bar{a}_n) \in \mathcal{V}(\langle E[A]\rangle)$ such that $\mathfrak{n}_f(\bar{a}) = t$.
Let $\alpha \in \mathcal{A}_n$ such that $\alpha(X) = \bar{a}_0 + \sum_{i=1}^n \bar{a}_i x_i$.
By definition we have

$$f_i^{(\mathbb{Z})} = 1 \iff f(\mathsf{p}_i) \neq \alpha(\mathsf{p}_i)$$

and

$$f_i^{(\mathbb{Z})} = 0 \iff f(\mathsf{p}_i) = \alpha(\mathsf{p}_i)\,.$$

Hence

$$\mathfrak{n}_f(\bar{a}) = \sum_{i=1}^{2^n} f_i^{(\mathbb{Z})}(\bar{a}) - t = 0 \iff |\{i \mid f(\mathsf{p}_i) \neq \alpha(\mathsf{p}_i)\}| = t$$

$$\iff \mathrm{d}(f, \alpha) = t\,.$$

and our claim follows directly. □

To compute the nonlinearity of f we can use Algorithm 10 with input f.

Algorithm 10 To compute the nonlinearity of the Boolean function f

Input: f

Output: nonlinearity of f

1: Compute $\mathfrak{n}_f$
2: $j \leftarrow 1$
3: **while** $\mathcal{V}(\mathcal{N}_f^j) = \emptyset$ **do**
4: $\quad j \leftarrow j + 1$
5: **return** j

Algorithm 10 can be modified to eliminate the while cycle. Instead of checking if a solution of the system

$$\begin{cases} a_0^2 - a_0 = 0 \\ \dots \\ a_n^2 - a_n = 0 \\ \mathfrak{n}_C(a_0, \dots, a_n) - j = 0 \end{cases} \tag{8.5}$$

exists in the affine algebra $\mathbb{Q}/\langle a_0^2 - a_0, \dots, a_n^2 - a_n \rangle$ for each $j \in \{1, \dots, 2^n\}$, we can add the variable t to the system

$$\begin{cases} a_0^2 - a_0 = 0 \\ \dots \\ a_n^2 - a_n = 0 \\ \mathfrak{n}_C(a_0, \dots, a_n) - t = 0 \end{cases} \tag{8.6}$$

and solve it in $\mathbb{Q}[t]/\langle a_0^2 - a_0, \dots, a_n^2 - a_n \rangle$, with respect to lexicographical monomial ordering, to find as a solution a polynomial $\mathfrak{t}(t)$, whose zeros are integers, representing the possible distances of the Boolean function f from the affine functions. We are interested in the smallest solution of $\mathfrak{t}(t)$.
We did not investigate further which of the two solutions is best.

8.4 Computing the nonlinearity using fast polynomial evaluation

Once the nonlinearity polynomial $\mathfrak{n}_f$ is defined, we can use another approach to compute the nonlinearity avoiding the hard task of solving a polynomial system of

equations.
We have to find the minimum nonnegative integer t in the set of the evaluations of $\mathfrak{n}_f$, that is, in $\{\mathfrak{n}_f(\bar{a}) \mid \bar{a} \in \{0,1\}^{n+1} \subset \mathbb{Z}^{n+1}\}$.
We write explicitly the modified algorithm.

Algorithm 11 To compute the nonlinearity of the Boolean function f

Input: f
Output: nonlinearity of f

1: **if** $f \in \mathcal{A}_n$ **then**
2: **return** 0
3: **else**
4: Compute $\mathfrak{n}_f$
5: Compute $m = \min\{\mathfrak{n}_f(\bar{a}) \mid \bar{a} \in \{0,1\}^{n+1}\}$
6: **return** m

Example 8.4.1. Consider the case $n = 2$, $f(x_1, x_2) = x_1x_2 + 1$. We have that $\underline{f} = (1,1,1,0)$ and $\underline{\mathfrak{g}_n} = (a_0, a_0 + a_1, a_0 + a_2, a_0 + a_1 + a_2)$.
Let us compute all $f_i^{(\mathbb{F})} = (\underline{\mathfrak{g}_n} + \underline{f})_i$ and $f_i^{(\mathbb{Z})}$,for $i = 1, \ldots, 2^2$:

$$\begin{aligned}
f_1^{(\mathbb{F})} &= a_0 + 1 && \to f_1^{(\mathbb{Z})} = -a_0 + 1 \\
f_2^{(\mathbb{F})} &= a_0 + a_1 + 1 && \to f_2^{(\mathbb{Z})} = 2a_0a_1 - a_0 - a_1 + 1 \\
f_3^{(\mathbb{F})} &= a_0 + a_2 + 1 && \to f_3^{(\mathbb{Z})} = 2a_0a_2 - a_0 - a_2 + 1 \\
f_4^{(\mathbb{F})} &= a_0 + a_1 + a_2 && \to f_4^{(\mathbb{Z})} = 4a_0a_1a_2 - 2a_0a_1 - 2a_0a_2 \\
& && \quad + a_0 - 2a_1a_2 + a_1 + a_2
\end{aligned}$$

Then $\mathfrak{n}_f = f_1^{(\mathbb{Z})} + f_2^{(\mathbb{Z})} + f_3^{(\mathbb{Z})} + f_4^{(\mathbb{Z})} = 4a_0a_1a_2 - 2a_0 - 2a_1a_2 + 3$ and since

$$\underline{\mathfrak{n}_f} = (3,1,3,1,3,1,1,3)$$

then the nonlinearity of f is 1.
Observe that the vector $\underline{\mathfrak{n}_f}$ represents all the distances of f from all possible affine functions in 2 variables, that is, from $0, 1, x_1, x_1 + 1, x_2, x_2 + 1, x_1 + x_2, x_1 + x_2 + 1$.

8.5 Properties of the nonlinearity polynomial

From now on, with abuse of notation, we sometimes consider 0 and 1 as elements of $\mathbb{F}$ and other times as elements of $\mathbb{Z}$.
We have the following definition

Definition 8.5.1. *Given* $b_1, \ldots, b_n \in \mathbb{F}$

$$b_1 \oplus \ldots \oplus b_n = \sum_{\mathbf{v}=(v_1,\ldots,v_n)\in\mathbb{F}^n, \mathbf{v}\neq\mathbf{0}} (-2)^{\mathrm{w}(\mathbf{v})-1} \cdot b_1^{v_1} \cdots b_n^{v_n} .$$

where the sum on the right is in $\mathbb{Z}$.

It is easy to show that $b_1 \oplus \ldots \oplus b_n \in \{0, 1\}$.
We give a theorem to compute the coefficients of the nonlinearity polynomial.

Theorem 8.5.2. *Let* $v = (v_0, v_1, \ldots, v_n) \in \mathbb{F}^{n+1}$, $\tilde{v} = (v_1, \ldots, v_n) \in \mathbb{F}^n$, $A^v = a_0^{v_0} \cdots a_n^{v_n} \in \mathbb{F}[A]$ *and* $c_v \in \mathbb{Z}$ *be such that* $\mathfrak{n}_f = \sum_{v\in\mathbb{F}^{n+1}} c_v A^v$. *Then the coefficients of* $\mathfrak{n}_f$ *can be computed as:*

$$c_v = \sum_{u\in\mathbb{F}^n} f(u) = \mathrm{w}(\underline{f}) \quad \text{if } v = 0 \tag{8.7}$$

$$c_v = (-2)^{\mathrm{w}(v)} \sum_{\substack{u\in\mathbb{F}^n \\ \tilde{v}\preceq u}} \left[f(u) - \frac{1}{2} \right] \quad \text{if } v \neq 0 \tag{8.8}$$

Proof. The nonlinearity polynomial is the integer sum of the 2^n numerical normal forms of the affine polynomials $\mathfrak{g}_n(A, u) \oplus f(u) \in \mathbb{F}[A]$, each identified by the vector $u \in \mathbb{F}^n$, i.e.:

$$\mathfrak{n}_f = \sum_{u\in\mathbb{F}^n} \mathrm{NNF}(\mathfrak{g}_n(A, u) \oplus f(u)) = \sum_{u\in\mathbb{F}^n} \mathrm{NNF}(a_0 \oplus a_1 u_1 \oplus \ldots \oplus a_n u_n \oplus f(u))$$

which is a polynomial in $\mathbb{Z}[A]$.
The NNF of $\mathfrak{g}_n(A, u) \oplus f(u)$ is a polynomial with 2^{n+1} terms, i.e.:

$$\mathrm{NNF}(\mathfrak{g}_n(A, u) \oplus f(u)) = \sum_{v\in\mathbb{F}^{n+1}} \lambda_v A^v ,$$

for some $\lambda_v \in \mathbb{Z}$, and by Proposition 3.1.9

$$\lambda_v(u) = (-1)^{\mathrm{w}(v)} \sum_{a\in\mathbb{F}^{n+1} | a\preceq v} (-1)^{\mathrm{w}(a)} \Big(\mathfrak{g}_n(a, u) \oplus f(u) \Big) .$$

Let us prove Equation (8.7). When $v = (0, \ldots, 0)$ we have

$$c_{(0,\ldots,0)} = \sum_{u\in\mathbb{F}^n} \big[\mathfrak{g}_n((0, \ldots, 0), u) \oplus f(u) \big] = \sum_{u\in\mathbb{F}^n} f(u) .$$

Let us prove Equation (8.8). Suppose $v \neq 0$.
Now the coefficient c_v of the monomial A^v of the nonlinearity polynomial is such that:

$$\begin{aligned} c_v &= \sum_{u\in\mathbb{F}^n} \lambda_v(u) = \\ &= \sum_{u\in\mathbb{F}^n} (-1)^{\mathrm{w}(v)} \sum_{\substack{a\in\mathbb{F}^{n+1},\\ a\preceq v}} (-1)^{\mathrm{w}(a)} \big[\mathfrak{g}_n(a,u) \oplus f(u)\big] = \\ &= (-1)^{\mathrm{w}(v)} \sum_{u\in\mathbb{F}^n} \sum_{\substack{a\in\mathbb{F}^{n+1},\\ a\preceq v}} (-1)^{\mathrm{w}(a)} \big[\mathfrak{g}_n(a,u) \oplus f(u)\big]\,. \end{aligned} \tag{8.9}$$

We prove that each u such that $\tilde{v} = (v_1, \ldots, v_n) \not\preceq u$ yields a zero term in the summation, as follows.
If $\tilde{v} \not\preceq u$ then $\exists i \in \{1, \ldots, n\}$ s.t. $v_i > u_i$, i.e. $v_i = 1, u_i = 0$. We claim that $\forall a \in \mathbb{F}^{n+1}$ s.t. $a \preceq v$ $\exists \bar{a} = (\bar{a}_0, \ldots, \bar{a}_n) \in \mathbb{F}^{n+1}$ s.t. $\bar{a} \preceq v$ and

$$(-1)^{\mathrm{w}(a)} \big[\mathfrak{g}_n(a,u) \oplus f(u)\big] + (-1)^{\mathrm{w}(\bar{a})} \big[\mathfrak{g}_n(\bar{a},u) \oplus f(u)\big] = 0 \tag{8.10}$$

It is sufficient to choose $\bar{a}_i \neq a_i$ and $\bar{a}_j = a_j$ for all $j \in \{1, \ldots, n\}, j \neq i$. Clearly $\bar{a} \preceq v$ and $a \preceq v$ since $v_i = 1$.
By direct substitution we obtain

$$\begin{aligned} &(-1)^{\mathrm{w}(a)} \big[\mathfrak{g}_n(a,u) \oplus f(u)\big] + (-1)^{\mathrm{w}(\bar{a})} \big[\mathfrak{g}_n(\bar{a},u) \oplus f(u)\big] = \\ =&(-1)^{\mathrm{w}(a)} \big[a_0 \oplus a_1u_1 \oplus \ldots \oplus a_iu_i \oplus \ldots \oplus a_nu_n\big] + \\ &(-1)^{\mathrm{w}(a)}(-1)\big[\bar{a}_0 \oplus \bar{a}_1u_1 \oplus \ldots \oplus \bar{a}_iu_i \oplus \ldots \oplus \bar{a}_nu_n\big] \\ =&(-1)^{\mathrm{w}(a)}[a_iu_i - \bar{a}_iu_i] = 0\,. \end{aligned}$$

Thanks to (8.10) we can continue from (8.9) and get

$$\begin{aligned} c_v = (-1)^{\mathrm{w}(v)} \sum_{\substack{u\in\mathbb{F}^n\\ \tilde{v}\preceq u}} \sum_{\substack{a\in\mathbb{F}^{n+1},\\ a\preceq v}} (-1)^{\mathrm{w}(a)} \big[&\mathfrak{g}_n(a,u) + f(u) \\ &-2\mathfrak{g}_n(a,u)f(u)\big]\,, \end{aligned} \tag{8.11}$$

where we used $a \oplus b = a + b - 2ab$.

Now we consider v, u fixed, and $\tilde{v} \preceq u$.
There are exactly $2^{\mathrm{w}(v)}$ vectors a such that $a \preceq v$, i.e.:

$$|\{a \in \mathbb{F}^{n+1} \mid a \preceq v\}| = 2^{\mathrm{w}(v)} \tag{8.12}$$

Now we want to study the internal summation in (8.11).
If $u = (0, \ldots, 0)$ then $\forall a = (a_0, \ldots, a_n) \preceq v$ we have $\mathfrak{g}_n(a,u) = a_0 \oplus a_1u_1 \oplus \ldots a_nu_n =$

a_0.
Otherwise, if $u \neq (0, \ldots, 0)$ we can consider the following set of indices $U = \{j \mid u_j = 1\} = \{j_1, \ldots, j_{\mathrm{w}(u)}\}$, which has size $\mathrm{w}(u)$.
Since $a \preceq v$ and $\tilde{v} \preceq u$ then $(a_1, \ldots, a_n) \preceq u$ by transitivity. For all $j \notin U$ we have $a_j = 0$, and then $\mathrm{w}(a_0, a_{j_1}, \ldots, a_{j_{\mathrm{w}(u)}}) = \mathrm{w}(a)$.
Thus, for any $u \in \mathbb{F}^n$ we have

$$\mathfrak{g}_n(a, u) = a_0 \oplus a_{j_1} \oplus \ldots \oplus a_{j_{\mathrm{w}(u)}} = \begin{cases} 1 \text{ if } \mathrm{w}(a) \text{ is odd} \\ 0 \text{ if } \mathrm{w}(a) \text{ is even} \end{cases} \tag{8.13}$$

and each of the two cases happens for exactly one half of the vectors $a \preceq v$. Clearly the two halves are disjoint.
This yields, from (8.9) and (8.11), the following chain of equalities:

$$\begin{aligned}
c_v &= \sum_{u \in \mathbb{F}^n} \lambda_v(u) = \\
&= (-1)^{\mathrm{w}(v)} \sum_{\substack{u \in \mathbb{F}^n \\ \tilde{v} \preceq u}} \Bigg[\sum_{\substack{a \in \mathbb{F}^{n+1}, \\ a \preceq v \\ \mathfrak{g}_n(a,u)=0}} (-1)^{\mathrm{w}(a)} f(u) + \\
&\qquad \sum_{\substack{a \in \mathbb{F}^{n+1}, \\ a \preceq v \\ \mathfrak{g}_n(a,u)=1}} (-1)^{\mathrm{w}(a)} (1 - f(u)) \Bigg] = \\
&= (-1)^{\mathrm{w}(v)} \sum_{\substack{u \in \mathbb{F}^n \\ \tilde{v} \preceq u}} \left[\sum_{\substack{a \in \mathbb{F}^{n+1}, \\ a \preceq v \\ \mathfrak{g}_n(a,u)=0}} f(u) + \sum_{\substack{a \in \mathbb{F}^{n+1}, \\ a \preceq v \\ \mathfrak{g}_n(a,u)=1}} (f(u) - 1) \right] = \\
&= (-1)^{\mathrm{w}(v)} \sum_{\substack{u \in \mathbb{F}^n \\ \tilde{v} \preceq u}} \left[2^{\mathrm{w}(v)-1} f(u) + 2^{\mathrm{w}(v)-1} (f(u) - 1) \right] = \\
&= (-1)^{\mathrm{w}(v)} \sum_{\substack{u \in \mathbb{F}^n \\ \tilde{v} \preceq u}} \left[2^{\mathrm{w}(v)} f(u) - 2^{\mathrm{w}(v)-1} \right] = \\
&= (-2)^{\mathrm{w}(v)} \sum_{\substack{u \in \mathbb{F}^n \\ \tilde{v} \preceq u}} \left[f(u) - \frac{1}{2} \right]
\end{aligned}$$

which proves the theorem. □

In particular we have:

Corollary 8.5.3. *Let* $u = (u_1, \ldots, u_n)$ *and*

$$\mathfrak{n}_f = \sum_{u \in \mathbb{F}^n} c_{(0,u)} a_1^{u_1} \cdot \ldots \cdot a_n^{u_n} + a_0 \sum_{u \in \mathbb{F}^n} c_{(1,u)} a_1^{u_1} \cdot \ldots \cdot a_n^{u_n} .$$

Then we have that:

$$c_{(1,0,\ldots,0)} = 2^n - 2\mathrm{w}(\underline{f}) \tag{8.14}$$

And $\forall \tilde{v} \in \mathbb{F}^n, \tilde{v} \neq 0$ *we have:*

$$c_{(1,\tilde{v})} = -2c_{(0,\tilde{v})}, \ . \tag{8.15}$$

Corollary 8.5.3 shows that it is sufficient to store half of the coefficients of $\mathfrak{n}_f$, precisely the coefficients of the monomials where a_0 does not appear.

