Handbook of Magma Functions

Volume 13

Coding Theory and Cryptography

John Cannon Wieb Bosma
Claus Fieker Allan Steel

Editors

Version 2.22

Sydney

June 9, 2016

MAGMA
COMPUTER ● ALGEBRA

Handbook of Magma Functions

Editors:

John Cannon *Wieb Bosma* *Claus Fieker* *Allan Steel*

Handbook Contributors:

Geoff Bailey, Wieb Bosma, Gavin Brown, Nils Bruin, John Cannon, Jon Carlson, Scott Contini, Bruce Cox, Brendan Creutz, Steve Donnelly, Tim Dokchitser, Willem de Graaf, Andreas-Stephan Elsenhans, Claus Fieker, Damien Fisher, Volker Gebhardt, Sergei Haller, Michael Harrison, Florian Hess, Derek Holt, David Howden, Al Kasprzyk, Markus Kirschmer, David Kohel, Axel Kohnert, Dimitri Leemans, Paulette Lieby, Graham Matthews, Scott Murray, Eamonn O'Brien, Dan Roozemond, Ben Smith, Bernd Souvignier, William Stein, Allan Steel, Damien Stehlé, Nicole Sutherland, Don Taylor, Bill Unger, Alexa van der Waall, Paul van Wamelen, Helena Verrill, John Voight, Mark Watkins, Greg White

Production Editors:

Wieb Bosma *Claus Fieker* *Allan Steel* *Nicole Sutherland*

HTML Production:

Claus Fieker *Allan Steel*

VOLUME 13: OVERVIEW

VOLUME 13: CONTENTS

PART XXI
CODING THEORY

158 LINEAR CODES OVER FINITE FIELDS

Chapter 158
LINEAR CODES OVER FINITE FIELDS

158.1 Introduction

Let K be a finite field and let V be the vector space of n-tuples over K. The *Hamming-distance* between elements x and y of V, denoted $d(x, y)$, is defined by

$$d(x, y) := \#\{ \ 1 \le i \le n \mid x_i \ne y_i \ \}.$$

The *minimum distance* d for a subset C of V is then

$$d = \min\{ \ d(x, y) \mid x \in C, y \in C, x \ne y \ \}.$$

The subset C of V is called an (n, M, d) *code* if the minimum distance for the subset C is d and $|C| = M$. Then V is referred to as the *ambient space* of C.

The code C is called a $[n, k, d]$ *linear code* if C is a k-dimensional subspace of V. Currently Magma supports not only linear codes, but also codes over finite fields which are only linear over some subfield. These are known as *additive codes* and can be found in Chapter 163. This chapter deals only with linear codes.

In this chapter, the term "code" will refer to a linear code. Magma provides machinery for studying linear codes over finite fields $F_q = GF(q)$, over the integer residue classes $\mathbf{Z}_m = \mathbf{Z}/m\mathbf{Z}$, and over galois rings $GR(p^n, k)$.

This chapter describes those functions which are applicable to codes over F_q. The highlights of the facilities provided for such codes include:

- The construction of codes in terms of generator matrices, parity check matrices and generating polynomials (cyclic codes).
- A large number of constructions for particular families of codes, e.g., quadratic residue codes.
- Highly optimized algorithms for the calculation of the minimum weight.
- Various forms of weight enumerator including the Macwilliams transform.
- A database that gives the user access to every best known linear code over $GF(2)$ of length up to 256, and 98% of best known linear codes over $GF(4)$ of length up to 100.
- Machinery that allows the user to construct algebraic-geometric codes from a curve defined over F_q.
- The computation of automorphism groups for codes over small fields.

The reader is referred to [MS78] as a general reference on coding theory.

158.2 Construction of Codes

158.2.1 Construction of General Linear Codes

```
LinearCode< R, n | L >
```

Create a code as a subspace of $V = R^{(n)}$ which is generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) An element of V.

(b) A set or sequence of elements of V.

(c) A sequence of n elements of K, defining an element of V.

(d) A set or sequence of sequences of type (c).

(e) A subspace of V.

(f) A set or sequence of subspaces of V.

Example H158E1

We define the ternary Golay code as a six-dimensional subspace of the vector space $K^{(11)}$, where K is $\mathbf{F}_3$. The ternary Golay code could be defined in a single statement as follows:

```
> K := FiniteField(3);
> C := LinearCode<K, 11 |
>     [1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1], [0, 1, 0, 0, 0, 0, 0, 1, 2, 2, 1],
>     [0, 0, 1, 0, 0, 0, 1, 0, 1, 2, 2], [0, 0, 0, 1, 0, 0, 2, 1, 0, 1, 2],
>     [0, 0, 0, 0, 1, 0, 2, 2, 1, 0, 1], [0, 0, 0, 0, 0, 1, 1, 2, 2, 1, 0]>;
```

Alternatively, if we want to see the code as a subspace of $K^{(11)}$, we would proceed as follows:

```
> K := FiniteField(3);
> K11 := VectorSpace(K, 11);
> C := LinearCode(sub<K11 |
>     [1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1], [0, 1, 0, 0, 0, 0, 0, 1, 2, 2, 1],
>     [0, 0, 1, 0, 0, 0, 1, 0, 1, 2, 2], [0, 0, 0, 1, 0, 0, 2, 1, 0, 1, 2],
>     [0, 0, 0, 0, 1, 0, 2, 2, 1, 0, 1], [0, 0, 0, 0, 0, 1, 1, 2, 2, 1, 0]>);
```

```
LinearCode(U)
```

Let V be the R-space $R^{(n)}$ and suppose that U is a subspace of V. The effect of this function is to define the linear code C corresponding to the subspace U. Suppose the code C being constructed has dimension k. The evaluation of this constructor results in the creation of the following objects:

(a) The generator matrix G for C, created as a $k \times n$ matrix belonging to the R-matrix space, $R^{(k \times n)}$.

(b) The parity check matrix H for C, created as an $(n-k) \times n$ matrix belonging to the R-matrix space, $R^{(n-k) \times n)}$.

```
LinearCode(A)
```

Given a $k \times n$ matrix A over the ring R, construct the linear code generated by the rows of A. Note that it is not assumed that the rank of A is k. The effect of this constructor is otherwise identical to that described above.

Example H158E2

We define a code by constructing a matrix in a K-matrix space and using its row space to generate the code:

```
> M := KMatrixSpace(FiniteField(5), 2, 4);
> G := M ! [1,1,1,2, 3,2,1,4];
> G;
[1 1 1 2]
[3 2 1 4]
> C := LinearCode(G);
> C;
[4, 2, 2] Linear Code over GF(5)
Generator matrix:
[1 0 4 0]
[0 1 2 2]
```

```
PermutationCode(u, G)
```

Given a finite permutation group G of degree n, and a vector u belonging to the n-dimensional vector space V over the ring R, construct the code C corresponding to the subspace of V spanned by the set of vectors obtained by applying the permutations of G to the vector u.

Example H158E3

We define G to be a permutation group of degree 7 and construct the code C as the F_2-code generated by applying the permutations of G to a certain vector:

```
> G := PSL(3, 2);
> G;
Permutation group G of degree 7
    (1, 4)(6, 7)
    (1, 3, 2)(4, 7, 5)
> V := VectorSpace(GF(2), 7);
> u := V ! [1, 0, 0, 1, 0, 1, 1];
> C := PermutationCode(u, G);
> C;
[7, 3, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0 1 1]
[0 1 0 1 1 1 0]
[0 0 1 0 1 1 1]
```

158.2.2 Some Trivial Linear Codes

```
ZeroCode(R, n)
```

Given a ring R and positive integer n, return the $[n, 0, n]$ code consisting of only the zero code word, (where the minimum weight is by convention equal to n).

```
RepetitionCode(R, n)
```

Given a ring R and positive integer n, return the $[n, 1, n]$ code over K generated by the all-ones vector.

```
ZeroSumCode(R, n)
```

Given a ring R and positive integer n, return the $[n, n-1, 2]$ code over R such that for all codewords $(c_1, c_2, \ldots, c_n)$ we have $\sum_i c_i = 0$.

```
UniverseCode(R, n)
```

Given a ring R and positive integer n, return the $[n, n, 1]$ code consisting of all possible codewords.

```
EvenWeightCode(n)
```

Given a positive integer n, return the $[n, n-1, 2]$ code over $\mathbf{F}_2$ such that all vectors have even weight. This is equivalent to the zero sum code over $\mathbf{F}_2$.

```
EvenWeightSubcode(C)
```

Given a linear code C over $\mathbf{F}_2$, return the subcode of C containing the vectors of even weight.

```
RandomLinearCode(K, n, k)
```

Given a finite field K and positive integers n and k, such that $0 < k \leq n$, the function returns a random linear code of length n and dimension k over the field K. The method employed is to successively choose random vectors from $K^{(n)}$ until generators for a k-dimensional subspace have been found.

```
CordaroWagnerCode(n)
```

Construct the Cordaro–Wagner code of length n, This is the 2-dimensional repetition code over $\mathbf{F}_2$ of length n and having the largest possible minimum weight.

Example H158E4

Over any specific finite field K, the zero code of length n is contained in every code of length n, and similarly every code of length n is contained in the universe code of length n. This is illustrated over $GF(2)$ for length 6 codes with an arbitrary code of length 6 dimension 3.

```
> K := GF(2);
> U := UniverseCode(K, 6);
> U;
[6, 6, 1] Linear Code over GF(2)
> Z := ZeroCode(K, 6);
> Z;
[6, 0, 6] Linear Code over GF(2)
> R := RandomLinearCode(K, 6, 3);
> (Z subset R) and (R subset U);
true
```

158.2.3 Some Basic Families of Codes

In this section we describe how to construct three very important families of codes: cyclic codes, Hamming codes and Reed-Muller codes. We choose to present these very important families at this stage since they are easily understood and they give us a nice collection of codes for use in examples.

Many more constructions will be described in subsequent sections. In particular, variations and generalizations of the cyclic code construction presented here will be given.

```
CyclicCode(n, g)
```

Let K be a finite field. Given a positive integer n and a univariate polynomial $g(x) \in K[x]$ of degree $n - k$ such that $g(x) \mid x^n - 1$, construct the $[n, k]$ cyclic code generated by $g(x)$.

Example H158E5

We construct the length 23 Golay code over $GF(2)$ as a cyclic code by factorizing the polynomial $x^{23} - 1$ over $GF(2)$ and constructing the cyclic code generated by one of the factors of degree 11.

```
> P<x> := PolynomialRing(FiniteField(2));
> F := Factorization(x^23 - 1);
> F;
[
    <x + 1, 1>,
    <x^11 + x^9 + x^7 + x^6 + x^5 + x + 1, 1>,
    <x^11 + x^10 + x^6 + x^5 + x^4 + x^2 + 1, 1>
]
> CyclicCode(23, F[2][1]);
[23, 12, 7] Cyclic Code over GF(2)
Generator matrix:
```

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

`HammingCode(K, r)`

Given a positive integer r, and a finite field K of cardinality q, construct the r-th order Hamming code over K of cardinality q. This code has length

$$n = (q^r - 1)/(q - 1).$$

Example H158E6

We construct the third order Hamming code over GF(2) together with its parity check matrix.

```
> H := HammingCode(FiniteField(2), 3);
> H;
[7, 4, 3] Hamming code (r = 3) over GF(2)
Generator matrix:
[1 0 0 0 0 1 1]
[0 1 0 0 1 1 0]
[0 0 1 0 1 0 1]
[0 0 0 1 1 1 1]
> ParityCheckMatrix(H);
[1 0 1 0 1 1 0]
[0 1 1 0 0 1 1]
[0 0 0 1 1 1 1]
```

`SimplexCode(r)`

Given a positive integer r, construct the $[2^r - 1, r, 2^{r-1}]$ binary simplex code, which is the dual of a Hamming code.

`ReedMullerCode(r, m)`

Given positive integers r and m, where $0 \leq r \leq m$, construct the r-th order binary Reed–Muller code of length $n = 2^m$.

Example H158E7

We construct the first order Reed–Muller code of length 16 and count the number of pairs of vectors whose components are orthogonal.

```
> R := ReedMullerCode(1, 4);
> R;
[16, 5, 8] Reed-Muller Code (r = 1, m = 4) over GF(2)
Generator matrix:
[1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1]
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1]
[0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
> #{<v, w>: v, w in R | IsZero(InnerProduct(v, w))};
1024
```

158.3 Invariants of a Code

158.3.1 Basic Numerical Invariants

`Length(C)`

Given an $[n, k]$ code C, return the block length n of C.

`Dimension(C)`

`NumberOfGenerators(C)`

The dimension k of the $[n, k]$ linear code C.

`#C`

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

`InformationRate(C)`

The information rate of the $[n, k]$ code C. This is the ratio k/n.

158.3.2 The Ambient Space and Alphabet

```
AmbientSpace(C)
```

The ambient space of the code C, i.e. the generic R-space V in which C is contained.

```
RSpace(C)
```

```
VectorSpace(C)
```

Given an $[n, k]$ linear code C, defined as a subspace U of the n-dimensional space V, return U as a subspace of V with basis corresponding to the rows of the generator matrix for C.

```
Generic(C)
```

Given an $[n, k]$ code C, return the generic $[n, n, 1]$ code in which C is contained.

```
Alphabet(C)
```

```
Field(C)
```

The underlying ring (or alphabet) R of the code C.

158.3.3 The Code Space

```
GeneratorMatrix(C)
```

```
BasisMatrix(C)
```

The generator matrix for the linear code C, returned as an element of $\mathrm{Hom}(U, V)$ where U is the information space of C and V is the ambient space of C.

```
Basis(C)
```

The current vector space basis for the linear code C, returned as a sequence of elements of C.

```
Generators(C)
```

The current vector space basis for the linear code C, returned as a set of elements of C.

```
C . i
```

Given an $[n, k]$ code C and a positive integer i, $1 \leq i \leq k$, return the i-th element of the current basis of C.

158.3.4 The Dual Space

`Dual(C)`

The code that is dual to the code C.

`ParityCheckMatrix(C)`

The parity check matrix for the code C, returned as an element of $\mathrm{Hom}(V, U)$.

Example H158E8

We create a Reed–Muller code and demonstrate some simple relations.

```
> R := ReedMullerCode(1, 3);
> R;
[8, 4, 4] Reed-Muller Code (r = 1, m = 3) over GF(2)
Generator matrix:
[1 0 0 1 0 1 1 0]
[0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 1 1]
[0 0 0 0 1 1 1 1]
> G := GeneratorMatrix(R);
> P := ParityCheckMatrix(R);
> P;
[1 0 0 1 0 1 1 0]
[0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 1 1]
[0 0 0 0 1 1 1 1]
> G * Transpose(P);
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
> D := LinearCode(P);
> Dual(R) eq D;
true
```

`Hull(C)`

The Hull of a code is the intersection between itself and its dual.

158.3.5 The Information Space and Information Sets

`InformationSpace(C)`

Given an $[n, k]$ linear code C, return the k-dimensional R-space U which is the space of information vectors for the code C.

`InformationSet(C)`

Given an $[n, k]$ linear code C over a finite field, return the current information set for C. The information set for C is an ordered set of k linearly independent columns of the generator matrix, such that the generator matrix is the identity matrix when restricted to these columns. The information set is returned as a sequence of k integers, giving the numbers of the columns that correspond to the information set.

`AllInformationSets(C)`

Given an $[n, k]$ linear code C over a finite field, return all the possible information sets of C as a (sorted) sequence of sequences of column indices. Each inner sequence contains a maximal set of indices of linearly independent columns in the generator matrix of C.

`StandardForm(C)`

Given an $[n, k]$ linear code C over a finite field, return the standard form D of C. A code is in *standard form* if the first k components of the code words correspond to the information set. Magma returns one of the many codes in standard form which is isomorphic to C. (The same code is returned each time.) Thus, the effect of this function is to return a code D whose generators come from the generator matrix of C with its columns permuted, so that the submatrix consisting of the first k columns of the generator matrix for D is the identity matrix. Two values are returned:

(a) The standard form code D;

(b) An isomorphism from C to D.

Example H158E9

We construct a Reed–Muller code C and its standard form S and then map a codeword of C into S.

```
> C := ReedMullerCode(1, 4);
> C;
[16, 5, 8] Reed-Muller Code (r = 1, m = 4) over GF(2)
Generator matrix:
[1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1]
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1]
[0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
> InformationSet(C);
[ 1, 2, 3, 5, 9 ]
```

```
> #AllInformationSets(C);
2688
> S, f := StandardForm(C);
> S;
[16, 5, 8] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 1 1 1 0 1 1 0 1 0 0 1]
[0 1 0 0 0 1 1 0 1 1 0 1 0 1 0 1]
[0 0 1 0 0 1 0 1 1 0 1 1 0 0 1 1]
[0 0 0 1 0 0 1 1 1 0 0 0 1 1 1 1]
[0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1]
> u := C.1;
> u;
(1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1)
> f(u);
(1 0 0 0 0 1 1 1 0 1 1 0 1 0 0 1)
```

158.3.6 The Syndrome Space

```
SyndromeSpace(C)
```

Given an $[n, k]$ linear code C, return the $(n-k)$-dimensional vector space W, which is the space of syndrome vectors for the code C.

158.3.7 The Generator Polynomial

The operations in this section are restricted to cyclic codes.

```
GeneratorPolynomial(C)
```

Given a cyclic code C over a finite field, return the generator polynomial of C. The generator polynomial of C is a divisor of $x^n - 1$, where n is the length of C.

```
CheckPolynomial(C)
```

Given a cyclic code C over a finite field, return the check polynomial of C as an element of $K[x]$. If $g(x)$ is the generator polynomial of C and $h(x)$ is the check polynomial of C, then $g(x)h(x) = 0 \pmod{x^n - 1}$, where n is the length of C.

```
Idempotent(C)
```

Given a cyclic code C, return the (polynomial) idempotent of C. If $c(x)$ is the idempotent of C, then $c(x)^2 = 0 \pmod{x^n - 1}$, where n is the length of C.

Example H158E10

We find the generator and check polynomials for the third order Hamming code over GF(2).

```
> K<w> := GF(2);
> P<x> := PolynomialRing(K);
> H := HammingCode(K, 3);
> g := GeneratorPolynomial(H);
> g;
x^3 + x + 1
> h := CheckPolynomial(H);
> h;
x^4 + x^2 + x + 1
> g*h mod (x^7 - 1);
0
> forall{ c : c in H | h * P!Eltseq(c) mod (x^7-1) eq 0 };
true
> e := Idempotent(H);
> e;
x^4 + x^2 + x
> e^2;
x^8 + x^4 + x^2
```

158.4 Operations on Codewords

158.4.1 Construction of a Codeword

`C ! [a_1, ..., a_n]`

`elt< C | a_1, ..., a_n >`

Given a code C which is defined as a subset of the R-space $R^{(n)}$, and elements $a_1, \ldots, a_n$ belonging to R, construct the codeword $(a_1, \ldots, a_n)$ of C. It is checked that the vector $(a_1, \ldots, a_n)$ is an element of C.

`C ! u`

Given a code C which is defined as a subset of the R-space $V = R^{(n)}$, and an element u belonging to V, create the codeword of C corresponding to u. The function will fail if u does not belong to C.

`C ! 0`

The zero word of the code C.

`Random(C)`

A random codeword of C.

