$\mathbb{Z}_2\mathbb{Z}_4$-ADDITIVE CODES

A MAGMA Package*

by

**Joaquim Borges, Cristina Fernández, Bernat Gastón,
Jaume Pujol, Josep Rifà and Mercè Villanueva**

Combinatoric, Coding and Security Group (CCSG)

Universitat Autònoma de Barcelona

Version 5.0

**Barcelona
June, 2022**

*This work has been partially supported by the Spanish MICINN under Grants PCI2006-A7-0616, MTM2006-03250, MTM2009-08435, TIN2010-17358, TIN2013-40524-P, TIN2016-77918-P and PID2019-104664GB-I00 (AEI / 10.13039/501100011033); and by the Catalan AGAUR under Grants 2009SGR1224 and 2014SGR-691.

Contents

Chapter 1

Preface

The *Combinatoric, Coding and Security Group* (CCSG) is a research group in the Department of Information and Communications Engineering (DEIC) at the Universitat Autònoma de Barcelona (UAB).

The research group CCSG has been uninterruptedly working since 1987 in several projects and research activities on Information Theory, Communications, Coding Theory, Source Coding, Cryptography, Electronic Voting, Network Coding, etc. The members of the group have been producing mainly results on optimal coding. Specifically, the research has been focused on uniformly-packed codes; perfect codes in the Hamming space; perfect codes in distance-regular graphs; the classification of optimal codes of a given length; and codes which are close to optimal codes by some properties, for example, Reed-Muller codes, Preparata codes, Kerdock codes and Hadamard codes.

Part of the research developed by CCSG deals with $\mathbb{Z}_2\mathbb{Z}_4$-linear codes. There is no symbolic software to work with these codes, so the members of CCSG have been developing this 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 current version of this package for $\mathbb{Z}_2\mathbb{Z}_4$-additive codes and this manual with the description of all functions can be downloaded from the web page `http://ccsg.uab.cat`. For any comment or further information about this package, you can send an e-mail to *support-ccsg@deic.uab.cat*.

The authors would like to thank Lorena Ronquillo and Jaume Pernas for their contributions developing some functions of Section 2.2.4, Roger Ten-Valls for the functions given in Section 2.2.3, Cristina Diéguez and Marta Pujol for helping in the development of some functions of Section 2.8, Daniel

Molinero for the implementation of the function to compute the covering radius, and Roland D. Barrolleta for the decoding functions given in Section 2.10. Finally, we would also like to thank Adrián Torres-Martín for his contributions to improve some parts of the package, to develop some functions given in Sections 2.2.4, and to add new methods to the functions in Section 2.8: two based on the Brouwer-Zimmermann algorithm and another one based on transforming the code into a suitable quaternary code.

Chapter 2

$\mathbb{Z}_2\mathbb{Z}_4$-additive codes

2.1 Introduction

MAGMA currently supports the basic facilities for linear codes over integer residue rings and galois rings [9, Chapter 161], including cyclic codes and the complete weight enumerator calculation. Moreover, specific functions for the special case of linear codes over $\mathbb{Z}_4$ (also called $\mathbb{Z}_4$-codes or quaternary linear codes), which are subgroups of $\mathbb{Z}_4^n$ are also supported [9, Chapter 162].

Linear codes over $\mathbb{Z}_4$ have been studied and became significant since, after applying the Gray map from $\mathbb{Z}_4$ to binary pairs, we obtain binary nonlinear codes better than any known binary linear code with the same parameters. More specifically, Hammons et. al. [13] show how to construct well known binary nonlinear codes like Kerdock codes and Delsarte-Goethals codes by applying the Gray map to linear codes over $\mathbb{Z}_4$. Further, they solve an old open problem on coding theory about that the weight enumerators of the nonlinear Kerdock and Preparata codes satisfy the MacWilliams identities. Later, several other binary nonlinear codes constructed using the Gray map on linear codes over $\mathbb{Z}_4$, and with the same parameters as some well known families of binary linear codes (for example, extended Hamming codes, Hadamard codes, and Reed-Muller codes) have been studied and classified [5, 15, 17, 20, 21, 24].

MAGMA also supports functions for additive codes over a finite field, which are a generalization of the linear codes over a finite field [9, Chapter 163] in a Hamming scheme. According to a more general definition of additive codes given by Delsarte in 1973 [10, 11], the additive codes are subgroups of the underlying abelian group in a translation association scheme. In the special case of a binary Hamming scheme, that is, when the underlying abelian group is of size $2^{n_{bin}}$, the only structures for the abelian group are

those of the form $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, with $\alpha + 2\beta = n_{bin}$. Therefore, the codes that are subgroups of $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ are also called additive codes. In order to distinguish them from the additive codes over a finite field, we call them $\mathbb{Z}_2\mathbb{Z}_4$-*additive codes*.

The $\mathbb{Z}_2\mathbb{Z}_4$-additive codes can be seen as a generalization of binary and quaternary linear codes, since they are subgroups of $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$. Note that when $\alpha = 0$, these codes are linear codes over $\mathbb{Z}_4$, and when $\beta = 0$, they are binary linear codes. As for linear codes over $\mathbb{Z}_4$, after applying the Gray map to the $\mathbb{Z}_4$ coordinates of a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, we obtain a binary code, which is not necessarily linear in general. The binary image of a $\mathbb{Z}_2\mathbb{Z}_4$-additive code under the extended Gray map is called $\mathbb{Z}_2\mathbb{Z}_4$-linear code. There are $\mathbb{Z}_2\mathbb{Z}_4$-linear codes in several important classes of binary codes. For example, $\mathbb{Z}_2\mathbb{Z}_4$-linear perfect single error-correcting codes (or 1-perfect codes) are found in [23] and fully characterized in [4]. Also, in subsequent papers [5, 18, 19], $\mathbb{Z}_2\mathbb{Z}_4$-linear extended 1-perfect and Hadamard codes are studied and classified. Therefore, $\mathbb{Z}_2\mathbb{Z}_4$-additive codes have allowed to classify more binary nonlinear codes, giving them an structure as linear codes over $\mathbb{Z}_4$ or $\mathbb{Z}_2\mathbb{Z}_4$-additive codes.

Part of the research developed by the Combinatorics, Coding and Security Group (CCSG) deals with linear codes over $\mathbb{Z}_4$, as well as $\mathbb{Z}_2\mathbb{Z}_4$-additive codes. Since there are no symbolic software to work with $\mathbb{Z}_2\mathbb{Z}_4$-additive codes, the members of CCSG have been developing this 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 $\mathbb{Z}_4$ to $\mathbb{Z}_2\mathbb{Z}_4$-additive codes, 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.

Let C be an $\mathbb{Z}_2\mathbb{Z}_4$-additive code. Since C is a subgroup of $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ for some non-negative integers α and β, it is isomorphic to an abelian structure $\mathbb{Z}_2^\gamma \times \mathbb{Z}_4^\delta$ for some non-negative integers γ and δ. Therefore, we have that the number of codewords is $|C| = 2^\gamma 4^\delta$ and $2^{\gamma+\delta}$ of them have order at most two. Let X (respectively Y) be the set of $\mathbb{Z}_2$ (respectively $\mathbb{Z}_4$) coordinate positions, so $|X| = \alpha$ and $|Y| = \beta$. Unless otherwise stated, the set X corresponds to the first α coordinates and Y corresponds to the last β coordinates. Call C_X (respectively C_Y) the code C restricted to the X (respectively Y) coordinates. Let C_b be the $\mathbb{Z}_2\mathbb{Z}_4$-additive subcode of C which contains exactly all codewords of order at most two and let κ be the dimension of $(C_b)_X$, which is a binary linear code. For the case $\alpha = 0$, we write $\kappa = 0$. Considering all these parameters, we say that C is of type $(\alpha, \beta; \gamma, \delta; \kappa)$. Note that the quaternary linear code C_Y is of type $(0, \beta; \gamma_Y, \delta; 0)$, where $0 \leq \gamma_Y \leq \gamma$, and

the binary linear code C_X is a code of length α and dimension γ_X, or equivalently of type $(\alpha, 0; \gamma_X, 0; \gamma_X)$, where $\kappa \leq \gamma_X \leq \gamma$. Any quaternary linear code of length β and type $2^\gamma 4^\delta$ is a $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(0, \beta; \gamma, \delta; 0)$.

A $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ can also be seen as a submodule of the $\mathbb{Z}_4$-module $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$. As a submodule, C is free if and only if $\gamma = 0$. Although C is not a free module in general, there exist $\{u_i\}_{i=1}^\gamma$ and $\{v_j\}_{j=1}^\delta$ such that every codeword in C can be uniquely expressible in the form

$$\sum_{i=1}^{\gamma} \lambda_i u_i + \sum_{j=1}^{\delta} \mu_j v_j,$$

where $\lambda_i \in \mathbb{Z}_2$ for all $1 \leq i \leq \gamma$, $\mu_j \in \mathbb{Z}_4$ for all $1 \leq j \leq \delta$ and u_i, v_j are codewords of order two and order four, respectively. The matrix of size $(\gamma+\delta)\times(\alpha+\beta)$ that has as rows the vectors $\{u_i\}_{i=1}^\gamma$ and $\{v_j\}_{j=1}^\delta$ is a generator matrix for C. Moreover, we can write this matrix as

$$\mathcal{G} = \left(\begin{array}{c|c} B_1 & 2B_3 \\ B_2 & Q \end{array} \right), \tag{2.1}$$

where B_1, B_2 are matrices over $\mathbb{Z}_2$ of size $\gamma \times \alpha$ and $\delta \times \alpha$, respectively; B_3 is a matrix over $\mathbb{Z}_4$ of size $\gamma \times \beta$ with all entries in $\{0, 1\} \subset \mathbb{Z}_4$; and Q is a matrix over $\mathbb{Z}_4$ of size $\delta \times \beta$ with quaternary row vectors of order four.

In this chapter, the term "code" will refer to a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, unless otherwise specified. In order to be able to use the functions for quaternary linear codes, from now on the $\mathbb{Z}_2\mathbb{Z}_4$-additive codes will be represented as quaternary linear codes replacing the ones by twos in the first α binary coordinates, so they will be subgroups of $\mathbb{Z}_4^{\alpha+\beta}$. However, these codes are not equivalent to the quaternary linear codes, since the inner product defined in $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ gives us that the dual code of a $\mathbb{Z}_2\mathbb{Z}_4$-additive code is not equivalent to the dual code of the corresponding quaternary linear code.

Finally, as general references on $\mathbb{Z}_2\mathbb{Z}_4$-additive codes, the reader is referred to [6, 7, 8], where most of the concepts on $\mathbb{Z}_2\mathbb{Z}_4$-additive codes implemented with the following functions are described in detail.

Note: Since the $\mathbb{Z}_2\mathbb{Z}_4$-additive codes are represented as quaternary linear codes, it is necessary to remember that for modules defined over rings with zero divisors, it is not possible to talk about the concept of dimension (the modules are not free). However, in MAGMA each code over such a ring has a unique generator matrix corresponding to the Howell form [14, 25]. The number of rows k in this generator matrix will be called the *pseudo-dimension* of the code. It should be noted that this pseudo-dimension is not invariant between equivalent codes, and so does not provide structural information like the dimension of a code over a finite field.

Note: In order to use the functions in Subsection 2.5.3, it is necessary to use MAGMA version 2.22 or later. It is also possible installing the package "Codes over Z4", which can be downloaded from the web page `http://ccsg.uab.cat`. The functions in this package expand the functionality for codes over $\mathbb{Z}_4$ in MAGMA. Specifically, there are functions to construct some families of codes over $\mathbb{Z}_4$, efficient functions for computing the rank and dimension of the kernel of any code over $\mathbb{Z}_4$, as well as general functions for computing coset representatives for a subcode in a code over $\mathbb{Z}_4$. There are also functions for computing the permutation automorphism group for Hadamard and extended perfect codes over $\mathbb{Z}_4$, and their orders. Functions related to the information space, information sets, syndrome space and coset leaders for codes over $\mathbb{Z}_4$ can also be found. Finally, functions to decode codes over $\mathbb{Z}_4$ by using different methods are also provided.

2.2 Construction of $\mathbb{Z}_2\mathbb{Z}_4$-Additive Codes

2.2.1 General $\mathbb{Z}_2\mathbb{Z}_4$-Additive Codes

`Z2Z4AdditiveCode(L, α)`

Create a $\mathbb{Z}_2\mathbb{Z}_4$-additive code $C \subseteq \mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$ given as a linear code over $\mathbb{Z}_4$ after replacing the ones in the first α coordinates by twos. The corresponding linear code over $\mathbb{Z}_4$ of C is obtained by using the function `LinearCode()` as a subspace of $\mathbb{Z}_4^{\alpha+\beta}$ generated by L, where

1. L is a sequence of elements of $\mathbb{Z}_4^{\alpha+\beta}$,
2. or, L is a subspace of $\mathbb{Z}_4^{\alpha+\beta}$,
3. or, L is a $m \times n$ matrix A over the ring $\mathbb{Z}_4$,
4. or, L is a quaternary linear code.

The $\mathbb{Z}_2\mathbb{Z}_4$-additive code C represented as a linear code over $\mathbb{Z}_4$ has the following attributes:

- `Alpha`: The length of the binary part of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code.
- `Length`: The length of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code.
- `Code`: The linear code over $\mathbb{Z}_4$ representing the $\mathbb{Z}_2\mathbb{Z}_4$-additive code, after replacing the ones in the first `Alpha` coordinates by twos.

- `MinimumLeeWeightLowerBound`: A lower bound on the minimum Lee weight of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code.
- `MinimumLeeWeightUpperBound`: An upper bound on the minimum Lee weight of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code.
- `MinimumLeeWeight`: The minimum Lee weight of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code.
- `MinimumLeeWeightWord`: A codeword of minimum Lee weight of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code.
- `LeeWeightDistribution`: The Lee weight distribution of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code.
- `CoveringRadius`: The covering radius of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code.

The first α coordinates of each element in L can be given as elements in $\{0,1\} \subset \mathbb{Z}_4$ or as elements in $\{0,2\} \subset \mathbb{Z}_4$. If the first α coordinates of each element in L are in $\{0,1\}$, then the ones are replaced by twos.

`Z2Z4AdditiveCode(C)`

Given a quaternary linear code C of length β and type $2^\gamma 4^\delta$, return the $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(0, \beta; \gamma, \delta; 0)$ corresponding to C.

`Z2Z4AdditiveCode(C)`

Given a binary linear code C of length α and dimension k, return the $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, 0; k, 0; k)$ corresponding to C. The code is represented as a linear code over $\mathbb{Z}_4$ after replacing the ones by twos.

`Z2Z4AdditiveCode(T)`

Given a sequence T of elements of the cartesian product set $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, return the $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ generated by T. The code is represented as a linear code over $\mathbb{Z}_4$ after replacing the ones in the first α coordinates by twos.

`Z2Z4AdditiveCode(M, N)`

Given a matrix M over $\mathbb{Z}_2$ with α columns and a matrix N over $\mathbb{Z}_4$ with β columns, return the $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ generated by the rows of the matrix $(M|N)$. When it is necessary, zero rows are added at the end of matrices M and N to assure that both have the same number of rows. The code is represented as a linear code over $\mathbb{Z}_4$ after replacing the ones in the first α coordinates by twos.

`Z2Z4AdditiveCode(C, D)`

Given a binary linear code C of length α and a quaternary linear code D of length β, return the $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ generated by the vectors of the form $(u|\mathbf{0})$ and $(\mathbf{0}|v)$, where $u \in C$ and $v \in D$. The code is represented as a linear code over $\mathbb{Z}_4$ after replacing the ones in the first α coordinates by twos. Note that the $\mathbb{Z}_2\mathbb{Z}_4$-additive code is separable.

Example H2E1

We define a $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ by giving a sequence of elements of $\mathbb{Z}_4^{\alpha+\beta}$, or equivalently a $m \times n$ matrix over $\mathbb{Z}_4$, and the length of the binary part of the code.

```
> Z4 := IntegerRing(4);
> R  := RSpace(Z4, 5);
> C1 := Z2Z4AdditiveCode([R![2,2,1,1,3], R![0,2,1,2,1],
                          R![2,2,2,2,2], R![2,0,1,1,1]], 2);
> C1;
(5, 64) Z2Z4-additive code of type (2, 3; 2, 2; 2)
Generator matrix:
[2 0 0 0 2]
[0 2 0 0 2]
[0 0 1 0 3]
[0 0 0 1 0]

> A  := Matrix(Z4, [[2,2,1,1,3],
                    [0,2,1,2,1],
                    [2,2,2,2,2],
                    [2,0,1,1,1]]);
> C2 := Z2Z4AdditiveCode(A, 2);
> C2;
(5, 64) Z2Z4-additive code of type (2, 3; 2, 2; 2)
Generator matrix:
[2 0 0 0 2]
[0 2 0 0 2]
[0 0 1 0 3]
[0 0 0 1 0]

> C1 eq C2;
true
```

Alternatively, we also define the same $\mathbb{Z}_2\mathbb{Z}_4$-additive codes by giving the first α coordinates of each vector as elements in $\{0, 1\}$ instead of in $\{0, 2\}$, replacing the twos by ones.

```
> C1b := Z2Z4AdditiveCode([R![1,1,1,1,3], R![0,1,1,2,1],
                           R![1,1,2,2,2], R![1,0,1,1,1]], 2);
> Ab  := Matrix(Z4, [[1,1,1,1,3],
                     [0,1,1,2,1],
                     [1,1,2,2,2],
                     [1,0,1,1,1]]);
```

```
> C2b := Z2Z4AdditiveCode(Ab, 2);
> (C1 eq C1b) and (C2 eq C2b);
true
```

Finally, we also define the previous $\mathbb{Z}_2\mathbb{Z}_4$-additive code from elements in the cartesian product set $\mathbb{Z}_2^2 \times \mathbb{Z}_4^3$.

```
> R2 := RSpace(IntegerRing(2), 2);
> R4 := RSpace(IntegerRing(4), 3);
> R2R4 := CartesianProduct(R2, R4);
> T := [R2R4!<R2![1,1], R4![1,1,3]>, R2R4!<R2![0,1], R4![1,2,1]>,
        R2R4!<R2![1,1], R4![2,2,2]>, R2R4!<R2![1,0], R4![1,1,1]>];
> C1car := Z2Z4AdditiveCode(T);
> C1 eq C1car;
true
```

Example H2E2

Any quaternary linear code and any binary linear code is also a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, since the $\mathbb{Z}_2\mathbb{Z}_4$-additive codes can be seen as a generalization of the quaternary linear codes, when $\alpha = 0$, and a generalization of the binary linear codes, when $\beta = 0$.

```
> B := RandomLinearCode(GF(2), 7, 3);
> B;
[7, 3, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 1]
[0 0 1 0 1 0 1]
[0 0 0 1 1 1 0]
> B_add := Z2Z4AdditiveCode(B);
> B_add;
(7, 8) Z2Z4-additive code of type (7, 0; 3, 0; 3)
Generator matrix:
[2 2 0 0 0 0 2]
[0 0 2 2 0 2 2]
[0 0 0 0 2 0 2]

> Q := RandomLinearCode(IntegerRing(4), 5, 2);
> Q;
((5, 4^2 2^0)) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 1 2]
[0 0 1 0 3]
> Q_add := Z2Z4AdditiveCode(Q);
> Q_add;
(5, 16) Z2Z4-additive code of type (0, 5; 0, 2; 0)
Generator matrix:
[1 2 0 0 0]
[0 0 1 0 2]
```

2.2.2 Some Trivial $\mathbb{Z}_2\mathbb{Z}_4$-Additive Codes

`Z2Z4AdditiveZeroCode(`α`,`β`)`

Given two non-negative integers α and β, return the $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; 0, 0; 0)$ consisting of only the zero codeword.