Corollary 8.5.4. *Each coefficient c of the nonlinearity polynomial $\mathfrak{n}_f$ is such that $|c| \leq 2^n$.*

Corollary 8.5.5. *Given the nonlinearity polynomial of f as*

$$\mathfrak{n}_f(a_0, \ldots, a_n) = c_{(0,\ldots,0)} + \sum_{\substack{(p_0,\ldots,p_n)\in\mathbb{F}^{n+1} \\ (p_0,\ldots,p_n)\neq(0,\ldots,0)}} c_{(p_0,\ldots,p_n)} a_0^{p_0} \cdot \ldots \cdot a_n^{p_n}$$

then the nonlinearity polynomial of $f \oplus 1$ is related to that of f by the following rule:

$$\mathfrak{n}_{f\oplus 1}(a_0, \ldots, a_n) = 2^n - c_{(0,\ldots,0)} + \sum_{\substack{(p_0,\ldots,p_n)\in\mathbb{F}^{n+1} \\ (p_0,\ldots,p_n)\neq(0,\ldots,0)}} -c_{(p_0,\ldots,p_n)} a_0^{p_0} \cdot \ldots \cdot a_n^{p_n}$$

A scheme that shows how to derive the coefficients of the nonlinearity polynomial in the case $n = 3$ can be seen in Tables 8.1 and 8.2.

u	$f(u) + \mathfrak{g}_n(a_0, a_1, a_2, a_3, u)$	1	a_3	a_2	a_2a_3	a_1	a_1a_3	a_1a_2	$a_1a_2a_3$
000	$v_1 + a_0$	v_1							
001	$v_2 + a_0 + a_3$	v_2	$1 - 2v_2$						
010	$v_2 + a_0 + a_2$	v_3		$1 - 2v_3$					
011	$v_2 + a_0 + a_2 + a_3$	v_4	$1 - 2v_4$	$1 - 2v_4$	$-2 + 4v_4$				
100	$v_2 + a_0 + a_1$	v_5				$1 - 2v_5$			
101	$v_2 + a_0 + a_1 + a_3$	v_6	$1 - 2v_6$			$1 - 2v_6$	$-2 + 4v_6$		
110	$v_2 + a_0 + a_1 + a_2$	v_7		$1 - 2v_7$		$1 - 2v_7$		$-2 + 4v_7$	
111	$v_2 + a_0 + a_1 + a_2 + a_3$	v_8	$1 - 2v_8$	$1 - 2v_8$	$-2 + 4v_8$	$1 - 2v_8$	$-2 + 4v_8$	$-2 + 4v_8$	$4 - 8v_8$

Table 8.1: Computation of the coefficients of the nonlinearity polynomial with $n = 3$. Each line represents the NNF coefficients of the terms of $f(u)+\mathfrak{g}_n(A, u)$ *not* containing a_0.

u	$f(u)+\mathfrak{g}_n(a_0,a_1,a_2,a_3,u)$	a_0	a_0a_3	a_0a_2	$a_0a_2a_3$	a_0a_1	$a_0a_1a_3$	$a_0a_1a_2$	$a_0a_1a_2a_3$
000	v_1+a_0	$1-2v_1$							
001	$v_2+a_0+a_3$	$1-2v_2$	$-2+4v_2$						
010	$v_2+a_0+a_2$	$1-2v_3$		$-2+4v_3$					
011	$v_2+a_0+a_2+a_3$	$1-2v_4$	$-2+4v_4$	$-2+4v_4$	$4-8v_4$				
100	$v_2+a_0+a_1$	$1-2v_5$				$-2+4v_5$			
101	$v_2+a_0+a_1+a_3$	$1-2v_6$	$-2+4v_6$			$-2+4v_6$	$4-8v_6$		
110	$v_2+a_0+a_1+a_2$	$1-2v_7$		$-2+4v_7$		$-2+4v_7$		$4-8v_7$	
111	$v_2+a_0+a_1+a_2+a_3$	$1-2v_8$	$-2+4v_8$	$-2+4v_8$	$4-8v_8$	$-2+4v_8$	$4-8v_8$	$4-8v_8$	$-8+16v_8$

Table 8.2: Computation of the coefficients of the nonlinearity polynomial with $n=3$. Each line represents the NNF coefficients of the terms of $f(u)+\mathfrak{g}_n(A,u)$ containing a_0.

8.6 Complexity of constructing the nonlinearity polynomial

We write the algorithm (Algorithm 12) to calculate the nonlinearity polynomial in $O(n2^n)$ integer operations.

Algorithm 12 Algorithm to calculate the nonlinearity polynomial $\mathfrak{n}_f$ in $O(n2^n)$ integter operations.

Input: The evaluation vector $\underline{f}$ of a Boolean function $f(x_1,\ldots,x_n)$

Output: the vector $c=(c_1,\ldots,c_{2^{n+1}})$ of the coefficients of $\mathfrak{n}_f$

Calculation of the coefficients of the monomials not containing a_0

1: $(c_1,\ldots,c_{2^n})=\underline{f}$
2: **for** $i=0,\ldots,n-1$ **do**
3: $\quad b\leftarrow 0$
4: $\quad$ **repeat**
5: $\quad\quad$ **for** $x=b,\ldots,b+2^i-1$ **do**
6: $\quad\quad\quad c_{x+1}\leftarrow c_{x+1}+c_{x+2^i+1}$
7: $\quad\quad\quad$ **if** $x=b$ **then**
8: $\quad\quad\quad\quad c_{x+2^i+1}\leftarrow 2^i-2c_{x+2^i+1}$
9: $\quad\quad\quad$ **else**
10: $\quad\quad\quad\quad c_{x+2^i+1}\leftarrow -2c_{x+2^i+1}$
11: $\quad\quad b\leftarrow b+2^{i+1}$
12: $\quad$ **until** $b=2^n$

Calculation of the coefficients of the monomials containing a_0

13: $c_{1+2^n}\leftarrow 2^n-2c_1$
14: **for** $i=2,\ldots,2^n$ **do**
15: $\quad c_{i+2^n}\leftarrow -2c_i$
16: **return** c

In Figure 8.1 Algorithm 12 is shown for $n = 3$.

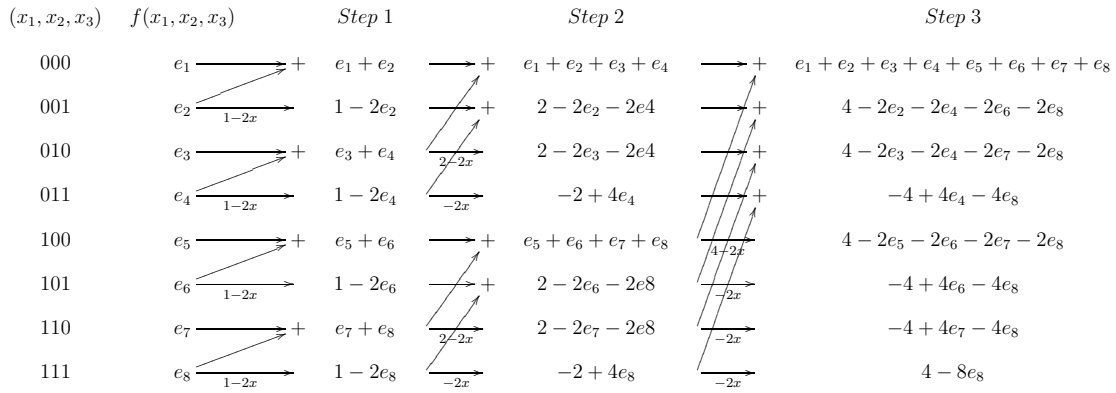

Figure 8.1: Butterfly scheme to obtain a fast computation of the nonlinearity polynomial coefficients, where $(e_1, \ldots, e_8) = (f(\mathsf{p}_1), \ldots, f(\mathsf{p}_8))$.

Theorem 8.6.1. *Algorithm 12 requires:*

1. *$O(n2^n)$ integer sums and doublings.*
   In particular $n2^n$ integer sums and $n2^{n-1}$ integer doublings, i.e. the big O constant is $c = 3/2$, provided doubling costs as summing.

2. *the storage of $O(2^n)$ integers of size less than or equal to 2^n.*

Proof. In the first part of Algorithm 12 (the computation of the coefficients of the monomials not containing a_0) the iteration on i is repeated n times.
For each i, Step 6 and Step 8 or 10 are repeated $2^i \frac{2^n}{2^{i+1}} = 2^{n/2}$ times (since b goes from 0 to 2^n by a step of 2^{i+1} and x performs 2^i steps). In Step 6 only one integer sum is performed, in Steps 8 we have one integer sum and one doubling, and in Step 10 only one doubling. Then the total amount of integer operation is

$$O(n2^n)$$

, where the constant c in the big O notation is $3/2$, provided doubling costs as summing.
Finally the computation of the coefficients of the monomials containing a_0 requires only 2^n integer doublings.
To store all the monomials of the nonlinearity polynomial we have to store 2^{n+1} integers, although Corollary 8.5.3 shows that it is sufficient to store only the first half of them, i.e. 2^n integers. By Corollary 8.5.4, their size is less than or equal to 2^n. □

8.7 Complexity considerations

First we recall that the complexity of computing the nonlinearity of a Boolean function with n variables, having as input its coefficients vector, is $O(n2^n)$ using the Fast Möbius and the Fast Walsh Transform.
We now want to analyze the complexity of Algorithm 9, 10, 11.

8.7.1 Some considerations on Algorithm 9

In Algorithm 9, almost all the computations are wasted evaluating all possible simple-t-monomials in 2^n variables, which are $\binom{2^n}{t}$. This number grows enormously even for small values of n and t. We investigated experimentally how many of the $\binom{2^n}{t}$ monomials are actually needed to compute the final Gröbner basis of J_t^n. Our experiment ran over all possible Boolean functions in 3 and 4 variables. The results are reported in Tables 8.3, 8.4 and 8.5.
In this tables, for each J_t^n there are four columns. Let G_t^n be the Gröbner basis of J_t^n. Under the column labeled #C we report the average number of *checked* monomials in 2^n variables before obtaining G_t^n.
Under the column labeled #S we report the average number of monomials which are actually *sufficient* to obtain G_t^n.
Under the columns labeled "m" e "M" we report, respectively, the minimum and the maximum number of sufficient monomials to find G_t^n running through all possible Boolean functions in n variables.
For example, to compute the Gröbner basis of the ideal J_2^3 associated to a Boolean function f whose nonlinearity is 2, we needed to check on average 24 monomials before finding the correct basis. Between the 24 monomials only 9.7 (on average) were sufficient to obtain the same basis, where the number of sufficient monomials never exceeded the range $8 - 11$.

	J_1^3				J_2^3				J_3^3			
NL	#S	m	M	#C	#S	m	M	#C	#S	m	M	#C
0	4	4	4	8	0	0	0	0	0	0	0	0
1	4.5	4	5	4.4	8.5	7	10	28	0	0	0	0
2	4.4	4	5	4	9.7	8	11	24	9.3	8	11	56

Table 8.3: Number of monomials needed to compute the Gröbner basis of the ideal J_t^3.

	J_1^4				J_2^4				J_3^4			
NL	#S	m	M	#C	#S	m	M	#C	#S	m	M	#C
0	5	5	5	16	0	0	0	0	0	0	0	0
1	5.25	4	6	8	8.75	8	11	120	0	0	0	0
2	4.83	4	6	5.67	9.97	8	12	62.83	14.50	12	18	560
3	4.62	4	6	4.76	9.92	8	12	42.72	15.76	13	19	315.04
4	4.53	4	6	4.42	9.83	8	12	37.49	15.81	13	19	246.19
5	4.46	4	5	4.19	10.11	8	12	34.39	15.89	13	19	215.68
6	4.43	4	5	4.00	9.71	8	11	24.00	17.29	16	19	156.86

Table 8.4: Number of monomials needed to compute the Gröbner basis of the ideal J_t^4, $t = 1, 2, 3$.

	J_4^4				J_5^4				J_6^4				J_7^4			
NL	#S	m	M	#C	#S	m	M	#C	#S	m	M	#C	#S	m	M	#C
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	20.18	15	23	1820	0	0	0	0	0	0	0	0	0	0	0	0
4	21.44	16	24	1319.96	23.99	22	29	4368	0	0	0	0	0	0	0	0
5	21.54	19	24	1003.15	26.00	24	28	3851.24	23.50	22	25	8008	0	0	0	0
6	19.57	19	20	671.71	28	28	28	2603.79	28	28	28	7608.79	16	16	16	11441

Table 8.5: Number of monomials needed to compute the Gröbner basis of the ideal J_t^4,$t = 4, 5, 6, 7$.

8.7.2 Algorithm 9 and 10

Since it is not easy to estimate the complexity of a Gröbner basis computation theoretically, we give some experimental results, shown in Table 8.6. In this table we report the coefficients of growth of the analyzed algorithms [1] , comparing them with the value $\log_2\left[\frac{(n+1)2^{n+1}}{n2^n}\right]$. For each algorithm we compute the average time t_n to compute the nonlinearity of a Boolean function with n variables and the average time t_{n+1} to compute the nonlinearity of a Boolean function with $n+1$ variables. Then we report in the table the value $\log_2\left(\frac{t_{n+1}}{t_n}\right)$. When Gröbner bases are computed, then graded reverse lexicographical order is used, with Magma [MAG] (Version 2.19) implementation of the Faugère $F4$ algorithm. Since the ideal $J_t^n(f)$ of Definition 8.2.2 is derived from the evaluation of $\binom{2^n}{t}$ monomials (generating at most the same number of equations), then the complexity of Algorithm 9 is equivalent to the complexity of

[1] To compute the values in the columns FWT and NLP+FPE we tested 15000 random Boolean functions from $n = 4$, since for $n = 3$ there are only $2^{(2^3)} = 256$ Boolean functions.

n	$\log_2\left[\frac{(n+1)2^{n+1}}{n2^n}\right]$	FWT	NLP+FPE	GB on $\mathbb{F}_p$	GB on $\mathbb{Q}$	GB on $\mathbb{F}$
2-3	1.53	-	-	1.45	1.86	2.50
3-4	1.31	-	-	1.88	2.27	7.51
4-5	1.22	0.90	1.02	2.33	2.91	-
5-6	1.17	0.98	1.09	2.64	3.23	-
6-7	1.14	1.01	1.13	2.76	4.29	-
7-8	1.12	1.22	1.07	3.24	-	-
8-9	1.11	0.95	1.17	3.48	-	-
9-10	1.09	1.25	1.07	-	-	-
10-11	1.09	1.07	1.11	-	-	-

Table 8.6: Experimental comparisons of the coefficients of growth of the analyzed algorithms.

solving a polynomial system of at most $\binom{2^n}{t}$ equations of degree d (where $1 < d \leq t$) in $n+1$ variables over the field $\mathbb{F}$. This method becomes almost impractical for $n = 5$. We recall that $t \leq 2^{n-1} - 2^{\frac{n}{2}-1}$ (see Equation 3.3).

The complexity of Algorithm 10 is equivalent to the complexity of solving a polynomial system of only $n+1$ field equations plus one single polynomial $\mathfrak{n}_f$ of degree at most $n+1$ in $n+1$ variables over the field $\mathbb{Q}$ (or over a prime field $\mathbb{F}_p$) with coefficients of size less then or equal to 2^n.
As shown in Table 8.6, solving the system by computing its Gröbner basis over a prime field $\mathbb{F}_p$ with $p \sim 2^n$ is much faster than computing the same base over $\mathbb{Q}$. It may be investigated if there are better size for the prime p, or even faster specialized algorithms to solve the system.

8.7.3 Algorithm 11

Theorem 8.7.1. *Algorithm 11 returns the nonlinearity of a Boolean function f given as evaluation vector, with n variables in*

$$O(n2^n)$$

integers operations (sums and doublings). The big O constant is 2

Proof. Algorithm 11 can be divided in three main steps:

1. Calculation of the nonlinearity polynomial $\mathfrak{n}_f$. This step, as shown in Theorem 8.6.1, requires $O(n2^n)$ (with big O constant 3/2) integer operations and $O(2^n)$ memory .

2. Evaluation of the nonlinearity polynomial $\mathfrak{n}_f$. This step can be performed using fast Möbius transform in $O(n2^n)$ (with big O constant $1/2$) integer sums and $O(2^n)$ memory.

3. Computation of the minimum $\mathfrak{n}_f(a)$ with $a \in \mathbb{Z}^{n+1}$. This step requires no more than $O(2^n)$ checks.

The overall complexity is then $O(n2^n)$ (with big O constant $c = 3/2 + 1/2 = 2$) integer operations and $O(2^n)$ memory. □

Part IV

MAGMA code

Functions for Part II

9.1 Nordstrom-Robinson code

The following MAGMA code defines a function to generate the Nordstrom-Robinson code as a binary matrix.

```
NordstromRobinsonCode := function()
// from:
// Huffman-Pless
// "Fundamentals of Error Correcting Codes"
// 2.3.4 - The Nordstrom-Robinson code
//
// The existence of the Nordstrom-Robinson code shows that
// A_2(16,6) = 256.
//
// The command:
// > NordstromRobinsonCode() ;
// returns a matrix whose rows are the codeword of the
// Nordstrom-Robinson code.

local C ;    // Extended Golay code
local G ;    // Generator matrix of C
local v ;    // vector of 24 bits
local CT ;   // subcode of C of 32 codewords with 0 in the first 8 components
local c ;    // list of 8 special codewords to create the cosets of CT
local CC ;   // list of 8 cosets of CT
local N ;    // concatenation of the cosets CC[i] (256 codewords of length 24)
local N16 ;  // Nordstrom-Robinson code:
             // - punturing of N in the first 8 components
             // - 256 codewords
             // - length 16
             // - distance 6

  ////////
  // STEP 1 - Let C be the [24, 12, 8] extended binary Golay code
  ////////

  C := GolayCode(GF(2),true) ;

  ////////
  // STEP 2 - Let C be the [24, 12, 8] extended binary Golay code
  //          chosen to contain the weight 8 codeword c = 11...100...0
  ////////

  G := GeneratorMatrix(C) ;

  G := SwapColumns(G,2,13) ;
  G := SwapColumns(G,3,15) ;
```

```
43  G := SwapColumns(G,4,17) ;
44  G := SwapColumns(G,5,18) ;
45  G := SwapColumns(G,6,19) ;
46  G := SwapColumns(G,7,23) ;
47  G := SwapColumns(G,8,24) ;
48
49  C := LinearCode(G) ;
50  v := Vector(GF(2),[1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]) ;
51
52  /////////
53  // STEP 3 - Let C(T) be the subcode of C which is zero on T
54  //          (T is the set consisting of the first eight coordinates)
55  /////////
56
57  CT := [] ;
58  i := 1 ;
59  for c in C do
60    if c[1] eq 0 and
61       c[2] eq 0 and
62       c[3] eq 0 and
63       c[4] eq 0 and
64       c[5] eq 0 and
65       c[6] eq 0 and
66       c[7] eq 0 and
67       c[8] eq 0 then
68      CT[i] := c ;
69      i := i + 1 ;
70    end if ;
71  end for ;
72
73  /////////
74  // STEP 4 - construct c[i] in C
75  //        - c[0] = (0 ... 0 )
76  //        - for 1 <= i <= 7 let
77  //          c[i] = a codeword of C with zeros in the first eight coordinates
78  //                 except coordinate i and coordinate 8
79  /////////
80
81  c := [] ;
82  for i in [1..7] do
83    for x in C do
84      if x[i] eq 1 and
85         x[8] eq 1 and
86         IntegerRing()!(x[1]) +
87         IntegerRing()!(x[2]) +
88         IntegerRing()!(x[3]) +
89         IntegerRing()!(x[4]) +
90         IntegerRing()!(x[5]) +
91         IntegerRing()!(x[6]) +
92         IntegerRing()!(x[7]) +
93         IntegerRing()!(x[8]) eq 2 then
94        c[i] := x ;
95      end if ;
96    end for ;
97  end for ;
98
99  c[8] := ZeroMatrix(GF(2),1,24)[1] ;
100
101 /////////
102 // STEP 5 - let CC[j] j be the coset c[j] + CT
```

```
  //          of CT in the extended Golay code C
  //          For 0 <= j <= 7
  /////////

  CC := [] ;
  for j in [1..8] do
    CC[j] := [] ;
    for i in [1..#CT] do
      CC[j][i] := CT[i] + c[j] ;
    end for ;
  end for ;

  /////////
  // STEP 6 -  Let N be the union of the eight cosets CC[1], ..., CC[8]
  /////////

  N := CC[1] cat CC[2] cat CC[3] cat CC[4] cat
       CC[5] cat CC[6] cat CC[7] cat CC[8] ;

  /////////
  // STEP 7 - The Nordstrom-Robinson code N16
  //          is the code obtained by puncturing N on T
  //          (set consisting of the first eight coordinates)
  /////////

  N16 := [] ;
  for i in [1..#N] do
    N16[i] := ZeroMatrix(GF(2),1,16)[1] ;
    for j in [1..Ncols(N[i])-8] do
      N16[i][j] := N[i][j+8] ;
    end for ;
  end for ;

  return Matrix(N16) ;
end function ;
```

9.2 Bound $\mathcal{A}$, $\mathcal{B}$

In this section we provide the code to compute Bound $\mathcal{A}$ and $\mathcal{B}$, and all the MAGMA functions used to obtain the results in Section 6.