158.4.2 Arithmetic Operations on Codewords

```
u + v
```

Sum of the codewords u and v, where u and v belong to the same linear code C.

```
-u
```

Additive inverse of the codeword u belonging to the linear code C.

```
u - v
```

Difference of the codewords u and v, where u and v belong to the same linear code C.

```
a * u
```

Given an element a belonging to the field K, and a codeword u belonging to the linear code C, return the codeword $a * u$.

```
Normalize(u)
```

Given an element u over a field, not the zero element, belonging to the linear code C, return $\frac{1}{a} * u$, where a is the first non-zero component of u. If u is the zero vector, it is returned as the value of this function. The net effect is that `Normalize(u)` always returns a vector v in the subspace generated by u, such that the first non-zero component of v is the unit of K.

```
Syndrome(w, C)
```

Given an $[n, k]$ linear code C over a finite field with parent vector space V, and a vector w belonging to V, construct the syndrome of w relative to the code C. This will be an element of the *syndrome space* of C.

158.4.3 Distance and Weight

```
Distance(u, v)
```

The Hamming distance between the codewords u and v, where u and v belong to the same code C.

```
Weight(u)
```

The Hamming weight of the codeword u, i.e., the number of non-zero components of u.

```
LeeWeight(u)
```

The Lee weight of the codeword u.

Example H158E11

We calculate all possible distances between code words of the non-extended Golay code over GF(3), and show the correspondence with all possible code word weights.

```
> C := GolayCode(GF(3),false);
> {Distance(v,w):v,w in C};
{ 0, 5, 6, 8, 9, 11 }
> {Weight(v):v in C};
{ 0, 5, 6, 8, 9, 11 }
```

158.4.4 Vector Space and Related Operations

`(u, v)`

`InnerProduct(u, v)`

Inner product of the vectors u and v with respect to the Euclidean norm, where u and v belong to the parent vector space of the code C.

`Support(w)`

Given a word w belonging to the $[n, k]$ code C, return its support as a subset of the integer set $\{1..n\}$. The support of w consists of the coordinates at which w has non-zero entries.

`Coordinates(C, u)`

Given an $[n, k]$ linear 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 the alphabet of C so that $u = a_1 * C.1 + \ldots + a_k * C.k$.

`Parent(w)`

Given a word w belonging to the code C, return the ambient space V of C.

`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.

`Rotate(~u, k)`

Given a vector u, destructively rotate u by k coordinate positions.

`Trace(u, S)`

`Trace(u)`

Given a vector u with components in K, and a subfield S of K, construct the vector with components in S obtained from u by taking the trace of each component with respect to S. If S is omitted, it is taken to be the prime field of K.

Example H158E12

We create a specific code word in the length 5 even weight code, after a failed attempt to create a code word of odd weight. We then display its support, find its coordinates with respect to the basis and then confirm it by way of re-construction.

```
> C := EvenWeightCode(5);
>  C![1,1,0,1,0];
>> C![1,1,0,1,0];
    ^
Runtime error in '!': Result is not in the given structure
> c := C![1,1,0,1,1];
> c;
(1 1 0 1 1)
> Support(c);
{ 1, 2, 4, 5 }
> Coordinates(C,c);
[ 1, 1, 0, 1 ]
> C.1 + C.2 + C.4;
(1 1 0 1 1)
```

158.4.5 Predicates for Codewords

`u eq v`

The function returns `true` if and only if the codewords u and v are equal.

`u ne v`

The function returns `true` if and only if the codewords u and v are not equal.

`IsZero(u)`

The function returns `true` if and only if the codeword u is the zero vector.

158.4.6 Accessing Components of a Codeword

`u[i]`

Given a codeword u belonging to the code C defined over the ring R, return the i-th component of u (as an element of R).

`u[i] := x;`

Given an element u belonging to a subcode C of the full R-space $V = R^n$, a positive integer i, $1 \leq i \leq n$, and an element x of R, this function returns a vector in V which is u with its i-th component redefined to be x.

158.5 Coset Leaders

```
CosetLeaders(C)
```

Given a code C with ambient space V over a finite field, return a set of coset leaders (vectors of minimal weight in their cosets) for C in V as an indexed set of vectors from V. Note that this function is only applicable when V and C are small. This function also returns a map from the syndrome space of C into the coset leaders (mapping a syndrome into its corresponding coset leader).

Example H158E13

We construct a Hamming code C, encode an information word using C, introduce one error, and then decode by calculating the syndrome of the "received" vector and applying the CosetLeaders map to the syndrome to recover the original vector.
First we set C to be the third order Hamming Code over the finite field with two elements.

```
> C := HammingCode(GF(2), 3);
> C;
[7, 4, 3] Hamming code (r = 3) over GF(2)
Generator matrix:
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[0 0 1 0 1 1 0]
[0 0 0 1 1 1 1]
```

Then we set L to be the set of coset leaders of C in its ambient space V and f to be the map which maps the syndrome of a vector in V to its coset leader in L.

```
> L, f := CosetLeaders(C);
> L;
{@
    (0 0 0 0 0 0 0),
    (1 0 0 0 0 0 0),
    (0 1 0 0 0 0 0),
    (0 0 1 0 0 0 0),
    (0 0 0 1 0 0 0),
    (0 0 0 0 1 0 0),
    (0 0 0 0 0 1 0),
    (0 0 0 0 0 0 1)
@}
```

Since C has dimension 4, the degree of the information space I of C is 4. We set i to be an "information vector" of length 4 in I, and then encode i using C by setting w to be the product of i by the generator matrix of C.

```
> I := InformationSpace(C);
> I;
Full Vector space of degree 4 over GF(2)
> i := I ! [1, 0, 1, 1];
> w := i * GeneratorMatrix(C);
> w;
(1 0 1 1 0 1 0)
```

Now we set r to be the same as w but with an error in the 7-th coordinate (so r is the "received vector").

```
> r := w;
> r[7] := 1;
> r;
(1 0 1 1 0 1 1)
```

Finally we let s be the syndrome of r with respect to C, apply f to s to get the coset leader l, and subtract l from r to get the corrected vector v. Finding the coordinates of v with respect to the basis of C (the rows of the generator matrix of C) gives the original information vector.

```
> s := Syndrome(r, C);
> s;
(1 1 1)
> l := f(s);
> l;
(0 0 0 0 0 0 1)
> v := r - l;
> v;
(1 0 1 1 0 1 0)
> res := I ! Coordinates(C, v);
> res;
(1 0 1 1)
```

158.6 Subcodes

158.6.1 The Subcode Constructor

`sub< C | L >`

Given an $[n, k]$ linear code C over R, construct the subcode of C, generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) An element of C;

(b) A set or sequence of elements of C;

(c) A sequence of n elements of R, defining an element of C;

(d) A set or sequence of sequences of type (c);

(e) A subcode of C;

`Subcode(C, k)`

Given an $[n, k]$ linear code C and an integer t, $1 \leq t < n$, return a subcode of C of dimension t.

`Subcode(C, S)`

Given an $[n, k]$ linear code C and a set S of integers, each of which lies in the range $[1, k]$, return the subcode of C generated by the basis elements whose positions appear in S.

`SubcodeBetweenCode(C1, C2, k)`

Given a linear code C_1 and a subcode C_2 of C_1, return a subcode of C_1 of dimension k containing C_2.

`SubcodeWordsOfWeight(C, S)`

Given an $[n, k]$ linear code C and a set S of integers, each of which lies in the range $[1, n]$, return the subcode of C generated by those words of C whose weights lie in S.

Example H158E14

We give an example of how `SubcodeBetweenCode` may be used to create a code nested in between a subcode pair.

```
> C1 := RepetitionCode(GF(2),6);
> C1;
[6, 1, 6] Cyclic Code over GF(2)
Generator matrix:
[1 1 1 1 1 1]
> C3 := EvenWeightCode(6);
> C3;
[6, 5, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 1]
[0 1 0 0 0 1]
[0 0 1 0 0 1]
[0 0 0 1 0 1]
[0 0 0 0 1 1]
> C1 subset C3;
true
> C2 := SubcodeBetweenCode(C3, C1, 4);
> C2;
[6, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 0]
[0 1 0 0 0 1]
[0 0 1 0 0 1]
[0 0 0 1 0 1]
> (C1 subset C2) and (C2 subset C3);
true
```

158.6.2 Sum, Intersection and Dual

For the following operators, C and D are codes defined as subsets (or subspaces) of the same R-space V.

```
C + D
```

The (vector space) sum of the linear codes C and D, where C and D are contained in the same K-space V.

```
C meet D
```

The intersection of the linear codes C and D, where C and D are contained in the same K-space V.

```
Dual(C)
```

The dual D of the linear code C. The dual consists of all codewords in the K-space V which are orthogonal to all codewords of C.

Example H158E15

Verify some simple results from the sum and intersection of subcodes with known basis.

```
> C := EvenWeightCode(5);
> C;
[5, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1]
[0 1 0 0 1]
[0 0 1 0 1]
[0 0 0 1 1]
> C1 := sub< C | C.1 >;
> C2 := sub< C | C.4 >;
> C3 := sub< C | { C.1 , C.4} >;
> (C1 + C2) eq C3;
true
> (C1 meet C3) eq C1;
true
```

Example H158E16

Verify the orthogonality of codewords in the dual for a `ReedSolomonCode`.

```
> K<w> := GF(8);
> R := ReedSolomonCode(K, 3);
> R;
[7, 5, 3] BCH code (d = 3, b = 1) over GF(2^3)
Generator matrix:
[  1   0   0   0   0 w^3 w^4]
[  0   1   0   0   0   1   1]
[  0   0   1   0   0 w^3 w^5]
[  0   0   0   1   0   w w^5]
[  0   0   0   0   1   w w^4]
> D := Dual(R);
> D;
[7, 2, 6] Cyclic Code over GF(2^3)
Generator matrix:
[  1   0 w^3   1 w^3   w   w]
[  0   1 w^4   1 w^5 w^5 w^4]
> {<u,v> : u in R, v in D | InnerProduct(u,v) ne 0};
{}
```

158.6.3 Membership and Equality

For the following operators, C and D are codes defined as a subset (or subspace) of the R-space V.

`u in C`

Return `true` if and only if the vector u of V belongs to the code C.

`u notin C`

Return `true` if and only if the vector u of V does not belong to the code C.

`C subset D`

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

`C notsubset D`

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

`C eq D`

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

`C ne D`

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

158.7 Properties of Codes

For the following operators, C and D are codes defined as a subset (or subspace) of the vector space V.

```
IsCyclic(C)
```

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

```
IsSelfDual(C)
```

Return `true` if and only if the linear code C is self-dual. (i.e. C equals the dual of C).

```
IsSelfOrthogonal(C)
```

Return `true` if and only if the linear code C is self-orthogonal (i.e., C is contained in the dual of C).

```
IsMaximumDistanceSeparable(C)
IsMDS(C)
```

Returns `true` if and only if the linear code C is maximum-distance separable; that is, has parameters $[n, k, n-k+1]$.

```
IsEquidistant(C)
```

Returns `true` if and only if the linear code C is equidistant.

```
IsPerfect(C)
```

Returns `true` if and only if the linear code C is perfect; that is, if and only if the cardinality of C is equal to the size of the sphere packing bound of C.

```
IsNearlyPerfect(C)
```

Returns `true` if and only if the binary linear code C is nearly perfect.

```
IsEven(C)
```

Returns `true` if and only if C is an even linear binary code, (i.e., all codewords have even weight). If `true`, then MAGMA will adjust the upper and lower minimum weight bounds of C if possible.

```
IsDoublyEven(C)
```

Returns `true` if and only if C is a doubly even linear binary code, (i.e., all codewords have weight divisible by 4). If `true`, then MAGMA will adjust the upper and lower minimum weight bounds of C if possible.

```
IsProjective(C)
```

Returns `true` if and only if the (non-quantum) code C is projective.

Example H158E17

We look at an extended quadratic residue code over $GF(2)$ which is self-dual, and then confirm it manually.

```
> C := ExtendCode( QRCode(GF(2),23) );
> C:Minimal;
[24, 12, 8] Linear Code over GF(2)
> IsSelfDual(C);
true
> D := Dual(C);
> D: Minimal;
[24, 12, 8] Linear Code over GF(2)
> C eq D;
true
```

Example H158E18

We look at the CordaroWagnerCode of length 6, which is self-orthogonal, and then confirm it manually.

```
> C := CordaroWagnerCode(6);
> C;
[6, 2, 4] Linear Code over GF(2)
Generator matrix:
[1 1 0 0 1 1]
[0 0 1 1 1 1]
> IsSelfOrthogonal(C);
true
> D := Dual(C);
> D;
[6, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0 1]
[0 1 0 1 0 1]
[0 0 1 1 0 0]
[0 0 0 0 1 1]
> C subset D;
true
```

158.8 The Weight Distribution

158.8.1 The Minimum Weight

In the case of a linear code, the minimum weight and distance are equivalent. It is clear that is substantially easier to determine the minimum weight of a (possibly non-linear) code than its minimum distance. The general principle underlying the minimum weight algorithm in Magma is the embedding of low-weight information vectors into the code space, in the hope that they will map onto low weight codewords.

Let C be a $[n, k]$ linear code over a finite field and G be its generator matrix. The minimum weight algorithm proceeds as follows: Starting with $r = 1$, all linear combinations of r rows of G are enumerated. By taking the minimum weight of each such combination, an upper bound, d_{upper}, on the minimum weight of C is obtained. A strictly increasing function, $d_{lower}(r)$, finds a lower bound on the minimum weight of the non-enumerated vectors for each computational step (the precise form of this function depends upon the algorithm being used). The algorithm terminates when $d_{lower}(r) \geq d_{upper}$, at which point the actual minimum weight is equal to d_{upper}.

The algorithm is used for non-cyclic codes, and is due to A.E. Brouwer and K.H. Zimmermann [BFK$^+$98]. The key idea is to construct as many different generator matrices for the same code as possible, each having a different information set and such that the information sets are as disjoint as possible. By maximizing the number of information sets, $d_{lower}(r)$ can be made increasingly accurate. Each information set will provide a different embedding of information vectors into the code, and thus the probability of a low-weight information vector mapping onto a low-weight codeword is increased.

A well known improvement attributed to Brouwer exists for cyclic codes, requiring the enumeration fo only one information set. A generalisation of this improvement has been made by G. White to quasicyclic codes, and any codes whose known automorphisms have large cycles. Functionality is included in this section for inputting partial knowledge of the automorphism group to take advantage of this improvement.

Information sets are discarded if their ranks are too low to contribute to the lower bound calculation. The user may also specify a lower bound, `RankLowerBound`, on the rank of information sets initially created.

`MinimumWeight(C: parameters)`

`MinimumDistance(C: parameters)`

`Method`	MonStgElt	*Default :* "*Auto*"
`RankLowerBound`	RngIntElt	*Default :* 0
`MaximumTime`	RngReSubElt	*Default :* ∞
`Nthreads`	RngIntElt	*Default :* 1

Determine the minimum weight of the words belonging to the code C, which is also the minimum distance between any two codewords. The parameter `RankLowerBound` sets a minimum rank on the information sets used in the calculation, while the parameter `MaximumTime` sets a time limit (in seconds of "user time") after which the calculation is aborted.

If the base field is $\mathbf{F}_2$ and the parameter Nthreads is set to a positive integer n, then n threads will be used in the computation, if POSIX threads are enabled. One can alternatively use the procedure SetNthreads to set the global number of threads to a value n so that n threads are always used by default in this algorithm unless overridden by the Nthreads parameter.

Sometimes a brute force calculation of the entire weight distribution can be a faster way to get the minimum weight 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 using setting it to either "Distribution" or "Zimmerman".

By setting the verbose flag "Code", information about the progress of the computation can be printed. An example to demonstrate the interpretation of the verbose output follows:

```
> SetVerbose("Code", true);
> SetSeed(1);
> MinimumWeight(RandomLinearCode(GF(2),85,26));
Linear Code over GF(2) of length 85 with 26 generators. Is not cyclic
Lower Bound: 1, Upper Bound: 60
Constructed 4 distinct generator matrices
Relative Ranks:  26  26  26   7
Starting search for low weight codewords...
Enumerating using 1 generator at a time:
     New codeword identified of weight 32, time 0.000
     New codeword identified of weight 28, time 0.000
     New codeword identified of weight 27, time 0.000
     New codeword identified of weight 25, time 0.000
     Discarding non-contributing rank 7 matrix
     New Relative Ranks:  26  26  26
   Completed Matrix  1:  lower =  4, upper = 25. Time so far: 0.000
     New codeword identified of weight 23, time 0.000
   Completed Matrix  2:  lower =  5, upper = 23. Time so far: 0.000
   Completed Matrix  3:  lower =  6, upper = 23. Time so far: 0.000
Enumerating using 2 generators at a time:
     New codeword identified of weight 20, time 0.000
   Completed Matrix  1:  lower =  7, upper = 20. Time so far: 0.000
   Completed Matrix  2:  lower =  8, upper = 20. Time so far: 0.000
   Completed Matrix  3:  lower =  9, upper = 20. Time so far: 0.000
Enumerating using 3 generators at a time:
     New codeword identified of weight 19, time 0.000
   Completed Matrix  1:  lower = 10, upper = 19. Time so far: 0.000
   Completed Matrix  2:  lower = 11, upper = 19. Time so far: 0.000
   Completed Matrix  3:  lower = 12, upper = 19. Time so far: 0.000
Enumerating using 4 generators at a time:
     New codeword identified of weight 18, time 0.000
   Completed Matrix  1:  lower = 13, upper = 18. Time so far: 0.000
     New codeword identified of weight 17, time 0.000
   Completed Matrix  2:  lower = 14, upper = 17. Time so far: 0.010
   Completed Matrix  3:  lower = 15, upper = 17. Time so far: 0.010
Termination predicted with 5 generators at matrix 2
Enumerating using 5 generators at a time:
   Completed Matrix  1:  lower = 16, upper = 17. Time so far: 0.020
   Completed Matrix  2:  lower = 17, upper = 17. Time so far: 0.030
Final Results: lower = 17, upper = 17, Total time: 0.030
17
```

Verbose output can be invaluable on long minimum weight calculations.

The algorithm constructs different (equivalent) generator matrices, each of which have pivots in different column positions of the code, called its *information set*. A generator matrix's *relative rank* is the size of its information set independent from the previously constructed matrices.

The algorithm proceeds by enumerating all combinations derived from r generators, for each successive r. Once r exceeds the difference between the actual rank of a matrix (i.e., the dimension), and its relative rank, then the lower bound on the minimum weight will increment by 1 for that step.

The upper bound on the minimum weight is determined by the minimum weight of codewords that are enumerated. Once these bounds meet the computation is complete.

`MinimumWeightBounds(C)`

Return the currently known lower and upper bounds on the minimum weight of code C.

`ResetMinimumWeightBounds(C)`

Undefine the minimum weight of the code C if it is known, and reset any known bounds on its value.

`VerifyMinimumDistanceLowerBound(C, d)`

`RankLowerBound` RngIntElt *Default :* 0

`MaximumTime` RngIntElt *Default :* ∞

The minimum weight algorithm is executed until it determines whether or not d is a lower bound for the minimum weight of the code C. (See the description of the function `MinimumWeight` for information on the parameters `RankLowerBound` and `MaximumTime` and on the verbose output). Three values are returned. The first of these is a boolean value, taking the value `true` if and only if d is verified to be a lower bound for the minimum weight of C, (`false` if the calculation is aborted due to time restrictions). The second return value is the best available lower bound for the minimum weight of C, and the third is a boolean which is `true` if this value is the actual minimum weight of C.