`Z2Z4AdditiveRepetitionCode(`α`,`β`)`

Given two non-negative integers α and β, return the $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; 1, 0; 1)$ generated by the vector $(1, \ldots, 1|2, \ldots, 2)$.

`Z2Z4AdditiveUniverseCode(`α`,`β`)`

Given two non-negative integers α and β, return the generic $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \alpha, \beta; \alpha)$ consisting of all possible codewords.

`Z2Z4AdditiveEvenWeightCode(`α`,`β`)`

Given two non-negative integers α and β, return the $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \alpha - 1, \beta; \alpha - 1)$ such that all vectors have even weight. If $\beta = 0$, then α must be greater than or equal to 2.

`RandomZ2Z4AdditiveCode(`α`,`β`,`γ`,`δ`,`κ`)`

Given two, three, four or five non-negative integers α, β, γ, δ, κ, return a random $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \gamma, \delta; \kappa)$. Note that there exists a $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ if and only if $\alpha, \beta, \gamma, \delta, \kappa \geq 0$, $\alpha + \beta > 0$, $\kappa \leq \min(\alpha, \gamma)$ and $0 < \delta + \gamma \leq \beta + \kappa$. When there are less than five integers, they correspond to the first ones in the above order and the rest are computed randomly.

Example H2E3

The zero $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; 0, 0; 0)$ is contained in every $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \gamma, \delta; \kappa)$, and similarly every $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ is contained in the universe $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \alpha, \beta; \alpha)$. The image under the Gray map of the even $\mathbb{Z}_2\mathbb{Z}_4$-additive code is the $[\alpha + 2\beta, \alpha + 2\beta - 1, 2]$ binary even code.

```
> a := 2; b := 3;
> U := Z2Z4AdditiveUniverseCode(a, b);
> Z := Z2Z4AdditiveZeroCode(a, b);
> R := RandomZ2Z4AdditiveCode(a, b);
> E := Z2Z4AdditiveEvenWeightCode(a, b);
> (Z subset R) and (R subset U);
true
> (Z subset E) and (E subset U);
true
> _, binaryE := HasLinearGrayMapImage(E);
> binaryE eq EvenWeightCode(a + 2*b);
true
```

2.2.3 General $\mathbb{Z}_2\mathbb{Z}_4$-Additive Cyclic Codes

A $\mathbb{Z}_2\mathbb{Z}_4$-additive code C is cyclic if for any codeword

$$c = (a_0, \ldots, a_{\alpha-1} \mid b_0, \ldots, b_{\beta-1}) \in C \subseteq \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta,$$

its double right cyclic shift $(a_{\alpha-1}, a_0, \ldots, a_{\alpha-2} \mid b_{\beta-1}, b_0, \ldots, b_{\beta-2})$ is also a codeword in C. An element $c = (a_0, \ldots, a_{\alpha-1} \mid b_0, \ldots, b_{\beta-1}) \in \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ can be identified with a module element consisting of two polynomials $c(x) = (a_0 + a_1 x + \cdots + a_{\alpha-1}x^{\alpha-1} \mid b_0 + b_1 x + \cdots + b_{\beta-1}x^{\beta-1}) = (a(x) \mid b(x)) \in R_{\alpha,\beta}$, where $R_{\alpha,\beta} = \mathbb{Z}_2[x]/(x^\alpha - 1) \times \mathbb{Z}_4[x]/(x^\beta - 1)$. This identification gives a one-to-one correspondence between the elements of $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ and $R_{\alpha,\beta}$.

Let $C(x)$ be the set of all polynomials associated to the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C. A subset $C \subseteq \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ is a $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code if and only if the subset $C(x) \subseteq R_{\alpha,\beta}$ is a $\mathbb{Z}_4[x]$-submodule of $R_{\alpha,\beta}$. Moreover, if C is a $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ with β odd, then

$$C(x) = \langle (p(x) \mid 0), (l(x) \mid f(x)h(x) + 2f(x)) \rangle,$$

where $p(x), l(x) \in \mathbb{Z}_2[x]/(x^\alpha - 1)$, $\deg(l(x)) < \deg(p(x))$, $p(x) | (x^\alpha - 1)$, $f(x), h(x) \in \mathbb{Z}_4[x]/(x^\beta - 1)$ with $f(x)h(x) | (x^\beta - 1)$, and $p(x)$ divides $\frac{x^\beta - 1}{f(x)} l(x)$ a $\mathbb{Z}_2[x]$. Note that if β is even, then $x^\beta - 1$ does not factorize uniquely over $\mathbb{Z}_4[x]$.

For more information about $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic codes, the reader is referred to [1, 3, 8], where these codes were introduced and have been studied deeply.

`Z2Z4CyclicCode(`α`, `β`, `p`, `l`, `f`, `h`)`

Given two non-negative integers α and β, and four polynomials $p(x)$, $l(x)$, $f(x)$ and $h(x)$, such that $p(x)$, $l(x) \in \mathbb{Z}_2[x]$ and $f(x)$, $h(x) \in \mathbb{Z}_4[x]$, construct the $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ generated by $(p(x) \mid 0)$ and $(l(x) \mid f(x)h(x) + 2f(x))$.

`Z2Z4CyclicCode(`α`, `β`, `a`, `b`)`

Given two non-negative integers α and β, and two polynomials $a(x) \in \mathbb{Z}_2[x]$ and $b(x) \in \mathbb{Z}_4[x]$, construct the $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ generated by $(a(x) \mid b(x))$.

`Z2Z4CyclicCode(`α`, `β`, G)`

Given two non-negative integers α and β, and a non-empty sequence G containing r tuples of polynomials, that is $G = [< a_1(x), b_1(x) >, \ldots, < a_r(x), b_r(x) >]$, where $a_i(x) \in \mathbb{Z}_2[x]$ and $b_i(x) \in \mathbb{Z}_4[x]$, for $1 \leq i \leq r$, construct the $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ generated by $(a_1(x) \mid b_1(x)), \ldots, (a_r(x) \mid b_r(x))$.

`Z2Z4CyclicCode(u)`

Given a vector $u = (u_\alpha | u_\beta) \in \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, represented as a tuple in the cartesian product set $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, construct the $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ generated by the double right cyclic shifts of the vector u.

`Z2Z4CyclicCode(`α`, u)`

Given a non-negative integer α and a vector $u = (u_\alpha | u_\beta) \in \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, represented as an element in $\mathbb{Z}_4^{\alpha+\beta}$ by replacing the ones in the first α coordinates by twos, construct the $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ generated by the double right cyclic shifts of the vector u. It is checked whether the elements in the first α coordinates are in $\{0, 2\}$.

`Z2Z4CyclicCode(G)`

Given a non-empty sequence of r vectors $G = [u_1, u_2, \ldots, u_r]$, where, for all $1 \leq i \leq r$, $u_i \in \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ is represented as a tuple in the cartesian product set $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, construct the $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ generated by the double right cyclic shifts of the vectors $u_1, u_2, \ldots, u_r$.

`Z2Z4CyclicCode(`α`, G)`

Given a non-negative integer α and a non-empty sequence of r vectors $G = [u_1, u_2, \ldots, u_r]$, where, for $1 \leq i \leq r$, $u_i \in \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ is represented as an element in $\mathbb{Z}_4^{\alpha+\beta}$ by replacing the ones in the first α coordinates by twos, construct the $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ generated by the double right cyclic shifts of the vectors $u_1, u_2, \ldots, u_r$. It is checked whether the elements in the first α coordinates are in $\{0, 2\}$.

`RandomZ2Z4CyclicCode(`α`, `β`)`

Given two non-negative integers α and β, with β odd, return a random $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ and a tuple containing the generator polynomials $< p(x), l(x), f(x), h(x) >$, where $p(x)$, $l(x) \in \mathbb{Z}_2[x]$ and $f(x)$, $h(x) \in \mathbb{Z}_4[x]$, satisfying the conditions described in Subsection 2.2.3.

Example H2E4

```
> PR2<x> := PolynomialRing(Integers(2));
> PR4<y> := PolynomialRing(Integers(4));
> alpha := 15;
> beta := 7;

> p := x^5+x^3+x+1;
```

```
> l := x^4+x^3+1;
> f := PR4!1;
> h := y^4+y^3+3*y^2+2*y+1;
> C1 := Z2Z4CyclicCode(alpha, beta, p, l, f, h);
> C2 := Z2Z4CyclicCode(alpha, beta, [<p, PR4!0>, <l, f*h + 2*f>]);
> C1 eq C2;
true
> IsCyclic(C1);
true

> R := RSpace(Integers(4), alpha + beta);
> g1 := R!([2*(Coefficient(PR4!p, i)) : i in [0 .. alpha - 1]]
            cat [Integers(4)!0 : j in [0 .. beta - 1]]);
> g1;
(2 2 0 2 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> g2 := R!([2*(Coefficient(PR4!l, i)) : i in [0 .. alpha - 1]]
            cat [Coefficient(f*h + 2*f, j) : j in [0 .. beta - 1]]);
> g2;
(2 0 0 2 2 0 0 0 0 0 0 0 0 0 3 2 3 1 1 0 0)
> C3 := Z2Z4CyclicCode(alpha, [g1, g2]);
> C1 eq C3;
true

> R2 := RSpace(Integers(2), alpha);
> R4 := RSpace(Integers(4), beta);
> R2R4 := CartesianProduct(R2, R4);
> g3 := R2R4!<[Coefficient(PR4!p, i) : i in [0 .. alpha - 1]],
              [Integers(4)!0 : j in [0 .. beta - 1]]>;
> g3;
<(1 1 0 1 0 1 0 0 0 0 0 0 0 0), (0 0 0 0 0 0 0)>
> g4 := R2R4!<[Coefficient(PR4!l, i) : i in [0 .. alpha - 1]],
              [Coefficient(f*h + 2*f, j) : j in [0 .. beta - 1]]>;
> g4;
<(1 0 0 1 1 0 0 0 0 0 0 0 0 0), (3 2 3 1 1 0 0)>
> C4 := Z2Z4CyclicCode([g3, g4]);
> C1 eq C4;
true

> C5 := Z2Z4CyclicCode(alpha, g1);
> C6 := Z2Z4CyclicCode(alpha, g2);
> (C5 subset C4) and (C6 subset C4);
true
> C5 eq Z2Z4CyclicCode(g3);
true
> C6 eq Z2Z4CyclicCode(g4);
true

> G := MinRowsGeneratorMatrix(C1);
> v := Eltseq(G[1]);
```

```
> v;
[ 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 2 ]
> u := Rotate(v[1..alpha], 3) cat Rotate(v[alpha+1..alpha+beta], 3);
> u;
[ 2, 2, 2, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0, 0, 0 ]
> R!u in C1;
true

> v := FromZ4toZ2Z4(G[1], alpha);
> v;
<(0 0 1 0 0 0 0 0 0 0 1 1 1 1), (0 0 0 0 0 0 2)>
> u := Rotate(v, 3);
> u;
<(1 1 1 0 0 1 0 0 0 0 0 0 0 1), (0 0 2 0 0 0 0)>
> u in C1;
true

> C7 := RandomZ2Z4CyclicCode(alpha, beta);
> IsZ2Z4AdditiveCode(C7) and IsCyclic(C7);
true
> (Z2Z4Type(C7)[1] eq alpha) and (Z2Z4Type(C7)[2] eq beta);
true
```

2.2.4 Families of $\mathbb{Z}_2\mathbb{Z}_4$-Additive Codes

`Z2Z4HadamardCode(`δ`, m :` *parameters*`)`

`OverZ4` BoolElt *Default*: `false`

The function returns a $\mathbb{Z}_2\mathbb{Z}_4$-additive Hadamard code of type $(\alpha, \beta; \gamma, \delta; \kappa)$. The parameter `OverZ4` specifies whether the code is over $\mathbb{Z}_4$, that is $\alpha = 0$, or, otherwise, $\alpha \neq 0$. The default value is `false`. When `OverZ4` is `true`, given an integer $m \geq 1$ and an integer δ such that $1 \leq \delta \leq \lfloor (m+1)/2 \rfloor$, return a $\mathbb{Z}_2\mathbb{Z}_4$-additive Hadamard code of type $(0, \beta; \gamma, \delta; 0)$, where $\beta = 2^{m-1}$ and $\gamma = m+1-2\delta$. When `OverZ4` is `false`, given an integer $m \geq 1$ and an integer δ such that $0 \leq \delta \leq \lfloor m/2 \rfloor$, return a $\mathbb{Z}_2\mathbb{Z}_4$-additive Hadamard code of type $(\alpha, \beta; \gamma, \delta; \gamma)$, where $\alpha = 2^{m-\delta}$, $\beta = 2^{m-1} - 2^{m-\delta-1}$ and $\gamma = m + 1 - 2\delta$. Moreover, return a generator matrix with $\gamma + \delta$ rows constructed in a recursive way, where the ones in the first α coordinates are represented by twos.

A $\mathbb{Z}_2\mathbb{Z}_4$-additive Hadamard code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ is a $\mathbb{Z}_2\mathbb{Z}_4$-additive code such that, after the Gray map, give a binary (not necessarily linear) code with the same parameters as the binary Hadamard code of length 2^m.

Z2Z4ExtendedPerfectCode(δ, m : *parameters*)

OverZ4 BoolElt *Default*: false

The function returns a $\mathbb{Z}_2\mathbb{Z}_4$-additive extended perfect code of type $(\alpha, \beta; \gamma, \delta; \kappa)$. The parameter OverZ4 specifies whether the code is over $\mathbb{Z}_4$, that is $\alpha = 0$, or, otherwise, $\alpha \neq 0$. The default value is false. When OverZ4 is true, given an integer $m \geq 1$ and an integer δ such that $1 \leq \delta \leq \lfloor (m+1)/2 \rfloor$, return a $\mathbb{Z}_2\mathbb{Z}_4$-additive extended perfect code, such that its additive dual code is of type $(0, \beta; \gamma, \delta; 0)$, where $\beta = 2^{m-1}$ and $\gamma = m + 1 - 2\delta$. When OverZ4 is false, given an integer $m \geq 1$ and an integer δ such that $0 \leq \delta \leq \lfloor m/2 \rfloor$, return a $\mathbb{Z}_2\mathbb{Z}_4$-additive extended perfect code, such that its additive dual code is of type $(\alpha, \beta; \gamma, \delta; \gamma)$, where $\alpha = 2^{m-\delta}$, $\beta = 2^{m-1} - 2^{m-\delta-1}$ and $\gamma = m + 1 - 2\delta$. Moreover, return a parity check matrix with $\gamma + \delta$ rows constructed in a recursive way, where the ones in the first α coordinates are represented by twos.

A $\mathbb{Z}_2\mathbb{Z}_4$-additive extended perfect code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ is a $\mathbb{Z}_2\mathbb{Z}_4$-additive code such that, after the Gray map, give a binary (not necessarily linear) code with the same parameters as the binary extended perfect code of length 2^m.

Example H2E5

Some $\mathbb{Z}_2\mathbb{Z}_4$-additive codes whose images under the Gray map are binary codes having the same parameters as some well-known families of binary linear codes are explored.