9.2.1 The Johnson bound

We have implemented our own version of the Johnson bound, since the one provided by MAGMA was somehow incomplete.

```
RR := function(m,r,l)
// RR is called by R, which returns a bound for R(m,r,l)
// using Johnson's techniques,
// ("A new upper bound for error-correcting codes",
// Selmer M. Johnson, 1962, Ire Transactions On Information Theory).
// Recall that R(m,r,l) = A_2(m,2r-2l,r)

```

```
8   local bound, k, t ;
9
10  // Check parameters m, r
11  if (m lt 1) or (not IsIntegral(m)) then // m >= 1
12    printf "Error! Parameter (1) must be an integer greater than or equal to 1\n"
      ;
13    return -1 ;
14  end if ;
15  if (r lt 0) or r gt m or (not IsIntegral(r) ) then // 0 <= r <= m
16    printf "Error! Parameter (2) must be an integer in the range [0 .. parameter
      (1) ]\n" ;
17    return -1 ;
18  end if;
19  if l lt 0 or l gt m then // 0 <= l <= m
20    printf "Error! Parameter (3) must be an integer in the range [0 .. parameter
      (1) ]\n" ;
21    return -1 ;
22  end if ;
23
24  // Border line cases
25  if r eq m and l eq m then // r = m, l = m
26    return 2 ;
27  end if ;
28  if r eq m and l lt m then // r = m, l < m
29    return 1 ;
30  end if ;
31  if r eq 0 then // r = 0
32    return 1 ;
33  end if ;
34
35  bound := -1 ;
36  if r^2 - m*l gt 0 then // we can apply R(m,r,l) <= Floor( m(r-l) / (r^2-ml) )
37    if l gt 0 then
38      bound := Min( m*(r-l) div (r^2-m*l) , Floor(m/r * $$(m-1,r-1,l-1) ) ) ;
39      // sometimes one more reduction returns a better lower bound
40    else // if l = 0 we can not check further
41      bound := m*(r-l) div (r^2-m*l) ;
42    end if ;
43  else // we can apply R(m,r,l) <= Floor( m/r * R(m-1,r-1,l-1) ) until l = 0
44    bound := Floor( m/r * $$(m-1,r-1,l-1) ) ;
45  end if ;
46
47  // search for the best R such that R(R-1)l >= (m-t)k^2 + t(k+1)^2 - rR,
48  // where R is the variable bound
49  k := r*bound div m ;
50  t := r*bound - m*k ;
51  while ( bound*(bound-1)*l lt (m-t)*k^2 + t*(k+1)^2 - r*bound ) do
52    //[bound,bound*(bound-1)*l , (m-t)*k^2 + t*(k+1)^2 - r*bound ] ;
53    bound := bound - 1 ;
54    k := Floor(r*bound/m) ;
55    t := r*bound - m*k ;
56  end while ;
57
58  return bound ;
59end function ;
60
61////////////////////////////////////////////////////////////////
62
63R := function(m,r,l)
64// R returns a bound for R(m,r,l) using Johnson's techniques,
```

```
65 // ("A new upper bound for error-correcting codes",
66 // Selmer M. Johnson, 1962, Ire Transactions On Information Theory).
67 // Recall that R(m,r,l) = A_2(m,2r-2l,r)
68 //
69 // INPUT:
70 //   - integer m  s.t. m >= 1
71 //   - integer r  s.t. 0 <= r <= m
72 //   - integer l  s.t. 0 <= l <= m
73 // OUTPUT:
74 //   - Johnson Bound for R(n,d,w)
75 //
76 // Calls function RR()
77 // Let R(m,r,l) be the maximum number of vectors of size m and weigth r
78 // such that the inner product of any pair of row vectors
79 // is less than or equal to l.
80 // Compute R(m,r,l) using bounds from Johnson 1963.
81 // Since R(m,r,l) = R(m,m-r,m-2r+l), then R chooses the minimum between them,
82 // i.e. Min( RR(m,r,l) , RR(m,m-r,m-2r+l) ).
83 // Returns -1 in case of bad parameters.
84
85   // Check parameters m, r
86   if (m lt 1) or (not IsIntegral(m)) then // m >= 1
87     printf "Error! Parameter (1) must be an integer greater than or equal to 1\n"
       ;
88     return -1 ;
89   end if ;
90   if (r lt 0) or r gt m or (not IsIntegral(r) ) then // 0 <= r <= m
91     printf "Error! Parameter (2) must be an integer in the range [0 .. parameter
       (1) ]\n" ;
92     return -1 ;
93   end if;
94   if l lt 0 or l gt m then // 0 <= l <= m
95     printf "Error! Parameter (3) must be an integer in the range [0 .. parameter
       (1) ]\n" ;
96     return -1 ;
97   end if ;
98
99   // Border line cases
100  if r eq m and l eq m then // r = m, l = m
101    return 2 ;
102  end if ;
103  if r eq m and l lt m then // r = m, l < m
104    return 1 ;
105  end if ;
106  if r eq 0 then // r = 0
107    return 1 ;
108  end if ;
109
110  //Return Min[ RR(m,r,l) , RR(m,m-r,m-2r+l) ] ;
111  if m-2*r+l lt 0 then
112    return RR(m,r,l) ;
113  end if;
114  return Min( RR(m,r,l) , RR(m,m-r,m-2*r+l) ) ;
115 end function ;
116
117 ////////////////////////////////////////////////////////////
118
119 JohnsonBound_2 := function (n,d)
120 // Compute Johnson Bound using algorithm from Johnson's Article 1962,
121 // ("A new upper bound for error-correcting codes",
```

```
122 // Selmer M. Johnson, 1962, Ire Transactions On Information Theory)
123 // Returns -1 in case of bad parameters
124
125   local e,denom1,denom2 ;
126
127   // Check parameters n, d
128   if (n lt 1) or (not IsIntegral(n)) then
129     printf "Error! Parameter (2) must be an integer greater than or equal to 1\n"
      ;
130     return -1 ;
131   end if ;
132   if (d lt 1) or (d gt n) or (not IsIntegral(d)) then
133     printf "Error! Parameter (3) must be an integer in the range [1 .. parameter
      (2) ]\n" ;
134     return -1 ;
135   end if ;
136   if IsEven(d) then // if d is even A_2(n,d) = A_2(n-1,d-1)
137     return $$(n-1,d-1) ;
138   end if ;
139   e := (d-1) div 2 ;
140
141   // Border line cases
142   if (d eq 1) then // If d = 1 return the vector space cardinality
143     return 2^n;
144   end if ;
145
146   // Choose the minimum from the two formula where the following terms are
       replaced:
147   // [n/(e+1)]  <-->  1 + (d+1  e+1)R(n,d+1,e+1)/( (n  e+1)-(d  e)R(n,d,e) )
148   denom1:= &+[Binomial(n,i): i in [0 .. e]] +
149             ( Binomial(n,e+1) - Binomial(d,e) * R(n,d,e) ) /
150            Floor(n/(e+1) ) ;
151   if ((Binomial(n,e+1)-Binomial(d,e)*R(n,d,e)) gt 0) then
152      denom2:=   &+[Binomial(n,i):i in [0..e]] +
153                 ( Binomial(n,e+1) - Binomial(d,e) * R(n,d,e) ) /
154                 (1+
155                  ( Binomial(d+1,e+1)*R(n,d+1,e+1) /
156                    (Binomial(n,e+1)-Binomial(d,e)*R(n,d,e)) )
157                 )  ;
158     return  Min ( Floor(2^n/denom1) , Floor(2^n/denom2) )  ;
159   else
160     return Floor( 2^n / denom1 );
161   end if;
162
163 end function ;
164
165 ////////////////////////////////////////////////////////////////
166
167 AA_ := function(K,n,d,w)
168
169   local q ;
170
171   q := #K ;
172   // Check parameters
173   if (not IsField(K)) then
174     printf "Error! parameter (1) must be a field\n" ;
175     return -1 ;
176   end if ;
177   if (n lt 1) or (not IsIntegral(n)) then // n >= 1
178     printf "Error! parameter (2) must be an integer greater than 1\n" ;
```

```
179     return -1 ;
180   end if ;
181   if (d lt 1) or (d gt n) or (not IsIntegral(d)) then // 1 <= d <= n
182     printf "Error! parameter (3) must be an integer in the range [1 .. parameter
      (2) ]\n" ;
183     return -1 ;
184   end if ;
185   if w lt 0 or w gt n or (not IsIntegral(w)) then // 0 <= w <= n
186     printf "Error! parameter (4) must be an integer in the range [1 .. parameter
      (2) ]\n" ;
187     return -1 ;
188   end if ;
189
190   // Border line cases
191   if w eq 0 then // w = 0
192    return 1 ;
193   end if ;
194   if n eq 1 or (d eq n and w eq n) then
195     return q - 1 ;
196   end if ;
197
198   //compute A_q(n,d,w)
199   if d gt 2*w then   // d > 2wn*d*(q-1) div (q*w^2-2*(q-1)*n*w+n*d*(q-1))
200     return 1 ;
201   end if ;
202   // check if Restricted Johnson Bound can be used
203   if ( q*w^2 - 2*(q-1)*n*w + n*d*(q-1) gt 0 ) then
204     if w gt 0 and n gt d+1 then // use that A_q(n,d,w) <= n*(q-1)/w *
      A_q(n-1,d-w-1)
205       return Min( n*d*(q-1) div (q*w^2-2*(q-1)*n*w+n*d*(q-1)) , n*(q-1) *
      $$(K,n-1,d,w-1) div w );
206     else
207       return n*d*(q-1) div (q*w^2-2*(q-1)*n*w+n*d*(q-1)) ;
208     end if ;
209   else // use that A_q(n,d,w) <= n*(q-1)/w * A_q(n-1,d-w-1)
210     return n*(q-1) * $$(K,n-1,d,w-1) div w ;
211   end if ;
212
213   return 0 ;
214 end function ;
215
216 ////////////////////////////////////////////////////////////////
217
218 A_ := function(K,n,d,w)
219 // INPUT:
220 //   - field characteristic q, must be a prime power
221 //   - integer n s.t. n >= 1
222 //   - integer d s.t. 1 <= d <= n
223 //   - integer w s.t. 0 <= w <= n
224 // OUTPUT:
225 //   - Johnson Bound for A_q(n,d,w)
226 //
227 // Calls function AA_()
228 // Compute A_q(n,d,w) using bounds from Huffman-Pless 2003
229 // If q = 2, A_q(n,d,w) = A_q(n,d,n-w), so A_ chooses the minimum between them,
      i.e. Min( AA_q(n,d,w) , AA_q(n,d,n-w) )
230
231   local q ;
232
233   q := #K ; // cardinality of the field
```

```
234
235   // Check parameters
236   if (not IsField(K)) then
237     printf "Error! parameter (1) must be a field\n" ;
238     return -1 ;
239   end if ;
240   if (n lt 1) or (not IsIntegral(n)) then // n >= 1
241     printf "Error! parameter (2) must be an integer greater than 1\n" ;
242     return -1 ;
243   end if ;
244   if (d lt 1) or (d gt n) or (not IsIntegral(d)) then // 1 <= d <= n
245     printf "Error! parameter (3) must be an integer in the range [1 .. parameter
      (2) ]\n" ;
246     return -1 ;
247   end if ;
248   if w lt 0 or w gt n or (not IsIntegral(w)) then // 0 <= w <= n
249     printf "Error! parameter (4) must be an integer in the range [1 .. parameter
      (2) ]\n" ;
250     return -1 ;
251   end if ;
252
253   q := #K ;
254
255   // Border line cases
256   if w eq 0 then // w = 0
257    return 1 ;
258   end if ;
259   if n eq 1 or (d eq n and w eq n) then
260     return q - 1 ;
261   end if ;
262
263   if q eq 2 then
264     return Min( AA_(K,n,d,w) , AA_(K,n,d,n-w) ) ;
265   else
266     return AA_(K,n,d,w) ;
267   end if ;
268
269 end function ;
270
271 ////////////////////////////////////////////////////////////
272
273 /*
274 Important note:
275 the formula implemented is taken from the original article
276 by Johnson
277 ("A new upper bound for error-correcting codes",
278 Selmer M. Johnson, 1962, Ire Transactions On Information Theory)
279 in the binary case, and from Huffman-Pless in the q_ary case
280 ("Fundamentals of Error Correcting Codes", W. Cary Huffman and Vera Pless,
281 2003, Cambridge University Press).
282 The bound strictly depends from the value A_q(n,d,w),
283 which is the maximum number of codewords for a q_ary code, length n, distance d,
284 and whose words have all weight w.
285 Since this value cannot be computed explicitly,
286 in this implementation A_q(n,d,w) is only bounded following
287 the techniques showed in the mentioned papaers.
288 Thus it is possible to obtain better values using the Johnson Bound
289 if the value A_q(n,d,w) is known or better bounded.
290 */
291 JohnsonBound_ := function(K,n,d)
```

```
292 // INPUT:
293 //   - field K
294 //   - integer n s.t. n >= 1
295 //   - integer d s.t. 1 <= d <= n
296 // OUTPUT:
297 //   - Johnson Bound for A_q(n,d)
298 //
299 // Calls function A_() which calls function AA_()
300 // Return the Johnson upper bound for the cardinality of a largest q_ary code
301 // of length n and minimum distance d.
302
303   local q, t, s, k , A ;
304
305   // Check parameters
306   if (not IsField(K)) then
307     printf "Error! parameter (1) must be a field\n" ;
308     return -1 ;
309   end if ;
310   if (n lt 1) or (not IsIntegral(n)) then
311     printf "Error! parameter (2) must be an integer greater than 1\n" ;
312     return -1 ;
313   end if ;
314   if (d lt 1) or (d gt n) or (not IsIntegral(d)) then
315     printf "Error! parameter (3) must be an integer in the range [1 .. parameter
      (2) ]\n" ;
316     return -1 ;
317   end if ;
318
319   q := #K ;
320
321   // Border line cases
322   if n eq 1 or d eq n then
323     return q ;
324   end if ;
325   if (d eq 1) then // If d = 1 return the vector space cardinality
326     return q^n;
327   end if ;
328
329   if IsEven(d) then // A_q(n,d) <= A_q(n-1,d-1)
330     return $$(K,n-1,d-1) ;
331   end if ;
332
333   k := 0 ;
334
335   t := (d-1) div 2 ;
336
337   //compute Sum_0^t (n i)*(q-1)^i
338   s := &+[Binomial(n,i)*(q-1)^i: i in [0 .. t]] ;
339
340   // UNCOMMENT the following lines (if statement)
341   // if you want to call the function JohnsonBound_2 when q = 2
342   if q eq 2 then //compute Johnson bound for A_2(n,d)
343       A := JohnsonBound_2(n,d) ;
344   else //compute Johnson bound for A_q(n,d) when q > 2
345     A := Floor( q^n / ( s + ( Binomial(n,t+1)*(q-1)^(t+1) -
      Binomial(d,t)*A_(K,n,d,d) ) / A_(K,n,d,t+1) ) ) ;
346   end if ;
347
348   return A ;
349 end function ;
```

9.2.2 The Linear Programmin bound

Here we provide a basic implementation of the Linear Programming bound, as presented in [HP03].

```
KrawtchouckPolynomial := function(q,n,k,x)
// returns the Krawtchouck polynomial in the variable x
// 0 <= k <= n
//

  return &+[(-1)^j * (q-1)^(k-j) * Binomial(x,j) * Binomial(n-x,k-j) : j in
    [0..k]] ;
end function ;

////////////////////////////////////////////////////////////

LPB := function(KK,n,d)
// LinearProgrammingBound := function(KK,n,d)
// return the basic version of the linear programming bound as in
// Huffman-Pless, "Fundamental of error correcting codes" - Theorem 2.6.4
// A_q(n,d) <= max { Sum_{w=0}^{n} (B_w) }
//
// we want to maximase the above mentioned sum with the following costraints:
// * B0 = 1
// * B1, ..., B(d-1) = 0
// * Bd, ..., Bn >= 0
// * Sum_w=0^n Bw * K(q,n,k,w) >= 0 ,
//     for 1 <= k <= n
//     where K is the krawtchouck polynomial
//
// furthermore, in the binary case and if d is even, then
// * Bw = 0 for all odd w
// * Bn <= 1
// * Sum_w=0^n Bw * K(q,n,k,w) >= 0 ,
//     for 1 <= k <= Floor(n/2)
//

local q ;
local R ;
local i ;
local L ; // max { Sum_{w=0}^{n} (B_w) }
local nv ; // number of variables
local nc ; // number of costraints
local lhs ; // nc x nv matrix,
            // representing the left-hand-side coeffs of the mc constraints
local rhs ; // nc x 1 matrix over the same ring as LHS,
            // representing the right-hand-side values of the mc constraints
local rel ; // nc x 1 matrix over the same ring as LHS,
            // representing the relations for each constraint, with:
            // *  1 for >=
            // *  0 for =
            // * -1 for <=
local obj ; // 1 x nv matrix over the same ring as LHS,
            // representing the coeffs of the objective function to be optimised
```

```
49
50    q := #KK ;
51    R := RealField() ;
52
53    if q eq 2 and IsEven(d) then
54      nv := Ceiling((n-d+1)/2) ; // would not work if d was odd
55      nc := Floor(n/2) ;
56
57      lhs := Matrix(R, nc, nv, []);
58      for k in [1..nc] do
59        i := 1 ;
60        for w := d to n by 2 do
61          lhs[k][i] := KrawtchouckPolynomial(q,n,k,w) ;
62          i := i + 1 ;
63        end for ;
64      end for ;
65
66    else
67      nv := n-d+1 ; // the nonzero variables are Bd, B(d+1), ..., Bn
68      nc := n ;
69
70      lhs := Matrix(R, nc, nv, []);
71      for k in [1..nc] do
72        i := 1 ;
73        for w := d to n do
74          lhs[k][i] := KrawtchouckPolynomial(q,n,k,w) ;
75          i := i + 1 ;
76        end for ;
77      end for ;
78    end if ;
79
80    rhs := Matrix(R, nc, 1,  [-KrawtchouckPolynomial(q,n,k,0) : k in [1..nc]]) ;
81    rel := Matrix(R, nc, 1,  [1 : k in [1..nc]] ) ;
82    obj := Matrix(R, 1, nv,  [1 : w in [1..nv]]) ;
83
84    L := MaximalSolution(lhs, rel, rhs, obj);
85
86    return Floor(1 + &+[L[1][i] : i in [1..Ncols(L)]]) ;
87
88  end function ;
```