`VerifyMinimumDistanceUpperBound(C, d)`

`VerifyMinimumWeightUpperBound(C, d)`

RankLowerBound	RngIntElt	*Default :* 0
MaximumTime	RngIntElt	*Default :* ∞

The minimum weight algorithm is executed until it determines whether or not d is an upper bound for the minimum weight of the code C. (See the description of the function `MinimumWeight` for information on the parameters `RankLowerBound` and `MaximumTime` and on the verbose output). Three values are returned. The first of these is a boolean value, taking the value `true` if and only if d is verified to be an upper bound for the minimum weight of C, (`false` if the calculation is aborted due to time restrictions). The second return value is the best available upper bound for the minimum weight of C, and the third is a boolean which is `true` if this value is the actual minimum weight of C.

`MinimumWord(C)`

Return one word of the code C having minimum weight.

`MinimumWords(C)`

NumWords	RngIntElt	*Default :*
Method	MonStgElt	*Default :* "*Auto*"
RankLowerBound	RngIntElt	*Default :* ∞
MaximumTime	RngReSubElt	*Default :* ∞

Given a linear code C, return the set of all words of C having minimum weight. If `NumWords` is set to a non-negative integer, then the algorithm will terminate after that total of words have been found. Similarly, if `MaximumTime` then the algorithm will abort if the specified time limit expires.

A variation of the Zimmermann minimum weight algorithm is generally used to collect the minimum words, although in some cases (such as small codes) a brute force enumeration may be used. 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 `"Zimmerman"`.

By setting the verbose flag `"Code"`, information about the progress of the computation can be printed.

Example H158E19

The function `BKLC(K, n, k)` returns the best known linear $[n, k]$-code over the field K. We use this function to construct the $[77, 34, 16]$ best known linear code and confirm a lower bound on its minimum weight (which is not as good as its actual minimum weight). We check to see whether the minimum weight of this code is at least 11 and in doing so we will actually get a slightly better bound, though it will be still less than the true minimum weight. Since the function `BLKC` will set the true minimum weight, it is first necessary to reset the bounds so that the minimum weight data is lost.

```
> a := BKLC(GF(2),77,34);
> a:Minimal;
[77, 34, 16] Linear Code over GF(2)
> ResetMinimumWeightBounds(a);
> MinimumWeightBounds(a);
1 44
> a:Minimal;
[77, 34] Linear Code over GF(2)
> SetVerbose("Code",true);
> IsLB, d_lower, IsMinWeight := VerifyMinimumWeightLowerBound(a, 11);
Linear Code over GF(2) of length 77 with 34 generators. Is not cyclic
Lower Bound: 1, Upper Bound: 44
Using congruence d mod 4 = 0
Constructed 3 distinct generator matrices
Relative Ranks:  34  34   6
Starting search for low weight codewords...
     Discarding non-contributing rank 6 matrix
Enumerating using 1 generator at a time:
     New codeword identified of weight 20, time 0.000
     New codeword identified of weight 16, time 0.000
   Completed Matrix  1:  lower =  4, upper = 16. Time so far: 0.000
   Completed Matrix  2:  lower =  4, upper = 16. Time so far: 0.000
Enumerating using 2 generators at a time:
   Completed Matrix  1:  lower =  8, upper = 16. Time so far: 0.000
   Completed Matrix  2:  lower =  8, upper = 16. Time so far: 0.000
Enumerating using 3 generators at a time:
   Completed Matrix  1:  lower =  8, upper = 16. Time so far: 0.000
   Completed Matrix  2:  lower =  8, upper = 16. Time so far: 0.000
Enumerating using 4 generators at a time:
   Completed Matrix  1:  lower = 12, upper = 16. Time so far: 0.010
Final Results: lower = 12, upper = 16, Total time: 0.010
> IsLB;
true
> d_lower, IsMinWeight;
12 false
```

`IncludeAutomorphism(~C, p)`

`IncludeAutomorphism(~C, G)`

Given some automorphism p or group of automorphisms G of the code C, which can either be a permutation of the columns or a full monomial permutation of the code. Then include these automorphism in the known automorphisms subgroup. Automorphisms with long cycles that can aid the minimum weight calculation should be added in this way.

`KnownAutomorphismSubgroup(C)`

Return the maximally known subgroup of the full group of automorphisms of the code C.

158.8.2 The Weight Distribution

`WeightDistribution(C)`

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

`WeightDistribution(C, u)`

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

`DualWeightDistribution(C)`

The weight distribution of the dual code of C (see `WeightDistribution`).

Example H158E20

We construct the second order Reed–Muller code of length 64, and calculate the its weight distribution and that of its dual code.

```
> R := ReedMullerCode(2, 6);
> #R;
4194304
> WeightDistribution(R);
[ <0, 1>, <16, 2604>, <24, 291648>, <28, 888832>, <32, 1828134>, <36, 888832>,
<40, 291648>, <48, 2604>, <64, 1> ]
> D := Dual(R);
> #D;
4398046511104
> time WeightDistribution(D);
[ <0, 1>, <8, 11160>, <12, 1749888>, <14, 22855680>, <16, 232081500>, <18,
1717223424>, <20, 9366150528>, <22, 38269550592>, <24, 119637587496>, <26,
286573658112>, <28, 533982211840>, <30, 771854598144>, <32, 874731154374>,
<34, 771854598144>, <36, 533982211840>, <38, 286573658112>, <40,
119637587496>, <42, 38269550592>, <44, 9366150528>, <46, 1717223424>,
<48, 232081500>, <50, 22855680>, <52, 1749888>, <56, 11160>, <64, 1> ]
```

```
PartialWeightDistribution(C, ub)
```

Return the weight distribution of the code C up to the specified upper bound. This function uses the minimum weight collection to collect word sets.

158.8.3 The Weight Enumerator

```
WeightEnumerator(C)
```

The (Hamming) weight enumerator $W_C(x,y)$ for the linear code C. The weight enumerator is defined by

$$W_C(x,y) = \sum_{u \in C} x^{n-wt(u)} y^{wt(u)}.$$

```
WeightEnumerator(C, u)
```

The (Hamming) weight enumerator $W_{C+u}(x,y)$ for the coset $C+u$.

```
CompleteWeightEnumerator(C)
```

The complete weight enumerator $\mathcal{W}_C(z_0, \dots, z_{q-1})$ for the linear code C where q is the size of the alphabet K of C. Let the q elements of K be denoted by $\omega_0, \dots, \omega_{q-1}$. If K is a prime field, we let ω_i be i (i.e. take the natural representation of each number). If K is a non-prime field, we let ω_0 be the zero element of K and let ω_i be α^{i-1} for $i = 1 \dots q-1$ where α is the primitive element of K. Now for a codeword u of C, let $s_i(u)$ be the number of components of u equal to ω_i. The complete weight enumerator is defined by

$$\mathcal{W}_C(z_0, \dots, z_{q-1}) = \sum_{u \in C} {z_0}^{s_0(u)} \dots {z_{q-1}}^{s_{q-1}(u)}.$$

```
CompleteWeightEnumerator(C, u)
```

The complete weight enumerator $\mathcal{W}_{C+u}(z_0, \dots, z_{q-1})$ for the coset $C+u$.

Example H158E21

We construct the cyclic ternary code of length 11 with generator polynomial $t^5 + t^4 + 2t^3 + t^2 + 2$ and calculate both its weight enumerator and its complete weight enumerator. To ensure the polynomials print out nicely, we assign names to the polynomial ring indeterminates in each case. These names will persist if further calls to `WeightEnumerator` and `CompleteWeightEnumerator` over the same alphabet are made.

```
> R<t> := PolynomialRing(GF(3));
> C := CyclicCode(11, t^5 + t^4 + 2*t^3 + t^2 + 2);
> W<x, y> := WeightEnumerator(C);
> W;
x^11 + 132*x^6*y^5 + 132*x^5*y^6 + 330*x^3*y^8 + 110*x^2*y^9 + 24*y^11
> CW<u, v, w> := CompleteWeightEnumerator(C);
> CW;
u^11 + 11*u^6*v^5 + 55*u^6*v^3*w^2 + 55*u^6*v^2*w^3 + 11*u^6*w^5 +
      11*u^5*v^6 + 110*u^5*v^3*w^3 + 11*u^5*w^6 + 55*u^3*v^6*w^2 +
      110*u^3*v^5*w^3 + 110*u^3*v^3*w^5 + 55*u^3*v^2*w^6 + 55*u^2*v^6*w^3 +
      55*u^2*v^3*w^6 + v^11 + 11*v^6*w^5 + 11*v^5*w^6 + w^11
```

The vector $u = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)$ does not lie in the code C and can be taken as a coset leader. We determine the weight enumerator of the coset containing u.

```
> u := AmbientSpace(C)![0,0,0,0,0,0,0,0,0,0,1];
> Wu := WeightEnumerator(C, u);
> Wu;
x^10*y + 30*x^7*y^4 + 66*x^6*y^5 + 108*x^5*y^6 + 180*x^4*y^7 + 165*x^3*y^8 +
    135*x^2*y^9 + 32*x*y^10 + 12*y^11
```

158.8.4 The MacWilliams Transform

`MacWilliamsTransform(n, k, q, W)`

Let C be a hypothetical $[n, k]$ linear code over a finite field of cardinality q. Let W be the weight distribution of C (in the form as returned by the function `WeightDistribution`). This function applies the MacWilliams transform to W to obtain the weight distribution W' of the dual code of C. The transform is a combinatorial algorithm based on n, k, q and W alone. Thus C itself need not exist—the function simply works with the sequence of integer pairs supplied by the user. Furthermore, if W is not the weight distribution of an actual code, the result W' will be meaningless and even negative weights may be returned.

`MacWilliamsTransform(n, k, K, W)`

Let C be a hypothetical $[n, k]$ linear code over a finite field K. Let W be the complete weight enumerator of C (in the form as returned by the function `CompleteWeightEnumerator`). This function applies the MacWilliams transform to W to obtain the complete weight enumerator W' of the dual code of C. The transform is a combinatorial algorithm based on K, n, k, and W alone. Thus C itself need not exist—the function simply manipulates the given polynomial. Furthermore, if W is not the weight distribution of an actual code, the weight enumerator W' will be meaningless.

Example H158E22

Let us suppose there exists a $[31, 11]$ code C over F_2 that has complete weight enumerator

$$u^{31} + 186u^{20}v^{11} + 310u^{19}v^{12} + 527u^{16}v^{15} + 527u^{15}v^{16} + 310u^{12}v^{19} + 186u^{11}v^{20} + v^{31}$$

We compute the weight distribution and the complete weight enumerator of the dual of the hypothetical code C.

```
> W := [ <0, 1>, <11, 186>, <12, 310>, <15, 527>, <16, 527>,
>       <19, 310>, <20, 186>, <31, 1> ];
>  MacWilliamsTransform(31, 11, 2, W);
[ <0, 1>, <6, 806>, <8, 7905>, <10, 41602>, <12, 142600>, <14,
251100>, <16, 301971>, <18, 195300>, <20, 85560>, <22, 18910>, <24,
2635>, <26, 186> ]
> R<u, v> := PolynomialRing(Integers(), 2);
> CWE := u^31 + 186*u^20*v^11 + 310*u^19*v^12 + 527*u^16*v^15 + 527*u^15*v^16 +
>          310*u^12*v^19 + 186*u^11*v^20 + v^31;
> MacWilliamsTransform(31, 11, GF(2), CWE);
u^31 + 806*u^25*v^6 + 7905*u^23*v^8 + 41602*u^21*v^10 + 142600*u^19*v^12 +
      251100*u^17*v^14 + 301971*u^15*v^16 + 195300*u^13*v^18 + 85560*u^11*v^20 +
      18910*u^9*v^22 + 2635*u^7*v^24 + 186*u^5*v^26
```

158.8.5 Words

The functions in this section only apply to codes over finite fields.

`Words(C, w: parameters)`

NumWords	RngIntElt	*Default :*
Method	MonStgElt	*Default :* “*Auto*”
RankLowerBound	RngIntElt	*Default :* ∞
MaximumTime	RngReSubElt	*Default :* ∞

Given a linear code C, return the set of all words of C having weight w. If `NumWords` is set to a non-negative integer c, then the algorithm will terminate after that total of words have been found. Similarly, if `MaximumTime` then the algorithm will abort if the specified time limit expires.

There are two methods for collecting words, one based on the Zimmermann minimum weight algorithm, and a brute force type calculation. 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 `"Zimmerman"`.

By setting the verbose flag `"Code"`, information about the progress of the computation can be printed.

`NumberOfWords(C, w)`

Given a linear code C, return the number of words of C having weight w.

`WordsOfBoundedWeight(C, l, u:` *parameters*`)`

`Cutoff`	RngIntElt	*Default :* ∞
`StoreWords`	BoolElt	*Default :* `true`

Given a linear code C, return the set of all words of C having weight between l and u, inclusive. If `Cutoff` is set to a non-negative integer c, then the algorithm will terminate after a total of c words have been found.

If `StoreWords` is true then any words of a *single weight* generated will be stored internally.

`ConstantWords(C, i)`

Given a linear code C, return the set of all words of C which have weight i and which consist of zeros and ones alone.

`NumberOfConstantWords(C, i)`

Given a linear code C, return the number of words of C which have weight i and which consist of zeros and ones alone.

Example H158E23

We construct the words of weight 11 and also the constant (zero-one) words of weight 11 in the length 23 cyclic code over $\mathbf{F}_3$ that is defined by the generator polynomial $x^{11} + x^{10} + x^9 + 2x^8 + 2x^7 + x^5 + x^3 + 2$.

```
> R<x> := PolynomialRing(GF(3));
> f := x^11 + x^10 + x^9 + 2*x^8 + 2*x^7 + x^5 + x^3 + 2;
> C := CyclicCode(23, f);
> C;
[23, 12, 8] BCH code (d = 5, b = 1) over GF(3)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 2 0 0 1 0 1 0 2 2 1 1]
[0 1 0 0 0 0 0 0 0 0 0 0 1 2 0 2 1 2 1 1 0 1 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 1 2 0 2 1 2 1 1 0 1]
[0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 1 0 1 1 0 2 0 2]
```

```
[0 0 0 0 1 0 0 0 0 0 0 0 2 1 0 2 1 1 1 0 2 0 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 2 1 2 2 0 1 2 1 1 2]
[0 0 0 0 0 0 1 0 0 0 0 0 2 1 2 2 2 0 0 0 1 2 2]
[0 0 0 0 0 0 0 1 0 0 0 0 2 2 1 0 2 0 0 2 2 2 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 2 2 1 0 2 0 0 2 2 2]
[0 0 0 0 0 0 0 0 0 1 0 0 2 0 2 0 1 1 2 2 2 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 2 0 2 0 1 1 2 2 2 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 2 0 2 0 1 1 2 2 2]
> time WeightDistribution(C);
[ <0, 1>, <8, 1518>, <9, 2530>, <11, 30912>, <12, 30912>, <14, 151800>, <15,
91080>, <17, 148764>, <18, 49588>, <20, 21252>, <21, 3036>, <23, 48> ]
Time: 0.030
```

Note that the minimum distance is 8. We calculate all words of weight 11 and the constant words of weight 11.

```
1518
> time W11 := Words(C, 11);
Time: 0.350
> #W11;
30912
> ZOW11 := ConstantWords(C, 11);
> #ZOW11;
23
> ZOW11 subset W11;
true
```

158.8.6 Covering Radius and Diameter

`CosetDistanceDistribution(C)`

Given a linear code C, determine the coset distance distribution of C, relative to C. The distance between C and a coset D of C is the Hamming weight of a vector of minimum weight in D. The distribution is returned as a sequence of pairs comprising a distance d and the number of cosets that are distance d from C.

`CoveringRadius(C)`

The covering radius of the linear code C.

`Diameter(C)`

The diameter of the code C (the largest weight of the codewords of C).

Example H158E24

We construct the second order Reed–Muller code of length 32, and calculate its coset distance distribution.

```
> R := ReedMullerCode(2, 5);
> R:Minimal;
[32, 16, 8] Reed-Muller Code (r = 2, m = 5) over GF(2)
> CD := CosetDistanceDistribution(R);
> CD;
[ <0, 1>, <1, 32>, <2, 496>, <3, 4960>, <4, 17515>, <5, 27776>, <6, 14756> ]
```

From the dimension of the code we know C has 2^{16} cosets. The coset distance distribution tells us that there are 32 cosets at distance 1 from C, 496 cosets are distance 2, etc. We confirm that all cosets are represented in the distribution.

```
> &+ [ t[2] : t in CD ];
65536
> CoveringRadius(R);
6
> Diameter(R);
32
> WeightDistribution(R);
[ <0, 1>, <8, 620>, <12, 13888>, <16, 36518>, <20, 13888>, <24, 620>, <32, 1> ]
```

The covering radius gives the maximum distance of any coset from the code, and, from the coset distance distribution, we see that this maximum distance is indeed 6. We can confirm the value (32) for the diameter by examining the weight distribution and seeing that 32 is the largest weight of a codeword.

158.9 Families of Linear Codes

158.9.1 Cyclic and Quasicyclic Codes

```
CyclicCode(u)
```

Given a vector u belonging to the R-space $R^{(n)}$, construct the $[n, k]$ cyclic code generated by the right cyclic shifts of the vector u.

```
CyclicCode(n, T, K)
```

Given a positive integer n and a set or sequence T of primitive n-th roots of unity from a finite field L, together with a subfield K of L, construct the cyclic code C over K of length n, such that the generator polynomial for C is the polynomial of least degree having the elements of T as roots.

```
QuasiCyclicCode(n, Gen)
```

Constructs the quasi-cyclic code of length n with generator polynomials given by the sequence of polynomials in *Gen*. Created by `HorizontalJoin` of each `GeneratorMatrix` from the `CyclicCode`'s generated by the polynomials in *Gen*. Requires that $|Gen| \mid n$.

```
QuasiCyclicCode(Gen)
```

Constructs the quasi-cyclic code of length n generated by simultaneous cyclic shifts of the vectors in *Gen*.

```
QuasiCyclicCode(n, Gen, h)
```

Constructs the quasi-cyclic code of length n with generator polynomials given by the sequence of polynomials in *Gen*. The `GeneratorMatrix`'s are joined 2 dimensionally, with height h. Requires that $h \mid (|Gen|)$ and $(|Gen|/h) \mid n$.

```
QuasiCyclicCode(Gen, h)
```

Constructs the quasi cyclic code generated by simultaneous cyclic shifts of the vectors in *Gen*, arranging them two dimensionally with height h.

```
ConstaCyclicCode(n, f, alpha)
```

Return the length n code generated by consta-cyclic shifts by α of the coefficients of f.

```
QuasiTwistedCyclicCode(n, Gen, alpha)
```

Construct the quasi-twisted cyclic code of length n pasting together the constacyclic codes with parameter α generated by the polynomials in *Gen*.

```
QuasiTwistedCyclicCode(Gen, alpha)
```

Construct the quasi-twisted cyclic code generated by simultaneous constacyclic shifts w.r.t. α of the codewords in *Gen*.