First, a $\mathbb{Z}_2\mathbb{Z}_4$-additive Hadamard code C with $\alpha = 0$ and another one with $\alpha \neq 0$ are defined. The matrix Gc is the quaternary matrix used to generate C where the ones in the first α coordinates are represented by twos.

```
> C, Gc := Z2Z4HadamardCode(3, 5 : OverZ4 := true);
> C`Alpha;
0
> (C`Code eq HadamardCodeZ4(3, 5)) and (C`Code eq LinearCode(Gc));
true
> HasLinearGrayMapImage(C);
false
> n := BinaryLength(C);
32
> (Z2Z4MinimumLeeDistance(C) eq n/2) and (#C eq 2*n);
true

> C, Gc := Z2Z4HadamardCode(2, 5);
> C`Alpha;
8
> C eq Z2Z4AdditiveCode(LinearCode(Gc), C`Alpha);
true
```

```
> HasLinearGrayMapImage(C);
false
> n := BinaryLength(C);
32
> (Z2Z4MinimumLeeDistance(C) eq n/2) and (#C eq 2*n);
true
```

Then, a $\mathbb{Z}_2\mathbb{Z}_4$-additive extended perfect code D with $\alpha \neq 0$ is defined such that its additive dual code is of type $(2^{5-2}, 2^{5-1} - 2^{5-2-1}; 5 + 1 - 2 \cdot 2, 3; 2^{5-2}) = (8, 12; 2, 2; 2)$. The matrix Gd is the quaternary matrix which is used to generate the additive dual code of D where the ones in the first α coordinates are represented by twos.

```
> C, Gc := Z2Z4HadamardCode(2, 5);
> D, Gd := Z2Z4ExtendedPerfectCode(2, 5);
> D eq Dual(C);
true
> Gc eq Gd;
true
> n := BinaryLength(D);
> #D eq 2^(n-1-Log(2,n));
true
> Z2Z4MinimumLeeDistance(D) eq 4;
true
```

`Z2Z4ReedMullerCode(s, r, m)`

`OverZ4` BoolElt *Default*: `false`

The function returns a $\mathbb{Z}_2\mathbb{Z}_4$-additive Reed-Muller code of type $(\alpha, \beta; \gamma, \delta; \kappa)$. The parameter `OverZ4` specifies whether the code is over $\mathbb{Z}_4$, that is $\alpha = 0$, or, otherwise, $\alpha \neq 0$. The default value is `false`. When `OverZ4` is `true`, given an integer $m \geq 1$, an integer s such that $0 \leq s \leq \lfloor (m-1)/2 \rfloor$, and an integer r such that $0 \leq r \leq m$, return an r-th order $\mathbb{Z}_2\mathbb{Z}_4$-additive Reed-Muller code $RM_s(r, m)$, with $\alpha = 0$ and $\beta = 2^{m-1}$. In this case, the code coincides with the one obtained by using `ReedMullerCodeRMZ4(s,r,m)`. When `OverZ4` is `false`, given $m \geq 1$, $0 \leq s \leq \lfloor m/2 \rfloor$, and $0 \leq r \leq m$, return an r-th order $\mathbb{Z}_2\mathbb{Z}_4$-additive Reed-Muller code $ARM_s(r, m)$, with $\alpha = 2^{m-s}$ and $\beta = 2^{m-1} - 2^{m-s-1}$. Moreover, in both cases, it returns a generator matrix with $\gamma + \delta$ rows constructed in a recursive way, where the ones in the first α coordinates are represented by twos.

An r-th order $\mathbb{Z}_2\mathbb{Z}_4$-additive Reed-Muller code is a $\mathbb{Z}_2\mathbb{Z}_4$-additive code such that, after the Gray map, gives a binary (not necessarily linear) code with the same parameters as the binary linear r-th order Reed-Muller code of length 2^m. For $RM_s(r,m)$ codes, the inclusion and duality properties are also satisfied, that is, $RM_s(r-1,m)$ is a subcode of $RM_s(r,m)$, $r > 0$, and $RM_s(r,m)$ is the Kronecker dual code of $RM_s(m-r-1,m)$, $r < m$. For $ARM_s(r,m)$ codes, only the inclusion property is satisfied, that is, $ARM_s(r-1,m)$ is a subcode of $ARM_s(r,m)$, $r > 0$.

Example H2E6

Taking the Reed-Muller codes $RM_1(1,4)$ and $RM_1(2,4)$, it can be seen that the former is a subcode of the latter, and that it is the Kronecker dual of the latter. Note that $RM_1(1,4)$ and $RM_1(2,4)$ are the same as the ones obtained by using the functions `HadamardCodeZ4(2, 4)` and `ExtendedPerfectCodeZ4(2, 4)`, respectively.

```
> C1 := Z2Z4ReedMullerCode(1, 1, 4 : OverZ4 := true);
> C2 := Z2Z4ReedMullerCode(1, 2, 4 : OverZ4 := true);
> C1;
(8, 32, 8) Z2Z4-additive code of type (0, 8; 1, 2; 0)
Generator matrix:
[1 0 3 2 1 0 3 2]
[0 1 2 3 0 1 2 3]
[0 0 0 0 2 2 2 2]
> C1 subset C2;
true
> DualKroneckerZ4(C2`Code) eq C1`Code;
true
> C1`Code eq HadamardCodeZ4(2, 4);
true
> C2`Code eq ExtendedPerfectCodeZ4(2, 4);
true
```

Similarly, taking the Reed-Muller codes $ARM_1(1,4)$ and $ARM_1(2,4)$, it can also be seen that the former is a subcode of the latter. In this case, $ARM_1(1,4)$ coincides with the one obtained by using the function `Z2Z4HadamardCode(1, 4)`, but $ARM_1(2,4)$ does not coincide with `Z2Z4ExtendedPerfectCode(1, 4)`, although they have the same parameters and are equivalent after the Gray map.

```
> C1 := Z2Z4ReedMullerCode(1, 1, 4);
> C2 := Z2Z4ReedMullerCode(1, 2, 4);
> C1;
(12, 32, 8) Z2Z4-additive code of type (8, 4; 3, 1; 3)
Generator matrix:
[2 0 0 2 0 2 2 0 1 3 3 1]
[0 2 0 2 0 2 0 2 1 1 1 1]
[0 0 2 2 0 0 2 2 0 2 0 2]
[0 0 0 0 2 2 2 2 0 0 2 2]
[0 0 0 0 0 0 0 0 2 2 2 2]
```

```
> C1 subset C2;
true
> C1 eq Z2Z4HadamardCode(1, 4);
true
>
> C2b := Z2Z4ExtendedPerfectCode(1, 4);
> C2 eq C2b;
false
> Z2Z4Type(C2) eq Z2Z4Type(C2b);
true
> LeeWeightDistribution(C2) eq LeeWeightDistribution(C2b);
true
```

`Z2Z4ReedMullerCodes(s, m)`

`OverZ4` BoolElt *Default*: `false`

The function returns a sequence containing a family of $\mathbb{Z}_2\mathbb{Z}_4$-additive Reed-Muller codes. The parameter `OverZ4` specifies whether the codes are over $\mathbb{Z}_4$, that is $\alpha = 0$, or, otherwise, $\alpha \neq 0$. The default value is `false`. When `OverZ4` is `true`, return the codes $RM_s(r, m)$, given any $m \geq 1$ and $0 \leq s \leq \lfloor (m-1)/2 \rfloor$. When `OverZ4` is `false`, return the codes $ARM_s(r, m)$, given any $m \geq 1$ and $0 \leq s \leq \lfloor m/2 \rfloor$.

The binary image of these codes under the Gray map gives a family of binary (not necessarily linear) codes with the same parameters as the binary linear Reed-Muller family of codes of length 2^m. Note that

$$RM_s(0, m) \subset RM_s(1, m) \subset \cdots \subset RM_s(m, m) \quad \text{and}$$

$$ARM_s(0, m) \subset ARM_s(1, m) \subset \cdots \subset ARM_s(m, m).$$

Example H2E7

The family of Reed-Muller codes over $\mathbb{Z}_4$ given by $m = 3$ and $s = 0$, and the family of $\mathbb{Z}_2\mathbb{Z}_4$-additive Reed-Muller codes with $\alpha \neq 0$ given by $m = 3$ and $s = 0$ are constructed.

```
> F := Z2Z4ReedMullerCodes(0, 3 : OverZ4 := true);
> F eq [Z2Z4ReedMullerCode(0, r, 3 : OverZ4 := true) :  r in [0..3]];
true
> (F[1] subset F[2]) and (F[2] subset F[3]) and (F[3] subset F[4]);
true
>
> AF := Z2Z4ReedMullerCodes(0, 3);
> AF eq [Z2Z4ReedMullerCode(0, r, 3) :  r in [0..3]];
true
> (AF[1] subset AF[2]) and (AF[2] subset AF[3]) and (AF[3] subset AF[4]);
true
```

`KlemmCodeZ4(m)`

Given an integer $m \geq 1$, return the Klemm code over $\mathbb{Z}_4$ of length $4m$ and type $2^{4m-2}4$ as a $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(0, 4m; 4m-2, 1; 0)$. Moreover, return a generator matrix G with $4m-1$ rows.

The Klemm code over $\mathbb{Z}_4$ of length $4m$ is the code generated by the following matrix:

$$G = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 & 1 \\ 0 & 2 & 0 & \dots & 0 & 2 \\ 0 & 0 & 2 & \dots & 0 & 2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 2 & 2 \end{pmatrix}.$$

2.3 Invariants of a $\mathbb{Z}_2\mathbb{Z}_4$-Additive Code

2.3.1 Basic Numerical Invariants

`Length(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the length $n = \alpha + \beta$, and the sequence $[\alpha, \beta]$ with the number of coordinates over $\mathbb{Z}_2$ and the number of coordinates over $\mathbb{Z}_4$, respectively.

`BinaryLength(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the binary length $n_{bin} = \alpha + 2\beta$ of C, which corresponds to the length of the binary code $C_{bin} = \Phi(C)$, where Φ is the Gray map defined in Subsection 2.5.1.

`PseudoDimension(C)`
`NumberOfGenerators(C)`
`Ngens(C)`

The number of generators (which equals the pseudo-dimension k) of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C as a quaternary linear code.

`Z2Z4Type(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return a sequence with the parameters $[\alpha, \beta, \gamma, \delta, \kappa]$, that is, the type of the code.

`#C`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the number of codewords belonging to C, that is $2^\gamma 4^\delta$.

`InformationRate(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the information rate of C, that is the ratio $(\gamma + 2\delta)/(\alpha + 2\beta)$.

Example H2E8

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, we compute its type, the number of codewords and its information rate.

```
> R := RSpace(IntegerRing(4), 6);
> C := Z2Z4AdditiveCode([R![2,2,1,1,3,1], R![2,2,2,2,2,2],
                         R![0,0,1,1,1,3]], 2);
> Z2Z4Type(C);
[ 2, 4, 2, 1, 1 ]
> #C;
16
> InformationRate(C);
0.40000000000000000000000000000
```

2.3.2 The Code Space

`AmbientSpace(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the ambient space of C, i.e., the $\mathbb{Z}_4$-submodule of $\mathbb{Z}_4^{\alpha+\beta}$ isomorphic to $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, where C is contained after replacing the ones in the first α coordinates by twos. It also returns the generic space $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ as a cartesian product.

`Generic(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the generic $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \alpha, \beta; \alpha)$ in which C is contained.

`Name(C, i)`

`C.i`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ and a positive integer i, return the i-th generator of C as a quaternary linear code, where the ones in the first α coordinates are represented by twos.

`Set(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the set containing all its codewords, where the ones in the first α coordinates are represented by twos.

`Z2Z4Set(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the set containing all its codewords as tuples in the cartesian product set $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$.

`Basis(C)`

The basis for the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ as a quaternary linear code, returned as a sequence of elements of C, where the ones in the first α coordinates are represented by twos.

`Generators(C)`

The generators for the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ as a quaternary linear code, returned as a set of elements of C, where the ones in the first α coordinates are represented by twos.

`GeneratorMatrix(C)`

The unique generator matrix for the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ as a quaternary linear code, corresponding to the Howell form (see [14, 25]). The ones in the first α coordinates are represented by twos.

`OrderTwoGenerators(C)`

The γ generators of order two of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, returned as a set of elements of C, where the ones in the first α coordinates are represented by twos.

`OrderFourGenerators(C)`

The δ generators of order four of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, returned as a set of elements of C, where the ones in the first α coordinates are represented by twos.

`OrderTwoSubcodeGenerators(C)`

The $\gamma + \delta$ generators of the $\mathbb{Z}_2\mathbb{Z}_4$-additive subcode C_b which contains all order two codewords of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, returned as a set of elements of C_b, where the ones in the first α coordinates are represented by twos.

`MinRowsGeneratorMatrix(C)`

A generator matrix of the form (2.1) for the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, with the minimum number of rows, that is with $\gamma + \delta$ rows: γ rows of order two and δ rows of order four. It also returns the parameters γ and δ. The ones in the first α coordinates are represented by twos.

Example H2E9

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, we compute its ambient spaces, the set of codewords and a generator matrix with the minimum number of rows $\gamma + \delta$.

```
> R := RSpace(IntegerRing(4), 6);
> C := Z2Z4AdditiveCode([R![2,2,1,1,3,1], R![2,2,2,2,2,2]], 2);
> V4, R2R4 := AmbientSpace(C);
> R2R4;
Cartesian Product<Full Vector space of degree 2 over IntegerRing(2), Full RSpace
of degree 4 over IntegerRing(4)>
> V4;
RSpace of degree 6, dimension 6 over IntegerRing(4)
Echelonized basis:
(2 0 0 0 0 0)
(0 2 0 0 0 0)
(0 0 1 0 0 0)
(0 0 0 1 0 0)
(0 0 0 0 1 0)
(0 0 0 0 0 1)
> V4 eq R;
false
> V4 subset R;
true

> Set(C);
{
    (2 2 0 0 0 0),
    (0 0 3 3 1 3),
    (0 0 2 2 2 2),
    (2 2 3 3 1 3),
    (0 0 1 1 3 1),
    (2 2 2 2 2 2),
    (2 2 1 1 3 1),
    (0 0 0 0 0 0)
}
> Z2Z4Set(C);
{
    <(1 1), (3 3 1 3)>,
    <(0 0), (1 1 3 1)>,
    <(1 1), (1 1 3 1)>,
    <(1 1), (0 0 0 0)>,
    <(1 1), (2 2 2 2)>,
    <(0 0), (0 0 0 0)>,
    <(0 0), (3 3 1 3)>,
    <(0 0), (2 2 2 2)>
}
> FromZ4toZ2Z4(Set(C), 2) eq Z2Z4Set(C);
true
> (Random(Set(C)) in V4) and (Random(Set(C)) in R);
true
> Random(Z2Z4Set(C)) in R2R4;
true
```

```
> G, gamma, delta := MinRowsGeneratorMatrix(C);
> G;
[2 2 0 0 0 0]
[0 0 1 1 3 1]
> Z2Z4Type(C)[3] eq gamma;
true
> Z2Z4Type(C)[4] eq delta;
true
```

In general, the unique generator matrix in Howell form does not coincide with the generator matrix with the minimum number of rows.

```
> R := RSpace(IntegerRing(4), 5);
> C := Z2Z4AdditiveCode([R![2,0,0,0,1], R![0,2,0,0,1],
                         R![0,0,0,2,0], R![0,0,0,0,2]], 2);
> GeneratorMatrix(C);
[2 0 0 0 1]
[0 2 0 0 1]
[0 0 0 2 0]
[0 0 0 0 2]
> MinRowsGeneratorMatrix(C);
[2 2 0 0 0]
[0 0 0 2 0]
[0 2 0 0 1]
2 1
```

2.3.3 Conversion Functions

The $\mathbb{Z}_2\mathbb{Z}_4$-additive codes are internally represented as quaternary linear codes, with their first α coordinates represented as elements in $\{0, 2\}$ over $\mathbb{Z}_4$ instead of elements in $\{0, 1\}$ over $\mathbb{Z}_2$. This section provides functions to convert the outputs given as a vector over $\mathbb{Z}_4^{\alpha+\beta}$, or as a sequence, set or matrix of vectors over $\mathbb{Z}_4^{\alpha+\beta}$, to a tuple, sequence or set of tuples in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$, replacing the twos over $\mathbb{Z}_4$ in the first α coordinates to ones over $\mathbb{Z}_2$.

`FromZ4toZ2Z4(u, α)`

Given a vector over $\mathbb{Z}_4^{\alpha+\beta}$ and an integer α, return the conversion of this vector to a tuple in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$, replacing the twos over $\mathbb{Z}_4$ in the first α coordinates to ones over $\mathbb{Z}_2$. It is checked whether the elements in the first α coordinates are in $\{0, 2\}$.

`FromZ4toZ2Z4(L, α)`

Given a sequence L of vectors over $\mathbb{Z}_4^{\alpha+\beta}$ and an integer α, return the conversion of these vectors to a sequence of tuples in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$, replacing the twos over $\mathbb{Z}_4$ in the first α coordinates to ones over $\mathbb{Z}_2$. It is checked whether the elements in the first α coordinates are in $\{0, 2\}$.

`FromZ4toZ2Z4(S, α)`

Given a set S of vectors over $\mathbb{Z}_4^{\alpha+\beta}$ and an integer α, return the conversion of these vectors to a set of tuples in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$, replacing the twos over $\mathbb{Z}_4$ in the first α coordinates to ones over $\mathbb{Z}_2$. It is checked whether the elements in the first α coordinates are in $\{0, 2\}$.

`FromZ4toZ2Z4(M, α)`

Given a matrix M over $\mathbb{Z}_4$ with $\alpha + \beta$ columns and an integer α, return the conversion from M to a sequence of tuples in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$, replacing the twos over $\mathbb{Z}_4$ in the first α coordinates to ones over $\mathbb{Z}_2$. It is checked whether the elements in the first α coordinates are in $\{0, 2\}$.

2.3.4 The Dual Space

The inner product in $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$ is defined uniquely by

$$\langle u, v \rangle = 2(\sum_{i=1}^{\alpha} u_i v_i) + \sum_{j=\alpha+1}^{\alpha+\beta} u_j v_j \in \mathbb{Z}_4, \tag{2.2}$$

where $u, v \in \mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$. Note that when $\alpha = 0$ the inner product is the usual one for vectors over $\mathbb{Z}_4$, and when $\beta = 0$ it is twice the usual one for vectors over $\mathbb{Z}_2$.