9.2.3 The best known nonlinear upper bound

Here we provide a function which computes the best nonlinear upper bound between the previously implemented upper bounds and those implemented in MAGMA.

```
1  Ball := function(KK,r,n)
2  //   returns the set B_q(r,n),
3  //   which is the set of vectors over the field KK, of length n and weight less
       than or equal to r
4
5    local B ;
6
7    // check parameters
8    if not IsField(KK) then
9      printf "Error! Parameter (1) must be a field.\n" ;
```

```
    return -1 ;
  end if ;
  if not IsIntegral(n) or n lt 1 then
    printf "Error! Parameter (2) must be an integer greater than parameter 0.\n" ;
    return -1 ;
  end if ;
  if not IsIntegral(r) or r gt n or r lt 0 then
    printf "Error! Parameter (3) must be an integer between 0 and parameter
    (3).\n" ;
    return -1 ;
  end if ;

  B := {} ;
  V := VectorSpace(KK,n) ;
  for v in V do
    if Weight(v) le r then
      B := B join {v} ;
    end if ;
  end for ;
  return B ;
end function ;

////////////////////////////////////////////////////////////////

BallSize := function(KK,r,n)
  return &+[Binomial(n,j)*(#KK-1)^j: j in [0 .. r]] ;
end function ;

////////////////////////////////////////////////////////////////

BestKnownNonlinearUpperBound_ := function(KK,n,d)
// Compute the best bound for A_q(n,d)

  local A, q, MM ;
  local plot ; // trace variables: they keep track of what is being used

  plot := false ;

  // check parameters
  if not IsField(KK) then
    printf "Function BestKnownNonlinearUpperBound_\n" ;
    printf "Error! Parameter (1) must be a field.\n" ;
    return -1, plot ;
  end if ;
  if not IsIntegral(d) or d lt 1 then
    printf "Function BestKnownNonlinearUpperBound_\n" ;
    printf "Error! Parameter (3) must be an integer greater than or equal to
    1.\n" ;
    return -1, plot ;
  end if ;
  if not IsIntegral(n) or n lt d then
    printf "Function BestKnownNonlinearUpperBound_\n" ;
    printf "Error! Parameter (2) must be an integer greater than or equal
    parameter (3).\n" ;
    return -1, plot ;
  end if ;

  q := Characteristic(KK)^Degree(KK) ;
  A := q^n ;

```

```
67  // Cases q = 2 , d > n , d > 2n/3 , d = 1 , d = 2
68  if q eq 2 then
69    if d gt n then
70      return 1, plot ;
71    end if ;
72    if d gt 2*(n)/3 then
73      return 2, plot ;
74    end if ;
75    if d eq 1 then
76      return 2^(n), plot ;
77    end if ;
78    if d eq 2 then
79      return 2^(n-1), plot ;
80    end if ;
81  end if ;
82
83  //Best known binary bounds from www.win.tue.nl/~aeb/codes/binary-1.html q = 2 ,
      n = 5..28 , d = 3..16
84  if (q eq 2) then
85    MM := Matrix(IntegerRing(),23,7,[
86            4,       2,       1,       1,       1,       1,       1,
87            8,       2,       1,       1,       1,       1,       1,
88           16,       2,       2,       1,       1,       1,       1,
89           20,       4,       2,       1,       1,       1,       1,
90           40,       6,       2,       2,       1,       1,       1,
91           72,      12,       2,       2,       1,       1,       1,
92          144,      24,       4,       2,       2,       1,       1,
93          256,      32,       4,       2,       2,       1,       1,
94          512,      64,       8,       2,       2,       2,       1,
95         1024,     128,      16,       4,       2,       2,       1,
96         2048,     256,      32,       4,       2,       2,       2,
97         3276,     340,      36,       6,       2,       2,       2,
98         6552,     673,      72,      10,       4,       2,       2,
99        13104,    1237,     135,      20,       4,       2,       2,
100       26168,    2279,     256,      40,       6,       2,       2,
101       43688,    4096,     512,      47,       8,       4,       2,
102       87333,    6941,    1024,      84,      12,       4,       2,
103      172361,   13674,    2048,     150,      24,       4,       2,
104      344308,   24106,    4096,     268,      48,       6,       4,
105      599184,   47538,    5421,     466,      55,       8,       4,
106     1198368,   84260,    9672,     836,      96,      14,       4,
107     2396736,  157285,   17768,    1585,     169,      28,       6,
108     4792950,  291269,   32151,    3170,     288,      56,       8
109   ]);
110   if (n le 27) and (n ge 5) and (d ge 3) and (d le 16) then
111     if IsEven(d) then
112       A := MM[n-5,(d-1) div 2] ;
113     else
114       A := MM[n+1-5,(d-1) div 2] ;
115     end if;
116   end if ;
117 end if ;
118
119 // if possible use Plotkin Bound, and return it, since all other bounds are
      worse
120 if (1-1/q)*n lt (d) then
121   plot := true ;
122   return Min(A,PlotkinBound(KK,n,d)), plot ;
123 end if ;
124
```

```
  A := Min({ A,
             // LPB(KK,n,d) ,             // use our implementation
             // LPB often returns "Numerical instability errors..."
             JohnsonBound_(KK,n,d) ,  // use our implementation
             SpherePackingBound(KK,n,d) ,
             // LevenshteinBound(KK,n,d) , // very slow
             // GriesmerBound(KK,n,d) , // it is only for linear codes
             EliasBound(KK,n,d) ,
             SingletonBound(KK,n,d)
           }) ;
  return A, plot ;
end function ;
```

9.2.4 Bound $\mathcal{B}$

```
BoundB := function(KK,n,d)
// Bound B from Bellini-Guerrini-Sala Article
// return
// * the Bound B and five parameters (i,s1,s2,s3,A)
//   used during the computations
// * -1, in case of error
// * -2, in case the bound does not apply

  local s1, s2, s3 ;
  local max_i, min_i ;
  local q ;
  local plot ;

  plot := false ;

  // check parameters
  if not IsField(KK) then
    printf "Function BoundB\n" ;
    printf "Error! Parameter (1) must be a field.\n" ;
    return -1, -1,-1,-1,-1,-1,plot ;
  end if ;
  if not IsIntegral(d) or d lt 2 then
    printf "Function BoundB\n" ;
    printf "Error! Parameter (3) must be an integer greater than 1.\n" ;
    return -1, -1,-1,-1,-1,-1,plot ;
  end if ;
  if not IsIntegral(n) or n lt d then
    printf "Function BoundB\n" ;
    printf "Error! Parameter (2) must be an integer greater than or equal
    parameter (3).\n" ;
    return -1, -1,-1,-1,-1,-1,plot ;
  end if ;

  // find the largest k satisfying |B(i,k)| <= A_q(n-k,d-2i) -
    |B(i,n-k)|/|B(d-2i-1,n-k)|
  q := Characteristic(KK)^Degree(KK) ;
  t := Floor((d-1)/2);

  for k in [1..n-d+1] do
    max_i := Min({ t , k }) ; // by hypothesis
```

```
    min_i := Max({ 0 , Ceiling((d-n+k)/2) }) ; // so that A_q(n-k,d-2i) can be
    computed

    for i in [min_i..max_i] do // note that i <= (d-1)/2, since the bound works
    for d<k
      // Compute |B(i,k)|
      s1 := &+[Binomial(k,j)*(q-1)^j: j in [0 .. i]] ;
      // Compute |B(i,n-k)|
      s2 := &+[Binomial(n-k,j)*(q-1)^j: j in [0 .. i]] ;
      // But we need to remove the zero vector
      s1 := s1 - 1 ;
      // Compute |B(d-2i-1,n-k)|
      s3 := &+[Binomial(n-k,j)*(q-1)^j: j in [0 .. d-2*i-1]] ;

      // Compute the best bound for A_q(n-k,d-2i)
      A,plot := BestKnownNonlinearUpperBound_(KK,n-k,d-2*i) ;
      // Check the inequality |B(i,k)| <= A_q(n-k,d-2i) -
    |B(i,n-k)|/|B(d-2i-1,n-k)|
      if s1 gt (A - Floor(s2/s3) + 0 ) then //
        return /*k - 1*/ q^(k-1), i,s1,s2,s3,A,plot  ;
      end if ;
    end for ;
  end for ;
  return q^(n-d+1), -2,-2,-2,-2,-2,plot ;
end function;
```

9.2.5 Bound $\mathcal{A}$

```
BoundA := function(KK,n,d)
// Bound A from Bellini-Guerrini-Sala Article
// return the Bound A

local min ;
local A ;
local q ;

  q := #KK ;
  min := q^n ;
  for t in [1..n-d] do
    for r in [0..Minimum({t,Floor((d-1)/2)})] do
      //r,t,n-t,d-2*r-1 ;
      A := q^t * (BestKnownNonlinearUpperBound_(KK,n-t,d-2*r) -
                  BallSize(KK,r,n-t)/BallSize(KK,d-2*r-1,n-t) + 1)
           / BallSize(KK,r,t) ;
      if A lt min then
        min := A ;
      end if ;
    end for ;
  end for ;

  if Floor(min) lt BestKnownNonlinearUpperBound_(KK,n,d) then
    "check these values..." ;
    "n,d = ", n,d ;
  end if ;
  return Floor(min) ;
end function ;
```

9.2.6 Comparison with known bounds

The following code has been used to compute the results in Tables 6.1 and 6.2, typing the command

```
time F,P,n := Percentage(2,29,3,100) ;
// F are the frequencies
// P are the percentage
// n is the total number of checked cases
```

and reading the results from the file *StatisticsAllBounds.txt*.
The computation took about 108593 seconds.

```
NextPrimePower := function (n)
// return the next prime power greater than or equal to n
  local m ;

  m := n ;
  if m lt 1 then
    return 2 ;
  end if ;
  m := m + 1 ;
  while IsPrimePower(m) eq false do
    m := m + 1 ;
  end while ;
  return m ;
end function ;

////////////////////////////////////////////////////////////

countnumberofcases := function(minq,maxq,n1,n2)
// returns 0 if minq=maxq and they are not prime powers
  local c ; // counter

  // check parameters
  if (minq gt maxq) or (n1 gt n2) or (n1 lt 3) then
    "ERROR!!" ;
    "First parameter must be less than second parameter" ;
    "Third parameter must be less than forth parameter" ;
    "Third parameter must be greater than 2" ;
  end if ;

  //check minq is a prime power
  if (minq eq 1) or (not IsPrimePower(minq)) then
    q := NextPrimePower(minq) ;
  else
    q := minq ;
  end if ;

  c := 0 ;
  while q le maxq do
   KK := GF(q) ;
```

```
   for n in [n1..n2] do
     for d in [3..n-1] do
       c := c + 1 ;
     end for ;
   end for ;
   q := NextPrimePower(q) ;
  end while ;

  return c ;
end function ;

//////////////////////////////////////////////////////////////

Percentage := function(minq,maxq,n1,n2)
// Builds a list containing the entries of the following table:
//
//      q       |  2   3   4   5   7   8   9  11  13  16  17  19  23  25  27  29
// -------------|-------------------------------------------------------------
// BoundA       |
// BoundB       |
// Griesmer     |
// Johnson      |
// Levenshtein  |
// Elias        |
// Hamming      |
// Singleton    |
// -------------|-------------------------------------------------------------
//
// each entry is the number of times
// the respective bound is the best known upper bound
// the results are printed in the file "StatisticsAllBounds.txt"

  local P ; // matrix of the frequencies
  local PP ; // matrix of the percentages
  local vBound ; // temporary list of all bounds for certain q,n,d
  local bestKB ; // contains the best known bound for certain q,n,d
  local nCases ; // number of cases checked
  local cc ; // number of cases for a certain q

  // check parameters
  if (minq gt maxq) or (n1 gt n2) or (n1 lt 3) then
    "ERROR!!" ;
    "First parameter must be less than second parameter" ;
    "Third parameter must be less than forth parameter" ;
    "Third parameter must be greater than 2" ;
  end if ;

  // check minq is a prime power
  if not IsPrime(minq) then
    q := NextPrimePower(minq) ;
  else
    q := minq ;
  end if ;

  // initialize matrix of the frequencies with all zero entries
  PP := Matrix(IntegerRing(),8,maxq,[]) ; // the matrix contains also
    q=1,6,10,... but this entries remain 0
  // matrix with the percentage
  P := Matrix(RealField(4),8,maxq,[]) ;

```

```
  nCases := 0 ;
  for n in [n1..n2] do
    for d in [3..n-1] do
      q := minq ;
      while q le maxq do
        KK := GF(q) ;
        nCases := nCases + 1 ;

        // insert bounds in a list
        vBound := [  Round(Log(q,BoundA(KK,n,d) ) ),
                     Round(Log(q,BoundB(KK,n,d) ) ) ,
                     //BoundC(KK,n,d) ,
                     Floor(Log(q,JohnsonBound_(KK,n,d) )) ,
                     Floor(Log(q,SpherePackingBound(KK,n,d) )) ,
                     Round(Log(q,GriesmerBound(KK,n,d) )) ,
                     Floor(Log(q,LevenshteinBound(KK,n,d) )) ,
                     Floor(Log(q,EliasBound(KK, n, d) ) ) ,
                     Floor(Log(q,SingletonBound(KK, n, d) ) )
                     // Plotkin wins whenever it can be applied
                  ] ;
        bestKB := Min({x : x in vBound}) ;
        for i in [1..#vBound] do
          // if the i_th bound in vBound is the best (both when it is only one
    both when it draws with other) then increment vBound[i][q]
          if vBound[i] eq bestKB then
            PP[i][q] := PP[i][q] + 1 ;
          end if ;
        end for ;
        q := NextPrimePower(q) ;
      end while ;
    end for ;
    if (n mod 50) eq 0 then
      fprintf "StatisticsAllBounds.txt","up to n = %o\nNumber of checked cases =
    %o\n%o\n", n,nCases,PP ;
    end if ;
  end for ;

  // compute the percentages
  for i in [1..Nrows(PP)] do
    for j in [1..Ncols(PP)] do
      cc := countnumberofcases(j,j,n1,n2) ;
      if cc eq 0 then
        P[i][j] := 0 ;
      else
        P[i][j] := PP[i][j]/cc ;
      end if ;
    end for ;
  end for ;

  fprintf "StatisticsAllBounds.txt","Total number of checked cases =
    %o\nFrequencies:\n%o\n\nPercentage:\n%o", nCases, PP, P ;
  return P, PP, nCases ;
end function ;
```

The following code has been used to compute the results in Tables 6.3 and 6.4, typing the command

```
time L,D := compare_boundB(2,29,3,100) ;
```

and reading the results from the file *NewResultsB.txt*.

```
compare_boundB := function(minq,maxq,n1,n2)
// when bound B beats or ties other bounds it records
// in a list LL a new element of the type:
//   q,n,d,A,B,J,H,G,E,S,L,P,i,delta,Aq,winB
//   win is 1 if bound B beats other bounds, 0 otherwise
//   delta is |B(i,n-k)|/|B(d-2i-1,n-k)|
//   it returns the list containing this records
//              and the list containing the number of wins for each n
//   print the results in the file NewResultsB

  local q, KK, bestBound, A, B, cp, cv, ct, dlimit, boundList, levB ;
  local LL, cLL ; // LL contains all record where Bound B beats other bounds, cLL
    is the counter of the list
  local DD ; // this list contains
  // DD[x,y] contains the number of times Bound B beats other bounds with
    distance y in characteristic x
  local i, s1, s2, s3, Aq, plot, cplot ; // trace variables

  if (minq gt maxq) or (n1 gt n2) or (n1 lt 3) then
    "ERROR!!" ;
    "First parameter must be less than second parameter" ;
    "Third parameter must be less than forth parameter" ;
    "Third parameter must be greater than 2" ;
  end if ;

  LL := [] ;
  DD := [] ;

  if not IsPrimePower(minq) then
    q := NextPrimePower(minq) ;
  else
    q := minq ;
  end if ;

  LL := [] ;
  cLL := 0 ;

  while q le maxq do
   KK := GF(q) ;

   fprintf "NewResultsB" , "\nq = %o\n", q ;
   DD[q] := [] ;
   for ii in [1..n2] do // initialize the list to all zeros
     DD[q][ii] := 0 ;
   end for ;

   cp := 0 ; // counts the number of times BoundB is less than or equal other
    bounds
   cv := 0 ; // counts the number of times BoundB is less than (BEATS) other
    bounds
   ct := 0 ; // counts the total number of d's tested
   cplot := 0 ; // counts how many times Aq(KK,n-k,d-2i) is bounded with plotkin
   cfl := 0 ; // counts how many times s2/s3 = 0
```

```
50
51    for n in [n1..n2] do
52      dlimit := n-1 ; // dlimit is the limiti until values are not obvious
53      if q eq 2 then dlimit := (2*n div 3) ; end if ;
54      for d in [3..dlimit] do
55        ct := ct + 1 ;
56        // to get a direct comparisons with other bounds
57        //       bestBound := BestKnownNonlinearUpperBound_(GF(q),n,d) ;
58        boundList := [
59                       Floor(Log(q,JohnsonBound_(KK,n,d))) ,  //use our
    implementation
60                       Floor(Log(q,SpherePackingBound(KK,n,d))) ,
61                       //LevenshteinBound(KK,n,d) , // very slow, check apart, only
    if needed
62                       Floor(Log(q,GriesmerBound(KK,n,d))) , // it is only for
    linear codes
63                       Floor(Log(q,EliasBound(KK,n,d))) ,
64                       Floor(Log(q,SingletonBound(KK,n,d) ))
65                      ] ;
66        bestBound := Min({x : x in boundList}) ;
67        // to get a direct comparisons with magma best upper bound
68        // bestBound := BDLCUpperBound(GF(q),n,d) ;
69        A := BoundA(KK,n,d) ;
70        A := Round(Log(q,A)) ; // use Round() because Floor() may return a wrong
    result if for example Log(11,11^14) = 13,999999999999..., while it should be
    14
71
72        B,i,s1,s2,s3,Aq,plot := BoundB(KK,n,d) ;
73        B := Round(Log(q,B)) ; // use Round() because Floor() may return a wrong
    result if for example Log(11,11^14) = 13,999999999999..., while it should be
    14
74
75        // to compare the bounds in the linear/systematic case, we must compare
    Floor(Log(q,.))
76        // B := Floor(Log(q,B)) ;
77        // bestBound := Floor(Log(q,bestBound)) ;
78        if B le bestBound then
79          levB := Floor(Log(q,LevenshteinBound(KK,n,d))) ;
80          boundList[#boundList+1] := levB ;
81          bestBound := Min({ bestBound, levB }) ; // check Levenshtein only if
    needed, because it is very slow
82          if (1-1/q)*n lt (d) then
83            v := [q, n, d, A, B,
84                  boundList[1], boundList[2],
85                  boundList[3], boundList[4],
86                  boundList[5], boundList[6],
87                  Floor(Log(q,PlotkinBound(KK,n,d))),
88                  i, Floor(s2/s3),Aq,0] ;
89          else // if plotkin cannot be applied fill its place with n
90            v := [q, n, d, A, B,
91                  boundList[1], boundList[2],
92                  boundList[3], boundList[4],
93                  boundList[5], boundList[6],
94                  n, i, Floor(s2/s3),Aq,0] ;
95          end if ;
96
97          if B eq bestBound then // count ties
98            cp := cp + 1 ;
99            cLL := cLL + 1 ;
100           LL[cLL] := v;
```

```
101
102           // check to be done only for BoundB
103           ////////////////////////////////////////////////////////////////////
104           if plot then // check if plotkin has been used to bound Aq(KK,n-k,d-2i)
105             cplot := cplot + 1 ;
106           end if ;
107           if Floor(s2/s3) eq 0 then // check if s2/s3 = 0
108             cfl := cfl + 1 ;
109           end if ;
110           ////////////////////////////////////////////////////////////////////
111
112         elif B lt bestBound then // count wins
113           cv := cv + 1 ;
114           v[#v] := 1 ;
115           cLL := cLL + 1 ;
116           LL[cLL] := v ;
117
118           // check to be done only for BoundB
119           ////////////////////////////////////////////////////////////////////
120           if plot then // check if plotkin has been used to bound Aq(KK,n-k,d-2i)
121             cplot := cplot + 1 ;
122           end if ;
123           if Floor(s2/s3) eq 0 then // check if s2/s3 = 0
124             cfl := cfl + 1 ;
125           end if ;
126           ////////////////////////////////////////////////////////////////////
127
128           DD[q][d] := DD[q][d] + 1 ;
129         end if ;
130
131       end if ;
132     end for ;
133   end for ;
134   fprintf "NewResultsB" , "Tie = %o over %o --> %o \nWin = %o over %o --> %o
     \nPlotkin used %o times --> %o\ns1/s2 is zero %o times --> %o\n",
135           cp, ct, RealField(4)!(cp/ct*100),
136           cv, ct, RealField(4)!(cv/ct*100),
137           cplot, RealField(4)!(cplot/(cv+cp)*100),
138           cfl, RealField(4)!(cfl/(cv+cp)*100)  ;
139  q := NextPrimePower(q) ;
140  end while ;
141  fprintf "NewResultsB" , "  q, n, d, A, B, J, H, G, E, S, L, P, i, delta, Aq,
      winB \n%o\n%o", LL,DD ;
142  printf "...finished checking!\n" ;
143  return LL, DD ;
144 end function ;
```