Example H158E25

Let the m factors of $x^n - 1$ be $f_i(x)$,$i = 0, \dots, m$ in any particular order. Then we can construct a chain of polynomials $g_k(x) = \prod_{i=0}^{k} f_i(x)$ such that $g_k(x) \mid g_{k+1}(x)$. This chain of polynomials will generate a nested chain of cyclic codes of length n, which is illustrated here for $n = 7$.

```
> P<x> := PolynomialRing(GF(2));
> n := 7;
> F := Factorization(x^n-1);
> F;
[
    <x + 1, 1>,
    <x^3 + x + 1, 1>,
    <x^3 + x^2 + 1, 1>
]
```

```
> Gens := [ &*[F[i][1]:i in [1..k]] : k in [1..#F] ];
> Gens;
[
    x + 1,
    x^4 + x^3 + x^2 + 1,
    x^7 + 1
]
> Codes := [ CyclicCode(n, Gens[k]) : k in [1..#Gens] ];
> Codes;
[
    [7, 6, 2] Cyclic Code over GF(2)
    Generator matrix:
    [1 0 0 0 0 0 1]
    [0 1 0 0 0 0 1]
    [0 0 1 0 0 0 1]
    [0 0 0 1 0 0 1]
    [0 0 0 0 1 0 1]
    [0 0 0 0 0 1 1],
     [7, 3, 4] Cyclic Code over GF(2)
    Generator matrix:
    [1 0 0 1 0 1 1]
    [0 1 0 1 1 1 0]
    [0 0 1 0 1 1 1],
    [7, 0, 7] Cyclic Code over GF(2)
]
> { Codes[k+1] subset Codes[k] : k in [1..#Codes-1] };
{ true }
```

158.9.2 BCH Codes and their Generalizations

```
BCHCode(K, n, d, b)
BCHCode(K, n, d)
```

Given a finite field $K = F_q$ and positive integers n, d and b such that $\gcd(n, q) = 1$, we define m to be the smallest integer such that $n \mid (q^m - 1)$, and α to be a primitive n-th root of unity in the degree m extension of K, $GF(q^m)$. This function constructs the BCH code of designated distance d as the cyclic code with generator polynomial

$$g(x) = \mathrm{lcm}\{m_1(x), \cdots, m_{d-1}(x)\}$$

where $m_i(x)$ is the minimum polynomial of α^{b+i-1}. The BCH code is an $[n, \geq (n - m(d-1)), \geq d]$ code over K. If b is omitted its value is taken to be 1, in which case the corresponding code is a *narrow sense* BCH code.

Example H158E26

We construct a BCH code of length 13 over GF(3) and designated minimum distance 3

```
> C := BCHCode(GF(3), 13, 3);
> C;
[13, 7, 4] BCH code (d = 3, b = 1) over GF(3)
Generator matrix:
[1 0 0 0 0 0 0 1 2 1 2 2 2]
[0 1 0 0 0 0 0 1 0 0 0 1 1]
[0 0 1 0 0 0 0 2 2 2 1 1 2]
[0 0 0 1 0 0 0 1 1 0 1 0 0]
[0 0 0 0 1 0 0 0 1 1 0 1 0]
[0 0 0 0 0 1 0 0 0 1 1 0 1]
[0 0 0 0 0 0 1 2 1 2 2 2 1]
```

```
GoppaCode(L, G)
```

Let K be the field $GF(q)$, let $G(z) = G$ be a polynomial defined over the degree m extension field F of K (i.e. the field $GF(q^m)$) and let $L = [\alpha_1, \dots, \alpha_n]$ be a sequence of elements of F such that $G(\alpha_i) \neq 0$ for all $\alpha_i \in L$. This function constructs the Goppa code $\Gamma(L, G)$ over K. If the degree of $G(z)$ is r, this is an $[n, k \geq n - mr, d \geq r + 1]$ code.

Example H158E27

We construct a Goppa code of length 31 over GF(2) with generator polynomial $G(z) = z^3 + z + 1$.

```
> q := 2^5;
> K<w> := FiniteField(q);
> P<z> := PolynomialRing(K);
> G := z^3 + z + 1;
> L := [w^i : i in [0 .. q - 2]];
> C := GoppaCode(L, G);
> C:Minimal;
[31, 16, 7] Goppa code (r = 3) over GF(2)
> WeightDistribution(C);
[ <0, 1>, <7, 105>, <8, 295>, <9, 570>, <10, 1333>, <11, 2626>,
 <12, 4250>, <13, 6270>, <14, 8150>, <15, 9188>, <16, 9193>,
 <17, 8090>, <18, 6240>, <19, 4270>, <20, 2590>, <21, 1418>,
 <22, 650>, <23, 195>, <24, 55>, <25, 36>, <26, 11> ]
```

```
ChienChoyCode(P, G, n, S)
```

Let P and G be polynomials over a finite field F, let n be an integer greater than one, and let S be a subfield of F. Suppose also that n is coprime to the cardinality of S, F is the splitting field of $x^n - 1$ over S, P and G are both coprime to $x^n - 1$ and both have degree less than n. This function constructs the Chien-Choy generalised BCH code with parameters P, G, n over S.

```
AlternantCode(A, Y, r, S)
```

```
AlternantCode(A, Y, r)
```

Let $A = [\alpha_1, \ldots, \alpha_n]$ be a sequence of n distinct elements taken from the degree m extension K of the finite field S, and let $Y = [y_1, \ldots, y_n]$ be a sequence of n non-zero elements from K. Let r be a positive integer. Given such A, Y, r, and S, this function constructs the alternant code $A(A, Y)$ over S. This is an $[n, k \geq n - mr, d \geq r + 1]$ code. If S is omitted, S is taken to be the prime subfield of K.

Example H158E28

We construct an alternant code over GF(2) based on sequences of elements in the extension field GF(2^4) of GF(2). The parameter r is taken to be 4, so the minimum weight 6 is greater than $r + 1$.

```
> q := 2^4;
> K<w> := GF(q);
> A := [w ^ i : i in [0 .. q - 2]];
> Y := [K ! 1 : i in [0 .. q - 2]];
> r := 4;
> C := AlternantCode(A, Y, r);
> C;
[15, 6, 6] Alternant code over GF(2)
Generator matrix:
[1 0 0 0 0 0 1 1 0 0 1 1 1 0 0]
[0 1 0 0 0 0 0 1 1 0 0 1 1 1 0]
[0 0 1 0 0 0 0 0 1 1 0 0 1 1 1]
[0 0 0 1 0 0 1 1 0 1 0 1 1 1 1]
[0 0 0 0 1 0 1 0 1 0 0 1 0 1 1]
[0 0 0 0 0 1 1 0 0 1 1 1 0 0 1]
```

```
NonPrimitiveAlternantCode(n, m, r)
```

Returns the $[n, k, d]$ non-primitive alternant code over $\mathbf{F}_2$, where $n - mr \leq k \leq n - r$ and $d \geq r + 1$.

```
FireCode(h, s, n)
```

Let K be the field $GF(q)$. Given a polynomial h in K[X], a nonnegative integer s, and a positive integer n, this function constructs a Fire code of length n with generator polynomial $h(X^s - 1)$.

```
GabidulinCode(A, W, Z, t)
```

Given sequences $A = [a_1, ...a_n]$, $W = [w_1, ...w_s]$, and $Z = [z_1, ...z_k]$, such that the $n+s$ elements of A and W are distinct and the elements of Z are non-zero, together with a positive integer t, construct the Gabidulin MDS code with parameters A, W, Z, t.

```
SrivastavaCode(A, W, mu, S)
```

Given sequences $A = [\alpha_1, ..., \alpha_n]$, $W = [w_1, ..., w_s]$ of elements from the extension field K of the finite field S, such that the elements of A are non-zero and the $n + s$ elements of A and W are distinct, together with an integer μ, construct the Srivastava code of parameters A, W, mu, over S.

```
GeneralizedSrivastavaCode(A, W, Z, t, S)
```

Given sequences $A = [\alpha_1, ..., \alpha_n]$, $W = [w_1, ..., w_s]$, and $Z = [z_1, ...z_k]$ of elements from the extension field K of the finite field S, such that the elements of A and Z are non-zero and the $n + s$ elements of A and W are distinct, together with a positive integer t, construct the generalized Srivastava code with parameters A, W, Z, t, over S.

158.9.3 Quadratic Residue Codes and their Generalizations

If p is an odd prime, the *quadratic residues modulo p* consist of the set of non-zero squares modulo p while the set of non-squares modulo p are termed the *quadratic nonresidues modulo p*.

```
QRCode(K, n)
```

Given a finite field $K = F_q$ and an odd prime n such that q is a quadratic residue modulo n, this function returns the quadratic residue code of length n over K. This corresponds to the cyclic code with generator polynomial $g_0(x) = \prod(x - \alpha^r)$, where α is a primitive n-th root of unity in some extension field of K, and the product is taken over all quadratic residues modulo p.

```
GolayCode(K, ext)
```

If the field K is $GF(2)$, construct the binary Golay code. If the field K is $GF(3)$, construct the ternary Golay code. If the boolean argument *ext* is `true`, construct the extended code in each case.

```
DoublyCirculantQRCode(p)
```

Given an odd prime p, this function returns the doubly circulant binary $[2p, p]$ code based on quadratic residues modulo p. A doubly circulant code has generator matrix of the form $[I \mid A]$, where A is a circulant matrix.

`DoublyCirculantQRCodeGF4(m, a)`

Given a prime power m that is greater than 2 and an integer a that is either 0 or 1, return a $[2m, m]$ doubly circulant linear code over GF(4). For details see [Gab02].

`BorderedDoublyCirculantQRCode(p, a, b)`

Given an odd prime p and integers a and b, this function returns the bordered doubly circulant binary $[2p + 1, p + 1]$ code based on quadratic residues modulo p. The construction is similar to that of a doubly circulant code except that the first p rows are extended by a mod 2 while the $p + 1$-th row is extended by b mod 2.

`TwistedQRCode(l, m)`

Given positive integers l and m, both coprime to 2, return a binary "twisted QR" code of length $l * m$.

`PowerResidueCode(K, n, p)`

Given a finite field $K = F_q$, a positive integer n and a prime p such that q is a p-th power residue modulo n, construct the p-th power residue code of length n.

Example H158E29

We construct a quadratic residue code of length 23 over GF(3).

```
> QRCode(GF(3), 23);
[23, 12, 8] Quadratic Residue code over GF(3)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 2 2 2 1 1 0 2 0 2 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 2 2 2 1 1 0 2 0 2 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 2 2 2 1 1 0 2 0 2]
[0 0 0 1 0 0 0 0 0 0 0 0 2 2 2 0 0 2 0 1 2 2 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 2 2 2 0 0 2 0 1 2 2]
[0 0 0 0 0 1 0 0 0 0 0 0 2 2 1 0 0 0 2 2 2 1 2]
[0 0 0 0 0 0 1 0 0 0 0 0 2 1 1 2 1 0 2 2 1 2 1]
[0 0 0 0 0 0 0 1 0 0 0 0 1 0 2 0 1 1 1 2 0 1 2]
[0 0 0 0 0 0 0 0 1 0 0 0 2 0 2 0 1 1 0 1 1 0 1]
[0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 2 1 2 0 2 1 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 2 1 2 0 2 1]
[0 0 0 0 0 0 0 0 0 0 0 1 1 1 2 2 0 1 0 1 0 0 2]
```

158.9.4 Reed–Solomon and Justesen Codes

```
ReedSolomonCode(K, d, b)
```

```
ReedSolomonCode(K, d)
```

Given a finite field $K = F_q$ and a positive integer d, return the Reed–Solomon code of length $n = q - 1$ with design distance d. This corresponds to `BCHCode(K, q-1, d)`. For details see [MS78, p.294].

If b is given as a non-negative integer then the primitive element is first raised to the b-th power.

```
ReedSolomonCode(n, d)
```

```
ReedSolomonCode(n, d, b)
```

Given an integer n such that $q = n + 1$ is a prime power, and a positive integer d, return the Reed–Solomon code over F_q of length n and designed minimum distance d.

If b is given as a non-negative integer then the primitive element is first raised to the b-th power.

```
GRSCode(A, V, k)
```

Let $A = [\alpha_1, \ldots, \alpha_n]$ be a sequence of n distinct elements taken from the finite field K, and let $V = [v_1, \ldots, v_n]$ be a sequence of n non-zero elements from K. Let k be a non-negative integer. Given such A, V, and k, this function constructs the generalized Reed–Solomon code $GRS_k(A, V)$ over K. This is an $[n, k' \leq k]$ code. For details see [MS78, p.303].

```
JustesenCode(N, K)
```

Given an integer N such that $N = 2^m - 1$ and a positive integer K, construct the binary linear Justesen code of length $2mN$ and dimension mK. For details see [MS78, p.307].

Example H158E30

We construct a generalized Reed–Solomon code over GF(2) based on sequences of elements in the extension field GF(2^3) of GF(2). The parameter k is taken to be 3, so the dimension 3 is at most k.

```
> q := 2^3;
> K<w> := GF(q);
> A := [w ^ i : i in [0 .. q - 2]];
> V := [K ! 1 : i in [0 .. q - 2]];
> k := 3;
> C := GRSCode(A, V, k);
[7, 3, 5] GRS code over GF(2^3)
Generator matrix:
[  1   0   0 w^3   w   1 w^3]
[  0   1   0 w^6 w^6   1 w^2]
[  0   0   1 w^5 w^4   1 w^4]
```

158.9.5 Maximum Distance Separable Codes

`MDSCode(K, k)`

Given a finite field $GF(q = 2^m)$, this function constructs the $[q+1, k, q-k+2]$ maximum distance separable code.

158.10 New Codes from Existing

The operations described here produce a new code by modifying in some way the codewords of a given code.

158.10.1 Standard Constructions

`AugmentCode(C)`

Given an $[n, k]$ binary code C, construct a new code C' by including the all-ones vector with the words of C (provided that it is not already in C).

`CodeComplement(C, C1)`

Given a subcode $C1$ of C, return a code $C2$ such that $C = C1 + C2$.

`DirectSum(C, D)`

Given an $[n_1, k_1]$ code C and an $[n_2, k_2]$ code D, both over the same field F, construct the direct sum of C and D. The direct sum consists of all vectors $u|v$, where $u \in C$ and $v \in D$.

`DirectSum(Q)`

Given a sequence of codes $Q = [C_1, \ldots, C_r]$, all defined over the same field F, construct the direct sum of the C_i.

`DirectProduct(C, D)`

`ProductCode(C, D)`

Given an $[n_1, k_1]$ code C and an $[n_2, k_2]$ code D, both over the same ring R, construct the direct product of C and D. The direct product has length $n_1 \cdot n_2$, dimension $k_1 \cdot k_2$, and its generator matrix is the Kronecker product of the basis matrices of C and D.

`ExtendCode(C)`

Given an $[n, k, d]$ code C form a new code C' from C by adding the appropriate extra coordinate to each vector of C such that the sum of the coordinates of the extended vector is zero. (Thus if C is a binary code, the construction will add a 0 at the end of every codeword having even weight, and a 1 at the end of every codeword having odd weight.)

`ExtendCode(C, n)`

Return the code C extended n times.

`PadCode(C, n)`

Add n zeros to the end of each codeword of C.

`ExpurgateCode(C)`

Construct a new code by deleting all the code words of C having odd weight.

`ExpurgateCode(C, L)`

The sequence L consists of codewords from C. The result is obtained by deleting the words in L from C.

`ExpurgateWeightCode(C, w)`

Delete a subspace generated by a word of weight w.

`LengthenCode(C)`

Given an $[n, k]$ binary code C, construct a new code by first adding the all-ones codeword, and then extending it by adding an overall parity check.

`PlotkinSum(C1, C2)`

Given two codes over the same alphabet, return the code consisting of all vectors of the form $u|u + v$, where $u \in C1$ and $v \in C2$. Zeros are appended where needed to make up any length differences in the two codes. The result is a $[n_1 + \max\{n_1, n_2\}, k_1 + k_2, \min\{2 * d_1, d_2\}]$ code.

`PlotkinSum(C1, C2, C3:` *parameters*`)`

`a` FldFinElt *Default :* -1

Given three codes over the same alphabet, return the code consisting of all vectors of the form $u|u + a * v|u + v + w$, where $u \in C1$, $v \in C2$ and $w \in C3$. Zeros are appended where needed to make up any length differences in the three codes. The result for the default case of `a := -1` is a $[n_1 + \max\{n_1, n_2\} + \max\{n_1, n_2, n_3\}, k_1 + k_2 + k_3, \min\{3 * d_1, 2 * d_2, d_3\}]$ code.

`PunctureCode(C, i)`

Given an $[n, k]$ code C, and an integer i, $1 \leq i \leq n$, construct a new code C' by deleting the i-th coordinate from each code word of C.

`PunctureCode(C, S)`

Given an $[n, k]$ code C and a set S of distinct integers $\{i_1, \cdots, \mathrm{i}_r\}$ each of which lies in the range $[1, n]$, construct a new code C' by deleting the components $i_1, \cdots, i_r$ from each code word of C.

`ShortenCode(C, i)`

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

`ShortenCode(C, S)`

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

Example H158E31

Using only two simple `RepetitionCode`'s and several standard constructions, we create a $[12, 4, 6]$ code. This is the best possible minimum weight for a code of this length and dimension, as is the minimum weight for all codes produced in this example.
Instead of printing each individual code out, the codes are named by convention as c_n_k_d, where n, k, d represent the `Length`,`Dimension` and `MinimumWeight` respectively.

```
> c_4_1_4 := RepetitionCode(GF(2),4);
> c_6_1_6 := RepetitionCode(GF(2),6);
> c_4_3_2 := Dual( c_4_1_4 );
> c_8_4_4 := PlotkinSum( c_4_3_2 , c_4_1_4 );
> c_7_4_3 := PunctureCode( c_8_4_4 , 8 );
> c_6_3_3 := ShortenCode(  c_7_4_3 , 7 );
> c_12_4_6 := PlotkinSum( c_6_3_3 , c_6_1_6 );
> c_12_4_6;
[12, 4, 6] Linear Code over GF(2)
Generator matrix:
[1 0 0 1 1 0 0 1 1 0 0 1]
[0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 1 1 1 1 0 0 1 1 1 1]
[0 0 0 0 0 0 1 1 1 1 1 1]
```

158.10.2 Changing the Alphabet of a Code

```
ExtendField(C, L)
```

Given an $[n, k, d]$ code C defined over the finite field K, and an extension field L of K, construct the code C' over L corresponding to C. The function also returns the embedding map from C into C'.

```
LinearCode(C, S)
```

Given an $[n, k, d]$ code C defined over the finite field K, and a subfield S of K such that the degree of K over S is m, construct a new $[N = mn, k, D \geq d]$ code C' over S by replacing each component of each codeword of C by its representation as a vector over S. The function also returns the isomorphism from C onto C'.

```
SubfieldRepresentationCode(C, K)
```

Given a linear code over $GF(q^m)$, return a code whose codewords are obtained from those of C by expanding each coordinate in $GF(q^m)$ as a vector of dimension m over $K = GF(q)$.

```
SubfieldRepresentationParityCode(C, K)
```

Given a linear code over $GF(q^m)$, return a code whose codewords are obtained from those of C by expanding each coordinate in $GF(q^m)$ as a vector of dimension $m + 1$ over $K = GF(q)$, including a parity check bit.

```
SubfieldSubcode(C, S)
RestrictField(C, S)
SubfieldSubcode(C)
RestrictField(C)
```

Given an $[n, k, d]$ code C defined over the field K, and a subfield S of K, construct a new $[n, K \leq k, D \geq d]$ code C' over S consisting of the codewords of C which have all their components in S. If S is omitted, it is taken to be the prime subfield of K. The function also returns the restriction map from C to the subfield subcode C'.

```
SubfieldCode(C, S)
```

Given an $[n, k]$ code C defined over the field K, and a subfield S of K, such that K is a degree m extension of S, construct a new $[n, k']$ code over S by expanding each element of K as a column vector over S. The new code will have $k' \leq km$.

```
Trace(C, F)
Trace(C)
```

Given a code C defined over the field K, and a subfield F of K, construct a new code C' over F consisting of the traces with respect to F of each of the codewords of L. If F is omitted, it is taken to be the prime subfield of K.