The *additive orthogonal code* of a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, denoted by $C^{\perp}$, is defined in the standard way

$$C^{\perp} = \{v \in \mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta} \mid \langle u, v \rangle = 0 \text{ for all } u \in C\}.$$

We will also call $C^{\perp}$ the *additive dual code* of C. Note that the additive dual code $C^{\perp}$ is also a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, that is a subgroup of $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

`Dual(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return the additive dual code of C.

`Z2Z4DualType(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return a sequence with the parameters $[\alpha, \beta, \gamma', \delta', \kappa']$, that is the type of the additive dual code of C. If C is of type $(\alpha, \beta; \gamma, \delta; \kappa)$, then its additive dual code is of type $(\alpha, \beta; \alpha + \gamma - 2\kappa, \beta - \gamma - \delta + \kappa; \alpha - \kappa)$.

`ParityCheckMatrix(C)`

The unique parity-check matrix for the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, that is the unique generator matrix for the additive dual code of C as a quaternary linear code, corresponding to the Howell form (see [14, 25]). The ones in the first α coordinates are represented by twos.

`MinRowsParityCheckMatrix(C)`

A parity-check matrix of the form (2.1) for the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, that is a generator matrix for the additive dual code of C, with the minimum number of rows, that is with $\gamma + \delta$ rows: γ rows of order two and δ rows of order four. The ones in the first α coordinates are represented by twos.

`Hull(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return the Hull of C, which is the intersection between itself and its additive dual.

Example H2E10

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, we check some of the parameters related to its additive dual code $C^\perp$.

```
> C := Z2Z4HadamardCode(2, 5);
> D := Dual(C);
> typeC := Z2Z4Type(C);
> typeD := Z2Z4Type(D);
> typeD eq Z2Z4DualType(C);
true
> typeD[1] eq typeC[1];
true
> typeD[2] eq typeC[2];
true
> typeD[3] eq typeC[1]+typeC[3]-2*typeC[5];
true
> typeD[4] eq typeC[2]-typeC[3]-typeC[4]+typeC[5];
true
> typeD[5] eq typeC[1]-typeC[5];
true

> G, gamma, delta := MinRowsParityCheckMatrix(C);
```

```
> Nrows(G) eq gamma+delta;
true
> gamma eq typeD[3];
true
> delta eq typeD[4];
true

> Hull(C) eq (C meet D);
true
```

2.3.5 The Generator Polynomial

`GeneratorPolynomials(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ with β odd, return a tuple containing the generator polynomials $< p(x), l(x), f(x), h(x) >$, where $p(x)$, $l(x) \in \mathbb{Z}_2[x]$ and $f(x)$, $h(x) \in \mathbb{Z}_4[x]$, satisfying the conditions described in Subsection 2.2.3 (see [1, 8]).

`DualGeneratorPolynomials(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ with β odd, return a tuple of polynomials $< p'(x), l'(x), f'(x), h'(x) >$, where $p'(x)$, $l'(x) \in \mathbb{Z}_2[x]$ and $f'(x)$, $h'(x) \in \mathbb{Z}_4[x]$. These polynomials are the generator polynomials of the additive dual code of C, and satisfy the conditions described in Subsection 2.2.3 (see [3]).

`DualGeneratorPolynomials(`α`, `β`, `p`, `l`, `f`, `h`)`

Given two non-negative integers α and β, with β odd, and four polynomials, $p(x)$, $l(x)$, $f(x)$ and $h(x)$, such that $p(x)$, $l(x) \in \mathbb{Z}_2[x]$ and $f(x)$, $h(x) \in \mathbb{Z}_4[x]$, return a tuple of polynomials $< p'(x), l'(x), f'(x), h'(x) >$, where $p'(x)$, $l'(x) \in \mathbb{Z}_2[x]$ and $f'(x)$, $h'(x) \in \mathbb{Z}_4[x]$. These polynomials are the generator polynomials of the additive dual code of the $\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ generated by the polynomials $p(x)$, $l(x)$, $f(x)$ and $h(x)$. They satisfy the conditions described in Subsection 2.2.3 (see [3]).

Example H2E11

```
> PR2<x> := PolynomialRing(Integers(2));
> PR4<y> := PolynomialRing(Integers(4));
> alpha := 15;
> beta := 7;

> U := Z2Z4AdditiveUniverseCode(alpha, beta);
```

```
> Z := Z2Z4AdditiveZeroCode(alpha, beta);
> IsCyclic(U);
true
> IsCyclic(Z);
true
> GeneratorPolynomials(U);
<1, 0, 1, 1>
> GeneratorPolynomials(Z);
<x^15 + 1, 0, y^7 + 3, 1>

> a1 := x^6+x^4+x^2+x;
> a2 := x^5+x^4+x;
> b1 := PR4!0;
> b2 := y^5+y^4+3*y^3+2*y^2+3*y;
> C1 := Z2Z4CyclicCode(alpha, beta, [<a1, b1>, <a2, b2>]);
> GeneratorPolynomials(C1);
<x^5 + x^3 + x + 1, x^4 + x^3 + 1, 1, y^4 + y^3 + 3*y^2 + 2*y + 1>
> p := x^5+x^3+x+1;
> l := x^4+x^3+1;
> f := PR4!1;
> h := y^4+y^3+3*y^2+2*y+1;
> C2 := Z2Z4CyclicCode(alpha, beta, [<p, PR4!0>, <l, f*h + 2*f>]);
> C1 eq C2;
true

> DualGeneratorPolynomials(C2) eq DualGeneratorPolynomials(alpha, beta, p, l, f, h);
true
> DualGeneratorPolynomials(C2) eq GeneratorPolynomials(Dual(C2));
true
```

2.3.6 Information Space and Information Sets

`InformationSpace(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the $\mathbb{Z}_4$-submodule of $\mathbb{Z}_4^{\gamma+\delta}$ isomorphic to $\mathbb{Z}_2^{\gamma} \times \mathbb{Z}_4^{\delta}$ such that the first γ coordinates are of order two, that is, the space of information vectors for C. The function also returns the $(\gamma + 2\delta)$-dimensional binary vector space, which is the space of information vectors for the corresponding binary code $C_{bin} = \Phi(C)$, where Φ is the Gray map. Finally, for the encoding process, it also returns the corresponding isomorphisms f and f_{bin} from these spaces of information vectors onto C and C_{bin}, respectively.

Example H2E12

```
> C := Z2Z4HadamardCode(2, 4);
> C;
(10, 32, 8) Z2Z4-additive code of type (4, 6; 1, 2; 1)
Generator matrix:
[2 0 0 2 1 3 1 0 3 2]
[0 2 0 2 0 2 1 1 1 1]
[0 0 2 2 1 1 0 1 2 3]
[0 0 0 0 2 2 0 2 0 2]
[0 0 0 0 0 0 2 2 2 2]

> V4, V2, f, fbin := InformationSpace(C);
> G := MinRowsGeneratorMatrix(C);

> (#V4 eq #C`Code) and (#V2 eq #C`Code);
true
> Set([f(i) : i in V4]) eq Set(C);
true
> Set([Z2Z4Mult(i, G, Z2Z4Type(C)[3]): i in V4]) eq Set(C);
true
> Set([i*G : i in V4]) eq Set(C);
false

> i := V4![2,3,1];
> c := f(i);
> c;
(2 0 0 2 3 1 1 2 3 0)
> c in C;
true
> c eq Z2Z4Mult(i, G, Z2Z4Type(C)[3]);
true
> c eq i*G;
false

> ibin := V2![1,1,0,0,1];
> ibin eq GrayMap(Z2Z4AdditiveUniverseCode(1, 2))(i);
true
> cbin := fbin(ibin);
> cbin;
(1 0 0 1 1 0 0 1 0 1 1 1 1 0 0 0)
> cbin eq GrayMap(C)(c);
true
```

`InformationSet(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return an information set $I = [i_1, \ldots, i_{\gamma+\delta}] \subseteq \{1, \ldots, \alpha + \beta\}$ for C such that $\{i_1, \ldots, i_\kappa\} \subseteq \{1, \ldots, \alpha\}$ and the code C punctured on $\{1, \ldots, \alpha + \beta\} \backslash \{i_{\gamma+1}, \ldots, i_{\gamma+\delta}\}$ is of type 4^δ, and the corresponding information set $\Phi(I) = [i_1, \ldots, i_\kappa, 2i_{\kappa+1} - 1 - \alpha, \ldots, 2i_\gamma - 1 - \alpha, 2i_{\gamma+1} - 1 - \alpha, 2i_{\gamma+1} - \alpha, \ldots, 2i_{\gamma+\delta} - 1 - \alpha, 2i_{\gamma+\delta} - \alpha] \subseteq \{1, \ldots, \alpha + 2\beta\}$ for the binary code $C_{bin} = \Phi(C)$, where Φ is the Gray map. The information sets I and $\Phi(I)$ are returned as a sequence of $\gamma + \delta$ and $\gamma + 2\delta$ integers, giving the coordinate positions that correspond to the information set of C and C_{bin}, respectively.

An information set I for C is an ordered set of $\gamma + \delta$ coordinate positions such that $|C^I| = 2^\gamma 4^\delta$, where $C^I = \{v^I : v \in C\}$ and v^I is the vector v restricted to the I coordinates. An information set J for C_{bin} is an ordered set of $\gamma + 2\delta$ coordinate positions such that $|C^J_{bin}| = 2^{\gamma+2\delta}$.

`IsInformationSet(C, I)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ and a sequence $I \subseteq \{1, \ldots, \alpha + \beta\}$ or $I \subseteq \{1, \ldots, \alpha + 2\beta\}$, return `true` if and only if $I \subseteq \{1, \ldots, \alpha + \beta\}$ is an information set for C. This function also returns another boolean, which is `true` if an only if $I \subseteq \{1, \ldots, \alpha + 2\beta\}$ is an information set for the corresponding binary code $C_{bin} = \Phi(C)$, where Φ is the Gray map.

An information set I for C is an ordered set of $\gamma + \delta$ coordinate positions such that $|C^I| = 2^\gamma 4^\delta$, where $C^I = \{v^I : v \in C\}$ and v^I is the vector v restricted to the I coordinates. An information set J for C_{bin} is an ordered set of $\gamma + 2\delta$ coordinate positions such that $|C^J_{bin}| = 2^{\gamma+2\delta}$.

Example H2E13

```
> C := Z2Z4HadamardCode(2, 5);
> C;
(20, 64, 16) Z2Z4-additive code of type (8, 12; 2, 2; 2)
Generator matrix:
[2 0 0 2 0 2 2 0 1 3 1 0 3 2 3 1 3 2 1 0]
[0 2 0 2 0 2 0 2 0 2 1 1 1 1 0 2 1 1 1 1]
[0 0 2 2 0 0 2 2 1 1 0 1 2 3 1 1 0 1 2 3]
[0 0 0 0 2 2 2 2 0 0 0 0 0 0 2 2 2 2 2 2]
[0 0 0 0 0 0 0 0 2 2 0 2 0 2 2 2 0 2 0 2]
[0 0 0 0 0 0 0 0 0 0 2 2 2 2 0 0 2 2 2 2]

> n := Length(C);
> gamma := Z2Z4Type(C)[3];
```

```
> delta := Z2Z4Type(C)[4];

> I, Ibin := InformationSet(C);
> I;
[ 1, 5, 19, 20 ]
> Ibin;
[ 1, 5, 29, 30, 31, 32 ]
> #PunctureCode(C, {1..n} diff Set(I))`Code eq #C;
true
> Cbin := GrayMapImage(C);
> V2 := VectorSpace(GF(2), gamma + 2*delta);
> #{V2![c[i] : i in Ibin] : c in Cbin} eq #Cbin;
true

> IsInformationSet(C, I);
true false
> IsInformationSet(C, Ibin);
false true

> IsInformationSet(C, [1, 5, 9, 11]);
true false
> IsInformationSet(C, [1, 5, 9, 10, 13, 14]);
false true

> R := RSpace(IntegerRing(4), 5);
> D := Z2Z4AdditiveCode([R![2,0,0,2,0], R![0,2,0,2,2], R![0,0,2,2,0]], 5);
> IsInformationSet(D, [1, 3, 5]);
true true
```

2.3.7 Syndrome Space and Coset Leaders

`SyndromeSpace(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the $\mathbb{Z}_4$-submodule of $\mathbb{Z}_4^{\alpha+\beta-\delta-\kappa}$ isomorphic to $\mathbb{Z}_2^{\alpha+\gamma-2\kappa} \times \mathbb{Z}_4^{\beta-\gamma-\delta+\kappa}$ such that the first $\alpha+\gamma-2\kappa$ coordinates are of order two, that is, the space of syndrome vectors for C. The function also returns the $(\alpha+2\beta-\gamma-2\delta)$-dimensional binary vector space, which is the space of syndrome vectors for the corresponding binary code $C_{bin} = \Phi(C)$, where Φ is the Gray map. Note that these spaces are computed by using the function `InformationSpace(C)` applied to the additive dual code of C, produced by function `Dual(C)`.

`Syndrome(u, C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, and a vector u from the ambient space $V = \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ or $V_2 = \mathbb{Z}_2^{\alpha+2\beta}$, construct the syndrome of u relative to the code C, by using the parity check matrix given by the function `MinRowsParityCheckMatrix(C)`. The vector $u \in V$ can be given either as a vector in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as a tuple in the cartesian product set $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$. The syndrome is an element of the syndrome space of C, considered as the $\mathbb{Z}_4$-submodule of $\mathbb{Z}_4^{\alpha+\beta-\delta-\kappa}$ isomorphic to $\mathbb{Z}_2^{\alpha+\gamma-2\kappa} \times \mathbb{Z}_4^{\beta-\gamma-\delta+\kappa}$ such that the first $\alpha + \gamma - 2\kappa$ coordinates are of order two.

`CosetLeaders(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, with ambient space $V = \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, return a set of coset leaders (vectors of minimal Lee weight in their cosets) for C in V as an indexed set of vectors from V. The elements in V are represented as elements in $\mathbb{Z}_4^{\alpha+\beta}$ by replacing the ones in the first α coordinates by twos. This function also returns a map from the syndrome space of C onto the coset leaders (mapping a syndrome into its corresponding coset leader). Note that this function is only applicable when V and C are small.

Example H2E14

```
> C := Z2Z4ExtendedPerfectCode(2, 5);
> C;
(20, 67108864) Z2Z4-additive code of type (8, 12; 6, 10; 6)
Generator matrix:
[2 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 3 2]
[0 2 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 2]
[0 0 2 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0]
[0 0 0 2 0 0 0 0 0 0 0 0 1 0 1 0 0 3 0]
[0 0 0 0 2 0 0 0 0 0 0 0 0 0 1 0 0 3 3]
[0 0 0 0 0 2 0 0 0 0 0 0 0 0 1 0 0 1 3]
[0 0 0 0 0 0 2 0 0 0 0 0 0 0 1 0 0 1 1]
[0 0 0 0 0 0 0 2 0 0 0 0 0 0 1 0 0 3 1]
[0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 1]
[0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 2 1]
[0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1]
[0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 2 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 3 3]
[0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 2]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 2]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 2 2]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 2]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 1]
```

```
> alpha := C‘Alpha;
> beta := Length(C) - alpha;
> V4, V2, f, fbin := InformationSpace(C);
> V4s, V2s := SyndromeSpace(C);
> (#V4 * #V4s) eq (2^alpha * 4^beta);
true
> (#V2 * #V2s) eq (2^alpha * 4^beta);
true

> i := V4![2,0,2,0,2,0,1,3,0,0,0,1,3,0,0,0];
> c := f(i);
> c;
(0 0 2 0 2 2 0 2 3 1 0 0 0 3 0 0 0 1 0 2)
> u := c;
> u[11] := u[11] + 3;
> u;
(0 0 2 0 2 2 0 2 3 1 3 0 0 3 0 0 0 1 0 2)

> s := Syndrome(u, C);
> s in V4s;
true
> H := MinRowsParityCheckMatrix(C);
> s eq Z2Z4Mult(u, Transpose(H), alpha);
true
> s eq u*Transpose(H);
false

> L, mapCosetLeaders := CosetLeaders(C);
> errorVector := mapCosetLeaders(s);
> errorVector;
(0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0)
> errorVector in L;
true
> u-errorVector eq c;
true
```

2.4 The Standard Form

A $\mathbb{Z}_2\mathbb{Z}_4$-additive code is in *standard form* if its generator matrix is of the form:

$$\left(\begin{array}{cc|ccc} I_\kappa & T_b & 2T_2 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & 2T_1 & 2I_{\gamma-\kappa} & \mathbf{0} \\ \hline \mathbf{0} & S_b & S_q & R & I_\delta \end{array}\right),$$

where T_b, T_1, T_2, R, S_b are matrices over $\mathbb{Z}_2$ and S_q is a matrix over $\mathbb{Z}_4$. Any $\mathbb{Z}_2\mathbb{Z}_4$-additive code C is permutation-equivalent to a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C_{SF}

which is in standard form (see [6]). Two $\mathbb{Z}_2\mathbb{Z}_4$-additive codes which differ only by a coordinate permutation are said to be *permutation-equivalent*.

`StandardForm(C)`

Given any $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return a permutation-equivalent $\mathbb{Z}_2\mathbb{Z}_4$-additive code C_{SF} in standard form, together with the corresponding isomorphism from C to C_{SF}, the generator matrix in standard form, and the coordinate permutation used to define the isomorphism.

`IsStandardFormMatrix(M,[`α`,`β`,`γ`,`δ`,`κ`])`

Return `true` if and only if the matrix M over $\mathbb{Z}_4$, where the ones in the first α coordinates are represented by twos, is a generator matrix of a $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \gamma, \delta; \kappa)$ in standard form.

