BINARY CODES
BINARY CODES DATABASES

MAGMA Packages

by

Jaume Pujol and Mercè Villanueva

Combinatorics, Coding and Security Group (CCSG)

Autonomous University of Barcelona

Barcelona
April 7, 2015

Contents

Preface

The research group CCSG (Combinatorics, Coding and Security Group) is one of the research groups in the dEIC (Department of Information and Communications Engineering) at the UAB (Universitat Autònoma de Barcelona) in Spain.

From 1987 the team CCSG has been uninterruptedly working in several projects and research activities on Information Theory, Communications, Coding Theory, Source Coding, Teledetection, Cryptography, Electronic Voting, e-Auctions, Mobile Agents, etc.

The more important know-how of CCSG is about algorithms for forward error correction (FEC), such as Golay codes, Hamming product codes, Reed-Solomon codes, Preparata and Preparata-like codes, (extended) nonlinear 1-perfect codes, $\mathbb{Z}_4$-linear codes, $\mathbb{Z}_2\mathbb{Z}_4$-linear codes, etc.; computations of the rank and the dimension of the kernel for nonlinear codes as binary 1-perfect codes, q-ary 1-perfect codes, Preparata codes, Hadamard codes, Kerdock codes, quaternary Reed-Muller codes, etc.; the existence and structural properties for 1-perfect codes, uniformly packed codes, completely regular codes, completely transitive codes, etc.

Currently, the research projects where CCSG is involved are supported by the Spanish MICINN under Grants TIN2010-17358 and TIN2013-40524-P, and by the Catalan AGAUR under Grant 2014SGR-691. Part of this research deals with $\mathbb{Z}_2\mathbb{Z}_4$-linear codes. There were no symbolic software to work with these codes, so the members of CCSG has been developing a new package that supports the basic facilities for $\mathbb{Z}_2\mathbb{Z}_4$-additive codes. Specifically, this Magma package generalizes most of the known functions for codes over the ring $\mathbb{Z}_4$, which are subgroups of $\mathbb{Z}_4^n$, to $\mathbb{Z}_2\mathbb{Z}_4$-additive codes, which are subgroups of $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, maintaining all the functionality for codes over $\mathbb{Z}_4$ and adding new functions which, not only generalize the previous ones, but introduce new variants when it is needed. The implementation is based on the results that appeared in [1, 2, 6, 7].

The Gray map image of codes over $\mathbb{Z}_4$, or in general $\mathbb{Z}_2\mathbb{Z}_4$-additive codes, denoted also as $\mathbb{Z}_2\mathbb{Z}_4$-linear codes, are usually binary nonlinear codes. Magma

does not have functions to deal, in an efficient way, with nonlinear codes in general. Therefore, the members of CCSG has also been developing packages that supports functions for binary nonlinear codes, which are represented as a union of cosets of a binary linear subcode, called kernel. The packages include functions to represent, manipulate, store and construct binary nonlinear codes in an efficient way, mainly when the codes have a large kernel. They also include functions to compute the minimum weight and distance of these codes, as long as functions to simulate the decoding process using nonlinear codes. The implementation is based on the results that appeared in [22, 23].

A beta version of these new packages for $\mathbb{Z}_2\mathbb{Z}_4$-additive codes and for binary nonlinear codes, and the manual with the description of all functions can be downloaded from the web page `http://www.ccsg.uab.es`. For any comment or further information about this package, you can send an e-mail to *support-ccsg@deic.uab.cat*.

- Chapter 1 has been developed by Jaume Pujol and Mercè Villanueva with the collaboration of Joan Cuadros, Miriam Gutiérrez, Víctor Ovalle, Marta Pujol, Laura Vidal and Fanxuan Zeng.

- Chapter 2 has been developed by Jaume Pujol and Mercè Villanueva with the collaboration of Victoria González and Fanxuan Zeng.

Chapter 1

Binary Codes

1.1 Introduction

MAGMA currently supports the basic facilities for codes over finite fields and codes over integer residue rings and galois rings, all that are linear codes (see [4, Chapters 127-130]). Therefore, MAGMA provides functions for the special case of binary linear codes, that is when the finite field is $GF(2)$, or equivalently the finite ring is $\mathbb{Z}_2$. This chapter describes functions which are applicable to binary codes not necessarily linear.

Let $\mathbb{Z}_2^n$ be the n-dimensional vector space over $\mathbb{Z}_2$. An (n, M, d) *binary code* C is a subset of $\mathbb{Z}_2^n$ with cardinality M and minimum Hamming distance d. 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 $a \in \mathbb{Z}_2^n$ 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. Moreover, if C is linear, the zero word belongs to C; but if C is nonlinear, the zero word does not need to belong to C. In this case, we can always consider a new binary code $C' = C + c = \{x + c \mid x \in C\}$ for any $c \in C$, which is equivalent to C, such that the zero word is included in C'. In this chapter, the term "code" will refer to a binary code not necessarily linear such that it contains the zero word, unless otherwise specified.

Two structural properties of binary codes are the rank and kernel. The *rank* of a binary code C, $r = rank(C)$, is simply the dimension of the linear span, $\langle C \rangle$, of C. The *kernel* of a binary code C is defined as $K(C) = \{x \in \mathbb{Z}_2^n \mid x + C = C\}$. If the zero word is in C, then $K(C)$ is a linear subspace of C. We will denote the dimension of the kernel of C by $k = ker(C)$. In

general, C can be written as the union of cosets of $K(C)$:

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

where $c_0 = \mathbf{0}$ and $t+1 = M/2^k$. These parameters can be used to distinguish between nonequivalent binary codes, since equivalent ones have the same parameters r and k. Therefore, they provide a sufficient condition which is not necessary, since two nonequivalent binary codes could have the same parameters r and k.

Let C be a binary code of length n such that the zero word belongs to C, with kernel $K(C)$ of dimension k and coset representatives $[c_1, \ldots, c_t]$. The *parity check system* of the binary code C is an $(n-k) \times (n+t)$ binary matrix $(G|S)$, where G is a generator matrix of the dual code $K(C)^{\perp}$ and $S = (G \cdot c_1 \; G \cdot c_2 \; \ldots \; G \cdot c_t)$. The *super dual* of the binary code C is the binary linear code generated by the parity check system $(G|S)$. Note that if C is a binary linear code, the super dual is the dual code $C^{\perp}$. From these definitions, we can establish the following properties [9, 10]:

- Let $col(S)$ denote the set of columns of the matrix S. Then, $c \in C$ if and only if $G \cdot c = \mathbf{0}$ or $G \cdot c \in col(S)$.
- Let $r = rank(C)$ and $k = ker(C)$. Then, $r = n - dim(G) + dim(S)$ and $k = n - dim(G)$.
- The super dual of a code C is unique, up to a permutation of the columns of the matrix S.
- Let π be any permutation of the set of coordinate positions. If $(G|S)$ is a parity check system for a nonlinear code C, then $(\pi(G)|S)$ will be a parity check system for the code $\pi(C)$.
- Let $C' = C + c_i$, for some $i \in \{1, \ldots, t\}$. If $(G|S)$ is a parity check system for C, then $(G|S')$ is a parity check system for C', where S' is obtained from S by adding the column vector $G \cdot c_i$ of S to all other columns of S.

1.2 Construction of Binary Codes

`BinaryCode(L)`

Creates a binary code C given by its super dual, which is a binary linear code of length $n + t$ generated by the parity check system $(G|S)$ for the binary code C. As it is explained above, the parity check system $(G|S)$ is constructed using the kernel $K(C)$ and the coset representatives $[c_1, \ldots, c_t]$ of the binary code obtained by L, where:

1. L is a sequence of elements of $V = \mathbb{Z}_2^n$,
2. or, L is a subspace of $V = \mathbb{Z}_2^n$,
3. or, L is an $m \times n$ matrix A over the ring $\mathbb{Z}_2$,
4. or, L is a binary linear code,
5. or, L is a quaternary linear code,
6. or, L is a $\mathbb{Z}_2\mathbb{Z}_4$-additive code.