158.10.3 Combining Codes

`C1 cat C2`

Given codes $C1$ and $C2$, both defined over the same field K, return the concatenation C of $C1$ and $C2$. If A and B are the generator matrices of $C1$ and $C2$, respectively, the concatenation of $C1$ and $C2$ is the code with generator matrix whose rows consist of each row of A concatenated with each row of B.

`Juxtaposition(C1, C2)`

Given an $[n_1, k, d_1]$ code $C1$ and an $[n_2, k, d_2]$ code $C2$ of the same dimension, where both codes are defined over the same field K, the function returns a $[n_1 + n_2, k, \geq d_1 + d_2]$ code whose generator matrix is `HorizontalJoin(A, B)`, where A and B are the generator matrices for codes $C1$ and $C2$, respectively.

`ConcatenatedCode(O, I)`

Given a $[N, K, D]$-code O defined over $GF(q^k)$ and a $[n, k, d]$-code I defined over $GF(q)$, construct the $[Nn, Kk, \delta \geq dD]$ concatenated code by taking O as outer code and I as inner code.

Example H158E32

We use the function `ConcatenatedCode` to construct a $[69, 19, 22]$ code over $GF(2)$, this being the best known code for this length and dimension. While it is theoretically possible for a code of minimum weight up to 24 to exist, the best binary $[69, 19]$ code at the time of writing (July 2001) has minimum weight 22.

```
> C1 := ShortenCode( QRCode(GF(4),29) , {24..29} );
> C1:Minimal;
[23, 9] Linear Code over GF(2^2)
> C2 := ConcatenatedCode( C1 , CordaroWagnerCode(3) );
> C2:Minimal;
[69, 18] Linear Code over GF(2)
> res := C2 + RepetitionCode(GF(2),69);
> res:Minimal;
[69, 19] Linear Code over GF(2)
> MinimumWeight(res);
22
> res:Minimal;
[69, 19, 22] Linear Code over GF(2)
```

```
ConstructionX(C1, C2, C3)
```

Let C_1, C_2 and C_3 be codes with parameters $[n_1, k_1, d_1]$, $[n_2, k_2, d_2]$ and $[n_3, k_3, d_3]$, respectively, where $C2$ is a union of b cosets of $C1$, (so $n_1 = n_2$ and $k_2 \leq k_1$) and $k_1 = k_2 + k_3$. The construction divides $C2$ into a union of cosets of $C1$ and attaches a different codeword of $C3$ to each coset. The new code has parameters $[n_1 + n_3, k_1, \geq \min\{\ d_2, d_1 + d_3\}]$. For further details see [MS78, p.581].

```
ConstructionXChain(S, C)
```

Given a sequence of codes S where all codes are subcodes of the first one, apply `ConstructionX` to $S[1], S[2]$ and C. Then compute the resulting subcodes from the other codes in S.

Example H158E33

We create a $[161, 29, 53]$ code by applying construction X to two BCH codes of length 127. To maximise the resulting minimum weight we take the best known $[34, 14]$ code as C_3. The construction sets a lower bound on the minimum weight, making the calculation of the true minimum weight much faster.

```
> SetPrintLevel("Minimal");
>
> C1 := BCHCode(GF(2), 127, 43);
> C2 := BCHCode(GF(2), 127, 55);
> C3 := BKLC(GF(2), 34, 14);
> C1; C2; C3;
[127, 29, 43] BCH code (d = 43, b = 1) over GF(2)
[127, 15, 55] BCH code (d = 55, b = 1) over GF(2)
[34, 14, 10] Linear Code over GF(2)
> CX := ConstructionX(C1, C2, C3);
> CX;
[161, 29] Linear Code over GF(2)
> time MinimumWeight(CX);
53
Time: 0.010
```

```
ConstructionX3(C1, C2, C3, D1, D2)
```

Given a chain of codes $C1 = [n, k_1, d_1]$, $C2 = [n, k_2, d_2]$, and $C3 = [n, k_3, d_3]$ with $k_3 < k_2 < k_1$, and suffix codes $D1 = [n_1, k_1 - k_2, e_1]$ and $D2 = [n_2, k_3 - k_2, e_2]$, construct a code $C = [n + n_1 + n_2, k_1, \geq min\{d_3, d_1 + e_1, d_2 + e_2\}$. For further details see [MS78, p.583].

```
ConstructionX3u(C1, C2, C3, D1, D2)
```

Given two chains of codes $C1 = [n, k_1] \subset C2 = [n, k_2] \subset C3 = [n, k_3]$ and $D1 = [n', k_1 - k_3] \subset D2 = [n', k_2 - k_3]$, return the codes $C = [n + n', k_1] \subset C' = [n + n', k_2]$ using Construction X with $C1, C3$ and $D1$ resp. $C2, C3$ and $D2$.

Example H158E34

We construct a best known $[74, 43, 11]$ code using construction X3. From a chain of BCH subcodes, we take an subcode to get the appropriate length, then use construction X3 with the best possible codes $D1, D2$. Because the construction algorithm sets a lower bound on the minimum weight, then it is quick to calculate afterwards.

```
> SetPrintLevel("Minimal");
> C1 := ExtendCode( BCHCode(GF(2), 63, 7) );
> C2 := ExtendCode( BCHCode(GF(2), 63, 9) );
> C3 := ExtendCode( BCHCode(GF(2), 63, 11) );
> C1; C2; C3;
[64, 45, 8] Linear Code over GF(2)
[64, 39, 10] Linear Code over GF(2)
[64, 36, 12] Linear Code over GF(2)
> CC := SubcodeBetweenCode(C1, C2, 43);
> CC;
[64, 43] Linear Code over GF(2)
> MinimumWeight(CC);
8
> CX3 := ConstructionX3(CC, C2, C3,
>                 BKLC(GF(2), 7, 4), BKLC(GF(2), 3, 3));
> CX3;
[74, 43] Linear Code over GF(2)
> time MinimumWeight(CX3);
11
Time: 0.000
```

```
ConstructionXX(C1, C2, C3, D2, D3)
```

Let the parameters of codes C_1, C_2, C_3 be $[n_1, k, d_1], [n_1, k - l_2, d_2]$ and $[n_1, k - l_3, d_3]$ respectively, where C_2 and C_3 are subcodes of C_1. Codes D_2, D_3 must have dimensions l_2, l_3, with parameters $[n_2, l_2, \delta_2], [n_3, l_3, \delta_3]$ say. The construction breaks C_1 up into cosets of C_2 and C_3 with the relevant tails added from D_2 and D_3. If the intersection of C_1 and C_2 has minimum distance d_0 then the newly constructed code will have parameters $[n_1 + n_2 + n_3, k, min\{d_0, d_2 + \delta_2, d_3 + \delta_3, d_1 + \delta_2 + \delta_3\}]$. For further details see [All84].

Example H158E35

We construct a best known $[73, 38, 13]$ code C using construction XX. For C_1, C_2, C_3 we take three cyclic (or BCH) codes of length 63, while for D_1, D_2 we use two Best Known Codes.

```
> SetPrintLevel("Minimal");
> C1 := BCHCode(GF(2),63,10,57);
> P<x> := PolynomialRing(GF(2));
> p := x^28 + x^25 + x^22 + x^21 + x^20 + x^17 + x^16
>      + x^15 + x^9 + x^8 + x^6 + x^5 + x + 1;
> C2 := CyclicCode(63, p);
> C3 := BCHCode(GF(2), 63, 10, 58);
> C1; C2; C3;
[63, 38] BCH code (d = 10, b = 57) over GF(2)
[63, 35] Cyclic Code over GF(2)
[63, 32] BCH code (d = 10, b = 58) over GF(2)
> MinimumDistance(C1 meet C2);
12
```

So the minimum distance of the code produced by Construction XX must be at least 12.

```
> C := ConstructionXX(C1, C2, C3, BKLC(GF(2),3,3), BKLC(GF(2),7,6) );
> C;
[73, 38] Linear Code over GF(2)
MinimumDistance(C);
13
```

Thus the actual minimum distance is one greater than the lower bound guaranteed by Construction XX.

`ZinovievCode(I, O)`

The arguments are as follows: The first argument must be a sequence I containing an increasing chain of r codes with parameters,

$$[n, k_1, d_1]_q \subset [n, k_2, d_2]_q \subset \ldots \subset [n, k_r, d_r]_q$$

where $0 = k_0 < k_1 < k_2 < \ldots < k_r$, (the *inner codes*). The second argument must be a sequence O of r codes with parameters $[N, K_i, D_i]_{Q_i}$, where $Q_i = q^{e_i}$ and $e_i = k_i - k_{i-1}$ for $i = 1 \ldots r$ (the *outer codes*). The function constructs a generalised concatenated $[n * N, K, D]_q$ code is constructed, where $K = e_1K_1 + \ldots + e_rK_r$ and $D = min(d_1D_1, \ldots, d_rD_r)$. For further details see [MS78, p.590].

Example H158E36

We create a $[72, 41, 12]$ code over $GF(2)$ using the `ZinovievCode` function, which is the best known code for this length and dimension. While it is theoretically possible for a $[72, 41]$ code to have minimum weight up to 14, at the time of writing (July 2001) the best known code has minimum weight 12. The minimum weight is not calculated since it is a lengthy calculation.

```
> I1 := RepetitionCode(GF(2),8);
> I2 := I1 + LinearCode( KMatrixSpace(GF(2),3,8) !
>                 [0,1,0,0,0,1,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1]
>                      );
> I3 := Dual(I1);
> Inner := [I1, I2, I3];
> Inner:Minimal;
[
    [8, 1, 8] Cyclic Code over GF(2),
    [8, 4, 4] Linear Code over GF(2),
    [8, 7, 2] Cyclic Code over GF(2)
]
>
> O1 := Dual(RepetitionCode(GF(2),9));
> O2 := BCHCode(GF(8),9,3,4);
> O3 := BCHCode(GF(8),9,6,7);
> Outer := [O1, O2, O3];
> Outer:Minimal;
[
    [9, 8, 2] Cyclic Code over GF(2),
    [9, 7, 3] BCH code (d = 3, b = 4) over GF(2^3),
    [9, 4, 6] BCH code (d = 6, b = 7) over GF(2^3)
]
>
> C := ZinovievCode(Inner, Outer);
> C:Minimal;
[72, 41] Linear Code over GF(2)
```

`ConstructionY1(C)`

Apply construction $Y1$ to the code C. This construction applies the shortening operation at the positions in the support of a word of minimal weight in the dual of C. If C is a $[n, k, d]$ code, whose dual code has minimum weight d', then the returned code has parameters $[n - d', k - d' + 1, \geq d]$. For further details see [MS78, p.592].

`ConstructionY1(C, w)`

Apply construction $Y1$ to the code C. This construction applies the shortening operation at the positions in the support of a word of weight w in the dual of C. If C is a $[n, k, d]$ code, then the returned code has parameters $[n - w, k - w + 1, \geq d]$.

158.11 Coding Theory and Cryptography

One of the few public-key cryptosystems which does not rely on number theory is the McEliece cryptosystem, whose security depends on coding theory. An attack on the McEliece cryptosystem must determine the coset leader (of known weight) from a user defined error coset. In general it is assumed that the code in question has no known structure, and it treated as a random code.

The best known attacks on the McEliece cryptosystem are a series of probabilistic enumeration-based algorithms.

158.11.1 Standard Attacks

```
McEliecesAttack(C, v, e)
```

MaxTime	FldReElt	*Default :* ∞
DirectEnumeration	BoolElt	*Default :* true

Perform the original decoding attack described by McEliece when he defined his cryptosystem. Random information sets are tested for being disjoint for the support of the desired error vector. This intrinsic attempts to enumerate a vector of weight e from the error coset $v + C$ for the given vector v.

If set to a non-zero positive value, the variable argument `MaxTime` aborts the computation if it goes to long. The argument `DirectEnumeration` controls whether or not the coset is enumerated directly, or whether the larger code generated by $\langle C, v\rangle$ is enumerated.

```
LeeBrickellsAttack(C, v, e, p)
```

MaxTime	FldReElt	*Default :* ∞
DirectEnumeration	BoolElt	*Default :* true

Perform the decoding attack described by Lee and Brickell. Random information sets are tested for having weight less than or equal to p. For most sized codes, the optimal input parameter for this attack is $p = 2$. This intrinsic attempts to enumerate a vector of weight e from the error coset $v + C$ for the given vector v.

If set to a non-zero positive value, the variable argument `MaxTime` aborts the computation if it goes to long. The argument `DirectEnumeration` controls whether or not the coset is enumerated directly, or whether the larger code generated by $\langle C, v\rangle$ is enumerated.

```
LeonsAttack(C, v, e, p, l)
```

MaxTime	FldReElt	*Default :* ∞
DirectEnumeration	BoolElt	*Default :* true

Perform the decoding attack described by Leon. For random information sets of size k, a punctured code of length $k + l$ is investigated for codewords of weight less than or equal to p. For small codes (up to length around 200), the optimal input parameter for this attack is $p = 2$ with l somewhere in the range $3 - 6$. For larger code $p = 3$ can sometimes be faster, with values of l in the range $7 - 10$. This intrinsic attempts to enumerate a vector of weight e from the error coset $v + C$ for the given vector v.

If set to a non-zero positive value, the variable argument `MaxTime` aborts the computation if it goes to long. The argument `DirectEnumeration` controls whether or not the coset is enumerated directly, or whether the larger code generated by $\langle C, v\rangle$ is enumerated.

```
SternsAttack(C, v, e, p, l)
```

MaxTime	FldReElt	*Default :* ∞
DirectEnumeration	BoolElt	*Default :* true

Perform the decoding attack described by Stern. For random information sets of size k, a punctured code of length $k+l$ is split into two subspaces. Each subspace is enumerated up to information weight p and collisions found with zero non-information weight. For small to mid-range codes (up to length around 500), the optimal input parameter for this attack is $p = 2$ with l somewhere in the range $9 - 13$. For larger code $p = 3$ can sometimes be faster, with values of l from 20 to much higher. This intrinsic attempts to enumerate a vector of weight e from the error coset $v + C$ for the given vector v.

If set to a non-zero positive value, the variable argument `MaxTime` aborts the computation if it goes to long. The argument `DirectEnumeration` controls whether or not the coset is enumerated directly, or whether the larger code generated by $\langle C, v\rangle$ is enumerated.

```
CanteautChabaudsAttack(C, v, e, p, l)
```

MaxTime	FldReElt	*Default :* ∞
DirectEnumeration	BoolElt	*Default :* true

Perform the decoding attack described by Canteaut and Chabaud. For random information sets of size k, a punctured code of length $k+l$ is split into two subspaces. Using the enumeration technique identical to that of Stern's attack, a different linear algebra process steps through information sets more quickly. The price for this is less independent information sets. This intrinsic attempts to enumerate a vector of weight e from the error coset $v + C$ for the given vector v.

For most codes (up to length around 1000), the optimal input parameter for this attack is $p = 1$ with l somewhere in the range $6 - 9$. For very large codes $p = 2$ can sometimes be faster, with values of l from 20 to much higher.

If set to a non-zero positive value, the variable argument `MaxTime` aborts the computation if it goes to long. The argument `DirectEnumeration` controls whether or not the coset is enumerated directly, or whether the larger code generated by $\langle C, v\rangle$ is enumerated.

158.11.2 Generalized Attacks

All of the decoding attacks on the McEliece cryptosystem can be put into a uniform framework, consisting of repeated operation of a two stage procedure. Magma allows the user to choose any combination of the implemented methods, which include improvements on the standard attacks.

```
DecodingAttack(C, v, e)
```

Enumeration	MonStgElt	*Default :* "*Standard*"
MatrixSequence	MonStgElt	*Default :* "*Random*"

`NumSteps`	RngIntElt	*Default :* 1
`p`	RngIntElt	*Default :* 2
`l`	RngIntElt	*Default :*
`MaxTime`	FldReElt	*Default :* ∞
`DirectEnumeration`	BoolElt	*Default :* `true`

Perform a generalized decoding attack by specifying the enumeration and matrix sequence procedures to be used. This intrinsic attempts to enumerate a vector of weight e from the error coset $v + C$ for the given vector v.

The parameter `Enumeration` can take the values `"Standard"`, `"Leon"` or `"HashTable"`, and correspond to the methods used in Lee and Brickells, Leons and Sterns attacks respectively.

The parameter `MatrixSequence` can take on the values `"Random"` or `"Stepped"`, corresponding to either a completely random sequence of information sets or a sequence of sets differing in one place.

The integer valued `NumSteps` offers a generalization of the stepped matrix process, taking a sequence of sets which differ at the specified number of places.

The parameter `p` and `l` describe the enumeration process, and their exact meaning depends on the enumeration process in question. See the earlier descriptions of the standard attacks for a full description of their meanings.

For codes of lengths anywhere between $500 - 1000$, the best performance can be obtained using a multiply stepped matrix sequence, using around 10 steps at a time. This is in conjunction with the hashtable enumeration technique using $p = 2$ and l in the range $15 - 20$.

If set to a non-zero positive value, the variable parameter `MaxTime` aborts the computation if it goes to long. The parameter `DirectEnumeration` controls whether or not the coset is enumerated directly, or whether the larger code generated by $\langle C, v\rangle$ is enumerated.

158.12 Bounds

Magma supplies various functions for computing lower and upper bounds for parameters associated with codes. It also contains tables of best known bounds for linear codes. The functions in this section only apply to codes over finite fields.

158.12.1 Best Known Bounds for Linear Codes

A Magma database allows the user access to tables giving the best known upper and lower bounds of the `Length`, `Dimension`, and `MinimumWeight` of linear codes. Tables are currently available relating to codes over $GF(2)$ and $GF(4)$ with $1 \leq$ `Length` ≤ 256, over $GF(3)$ with $1 \leq$ `Length` ≤ 243, over $GF(5), GF(8)$, and $GF(9)$ with $1 \leq$ `Length` ≤ 130, and over $GF(7)$ with $1 \leq$ `Length` ≤ 100.

`BKLCLowerBound(F, n, k)`

Returns the best known lower bound on the maximum possible minimum weight of a linear code over finite field F having length n and dimension k.

`BKLCUpperBound(F, n, k)`

Returns the best known upper bound on the minimum weight of a linear code over finite field F of length n and dimension k.

`BLLCLowerBound(F, k, d)`

Returns the best known lower bound on the minimum possible length of a linear code over finite field F having dimension k and minimum weight at least d. If the required length is out of the range of the database then no bound is available and -1 is returned.

`BLLCUpperBound(F, k, d)`

Returns the best known upper bound on the minimum possible length of a linear code over finite field F of dimension k and minimum weight at least d. If the required length is out of the range of the database then no bound is available and -1 is returned.

`BDLCLowerBound(F, n, d)`

Returns the best known lower bound on the maximum possible dimension of a linear code over finite field F having length n and minimum weight at least d.