Example H2E15

We compute the standard form of a $\mathbb{Z}_2\mathbb{Z}_4$-additive code. Note that the number of rows in the general matrix of the standard code may be less than the number of rows in the matrix of the original code.

```
> R := RSpace(IntegerRing(4), 8);
> C := Z2Z4AdditiveCode([R![0,0,2,1,1,1,2,1], R![0,0,0,2,0,1,3,1],
                         R![0,0,0,0,2,1,3,1], R![0,2,2,0,0,2,2,0],
                         R![2,2,0,0,1,2,0,2]], 3);
> C;
(8, 256) Z2Z4-additive code of type (3, 5; 2, 3; 1)
Generator matrix:
[2 0 0 1 0 0 1 2]
[0 2 0 1 1 0 1 2]
[0 0 2 1 1 0 1 0]
[0 0 0 2 0 0 0 0]
[0 0 0 0 2 0 0 0]
[0 0 0 0 0 1 1 1]
[0 0 0 0 0 0 2 0]

> C_SF, f, G_SF, p := StandardForm(C);
> G_SF;
[2 0 2 2 0 0 0 0]
[0 0 0 0 2 0 0 0]
[0 2 2 2 0 1 0 0]
[0 2 0 2 1 0 1 0]
[0 2 0 3 1 0 0 1]
> IsStandardFormMatrix(G_SF, Z2Z4Type(C));
true
> C eq C_SF;
false
> C_SF eq Z2Z4AdditiveCode(G_SF, C`Alpha);
true
```

```
> C‘Code^p eq C_SF‘Code;
true
> {f(c) : c in C‘Code} eq Set(C_SF‘Code);
true
> {c^p : c in C‘Code} eq Set(C_SF‘Code);
true
```

2.5 Derived Binary and Quaternary Codes

2.5.1 The Gray Map

The $\mathbb{Z}_2\mathbb{Z}_4$-additive codes are subgroups C of $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ and the corresponding binary codes are $C_{bin} = \Phi(C)$, where Φ is the following extension of the usual Gray map: $\Phi : \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta \longrightarrow \mathbb{Z}_2^{n_{bin}}$, where $n_{bin} = \alpha + 2\beta$, given by

$$\Phi(x, y) = (x, \phi(y_1), \ldots, \phi(y_\beta)) \quad \forall x \in \mathbb{Z}_2^\alpha,\ \forall y = (y_1, \ldots, y_\beta) \in \mathbb{Z}_4^\beta;$$

where $\phi : \mathbb{Z}_4 \longrightarrow \mathbb{Z}_2^2$ is the usual Gray map, that is,

$$\phi(0) = (0, 0),\ \phi(1) = (0, 1),\ \phi(2) = (1, 1),\ \phi(3) = (1, 0).$$

This Gray map is an isometry which transforms Lee distances defined in codes C over $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ to Hamming distances defined in the binary codes $C_{bin} = \Phi(C)$ (see Subsection 2.6.4). Note that the length of the binary code C_{bin} is $n_{bin} = \alpha + 2\beta$.

`GrayMap(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return the Gray map for C. This is the map Φ from C to $\Phi(C)$, as defined above.

`GrayMapImage(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return the image of C under the Gray map as a sequence of vectors in $\mathbb{Z}_2^{\alpha+2\beta}$. As the resulting image may not be a binary linear code, a sequence of vectors is returned rather than a code.

`HasLinearGrayMapImage(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return `true` if and only if the image of C under the Gray map is a binary linear code. If so, the function also returns the image B as a binary linear code, together with the bijection $\Phi : C \to B$.

Example H2E16

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, we compute its image under the Gray map. This image is not necessarily a binary linear code.

```
> R := RSpace(IntegerRing(4), 4);
> C := Z2Z4AdditiveCode([R![2,0,0,2], R![0,1,0,3], R![0,0,1,3]], 1);
> C;
(4, 32) Z2Z4-additive code of type (1, 3; 1, 2; 1)
Generator matrix:
[2 0 0 2]
[0 1 0 3]
[0 0 1 3]

> HasLinearGrayMapImage(C);
false
> Cb := GrayMapImage(C);
> #Cb;
32

> R := RSpace(IntegerRing(4), 6);
> D := Z2Z4AdditiveCode([R![2,2,1,1,3,1], R![2,2,2,2,2,2], R![0,0,1,1,1,3]], 2);
> D;
(6, 16) Z2Z4-additive code of type (2, 4; 2, 1; 1)
Generator matrix:
[2 2 0 0 0 0]
[0 0 1 1 1 3]
[0 0 0 0 2 2]

> f := GrayMap(D);
> Name(D, 1);
(2 2 0 0 0 0)
> f(Name(D, 1));
(1 1 0 0 0 0 0 0 0 0)
> Name(D, 2);
(0 0 1 1 1 3)
> f(Name(D, 2));
(0 0 0 1 0 1 0 1 1 0)
> Name(D, 3);
(0 0 0 0 2 2)
> f(Name(D, 3));
(0 0 0 0 0 0 1 1 1 1)
> isL, B, f := HasLinearGrayMapImage(D);
> isL;
true
> B;
[10, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 1 0 0 0 0 0 0 0 0]
[0 0 1 0 1 0 0 1 1 0]
[0 0 0 1 0 1 0 1 1 0]
[0 0 0 0 0 0 1 1 1 1]

> f(Name(D, 1)) in B;
```

```
true
> f(Name(D, 2)) in B;
true
> f(Name(D, 3)) in B;
true
> Length(B) eq BinaryLength(D);
true
```

2.5.2 Subcodes C_X and C_Y

`LinearBinaryCode(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the binary linear code C_X of length α which is the punctured code of C by deleting the coordinates outside X, where X is the set of the first α coordinates.

`LinearQuaternaryCode(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the quaternary linear code C_Y of length β which is the punctured code of C by deleting the coordinates outside Y, where Y is the set of the last β coordinates.

Example H2E17

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, we compute the corresponding binary linear code C_X and the corresponding quaternary linear code C_Y.

```
> R  := RSpace(IntegerRing(4), 4);
> C  := Z2Z4AdditiveCode([R![2,0,0,2], R![0,1,0,3], R![0,0,1,3]], 1);
> CX := LinearBinaryCode(C);
> CX;
[1, 1, 1] Cyclic Linear Code over GF(2)
> CY := LinearQuaternaryCode(C);
> CY;
((3, 4^2 2^1)) Cyclic Linear Code over IntegerRing(4)
Generator matrix:
[1 0 1]
[0 1 1]
[0 0 2]
> _, n := Length(C);
> alpha := n[1]; beta := n[2];
> alpha; beta;
1
3
> (Length(CX) eq alpha) and (Length(CY) eq beta);
true
```

2.5.3 Span and Kernel Codes

`SpanZ2Code(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return $S_C = \Phi^{-1}(S_{bin})$ as a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, and $S_{bin} = \langle C_{bin} \rangle$, that is the linear span of C_{bin}, as a binary linear code of length $\alpha + 2\beta$, where $C_{bin} = \Phi(C)$ and Φ is the Gray map.

`KernelZ2Code(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return its kernel K_C as a $\mathbb{Z}_2\mathbb{Z}_4$-additive subcode of C, and $K_{bin} = \Phi(K_C)$ as a binary linear subcode of C_{bin} of length $\alpha + 2\beta$, where $C_{bin} = \Phi(C)$ and Φ is the Gray map.

The kernel K_C contains the codewords v such that $2v * u \in C$ for all $u \in C$, where $*$ denotes the component-wise product. Equivalently, the kernel $K_{bin} = \Phi(K_C)$ contains the codewords $c \in C_{bin}$ such that $c + C_{bin} = C_{bin}$, where $C_{bin} = \Phi(C)$ and Φ is the Gray map.

`KernelCosetRepresentatives(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the coset representatives $[c_1, \ldots, c_t]$ as a sequence of codewords of C, such that $C = K_C \cup \bigcup_{i=1}^{t} \big(K_C + c_i\big)$, where K_C is the kernel of C as a $\mathbb{Z}_2\mathbb{Z}_4$-additive subcode. It also returns the coset representatives of the corresponding binary code $C_{bin} = \Phi(C)$ as a sequence of binary codewords $[\Phi(c_1), \ldots, \Phi(c_t)]$, such that $C_{bin} = K_{bin} \cup \bigcup_{i=1}^{t} \big(K_{bin} + \Phi(c_i)\big)$, where $K_{bin} = \Phi(K_C)$ and Φ is the Gray map.

`DimensionOfSpanZ2(C)`

`RankZ2(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return the dimension of the linear span of C_{bin}, that is, the dimension of $\langle C_{bin} \rangle$, where $C_{bin} = \Phi(C)$ and Φ is the Gray map.

`DimensionOfKernelZ2(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return the dimension of the Gray map image of its $\mathbb{Z}_2\mathbb{Z}_4$-additive kernel K_C, that is the dimension of $K_{bin} = \Phi(K_C)$, where Φ is the Gray map. Note that K_{bin} is always a binary linear code.

Example H2E18

```
> M1 := Matrix(Integers(4), [[0,2,2,1,3,1],
```

```
                              [0,0,0,2,2,2]]);
> C1 := Z2Z4AdditiveCode(M1, 3);
> HasLinearGrayMapImage(C1);
true [9, 2, 5] Linear Code over GF(2)
Generator matrix:
[0 1 1 0 1 1 0 0 1]
[0 0 0 1 1 1 1 1 1]
Mapping from: ((6, 4^1 2^0)) Linear Code over IntegerRing(4) to [9, 2, 5] Linear
Code over GF(2) given by a rule
> DimensionOfSpanZ2(C1) eq DimensionOfKernelZ2(C1);
true
> S, Sb := SpanZ2Code(C1);
> K, Kb := KernelZ2Code(C1);
> (K eq C1) and (C1 eq S);
true
> Kb eq Sb;
true

> M2 := Matrix(Integers(4),[[2,0,0,0,0,0,0,0,0,0,0,2],
                            [0,2,0,0,0,0,0,0,0,0,0,2],
                            [0,0,2,0,0,0,0,0,0,0,0,2],
                            [0,0,0,2,0,0,0,0,0,0,0,2],
                            [0,0,0,0,2,0,0,0,0,0,0,0],
                            [0,0,0,0,0,2,0,0,0,0,0,2],
                            [0,0,0,0,0,0,1,0,0,0,0,1],
                            [0,0,0,0,0,0,0,1,0,1,0,0],
                            [0,0,0,0,0,0,0,0,1,0,0,1],
                            [0,0,0,0,0,0,0,0,0,2,0,2],
                            [0,0,0,0,0,0,0,0,0,0,1,2]]);
> C2 := Z2Z4AdditiveCode(M2, 6);
> HasLinearGrayMapImage(C2);
false
> DimensionOfSpanZ2(C2) eq DimensionOfKernelZ2(C2);
false
> S, Sb := SpanZ2Code(C2);
> K, Kb := KernelZ2Code(C2);
> (K subset C2) and (C2 subset S);
true
> Kb subset Sb;
true
```

2.5.4 Coset Representatives

`CosetRepresentatives(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, with ambient space $V = \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, return a set of coset representatives (not necessarily of minimal weight in their cosets) for C in V as an indexed set of vectors from V. The elements in V are represented as elements in $\mathbb{Z}_4^{\alpha+\beta}$ by replacing the ones in the first α coordinates by twos. The set of coset representatives $\{c_0, c_1, \ldots, c_t\}$ satisfies the conditions that c_0 is the zero codeword, and $V = \bigcup_{i=0}^{t} \big(C + c_i\big)$. Note that this function is only applicable when V and C are small.

`CosetRepresentatives(C, S)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, and a $\mathbb{Z}_2\mathbb{Z}_4$-additive subcode S of C, return a set of coset representatives (not necessarily of minimal weight in their cosets) for S in C as an indexed set of codewords from C. The codewords in $C \subseteq \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ are represented as elements in $\mathbb{Z}_4^{\alpha+\beta}$ by replacing the ones in the first α coordinates by twos. The set of coset representatives $\{c_0, c_1, \ldots, c_t\}$ satisfies the conditions that c_0 is the zero codeword, and $C = \bigcup_{i=0}^{t} \big(S + c_i\big)$. Note that this function is only applicable when S and C are small.

Example H2E19

```
> R := RSpace(Integers(4), 5);
> C := Z2Z4AdditiveCode([R![2,0,0,2,0],
                         R![2,0,1,2,3],
                         R![0,0,2,1,3]], 1);
> L := CosetRepresentatives(C);
> {x : x in Set(R) | x[1] in {0,2}} eq {v+ci : v in Set(C), ci in L};
true

> K := KernelZ2Code(C);
> L := CosetRepresentatives(C, K);
> {C!0} join Set(KernelCosetRepresentatives(C)) eq L;
true
> Set(C) eq {v+ci : v in Set(K), ci in L};
true
```

2.6 Operations on Codewords

A $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ is represented as a quaternary linear code (or equivalently, a subspace of $\mathbb{Z}_4^{\alpha+\beta}$), where the ones in the first α coordinates are represented by twos (see Subsection 2.2.1). The elements in $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ are represented as elements in $\mathbb{Z}_4^{\alpha+\beta}$ by replacing the ones in the first α coordinates by twos, or as tuples in the cartesian product set $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$.

2.6.1 Construction of a Codeword

`C ! [` $a_1, \ldots, a_n$ `]`

`elt< C‘Code |` $a_1, \ldots, a_n$ `>`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, which is represented as a subspace of $\mathbb{Z}_4^n$ with $n = \alpha + \beta$, and elements $a_1, \ldots, a_n$ belonging to $\mathbb{Z}_4$, construct the codeword $(a_1, \ldots, a_n)$ of C. It is checked whether the vector $(a_1, \ldots, a_n)$ is an element of `C‘Code`.

`C ! u`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, which is represented as a subspace of $\mathbb{Z}_4^{\alpha+\beta}$, and an element u belonging to $\mathbb{Z}_4^{\alpha+\beta}$, create the codeword of C corresponding to u. The function will fail if u does not belong to C.

`C ! 0`

The zero codeword of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C.

`Random(C)`

A random codeword of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, which is a vector in $\mathbb{Z}_4^{\alpha+\beta}$ where the ones in the first α coordinates are represented by twos.

Example H2E20

We create some elements of a $\mathbb{Z}_2\mathbb{Z}_4$-additive code.

```
> R := RSpace(IntegerRing(4), 4);
> C := Z2Z4AdditiveCode([R![2,0,0,2], R![0,1,0,3], R![0,0,1,3]], 1);
> C!0;
(0 0 0 0)
> C![2,0,0,2];
(2 0 0 2)
> elt< C‘Code | 2,1,1,0 >
(2 1 1 0)
> Random(C);
(2 0 1 1)
```

If the given vector does not lie in the given $\mathbb{Z}_2\mathbb{Z}_4$-additive code, then an error will result.

```
> C![2,1,0,0];

>> C![2,1,0,0]
    ^
Runtime error in '!': Illegal coercion
```

2.6.2 Operations on Codewords and Vectors

`u + v`

Sum of the codewords u and v, where u and v belong to the same $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$. The codewords u and v can be given either as vectors in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as tuples in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

`- u`

Additive inverse of the codeword u belonging to the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$. The codeword u can be given either as a vector in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as a tuple in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

`u - v`

Difference of the codewords u and v, where u and v belong to the same $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$. The codewords u and v can be given either as vectors in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as tuples in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

`a * u`

Given an element a belonging to $\mathbb{Z}_4$, and a codeword u belonging to the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the codeword $a \cdot u$. The codeword u can be given either as a vector in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as a tuple in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

`u · v`

`Z2Z4InnerProduct(u, v)`

The inner product $\langle u, v \rangle$ in $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$ defined in Subsection 2.3.4. The vectors u and v are represented as tuples in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

`Z2Z4InnerProduct(u, v, α)`

The inner product $\langle u, v \rangle$ in $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$ defined in Subsection 2.3.4. The vectors u and v are represented as elements in $\mathbb{Z}_4^{\alpha+\beta}$ by replacing the ones in the first α coordinates by twos.

`u * v`

`Z2Z4StarProduct(u, v)`

The component-wise product of u and v in $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$. The vectors u and v are represented as tuples in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

`Z2Z4StarProduct(u, v, α)`

The component-wise product of u and v in $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$. The vectors u and v are represented as elements in $\mathbb{Z}_4^{\alpha+\beta}$ by replacing the ones in the first α coordinates by twos.

`Normalize(u)`

Given a codeword u belonging to the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, which is represented as a subspace of $\mathbb{Z}_4^{\alpha+\beta}$, return the normalization of u, which is the unique vector v such that $v = a \cdot u$ for some scalar a in $\mathbb{Z}_4$ such that the first non-zero entry of v is the canonical associate in $\mathbb{Z}_4$ of the first non-zero entry of u (v is zero if u is zero). The codeword u can be given either as a vector in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as a tuple in the cartesian product set $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$.

`Support(u)`

Given a codeword u belonging to the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, which is represented as a subspace of $\mathbb{Z}_4^{\alpha+\beta}$, return its support as a subset of the integer set $\{1, \ldots, \alpha + \beta\}$. The support of u consists of the coordinates at which u has non-zero entries. The codeword u can be given either as a vector in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as a tuple in the cartesian product set $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$.

`Coordinates(C, u)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C and a codeword u of C, return the coordinates of u with respect to C. The coordinates of u are returned as a sequence $Q = [a_1, \ldots, a_k]$ of elements from $\mathbb{Z}_4$ so that $u = a_1 \cdot C.1 + \cdots + a_k \cdot C.k$. The codeword u can be given either as a vector in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as a tuple in the cartesian product set $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$.

`Rotate(u, k)`

Given a vector u, return the vector obtained from u by cyclically shifting its components to the right by k coordinate positions. If u is an element of the cartesian product set $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, return the element obtained from u by cyclically shifting to the right its first α components and last β components, separately, by k coordinate positions.

`Rotate(~u, k)`

Given a vector u, destructively rotate u by k coordinate positions. If u is an element of the cartesian product set $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$, return the element obtained from u by cyclically shifting to the right its first α components and last β components, separately, by k coordinate positions.