Note that in general, depending on the cardinality M of the binary code C, this function could take some time to compute $K(C)$ and $[c_1, \ldots, c_t]$, in order to return the binary code given by its super dual.

If L is a quaternary linear code or a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, then the binary code corresponds to the image of L under the Gray map.

If the zero word is not in L, then L is substituted by $L + c$, where c is the first element in L.

This constructor appends five attributes to the code category:

- `Length`: The length n of the binary code.
- `Kernel`: The kernel of the binary code as a binary linear subcode.
- `CosetRepresentatives`: The sequence of coset representatives $[c_1, \ldots, c_t]$.
- `MinimumDistance`: The minimum (Hamming) distance d.
- `IsLinear`: It is `true` if and only if C is a binary linear code.

If L is a subspace of $V = \mathbb{Z}_2^n$ or a binary linear code, this function returns the dual code. In this case, the `Kernel` attribute is identical to the binary linear code L defined by the function `LinearCode()`, the `CosetRepresentatives` attribute is the empty sequence, the `Length` attribute is n and the `IsLinear` attribute is `true`.

`BinaryCode(L, K)`

`BinaryCode(L, K, Rep)`

`BinaryCode(K, Rep)`

Creates a binary code C given by its super dual, which is a binary linear code of length $n + t$ generated by the parity check system $(G|S)$ for the binary code C. The parity check system $(G|S)$ is constructed using the kernel $K(C)$ and the coset representatives $[c_1, \ldots, c_t]$ of the binary code C.

Depending on the parameters of the function, the binary code C is constructed from a sequence L of elements of $V = \mathbb{Z}_2^n$, and a partial kernel K; from a sequence L of elements of $V = \mathbb{Z}_2^n$, a partial kernel K, and the corresponding sequence of partial coset representatives *Rep*; or from a partial kernel K, and the corresponding sequence of partial coset representatives *Rep*. The partial kernel K is given as a binary linear code.

This constructor appends the same attributes as the previous function.

`BinarySuperDualCode(K, Rep)`

Given the kernel K as a linear code of length n and the corresponding sequence of t coset representatives *Rep* of a binary code C, return the super dual of C, that is, the binary linear code of length $n + t$ generated by the parity check system $(G|S)$ constructed using K and *Rep*.

If *Rep* is an empty sequence, this function returns the dual of the kernel K, which is also the dual of C.

Example H1E1

We can define a binary code by giving a sequence of elements of a vector space $V = \mathbb{Z}_2^n$, or a matrix over $\mathbb{Z}_2$. Note that the `BinaryCode` function returns the super dual of a binary code C and the codewords of C can be generated as the union of the cosets of its kernel.

```
> V  := VectorSpace(GF(2), 4);
> L  := [V!0,V![1,0,0,0],V![0,1,0,1],V![1,1,1,1]];
> C1 := BinaryCode(L);
> C1;
[7, 4, 2] Linear Code over GF(2)
Generator matrix:
```

```
[1 0 0 0 1 0 1]
[0 1 0 0 0 1 1]
[0 0 1 0 0 0 1]
[0 0 0 1 0 1 1]
> IsBinaryLinearCode(C1);
false

> A  := Matrix(L);
> C2 := BinaryCode(A);
> C2;
[7, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 0 1]
[0 1 0 0 0 1 1]
[0 0 1 0 0 0 1]
[0 0 0 1 0 1 1]
> IsBinaryEqual(C1, C2);
true
```

A binary linear code C can be generated as a binary code with this constructor. Note that in this case the `BinaryCode` function returns the dual of C.

```
> C := LinearCode(sub<V|[[0,0,1,1],[1,0,1,1]]>);
> D := BinaryCode(sub<V|[[0,0,1,1],[1,0,1,1]]>);
> D;
[4, 2, 1] Linear Code over GF(2)
Generator matrix:
[0 1 0 0]
[0 0 1 1]
> IsBinaryLinearCode(D);
true
> Dual(C) eq D;
true
```

`BinaryUniverseCode(`n`)`

Given a positive integer n, return the binary code of length n consisting of all possible codewords.

`BinaryZeroCode(`n`)`

Given a positive integer n, return the binary code of length n consisting of only the zero codeword.

`BinaryRandomCode(`n`, `M`)`

Given two positive integers n and M, return a random binary code of length n and cardinality M.

`BinaryRandomCode(`n`, `M`, `k`)`

Given three positive integers n, M and k, return a random binary code of length n, cardinality M and with a kernel of minimum dimension k.

`BinaryIsomorphicCode(C, p)`

Given a binary code C of length n and an element p belonging to a permutation group of degree n, return the binary code $p(C) = \{p(c) \mid c \in C\}$, where $p(c)$ is obtained from a codeword c by permuting the coordinate positions of c according to p. The corresponding function for a linear code C is $C^\wedge p$.

Example H1E2

We can also construct random binary codes with a given length, number of codewords and, optionally, with a given minimum kernel dimension.

```
> C1 := BinaryRandomCode(8, 56);
> C1;
[14, 5, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 1 0 1 0 1 0]
[0 1 0 0 0 0 1 0 1 0 1 0 0 0]
[0 0 1 0 0 0 1 1 1 0 1 0 0 1]
[0 0 0 1 0 1 1 1 0 1 1 1 1 1]
[0 0 0 0 1 0 0 1 0 0 0 1 1 1]
> (BinaryLength(C1) eq 8) and (BinaryCardinal(C1) eq 56);
true

> C2 := BinaryRandomCode(8, 48, 2);
> C2;
[19, 6, 6] Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 1 0]
[0 1 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 0 1]
[0 0 1 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1 1]
[0 0 0 0 1 0 0 0 0 0 1 1 1 1 0 1 1 1 1]
[0 0 0 0 0 1 0 1 0 0 0 0 1 1 0 0 1 1 1]
[0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 1]
> BinaryDimensionOfKernel(C2) ge 2;
true
```

The binary zero code of length n is contained in every binary code of length n, and similarly every binary code of length n is contained in the binary universe code of length n.

```
> U := BinaryUniverseCode(10);
> Z := BinaryZeroCode(10);
> R := BinaryRandomCode(10, 50);
> IsBinarySubset(Z, R) and IsBinarySubset(R, U);
true
```

1.3 Invariants of a Binary Code

`BinaryLength(C)`

Given a binary code C, return the length of the code.

`BinaryCardinal(C)`

Given a binary code C, return the number of words belonging to C.

`BinaryParameters(C)`

Given a binary code C, return a list with the parameters $[n, M, d]$, where n is the length of the code, M the number of codewords and d the minimum (Hamming) distance.

`BinarySpanCode(C)`

Given a binary code C of length n, return the linear span of C, that is, the binary linear code generated by the codewords of C.

`BinaryDimensionOfSpan(C)`

`BinaryRank(C)`

Given a binary code C of length n, return its rank. The rank of a binary code C is the dimension of the linear span of C over $\mathbb{Z}_2$.

`BinaryKernelCode(C)`

Given a binary code C of length n, return its kernel as a binary linear code, and the representatives of the cosets as a list of binary vectors of length n. The kernel of a binary code C is the set of codewords x such that $x + C = C$.

`BinaryDimensionOfKernel(C)`

Given a binary code C of length n, return the dimension of its kernel. The kernel of a binary code C is the set of codewords x such that $x + C = C$.