`BDLCUpperBound(F, n, d)`

Returns the best known upper bound on the dimension of a linear code over finite field F having length n and minimum weight at least d.

158.12.2 Bounds on the Cardinality of a Largest Code

`EliasBound(K, n, d)`

Return the Elias upper bound of the cardinality of a largest code of length n and minimum distance d over the field K.

`GriesmerBound(K, n, d)`

Return the Griesmer upper bound of the cardinality of a largest code of length n and minimum distance d over the field K.

`JohnsonBound(n, d)`

Return the Johnson upper bound of the cardinality of a largest binary code of length n and minimum distance d.

`LevenshteinBound(K, n, d)`

Return the Levenshtein upper bound of the cardinality of a largest code of length n and minimum distance d over the field K.

```
PlotkinBound(K, n, d)
```

Return the Plotkin upper bound on the cardinality of a (possibly non-linear) code of length n and minimum distance d over the field K. The bound is formed by calculating the maximal possible average distance between codewords.

For binary codes the bound exists for $n \leq 2d$, (d even), or $n \leq 2d + 1$ (d odd). For codes over general fields the bound exists for $d > (1 - 1/\#K) * n$.

```
SingletonBound(K, n, d)
```

Return the Singleton upper bound of the cardinality of a largest code of length n and minimum distance d over the field K.

```
SpherePackingBound(K, n, d)
```

Return the Hamming sphere packing upper bound on the cardinality of a largest codes of length n and minimum distance d over the field K.

```
GilbertVarshamovBound(K, n, d)
```

Return the Gilbert–Varshamov lower bound of the cardinality of a largest code (possibly non-linear) of length n and minimum distance d over the field K.

```
GilbertVarshamovLinearBound(K, n, d)
```

Return the Gilbert–Varshamov lower bound of the cardinality of a largest linear code of length n and minimum distance d over the field K.

```
VanLintBound(K, n, d)
```

Return the van Lint lower bound of the cardinality of a largest code of length n and minimum distance d over the field K.

Example H158E37

We compare computed and stored values of best known upper bounds of the dimension of binary linear codes of length 20. The cardinality of a linear code of dimension k over $\mathbf{F}_q$ is q^k, and so the computed bounds on cardinality are compared with the stored bounds on dimension by taking logs.

```
> n:=20;
> K := GF(2);
> [ Ilog(#K, Minimum({GriesmerBound(K, n, d), EliasBound(K, n, d),
>                       JohnsonBound(n, d) , LevenshteinBound(K, n, d),
>                       SpherePackingBound(K, n, d)})) : d in [1..n] ];
[ 20, 19, 15, 14, 12, 11, 9, 8, 5, 4, 3, 2, 2, 1, 1, 1, 1, 1, 1, 1 ]
> [ BDLCUpperBound(K, n, d) : d in [1..n] ];
[ 20, 19, 15, 14, 11, 10, 9, 8, 5, 4, 3, 2, 2, 1, 1, 1, 1, 1, 1, 1 ]
```

158.12.3 Bounds on the Minimum Distance

```
BCHBound(C)
```

Given a cyclic code C, return the BCH bound for C. This a lower bound on the minimum weight of C.

```
GriesmerMinimumWeightBound(K, n, k)
```

Return the Griesmer upper bound of the minimum weight of a linear code of length n and dimension k over the field K.

158.12.4 Asymptotic Bounds on the Information Rate

```
EliasAsymptoticBound(K, delta)
```

Return the Elias asymptotic upper bound of the information rate for δ in $[0, 1]$ over the field K.

```
McElieceEtAlAsymptoticBound(delta)
```

Return the McEliece–Rodemich–Rumsey–Welch asymptotic upper bound of the binary information rate for δ in $[0, 1]$.

```
PlotkinAsymptoticBound(K, delta)
```

Return the Plotkin asymptotic upper bound of the information rate for δ in $[0, 1]$ over the field K.

```
SingletonAsymptoticBound(delta)
```

Return the Singleton asymptotic upper bound of the information rate for δ in $[0, 1]$ over any finite field.

```
HammingAsymptoticBound(K, delta)
```

Return the Hamming asymptotic upper bound of the information rate for δ in $[0, 1]$ over the field K.

```
GilbertVarshamovAsymptoticBound(K, delta)
```

Return the Gilbert–Varshamov asymptotic lower bound of the information rate for δ in $[0, 1]$ over the field K.

158.12.5 Other Bounds

```
GriesmerLengthBound(K, k, d)
```

Return the Griesmer lower bound of the length of a linear code of dimension k and minimum distance d over K.

158.13 Best Known Linear Codes

An $[n, k]$ linear code C is said to be a *best known linear* $[n, k]$ *code* (BKLC) if C has the highest minimum weight among all known $[n, k]$ linear codes.

An $[n, k]$ linear code C is said to be an *optimal linear* $[n, k]$ *code* if the minimum weight of C achieves the theoretical upper bound on the minimum weight of $[n, k]$ linear codes.

Magma currently has databases for best known linear codes over $GF(q)$ for $q = 2, 3, 4, 5, 7, 8, 9$. There is also a database of best known quantum codes that can be found in Chapter 164. The database for codes over $GF(2)$ contains constructions of best codes of length up to $n_{\mathrm{max}} = 256$. The codes of length up to $n_{\mathrm{opt}} = 31$ are optimal. The database is complete in the sense that it contains a construction for every set of parameters. Thus the user has access to 33 152 best-known binary codes.

The database for codes over $GF(3)$ contains constructions of best codes of up to length $n_{\mathrm{max}} = 243$. The codes of length up to $n_{\mathrm{opt}} = 21$ are optimal. The database is complete up to length $n_{\mathrm{complete}} = 100$. Many of the codes constructed in this database are vast improvements on the previously known bounds for best codes over $GF(3)$. The database of codes over $GF(3)$ is a contribution of Markus Grassl, Karlsruhe.

The database for codes over $GF(4)$ contains constructions of best codes of up to length $n_{\mathrm{max}} = 256$. The codes of length up to $n_{\mathrm{opt}} = 18$ are optimal. The database is over 65% complete with the first missing code coming at length 98. Many of the codes constructed in this database are vast improvements on the previously known bounds for best codes over $GF(4)$.

Similar databases for other small fields have been added in V2.14. They are contributions of Markus Grassl, Karlsruhe. The statistics of all databases are summarised in the following table.

	$GF(2)$	$GF(3)$	$GF(4)$	$GF(5)$	$GF(7)$	$GF(8)$	$GF(9)$
n_{max}	256	243	256	130	100	130	130
n_{opt}	31	21	18	15	14	14	16
n_{complete}	256	100	97	80	68	76	93
total	33 152	29 889	33 152	8 645	5 150	8 645	8 645
missing	0	6 545	11 379	527	381	1 763	1 333
filled	100%	78.10%	65.67%	93.90%	92.60%	79.61%	84.58%

Compared to previous released versions of the Magma BKLC database, 1308 codes over $GF(2)$, 102 codes over $GF(3)$ and 160 codes over $GF(4)$ have been improved, and the maximal length for codes over $GF(3)$ and $GF(4)$ has been increased to 243 and 256, respectively.

Best known upper and lower bounds on the minimum weight for $[n, k]$ linear codes are also available (see section 158.12.1).

The Magma BKLC database makes use of the tables of bounds compiled by A. E. Brouwer [Bro98]. The online version of these tables [Bro] has been discontinued. Similar tables are now maintained by Markus Grassl [Gra]. Any improvements, errors, or problems with the Magma BKLC database should be reported to `codes@codetables.de`.

It should be noted that the Magma BKLC database is unrelated to the similar (but rather incomplete) BKLC database forming part of GUAVA, a share package in GAP3.

A significant number of entries in the Magma BKLC database provide better codes than the corresponding ones listed in Brouwer's tables.

The construction of the Magma BKLC database has been undertaken by John Cannon (Sydney), Markus Grassl (Karlsruhe) and Greg White (Sydney). The authors wish to express their appreciation to the following people who generously supplied codes, constructions or other assistance: Nuh Aydin, Anton Betten, Michael Braun, Iliya Bouyukliev, Andries Brouwer, Tat Chan, Zhi Chen, Rumen Daskalov, Scott Duplichan, Iwan Duursma, Yves Edel, Sebastian Egner, Peter Farkas, Damien Fisher, Philippe Gaborit, Willi Geiselmann, Stephan Grosse, Aaron Gulliver, Masaaki Harada, Ray Hill, Plamen Hristov, David Jaffe, Axel Kohnert, San Ling, Simon Litsyn, Pawel Lizak, Tatsuya Maruta, Masami Mohri, Masakatu Morii, Harald Niederreiter, Ayoub Otmani, Fernanda Pambianco, James B. Shearer, Neil Sloane, Roberta Sabin, Cen Tjhai, Ludo Tolhuizen, Martin Tomlinson, Gerard van der Geer, Henk van Tilborg, Chaoping Xing, Karl-Heinz Zimmermann, Johannes Zwanzger.

Given any two of the parameters: length, dimension, and minimum weight, then Magma will return the code with the best possible value of the omitted parameter. Given a specified length and minimum weight, for example, will result in a corresponding code of maximal possible dimension.

The user can display the method used to construct a particular BKLC code through use of a verbose mode, triggered by the verbose flag `BestCode`. When it is set to `true`, all of the functions in this section will output the steps involved in each code they construct. While some codes are defined by stored generator matrices, and some use constructions which are not general enough, or safe enough, to be available to the user, most codes are constructed using standard Magma functions. Note that having the verbose flag `Code` set to `true` at the same time can produce mixed and confusing output, since the database uses functions which have verbose outputs dependent on this flag.

```
BKLC(K, n, k)
BestKnownLinearCode(K, n, k)
```

Given a finite field K, a positive integer n, and a non-negative integer k such that $k \leq n$, return an $[n, k]$ linear code over K which has the largest minimum weight among all known $[n, k]$ linear codes. A second boolean return value signals whether or not the desired code exists in the database.

The databases currently available are over $GF(q)$ for $q = 2, 3, 4, 5, 7, 8, 9$ of length up to $n_{\max}$ as given in the table above.

If the verbose flag `BestCode` is set to true then the method used to construct the code will be printed.

```
BLLC(K, k, d)
BestLengthLinearCode(K, k, d)
```

Given a finite field K, and positive integers k and d, return a linear code over K with dimension k and minimum weight at least d which has the shortest length among known codes. A second boolean return value signals whether or not the desired code exists in the database.

The databases currently available are over $GF(q)$ for $q = 2, 3, 4, 5, 7, 8, 9$ of length up to $n_{\max}$ as given in the table above.

If the verbose flag `BestCode` is set to true then the method used to construct the code will be printed.

```
BDLC(K, n, d)
```

```
BestDimensionLinearCode(K, n, d)
```

Given a finite field K, a positive integer n, and a positive integer d such that $d \leq n$, return a linear code over K with length n and minimum weight $\geq d$ which has the largest dimension among known codes. A second boolean return value signals whether or not the desired code exists in the database.

The databases currently available are over $GF(q)$ for $q = 2, 3, 4, 5, 7, 8, 9$ of length up to $n_{\max}$ as given in the table above.

If the verbose flag `BestCode` is set to true then the method used to construct the code will be printed.

Example H158E38

We look at some best known linear codes over $GF(2)$. Since the database over $GF(2)$ is completely filled, we can ignore the second boolean return value.

```
> C := BKLC(GF(2),23,12);
> C;
[23, 12, 7] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1]
[0 0 0 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1]
[0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 1 1 1 0 0 0 1 1]
[0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1]
[0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 0 1]
> WeightDistribution(C);
[ <0, 1>, <7, 253>, <8, 506>, <11, 1288>, <12, 1288>, <15, 506>,
<16, 253>, <23, 1> ]
> BKLCLowerBound(GF(2),23,12), BKLCUpperBound(GF(2),23,12);
7 7
```

So we see that this code is optimal, in the sense that it meets the best known upper bound on its minimum weight. (All best known binary codes of length up to 31 are optimal).

However larger best known codes are not optimal, making it theoretically possible that better codes exist.

```
> C := BKLC(GF(2),145,36);
> C:Minimal;
[145, 36, 42] Linear Code over GF(2)
> BKLCLowerBound(GF(2),145,36), BKLCUpperBound(GF(2),145,36);
42 52
```

Example H158E39

We look at some best known codes over $GF(4)$. Since this database is only approximately 66% complete, it is necessary to check the second boolean return value to know if the database contained the desired code.

```
> F<w> := GF(4);
> C, has_code := BKLC(F, 14, 9);
> has_code;
true
> C;
[14, 9, 4] Linear Code over GF(2^2)
Generator matrix:
[  1   0   0   0   0   0   0   0   0   0   0 w^2   w   1]
[  0   1   0   0   0   0   0   0   0   0   1   1   1 w^2]
[  0   0   1   0   0   0   0   0   0   0 w^2   w   0   w]
[  0   0   0   1   0   0   0   0   0   0   w   1   1   w]
[  0   0   0   0   1   0   0   0   0   0   w   0   w w^2]
[  0   0   0   0   0   1   0   0   0   0 w^2   1   1   1]
[  0   0   0   0   0   0   1   0   0   0   1   w w^2   0]
[  0   0   0   0   0   0   0   1   0   0   0   1   w w^2]
[  0   0   0   0   0   0   0   0   1   0 w^2 w^2   0   1]
> BKLCLowerBound(F, 14, 9), BKLCUpperBound(F, 14, 9);
4 4
```

Since the database over $GF(4)$ is completely filled up to length 97 the boolean value was in fact unnecessary in this case. We see that the minimum weight of this code reaches the theoretical upper bound, as do all best known codes over $GF(4)$ up to length 18.
For longer lengths we have the possibility that the database may not contain the desired code.

```
> C, has_code := BKLC(F, 98, 57);
> has_code;
false
> C;
[98, 0, 98] Cyclic Linear Code over GF(2^2)
>
> C, has_code := BKLC(F, 98, 58);
> has_code;
true
> C:Minimal;
```

```
[98, 58, 16] Linear Code over GF(2^2)
```

Example H158E40

We search for best known codes using dimension and minimum weight, looking at codes over $GF(2)$ of dimension 85. Even though the database over $GF(2)$ is 100% filled up to length 256, the code required may be longer than that so we have to check the second boolean return value.

```
> C, has_code := BestLengthLinearCode(GF(2),85,23);
> has_code;
true
> C:Minimal;
[166, 85, 23] Linear Code over GF(2)
>
> C, has_code := BestLengthLinearCode(GF(2),85,45);
> has_code;
true
> C:Minimal;
[233, 85, 45] Linear Code over GF(2)
>
> C, has_code := BestLengthLinearCode(GF(2),85,58);
> has_code;
false
```

Example H158E41

For a given minimum weight, we find the maximal known possible dimensions for a variety of code lengths over $GF(4)$.
For lengths < 98 we know the database is filled so we do not need to check the second boolean return value.

```
> F<w> := GF(4);
> C := BDLC(F, 12, 8);
> C;
[12, 3, 8] Linear Code over GF(2^2)
Generator matrix:
[  1   0   0   w w^2   w w^2   w   w   1   w   w]
[  0   1   0   w w^2   1   0 w^2   w   0 w^2 w^2]
[  0   0   1   0   1 w^2 w^2 w^2   0 w^2   w   w]
>
> C := BDLC(F, 27, 8);
> C:Minimal;
[27, 15, 9] Linear Code over GF(2^2)
> C := BDLC(F, 67, 8);
> C:Minimal;
[67, 52, 8] Linear Code over GF(2^2)
```

But for lengths ≥ 98 there may be gaps in the database so to be safe we check the second value.

```
> C, has_code := BDLC(F, 99, 8);
> has_code;
true
> C:Minimal;
[99, 81, 8] Linear Code over GF(2^2)
> C, has_code := BDLC(F, 195, 8);
> has_code;
true
> C:Minimal;
[195, 174, 8] Linear Code over GF(2^2)
```

Example H158E42

We find the best known code of length 54 and dimension 36, then using the output of the verbose mode we re-create this code manually.

```
> SetPrintLevel("Minimal");
> SetVerbose("BestCode",true);
> a := BKLC(GF(2), 54, 36);
Construction of a [ 54 , 36 , 8 ] Code:
[1]:  [63, 46, 7] Cyclic Code over GF(2)
       CyclicCode of length 63 with generating polynomial x^17 +
       x^16 + x^15 + x^13 + x^12 + x^8 + x^6 + x^4 + x^3 + x^2 +
       1
[2]:  [64, 46, 8] Linear Code over GF(2)
       ExtendCode [1] by 1
[3]:  [54, 36, 8] Linear Code over GF(2)
       Shortening of [2] at { 55 .. 64 }
> a;
[54, 36, 8] Linear Code over GF(2)
>
> P<x> := PolynomialRing(GF(2));
> p := x^17 + x^16 + x^15 + x^13 + x^12 + x^8 + x^6 + x^4 +
>                                                 x^3 + x^2 + 1;
> C1 := CyclicCode(63, p);
> C1;
[63, 46] Cyclic Code over GF(2)
> C2 := ExtendCode(C1);
> C2;
[64, 46] Linear Code over GF(2)
> C3 := ShortenCode(C2, {55 .. 64});
> C3;
[54, 36, 8] Linear Code over GF(2)
>
> C3 eq a;
true
```

158.14 Decoding

Magma supplies functions for decoding vectors from the ambient space of a linear code C. The functions in this section only apply to codes over finite fields.

158.14.1 Syndrome Decoding

While syndrome decoding applies to any linear code it is restricted to codes having small codimension since it needs to calculate the coset leaders.

`SyndromeDecoding(C, v)`

Given a linear code C and a vector v from the ambient space V of C, attempt to decode v with respect to C.

If the decoding algorithm succeeds in computing a vector v' as the decoded version of v, then the function returns `true` and v'. If the decoding algorithm does not succeed in decoding v, then the function returns `false` and the zero vector.

`SyndromeDecoding(C, Q)`

Given a linear code C and a sequence Q of vectors from the ambient space V of C, attempt to decode the vectors of Q with respect to C. This function is similar to the function `SyndromeDecoding(C, v)` except that rather than decoding a single vector, it decodes a sequence of vectors and returns a sequence of booleans and a sequence of decoded vectors corresponding to the given sequence.

Example H158E43

We create a code C and a vector v of C and then perturb v to a new vector w. We then decode w to find v again.

```
> C := GolayCode(GF(2), false);
> v := C ! [1,1,1,1,0,0,0,1,0,0,1,1,0,0,0,1,0,0,0,1,1,1,1];
> w := v;
> w[5] := 1 - w[5];
> w[20] := 1 - w[20];
> v;
(1 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 1 1)
> w;
(1 1 1 1 1 0 0 1 0 0 1 1 0 0 0 1 0 0 0 0 1 1 1)
> v - w;
(0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0)
> b, d := SyndromeDecoding(C, w);
> b;
true
> d;
(1 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 1 1)
> d eq v;
true
> SyndromeDecoding(C, [w]);
[ true ]
```

```
[
    (1 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 1 1)
]
```

158.14.2 Euclidean Decoding

The Euclidean decoding algorithm applies to alternant codes which include BCH, Goppa, and Reed–Solomon codes. While the Euclidean algorithm cannot correct as many errors as can the syndrome algorithm, in general it is much faster and so can be applied to much larger codes. This is because the syndrome algorithm needs to determine the coset leaders of the code and so become inapplicable as soon as the codimension of the code is moderately large.