Functions for Part III

10.1 Traverso's algorithm

We report here and implentatione of Algorithm 1, and of the functions needed to compute the Gröbner description, the Gröbner and linear representation (see Section 1.4.1).

```
NextConfiguration := function (LL,MAX,MIN)
//
// Example:
// nc := NextConfiguration([0,0,0],[1,2,2],[0,0,0]) ; nc ;
// // [0,0,1] ;
// nc := NextConfiguration([0,0,1],[1,2,2],[0,0,0]) ; nc ;
// // [ 0, 0, 2 ]
// nc := NextConfiguration([0,0,2],[1,2,2],[0,0,0]) ; nc ;
// // [ 0, 1, 0 ]
// ...
// nc := NextConfiguration([1,2,1],[1,2,2],[0,0,0]) ; nc ;
// // [ 1, 2, 2 ]
// nc := NextConfiguration([1,2,2],[1,2,2],[0,0,0]) ; nc ;
// // [ 1, 2, 2 ]
//

local L, i ;
  L := LL ;
  // CHECKS
  if #L ne #MAX then
    return "ERROR! The two list must have the same length!" ;
  end if ;
  for j in [1..#L] do
    if L[j] lt MIN[j] or L[j] gt MAX[j] then
      return "ERROR! The input sequence is out of range!" ;
    end if ;
  end for ;

  // CHECK IF FINISHED
  if MAX eq L then
    return L ;
  // FIND NEW CONFIGURATION
  else
    // find the rightmost element to increase
    i := #L ;
    while L[i] eq MAX[i] do
      i := i - 1 ;
    end while ;
    L[i] := L[i] + 1 ;
    for j in [i+1..#L] do
      L[j] := MIN[j] ;
```

```
    end for ;
  end if ;

  return L ;
end function ;

////////////////////////////////////////////////

HilbertStaircase := function(G)
// G must be a reduced groebner basis
  return [LeadingMonomial(g) : g in G] ;
end function ;

////////////////////////////////////////////////

MonomialsUnderHilbertStaircase := function(G)
// Returns a list containing
// all the monomials under the Hilbert Staircase
// The monomials are in the ring R|G
//
// G must be a reduced groebner basis
// of a finite dimensional ideal!!
//
local HS ; // leading monomials of G
local N ; // monomials under the Hilbert Staircase
local E, temp ;
local R ; // polynomial ring
local RG ; // R/G
local max, ind ;
local extr ;

  R := Parent(G[1]) ;
  RG := quo<R | G> ;
  HS := HilbertStaircase(G) ;
  N := {} ;

  // FIND "SINGLE-VARIABLE" LEADING MONOMIAL
  extr := [0 : i in [1..Rank(R)]] ;
  for m in HS do
    E := Exponents(m) ;
    if #[x : x in E | x ne 0] eq 1 then
      max,ind := Max(E) ;
      extr[ind] := E[ind] ;
    end if ;
  end for ;

  // CREATE HYPER-CUBE
  N := [] ;
  E := [0 : i in [1..Rank(R)]] ;
  Append(~N,E) ;
  repeat
    E := NextConfiguration(E,extr,[0 : i in [1..#E]]) ;
    Append(~N,E) ;
  until E eq extr ;

  // EXCLUDE MONOMIAL OVER THE STAIRCASE
  for m in HS do
    E := Exponents(m) ;
    temp := E ;
    Exclude(~N,temp) ;
```

```
102     repeat
103       temp := NextConfiguration(temp,extr,E) ;
104       Exclude(~N,temp) ;
105     until temp eq extr ;
106   end for ;
107 
108   return Sort([&*[RG.i^e[i] : i in [1..Rank(R)]] : e in N]) ;
109 end function ;
110 
111 ////////////////////////////////////////////////
112 
113 IdealDegree := function(I)
114 // computes the number of elements under the Hilbert Staircase
115 // #N(I)
116 // Definition 27.12.1, "SPES II", Mora
117 
118   if not IsZeroDimensional(I) then
119     "The degree can be computed only for a zero dimensional ideal!!" ;
120     return -1 ;
121   end if ;
122 
123   return #MonomialsUnderHilbertStaircase(Groebner(I)) ;
124 end function ;
125 
126 ////////////////////////////////////////////////
127 
128 GroebnerRepresentation := function(I,Q)
129 // Q = {q1,...,qs}
130 // is a linear indipendent set such that
131 // R[x1,...,xk]/I = Span of Q with respect to K
132 // see def. 29.3.2, "SPES II", Mora
133 //
134 // Example:
135 // K := Rationals() ;
136 // R<x2,x1> := PolynomialRing(K,2,"grevlex") ;
137 // f := [
138 //       x2^3 - x1*x2^2,
139 //       x1^2*x2,
140 //       x1^3 - x2^2 + x1*x2,
141 //       x2^4
142 //      ] ;
143 // I := Ideal(f) ;
144 // Q := MonomialsUnderHilbertStaircase(G) ;
145 // GroebnerRepresentation(I,Q) ;
146 //
147 
148 local K ; // base field
149 local R ; // polynomial ring over K
150 local G ; // groebner basis of I
151 local s ; // number of elements in Q
152 local n ; // number of variables x1,...,xn
153 local M ; // set of n square matrices
154 local Xh_ql ; // RG.h*Q[l]
155 local Mon ; // monomials of Xh_ql
156 local Coeff ; // coefficients of Xh_ql
157 local ind ;
158 
159   R := Parent(I[1]) ;
160   K := BaseRing(R) ;
161 
```

```
  G := GroebnerBasis(I) ;

  RG := quo<R | G> ;

  n := Rank(RG) ;
  s := #Q ;
  M := [] ;
  for h in [1..n] do
    M[h] := Matrix(K,s,s,[]) ;
    for l in [1..s] do
      Xh_ql := RG.h*RG!Q[l] ;
      Mon := Monomials(Xh_ql) ;
      Coeff := Coefficients(Xh_ql) ;
      for j in [1..s] do
        ind := Index(Mon,Q[j]) ;
        if ind ne 0 then
          M[h][l][j] := Coeff[ind] ;
        end if ;
      end for ;
    end for ;
  end for ;

  return Q, M ;
end function ;

////////////////////////////////////////////////

LinearRepresentation := function(I : vect := false)
// A linear representation of an Ideal I
// is a Groebner representation where Q is the set
// of the monomials under the Hilbert Staircase
// EXAMPLE 29.2.1, "SPES II", Mora
// K := Rationals() ;
// R<x2,x1> := PolynomialRing(K,2,"grevlex") ;
// f := [
//       x2^3 - x1*x2^2,
//       x1^2*x2,
//       x1^3 - x2^2 + x1*x2,
//       x2^4
//      ] ;
// I := Ideal(f) ;
// LinearRepresentation(I) ;
//
// if vect = true Q is returned as a vector of vectors and M as a matrix
// if vect = false Q is returned as a vector of monomials and M as a list of
    monomials
//

local K ; // base field
local R ; // polynomial ring over K
local Q ; // R[x1,...,xk]/I = Span of Q with respect to GF(2)
          // i.e. monomials under the Hilbert Staircase
local G ; // groebner basis of I
local s ; // number of elements in Q
local n ; // number of variables x1,...,xn
local M ; // set of n square matrices
local Xh_ql ; // RG.h*Q[l]
local Mon ; // monomials of Xh_ql
local Coeff ; // coefficients of Xh_ql
local ind ;
```

```
221
222   R := Parent(I[1]) ;
223   K := BaseRing(R) ;
224
225   G := GroebnerBasis(I) ;
226   Q := MonomialsUnderHilbertStaircase(G) ;
227
228   RG := quo<R | G> ;
229
230   n := Rank(RG) ;
231   s := #Q ;
232   M := [] ;
233
234   if vect then // VECTORIAL CASE
235     for h in [1..n] do
236       M[h] := Matrix(K,s,s,[]) ;
237       for l in [1..s] do
238         Xh_ql := RG.h*RG!Q[l] ;
239         Mon := Monomials(Xh_ql) ;
240         Coeff := Coefficients(Xh_ql) ;
241         for j in [1..s] do
242           ind := Index(Mon,Q[j]) ;
243           if ind ne 0 then
244             M[h][l][j] := Coeff[ind] ;
245           end if ;
246         end for ;
247       end for ;
248     end for ;
249     Q := [] ;
250     for i in [1..s] do
251       Q[i] := Vector(BaseRing(RG),[0 : j in [1..s]]) ;
252       Q[i][i] := 1 ;
253     end for ;
254   else // POLYNOMIAL CASE
255     for h in [1..n] do
256       M[h] := [] ;
257       for l in [1..s] do
258         M[h][l] := RG.h * Q[l] ;
259       end for ;
260     end for ;
261   end if ;
262
263   return Q, M ;
264 end function ;
265
266 ////////////////////////////////////////////////
267
268 LinearRepresentationPOLY := function(I)
269 // A linear representation of an Ideal I
270 // is a Groebner representation where Q is the set
271 // of the monomials under the Hilbert Staircase
272 // EXAMPLE 29.2.1, "SPES II", Mora
273 // K := Rationals() ;
274 // R<x2,x1> := PolynomialRing(K,2,"grevlex") ;
275 // f := [
276 //       x2^3 - x1*x2^2,
277 //       x1^2*x2,
278 //       x1^3 - x2^2 + x1*x2,
279 //       x2^4
280 //      ] ;
```

```
// I := Ideal(f) ;
// LinearRepresentation(I) ;
//
// if vect = true Q is returned as a vector of vectors and M as a matrix
// if vect = false Q is returned as a vector of monomials and M as a list of
    monomials
//
local R ;// polynomial ring
local A ; // affine algebra R/I
local Q ;

  if I eq [] then
    return [] ;
  end if ;

  R := Parent(I[1]) ;
  A := quo< R | I > ;

  return SetToSequence(MonomialBasis(A)) ;
end function ;

////////////////////////////////////////////////

GroebnerDescription := function(g,Q : vect:=true)
// g must be a polynomial in R
// Q must be the set of the monomials under the Hilbert Staircase
//   where each monomial is in R/G,
//   where G is a Groebner basis
//
// The complexity to compute Groebner Description
// should be
// O(uds^2), where:
// - s is the number of elements in Q
// - d is the degree of g
// - u is the number of monomials of g in R
// The complexity can be reduced to
// O(Hor(f)s^2), where
// - Hor(f) <= ud, is the Horner complexity of f, i.e.
//   the number of + required by the recursive Horner representations
//
// if vect = true the description is given as a vector
// else it is given as a polynomial in the algebra with base Q
//

local R ;
local RG ;
local rem ; // remainder of g mod I
local rem_c ; // coefficients of the remainder
local rem_m ; // monomials of the remainder
local GD ; // groebner description of g with respect to Q

  if g eq 0 then
    return Vector([Parent(Q[1])!0 : i in [1..#Q]]) ;
  end if ;

  R := Parent(g) ;
  RG := Parent(Q[1]) ;

  rem := RG!Evaluate(g,[RG.i : i in [1..Rank(RG)]]) ;
  if not vect then
```

```
340     return rem ;
341   else
342     rem_c := Coefficients(rem) ;
343     rem_m := Monomials(rem) ;
344
345     GD := [Parent(rem_c[1])!0 : i in [1..#Q]] ;
346     for i in [1..#rem_m] do
347       GD[Index(Q,rem_m[i])] := rem_c[i] ;
348     end for ;
349
350     return Vector(GD) ;
351   end if ;
352 end function ;
353
354 ////////////////////////////////////////////////
355
356 Traverso := function( QQ, MM, GD : verb:=true )
357 // from "SPES II", Mora, Fig 29.3, Traverso's Algorithm
358 // Given
359 // - a linear representation (Q,M) of an ideal I
360 // - r groebner descriptions GD = {c_1,...,c_r}
361 //   of r new polynomials not in I
362 // returns the linear representation of an ideal J
363 // where J = I U GD = I U {c_1,...,c_r}
364 // INPUT:
365 // - Q, monomials under the Hilbert Staircase
366 // - M, multiplication tables for each variable
367 // - GD, sequence of r Groebner descriptions
368 //
369 // EXAMPLE:
370 // q := 2 ; k := 2 ;
371 // R := PolynomialRing(GF(q),k,"grevlex") ;
372 // G := [R.i^q-R.i : i in [1..k]] ;
373 // G := GroebnerBasis(G) ;
374 // Q,_ := LinearRepresentation(G : vect := false ) ;
375 // _,M := LinearRepresentation(G : vect := true ) ;
376 // c := Vector(GF(2),[1,0,0,1]) ;
377 // Q1,M1 := TraversoVECT(Q,M,[c]) ;
378 //
379
380 local n ; // number of variables
381 local Q ; //
382 local M ; //
383 local s ; // number of elements in Q
384 local B ; // set of r Groebner descriptions
385 local c ; // single Groebner description
386 local iota ; //
387 local Q_iota, M_iota, c_iota, d_iota ;
388 local B1 ; //
389 local temp ; //
390
391   M := MM ;
392   Q := QQ ;
393   n := #M ;
394   s := #Q ;
395   I := [i : i in [1..s]] ;
396   B := GD ;
397
398   if verb then
399     "------ BEGIN TRAVERSO'S ALGORITHM ------" ;
```

```
400   end if ;
401   while B ne [] do
402
403     c := B[1] ; // or c := Random(B) ; // is there an efficient choice?
404
405     if verb then
406       "-----------------------------------------" ;
407       "-----------------------------------------" ;
408       "B = ", B ;
409       "I = ", I ;
410       "c = ", c ;
411     end if ;
412
413     Exclude(~B,c) ; // remove c from B
414
415     for m in M do
416       temp := c*m ;
417       if Weight(temp) ne 0 and not temp in B then // c*m != 0...0 and c*m not in B
418         Append(~B,temp) ;
419       end if ;
420     end for ;
421
422     iota := Maximum( { j : j in [1..Ncols(c)] | c[j] ne 0 } ) ;
423
424     // UPDATE Q
425     Q_iota := Q[iota] ;
426     Remove(~Q, iota) ;
427     s := #Q ;
428
429     // SAVE iota COLUMNS AND REMOVE THEM FROM M
430     M_iota := [RemoveRow(Submatrix(M[h],1,iota,Nrows(M[h]),1),iota) : h in [1..n]
      ] ;
431     M := [RemoveRowColumn(M[h],iota,iota) : h in [1..n]] ;
432
433     // SAVE iota COORDINATE AND REMOVE IT FROM c
434     c_iota := c[iota] ;
435     c := RemoveColumn(c,iota)[1] ;
436
437     if verb then
438       "---------- B U [c*m : m in M] ----------" ;
439       "B = ", B ;
440       "I = ", I ;
441       "iota = ", iota ;
442       printf "Q[%o] = %o \n",iota,Q_iota ;
443       printf "c[%o]^-1 = %o \n",iota,c_iota^-1 ;
444     end if ;
445
446     // REPLACE Q[iota] IN MULTIPLICATION TABLES
447     for h in [1..n] do
448       for j in [1..Nrows(M[h])] do // for j in I do
449         for l in [1..Ncols(M[h])] do // for l in I do
450           M[h][l][j] := M[h][l][j] - c_iota^-1 * c[j] * M_iota[h][l][1] ;
451         end for ;
452       end for ;
453     end for ;
454
455     if verb then
456       "M = ", M ;
457     end if ;
458
```

```
459     B1 := B ;
460     B := [] ;
461
462     // REPLACE Q[iota] IN GROEBNER DESCRIPTIONS
463     for x in B1 do
464       d := x ;
465       d_iota := d[iota] ;
466       d := RemoveColumn(d,iota)[1] ;
467       for j in [1..s] do
468         d[j] := d[j] - c_iota^-1 * c[j] * d_iota ;
469       end for ;
470       if (Weight(d) ne 0) and (not d in B) then // d != 0...0
471         Append(~B,d) ;
472       end if ;
473     end for ;
474   end while ;
475
476   return Q, M ;
477 end function ;
```

10.2 Basic functions

Here we present the underlying functions needed to compute the minimum weight of a nonlinear code (using the techniques of Section 7), and the nonlinearity of a B.f. (using the techniques of Section 8).