Example H1E3

Given a binary code C, we compute its length, number of codewords, minimum distance, rank and kernel dimension.

```
> V21 := VectorSpace(GF(2), 21);
> C21_kernel := LinearCode(sub< V21 |
    [1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,1,0,0],
    [1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,0,0,0,0,1,0],
    [1,1,1,0,0,1,1,0,0,0,1,1,0,0,0,1,1,1,0,0,1]
>);
> C21_representatives := [
    V21![0,0,0,1,0,1,1,0,1,1,0,1,0,1,0,1,1,0,0,0,0],
```

```
    V21![0,0,1,0,1,0,1,1,1,0,0,0,0,0,1,1,1,1,0,0,0],
    V21![0,1,0,1,0,1,1,0,1,0,1,0,1,0,1,0,0,1,0,0,0],
    V21![0,1,1,0,0,0,0,1,1,1,1,1,1,1,0,0,1,0,0,0,0,0],
    V21![1,0,0,0,0,1,0,0,0,1,1,0,1,1,1,1,0,0,0,0,1],
    V21![1,0,1,0,0,1,0,1,0,1,0,1,0,0,1,0,1,0,0,1,0],
    V21![1,1,0,1,0,0,1,1,0,0,0,0,0,1,1,0,1,0,0,0,1]
];
> C := BinaryCode(C21_kernel, C21_representatives);
> C;
[28, 18] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 0 1 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 1 1 1]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 1 1 1]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 0 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 1 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 0 0 1]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 1 1 1 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 0 1 1 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 1 1 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0 1 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 0 0 1 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 0 1 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 0 0 1 1]

> BinaryLength(C);
21
> BinaryCardinal(C);
64
> BinaryMinimumDistance(C);
9
> BinaryParameters(C);
[ 21, 64, 9 ]

> r := BinaryDimensionOfSpan(C);
> r;
9
> k := BinaryDimensionOfKernel(C);
> k;
3
> kernel, cosetRepresentatives := BinaryKernelCode(C);
> ((2^k)*(#cosetRepresentatives+1)) eq BinaryCardinal(C);
true
```

1.4 The Information Sets

`IsBinarySystematic(C)`

Return `true` if and only if the binary code C of length n is systematic, that is, there is a set of t coordinate positions $I \subseteq \{1, \ldots, n\}$ such that $|C| = |C_I| = 2^t$, where $C_I = \{v_I : v \in C\}$ and v_I denote the restriction of the vector v to the coordinates in I.

`BinaryInformationSpace(C)`

Given a systematic binary code C of cardinality 2^t, return the vector space $U = \mathbb{Z}_2^t$, which is the space of information vectors for the code C.

`BinaryInformationSet(C)`

Given a systematic binary code C of length n, return an information set for C. An information set for C is an ordered set of t coordinate positions $I \subseteq \{1, \ldots, n\}$ such that $|C| = |C_I| = 2^t$, where $C_I = \{v_I | v \in C\}$ and v_I denote the restriction of the vector v to the coordinates in I. The information set is returned as a sequence of t integers, giving the numbers of the columns that correspond to the information set.

`BinaryAllInformationSets(C)`

Given a systematic binary code C of length n, return all the possible information sets of C as a (sorted) sequence of sequences of column indices. Each inner sequence contains a set of t coordinate positions $I \subseteq \{1, \ldots, n\}$ such that $|C| = |C_I| = 2^t$, where $C_I = \{v_I : v \in C\}$ and v_I denote the restriction of the vector v to the coordinates in I.

`BinaryStandardForm(C)`

Given a systematic binary code C of length n, return the standard form D of C. A systematic binary code of cardinality 2^t is in standard form if the first t components of the codewords correspond to the information set. MAGMA returns one of the many codes in standard form which is isomorphic to C. (The same code is returned each time.) Thus, the effect of this function is to return a code D whose codewords come from the codewords of C with its coordinates permuted, so that the codewords of D restricted to the first t coordinates gives all the elements of the vector space $U = \mathbb{Z}_2^t$ which is the space of information vectors for the code C. Two values are returned:

(a) The standard form code D;

(b) An isomorphism from C to D.

Example H1E4

```
>
```

1.5 Operations on Codewords

`BinarySet(C)`

Given a binary code C, return the set of all codewords of C.

`BinaryRandom(C)`

Return a random codeword of the binary code C.

Example H1E5

```
> C := LinearCode(Matrix(Integers(4), [[1,0,3,0,3], [0,1,0,2,3]]));
> HasLinearGrayMapImage(C);
false
> Cb := BinaryCode(C);
> kernel, cosetRepresentatives := BinaryKernelCode(Cb);
> kernel;
[10, 2, 4] Quasicyclic of degree 5 Linear Code over GF(2)
Generator matrix:
[1 1 0 0 1 1 0 0 1 1]
[0 0 1 1 0 0 0 0 1 1]
> cosetRepresentatives;
[
    (0 1 0 0 1 0 0 0 1 0),
    (0 0 0 1 0 0 1 1 1 0),
    (0 1 0 1 1 0 1 1 1 1)
]
> BinarySet(Cb) eq Set(GrayMapImage(C));
true

> c := BinaryRandom(Cb);
> IsInBinaryCode(Cb, c);
true
```

1.6 Membership and Equality

`IsInBinaryCode(C, u)`

Return `true` if and only if the vector u of $V = \mathbb{Z}_2^n$ belongs to the binary code C of length n.

`IsNotInBinaryCode(C, u)`

Return `true` if and only if the vector u of $V = \mathbb{Z}_2^n$ does not belong to the binary code C of length n.

`IsBinarySubset(C, D)`

Return `true` if and only if the binary code C is a subcode of the binary code D.

`IsBinaryNotSubset(C, D)`

Return `true` if and only if the binary code C is not a subcode of the binary code D.

`IsBinaryEqual(C, D)`

Return `true` if and only if the binary codes C and D are equal.

`IsBinaryNotEqual(C, D)`

Return `true` if and only if the binary codes C and D are not equal.

Example H1E6

```
> V  := VectorSpace(GF(2), 5);
> C1 := BinaryCode([V![1,1,1,1,1],V![0,0,0,0,0],
                    V![1,1,0,0,0],V![1,0,1,1,1]]);
> C2 := BinaryCode(Matrix([V![1,1,1,1,1],V![0,0,0,0,0],
                           V![1,1,0,0,0],V![1,0,1,1,1]]));
> IsBinaryEqual(C1, C2);
true

> C3 := BinaryRandomCode(8, 64, 2);
> C4 := BinaryRandomCode(8, 32);
> IsBinarySubset(C3, C4);
false

> C5 := BinaryCode(C4`Kernel);
> IsBinarySubset(C5, C4);
true
```

1.7 Properties of Binary Codes

`IsBinaryCode(C)`

Return `true` if and only if C is a binary code.

`IsBinaryLinearCode(C)`

`IsZ2LinearCode(C)`

Return `true` if and only if C is a binary linear code.

`IsZ4LinearCode(C, p)`

Given a binary code C of length n and a permutation p, return `true` if and only if the binary code $p(C) = \{p(c) \mid c \in C\}$ is the image under the Gray map of a code over $\mathbb{Z}_4$, or equivalently, a quaternary linear code.

`IsZ2Z4LinearCode(C, α, p)`

Given a binary code C of length n, a positive integer α such that $\alpha \leq n$, and a permutation p, return `true` if and only if the binary code $p(C) = \{p(c) \mid c \in C\}$ is the image under the Gray map of a $\mathbb{Z}_2\mathbb{Z}_4$-additive code over $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, where $\beta = (n - \alpha)/2$.