`EuclideanDecoding(C, v)`

Given a linear alternant code C and a vector v from the ambient space V of C, attempt to decode v with respect to C.

If the decoding algorithm succeeds in computing a vector v' as the decoded version of v, then the function returns `true` and v'. It may even happen that v' is not in C because there are too many errors in v to correct. If the decoding algorithm does not succeed in decoding v, then the function returns `false` and the zero vector.

`EuclideanDecoding(C, Q)`

Given a linear alternant code C and a sequence Q of vectors from the ambient space V of C, attempt to decode the vectors of Q with respect to C. This function is similar to the function `EuclideanDecoding(C, v)` except that rather than decoding a single vector, it decodes a sequence of vectors and returns a sequence of booleans and a sequence of decoded vectors corresponding to the given sequence.

Example H158E44

We create a Reed-Solomon code C over $GF(2^8)$ with designated minimum distance 12.

```
> K := GF(2^8);
> C := ReedSolomonCode(K, 12);
> C:Minimal;
[255, 244, 12] "BCH code (d = 12, b = 1)" Linear Code over GF(2^8)
```

So our code has length 255 and dimension 244. With minimum distance 12 it will correct 5 or fewer errors. We demonstrate this by introducing 5 errors into a random codeword c.

```
> c := Random(C);
> V := VectorSpace(C);
> e := V![ 0 :i in [1..255]];
> for i := 1 to 5 do
> j := Random(1, 255);
> k := Random(K);
> e[j] := k;
> end for;
> d := c + e;
> _, cc := EuclideanDecoding(C, d);
> c eq cc;
true;
```

If we introduce 6 or more errors the decoding will usually fail:-

```
> e := V![ 0 :i in [1..255]];
> for i := 1 to 6 do
> j := Random(1, 255);
> k := Random(K);
> e[j] := k;
> end for;
> d := c + e;
> _, cc := EuclideanDecoding(C, d);
> c eq cc;
false
```

158.14.3 Permutation Decoding

`IsPermutationDecodeSet(C, I, S, s)`

Given

- an $[n,k]$ linear code C over a finite field K;
- an information set $I \subseteq \{1,\ldots,n\}$ for C as a sequence of coordinate positions;
- a sequence S of elements in the group of monomial matrices of degree n over K, OR if C is a binary code, a sequence of elements in the symmetric group Sym(n) acting on the set $\{1,\ldots,n\}$. In either case S must be an s-PD-set for C with respect to I;
- and an integer $s \in \{1,\ldots,t\}$, where t is the error-correcting capability of C;

this intrinsic returns `true` if and only if S is an s-PD-set for C with respect to the information set I.

Depending on the length of the code C, its dimension k, and the integer s, this function could take some time to compute whether S is an s-PD-set for C with respect to I. Specifically, if the function returns `true`, it is necessary to check $\sum_{i=1}^{s}\binom{k}{i}\cdot\binom{n-k}{s-i}$ s-sets.

The verbose flag `IsPDSetFlag` is set to level 0 by default. If it is set to level 1, the total time used to check the condition is shown. Moreover, the reason the function returns `false` is also shown, that is, whether I is not an information set, S is not a subset of the monomial automorphism group of C or S is not an s-PD-set. If it is set to level 2, the percentage of the computation process performed is also printed.

`PermutationDecode(C, I, S, s, u)`

Given

- an $[n,k]$ linear code C over a finite field K;
- an information set $I \subseteq \{1,\dots,n\}$ for C as a sequence of coordinate positions;
- a sequence S of elements in the group of monomial matrices of degree n over K, OR if C is a binary code, a sequence of elements in the symmetric group Sym(n) acting on the set $\{1,\dots,n\}$. In either case S must be an s-PD-set for C with respect to I;
- an integer $s \in \{1,\dots,t\}$, where t is the error-correcting capability of C;
- and a vector u from the ambient space V of C,

the intrinsic attempts to decode u with respect to C. 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` and the codeword u'. If the decoding algorithm does not succeed in decoding u, then the function returns `false` and the zero vector in V.

The decoding algorithm works by moving all errors in the received vector $u = c + e$, where $c \in C$ and $e \in V$ is the error vector with at most t errors, out of the information positions, that is, moving the nonzero coordinates of e out of the information set I for C, by using an automorphism of C. Note that the function does not check any of the conditions that I is an information set for C, S is an s-PD-set for C with respect to I, or $s \leq t$.

`PermutationDecode(C, I, S, s, Q)`

Given

- an $[n,k]$ linear code C over a finite field K;
- an information set $I \subseteq \{1,\dots,n\}$ for C as a sequence of coordinate positions;
- a sequence S of elements in the group of monomial matrices of degree n over K, OR if C is a binary code, a sequence of elements in the symmetric group Sym(n) acting on the set $\{1,\dots,n\}$. In either case S must be an s-PD-set for C with respect to I;
- an integer $s \in \{1,\dots,t\}$, where t is the error-correcting capability of C;
- a sequence Q of vectors from the ambient space V of C,

the intrinsic attempts to decode the vectors of Q with respect to C. This function is similar to the function `PermutationDecode(C, I, S, s, u)` except that rather than decoding a single vector, it decodes a sequence of vectors and returns a sequence of booleans and a sequence of decoded vectors corresponding to the given sequence. The algorithm used is as for the function `PermutationDecode(C, I, S, s, u)`.

```
PDSetSimplexCode(K, m)
```

Given a finite field K of cardinality q, and a positive integer m, the intrinsic constructs the $[n = (q^m - 1)/(q - 1), m, q^{m-1}]$ linear simplex code C over K, as `Dual(HammingCode(K, m))`, and then searches for an s-PD-set for C. The function returns an information set I for C together with a subset S of the monomial automorphism group of C such that S is an s-PD-set for C with respect to I, where $s = \lfloor (q^m - 1)/(m(q - 1)) \rfloor - 1$.

The information set I is returned as a sequence of m integers, giving the coordinate positions that correspond to the information set for C. The set S is also returned as a sequence, which contains the $s + 1$ elements in the group of monomial matrices of degree n over K described in [FKM12]. When K is $GF(2)$, the function also returns the elements of S represented as elements in the symmetric group $Sym(n)$ of permutations on the set $\{1, \dots, n\}$.

```
PDSetHadamardCode(m)
```

Given a positive integer m, the intrinsic constructs the $[2^m, m + 1, 2^{m-1}]$ binary linear Hadamard code C, as `Dual(ExtendCode(HammingCode(GF(2), m)))`, and then searches for an s-PD-set for C. The function returns an information set $I \subseteq \{1, \dots, 2^m\}$ for C together with a subset S of the permutation automorphism group of C such that S is an s-PD-set for C with respect to I, where $s = \lfloor 2^m/(m + 1) \rfloor - 1$.

The information set I is returned as a sequence of $m + 1$ integers, giving the coordinate positions that correspond to the information set for C. The set S is also returned as a sequence, which contains the $s + 1$ elements in the group of permutation matrices of degree 2^m over $GF(2)$ described in [BV16a]. The function also returns the elements of S represented as elements in the symmetric group $\mathrm{Sym}(2^m)$ of permutations on the set $\{1, \dots, 2^m\}$.

Example H158E45

```
> C := Dual(ExtendCode(HammingCode(GF(2), 5)));
> C;
[32, 6, 16] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 1 1 1 0 0 1 0 0 0 1 0 1 0 1 1 1 1 0 1 1 0 1 0 0 1 1]
[0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 0]
[0 0 1 0 0 0 1 0 1 0 1 1 1 1 0 1 1 0 1 0 0 1 1 0 0 0 0 0 1 1 1 1]
[0 0 0 1 0 0 1 0 1 1 0 0 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 0 1 0 0]
[0 0 0 0 1 0 0 1 0 1 1 0 0 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 0 1 0]
[0 0 0 0 0 1 1 1 0 0 1 0 0 0 1 0 1 0 1 1 1 1 0 1 1 0 1 0 0 1 1 1]
> I, SMAut, SPAut := PDSetHadamardCode(5);
> I in AllInformationSets(C);
true
> s := #SMAut-1;  s;
4
> [ LinearCode(GeneratorMatrix(C)*SMAut[i]) eq C : i in [1..s+1] ];
```

```
[true, true, true, true, true];
> [ LinearCode(GeneratorMatrix(C)^SPAut[i]) eq C : i in [1..s+1] ];
[true, true, true, true, true];
> IsPermutationDecodeSet(C, I, SMAut, s);
true
> IsPermutationDecodeSet(C, I, SPAut, s);
true
> c := C ! [1^^32];
> c in C;
true
> u := c;
> u[1] := c[1] + 1;
> u[2] := c[2] + 1;
> u[4] := c[4] + 1;
> u[32] := c[32] + 1;
> u in C;
false
> isDecoded, uDecoded := PermutationDecode(C, I, SMAut, s, u);
> isDecoded;
true
> uDecoded eq c;
true
> isDecoded, uDecoded := PermutationDecode(C, I, SPAut, s, u);
> isDecoded;
true
> uDecoded eq c;
true
```

Example H158E46

```
> K<a> := GF(4);
> C := Dual(HammingCode(K, 3));
> C;
[21, 3, 16] Linear Code over GF(2^2)
Generator matrix:
[1 0 a^2 a 1 0 a^2 a 1 a^2 0 1 a a 1 0 a^2 1 a a^2 0]
[0 1 1 1 1 0 0 0 0 a^2 a^2 a^2 a^2 a a a a 1 1 1 1]
[0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
> I, SMAut := PDSetSimplexCode(K, 3);
> I in AllInformationSets(C);
true
> s := #SMAut-1;  s;
6
> [ LinearCode(GeneratorMatrix(C)*SMAut[i]) eq C : i in [1..s+1] ];
[true, true, true, true, true, true, true];
> IsPermutationDecodeSet(C, I, SMAut, s);
true
```

```
> c := C ! [0,1,1,1,1,0,0,0,0,a^2,a^2,a^2,a^2,a,a,a,a,1,1,1,1];
> c in C;
true
> u := c;
> u[1] := c[1] + a;
> u[2] := c[2] + a^2;
> u[3] := c[3] + a;
> u[4] := c[4] + a^2;
> u[5] := c[5] + a;
> u[6] := c[6] + a^2;
> u in C;
false
> isDecoded, uDecoded := PermutationDecode(C, I, SMAut, s, u);
> isDecoded;
true
> uDecoded eq c;
true
```

158.15 Transforms

158.15.1 Mattson–Solomon Transforms

```
MattsonSolomonTransform(f, n)
```

Given f, a polynomial over a finite field containing a primitive n-th root of unity, return the Mattson–Solomon transform of parameter n.

```
InverseMattsonSolomonTransform(A, n)
```

Given A, a polynomial over a finite field containing a primitive n-th root of unity, return the inverse Mattson–Solomon transform of parameter n.

Example H158E47

We compute the Mattson–Solomon transform of parameter $n = 7$ of the polynomial x^4+x^2+x+1 over $\mathbf{F}_{2^{12}}$.

```
> n := 7;
> K := GF(2, 12);
> FP<x> := PolynomialRing(K);
> f := x^4 + x^2 + x + 1;
> A := MattsonSolomonTransform(f, n);
> A;
x^6 + x^5 + x^3
```

158.15.2 Krawchouk Polynomials

```
KrawchoukPolynomial(K, n, k)
```

Return the Krawchouk polynomial of parameters k and n in K over the rational field.

```
KrawchoukTransform(f, K, n)
```

Return the Krawchouk transform of the polynomial f over the rational field with respect to the vector space K^n.

```
InverseKrawchouk(A, K, n)
```

Return the inverse Krawchouk transform of the polynomial A over the rational field with respect to the vector space K^n.

158.16 Automorphism Groups

158.16.1 Introduction

Let C be an $[n, k]$ linear code and G a permutation group of degree n. Then G acts on C in the following way: for a codeword v of C and a permutation x of G, the image of v under x is obtained from v by permuting the coordinate positions of v according to x. We call this the *permutation* action of G on C.

If C is a non-binary code over a finite field, there is also a *monomial* action on C. Let K be the alphabet of C. A monomial permutation of monomial degree n is equivalent to a permutation s on $K^* \times \{1, \dots, n\}$ which satisfies the following property:

$$(\alpha, i)^s = (\beta, j) \text{ implies } (\gamma\alpha, i)^s = (\gamma\beta, j)$$

for all $\alpha, \beta, \gamma \in K^*$ and $i, j \in \{1, \dots, n\}$. The actual degree of s is $(q-1)n$. Note that s is completely determined by its action on the points $(1, i)$ for each i, and the matrix representation of s is also determined by its action on the elements $(1, i)$, for $1 \le i \le n$. To represent a monomial permutation of monomial degree n, we number the pair (α, i) by $(q-1)(i-1) + \alpha$ and then use a permutation s of degree $(q-1)n$.

The functions in this section allow one to investigate such actions. The algorithms in Magma to compute with such actions are backtrack searches due to Jeff Leon [Leo82][Leo97]. There are 4 algorithms which are provided for codes of length n over a field of cardinality q:

(a) *Automorphism group* or *Monomial group.* Computes the group of monomials which map a code into itself, where monomials are represented as permutations of degree $(q-1)n$ (so the group has degree $(q-1)n$). For this function q may be any small prime or 4.

(b) *Permutation group.* Computes the group of permutations which map a code into itself (so the group has degree n). For this function q may be any small prime or 4.

(c) *Equivalence test.* Computes whether there is a monomial permutation which maps a code to another code and, if so, returns the monomial as a permutation of degree $(q-1)n$. For this function, q may be any small prime or 4.

(d) *Isomorphism test.* Computes whether there is a permutation which maps a code to another code and returns the permutation (of degree n) if so. For this function q may only be 2.

For more information on permutation group actions and orbits, see Chapter 63.

158.16.2 Group Actions

```
v ^ x
```

Given a codeword v belonging to the $[n, k]$ code C and an element x belonging to a permutation group G, construct the vector w obtained from v by the action of x. If G has degree n, the permutation action is used; otherwise G should have degree $n(q-1)$ and the monomial action is used.

```
v ^ G
```

Given a codeword v belonging to the $[n, k]$ code C and a permutation group G (with permutation or monomial action on C), construct the vector orbit Y of v under the action of G. The orbit Y is a G-set for the group G.

```
C ^ x
```

Given an $[n, k]$ code C and an element x belonging to a permutation group G (with permutation or monomial action on C), construct the code consisting of all the images of the codewords of C under the action of x.

```
C ^ G
```

Given an $[n, k]$ code C and a permutation group G (with permutation or monomial action on C), construct the orbit Y of C under the action of G. The orbit Y is a G-set for the group G.

```
S ^ x
```

Given a set or sequence S of codewords belonging to the $[n, k]$ code C and an element x belonging to a permutation group (with permutation or monomial action on the codewords), construct the set or sequence of the vectors obtained by permuting the coordinate positions of v, for each v in S, according to the permutation x.

```
S ^ x
```

Given a set or sequence S of codes of length n and an element x belonging to a permutation group (with permutation or monomial action on the codes) construct the set or sequence of the codes consisting of all the images of the codewords of C under the action of x.

```
Fix(C, G)
```

Given an $[n, k]$ code C and a permutation group G of degree n, find the subcode of C which consists of those vectors of C which are fixed by the elements of G. That is, the subcode consists of those codewords that are fixed by the group G.

158.16.3 Automorphism Group

`AutomorphismGroup(C: parameters)`

`MonomialGroup(C: parameters)`

`Weight` RngIntElt *Default :* 0

The automorphism group A of the $[n, k]$ linear code C over the field K, where A is the group of all monomial-action permutations which preserve the code. Thus both permutation of coordinates and multiplication of components by non-zero elements from K is allowed, and the degree of A is $n(q-1)$ where q is the cardinality of K. A power structure P and transfer map t are also returned, so that, given a permutation g from A, one can create a map $f = t(g)$ which represents the automorphism g as a mapping $P : C \to C$.

If the code is known to have very few words of low weight, then it may take some time to compute the *support* of the code (a set of low weight words). The optional parameter `Weight` can be used to specify the set of vectors of the specified weight to be used as the support in the algorithm. This set should be of a reasonable size, (possibly hundreds for a large code), while also keeping the weight as small as possible.

Warning: If `Weight` specifies a set that is too small, then the algorithm risks getting stuck.

`PermutationGroup(C)`

The permutation group G of the $[n, k]$ linear code C over the field K, where G is the group of all permutation-action permutations which preserve the code. Thus only permutation of coordinates is allowed, and the degree of G is always n. A power structure P and transfer map t are also returned, so that, given a permutation g from G, one can create a map $f = t(g)$ which represents the automorphism g as a mapping $P : C \to C$.

`AutomorphismSubgroup(C)`

`MonomialSubgroup(C)`

A subgroup of the (monomial) automorphism group A of the code C. If the automorphism group of C is already known then the group returned is the full automorphism group, otherwise it will be a subgroup generated by one element. This allows one to find just one automorphism of C if desired. A power structure P and transfer map t are also returned, so that, given a permutation g from A, one can create a map $f = t(g)$ which represents the automorphism g as a mapping $P : C \to C$.

`AutomorphismGroupStabilizer(C, k)`

`MonomialGroupStabilizer(C, k)`

The subgroup of the (monomial) automorphism group A of the code C, which stabilizes the first k base points as chosen by the backtrack search. These base points may be different to those of the returned group. A power structure P and transfer map t are also returned, so that, given a permutation g from A, one can create a map $f = t(g)$ which represents the automorphism g as a mapping $P : C \to C$.

`Aut(C)`

The power structure A of all automorphisms of the code C (with monomial action), together with the transfer map t into A from the generic symmetric group associated with the automorphism group of C.

`Aut(C, T)`

The power structure A of all automorphisms of the code C, together with the transfer map t into A from the generic symmetric group associated with the automorphism group of C; the string T determines which action type should be used: `"Monomial"` or `"Permutation"`.

Example H158E48

We compute the automorphism group of the second order Reed–Muller code of length 64.

```
> C := ReedMullerCode(2, 6);
> aut := AutomorphismGroup(C);
> FactoredOrder(aut);
[ <2, 21>, [3, 4>, <5, 1>, <7, 2>, <31, 1> ]
> CompositionFactors(aut);
    G
    |  A(5, 2)                = L(6, 2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
```

Example H158E49

We compute the automorphism group of a BCH code using the set of vectors of minimal weight as the invariant set. We look first at its weight distribution to confirm that there is sufficient vectors.

```
> C := BCHCode(GF(2),23,2);
> C;
[23, 12, 7] BCH code (d = 2, b = 1) over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1]
[0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 1 1 1 0 0 0 1 1]
[0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1]
[0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 0 1]
> WeightDistribution(C);
[ <0, 1>, <7, 253>, <8, 506>, <11, 1288>, <12, 1288>, <15, 506>,
<16, 253>, <23, 1> ]
> AutomorphismGroup(C : Weight := MinimumWeight(C) );
Permutation group acting on a set of cardinality 23
Order = 10200960 = 2^7 * 3^2 * 5 * 7 * 11 * 23
    (6, 15, 12)(7, 20, 19)(8, 9, 17)(10, 13, 22)(11, 14, 23)(16,
        18, 21)
    (5, 17, 9)(6, 21, 20)(7, 16, 23)(10, 13, 22)(11, 12, 19)(14,
        18, 15)
    (5, 16, 21)(6, 23, 19)(7, 17, 12)(8, 14, 15)(9, 20, 11)(10,
        13, 22)
    (1, 2)(4, 12)(5, 7)(6, 17)(9, 10)(13, 21)(15, 18)(22, 23)
    (2, 3)(4, 21, 6, 20)(5, 10, 12, 17)(7, 13, 9, 11)(15, 18)(16,
        19, 22, 23)
    (3, 8)(4, 5, 20, 22)(6, 16, 21, 12)(7, 11, 13, 9)(10, 17, 19,
        23)(15, 18)
    (4, 8, 6, 21, 20)(5, 10, 14, 17, 12)(7, 22, 18, 16, 9)(11,
        23, 19, 13, 15)
```

158.16.4 Equivalence and Isomorphism of Codes

```
IsIsomorphic(C, D: parameters)
IsEquivalent(C, D: parameters)
```

`AutomorphismGroups`	MonStgElt	*Default :* "*Right*"
`Weight`	RngIntElt	*Default :* 0

Given $[n, k]$ codes C and D, this function returns `true` if and only if C is equivalent to D. If C is equivalent to D, an equivalence map f is also returned from C onto D. The equivalence is with respect to the monomial action. The function first computes none, one, or both of the automorphism groups of the left and right codes. This may assist the isomorphism testing.