`Parent(u)`

Given a codeword u belonging to the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return the $\mathbb{Z}_4$-space $\mathbb{Z}_4^{\alpha+\beta}$, where C is contained after replacing the ones in the first α coordinates by twos, if u is given as a vector in $\mathbb{Z}_4^{\alpha+\beta}$. It returns the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$ if u is given as a tuple in this set.

Example H2E21

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, we explore various operations on its codewords represented as vector in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos.

```
> R := RSpace(IntegerRing(4), 4);
> C := Z2Z4AdditiveCode([R![2,0,0,2], R![0,1,0,3], R![0,0,1,3]], 1);
> u := Name(C, 1);
> v := Name(C, 2);
> u; v;
(2 0 0 2)
(0 1 0 3)
> u+v;
(2 1 0 1)
> 2*v;
(0 2 0 2)
> u+v in C;
true
> Z2Z4InnerProduct(u, v, 1);
2
> Support(u);
{ 1, 4 }
> Coordinates(C, u+2*v);
[ 1, 2, 0 ]
> Parent(u) eq R;
true
```

Example H2E22

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, we explore various operations on its codewords represented as tuples in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

```
> R2 := RSpace(IntegerRing(2), 1);
> R4 := RSpace(IntegerRing(4), 3);
> R2R4 := CartesianProduct(R2, R4);
> C := Z2Z4AdditiveCode([R2R4!<R2![1], R4![0,0,2]>,
                         R2R4!<R2![0], R4![1,0,3]>,
                         R2R4!<R2![0], R4![0,1,3]>]);
> u := FromZ4toZ2Z4(C![2,0,0,2], 1);
> v := FromZ4toZ2Z4(C![0,1,0,3], 1);
> u; v;
<(1), (0 0 2)>
<(0), (1 0 3)>
```

```
> u + v;
<(1), (1 0 1)>
> 2*v;
<(0), (2 0 2)>
> u+v in C;
true
> Z2Z4InnerProduct(u, v);
2
> Support(u);
{ 1, 4 }
> Coordinates(C, u+2*v);
[ 1, 2, 0 ]
> Parent(u) eq R2R4;
true
```

2.6.3 Operations on Vectors and Matrices

`Z2Z4Mult(u, A, α)`

Given a vector u belonging to $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ represented as an element in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; and a matrix A over $\mathbb{Z}_4$ having $\alpha + \beta$ rows and the entries in the first α rows in $\{0, 2\}$; return the vector $u * A$.

`Z2Z4Mult(A, B, α)`

Given a $m \times (\alpha+\beta)$ matrix A where the rows belong to $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ represented as $\mathbb{Z}_4^{\alpha+\beta}$, that is, having the entries in the first α columns in $\{0, 2\}$; and a $(\alpha + \beta) \times p$ matrix B over $\mathbb{Z}_4$ having the entries in the first α rows in $\{0, 2\}$, return the $m \times p$ matrix $A * B$ having in the i-th row the vector `Z2Z4Mult(`u_i`, B)`, where u_i is the i-th row of A.

Example H2E23

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, an information vector is encoded by using the generator matrix, it is checked that the syndrome of the obtained codeword is zero, then some errors are introduced to give the vector u, and finally the syndrome of u is computed by using the parity check matrix.

```
> R := RSpace(IntegerRing(4), 6);
> C := Z2Z4AdditiveCode([R![2,0,0,2,0,2],
                         R![0,2,0,3,1,1],
                         R![0,0,1,3,0,2]], 2);

> G := MinRowsGeneratorMatrix(C);
> i := RSpace(IntegerRing(4),3)![2,1,3];
> c := Z2Z4Mult(i, G, Z2Z4Type(C)[3]);
> c;
```

```
(2 0 3 1 2 2)
> c in C;
true
> Z2Z4Mult(i, G, Z2Z4Type(C)[3]) eq i*G;
false

> H := MinRowsParityCheckMatrix(C);
> Z2Z4Mult(c, Transpose(H), C`Alpha);
(0 0 0)
> Z2Z4Mult(c, Transpose(H), C`Alpha) eq c*Transpose(H);
false
> u := c;
> u[2] := u[2] + 2;
> u[4] := u[4] + 3;
> Z2Z4Mult(u, Transpose(H), C`Alpha);
(2 2 1)
> Z2Z4Mult(u, Transpose(H), C`Alpha) eq u*Transpose(H);
false
```

2.6.4 Distance and Weight

`Weight(v)`

The Hamming weight of the codeword v, i.e., the number of non-zero components of v. The codeword v can be given either as a vector in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as a tuple in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

`Distance(u, v)`

The Hamming distance between the codewords u and v, where u and v belong to the same $\mathbb{Z}_2\mathbb{Z}_4$-additive code C. This is defined to be the Hamming weight of $u - v$. The codewords u and v can be given either as vectors in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as tuples in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

`LeeWeight(v)`

The Lee weight of the codeword v, i.e., the Hamming weight of the binary part (that is, the first α coordinates) of v plus the Lee weight of the quaternary part (that is, the rest of the coordinates) of v (see [6]). Equivalently, it corresponds to the number of non-zero components of $\Phi(v)$, where Φ is the Gray map defined in Subsection 2.5.1. The codeword v is represented as a tuple in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

`LeeWeight(v, α)`

The Lee weight of the codeword v, i.e., the Hamming weight of the binary part (that is, the first α coordinates) of v plus the Lee weight of the quaternary part (that is, the rest of the coordinates) of v (see [6]). Equivalently, it corresponds to the number of non-zero components of $\Phi(v)$, where Φ is the Gray map defined in Subsection 2.5.1. The codeword v is represented as an element in $\mathbb{Z}_4^{\alpha+\beta}$ by replacing the ones in the first α coordinates by twos.

`LeeDistance(u, v)`

The Lee distance between the codewords u and v, where u and v belong to the same $\mathbb{Z}_2\mathbb{Z}_4$-additive code C. This is defined to be the Lee weight of $u - v$. The codewords u and v are represented as tuples in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

`LeeDistance(u, v, α)`

The Lee distance between the codewords u and v, where u and v belong to the same $\mathbb{Z}_2\mathbb{Z}_4$-additive code C. This is defined to be the Lee weight of $u - v$. The codewords u and v are represented as elements in $\mathbb{Z}_4^{\alpha+\beta}$ by replacing the ones in the first α coordinates by twos.

Example H2E24

We calculate the Hamming weight and distance of some vectors in $\mathbb{Z}_4^{\alpha+\beta}$, as well as the Lee weight and distance of these vectors. Note that when $\alpha = 0$, the functions for the Lee weight and distance return the same as that the corresponding functions for vectors over $\mathbb{Z}_4$.

```
> R := RSpace(IntegerRing(4), 4);
> u := R![2,1,2,3];
> v := R![0,0,2,1];
> Distance(u, v);
3
> Distance(u, v) eq Weight(u-v);
true
> LeeDistance(u, v, 1);
4
> LeeDistance(u, v, 1) eq LeeWeight(u-v, 1);
true
> LeeDistance(u, v, 0) eq LeeDistance(u, v);
true
> LeeWeight(u, 0) eq LeeWeight(u);
true
```

Example H2E25

We calculate the Hamming weight and distance of some tuples in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$, as well as the Lee weight and distance of these tuples.

```
> R2 := RSpace(IntegerRing(2), 1);
> R4 := RSpace(IntegerRing(4), 3);
> R2R4 := CartesianProduct(R2, R4);
> u := FromZ4toZ2Z4(RSpace(Integers(4), 4)![2,1,2,3], 1);
> v := FromZ4toZ2Z4(RSpace(Integers(4), 4)![0,0,2,1], 1);
> u; v;
<(1), (1 2 3)>
<(0), (0 2 1)>
> Distance(u, v);
3
> Distance(u, v) eq Weight(u-v);
true
> LeeDistance(u, v);
4
> LeeDistance(u, v) eq LeeWeight(u-v);
true
```

2.6.5 Accessing Components of a Codeword

`u[i]`

Given a codeword u belonging to the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, which is represented as a subspace of $\mathbb{Z}_4^{\alpha+\beta}$, return the i-th component of u (as an element of $\mathbb{Z}_4$).

`u[i] := x;`

Given an element u belonging to a $\mathbb{Z}_2\mathbb{Z}_4$-additive subcode C of the full $\mathbb{Z}_4$-space $\mathbb{Z}_4^{\alpha+\beta}$, a positive integer i, $1 \leq i \leq \alpha+\beta$, and an element x of $\mathbb{Z}_4$, this function returns a vector in $\mathbb{Z}_4^{\alpha+\beta}$ which is u with its i-th component redefined to be x. It is not checked whether the elements in the first α coordinates are in $\{0, 2\}$.

2.7 Boolean Predicates

Again, note that all elements in $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$ are represented as elements in $\mathbb{Z}_4^{\alpha+\beta}$ by replacing the ones in the first α coordinates by twos.

2.7.1 Membership and Equality

`u in C`

Return `true` if and only if the vector u belongs to the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$. The vector u can be given either as a vector in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as a tuple in the cartesian product set $\mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$.

`u notin C`

Return `true` if and only if the vector u does not belong to the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$. The vector u can be given either as a vector in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as a tuple in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

`C subset D`

Return `true` if and only if the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C is a subcode of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code D.

`C notsubset D`

Return `true` if and only if the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C is not a subcode of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code D.

`C eq D`

Return `true` if and only if the $\mathbb{Z}_2\mathbb{Z}_4$-additive codes C and D are equal.

`C ne D`

Return `true` if and only if the $\mathbb{Z}_2\mathbb{Z}_4$-additive codes C and D are not equal.

`IsZero(u)`

Return `true` if and only if the codeword u is the zero vector. The codeword u can be given either as a vector in $\mathbb{Z}_4^{\alpha+\beta}$ or as a tuple in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

2.7.2 Properties

`IsZ2Z4AdditiveCode(C)`

Return `true` if and only if C is a $\mathbb{Z}_2\mathbb{Z}_4$-additive code.

`IsSelfDual(C)`

Return `true` if and only if the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C is additive self-dual, i.e., C equals the additive dual code of C.

`IsSelfOrthogonal(C)`

Return `true` if and only if the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C is additive self-orthogonal, that is, return whether C is contained in the additive dual code of C.

`IsSeparable(C)`

Return `true` if and only if C is a separable $\mathbb{Z}_2\mathbb{Z}_4$-additive code. A $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ is separable if $C = C_X \times C_Y$, where C_X and C_Y are the punctured codes of C by deleting the last β and first α coordinates, respectively.

`IsAntipodal(C)`

Return `true` if and only if C is an antipodal $\mathbb{Z}_2\mathbb{Z}_4$-additive code. A $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ is antipodal if $C_{bin} = \Phi(C)$ is antipodal, where Φ is the Gray map defined in Subsection 2.5.1, o equivalently if $(1, \ldots, 1 \mid 2, \ldots, 2)$ belongs to C.

`IsCyclic(C)`

Return `true` if and only if the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C is cyclic.

Example H2E26

We consider a $\mathbb{Z}_2\mathbb{Z}_4$-additive code and examine some of its properties.

```
> R := RSpace(IntegerRing(4), 5);
> C := Z2Z4AdditiveCode([R![2,2,2,0,0], R![0,0,0,2,0], R![0,2,1,0,1]], 2);
> C;
(5, 16) Z2Z4-additive code of type (2, 3; 2, 1; 1)
Generator matrix:
[2 0 1 0 3]
[0 2 1 0 1]
[0 0 2 0 2]
[0 0 0 2 0]

> IsSelfOrthogonal(C);
true
> IsSelfDual(C);
true
> C eq Dual(C);
true
> IsSeparable(C);
false
> IsAntipodal(C);
false
> IsCyclic(C);
false
```

`IsCompletelyRegular(C : parameters)`

`MaximumCosetWeight`	RngIntElt	*Default*: ρ_C
`MaximumTime`	RngIntElt	*Default*: ∞

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return `true` if and only if C_{bin} is completely regular, where $C_{bin} = \Phi(C)$ and Φ is the Gray map defined in Subsection 2.5.1. If return `true`, then it also returns the intersection array, a sequence with the number of coset leaders of C_{bin} of each weight, and the covering radius. Note that, if no optional parameters are specified, then this function is only applicable when C is small.

A code $C_{bin} \subseteq \mathbb{Z}_2^{\alpha+2\beta}$ is called completely regular if, for all $i \geq 0$, every vector $x \in C_i$ has the same number c_i of neighbours in C_{i-1} and the same number b_i of neighbours in C_{i+1}, where $C_i = \{x \in \mathbb{Z}_2^{\alpha+2\beta} : d(x, C_{bin}) = i\}$. The intersection array is given as a sequence with the parameters $[[b_0, \ldots, b_{\rho_C-1}], [c_1, \ldots, c_{\rho_C}]]$, where ρ_C is the covering radius of C_{bin}.

The parameter `MaximumCosetWeight` sets the maximum weight of the coset leaders which are used in the calculation to check whether the code is completely regular or not. In this case, the function returns `true` if and only if for all $0 \leq i \leq$ `MaximumCosetWeight`, every vector $x \in C_i$ has the same number c_i of neighbours in C_{i-1} and the same number b_i of neighbours in C_{i+1}. The intersection array is given as a sequence with the parameters $[[b_0, \ldots, b_{m-1}], [c_1, \ldots, c_m]]$, where $m =$ `MaximumCosetWeight`. The sequence with the number of coset leaders of C_{bin} of each weight is given until `MaximumCosetWeight`. The covering radius is not given unless `MaximumCosetWeight` $\geq \rho_C$. The default value is ρ_C.

The parameter `MaximumTime` sets a time limit (in seconds of "user time") after which the calculation is aborted. According to this parameter, there is a `MaximumCosetWeight` depending on the maximum weight of the computed coset leaders. In this case, the function returns the same as if this parameter `MaximumCosetWeight` had been specified. The default value is ∞, when there is no restriction on time.

Example H2E27

```
> M := Matrix(Integers(4),
       [[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2],
        [0,2,0,2,0,2,0,2,0,2,0,2,0,2,0,2,0,0,0,0,0,0,0,0,0,0],
        [0,2,0,2,0,2,0,2,0,2,0,2,0,2,0,2,1,1,1,1,1,1,1,1,1,1],
        [0,0,2,2,0,0,2,2,0,0,2,2,0,0,2,2,0,2,0,2,0,2,0,2,0,2],
        [0,0,0,2,0,0,0,2,0,0,0,2,0,0,0,2,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,2,0,0,0,2,0,0,0,2,0,0,0,2,0,1,0,1,0,1,0,1,0,1],
        [0,0,0,0,2,2,2,2,0,0,0,0,2,2,2,2,0,0,2,2,0,0,2,2,0,0],
        [0,0,0,0,0,2,0,2,0,0,0,0,0,2,0,2,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,2,0,2,0,0,0,0,0,2,0,2,0,0,1,1,0,0,1,1,0,0],
        [0,0,0,0,0,0,2,2,0,0,0,0,0,0,2,2,0,0,0,2,0,0,0,2,0,0],
        [0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,2,0,0,0,1,0,0,0,1,0,0],
        [0,0,0,0,0,0,0,0,2,2,2,2,2,2,2,2,0,0,0,0,2,2,2,2,0,0],
        [0,0,0,0,0,0,0,0,0,2,0,2,0,2,0,2,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,2,0,2,0,2,0,2,0,0,0,0,1,1,1,1,0,0],
        [0,0,0,0,0,0,0,0,0,0,2,2,0,0,2,2,0,0,0,0,0,0,2,0,2,0],
        [0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,2,0,0,0,0,0,0,1,0,1,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,2,2,2,2,0,0,0,0,0,0,2,2,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,2,0,0,0,0,0,0,0,1,1,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,2,0,0,0,0,0,0,0,0,2,2]]);
```

```
> C := Z2Z4AdditiveCode(M, 16);

> IsCompletelyRegular(C);
true [
    [ 32, 31 ],
    [ 1, 32 ]
]
[ 32, 31 ]
2
> IsCompletelyRegular(C : MaximumCosetWeight := 1);
true [
    [ 32 ],
    [ 1 ]
]
[ 32 ]
```

2.8 The Weight Distribution

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code of type $(\alpha, \beta; \gamma, \delta; \kappa)$, we recall that the Lee weight of a codeword v is the Hamming weight of the binary part of v (that is, the first α coordinates) plus the Lee weight of the quaternary part of v (that is, the rest of the coordinates) (see [6]). Moreover, it corresponds to the Hamming weight of $\Phi(v)$, where Φ is the Gray map defined in Subsection 2.5.1. For $\mathbb{Z}_2\mathbb{Z}_4$-additive codes, Lee weight and Lee distance distributions coincide (in particular, minimum Lee weight and minimum Lee distance coincide).

There are five different methods available to compute the minimum Lee weight of a $\mathbb{Z}_2\mathbb{Z}_4$-additive code. The first one is by using brute force, which is equivalent to compute the whole Lee weight distribution of the code. The second one is by using the representation of the code as the union of cosets of the kernel defined in Subsection 2.5.3, and the known Brouwer-Zimmermann's algorithm applied to binary linear codes given by the Gray map image of cosets of the kernel [26]. The third method uses Brouwer's algorithm adapted to work directly with $\mathbb{Z}_2\mathbb{Z}_4$-additive codes, and the fourth one uses Zimmermann's variation (also known as Brouwer-Zimmermann method) adapted to $\mathbb{Z}_2\mathbb{Z}_4$-additive codes. The fifth method considers the code as a quaternary code and uses the minimum weight function already implemented in Magma. Each method may perform better in different circumstances, which are usually related to the parameters of the code and the minimum distance itself. For small codes, the brute force method can be faster than any other method, but as the number of codewords grows it becomes unfeasible.