`IsBinaryDistanceInvariant(C)`

Return `true` if and only if the binary code C is distance invariant. Note that any linear code, the Gray map image of any code over $\mathbb{Z}_4$, or the Gray map image of any $\mathbb{Z}_2\mathbb{Z}_4$-additive code over $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ is distance invariant.

Example H1E7

```
> V  := VectorSpace(GF(2), 3);
> C1 := BinaryCode([V!0,V![0,1,0],V![0,0,1]]);
> IsBinaryCode(C1);
true
> IsBinaryLinearCode(C1);
false

> V  := RSpace(IntegerRing(4), 4);
> C  := Z2Z4AdditiveCode([V![2,2,1,1],V![0,2,1,2],
                          V![2,2,2,2],V![2,0,1,1]] : Alpha:=2);
> Cb := BinaryCode(C);
> p1 := Sym(BinaryLength(Cb))!(1,2);
> Q  := RandomLinearCode(IntegerRing(4), 5, 2);
> Qb := BinaryCode(Q);
> p2 := Sym(BinaryLength(Qb))!(1,2);
> IsZ2Z4LinearCode(Cb, 2, p1) and IsZ4LinearCode(Qb, p2);
true

> IsBinaryDistanceInvariant(C1);
false
> IsBinaryDistanceInvariant(Cb);
true
> IsBinaryDistanceInvariant(Qb);
true
```

1.8 The Weight and Distance Distribution

`BinaryMinimumWeight(C)`

Given a binary code C, return the minimum (Hamming) weight of the words belonging to the code C.

`BinaryMinimumDistance(C)`

Given a binary code C, return the minimum (Hamming) distance of the words belonging to the code C.

Note that for distance invariant codes, the minimum weight and distance coincide.

`BinaryMinimumWeightWord(C)`

Given a binary code C, return one codeword of C having minimum weight.

`BinaryMinimumWeightWords(C)`

Given a binary code C, return the set of all codewords of C having minimum weight.

`BinaryMinimumDistanceWord(C, u)`

Given a binary code C and a codeword u, return one codeword of C at minimum distance of u.

`BinaryMinimumDistanceWords(C, u)`

Given a binary code C and a codeword u, return the set of all codewords of C at minimum distance of u.

`BinaryWeightDistribution(C)`

Determine the weight distribution of the binary code C. The distribution is returned in the form of a sequence of tuples, where the i-th tuple contains the i-th weight, w_i say, and the number of codewords having weight w_i.

`BinaryDistanceDistribution(C)`

Determine the distance distribution of the binary code C. The distribution is returned in the form of a sequence of tuples, where the i-th tuple contains the i-th distance, d_i say, and the number of ordered pairs of codewords at distance d_i apart, divided by the number of codewords of C.

Note that the translated code $C+u$, where u is a vector from the ambient space $V = \mathbb{Z}_2^n$ of C, has the same distance distribution as C. Moreover, for distance invariant codes, the weight and distance distribution coincide.

`BinaryFormalDualWeightDistribution(C)`

Determine the weight distribution of the formal dual of the binary code C as long as the cardinal of the code C is a power of two. The distribution is returned in the form of a sequence of tuples, where the i-th tuple contains the i-th weight, w_i say, and the number of codewords having weight w_i.

`BinaryWeightDistribution(C, u)`

Determine the weight distribution of the translated code $C + u$ of the binary code C, where u is a vector from the ambient space $V = \mathbb{Z}_2^n$ of C. The distribution is returned in the form of a sequence of tuples, where the i-th tuple contains the i-th weight, w_i say, and the number of codewords having weight w_i.

Note that the weight distribution of the translated code $C + u$ coincides with the distance distribution of C to u given by the function `BinaryDistanceDistribution(C, u)`.

`BinaryDistanceDistribution(C, u)`

Determine the distance distribution of the binary code C to u, where u is a vector from the ambient space $V = \mathbb{Z}_2^n$ of C. The distribution is returned in the form of a sequence of tuples, where the i-th tuple contains the i-th distance, d_i say, and the number of codewords at distance d_i of u.

Note that the distance distribution of C to u coincides with the weight distribution of the translated code $C + u$ given by the function `BinaryWeightDistribution(C, u)`.

`BinaryWeightWords(C, w)`

Given a binary code C and an integer w, return the set of all codewords of C having weight w.

`BinaryWeightNumberOfWords(C, w)`

Given a binary code C and an integer w, return the number of codewords of C having weight w.

`BinaryDistanceWords(C, d, u)`

Given a binary code C, an integer d and a codeword u, return the set of all codewords of C at distance d of u.

`BinaryDistanceNumberOfWords(C, d, u)`

Given a binary code C, an integer d and a codeword u, return the number of codewords of C at distance d of u.

`BinaryMinimumDistanceGraph(C)`

Given a binary code C, return its minimum distance graph. The minimum distance graph of a code is a graph G, where the vertex set is the set of codewords of C, and two vertices are connected by an edge if the distance between them is the minimum distance of C.

Example H1E8

```
> V := VectorSpace(GF(2),31);
> C_kernel := SimplexCode(5);
> C_representatives := [
    V![ 0,0,1,0,0,0,1,1,1,0,0,1,1,0,1,0,0,1,1,1,1,0,0,0,1,0,1,1,1,1,0],
    V![ 0,1,0,1,1,0,1,0,1,0,1,1,1,1,0,0,1,0,1,1,1,0,1,0,0,1,1,1,1,0,1],
    V![ 0,0,0,0,0,1,1,1,0,0,0,1,1,1,0,1,1,0,1,0,0,0,1,1,1,1,0,1,0,1,1]
];
> C := BinaryCode(C_kernel, C_representatives);
> BinaryMinimumWeight(C);
10;
> BinaryMinimumDistance(C);
8
> IsBinaryDistanceInvariant(C);
false

> BinaryMinimumWeightWords(C);
{
    (1 0 0 0 0 0 1 0 1 0 1 1 0 1 1 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0),
    (0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 0 1 1 0 0 0 1 1 0 0 0 1 0 1 0),
    (0 1 1 1 0 0 1 0 0 0 0 1 0 1 0 0 1 1 0 0 0 1 0 0 0 0 0 1 0 0 0)
}
> BinaryMinimumDistanceWords(C, V!0) eq BinaryMinimumWeightWords(C);
true
> atMinDistance1 := BinaryMinimumDistanceWords(C, C_representatives[1]);
> atMinDistance1;
{
    (0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 0 1 1 1 1 1 1)
}
> atMinDistance2 := { c : c in BinarySet(C) |
                        Distance(c, C_representatives[1]) eq 8 };
> atMinDistance2;
{
    (0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 0 1 1 1 1 1 1)
}
> atMinDistance1 eq atMinDistance2;
true

> weightDistribution := BinaryWeightDistribution(C);
> weightDistribution;
[ <0, 1>, <10, 3>, <11, 4>, <12, 9>, <13, 6>, <14, 15>, <15, 8>, <16, 50>,
```

```
<17, 8>, <18, 13>, <19, 4>, <20, 3>, <21, 2>, <22, 1>, <24, 1> ]
> &+[wD[2] : wD in weightDistribution ] eq BinaryCardinal(C);
true
> [BinaryWeightNumberOfWords(C, wD[1]) : wD in weightDistribution ];
[ 1, 3, 4, 9, 6, 15, 8, 50, 8, 13, 4, 3, 2, 1, 1 ]

> distanceDistribution := BinaryDistanceDistribution(C);
> distanceDistribution;
[ <0, 1>, <8, 1/2>, <9, 1>, <10, 3/2>, <11, 5>, <12, 8>, <13, 8>,
<14, 15/2>, <15, 12>, <16, 50>, <17, 13>, <18, 13/2>, <19, 7>, <20, 4>,
<21, 2>, <22, 1/2>, <24, 1/2> ]
> &+[dD[2] : dD in distanceDistribution ] eq BinaryCardinal(C);
true
> [&+[BinaryDistanceNumberOfWords(C, dD[1], c) : c in BinarySet(C)]/BinaryCardinal(C)
                                              : dD in distanceDistribution];
[ 1, 1/2, 1, 3/2, 5, 8, 8, 15/2, 12, 50, 13, 13/2, 7, 4, 2, 1/2, 1/2 ]

> distanceDistribution := BinaryDistanceDistribution(C, C_representatives[1]);
> distanceDistribution;
[ <0, 1>, <8, 1>, <9, 1>, <10, 2>, <11, 2>, <12, 10>, <13, 7>, <14, 8>,
<15, 8>, <16, 61>, <17, 7>, <18, 6>, <19, 6>, <20, 6>, <21, 1>, <24, 1> ]
> &+[dD[2] : dD in distanceDistribution ] eq BinaryCardinal(C);
true
> [BinaryDistanceNumberOfWords(C, dD[1], C_representatives[1]) :
      dD in distanceDistribution ];
[ 1, 1, 1, 2, 2, 10, 7, 8, 8, 61, 7, 6, 6, 6, 1, 1 ]
```