10.2.1 Algebraic and numerical normal form

First, some functions regarding algebraic and numerical normal form are listed.

```
1  CoefficientVectorToPolynomial := function(cc : leastleft:=false) // OK!
2  // given the vector of the coefficients
3  // (most significant on the left if leastleft = false)
4  // returns the polynomial with those coefficients
5  // inverse function of PolynomialCoefficients or ANFCoefficients
6  //
7  // Example:
8  // R := PolynomialRing(GF(2),3) ;
9  // p := R.1*R.2 + R.1 + 1 ;
10 // c := ANFCoefficients(p) ;
11 // CoefficientVectorToPolynomial(c) ;
12 // // $.1*$.2 + $.1 + 1
13 // R := PolynomialRing(Rationals(),3) ;
14 // p := 3*R.1*R.2 + 2*R.1 + 4 ;
15 // c := PolynomialCoefficients(p) ;
16 // CoefficientVectorToPolynomial(c) ;
17 // // 3*$.1*$.2 + 2*$.1 + 4
18 //
19
20 local p ; // polynomial to be returned
21 local K ; // field of the components of c
22 local n ; // number of variables
23 local V ; // vector space over R of dimension n
```

```
local R ; // polynomial ring of p
local j ;

  if leastleft then
    c := Vector(Reverse(ElementToSequence(cc))) ;
  else
    c := cc ;
  end if ;

  K := Parent(c[1]) ;
  n := Integers()!Log(2,Ncols(c)) ;
  V := VectorSpace(GF(2),n) ;
  R := PolynomialRing(K,n) ;

  p := 0 ;
  j := Ncols(c) ;
  for v in V do
    p := p + c[j]*&*[R.i^Integers()!v[n-i+1] : i in [1..n]] ;
    j := j - 1 ;
  end for ;

  return p ;
end function ;

////////////////////////////////////////////////////////////////

ANFCoefficients := function(f : leastleft := false) // OK!
// given a BF over the ring R
// returns the vector of the coefficients of the Algebraic Normal Form of f
//
// Ex:
// if leastleft = false (default) then
// f := b12R.1*R.2 + b1*R.1 + b2*R.2 + b0 ;
// returns:
// c = (b12, b1, b2, b0) ;
// otherwise
// c = (b0, b1, b2, b12) ;
// ATTENTION! The order of the monomials depends on the order defined over R
//            and thus also the order of c!!!
//
// => NOTE: the ordering is the one defined by the function
//          LexPolynomialRing(GF(2),n)
//
// NOTE2: algorithm is "slow". Could be improved.
// f := RandomBooleanPolynomial(15);
// time a := ANFCoefficients(f) ;
// Time: 911.440


local R ; // ring of f of n variables
local n ;
local c ; // vector of coefficients
local mf ; // monomials of f
local mgb ; // all possible monomials in R
local gb ; // generic boolean polynomial

  n := Rank(Parent(f)) ;
  R := Parent(f) ;

  c := Zero(VectorSpace(GF(2),2^n)) ;
```

```
  gb := Evaluate(Generic_boolean_polynomial(R),[1 : i in [1..2^n]] cat [R.i : i
    in [1..n]] ) ;

  mgb := Monomials(gb) ;
  mf := Monomials(f) ;

  for i in [1..#mgb] do
    if mgb[i] in mf then
      c[i] := 1 ;
    end if ;
  end for ;

  if leastleft then
    // to have the least significant coeff (i.e. the constant term) on the left
    return ReverseColumns(c)[1] ;
  else
    // to have the most significant coeff on the left
    return c ;
  end if ;
end function ;

////////////////////////////////////////////////////////////////

AlgebraicNormalForm := function(TT)
// INPUT:
// - TT, sequence of the evalution vector (truth table) of f
// OUTPUT:
// - f, Algebrai Normal Form of the truth table TT
//

local C, f;
local V, n, Q, X, k, pr ;

  n := IntegerRing()!Log(2,#TT) ;
  V := VectorSpace(GF(2),n) ;
  Q := BooleanPolynomialRing(n);
  X := [ Q.i : i in [1..n]] ;

  f := Q!TT[1] ;
  k := 1 ;

  for v in V do
    if Evaluate(f,ElementToSequence(v)) ne TT[k] then
      pr := Q!1 ;
      for j in [1..n] do
        pr := pr * X[j]^(IntegerRing()!v[j]) ;
      end for ;
      f := f + pr ;
    end if ;
    k := k + 1 ;
  end for ;

  return Q!f ;
end function ;

////////////////////////////////////////////////////////////////

Sint := function(S)
// performs the BINary Sum as INTeger Sum:
// EX:
```

```
// a+b                    --> over the binary field is
// a+b-2ab                --> over the integer ring (or the rational field)
// a+b+c                  --> over the binary field is
// a+b+c-2ab-2bc-2ac+4abc --> over the integer ring (or the rational field)
// ...
// To use it:
// Q := PolynomialRing(RationalField(),3+1) ;
// Sint([Q.1,Q.2]) ;
// -2*$.1*$.2 + $.1 + $.2
// Sint([GF(2)!1,1,1]) ;
// 1
// Sint([1,1,1]) ;
// 1

local sum ;
local Si ;

  if #S eq 0 then // zero sequence
    return 0 ;
  end if ;

  if Category(Parent(S[1])) eq Category(GF(2)) then
    Si := [IntegerRing()!S[i] : i in [1..#S]] ;
  else
    Si := [S[i] : i in [1..#S]] ;
  end if ;

  sum := Si[1] ;
  for i in [2..#Si] do
    sum := sum + Si[i] - 2*sum*Si[i] ;
  end for ;

  return sum ;
end function ;

////////////////////////////////////////////////////////////////

NextMonomialOfWeight := function(m,t)
// given a simple t-monomial m in R s.t. m is the product of t variables
// returns the "next" monomial with t variables over R[x1,...,xn]
// following a pre-determined rule which assign:
// i --> x_{i_1}*...*x_{i_t}
//
// RULE:
// - counting from the left, move the first free index to the right
// - when and index is moved to the right (increased)
//   then all previous indexes must be brought
//   to the starting position (leftmost)
//
// To use it:
// R := PolynomialRing(GF(2),5) ;
// NextMonomialOfWeight(R.1*R.2*R.3,3) ;
// $.1*$.2*$.4
//
// if m = 1 return the first monomial of R
// write R!1 to indicate the monomial "1"
//

local ind ; // list of t indexes
local R ;
```

```
local n ;
local counter ;

  R := Parent(m) ;
  n := Rank(R) ;

  // CHECKS
  for i in [1..n] do
    if Degree(m,i) gt 1 then
      "ERROR! The monomial m is not a simple t-monomial!" ;
      return 0 ;
    end if ;
  end for ;

  // CHECK t, THE WEIGHT OF m
  if t ne Degree(m) and m ne 1 then
      "ERROR! The monomial m has degree different from t" ;
      return 0 ;
  end if ;

  // EXTRACT INDEXES
  if m eq 1 then
    return &*[R.(j) : j in [1..t]] ;
  else
    ind := [i : i in [1..n] | Degree(m,i) eq 1] ;
  end if ;

  // FIND NEXT "MONOMIAL"
  if ind[1] eq n-t+1 then
    "ERROR! The monomial inserted is the last of the list!" ;
    return m;
  else
    j := 1 ;
    while j le #ind-1 and ind[j]+1 eq ind[j+1] and ind[j] ne n do
      j := j + 1 ;
    end while ;

    // INCREASE ind[j]
    ind[j] := ind[j] + 1 ;

    // RESET ind[1], ..., ind[j-1]
    for k in [1..j-1] do
      ind[k] := k ;
    end for ;

    return &*[R.(ind[j]) : j in [1..#ind]] ;
  end if ;

end function ;

////////////////////////////////////////////////////////////

NNFfromANF := function(f)
// for polynomials
local R, r, Mf, L, m, Q ;

  R := PolynomialRing(Rationals(),Rank(Parent(f)) ) ;
  r := Rank(R) ;
  Q := quo< R | [R.i^2-R.i : i in [1..r]] > ;

```

```
  Mf := Monomials(f) ;
  L := [] ;
  if R!1 in Mf then
   L := L cat [R!1] ;
  end if ;

  for i in [1..r] do
    m := R!1 ;
    for j in [1..Binomial(r,i)] do
      m := NextMonomialOfWeight(m,i) ;
      if m in Mf then
        L := L cat [m] ;
      end if ;

    end for ;
  end for ;

  return Sint([Q!x : x in L]) ;
end function ;
```

10.2.2 Fast transforms

Now we show how to implement fast Fourier like transforms.

```
FastMobiusTransform := function(c_f:leastleft:=false)
// This function allows to obtain the evaluation of a BF f with n variables
// in only n2^n steps (instead 2^2n).
// It is supposed that coeffs are given from the highest monomial to the lowest.
// Given the vector of coefficients of a BF over the ring R
// returns the evaluation vector of f
// using the fast mobius transform.
// Since this operation is an involution,
// if it is applied to an evaluation vector
// than the vector of coefficients of f is returned.
//
// ATTENTION! The order of the monomials depends on the order defined over the
//            ring of the function f of which c_f are the coefficients...
//            Thus changing the ring of f, the order of the coefficients changes
//            (eventhough the weight is obviously the same)
//            and also the FMT changes (eventhough its weight doesn't)
//
// Reference:
// Cagdas Calik, PhD Thesis, Pag. 9, chap. 2.3
// http://sc.iam.metu.edu.tr/iamWarehouse/iam_Bibliography/
//        web/index.php/attachments/single/219
//
// EX:
// f := Vector(GF(2), [1,0,0,0,0,0,1,0]) ; // i.e. f = xyz+x with "lex"
// FastMobiusTransform(f) ;
// (0 1 0 1 0 1 0 0)
//

local n ; // number of variables of f
local ev_f ;
local lb ; // length of blocks
local nb ; // number of blocks
```

```
33
34    n := IntegerRing()!Log(2,Ncols(c_f)) ;
35    if leastleft then
36      ev_f := Vector(GF(2),ElementToSequence(c_f)) ;
37    else
38      ev_f := Vector(GF(2),Reverse(ElementToSequence(c_f))) ;
39    end if ;
40
41    for i in [1..n] do // n steps, the n_th step returns the evaluation vector
42
43      lb := 2^i ;
44      nb := (2^n) div lb ;
45      for j in [1..nb] do // for each block
46
47        for k in [1..lb div 2] do // for the first half of the block
48          ev_f[k+(j-1)*lb] := ev_f[k+(j-1)*lb] ;
49        end for ;
50
51        for k in [(lb div 2)+1..lb] do // for the second half of the block
52          ev_f[k+(j-1)*lb] := ev_f[k+(j-1)*lb] + ev_f[k+(j-1)*lb-(lb div 2)] ;
53        end for ;
54
55      end for ;
56    end for ;
57
58    return ev_f ;
59  end function ;
60
61  ////////////////////////////////////////////////////////////////
62
63  FastWalshSpectrum := function (TT)
64  // INPUT:
65  // - TT, the polarity truth table (evaluation vector) of f
66  // OUTPUT:
67  // - fws_f, the walsh spectrum of TT
68  //
69  // If f is given by its ANF, the fastest way to compute the Walsh Spectrum is:
70  // 1 - to compute the coefficients vector of the ANF of f,
71  // 2 - than the truth table with Fast Mobius Transform
72  // 3 - get the polarity truth table of f
73  // 4 - than use Fast Walsh Transform (Spectrum)
74  //
75  // i.e., to use it:
76  // R := BooleanPolynomialRing(3) ;
77  // f := R.1*R.2 + R.3 ;
78  // WalshSpectrum(f) ;
79  // [ 0, 0, 0, 0, 4, 4, 4, -4 ]
80  // fmt := FastMobiusTransform(ANFCoefficients(f)) ;
81  // FastWalshSpectrum([1-2*IntegerRing()!fmt[i] : i in [1..Ncols(fmt)]]) ;
82  // [ 0, 0, 0, 0, 4, 4, 4, -4 ]
83  //
84  // Reference:
85  // Cagdas Calik, PhD Thesis, Pag. 11, chap. 2.4
86  // http://sc.iam.metu.edu.tr/iamWarehouse/iam_Bibliography/
87  //        web/index.php/attachments/single/219
88  //
89
90  local n ; // number of variables of f
91  local fws_F ;
92  local lb ; // length of blocks
```

```
93 local nb ; // number of blocks
94 local temp ;
95
96   n := IntegerRing()!Log(2,#TT) ;
97   fws_f := [IntegerRing()!TT[i] : i in [1..#TT]] ;
98   temp := [] ;
99
100  for i in [1..n] do // n steps, the n_th step returns the evaluation vector
101
102    lb := 2^i ;
103    nb := (2^n) div lb ;
104    for j in [1..nb] do // for each block
105
106      for k in [1..lb div 2] do // for the first half of the block
107        temp[k+(j-1)*lb] := fws_f[k+(j-1)*lb] + fws_f[k+(j-1)*lb+(lb div 2)];
108      end for ;
109
110      for k in [(lb div 2)+1..lb] do // for the second half of the block
111        temp[k+(j-1)*lb] := fws_f[k+(j-1)*lb-(lb div 2)] - fws_f[k+(j-1)*lb] ;
112      end for ;
113    end for ;
114    fws_f := temp ;
115  end for ;
116
117  return fws_f ;
118 end function ;
119
120 ////////////////////////////////////////////////////////////
121
122 NonLinearityFWT := function (bf:inc:=false)
123 // computes the non linearity of f using the FAST Walsh trasform
124 // to be improved:
125 // - the function ANFCoefficients could be improved
126 //   thus it is possible to input the vector of coefficients of f
127 // - the max could be computed in the last step
128 // - if inc = false and bf is a vector then it is supposed it is the truth table
129 //   if inc = true  and bf is a vector then it is supposed it is the vector of
    coeffs
130 //
131 // Example:
132 // f := RandomBooleanPolynomial(3) ;
133 // NonLinearityFWT(f) ; // input a Boolean polynomial
134 // // 2
135 // v := ANFCoefficients(f) ;
136 // NonLinearityFWT(v:inc:=true ; // input the vector of coefficients
137 // // 2
138 //
139
140 local TT ;  // truth table of f
141 local PTT ; // polarity truth table of f
142 local fws ;
143 local nv ;
144 local anfc ;
145 local max ;
146
147  if Dimension(Parent(bf)) eq -1 then
148  // Dimension(Parent(f)) is -1 if f is a boolean polynomial,
149  // while if f is a vector the function gives the dimension of the vector space
150    anfc := ANFCoefficients(bf) ;
151    TT := FastMobiusTransform(anfc) ;
```

```
    nv := IntegerRing()!Log(2,Rank(Parent(TT))) ;
  else
    if inc then
      anfc := bf ;
      TT := FastMobiusTransform(anfc) ;
    else
      TT := bf ;
    end if ;
    nv := IntegerRing()!Log(2,Rank(Parent(TT))) ;
  end if ;
  // NOW:
  // - bf IS A BOOLEAN POLYNOMIAL
  // - anfc IS THE VECTOR OF COEFFICIENTS OF bf
  // - TT IS THE TRUTH TABLE OF bf

/*
  // EXTRACT COEFFICIENTS if needed
  if Dimension(Parent(f)) eq -1 then
  // Dimension(Parent(f)) is -1 if f is a boolean polynomial,
  // while if f is a vector the function gives the dimension of the vector space
    anfc := ANFCoefficients(f) ;
    nv := Rank(Parent(f)) ;
  else
    anfc := f ;
    nv := IntegerRing()!Log(2,Ncols(anfc)) ;
  end if ;
*/

  // compute the TRUTH TABLE of f using fast mobius transform
  // TT := FastMobiusTransform(anfc) ;

  // compute the POLARITY TRUTH TABLE of f: 1 -> -1, 0 -> 1
  PTT := [] ;
  for i in [1..Ncols(TT)] do
    if TT[i] eq 1 then
      PTT[i] := -1 ;
    else
      PTT[i] := 1 ;
    end if ;
  end for ;

  // Compute the WALSH SPECTRUM using Fast Walsh Transform
  fws := FastWalshSpectrum(PTT) ;

  // find the MAX entry of the walsh spectrum
  max := Max({AbsoluteValue(fws[i]) : i in [1..#fws]}) div 2 ;

  return 2^(nv-1) - (max) ;
end function ;

//////////////////////////////////////////////////////////////
```