Brouwer's and Brouwer-Zimmermann algorithms are enumeration methods. They are based on having different generator matrices of the code in standard form, each one with respect to a different set of information coordinate positions. In each step of the enumeration process, all subsets of r rows of the generator matrices are used. After each step, a lower and upper bounds of the minimum weight are obtained. When the upper bound is equal or smaller than the lower bound, the result is obtained, without enumerating necessarily all codewords. The difference between these two methods is that Brouwer's original algorithm only uses generator matrices with no overlap between information coordinate positions, while Zimmerman's variation allows for an overlap between them in order to produce more generator matrices and benefit from all coordinate positions. The reader is referred to [2, 27] for more information on these algorithms.

Both Brouwer's and Brouwer-Zimmermann methods perform well for codes with a low minimum distance. Zimmermann's variation improves greatly when Brouwer's method leaves many coordinate positions unused. For example, this occurs when there are many more quaternary coordinates than binary coordinates. In this case, Brouwer's method produces different information sets with disjoint binary coordinates positions only until the binary coordinates have run out. On the other hand, Zimmermann's variation reuses some binary coordinates along with new quaternary coordinates in order to use all coordinate positions. Using all coordinate positions helps to increase the lower bound faster, but at the same time, the method is slowed down because of the rank deficits of the generator matrices (the number of reused information coordinates associated to the matrix). There are cases where many of these matrices have large rank deficits. The method automatically discards the matrices that do not contribute to a faster increase of the lower bound, but there may be matrices that do contribute and do not offset the increase in the number of enumerated codewords in each step. In these cases, Brouwer's method may perform better.

The method based on the cosets of the kernel is not as much affected by a high minimum distance and performs well in general. However, the main weakness of the method is that it becomes slower when the number of cosets grows. This is controlled exponentially by the parameter δ, since the codewords of order two are always included in the kernel. This method performs particularly well compared to the Brouwer-Zimmermann method when $\gamma > \kappa$. In this case, there are two types of information coordinate positions: free and torsion positions. Using the Brouwer-Zimmermann method, only the free part (δ positions) contributes to increase the lower bound, while the torsion part ($\gamma - \kappa$ positions) is treated as an overlap, and hence it induces a rank deficit. On the other hand, the method based on the kernel works

directly on binary linear codes, which do not suffer from this torsion problem.

Finally, the fifth method considers the code as a quaternary code, by multiplying the first α coordinates by 2 and by duplicating the last β coordinates. The resulting code is a quaternary code of length $\alpha+2\beta$ and type $2^\gamma 4^\delta$ with the same weight distribution. Note that this method does not consider the first κ positions as information coordinate positions, hence it does not benefit from a large value of κ.

The functions defined in this package include a selector function in order to choose the most suitable method. In general, this is not easy to know without computing the minimum distance itself. Assuming a random $\mathbb{Z}_2\mathbb{Z}_4$-additive code, a probabilistic argument is used along with the concept of work factors in order to make a reasonable decision. Surely, there are codes for which the selected method may not be optimal. In any case, the user can also decide which method to use by setting the parameter `Method` to `"Distribution"`, `"KernelCosets"`, `"Brouwer"`, `"Zimmermann"`, or `"Quaternary"`.

The functions in this section take into account the following attributes: `MinimumLeeWeight`, `MinimumLeeWeightWord`, `LeeWeightDistribution`, `MinimumLeeWeightLowerBound`, `MinimumLeeWeightUpperBound` associated to a $\mathbb{Z}_2\mathbb{Z}_4$-additive code. The verbose flag `IgnoreWeightAttributes` is set to level 0 by default. In this case, the functions check whether these attributes are already assigned and, if they are, return them directly. If the flag is set to level 1, these attributes are ignored. This can be useful to perform tests and comparisons between the different available methods.

2.8.1 The Minimum Weight

`Z2Z4MinimumLeeWeight(C : parameters)`

`Z2Z4MinimumLeeDistance(C : parameters)`

`Method` MonStgElt *Default*: "Auto"

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return the minimum Lee weight of the codewords belonging to the code C, which is also the minimum Lee distance between any two codewords.

Depending on the parameters of the code C, some methods to obtain the minimum Lee weight may be faster than others. For example, sometimes, a brute force calculation of the entire Lee weight distribution can be a faster way for small codes. When the parameter `Method` is set to the default `"Auto"`, then the method is internally chosen. The user can specify which method they want to use, setting the parameter `Method` to `"Distribution"`, `"KernelCosets"`, `"Brouwer"`, `"Zimmermann"`, or `"Quaternary"`.

`MinimumWord(C)`

`Method` MonStgElt *Default*: "Auto"

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return one codeword of the code C having minimum Lee weight.

Depending on the parameters of the code C, some methods to obtain one codeword of minimum Lee weight may be faster than others. For example, sometimes, a brute force calculation of the entire Lee weight distribution can be a faster way for small codes. When the parameter `Method` is set to the default `"Auto"`, then the method is internally chosen. The user can specify which method they want to use, setting the parameter `Method` to `"Distribution"`, `"KernelCosets"`, `"Brouwer"`, `"Zimmermann"`, or `"Quaternary"`.

`MinimumWords(C : parameters)`

`NumWords` RngIntElt *Default*: -

`Method` MonStgElt *Default*: "Auto"

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, return the set of all codewords of C having minimum Lee weight. If `NumWords` is set to a non-negative integer, then the algorithm will terminate after at least that total of codewords have been found.

Depending on the parameters of the code C, some methods to collect the codewords of minimum Lee weight may be faster than others. For example, sometimes, a brute force calculation of the entire Lee weight distribution can be a faster way for small codes. When the parameter `Method` is set to the default `"Auto"`, then the method is internally chosen. The user can specify which method they want to use, setting the parameter `Method` to `"Distribution"`, `"KernelCosets"`, `"Brouwer"`, `"Zimmermann"`, or `"Quaternary"`.

Example H2E28

We compute the minimum Lee weight of four $\mathbb{Z}_2\mathbb{Z}_4$-additive codes by using different methods. We show that if the code is small, a brute force calculation can be faster than the

other methods. The Brouwer-Zimmermann method performs well for codes with a low minimum distance, such as extended perfect codes. When this method uses matrices with very large rank deficits, as with Hadamard codes, Brouwer's original method may perform better. The method based on the kernel is not as affected by a high minimum distance and works well in general, but it becomes slower when the number of cosets grows.

The verbose flag `IgnoreWeightAttributes` is set to level 1 in order to compare the computing time between the different available methods. At the end of the example, it is set back to level 0.

```
> SetVerbose("IgnoreWeightAttributes", 1);

> C := Z2Z4HadamardCode(3, 11);
> #C;
4096
> time Z2Z4MinimumLeeWeight(C : Method := "Distribution");
1024
Time: 1.400
> time Z2Z4MinimumLeeWeight(C : Method := "KernelCosets");
1024
Time: 1.700
> time Z2Z4MinimumLeeWeight(C : Method := "Brouwer");
1024
Time: 39.250
> time Z2Z4MinimumLeeWeight(C : Method := "Zimmermann");
1024
Time: 398.790
> time Z2Z4MinimumLeeWeight(C : Method := "Quaternary");
1024
Time: 216.820

> C := Z2Z4ExtendedPerfectCode(2, 5);
> #C;
67108864
> time Z2Z4MinimumLeeWeight(C : Method := "Distribution");
4
Time: 781.040
> time Z2Z4MinimumLeeWeight(C : Method := "KernelCosets");
4
Time: 0.090
> time Z2Z4MinimumLeeWeight(C : Method := "Brouwer");
4
Time: 0.060
> time Z2Z4MinimumLeeWeight(C : Method := "Zimmermann");
4
Time: 0.050
> time Z2Z4MinimumLeeWeight(C : Method := "Quaternary");
4
Time: 0.000
```

```
> C := Z2Z4ExtendedPerfectCode(2, 6);
> #C;
144115188075855872
> time Z2Z4MinimumLeeWeight(C : Method := "KernelCosets");
4
Time: 656.480
> time Z2Z4MinimumLeeWeight(C : Method := "Brouwer");
4
Time: 0.610
> time Z2Z4MinimumLeeWeight(C : Method := "Zimmermann");
4
Time: 0.500
> time Z2Z4MinimumLeeWeight(C : Method := "Quaternary");
4
Time: 2.440

> C := Z2Z4ReedMullerCode(2, 2, 6);
> #C;
4194304
> time Z2Z4MinimumLeeWeight(C : Method := "Distribution");
16
Time: 74.100
> time Z2Z4MinimumLeeWeight(C : Method := "KernelCosets");
16
Time: 0.090
> time Z2Z4MinimumLeeWeight(C : Method := "Brouwer");
16
Time: 13.900
> time Z2Z4MinimumLeeWeight(C : Method := "Zimmermann");
16
Time: 14.940
> time Z2Z4MinimumLeeWeight(C : Method := "Quaternary");
16
Time: 0.040
```

Then, we check some relations given by the functions related to the minimum Lee weight.

```
> C := Z2Z4HadamardCode(2, 5);
> Z2Z4MinimumLeeWeight(C) eq LeeWeight(MinimumWord(C), C`Alpha);
true
> MinimumWord(C) in MinimumWords(C);
true
> LeeWeightDistribution(C);
[ <0, 1>, <16, 62>, <32, 1> ]
> MinimumWords(C) eq MinimumWords(C : NumWords := 62);
true

> SetVerbose("IgnoreWeightAttributes", 0);
```

2.8.2 The Weight Distribution

`LeeWeightDistribution(C)`

`Method` MonStgElt *Default*: "Auto"

Determine the Lee weight distribution for the $\mathbb{Z}_2\mathbb{Z}_4$-additive 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 Lee weight w_i.

When the parameter `Method` is set to the default `"Auto"`, then the method is internally chosen. The user can specify which method they want using setting it to either `"Distribution"` or `"KernelCoset"`. In the first case, a brute force enumeration is used. In the second case, the representation of the code as the union of cosets of the kernel and the function `WeightDistribution` to compute the weight distribution of a coset of a binary linear code are used.

`DualLeeWeightDistribution(C)`

`Method` MonStgElt *Default*: "Auto"

The Lee weight distribution of the additive dual code of C (see `LeeWeightDistribution`).

`LeeWeightEnumerator(C)`

`Method` MonStgElt *Default*: "Auto"

Determine the Lee weight enumerator for the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C. The Lee weight enumerator of C is the polynomial $\sum_{v\in C}(X^{\alpha+2\beta-w_L(v)}Y^{w_L(v)})$, where $w_L(v)$ is the Lee weight of v. The result will lie in a global multivariate polynomial ring over $\mathbb{Z}$ with two variables. The angle-bracket notation may be used to assign names to the indeterminates.

When the parameter `Method` is set to the default `"Auto"`, then the method is internally chosen. The user can specify which method they want using setting it to either `"Distribution"` or `"KernelCoset"`. In the first case, to compute the Lee weight distribution a brute force enumeration is used. In the second case, the representation of the code as the union of cosets of the kernel and the function `WeightDistribution` to compute the weight distribution of a coset of a binary linear code are used.

`ExternalDistance(C)`

`Method` MonStgElt *Default*: "Auto"

Determine the external distance for the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C. The external distance of a code is the number of different non-zero Lee weights of the additive dual code of C.

When the parameter `Method` is set to the default `"Auto"`, then the method is internally chosen. The user can specify which method they want using setting it to either `"Distribution"` or `"KernelCoset"`. In the first case, to compute the Lee weight distribution a brute force enumeration is used. In the second case, the representation of the code as the union of cosets of the kernel and the function `WeightDistribution` to compute the weight distribution of a coset of a binary linear code are used.

Example H2E29

```
> C := Z2Z4HadamardCode(2, 5);
> LeeWeightDistribution(C);
[ <0, 1>, <16, 62>, <32, 1> ]
> Z2Z4MinimumLeeWeight(C) eq LeeWeightDistribution(C)[2][1];
true
> DualLeeWeightDistribution(C) eq LeeWeightDistribution(Dual(C));
true
> ExternalDistance(C) eq #DualLeeWeightDistribution(C)-1;
true
```

2.8.3 Covering Radius

`CoveringRadius(C)`

`MaximumTime` RngIntElt *Default*: ∞

The covering radius of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, which is the smallest radius ρ (considering the Lee distance) such that the spheres of radius ρ centered at all codewords of C cover the ambient space $V = \mathbb{Z}_2^\alpha \times \mathbb{Z}_4^\beta$. Note that this function is only applicable when C is small.

The parameter `MaximumTime` sets a time limit (in seconds of "user time") after which the calculation is aborted. The default value is infinity, when there is no restriction on time.

`CoveringRadiusBounds(C)`

Return a lower and upper bounds on the covering radius of the $\mathbb{Z}_2\mathbb{Z}_4$-additive code C. The lower bound is given by the maximum between the error correcting capability of C and the covering radius of the linear span of C_{bin}, where $C_{bin} = \Phi(C)$ and Φ is the Gray map defined in Subsection 2.5.1. The upper bound is given by the minimum between the external distance of C and the covering radius of $K_{bin} = \Phi(K_C)$, where K_C is the kernel of C as defined in Subsection 2.5.3. Note that this function is only applicable when C is small.

Example H2E30

```
> C1 := Z2Z4HadamardCode(2, 4);
> CoveringRadiusBounds(C1);
6 6
> CoveringRadius(C1);
6

> C2 := ExtendCode(Z2Z4ReedMullerCode(1, 2, 4));
> CoveringRadiusBounds(C2);
3 5
> CoveringRadius(C2);
3

> C3 := Z2Z4HadamardCode(2, 5);
> CoveringRadius(C3 : MaximumTime := 60);
7
> CoveringRadiusBounds(C3);
12 14
```

2.9 Constructing New Codes from Old

2.9.1 Construction of Subcodes

`OrderTwoSubcode(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, return the $\mathbb{Z}_2\mathbb{Z}_4$-additive subcode C_b which contains all order two codewords of C.

`Subcode(C, t1, t2)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ and two integers, $t1$ and $t2$, such that $1 \leq t1 \leq \gamma$ and $1 \leq t2 \leq \delta$, return a $\mathbb{Z}_2\mathbb{Z}_4$-additive subcode of C of type $(\alpha, \beta; t1, t2; \kappa')$, where $\kappa' \leq \kappa$. This $\mathbb{Z}_2\mathbb{Z}_4$-additive subcode is generated by the first $t1$ rows of order two and the first $t2$ rows of order four in the generator matrix given by the function `MinRowsGeneratorMatrix(C)`.

`Subcode(C, S1, S2)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ and two sets of integers, $S1$ and $S2$, such that each of their elements lies in the range $[1, \gamma]$ and $[1, \delta]$, respectively, return a $\mathbb{Z}_2\mathbb{Z}_4$-additive subcode of C of type $(\alpha, \beta; |S1|, |S2|; \kappa')$, where $\kappa' \leq \kappa$. This $\mathbb{Z}_2\mathbb{Z}_4$-additive subcode is generated by the rows of order two and four whose positions appear in $S1$ and $S2$, respectively, in the generator matrix given by the function `MinRowsGeneratorMatrix(C)`.

Example H2E31

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, we compute some subcodes of this code.

```
> C := RandomZ2Z4AdditiveCode(2, 4, 2, 3);
> C;
(6, 256) Z2Z4-additive code of type (2, 4; 2, 3; 1)
Generator matrix:
[2 0 0 0 0 0]
[0 2 0 0 0 1]
[0 0 1 0 1 0]
[0 0 0 1 1 1]
[0 0 0 0 2 0]
[0 0 0 0 0 2]
> Z2Z4Type(C);
[ 2, 4, 2, 3, 1 ]
> MinRowsGeneratorMatrix(C);
[2 0 0 0 0 0]
[0 0 0 0 2 0]
[0 2 0 0 0 1]
[0 2 0 1 1 0]
[0 0 1 0 1 0]
2 3

> C1 := Subcode(C, 2, 1);
> C1;
(6, 16) Z2Z4-additive code of type (2, 4; 2, 1; 1)
Generator matrix:
[2 0 0 0 0 0]
[0 2 0 0 0 1]
[0 0 0 0 2 0]
[0 0 0 0 0 2]
> C1 subset C;
true
> Z2Z4Type(C1);
[ 2, 4, 2, 1, 1 ]
> MinRowsGeneratorMatrix(C1);
[2 0 0 0 0 0]
[0 0 0 0 2 0]
```

```
[0 2 0 0 0 1]
2 1

> C2 := Subcode(C, {2}, {1,3});
> C2;
(6, 32) Z2Z4-additive code of type (2, 4; 1, 2; 0)
Generator matrix:
[0 2 0 0 0 1]
[0 0 1 0 1 0]
[0 0 0 0 2 0]
[0 0 0 0 0 2]
> MinRowsGeneratorMatrix(C2);
[0 0 0 0 2 0]
[0 2 0 0 0 1]
[0 0 1 0 1 0]
1 2
```

2.9.2 Sum and Intersection

`C + D`

The (vector space) sum of the $\mathbb{Z}_2\mathbb{Z}_4$-additive codes C and D, where C and D have the same parameters α and β, hence also the same length.

`C meet D`

The intersection of the $\mathbb{Z}_2\mathbb{Z}_4$-additive codes C and D, where C and D have the same parameters α and β, hence also the same length.

Example H2E32

We verify some simple results from the sum and intersection of $\mathbb{Z}_2\mathbb{Z}_4$-additive subcodes.

```
> R  := RSpace(IntegerRing(4), 4);
> C  := Z2Z4AdditiveCode([R![2,0,0,2], R![0,1,0,3], R![0,0,1,3]], 1);

> C1 := Subcode(C, 1, 0);
> C2 := Subcode(C, 0, 1);
> C3 := Subcode(C, 1, 1);

> (C1 + C2) eq C3;
true
> (C1 meet C3) eq C1;
true
```

2.9.3 Standard Constructions

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C, which is represented as a subspace of $\mathbb{Z}_4^{\alpha+\beta}$, any codeword u belonging to C can be written as $u = (u_1|u_2)$, where $u_1 \in \mathbb{Z}_4^{\alpha}$ and $u_2 \in \mathbb{Z}_4^{\beta}$.