`BinaryVectorDistanceDistribution(C : parameters)`

`MaximumTime`	RngIntElt	*Default* : ∞
`IsDistanceInvariant`	BoolElt	*Default* : `false`

Given a binary code C of length n, with ambient space $V = \mathbb{Z}_2^n$, attempt to determine the distance distribution of all vectors in V to C. The distance between C and a vector $u \in V$ is the Hamming weight of a vector of minimum weight in $C + u$. The distribution is returned as a sequence of pairs comprising a distance d and the number of vectors in V that are distance d from C, divided by the number of codewords of C.

When C is linear, it returns the same as the function `CosetDistanceDistribution(C)` and `true`. When C is nonlinear and it is known that it is distance invariant (for example, when it is the Gray map image of a code over $\mathbb{Z}_4$ or a $\mathbb{Z}_2\mathbb{Z}_4$-additive code), the parameter `IsDistanceInvariant` can be assigned to `true`.

Note that this function is only applicable when V and C are small. When C is nonlinear, the parameter `MaximumTime` sets a time limit (in seconds of "user time") after which the calculation is aborted. In this case, if the computation has not finished, the function returns a partial distance distribution and `false`, otherwise it returns the complete distance distribution and `true`. Note that, in the first case, the maximum distance in the distance distribution gives a lower bound of the covering radius; and in the second case, it gives exactly the covering radius. The default value of the parameter `MaximumTime` is infinite, when there is no restriction on time.

`BinaryCoveringRadius(C : parameters)`

`MaximumTime`	RngIntElt	*Default* : ∞
`IsDistanceInvariant`	BoolElt	*Default* : `false`

Given a binary code C of length n, with ambient space $V = \mathbb{Z}_2^n$, attempt to compute the covering radius of C. The covering radius of C is the smallest integer ρ such that all vectors in V are within Hamming distance ρ of some codeword, that is, $\rho = \max\{d(u, C) : u \in V\}$, where $d(u, C)$ is the distance between C and the vector $u \in V$. The function returns the covering radius ρ, a word at distance ρ of C, and `true`.

When C is linear, it returns the same as the function `CoveringRadius(C)`. When C is nonlinear and it is known that it is distance invariant (for example, when it is the Gray map image of a code over $\mathbb{Z}_4$ or a $\mathbb{Z}_2\mathbb{Z}_4$-additive code), the parameter `IsDistanceInvariant` can be assigned to `true`.

Note that this function is only applicable when V and C are small. When C is nonlinear, the parameter `MaximumTime` sets a time limit (in seconds of "user time") after which the calculation is aborted. In this case, if the computation has not finished, the function returns a lower bound of the covering radius, ρ_L, a word at distance ρ_L of C, and `false`. The default value of the parameter `MaximumTime` is infinite, when there is no restriction on time.

`BinaryCoveringRadiusBounds(C)`

Given a binary code C, return a lower bound and an upper bound of the covering radius of C. These lower and upper bounds are given by the covering radius of the linear span and kernel of C, respectively. Note that $\rho_{\langle C\rangle} \leq \rho \leq \rho_{K(C)}$, where ρ, $\rho_{K(C)}$ and $\rho_{\langle C\rangle}$ are the covering radius of the binary code C, the kernel $K(C)$ and the span $\langle C\rangle$, respectively.

Example H1E9

We construct the binary code of length 16 given by the Gray map image of a Preparata code of length 8 over $\mathbb{Z}_4$.

```
> C := BinaryCode(PreparataCode(3));
> time vDD := BinaryVectorDistanceDistribution(C);
Time: 3.270
> time vDD := BinaryVectorDistanceDistribution(C : IsDistanceInvariant := true);
Time: 0.410
> vDD;
[ <0, 1>, <1, 16>, <2, 120>, <3, 112>, <4, 7> ]
```

From the size of the code we know C has 256 translates which are disjoint because the code is distance invariant. The translate distance distribution tells us that there are 16 translates at distance 1 from C, 120 translates are distance 2, 112 are distance 3 and 7 are distance 4. We confirm that all translates are represented in the distribution.

The covering radius gives the maximum distance of any translate from the code, and, from the translate distance distribution, we see that this maximum distance is indeed 4.

```
> 2^BinaryLength(C)/BinaryCardinal(C);
256
> &+ [ t[2] : t in vDD ];
256
> BinaryCoveringRadius(C : IsDistanceInvariant := true);
4 (0 0 1 0 0 0 0 0 0 0 1 0 1 0 1 0)
true
> V := VectorSpace(GF(2),16);
> BinaryWeightDistribution(C, V![0,0,1,0,0,0,0,0,0,0,1,0,1,0,1,0]);
[ <4, 20>, <6, 48>, <8, 120>, <10, 48>, <12, 20> ]
```

We construct the binary code of length 32 given by the Gray map image of a Preparata code of length 16 over $\mathbb{Z}_4$.

```
> C := BinaryCode(PreparataCode(4));
> BinaryCoveringRadius(C : MaximumTime := 5);
3 (0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
false
> BinaryCoveringRadius(C : MaximumTime := 50);
4 (0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
false
> BinaryCoveringRadiusBounds(C);
2 10
```

1.9 Union, Intersection and Dual

`BinaryUnion(C,D)`

Return the union of the binary codes C and D, both of the same length.

`BinaryIntersection(C,D)`

Return the intersection of the binary codes C and D, both of the same length.

`BinaryDual(C)`

Return the dual of the binary code C of length n. The dual consists of all vectors in the vector space $V = \mathbb{Z}_2^n$ which are orthogonal to all codewords of C.