The next function is an implementation of Algorithm 6.

```
FastIntegerPolynomialEvaluation := function(c_f : leastleft:=false)
// This function allows to obtain the evaluation of an integer Polynomial f
// with n variables
// evaluated in points whose compontents are 0's and 1's
```

```
// in only n2^n steps.
// Given the vector of coefficients of a Polynomial over the ring R
// RETURNs the evaluation vector of f
// (with respect to points of 0's and 1's components
//  following by default the order 000,001,010,011,...)
// using a technique like the fast mobius transform.
//
// ATTENTION! The order of the monomials depends on the order defined over the
//            ring of the function f of which c_f are the coefficients...
//            Thus changing the ring of f, the order of the coefficients changes
//            (eventhough the weight is obviously the same)
//            and also the FMT changes (eventhough its weight doesn't)
//
// Reference:
// Cagdas Calik, PhD Thesis, Pag. 9, chap. 2.3
// http://sc.iam.metu.edu.tr/iamWarehouse/iam_Bibliography/
//        web/index.php/attachments/single/219
//
// EX:
// R := PolynomialRing(Rationals(),3) ;
// f := 8*R.1*R.2*R.3 + 3*R.1 ;
// PolynomialCoefficients(f) ;
// // (8 0 0 3 0 0 0 0)
// for v in VectorSpace(GF(2),Rank(Parent(f))) do
//   printf "%o->%o\n",v,Evaluate(f,[IntegerRing()!v[i] : i in [1..Ncols(v)]]) ;
// end for ;
// // (0 0 0)->0
// // (0 0 1)->0
// // (0 1 0)->0
// // (0 1 1)->0
// // (1 0 0)->3
// // (1 0 1)->3
// // (1 1 0)->3
// // (1 1 1)->11
// FastIntegerPolynomialEvaluation(PolynomialCoefficients(f)) ;
// // ( 0  0  0  0  3  3  3 11)
//

local n ; // number of variables of f
local ev_f ;
local lb ; // length of blocks
local nb ; // number of blocks

  n := IntegerRing()!Log(2,Ncols(c_f)) ;

  if leastleft then
    ev_f := Vector(Rationals(),[Integers()!c_f[i] : i in [1..Ncols(c_f)]]) ;
  else
    ev_f := Vector(Rationals(),Reverse([Integers()!c_f[i] : i in
    [1..Ncols(c_f)]])) ;
  end if ;

  for i in [1..n] do // n steps, the n_th step returns the evaluation vector

    lb := 2^i ;
    nb := (2^n) div lb ;
    for j in [1..nb] do // for each block

      for k in [1..lb div 2] do // for the first half of the block
        ev_f[k+(j-1)*lb] := ev_f[k+(j-1)*lb] ;
```

```
64      end for ;
65
66      for k in [(lb div 2)+1..lb] do // for the second half of the block
67        ev_f[k+(j-1)*lb] := ev_f[k+(j-1)*lb] + ev_f[k+(j-1)*lb-(lb div 2)] ;
68      end for ;
69
70    end for ;
71  end for ;
72
73  return ev_f ;
74 end function ;
75
76 ////////////////////////////////////////////////
77
78 FastNNFfromTT := function(f : leastleft:=false) //
79 // This function allows to obtain
80 // the Numeric Normal Form from the TruthTable of f
81 // in n2^(n-1) steps
82 // see:
83 // "A new representation of Boolean functions" - Carlet and Guillot, Section 3.1
84 // it is the opposite function of FastIntegerPolynomialEvaluation
85 //
86 // Example:
87 //
    FastIntegerPolynomialEvaluation(FastNNFfromTT(Vector(GF(2),[0,1,0,1,0,0,0,0])))
    ;
88 // // (0 1 0 1 0 0 0 0)
89 // FastNNFfromTT(Vector(RationalField(),[0,1,1,0])) ;
90 // // the coefficients must be rationals or binary
91 // // ( -2  1  1 0)
92 // R := PolynomialRing(RationalField(),2) ;
93 // // the coefficients must be rationals
94 // f := R.1 + R.2 ;
95 // FastNNFfromTT(f) ;
96 // // ( -2  1  1 0)
97 //
98
99 //local anfc ; // coefficients of the algebrai normal form
100 local ev_f ;
101 local n ; // number of varibale of f
102 local b ;
103 local temp ;
104 /*
105   // EXTRACT COEFFICIENTS if needed
106   if Dimension(Parent(f)) eq -1 then
107   // Dimension(Parent(f)) is -1 if f is a boolean polynomial,
108   // while if f is a vector the function gives the dimension of the vector space
109     if Category(BaseRing(Parent(f))) eq Category(GF(2)) then
110       temp := PolynomialCoefficients(f) ;
111       anfc := [IntegerRing()!temp[i] : i in [1..Ncols(temp)]] ;
112     elif Category(BaseRing(Parent(f))) eq Category(RationalField()) then
113       anfc := PolynomialCoefficients(f) ;
114     else
115       "ERROR! the base ring of f is not either GF(2) nor the RationalField()!" ;
116     end if ;
117     n := Rank(Parent(f)) ;
118   else
119     if Category(Parent(f[1])) eq Category(GF(2)) then
120       anfc := Vector([IntegerRing()!f[i] : i in [1..Ncols(f)]]) ;
121     elif Category(Parent(f[1])) eq Category(RationalField()) then
```

```
      anfc := f ;
    else
      "ERROR! the base ring of f is not either GF(2) nor the RationalField()!" ;
    end if ;
    n := IntegerRing()!Log(2,Ncols(anfc)) ;
  end if ;
*/
  n := IntegerRing()!Log(2,Ncols(f)) ;
//  ev_f := Reverse([Integers()!f[i] : i in [1..2^n]]) ;
  ev_f := [Integers()!f[i] : i in [1..2^n]] ;
//  ev_f := [f[i] : i in [1..2^n]] ;

  // COMPUTATION OF THE NNF
  for i in [0..n-1] do
    b := 0 ;
    repeat
      for x in [b..b+2^i-1]do
//        ev_f[x + 1] := ev_f[x+2^i + 1] - ev_f[x + 1] ;
        ev_f[x+2^i + 1] := ev_f[x+2^i + 1] - ev_f[x + 1] ;
      end for ;
      b := b+2^(i+1) ;
    until (b eq 2^n) ;
  end for ;

  if leastleft then
    return Vector(ev_f) ;
  else
    return Vector(Reverse(ev_f)) ;
  end if ;
end function ;
```

10.3 Minimum weight algorithms

The following code is part of our contribution.

```
DefiningPolynomialsFromCode := function(CC : poly := true)
// if poly = false then the vector of coefficients
// of the ANF of the defining polynomials are returned
// otherwise the algebraic normal form as a polynomial
// Example:
// C := Matrix(GF(2),4,5,[0,1,0,0,1, 1,1,1,0,1, 1,0,0,0,0, 1,0,0,1,1]) ;
// DefiningPolynomialsFromCode(C) ;
// [
//     $.1*$.2 + $.1 + $.2,
//     $.2 + 1,
//     $.1*$.2 + $.1,
//     $.1*$.2,
//     $.1*$.2 + $.2 + 1
// ]
// DefiningPolynomialsFromCode(C:poly:=false) ;
// [
//     (1 1 1 0),
//     (0 1 0 1),
//     (1 0 1 0),
//     (1 0 0 0),
```

```
//     (1 1 0 1)
// ]
//

  local F ;
  local C ;

  C := Matrix(CC) ;
  F := [] ;
  for i in [1..Ncols(C)] do
    if poly then
      F[i] := AlgebraicNormalForm([C[j][i] : j in [1..Nrows(C)]]) ;
    else
      F[i] := FastMobiusTransform(FastMobiusTransform(Transpose(C)[i])) ;
   end if ;
  end for ;

  return F ;
end function ;
```

The next two functions compute the weight polynomial and the weight ideal of, respectively, Definition 7.5.1 and Definition 7.5.4.

```
WeightPolynomial := function(CC : verb := false)
// compute the integer weight polynomial of a binary code C
// given as a list of codewords (or a matrix)
// Example:
// C := Matrix(GF(2),4,5,[0,1,0,0,1, 1,1,0,0,1, 1,0,1,0,0, 1,0,0,1,1]) ;
// WeightPolynomial(C) ;
// // $.1 + 2
//

local C ;
local F ; // defining polynomials of C in AND
local Fn ; // defining polynomials of C in NNF
local n, k ; // length and dimension of C
local R, Q ; // polynomial rings, Q is modulo the field equations

  C := Matrix(CC) ;
  k := Floor(Log(2,Nrows(C))) ;
  n := Ncols(C) ;
  F := DefiningPolynomialsFromCode(C) ;

  if verb then
    printf "Defining polynomials of C in ANF:\n%o\n",F ;
  end if ;

  R := PolynomialRing(Rationals(),k ) ;
  Q := quo< R | [R.i^2-R.i : i in [1..k]] > ;
  Fn := [] ;
  for i in [1..n] do
// uncomment the following to have the weight polynomial over Q
//    Fn[i] := Evaluate(NNFfromANF(F[i]),[Q.i : i in [1..k]] ) ;
    Fn[i] := Evaluate(NNFfromANF(F[i]),[R.i : i in [1..k]] ) ;
  end for ;

  if verb then
    printf "Defining polynomials of C in NNF:\n%o\n",Fn ;
```

```
  end if ;

  return &+[Fn[i] : i in [1..#Fn]] ;
end function ;

//////////////////////////////////////////////////////////////

IntegerWeightsIdeal := function(C,t : verb := false)
// compute the ideal whose variety
// contains the words which,
// encoded as codewords of C by the defining polynomials,
// have weight exactly t
//
// Example:
// C := Matrix(GF(2),4,5,[0,1,0,0,1, 1,1,0,0,1, 1,0,1,0,0, 1,0,0,1,1]) ;
// W3 := IntegerWeightsIdeal(C,3) ;
// W2 := IntegerWeightsIdeal(C,2) ;
// Variety(Ideal(W3)) ;
// // [ <1, 0>, <1, 1> ]
// Variety(Ideal(W2)) ;
// // [ <0, 0>, <0, 1> ]
//
local k ; // dimension of C
local wp ; // weight polynomial
local Q ; // polynomial ring of wp
local W ;

  k := Floor(Log(2,Nrows(C))) ;
  wp := WeightPolynomial(C : verb) ;

  if verb then
    printf "Weight polynomial of C:\n%o\n", wp ;
  end if ;

  Q := Parent(wp) ;
  W := [] ;
  for i in [1..k] do
    W[i] := Q.i^2 - Q.i ;
  end for ;

  W[#W+1] := wp - t;

  // uncomment to return an ideal instead of a list of polynomials
  // return Ideal(W) ;
  return W ;
end function ;
```

The next function is an extension of Algorithm 5, i.e. it is an implementation of Algorithm 8 described in Section 7.7.2.

```
MinWeight := function(C:compute_poly:=false)
// To compute the minimum weight of a binary code C of size m
// using B.f.'s in NNF representation of the code C.
// if m is not a power of 2 then the code C is splitted in
// subcodes of size a power of 2.
// If compute_poly = false,
// then polynomials are represented as vectors.
//
```

```
// Example:
// C := Matrix(GF(2),4,5,[0,1,0,0,1, 1,1,0,0,1, 1,0,1,0,0, 1,0,0,1,1]) ;
// w := MinWeightPoly(C) ; w ;
// 2
//

local n ; // length of C
local m ; // number of codewords of C
local k ; // minimum integer such that 2^k > m
local Bm ; // binary decomposition of m
local subC ;
local w ; // minimum weight
local j, temp ;
local T ; // temporary code
local Fv, Fp ; // list of subcodes as vector and as polynomials
local WPv ; // list of weight polynomials
local EV ; // list of evaluation vectors of the weights polynomials
local t, TIME ;

  TIME := [] ;

  n := Ncols(C) ;
  m := Nrows(C) ;
  k := Ceiling(Log(2,m)) ;

t := Cputime() ;
  // CONTRUCT SUBCODES OF CARDINALITY 2^i
  Bm := IntegerToSequence(m,2) ;
  subC := [* *] ;
  j := 1 ;
  for i in [1..#Bm] do
    if Bm[i] eq 1 then
      if #subC eq 0 then
        temp := 0 ;
      else
        temp := &+[Nrows(subC[h]) : h in [1..#subC]] ;
      end if ;
      subC[j] := Matrix(GF(2),2^(i-1),n,[C[h] : h in [temp+1..temp+2^(i-1)]]) ;
      j := j + 1 ;
    end if ;
  end for ;
TIME[#TIME+1] := Cputime(t) ;

  // REPRESENT THE CODES AS A SET OF B.f.'s in NNF as vectors
t := Cputime() ;
  Fv := [* *] ;
  for i in [1..#subC] do
    Fv[i] := [] ;
    T := Transpose(subC[i]) ;
    for h in [1..Nrows(T)] do
      Fv[i][h] := FastNNFfromTT(T[h] : leastleft := true) ;
    end for ;
  end for ;
TIME[#TIME+1] := Cputime(t) ;

  // REPRESENT THE CODES AS A SET OF B.f.'s in NNF as polynomials
t := Cputime() ;
  if compute_poly then
    Fp := [* *] ;
    for i in [1..#subC] do
```

```
      Fp[i] := [* *] ;
      T := Transpose(subC[i]) ;
      for h in [1..Nrows(T)] do
        Fp[i][h] := NNFfromANF(AlgebraicNormalForm(ElementToSequence(T[h]))) ;
      end for ;
    end for ;
  end if ;
TIME[#TIME+1] := Cputime(t) ;
  // NOTE:
  // CoefficientVectorToPolynomial(Fv[i][j]:leastleft:=true) = Fp[i][j]
  // order points in TT: 000 100 010 110 001 101 011 111

  // CREATE WEIGHT POLYNOMIAL FOR EACH CODE as vector
t := Cputime() ;
  WPv := [* *] ;
  for i in [1..#Fv] do
    WPv[i] := &+[Fv[i][h] : h in [1..#Fv[i]]] ;
  end for ;
TIME[#TIME+1] := Cputime(t) ;

t := Cputime() ;
  // COMPUTE EVALUATION VECTOR FOR EACH WEIGHT POLYNOMIAL
  EV := [* *] ;
  for i in [1..#WPv] do
    EV[i] := FastIntegerPolynomialEvaluation(WPv[i]:leastleft:=true) ;
  end for ;
TIME[#TIME+1] := Cputime(t) ;

t := Cputime() ;
  // compute the minimum weight
  w := Min({Min({EV[i][j] : j in [1..Ncols(EV[i])]}) : i in [1..#EV] }) ;
TIME[#TIME+1] := Cputime(t) ;

  if compute_poly then
    return w, TIME, Fp, Fv, WPv, EV ;
  else
    return w, TIME, Fv, WPv, EV ;
  end if ;
end function ;
```

10.4 Nonlinearity algorithms

The following code is part of our contribution. The next two functions compute the nonlinearity polynomial (Algorithm 12) and the ideal $\mathcal{N}_f^t$ of Definition 8.3.1.

```
NonlinearityPolynomial := function(f : incoeff:=false, leastleft:=false)
// FAST NONLINEARITY POLYNOMIAL
// Compute the nonlinearity polynomial of a Boolean function f
// using a butterfly algorithm (as Fast Fourier Transform)
// INPUT:
// - f, either the anf coeffs or the evaluation vector of a boolean function
//   (if incoeff=false then ev.vect., if incoeff=true then anf coeff)
// OUTPUT:
// - nlp, the coeffs vector of the nonlinearity polynomial
//   (if leastleft=false the leftmost coeff is the most  significant monomial,
```

```
//    if leastleft=true  the leftmost coeff is the least significant monomial)
//
// Example:
// R := PolynomialRing(GF(2),3) ;
// f := R.1*R.2 + R.3 ;
// c := ANFCoefficients(f) ;
// ev_f := EvaluationVector(f) ;
// NonlinearityPolynomial(c:incoeff:=true) ;
// // (-8  0  0  0  0  0  4  0  4  0  0  0  0  0 -2  4)
// NonlinearityPolynomial(ev_f) ;
// // (-8  0  0  0  0  0  4  0  4  0  0  0  0  0 -2  4)
// NonlinearityPolynomial(ev_f:leastleft:=true) ;
// // ( 4 -2  0  0  0  0  0  4  0  4  0  0  0  0  0 -8)
//

local ev_f ; // evaluation vector of f
local n ; // number of variables of f
local nlp ; // nonlinearity polynomial coefficients
local b ; // counter

  /* some checks to transform f */
  if incoeff then // if f is given by its anf coefficients
    ev_f := FastMobiusTransform(f) ;
  else // if f is given by its evaluation vector
    ev_f := f ;
  end if ;

  n := Integers()!Log(2,Ncols(ev_f)) ;

  // FAST TRANSFORM:
  // FIRST  HALF OF THE NLP
  nlp := [Integers()!ev_f[i] : i in [1..Ncols(ev_f)]] ;
  for i in [0..n-1] do
    b := 0 ;
    repeat
      for x in [b..b+2^i-1] do // for each block
        // x+1 ; // upper index
        // x+2^i + 1 ; // lower index
        nlp[x+1]     := nlp[x+1] + nlp[x+2^i+1] ;
        if x eq b then
          nlp[x+2^i+1] := 2^(i) -2*nlp[x+2^i+1] ;
        else
          nlp[x+2^i+1] := -2*nlp[x+2^i+1] ;
        end if ;
      end for ;
      b := b+2^(i+1) ;
    until (b eq 2^n) ;
  end for ;

  // SECOND HALF OF THE NLP
  nlp[1+2^n] := 2^n -2*nlp[1] ;
  for i in [2..2^n] do
    nlp[i+2^n] := -2*nlp[i] ;
  end for ;

  if leastleft then
    return Vector(nlp) ;
  else
    return Vector(Reverse(nlp)) ;
  end if ;
```

```
end function ;

//////////////////////////////////////////////////////////////

gbJRAT := function (t,Nf)
// Returns the Groebner Basis of the ideal J_{n,t}(f) OVER THE RATIONALS.
// The ideal J_{n,t}(f) has now as generators:
// - the field equations: xi^2 - xi
// - Nf, the polynomial composed by the sum of the elements of (ev_gn + ev_f)
//

local R ; // ring of f over K
local n ; // number of variables of f
local K ; // rational field
local Q ; // Q[ a_1, ..., a_n, a_{n+}) ]
local G ; // Groebner basis of J

  R := Parent(Nf) ;
  n := Rank(R) ;
  K := RationalField() ;
  Q := PolynomialRing(K,n,"grevlex") ; // much faster

  G := [] ;
  // ADD FIELD EQUATIONS for the variables of Q in J
  for i in [1..(n)] do
    G[i] := Q.i^2 - Q.i ;
  end for ;

  // ADD the NONLINEARITY POLYNOMIAL
  G[n+1] := Q!Nf - t ;
  //  G := G cat [Q!NonlinearityPolynomial(f) - t] ;

  return GroebnerBasis(G)  ;
end function ;
```

The next two commands

```
NonLinearityRAT(f:alg:=1);
NonLinearityRAT(f:alg:=2);
```

perform, respectively, Algorithm 10 and Algorithm 11.

```
NonLinearityRAT := function(bf : alg := 1, turnoffcheck := false, inc:=false)
// the nonlinearity of f is x
// if gbJRAT(x,f) != {1}
// if alg = 1 uses Groebner basis
// if alg = 2 the minimum evaluation different from 0 is returned
//             with the fast transform method
//             (if inc=false (default) then ev.vect., if inc=true then anf coeff)
//             uses:
//             - FastIntegerPolynomialEvaluation(PolynomialCoefficients(f))
// if no algorithm is defined returns -1
//
// Example:
// R := PolynomialRing(GF(2),3) ;
// f := R.1*R.2*R.3 + R.1 ;
```

```
// NonLinearityRAT(f:alg:=2);
// // 1
// NonLinearityRAT(f:alg:=1);
// // 1

local i ;
local V ;
local S ;
local n ;
local temp ; // temporary value
local anfc ; // vector of coefficients of bf
local f ;    // truth table of bf
local Nf ; // nonlinearity polynomial of f

  //////////////////// - COMPUTE TRUTH TABLE - ////////////////////////////

  if Dimension(Parent(bf)) eq -1 then
  // Dimension(Parent(f)) is -1 if f is a boolean polynomial,
  // while if f is a vector the function gives the dimension of the vector space
    anfc := ANFCoefficients(bf) ;
    f := FastMobiusTransform(anfc) ;
    n := IntegerRing()!Log(2,Rank(Parent(f))) ; // only needed in alg2 and alg3
  else
    if inc then
      anfc := bf ;
      f := FastMobiusTransform(anfc) ;
    else
      f := bf ;
    end if ;
    n := IntegerRing()!Log(2,Rank(Parent(f))) ; // only needed in alg2 and alg3
  end if ;
  // NOW:
  // - bf IS A BOOLEAN POLYNOMIAL
  // - anfc IS THE VECTOR OF COEFFICIENTS OF bf
  // - f IS THE TRUTH TABLE OF bf

  ////////////////////////// - ALG 1 - ///////////////////////////////////

  if alg eq 1 then // check if the base gbJRAT contains 1

    // compute nonlinearity polynomial
    if inc then
      Nf := NonlinearityPolynomial(anfc:incoeff:=true) ; // f given as
    coefficients
    else
      Nf := NonlinearityPolynomial(f:incoeff:=false) ; // f given as coefficients
    end if ;
    Nf := CoefficientVectorToPolynomial(Nf) ; // tranform Nf in a polynomial
    // find Groebner basis
    i := 0 ;
    while 1 in gbJRAT(i,Nf) do // gbJRAT works faster with
                               // Nf nonlin.pol. as coeff vector
      i := i + 1 ;
    end while ;

  ////////////////////////// - ALG 2 - ///////////////////////////////////

  elif alg eq 2 then // FAST TRANSFORM to compute the evaluation
    if inc then
```

```
73        Nf := NonlinearityPolynomial(anfc:incoeff:=true) ; // f given as
      coefficients
74      else
75        Nf := NonlinearityPolynomial(f:incoeff:=false) ; // f given as coefficients
76      end if ;
77      temp := FastIntegerPolynomialEvaluation(Nf) ;
78      i := Min({temp[j] : j in [1..Ncols(temp)] }) ;
79
80    ////////////////////////// - NO ALG - ////////////////////////////////////
81
82    else
83      printf "ERROR! Algorithm %o not defined\n", alg ;
84      return -1 ; // if no algorithm is defined returns -1
85    end if ;
86
87    ////////////////////////// - CHECK - ////////////////////////////////////
88
89    if not turnoffcheck then
90      if inc then
91        temp := NonLinearityFWT(anfc:inc:=true) ;
92      else
93        temp := NonLinearityFWT(bf) ;
94      end if ;
95      if temp ne i then
96        "i = ", i ;
97        "n = ", n ;
98        "f = ", f ;
99        "ERROR!! the function '' with alg = ", alg ;
100       "returned a different value with respect to the function
      'NonLinearityFWT'!!" ;
101       return f ;
102     end if ;
103   end if ;
104
105   return i ;
106 end function ;
```

Bibliography

[ACFP12] M.R. Albrecht, C. Cid, J.C. Faugere, and L. Perret, *On the relation between the mxl family of algorithms and gröbner basis algorithms*, Journal of Symbolic Computation **47** (2012), no. 8, 926–941.