`DirectSum(C, D)`

Given $\mathbb{Z}_2\mathbb{Z}_4$-additive codes C and D, construct the direct sum of C and D. The direct sum is a $\mathbb{Z}_2\mathbb{Z}_4$-additive code that consists of all vectors of the form $(u_1, v_1|u_2, v_2)$, where $(u_1|u_2) \in C$ and $(v_1|v_2) \in D$.

`Concatenation(C, D)`

Given $\mathbb{Z}_2\mathbb{Z}_4$-additive codes C and D, return the concatenation of C and D. If $G_c = (A_\alpha|A_\beta)$ and $G_d = (B_\alpha|B_\beta)$ are the generator matrices of C and D, respectively, the concatenation of C and D is the $\mathbb{Z}_2\mathbb{Z}_4$-additive code with generator matrix whose rows consist of each row $(a_\alpha|a_\beta)$ of G_c concatenated with each row $(b_\alpha|b_\beta)$ of G_d in the following way: $(a_\alpha, b_\alpha|a_\beta, b_\beta)$.

`PlotkinSum(C, D)`

Given $\mathbb{Z}_2\mathbb{Z}_4$-additive codes C and D both with the same parameters α and β, hence also the same length, construct the Plotkin sum of C and D. The Plotkin sum is a $\mathbb{Z}_2\mathbb{Z}_4$-additive code that consists of all vectors of the form $(u_\alpha, u_\alpha + v_\alpha|u_\beta, u_\beta + v_\beta)$, where $(u_\alpha|u_\beta) \in C$ and $(v_\alpha|v_\beta) \in D$.

`PunctureCode(C, i)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C and an integer i, $1 \leq i \leq \alpha + \beta$, construct a new $\mathbb{Z}_2\mathbb{Z}_4$-additive code C' by deleting the i-th coordinate from each codeword of C.

`PunctureCode(C, S)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C and a set S of distinct integers $\{i_1, \ldots, i_r\}$ each of which lies in the range $[1, \alpha + \beta]$, construct a new $\mathbb{Z}_2\mathbb{Z}_4$-additive code C' by deleting the components $i_1, \ldots, i_r$ from each codeword of C.

`ShortenCode(C, i)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C and an integer i, $1 \leq i \leq \alpha + \beta$, construct a new $\mathbb{Z}_2\mathbb{Z}_4$-additive 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.

`ShortenCode(C, S)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C and a set S of distinct integers $\{i_1, \ldots, i_r\}$ each of which lies in the range $[1, \alpha + \beta]$, construct a new $\mathbb{Z}_2\mathbb{Z}_4$-additive 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.

`ExtendCode(C)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C form a new $\mathbb{Z}_2\mathbb{Z}_4$-additive code C' from C by adding the appropriate extra binary coordinate to each codeword v of C such that $\Phi(v)$ has even Hamming weight, where Φ is the Gray map defined in Subsection 2.5.1.

Example H2E33

We combine $\mathbb{Z}_2\mathbb{Z}_4$-additive codes in diferent ways and look at the parameters α and β of the new $\mathbb{Z}_2\mathbb{Z}_4$-additive codes.

```
> C1 := RandomZ2Z4AdditiveCode(2, 3);
> C2 := RandomZ2Z4AdditiveCode(1, 4);
> Length(C1);
5 [ 2, 3 ]
> Length(C2);
5 [ 1, 4 ]

> C3 := DirectSum(C1, C2);
> Length(C3);
10 [ 3, 7 ]
> C4 := Concatenation(C1, C2);
> Length(C4);
10 [ 3, 7 ]
> C4 subset C3;
true

> C5 := PunctureCode(C1, 3);
> Length(C5);
4 [ 2, 2 ]
```

2.10 Decoding

This section describes functions for decoding vectors from the ambient space of a $\mathbb{Z}_2\mathbb{Z}_4$-additive code, or the corresponding space over $\mathbb{Z}_2$ under the Gray map, using two different algorithms: coset decoding and syndrome decoding. The reader is referred to [12, 26] for more information on coset decoding; and to [13, 16, 28] on syndrome decoding.

2.10.1 Coset Decoding

`CosetDecode(C, u : parameters)`

`MinWeightCode`	RngIntElt	*Default* : -
`MinWeightKernel`	RngIntElt	*Default* : -

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, and a vector u from the ambient space $V = \mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$ or $V_2 = \mathbb{Z}_2^{\alpha+2\beta}$, attempt to decode u with respect to C. The vector $u \in V$ can be given either as a vector in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as a tuple in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$. If the decoding algorithm succeeds in computing a vector $u' \in C$ as the decoded version of $u \in V$, then the function returns `true`, u' and $\Phi(u')$, where Φ is the Gray map. If the decoding algorithm does not succeed in decoding u, then the function returns `false`, the zero vector in V and the zero vector in V_2.

The coset decoding algorithm considers the binary linear code $C_u = C_{bin} \cup (C_{bin} + \Phi(u))$, when $C_{bin} = \Phi(C)$ is linear. On the other hand, when C_{bin} is nonlinear, we have $C_{bin} = \bigcup_{i=0}^{t}(K_{bin} + \Phi(c_i))$, where $K_{bin} = \Phi(K_C)$, K_C is the kernel of C as a $\mathbb{Z}_2\mathbb{Z}_4$-additive subcode, $[c_0, c_1, \ldots, c_t]$ are the coset representatives of C with respect to K_C (not necessarily of minimal weight in their cosets) and c_0 is the zero codeword. In this case, the algorithm considers the binary linear codes $K_0 = K_{bin} \cup (K_{bin} + \Phi(u))$, $K_1 = K_{bin} \cup (K_{bin} + \Phi(c_1) + \Phi(u))$, ..., $K_t = K_{bin} \cup (K_{bin} + \Phi(c_t) + \Phi(u))$.

If the parameter `MinWeightCode` is not assigned, then the minimum Lee weight of C, which coincides with the minimum weight of C_{bin}, denoted by d, is computed. Note that the minimum distance of C_{bin} coincides with its minimum weight. If C_{bin} is linear and the minimum weight of C_u is less than d, then $\Phi(u') = \Phi(u) + e$, where e is a word of minimum weight of C_u; otherwise, the decoding algorithm returns `false`. On the other hand, if C_{bin} is nonlinear and the minimum weight of $\cup_{i=0}^{t} K_i$ is less than the minimum weight of K_{bin}, then $\Phi(u') = \Phi(u) + e$, 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 Hamming weight of K_{bin} is computed.

`CosetDecode(C, Q : parameters)`

`MinWeightCode`	RngIntElt	*Default* : -
`MinWeightKernel`	RngIntElt	*Default* : -

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$ and a sequence Q of vectors from the ambient space $V = \mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$ or $V_2 = \mathbb{Z}_2^{\alpha+2\beta}$, attempt to decode the vectors of Q with respect to C. This function is similar to the function `Z2Z4CosetDecode(C, u)` except that rather than decoding a single vector, it decodes a sequence of vectors and returns a sequence of booleans and two sequences of decoded vectors corresponding to the given sequence. The algorithm used and effect of the parameters `MinWeightCode` and `MinWeightKernel` are identical to those for the function `Z2Z4CosetDecode(C, u)`.

Example H2E34

Starting with the $\mathbb{Z}_2\mathbb{Z}_4$-additive Hadamard code C of type $(8, 12; 2, 2; 2)$, a codeword $c \in C$ is selected and then perturbed to give a vector u in the ambient space of C. The vector u is then decoded to recover c.

```
> C := Z2Z4HadamardCode(2, 5);
> C;
(20, 64) Z2Z4-additive code of type (8, 12; 2, 2; 2)
Generator matrix:
[2 0 0 2 0 2 2 0 1 3 1 0 3 2 3 1 3 2 1 0]
[0 2 0 2 0 2 0 2 0 2 1 1 1 1 0 2 1 1 1 1]
[0 0 2 2 0 0 2 2 1 1 0 1 2 3 1 1 0 1 2 3]
[0 0 0 0 2 2 2 2 0 0 0 0 0 0 2 2 2 2 2 2]
[0 0 0 0 0 0 0 0 2 2 0 2 0 2 2 2 0 2 0 2]
[0 0 0 0 0 0 0 0 0 0 2 2 2 2 0 0 2 2 2 2]

> alpha := C`Alpha;
> beta := Length(C) - alpha;
> d := Z2Z4MinimumLeeDistance(C);
> t := Floor((d-1)/2);
> t;
7

> c := C![0,0,2,2,0,0,2,2,1,1,0,1,2,3,1,1,0,1,2,3];
> c in C;
true
> u := c;
> u[5] := u[5] + 2;
> u[12] := u[12] + 1;
> u[13] := u[13] + 3;
> u[16] := u[16] + 2;
> u;
(0 0 2 2 2 0 2 2 1 1 0 2 1 3 1 3 0 1 2 3)
> grayMap := GrayMap(Z2Z4AdditiveUniverseCode(alpha, beta));
> grayMap(c-u);
(0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0)

> isDecoded, uDecoded := CosetDecode(C, u : MinWeightCode := d);
> isDecoded;
true
> uDecoded eq c;
true
```

2.10.2 Syndrome Decoding

`SyndromeDecode(C, u)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, and a vector u from the ambient space $V = \mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$ or $V_2 = \mathbb{Z}_2^{\alpha+2\beta}$, attempt to decode u with respect to C. The vector $u \in V$ can be given either as a vector in $\mathbb{Z}_4^{\alpha+\beta}$, where the ones in the first α coordinates are represented by twos; or as a tuple in the cartesian product set $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$.

The decoding algorithm always succeeds in computing a vector $u' \in C$ as the decoded version of $u \in V$, and the function returns `true`, u' and $\Phi(u')$, where Φ is the Gray map. Although the function never returns `false`, the first output parameter `true` is given to be consistent with the other decoding functions.

The syndrome decoding algorithm consists of computing a table pairing each possible syndrome s with a vector of minimum Lee weight e_s, called coset leader, in the coset of C containing all vectors having syndrome s. After receiving a vector u, its syndrome s is computed using the parity check matrix. Then, u is decoded into the codeword $c = u - e_s$.

`SyndromeDecode(C, Q)`

Given a $\mathbb{Z}_2\mathbb{Z}_4$-additive code C of type $(\alpha, \beta; \gamma, \delta; \kappa)$, and a sequence Q of vectors from the ambient space $V = \mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$ or $V_2 = \mathbb{Z}_2^{\alpha+2\beta}$, attempt to decode the vectors of Q with respect to C. This function is similar to the function `Z2Z4SyndromeDecode(C, u)` except that rather than decoding a single vector, it decodes a sequence of vectors and returns a sequence of booleans and two sequences of decoded vectors corresponding to the given sequence. The algorithm used is the same as that of function `Z2Z4SyndromeDecode(C, u)`.

Example H2E35

The $\mathbb{Z}_2\mathbb{Z}_4$-additive Hadamard code C of type $(4, 6; 1, 2; 1)$ is constructed. Next, information bits are encoded using C and three errors are introduced to give the vector u. Then u is decoded by calculating its syndrome and applying the map, given by the `Z2Z4CosetLeaders` function, to the syndrome to recover the original vector.

```
> C := Z2Z4HadamardCode(2, 4);
> C;
(10, 32) Z2Z4-additive code of type (4, 6; 1, 2; 1)
Generator matrix:
[2 0 0 2 1 3 1 0 3 2]
[0 2 0 2 0 2 1 1 1 1]
[0 0 2 2 1 1 0 1 2 3]
[0 0 0 0 2 2 0 2 0 2]
[0 0 0 0 0 0 2 2 2 2]

> alpha := C`Alpha;
```

```
> beta := Length(C) - alpha;
> t := Floor((Z2Z4MinimumLeeDistance(C)-1)/2);
> t;
3

> V4, V2, f, fbin := InformationSpace(C);
> i := V4![2,1,0];
> c := f(i);
> c;
(2 2 0 0 1 1 0 3 2 1)
> u := c;
> u[2] := u[2] + 2;
> u[7] := u[7] + 3;
> u[9] := u[9] + 1;
> u;
(2 0 0 0 1 1 3 3 3 1)
> grayMap := GrayMap(Z2Z4AdditiveUniverseCode(alpha, beta));
> grayMap(c-u);
(0 1 0 0 0 0 0 0 1 0 0 1 0 0 0)

> isDecoded, uDecoded := SyndromeDecode(C, u);
> isDecoded;
true
> uDecoded eq c;
true

> L, mapCosetLeaders := CosetLeaders(C);
> errorVector := mapCosetLeaders(Syndrome(u, C));
> errorVector;
(0 2 0 0 0 0 3 0 1 0)
> u-errorVector eq c;
true
```

Bibliography

[1] T. Abualrub, I. Siap, H. Aydin, "$\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic codes," *IEEE Trans. on Information Theory*, vol. 60, no. 3, pp. 1508-1514, 2014.

[2] A. Betten, M. Braun, H. Fripertinger, A. Kerber, A. Kohnert, A. Wassermann, *Error-Correcting Linear Codes: Classification by Isometry and Applications*, Algorithms and Computation in Mathematics, vol. 18, Springer, 2006.

[3] J. Borges, C. Fernández-Córdoba, R. Ten-Valls, "$\mathbb{Z}_2\mathbb{Z}_4$-additive cyclic codes, generator polynomials and dual codes," *IEEE Trans. on Information Theory*, vol. 62, no. 11, pp. 6348-6354, 2016.

[4] J. Borges, J. Rifà, "A characterization of 1-perfect additive codes," *IEEE Trans. on Information Theory*, vol. 45, no. 5, pp. 1688-1697, 1999.

[5] J. Borges, K. T. Phelps, J. Rifà, "The rank and kernel of extended 1-perfect $\mathbb{Z}_4$-linear and additive non-$\mathbb{Z}_4$-linear codes," *IEEE Trans. on Information Theory*, vol. 49, no. 8, pp. 2028-2034, 2003.

[6] J. Borges, C. Fernández-Córdoba, J. Pujol, J. Rifà, 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.

[7] J. Borges, C. Fernández-Córdoba, J. Pujol, J. Rifà, M. Villanueva, "Survey on $\mathbb{Z}_2\mathbb{Z}_4$-additive codes," In *Contact Forum Galois Geometries and Applications 2012*, Proc. of the Royal Flemish Academy of Belgium for Science and the Arts (October 5), pp. 19–67, 2014.

[8] J. Borges, C. Fernández-Córdoba, J. Pujol, J. Rifà, M. Villanueva, "$\mathbb{Z}_2\mathbb{Z}_4$-Linear Codes," Springer, 2022.

[9] J. J. Cannon, W. Bosma, C. Fieker, A. Steel (Eds.) *Handbook of* Magma *Functions*, Edition 2.26-4, 6347 pages, 2021.

[10] P. Delsarte, "An algebraic approach to the association schemes of coding theory," *Philips Res. Rep. Suppl.*, vol. 10, no. 2, pp. 1-97, 1973.

[11] P. Delsarte, V. Levenshtein, "Association schemes and coding theory," *IEEE Trans. on Information Theory*, vol. 44, no. 6, pp. 2477–2504, 1998.

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

[13] A. R. Hammons, P. V. Kumar, A. R. Calderbank, N. J. A. Sloane, P. Solé, "The $\mathbb{Z}_4$-linearity of Kerdock, Preparata, Goethals and related codes," *IEEE Trans. on Information Theory*, vol. 40, no. 2, pp. 301-319, 1994.

[14] J. A. Howell, "Spans in the module $\mathbb{Z}_m^s$," *Linear and Multilinear Algebra*, vol. 19, no. 1, pp. 67-77, 1986.

[15] D. S. Krotov, "$\mathbb{Z}_4$-linear Hadamard and extended perfect codes," in *Proc. of the International Workshop on Coding and Cryptography*, Paris (France), Jan. 8-12, pp. 329–334, 2001.

[16] F. I. MacWilliams, N. J. Sloane, *The Theory of Error-Correcting Codes*, North-Holland, New York, 1977.

[17] J. Pernas, J. Pujol, M. Villanueva, "Classification of some families of quaternary Reed-Muller codes," *IEEE Trans. on Information Theory*, vol. 57, no. 9, pp. 6043-6051, 2011.

[18] K. T. Phelps, J. Rifà, "On binary 1-perfect additive codes: some structural properties," *IEEE Trans. on Information Theory*, vol. 48, no. 9, pp. 2587-2592, 2002.

[19] K. T. Phelps, J. Rifà, M. Villanueva, "On the additive ($\mathbb{Z}_4$-linear and non-$\mathbb{Z}_4$-linear) Hadamard codes: rank and kernel," *IEEE Trans. on Information Theory*, vol. 52, no. 1, pp. 316-319, 2006.

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

[21] J. Pujol, J. Rifà, 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.

[22] J. Rifà, J. M. Basart, L. Huguet, "On completely regular propelinear codes," in *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, AAECC. Lecture Notes in Computer Science*, vol 357, pp. 341-355, 1988.

[23] J. Rifà, J. Pujol, "Translation invariant propelinear codes," *IEEE Trans. on Information Theory*, vol. 43, no. 2, pp. 590-598, 1997.

[24] F. I. Solov'eva "On $\mathbb{Z}_4$-linear codes with parameters of Reed-Muller codes," *Problems of Information Transmission*, vol. 43, no. 1, pp. 26-32, 2007.

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

[26] M. Villanueva, F. Zeng, J. Pujol, "Efficient representation of binary nonlinear codes: constructions and minimum distance computation," *Designs, Codes and Cryptography*, vol. 76, no. 1, pp. 3-21, 2015.

[27] G. White, *Enumeration-Based Algorithms in Linear Coding Theory*, PhD Thesis, University of Sydney, 2006.

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