Example H1E10

We verify some simple results from the union, intersection and dual of binary codes.

```
> C1 := BinaryRandomCode(5,4);
> C2 := BinaryRandomCode(5,4);
> C1UnionC2 := BinaryUnion(C1,C2);
> C1InterC2 := BinaryIntersection(C1,C2);
> BinaryCardinal(C1) + BinaryCardinal(C2) eq
> BinaryCardinal(C1UnionC2) + BinaryCardinal(C1InterC2);
true

> U := BinaryUniverseCode(5);
> IsBinaryEqual(BinaryUnion(C1,U),U);
true
> IsBinaryEqual(BinaryIntersection(C1,U),C1);
true

> Z := BinaryZeroCode(5);
> IsBinaryEqual(BinaryUnion(C1,Z),C1);
true
> IsBinaryEqual(BinaryIntersection(C1,Z),Z);
true

> V := VectorSpace(GF(2),5);
> C := BinaryCode([V![1,0,0,1,0],V![1,1,1,1,1],V![0,0,0,0,0],V![0,0,0,0,1]]);
> CL1 := BinaryCode([V![0,0,0,0,0],V![0,1,1,0,1],V![0,0,1,1,0],V![0,1,0,1,1]]);
> CL2 := LinearCode(sub< V | [1,0,0,0,1],[0,1,0,0,1],[0,0,1,0,1],[0,0,0,1,1]>);
> BinaryDual(C) eq Dual(BinarySpanCode(C));
true
> BinaryDual(CL1) eq Dual(BinarySpanCode(CL1));
true
> BinaryDual(CL2) eq Dual(BinarySpanCode(CL2));
true
```

```
> BinaryDual(CL1) eq Dual(BinaryKernelCode(CL1));
true
> BinaryDual(CL2) eq Dual(BinaryKernelCode(BinaryCode(CL2)));
true
```

1.10 New Codes from Existing

`BinaryDirectSum(C, D)`

Given binary codes C and D, construct the direct sum of C and D. The direct sum is a binary code that consists of all vectors of the form (u, v), where $u \in C$ and $v \in D$.

`BinaryDirectSum(Q)`

Given a sequence of binary codes $Q = [C_1, \ldots, C_r]$, construct the direct sum of all these binary codes C_i, $1 \leq i \leq r$. The direct sum is a binary code that consists of all vectors of the form $(u_1, \ldots, u_r)$, where $u_i \in C_i$, $1 \leq i \leq r$.

`BinaryExtendCode(C)`

Given a binary code C form a new binary code C' from C by adding the appropriate extra coordinate to each vector of C such that the sum of the coordinates of the extended vector is zero.

`BinaryPunctureCode(C, i)`

Given a binary code C of length n and an integer i, $1 \leq i \leq n$, construct a new binary code C' by deleting the i-th coordinate from each codeword of C.

`BinaryPunctureCode(C, S)`

Given a binary code C of length n and a set S of distinct integers $\{i_1, \ldots, i_r\}$ each of which lies in the range $[1, n]$, construct a new binary code C' by deleting the components i_1, $\ldots$, i_r from each codeword of C.

`BinaryShortenCode(C, i)`

Given a binary code C of length n and an integer i, $1 \leq i \leq n$, construct a new binary code from C by selecting only those codewords of C having a zero as their i-th component and deleting the i-th component from these codewords. Thus, the resulting code will have length $n - 1$.

`BinaryShortenCode(C, S)`

Given a binary code C of length n and a set S of distinct integers $\{i_1, \ldots, i_r\}$ each of which lies in the range $[1, n]$, construct a new binary code from C by selecting only those codewords of C having zeros in each of the coordinate positions $i_1, \ldots, i_r$, and deleting these components. Thus, the resulting code will have length $n - r$.

`BinaryPlotkinSum(C, D)`

Given binary codes C and D both of the same length, construct the Plotkin sum of C and D. The Plotkin sum is a binary code that consists of all vectors of the form $(u, u + v)$, where $u \in C$ and $v \in D$.

Example H1E11

We combine binary codes in different ways and look at the length of the new binary codes.

```
> C1 := BinaryRandomCode(5,4);
> C2 := BinaryRandomCode(7,3);
> BinaryLength(C1);
5
> BinaryLength(C2);
7

> C3 := BinaryDirectSum(C1,C2);
> BinaryLength(C3);
12
> C4 := BinaryDirectSum([C1,C2,C3]);
> BinaryLength(C4);
24
> C5 := BinaryExtendCode(C2);
> BinaryLength(C5);
8
> C6 := BinaryPunctureCode(C2,4);
> BinaryLength(C6);
6
> C7 := BinaryShortenCode(C2,{4,5});
> BinaryLength(C7);
5
> C8 := BinaryPlotkinSum(C2,C2);
> BinaryLength(C8);
14
```

1.11 Decoding

`BinaryCosetDecode(C, d, u : parameters)`

`MinWeigthKernel` RngIntElt *Default* : -

Given a binary code C, its minimum distance d and a vector u from the ambient space V of C, attempt to decode u with respect to C. If the decoding algorithm succeeds in computing a vector u' as the decoded version of u, then the function returns `true` and u'. If the decoding algorithm does not succeed in decoding u, then the function returns `false` and the zero vector.

The algorithm considers the linear code $C_u = C \cup (C + u)$ when C is linear, or the linear codes $K_0 = K \cup (K + u)$, $K_1 = K \cup (K + v_1 + u)$, ..., $K_t = K \cup (K + v_t + u)$, where K is the kernel of C and $C = \bigcup_{i=0}^{t}(K + v_i)$, when C is nonlinear. If C is linear and the minimum weight of C_u is less than d, then $u' = u + e$, where e is a word of minimum weight of C_u; otherwise, the decoding algorithm returns `false`. On the other hand, if C is nonlinear and the minimum weight of $\cup_{i=0}^{t} K_i$ is less than the minimum weight of K, then $u' = u + e$, where e is a word of minimum weight of $\cup_{i=0}^{t} K_i$; otherwise, the decoding algorithm returns `false`. If the parameter `MinWeightKernel` is not assigned, then the minimum weight of K is computed.

Example H1E12

We create a binary nonlinear code C and a vector c of C and then perturb c to a new vector u. We then decode u to find c again.

```
> V := VectorSpace(GF(2),31);
> C_kernel := SimplexCode(5);
> C_representatives := [
    V![ 0,0,1,0,0,0,1,1,1,0,0,1,1,0,1,0,0,1,1,1,1,0,0,0,1,0,1,1,1,1,0],
    V![ 0,1,0,1,1,0,1,0,1,0,1,1,1,1,0,0,1,0,1,1,1,0,1,0,0,1,1,1,1,0,1],
    V![ 0,0,0,0,0,1,1,1,0,0,0,1,1,1,0,1,1,0,1,0,0,0,1,1,1,1,0,1,0,1,1]
];
> C := BinaryCode(C_kernel, C_representatives);
> d := BinaryMinimumDistance(C);
> d;
8

> c := V ! [0,1,1,1,0,1,1,0,0,1,0,0,0,1,0,0,0,1,1,0,0,1,1,1,1,0,0,1,1,1,1,0,0];
> IsInBinaryCode(C, c);
true
> u := c;
> u[5] := u[5] + 1;
```

```
> u[12] := u[12] + 1;
> c;
(0 1 1 1 0 1 1 0 0 1 0 0 0 1 0 0 0 1 1 0 0 1 1 1 0 0 1 1 1 0 0)
> u;
(0 1 1 1 1 1 1 0 0 1 0 1 0 1 0 0 0 1 1 0 0 1 1 1 0 0 1 1 1 0 0)
> c-u;
(0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)

> isDecoded, cDecoded := BinaryCosetDecode(C, d, u);
> isDecoded;
true
> cDecoded eq c;
true
```

Chapter 2

Binary Codes Database

2.1 Introduction

MAGMA currently contains databases of the best known linear codes over $GF(q)$ for $q = 2, 3, 4, 5, 7, 8, 9$ (see [4, Chapter 148.13]). It also includes a database of the best known quantum codes (see [4, Chapter 153]). This chapter describes databases of known binary nonlinear codes. In this version, we include all binary 1-perfect codes of length 15 (Section 2.2); and all binary extended 1-perfect codes of length 16 (Section 2.2).

A *binary 1-perfect code* C of length n is a binary code, with minimum Hamming distance $d = 3$, such that all the vectors in $\mathbb{Z}_2^n$ are within distance one from a codeword. For any $t > 1$ there exists exactly one binary linear 1-perfect code of length $2^t - 1$, up to equivalence, which is the well known *Hamming code*. An *extended code* of the code C is a code resulting from adding an overall parity-check digit to each codeword of C.