[AMM03] M. E. Alonso, M. G. Marinari, and T. Mora, *The big mother of all dualities: Möller algorithm*, Comm. Algebra **31** (2003), no. 2, 783–818.

[AS88] W. Auzinger and H. J. Stetter, *An elimination algorithm for the computation of all zeros of a system of multivariate polynomial equations*, Internat. Schriftenreihe Numer. Math. **86** (1988), 11–30.

[Bas65] L.A. Bassalygo, *New upper bounds for error correcting codes*, Problemy Peredachi Informatsii **1** (1965), no. 4, 41–44.

[BCDM10] J. Buchmann, D. Cabarcas, J. Ding, and M.S.E. Mohamed, *Flexible partial enlargement to accelerate gröbner basis computation over\ mathbb {F} _ 2*, Progress in Cryptology–AFRICACRYPT 2010, Springer, 2010, pp. 69–81.

[BD09] M. Brickenstein and A. Dreyer, *Polybori: A framework for gröbner-basis computations with boolean polynomials*, Journal of Symbolic Computation **44** (2009), no. 9, 1326–1345.

[BDMM09] J. A Buchmann, J. Ding, M.S.E. Mohamed, and W.S.A.E. Mohamed, *Mutantxl: Solving multivariate polynomial equations for cryptanalysis*, Symmetric Cryptography **9031** (2009).

[Bel14a] E. Bellini, *Yet another algorithm to compute the nonlinearity of a boolean function*, Yet Another Conference on Cryptography, YACC 2014 (Toulon), 2014.

[Bel14b] ———, *Yet another algorithm to compute the nonlinearity of a boolean function*, Preprint http://arxiv.org/abs/1404.2471, 2014.

[BFCP+10] J. Borges, C. Fernández-Córdoba, J. Pujol, J. Rifà, and M. Villanueva, *{{{\ mathbb Z} _2}{{\ mathbb Z} _4}}-linear codes: generator matrices and duality*, Designs, Codes and Cryptography **54** (2010), no. 2, 167–179.

[BFP09] L. Bettale, JC. Faugère, and L. Perret, *Hybrid approach for solving multivariate systems over finite fields*, Journal of Mathematical Cryptology **3** (2009), no. 3, 177–197.

[BFP12] L. Bettale, J.-C. Faugère, and L. Perret, *Solving polynomial systems over finite fields: Improved analysis of the hybrid approach*, Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation, ACM, 2012, pp. 67–74.

[BGH83] H. Bauer, B. Ganter, and F. Hergert, *Algebraic techniques for nonlinear codes*, Combinatorica **3** (1983), no. 1, 21–33.

[BGS14] E. Bellini, E. Guerrini, and M. Sala, *Some bounds on the size of codes*, IEEE Trans. Inform. Theory **60** (2014), no. 3, 1475–1480.

[BJMM12] A. Becker, A. Joux, A. May, and A. Meurer, *Decoding random binary linear codes in 2 n/20: how 1+ 1= 0 improves information set decoding*, Advances in Cryptology–EUROCRYPT 2012, Springer, 2012, pp. 520–536.

[BLP11] D. J Bernstein, T. Lange, and C. Peters, *Smaller decoding exponents: ball-collision decoding*, Advances in Cryptology–CRYPTO 2011, Springer, 2011, pp. 743–760.

[BM76] I. F. Blake and R. C. Mullin, *An introduction to algebraic and combinatorial coding theory*, Academic Press, Inc., 1976.

[Broa] A.E. Brouwer, *Table of general binary codes.*

[Brob] ———, *Table of general ternary codes.*

[BSS14] E. Bellini, I. Simonetti, and M. Sala, *Nonlinearity of Boolean functions: an algorithmic approach based on multivariate polynomials*, Preprint http://arxiv.org/abs/1404.2741, 2014.

[Buc98] B. Buchberger, *An algorithmical criterion for the solvability of algebraic systems of equations*, London Math. Soc. LNS **251** (1998), 535–545.

[Buc06] ———, *Bruno Buchberger's PhD thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal*, J. Symb. Comput. **41** (2006), no. 3-4, 475–511.

[BvLW83] R. D. Baker, J. H. van Lint, and R. M. Wilson, *On the Preparata and Goethals codes*, IEEE Trans. on Inf. Th. **29** (1983), no. 3, 342–345.

[Çal13a] Çağdaş Çalık, *Computing cryptographic properties of boolean functions from the algebraic normal form representation*, Ph.D. thesis, Middle East Technical University, 2013.

[Çal13b] ———, *Nonlinearity computation for sparse boolean functions*, arXiv preprint arXiv:1305.0860 (2013).

[Car02] C. Carlet, *On the coset weight divisibility and nonlinearity of resilient and correlation-immune functions*, Sequences and their Applications, Springer, 2002, pp. 131–144.

[Car10] C. Carlet, *Boolean functions for cryptography and error correcting codes*, Boolean Models and Methods in Mathematics, Computer Science, and Engineering (2010), 257–397.

[CBFS13] J. Cannon, W. Bosma, C. Fieker, and A. Steel, *Handbook of MAGMA functions*, MAGMA Computer Algebra, Sydney, v2.19 ed., April 2013, Volume 9 - Commutative Algebra and Algebraic Geometry, Chapter 105.4 - Gröbner basis, pp.3192, http://magma.maths.usyd.edu.au/magma/handbook/text/1161#12745.

[CC98] A. Canteaut and F. Chabaud, *A new algorithm for finding minimum-weight words in a linear code: application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511*, IEEE Transactions on Information Theory **44** (1998), no. 1, 367.

[CG99] C. Carlet and P. Guillot, *A new representation of Boolean functions*, Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Springer, 1999, pp. 94–103.

[CG01] C. Carlet and P. Guillot, *Bent, resilient functions and the Numerical Normal Form*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science **56** (2001), 87–96.

[CKPS00] N. Courtois, A. Klimov, J. Patarin, and A. Shamir, *Efficient algorithms for solving overdefined systems of multivariate polynomial equations*, Proc. of EUROCRYPT 2000, LNCS, vol. 1807, Springer, 2000, pp. 392–407.

[CLO07] D. Cox, J. Little, and D. O'Shea, *Ideals, varieties, and algorithms*, third ed., Springer, 2007, An introduction to computational algebraic geometry and commutative algebra.

[Cou04] N. T. Courtois, *Algebraic attacks over gf (2 k), application to hfe challenge 2 and sflash-v2*, Public Key Cryptography–PKC 2004, Springer, 2004, pp. 201–217.

[CP02] N. T. Courtois and J. Pieprzyk, *Cryptanalysis of block ciphers with overdefined systems of equations*, Advances in Cryptology–ASIACRYPT 2002, Springer, 2002, pp. 267–287.

[CP03] N. T. Courtois and J. Patarin, *About the xl algorithm over gf (2)*, Topics in Cryptology–CT-RSA 2003, Springer, 2003, pp. 141–157.

[Del73] P. Delsarte, *An algebraic approach to the association schemes of coding theory*, Philips Res. Rep. Suppl. (1973), no. 10, vi+97.

[dLG01] W. de Launey and D. M. Gordon, *A remark on Plotkin's bound*, IEEE Trans. on Inf. Th. **47** (2001), no. 1, 352–355.

[Fau99] J. C. Faugére, *A new efficient algorithm for computing Gröbner bases* (F_4), J. Pure Appl. Algebra **139** (1999), no. 1-3, 61–88.

[Fau02] J. C. Faugère, *A new efficient algorithm for computing Gröbner bases without reduction to zero* (F_5), Proc. of ISSAC 2002 (New York), ACM, 2002, pp. 75–83.

[FEDS10] JC. Faugère, M. S. El Din, and PJ. Spaenlehauer, *Computing loci of rank defects of linear matrices using gröbner bases and applications to cryptology*, Proc. of the 2010 International Symposium on Symbolic and Algebraic Computation, ACM, 2010, pp. 257–264.

[FEDS11] JC. Faugere, M. S. El Din, and PJ. Spaenlehauer, *Gröbner bases of bihomogeneous ideals generated by polynomials of bidegree (1, 1): Algorithms and complexity*, Journal of Symbolic Computation **46** (2011), no. 4, 406–437.

[FGLM93] J. C. Faugère, P. Gianni, D. Lazard, and T. Mora, *Efficient computation of zero-dimensional Gröbner bases by change of ordering*, J. Symbolic Comput. **16** (1993), no. 4, 329–344.

[FM11] JC. Faugère and C. Mou, *Fast algorithm for change of ordering of zero-dimensional gröbner bases with sparse multiplication matrices*, Proc. of the 36th international symposium on Symbolic and algebraic computation, ACM, 2011, pp. 115–122.

[FR09] JC. Faugère and S. Rahmany, *Solving systems of polynomial equations with symmetries using sagbi-gröbner bases*, Proc. of the 2009 international symposium on Symbolic and algebraic computation, ACM, 2009, pp. 151–158.

[Gei09] O. Geil, *Algebraic geometry codes from order domains*, Gröbner Bases, Coding, and Cryptography (M. Sala, T. Mora, L. Perret, S. Sakata, and C. Traverso, eds.), RISC Book Series, Springer, Heidelberg, 2009, pp. 121–141.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of np-completeness*, WH Freeman & Co., San Francisco (1979).

[GOS06] E. Guerrini, M. Orsini, and M. Sala, *Computing the distance distribution of systematic non-linear codes*, BCRI preprint, www.bcri.ucc.ie 50, UCC, Cork, Ireland, 2006.

[GOS09] E. Guerrini, E. Orsini, and I. Simonetti, *Gröbner bases for the distance distribution of systematic codes*, Gröbner Bases, Coding, and Cryptography, Springer, 2009, pp. 367–372.

[GOS10] E. Guerrini, M. Orsini, and M. Sala, *Computing the distance distribution of systematic non-linear codes*, Journal of Algebra and Its Applications **9** (2010), no. 2, 241–256.

[Gri60] J.H. Griesmer, *A bound for error-correcting codes*, IBM Journal of Research and Development **4** (1960), no. 5, 532–542.

[GS07] E. Guerrini and M. Sala, *An algebraic approach to the classification of some non-linear codes*, Proc. of WCC 2007 **INRIA** (2007), 177–185.

[Gue05] E. Guerrini, *On distance and optimality in non-linear codes*, Master's thesis (laurea), Univ. of Pisa, Dept. of Math., 2005.

[Gue09] Eleonora Guerrini, *Systematic codes and polynomial ideals*, Ph.D. thesis, University of Trento, 2009.

[HKC+94] Jr. A. R. Hammons, P. V. Kumar, A. R. Calderbank, N. J. A. Sloane, and P. Solé, *The $\mathbf{Z}_4$-linearity of Kerdock, Preparata, Goethals, and related codes*, IEEE Trans. on Inf. Th. **40** (1994), no. 2, 301–319.

[HP03] W. C. Huffman and V. Pless, *Fundamentals of error-correcting codes*, Cambridge University Press, 2003.

[Joh62] S. Johnson, *A new upper bound for error-correcting codes*, Information Theory, IRE Transactions on **8** (1962), no. 3, 203–207.

[Joh71] ———, *On upper bounds for unrestricted binary-error-correcting codes*, Information Theory, IEEE Transactions on **17** (1971), no. 4, 466–478.

[J.P93] Cardinal J.P., *Dualité et algorithms itératifs pour la résolution de systémes polynomiaux*, Ph.D. thesis, 1993.

[Ker72] A. M. Kerdock, *A class of low-rate nonlinear binary codes*, Information and Control **20** (1972), 182–187; ibid. **21** (1972), 395.

[Lev95] V. I. Levenshtein, *Krawtchouk polynomials and universal bounds for codes and designs in hamming spaces*, Information Theory, IEEE Transactions on **41** (1995), no. 5, 1303–1321.

[Lev98] V.I. Levenshtein, *Universal bounds for codes and designs*, Handbook of Coding Theory (V. S. Pless and W. C. Huffman, eds.), vol. 1, Elsevier, 1998, pp. 499–648.

[LL98] T. Laihonen and S. Litsyn, *On upper bounds for minimum distance and covering radius of non-binary codes*, Des. Codes Cryptogr. **14** (1998), no. 1, 71–80.

[LT96] S. Litsyn and A. Tietäväinen, *Upper bounds on the covering radius of a code with a given dual distance*, European Journal of Combinatorics **17** (1996), no. 2, 265–270.

[MAG] *MAGMA: Computational Algebra System for Algebra, Number Theory and Geometry*, The University of Sydney Computational Algebra Group., http://magma.maths.usyd.edu.au/magma.

[MCD+10] M. S. E. Mohamed, D. Cabarcas, J. Ding, J. Buchmann, and S. Bulygin, *Mxl3: An efficient algorithm for computing gröbner bases of zero-dimensional ideals*, Information, Security and Cryptology–ICISC 2009, Springer, 2010, pp. 87–100.

[MMDB08] M.S.E. Mohamed, W.S.A.E. Mohamed, J. Ding, and J. Buchmann, *Mxl2: Solving polynomial equations over gf (2) using an improved mutant strategy*, Post-Quantum Cryptography, Springer, 2008, pp. 203–215.

[Mor05] T. Mora, *Solving polynomial equation systems. II, Macaulay's paradigm and Gröbner technology*, Encyclopedia of Mathematics and its Applications, vol. 99, Cambridge University Press, 2005.

[Mor09] ———, *Gröbner technology*, Gröbner Bases, Coding, and Cryptography (M. Sala, T. Mora, L. Perret, S. Sakata, and C. Traverso, eds.), RISC Book Series, Springer, Heidelberg, 2009, pp. 11–26.

[Mou05] B. Mourrain, *Bezoutian and quotient ring structure*, J. Symbolic Comput. **39** (2005), no. 3-4, 397–415.

[MS77] F. J. MacWilliams and N. J. A. Sloane, *The theory of error-correcting codes. I*, North-Holland Publishing Co., Amsterdam, 1977, North-Holland Mathematical Library, Vol. 16.

[PBH98] V. Pless, R.A. Brualdi, and W.C. Huffman, *Handbook of coding theory*, Elsevier Science Inc., 1998.

[Pet10] C. Peters, *Information-set decoding for linear codes over f q*, Post-Quantum Cryptography, Springer, 2010, pp. 81–94.

[Plo60] M. Plotkin, *Binary codes with specified minimum distance*, Information Theory, IRE Transactions on **6** (1960), no. 4, 445–450.

[Pre68] F. P. Preparata, *A class of optimum nonlinear double-error correcting codes*, Inform. Control **13** (1968), no. 13, 378–400.

[PVZ12] J. Pujol, M. Villanueva, and F. Zeng, *Minimum distance of binary nonlinear codes.*

[Rom92] S. Roman, *Coding and information theory*, Graduate Texts in Mathematics, vol. 134, Springer-Verlag, New York, 1992.

[Sch05] A. Schrijver, *New code upper bounds from the terwiller algebra and semidefinite programming*, IEEE Trans. Inform. Theory **51** (2005), no. 8, 2859–2866.

[Sim07] I. Simonetti, *On some applications of commutative algebra to Boolean functions and their non-linearity*, Ph.D. thesis, University of Trento, 2007.

[Sim09] ———, *On the non-linearity of Boolean functions*, Gröbner Bases, Coding, and Cryptography (M. Mora T. Perret L. Sakata S. Sala and C. Traverso, eds.), RISC Book Series, Springer, Heidelberg, 2009, pp. 409–413.

[SS65] G. Solomon and J. J. Stiffler, *Algebraically punctured cyclic codes*, Information and Control **8** (1965), no. 2, 170–179.

[SS07a] M. Sala and I. Simonetti, *An algebraic description of Boolean functions*, Proc. of WCC 2007 (2007), 343–349.

[SS07b] ———, *On the non-linearity of Boolean functions and Gröbner bases*, preprint, submitted, 2007.

[ST09] M. Mora T. Perret L. Sakata S. Sala and C. Traverso, *Gröbner Bases, Coding, and Cryptography*, RISC Book Series, Springer, Heidelberg, 2009.

[Ste13] Allan Steel, *A dense variant of the F4 groebner basis algorithm*, http://magma.maths.usyd.edu.au/users/allan/densef4/#RH, December 2013, tinyurl.com/DenseF4.

[Tra92a] C. Traverso, *Linear gröbner methods and "natural" representations of algebraic numbers*, July 1992, Draft.

[Tra92b] ———, *Natural representation of algebraic numbers*, Conference at MEGA-92, 1992.

[Zim96] K.H. Zimmermann, *Integral hecke modules, integral generalized reed-muller codes, and linear codes*, Berichte des Forschungsschwerpunktes Informations- und Kommunikationstechnik, Techn. Univ. Hamburg-Harburg, 1996.

[ZL84] V. A. Zinov'ev and S.N. Litsyn, *On Shortening of Codes*, Problemy Peredachi Informatsii **20** (1984), no. 1, 3–11.