2.2 Databases of binary 1-perfect codes of length 15 and binary extended 1-perfect codes of length 16

It is known that there are 5983 nonequivalent binary 1-perfect codes of length 15 [14, 15]. The distribution of these codes, according to the rank and dimension of the kernel, is summarised in the following table:

	Dimension of the kernel											
Rank	1	2	3	4	5	6	7	8	9	10	11	Total
11											1	1
12							12	3	3			18
13				224	262	176	28	13				703
14		163	1287	2334	941	129	8	1				4863
15	19	14	8	338	19							398
												5983

It is also known that there are 2165 nonequivalent binary extended 1-perfect codes of length 16 [14, 15]. The distribution of these codes according, to the rank and dimension of the kernel, is summarised in the following table:

	Dimension of the kernel											
Rank	1	2	3	4	5	6	7	8	9	10	11	Total
11											1	1
12							8	2	2			12
13				82	89	67	11	7				256
14		102	449	786	326	53	4	1				1721
15	18	14	8	123	12							175
												2165

Each code in the database is a record with the following information:

- `Length`: The length of the code.
- `Kernel`: A base of the linear subspace that represents the kernel, as a sequence of binary vectors.
- `CosetRepresentatives`: A sequence of coset representatives (without including the zero word), as a sequence of binary vectors.
- `WordsMinWeight`: The 35 codewords of weight 3 for binary 1-perfect codes, or the 140 codewords of weight 4 for binary extended 1-perfect codes, as a sequence of binary vectors.
- `KernelDimension`: The dimension of the kernel of the code.
- `Rank`: The dimension of the linear span of the code.
- `Type`: An integer number to identify the code. In this version, it has value 0.

- `SymGroup`: A sequence of generators of the group of symmetries.
- `SymOrder`: The order of the group of symmetries.
- `AutOrder`: The order of the automorphism group.

Each record in the database is determined by three parameters, (r, k, s), where r is the rank of the code, k is the dimension of the kernel of the code, and s is the position of the code among all codes having the same rank and dimension of the kernel.

`BinaryPerfectCodesLength15Database()`

`BPC15Database()`

Returns the database of all nonequivalent binary 1-perfect codes of length 15. There are exactly 5983 codes in this database.

`BinaryExtendedPerfectCodesLength16Database()`

`BEPC16Database()`

Returns the database of all nonequivalent binary extended 1-perfect codes of length 16. There are exactly 2165 codes in this database.

`BCDHeaderInformation(DB)`

Print header information (version, length, number of codewords and number of codes) of the database DB.

`BCDDatabaseInformation(DB)`

Print a table with the distribution of the codes in the database DB, classified by their rank and dimension of the kernel.

`BCDRecordEmpty(DB)`

Return an empty record of the database DB.

Example H2E1

```
> DB := BPC15Database();

> BCDHeaderInformation(DB);
Database of binary 1-perfect codes of length 15. Version 1.5
There are 5983 codes in the database.
Each code has 2048 binary codewords.
```

```
> BCDDatabaseInformation(DB);
                      Dimension of the kernel
Rank   |    1    2    3    4    5    6    7    8    9   10   11 |  Total
-------------------------------------------------------------------------
   11  |                                                       1 |      1
   12  |                                   12    3    3          |     18
   13  |                 224  262  176   28   13                |    703
   14  |       163 1287 2334  941  129    8    1                |   4863
   15  |   19   14    8  338   19                                |    398
-------------------------------------------------------------------------
                                                                    5983
> R := BCDRecordEmpty(DB);
> Names(R);
[ Length, Kernel, CosetRepresentatives, WordsMinWeight, KernelDimension,
Rank, Type, SymGroup, SymOrder, AutOrder ]
```

`BCDDimensionsOfSpan(DB)`

`BCDDimensionsOfSpan(DB, k)`

`BCDRanks(DB)`

`BCDRanks(DB, k)`

Returns the sequence of ranks for which there is at least one code of that rank in the database DB. If a second parameter k is specified, then it returns the sequence of ranks for which there is at least one code of dimension of the kernel k and that rank in the database DB.

`BCDDimensionsOfKernel(DB)`

`BCDDimensionsOfKernel(DB, r)`

Returns the sequence of dimensions of the kernel for which there is at least one code of that dimension in the database DB. If a second parameter r is specified, then it returns the sequence of dimensions of the kernel for which there is at least one code of rank r and that dimension of the kernel in the database DB.

`BCDNumberOfCodes(DB, r, k)`

`BCDNumberOfCodes(DB, r)`

Returns the number of codes of rank r and dimension of the kernel k in the database DB. If k is omitted, then it returns the number of codes of rank r.

`BCDRecord(DB, r, k, s)`
`BCDRecord(DB, r, k)`
`BCDRecord(DB, r)`

Returns the database record determined by the parameters (r, k, s), where $1 \leq s \leq$ `BCDNumberOfCodes(DB,r,k)`, that is, the s-th database record containing a code of rank r and dimension of the kernel k. If s is omitted, then it returns the sequence of database records corresponding to all codes of rank r and dimension of the kernel k. If s and k are omitted, then it returns the sequence of database records corresponding to all codes of rank r.

`BCDRandom(DB, r, k)`
`BCDRandom(DB, r)`
`BCDRandom(DB)`

Returns a random database record containing a code of rank r and dimension of the kernel k. If k is omitted, then it returns a random database record with a code of rank r. If k and r are omitted, then it returns a random database record in the database DB.

`BCDCode(R)`
`BCDCode(S, i)`

Given a database record R, returns the binary code determined by R. Given a sequence of database records S and an integer i, returns the binary code determined by the record $S[i]$.

Example H2E2

```
> DB := BEPC16Database();
> ranks := BCDRanks(DB);
> ranks;
[ 11, 12, 13, 14, 15 ]
> [BCDDimensionsOfKernel(DB,r) : r in ranks];
[
    [ 11 ],
    [ 7, 8, 9 ],
    [ 4, 5, 6, 7, 8 ],
    [ 2, 3, 4, 5, 6, 7, 8 ],
    [ 1, 2, 3, 4, 5 ]
]
> BCDNumberOfCodes(DB, 14, 4);
786
> R := BCDRecord(DB, 14, 4, 12);
> C := BCDCode(R);
```

```
> IsBinaryCode(C);
true
> IsBinaryExtendedPerfectCode(C);
true
```

`BCDKernel(R)`

`BCDKernel(S, i)`

Given a database record R, returns the kernel of the binary code determined by R, as a binary linear code. Given a sequence of database records S and an integer i, returns the kernel of the binary code determined by the record $S[i]$, as a binary linear code.

`BCDCosetRepresentatives(R)`

`BCDCosetRepresentatives(S, i)`

Given a database record R, returns the sequence of coset representatives of the binary code determined by R. Given a sequence of database records S and an integer i, returns the sequence of coset representatives of the binary code determined by the record $S[i]$.

`BCDRank(R)`

`BCDRank(S, i)`

`BCDDimensionOfSpan(R)`

`BCDDimensionOfSpan(S, i)`

Given a database record R, returns the rank of the binary code determined by R. Given a sequence of database records S and an integer i, returns the rank of the binary code determined by the record $S[i]$.

`BCDDimensionOfKernel(R)`

`BCDDimensionOfKernel(S, i)`

Given a database record R, returns the dimension of the kernel of the binary code determined by R. Given a sequence of database records S and an integer i, returns the dimension of the kernel of the binary code determined by the record $S[i]$.

`BCDMinimumWords(R)`

`BCDMinimumWords(S, i)`

Given a database record R, returns the sequence of codewords of minimum weight of the binary code determined by R. Given a sequence of database records S and an integer i, returns the sequence of codewords of minimum weight of the binary code determined by the record $S[i]$.

`BCDSymmetriesGroup(R)`

`BCDSymmetriesGroup(S, i)`

Given a database record R, returns the group of symmetries of the binary code determined by R. Given a sequence of database records S and an integer i, returns the group of symmetries of the binary code determined by the record $S[i]$.

`BCDSymmetryGroupOrder(R)`

`BCDSymmetryGroupOrder(S, i)`

Given a database record R, returns the order of the group of symmetries of the binary code determined by R. Given a sequence of database records S and an integer i, returns the order of the group of symmetries of the binary code determined by the record $S[i]$.

`BCDAutomorphismGroupOrder(R)`

`BCDAutomorphismGroupOrder(S, i)`

Given a database record R, returns the order of the automorphism group of the binary code determined by R. Given a sequence of database records S and an integer i, returns the order of the automorphism group of the binary code determined by the record $S[i]$.

`BCDInvariants(R)`

`BCDInvariants(S, i)`

Given a database record R, returns the invariants contained in the following fields: `Length`, `KernelDimension`, `Rank`, `SymOrder`, `AutOrder`, of the binary code determined by R, as a sequence of integers. Given a sequence of database records S and an integer i, returns the above invariants of the binary code determined by the record $S[i]$, as a sequence of integers.

`BCDClassification(DB, C)`

Given a binary 1-perfect code C of length 15 or a binary extended 1-perfect code C of length 16, returns the parameters (r, k, s) of the unique binary 1-perfect code or binary extended 1-perfect code equivalent to C in the database DB.

The verbose flag `NonlinearDB` is 0 by default. If it is set to `1`, then the process used to classify the code is printed. The computation is slow for codes with small kernel or when there are a lot of codes with the same rank r and dimension of the kernel k as the code C.

Example H2E3

```
> H := HammingCode(GF(2), 4);
> Length(H);
15
> C := BinaryCode(H);
> BCDClassification(DB, C);
[ 11, 11, 1 ]

> // An example of slow computation
> SetVerbose("NonlinearDB", 1);
> R := BCDRandom(DB, 14);
> BCDInvariants(R);
[ 15, 5, 14, 4, 512 ]
> C := BCDCode(R);
> BCDClassification(DB, C);
Computing invariants...
Rank 14, kernelDim 5 candidates: 941
Symmetry group order 4 candidates: 485
STS candidates: 56
Automorphism group order 512 candidates: 30
Took 165.940 seconds
[ 14, 5, 585 ]
```

Bibliography

[1] J. Borges, C. Fernández, J. Pujol, J. Rifà and M. Villanueva, "On $\mathbb{Z}_2\mathbb{Z}_4$-linear codes and duality," *V Jornadas de Matemática Discreta y Algorítmica*, Soria (Spain), July 11-14, pp. 171-177, 2006.

[2] J. Borges, C. Fernández, J. Pujol, J. Rifà, and M. Villanueva, "$\mathbb{Z}_2\mathbb{Z}_4$-linear codes: generator matrices and duality," *Designs, Codes and Cryptography*, vol. 54, no. 2, pp. 167-179, 2010.

[3] A. Brouwer, "Table of general binary codes," http://www.win.tue.nl/ aeb/codes/binary.html#toc1

[4] J. J. Cannon and W. Bosma (Eds.) *Handbook of* Magma *Functions*, Edition 2.13, 4350 pages, 2006.

[5] P. Delsarte, "An algebraic approach to the association schemes of coding theory," *Philips Research Rep. Suppl.*, vol. 10, 1973.

[6] C. Fernández-Córdoba, J. Pujol, and M. Villanueva, "On rank and kernel of $\mathbb{Z}_4$-linear codes," *Lecture Notes in Computer Science*, n. 5228, 2008.

[7] C. Fernández-Córdoba, J. Pujol, and M. Villanueva, "$\mathbb{Z}_2\mathbb{Z}_4$-linear codes: rank and kernel," *Designs, Codes Cryptography*, vol. 56, no. 1, pp. 43-59, 2010.

[8] A.R. Hammons, P.V. Kumar, A.R. Calderbank, N.J.A. Sloane and P. Solé, "The $\mathbb{Z}_4$-linearity of kerdock, preparata, goethals and related codes," *IEEE Trans. on Information Theory*, vol. 40, pp. 301-319, 1994.

[9] O. Heden, "On perfect p-ary codes of length $p+1$", *Designs, Codes and Cryptography*, vol. 46, no. 1, pp. 45-56, 2008.

[10] O. Heden, "Perfect codes from the dual point of view I", *Discrete Mathematics*, vol. 308, no. 24, pp. 6141-6156, 2008.

[11] J. A. Howell, "Spans in the module $\mathbb{Z}_m^s$," *Linear and Multilinear Algebra*, 19, pp. 67-77, 1986.

[12] D. S. Krotov, "$\mathbb{Z}_4$-linear Hadamard and extended perfect codes," in *WCC2001, International Workshop on Coding and Cryptography*, ser. Electron. Notes Discrete Math., 2001, vol. 6, pp. 107–112.

[13] S. Litsyn, E.M. Rains and N.J.A Sloane, "Table of nonlinear binary codes," `http://www.eng.tau.ac.il/ litsyn/tableand/`

[14] P. R. J. Östergård and O. Pottonen, "The perfect binary one-error-correcting codes of length 15: part I-classification," *IEEE Trans. on Information Theory*, vol. 55, no. 10, pp. 4657-4660, 2009.

[15] P. R. J. Östergård, O. Pottonen, and K. T. Phelps, "The perfect binary one-error-correcting codes of length 15: Part IIproperties," *IEEE Trans. on Information Theory*, vol. 56, no. 6, pp. 2571-2582, 2010.

[16] K. T. Phelps, J. Rifà, and M. Villanueva, "Kernels and p-kernels of p^r-ary 1-perfect codes," *Designs, Codes and Cryptography*, vol. 37, no. 2, pp. 243-261, 2001.

[17] J. Pujol, J. Rifà, F. I. Solov'eva, "Quaternary Plotkin constructions and Quaternary Reed-Muller codes," *Lecture Notes in Computer Science* n. 4851, pp. 148-157, 2007.

[18] J. Pujol, J. Rifà, and F. I. Solov'eva, "Construction of $\mathbb{Z}_4$-linear Reed–Muller codes," *IEEE Trans. on Information Theory*, vol. 55, no. 1, pp. 99–104, 2009.

[19] J. Pujol, F. Zeng, and M. Villanueva, "Representation, constructions and minimum distance computation of binary nonlinear codes," in Proceedings of 19th International Conference on Aplications of Computer Algebra, Málaga, Spain, July 2-6, pp. 142-143, 2013.

[20] F. I. Solov'eva "On $\mathbb{Z}_4$-linear codes with parameters of Reed-Muller codes," *Problems of Information Transmission*, vol. 43(1), pp. 26-32, 2007.

[21] A. Storjohann and T. Mulders, "Fast algorithms for linear algebra modulo N," *Lecture Notes In Computer Science*, vol. 1461, pp. 139-150, 1998.

[22] M. Villanueva, F. Zeng, and J. Pujol, "Nonlinear Q-ary codes: Constructions and minimum distance computation," in Proceedings of Encuentros

de Álgebra Computacional y Aplicaciones, Barcelona, Spain, June 18-20, pp. 171-174, 2014.

[23] M. Villanueva, F. Zeng, and J. Pujol, "Efficient representation of binary nonlinear codes: constructions and minimum distance computation," accepted to be published in *Designs, Codes and Cryptography*, DOI: 10.1007/s10623-014-0028-4

[24] Z.-X. Wan, *Quaternary Codes*, World Scientific, 1997.