The parameter `AutomorphismGroups`, with valid string values `Both`, `Left`, `Right`, `None`, may be used to specify which of the automorphism groups should be constructed first if not already known. The default is `Right`.

In rare cases this algorithm can get stuck, due to an insufficient set of invariant vectors. In this case, the optional parameter `Weight` can be used to specify this set to be the vectors of the specified weight. This set should be of a reasonable size, (possibly hundreds for large codes), while also keeping the weight as small as possible.

Warning: If `Weight` specifies a set that is too small, then the algorithm risks getting stuck.

158.17 Bibliography

[**All84**] W.O. Alltop. A method for extending binary linear codes. *IEEE Trans. Inform. Theory*, 30:871 – 872, 1984.

[**BFK$^+$98**] A. Betten, H. Fripertinger, A. Kerber, A. Wassermann, and K.-H. Zimmermann. *Codierungstheorie – Konstruktion und Anwendung linearer Codes*. Springer-Verlag, Berlin–Heidelberg–New York, 1998.

[**Bro**] A. E. Brouwer. Bounds on the minimum distance of linear codes. URL:http://www.win.tue.nl/˜aeb/voorlincod.html.

[**Bro98**] A. E. Brouwer. Bounds on the size of linear codes. In *Handbook of coding theory, Vol. I, II*, pages 295–461. North-Holland, Amsterdam, 1998.

[**BV16**] R. Barrolleta and M. Villanueva. Partial permutation decoding for binary linear and $\mathbf{Z}_4$-linear Hadamard codes. Submitted to *Designs, Codes and Cryptography*, arXiv:1512.01839, 2016.

[**FKM12**] W. Fish, J. Key, and E. Mwambene. Partial permutation decoding for simplex codes. *Advances in Mathematics of Communications*, 6(4):505–516, 2012.

[**Gab02**] Philippe Gaborit. Quadratic Double Circulant Codes over Fields. *Journal of Combinatorial Theory*, 97:85–107, 2002.

[**Gra**] Markus Grassl. Bounds on the minimum distance of linear codes. URL:http://www.codetables.de/.

[**Leo82**] Jeffrey S. Leon. Computing automorphism groups of error-correcting codes. *IEEE Trans. Inform. Theory*, IT-28:496–511, 1982.

[**Leo97**] Jeffrey S. Leon. Partitions, refinements, and permutation group computation. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation II*, volume 28 of *Dimacs series in Discrete Mathematics and Computer Science*, pages 123–158, Providence R.I., 1997. Amer. Math. Soc.

[**MS78**] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes.* North Holland, New York, 1978.

159 ALGEBRAIC-GEOMETRIC CODES

Chapter 159

ALGEBRAIC-GEOMETRIC CODES

159.1 Introduction

Algebraic–Geometric Codes (AG–codes) are a family of linear codes described by Goppa in [Gop81a, Gop81b]. Let $\mathcal{X}$ be an irreducible projective plane curve of genus g, defined by a (absolutely irreducible) homogeneous polynomial $H(X, Y, Z)$ over a finite field $K = \mathbf{F}_q$. A *place* of $\mathcal{X}$ is the maximal ideal of a discrete valuation subring of $\overline{K}(\mathcal{X})$. We denote by v_P the valuation at place P. Its *degree* is the degree of its residue class field over K. A *divisor* is an element D of the free abelian group over the set of places of $\mathcal{X}$. Namely, such an element can be written additively:

$$D = \sum_{P \in \mathrm{Pl}(\mathcal{X})} n_P \cdot P \ ,$$

where all but finitely many $n_P \in \mathbf{Z}$ are zero. The set of places with nonzero multiplicity is the *support* of D, denoted by Supp$D$. The set of divisors can be equipped with a natural partial order $\leq$ defined by:

$$D = \sum_{P \in \mathrm{Pl}(\mathcal{X})} n_P \cdot P \leq D' = \sum_{P \in \mathrm{Pl}(\mathcal{X})} n'_P \cdot P \quad \iff \quad n_P \leq n'_P \text{ for all } P.$$

If $f \in K(\mathcal{X})$, then one can define the *principal divisor*:

$$(f) = \sum_{P \in \mathrm{Pl}(\mathcal{X})} v_P(f) \cdot P \ .$$

A divisor D is said to be *defined* over K if it is stable under the natural action of Gal$(\overline{K}/K)$.

If $(P_1, \ldots, P_n)$ is a tuple of places of degree 1, then for a function $f \in K(\mathcal{X})$, $f(P_i)$ can be seen as an element of K. If D is a divisor defined over K, then the *Riemann–Roch* space $\mathcal{L}(D)$ of D is the K-vector space of dimension k:

$$\mathcal{L}(D) = \{f \in K(\mathcal{X})^* \mid (f) + D \geq 0\} \cup \{0\}.$$

Now provided D has support disjoint from $S = \{P_1, \ldots, P_n\}$, we can define the *algebraic geometric code* to be the $[n, k]_q$-code:

$$C = C(S, D) = \{(f(P_1), \ldots, f(P_n)) \ , f \in \mathcal{L}(D)\}.$$

If $\langle f_1, \ldots, f_k \rangle$ is a base of $\mathcal{L}_K(D)$ as a K-vector space, then a generator matrix for C is:

$$G = \begin{pmatrix} f_1(P_1) & \cdots & f_1(P_n) \\ \vdots & \ddots & \vdots \\ f_k(P_1) & \cdots & f_k(P_n) \end{pmatrix} .$$

Standard references are [Sti93] and [TV91].

There are two different implementations of the construction of AG–Codes in Magma. The first was implemented by Lancelot Pecquet and is based on the work of Hache [HLB95, Hac96]. The second approach exploits the divisor machinery for function fields implemented by Florian Hess. In Magma V2.8, only the second implementation is exported. It is intended to rework the Pecquet version to take advantage of the new curve machinery before releasing it.

159.2 Creation of an Algebraic Geometric Code

```
AlgebraicGeometricCode(S, D)
```

```
AGCode(S, D)
```

Suppose X is an irreducible plane curve. Let S be a sequence of places of X having degree 1 and let D be a divisor of X whose support is disjoint from the support of S. The function returns the (weakly) algebraic–geometric code obtained by evaluating functions of the Riemann–Roch space of D at the points of S. The degree of D need not be bounded by the cardinality of S.

```
AlgebraicGeometricDualCode(S, D)
```

```
AGDualCode(S, D)
```

Construct the dual of the algebraic geometric code constructed from the sequence of places S and the divisor D, which corresponds to a differential code. In order to take advantage of the algebraic geometric structure, the dual must be constructed in this way, and not by directly calling the function `Dual`.

```
HermitianCode(q, r)
```

Given the prime power q and a positive integer r, construct a Hermitian code C with respect to the Hermitian curve

$$X = x^{(q+1)} + y^{(q+1)} + z^{(q+1)}$$

defined over $\mathbf{F}_{q^2}$. The support of C consists of all places of degree one of X over $\mathbf{F}_{q^2}$, with the exception of the place over $P = (1 : 1 : 0)$. The divisor used to define a Riemann–Roch space is $r * P$.

Example H159E1

We construct a $[25, 9, 16]$ code over F_{16} using the genus 1 curve $x^3 + x^2z + y^3 + y^2z + z^3$.

```
> F<w> := GF(16);
> P2<x,y,z> := ProjectiveSpace(F, 2);
> f := x^3+x^2*z+y^3+y^2*z+z^3;
> X := Curve(P2, f);
> g := Genus(X);
> g;
1
> places1 := Places(X, 1);
> #places1;
25
```

We now need to find an appropriate divisor D. Since we require a code of dimension $k = 9$ we take the divisor corresponding to a place of degree $k + g - 1 = 9$ (g is the genus of the curve).

```
> found, place_k := HasPlace(X, 9+g-1);
> D := DivisorGroup(X) ! place_k;
> C := AlgebraicGeometricCode(places1, D);
> C;
[25, 9] Linear Code over GF(2^4)
Generator matrix:
[1 0 0 0 0 0 0 0 0 w^9 w w^8 0 w^5 1 1 w^5 w^7 w^8 w^10 w^4 w^11 w^5
   w^10 w^8]
[0 1 0 0 0 0 0 0 0 w^4 w^5 1 w^10 w^4 w^11 w^12 0 1 w^2 w^4 w^6 w^4 w^5
   w^14 w^4]
[0 0 1 0 0 0 0 0 0 w^5 w^13 w^7 w^10 w^7 w^5 w^14 w^14 w 1 w^10 w^9 1 0
   w^5 w]
[0 0 0 1 0 0 0 0 0 w^8 w^3 w^3 w^12 w^7 w^10 w w^6 0 w^7 w^10 w^4 w^9
   w^14 w^8 w^12]
[0 0 0 0 1 0 0 0 0 w^8 1 w^4 w^7 w^5 w w^8 w w^5 w w^13 0 w^14 w^14 w^14
   w^6]
[0 0 0 0 0 1 0 0 0 1 1 w^12 w^14 w^9 w^10 w^6 w^6 w^7 w^10 w^4 w^3 w^13
   w^13 w^3 w^4]
[0 0 0 0 0 0 1 0 0 w w w^10 w^4 w^12 w w^13 w^4 w w^2 w^3 w^3 w^12 w^10
   w^5 w^13]
[0 0 0 0 0 0 0 1 0 w^13 w^6 w^12 w^2 w^3 w^7 w^3 w^4 w^14 w^4 w^11 w^4 w
   w^6 w^4 w^14]
[0 0 0 0 0 0 0 0 1 0 w^11 w w^7 w^12 w^4 w^3 w^6 w^12 w^3 w^13 w^2 w^11
   w^10 w^3 1]
> MinimumDistance(C);
16
```

Example H159E2

We construct a $[44, 12, 29]$ code over F_{16} using the genus 4 curve $(y^2 + xy + x^2)z^3 + y^3z^2 + (xy^3 + x^2y^2 + x^3y + x^4)z + x^3y^2 + x^4y + x^5$ and taking as the divisor a multiple of a degree 1 place.

```
> k<w> := GF(16);
> P2<x,y,z> := ProjectiveSpace(k, 2);
> f := (y^2+x*y+x^2)*z^3+y^3*z^2+(x*y^3+x^2*y^2+x^3*y+x^4)*z+x^3*y^2+x^4*y+x^5;
> X := Curve(P2, f);
> g := Genus(X);
> g;
4
```

We find all the places of degree 1.

```
> places1 := Places(X, 1);
> #places1;
45
```

We choose as our divisor $15 * P1$, where $P1$ is a place of degree 1. Before applying the AG-Code construction we must remove $P1$ from the set of places of degree 1.

```
> P1 := Random(places1);
> Exclude(~places1, P1);
> #places1;
44
> D := 15 * (DivisorGroup(X) ! P1);
> C := AlgebraicGeometricCode(places1, D);
> C:Minimal;
[44, 12] Linear Code over GF(2^4)
> MinimumWeight(C);
29
```

159.3 Properties of AG–Codes

`IsWeaklyAG(C)`

Return `true` if and only if the code C is a weakly algebraic–geometric code, i.e. C has been constructed as an algebraic–geometric code with respect to a divisor of any degree.

`IsWeaklyAGDual(C)`

Return `true` if and only if the code C was constructed as the dual of a weakly algebraic–geometric code.

`IsAlgebraicGeometric(C)`

Return `true` if and only if the code C is of algebraic–geometric construction of length n, built from a divisor D with $deg(D) < n$.

```
IsStronglyAG(C)
```

Return `true` if and only if C is an algebraic–geometric code of length n constructed from a divisor D satisfying $2g - 2 < deg(D) < n$, where g is the genus of the curve.

159.4 Access Functions

At the time an AG–Code is constructed a number of attributes describing its construction are stored along with the code. The functions in this section give the user access to these attributes.

```
Curve(C)
```

Given an algebraic–geometric code C, returns the curve from which C was defined.

```
GeometricSupport(C)
```

Given an algebraic–geometric code C, return the sequence of places which forms the support for C.

```
Divisor(C)
```

Given an algebraic–geometric code C, return the divisor from which C was constructed.

```
GoppaDesignedDistance(C)
```

Given an algebraic–geometric code C constructed from a divisor D, return the Goppa designed distance $n - deg(D)$.

159.5 Decoding AG Codes

Specialized decoding algorithms exist for differential code, those which are the duals of the standard algebraic-geometric codes. These algorithms generally require as input another divisor on the curve whose support is disjoint from the divisor defining the code.

```
AGDecode(C, v, Fd)
```

Decode the received vector v of the dual algebraic geometric code C using the divisor Fd.

Example H159E3

An algebraic-geometric code with Goppa designated distance of 3 is used to correct one error.

```
> q := 8;
> F<a> := GF(q);
> PS<x,y,z> := ProjectiveSpace(F, 2);
> W := x^3*y + y^3*z + x*z^3;
> Cv := Curve(PS, W);
> FF<X,Y> := FunctionField(Cv);
> Pl := Places(Cv, 1);
> plc := Place(Cv ! [0,1,0]);
> P := [ Pl[i] : i in [1..#Pl] | Pl[i] ne plc ];
> G := 11*plc;
> C := AGDualCode(P, G);
>
> v := Random(C);
> rec_vec := v;
> rec_vec[Random(1,Length(C))] +:= Random(F);
> res := AGDecode(C, v, 4*plc);
> res eq v;
true
```

159.6 Toric Codes

`ToricCode(P, q)`

The linear code C over the finite field $\mathbf{F}_q$ associated with the lattice points of the polygon P.

To achieve this, after a translation so that the lattice points of P lie in the first quadrant, as close to the origin as possible, these points must lie in the box $[0, q-2] \times [0, q-2]$. Then the code is the monomial evaluation code where each point (a, b) corresponds to the monomial $x^a y^b$, and these monomials are evaluated at the points of the torus $(\mathbf{F}_q^*)^2$.

`ToricCode(S, q)`

`ToricCode(S, q)`

The linear code C over the finite field $\mathbf{F}_q$ associated with the lattice points in S. (Note that the points will be translated to lie within a box at the origin of the first quadrant, as is usual.)

Example H159E4

We construct the toric code based on the lattice points in the polygon with vertices $(3,0)$, $(5,0)$, $(3,3)$, $(1,5)$, $(0,3)$, $(0,1)$.

```
> P := Polytope( [[3,0], [5,0], [3,3], [1,5], [0,3], [0,1]] );
> C := ToricCode(P, 7);
> [ Length(C), Dimension(C), MinimumDistance(C) ];
[ 36, 19, 12 ]
```

We can compare this with the current database of best known linear codes.

```
> BKLCLowerBound(Field(C), Length(C), Dimension(C));
11
```

159.7 Bibliography

[**Gop81a**] V. D. Goppa. Codes on algebraic curves. *Dokl. Akad. Nauk SSSR*, 259(6): 1289–1290, 1981.

[**Gop81b**] V. D. Goppa. Codes on algebraic curves. *Soviet Math. Dokl.*, 24(1):170–172, 1981.

[**Hac96**] Gaétan Haché. *Construction effective des codes géométriques.* PhD thesis, l'Université Paris 6, 1996.

[**HLB95**] Gaétan Haché and Dominique Le Brigand. Effective construction of algebraic geometry codes. *IEEE Trans. Inform. Theory*, 41(6, part 1):1615–1628, 1995. Special issue on algebraic geometry codes.

[**Sti93**] Henning Stichtenoth. *Algebraic function fields and codes.* Springer-Verlag, Berlin, 1993.

[**TV91**] M. A. Tsfasman and S. G. Vlăduţ. *Algebraic-geometric codes.* Kluwer Academic Publishers Group, Dordrecht, 1991. Translated from the Russian by the authors.

160 LOW DENSITY PARITY CHECK CODES

Chapter 160
LOW DENSITY PARITY CHECK CODES

160.1 Introduction

Low density parity check (LDPC) codes are among the best performing codes in practice, being capable of correcting errors close to the Shannon limit. Magma provides facilities for the construction, decoding, simulation and analysis of LDPC codes.

160.1.1 Constructing LDPC Codes

LDPC codes come in two main varieties, *regular* and *irregular*, defined by the row and column weights of the sparse parity check matrix. If all columns in the parity check matrix have some constant weight a, and all rows have some constant weight b, then the LDPC code is said to be (a, b)-regular. When either the columns or the rows have a distribution of weights, the LDPC code is said to be irregular.

Currently, there do not exist many techniques for the explicit construction of LDPC codes. More commonly, these codes are selected at random from an ensemble, and their properties determined through simulation.

```
LDPCCode(H)
```

Given a sparse binary matrix H, return the LDPC code which has H as its parity check matrix.

```
GallagerCode(n, a, b)
```

Return a random (a, b)-regular LDPC code of length n, using Gallager's original method of construction. The row weight a must dividethe length n.

```
RegularLDPCEnsemble(n, a, b)
```

Return a random code from the ensemble of (a, b)-regular binary LDPC codes.

```
IrregularLDPCEnsemble(n, Sc, Sv)
```

Given (unnormalized) distributions for the variable and check weights, return length n irregular LDPC codes whose degree distributions match the given distribution. The arguments Sv and Sc are sequences of real numbers, where the i-th entry indicates what percentage of the variable (resp. check) nodes should have weight i.

Note that the distributions will not be matched perfectly unless everything is in complete balance.

```
MargulisCode(p)
```

Return the (3,6)-regular binary LDPC code of length $2(p^3 - p)$ using the group-based construction of Margulis.

Example H160E1

Most LDPC constructions are generated pseudo-randomly from an ensemble, so the same function will return a different code each time. To be able to re-use an LDPC code, the sparse parity check matrix must be saved.

```
> C1 := RegularLDPCEnsemble(10, 2, 4);
> C2 := RegularLDPCEnsemble(10, 2, 4);
> C1 eq C2;
false
> LDPCMatrix(C1):Magma;
SparseMatrix(GF(2), 5, 10, [
    4, 2,1, 3,1, 4,1, 6,1,
    4, 1,1, 7,1, 9,1, 10,1,
    4, 1,1, 2,1, 3,1, 7,1,
    4, 5,1, 6,1, 8,1, 9,1,
    4, 4,1, 5,1, 8,1, 10,1
])
> H := LDPCMatrix(C1);
> C3 := LDPCCode(H);
> C3 eq C1;
true
```

160.1.2 Access Functions

Since a code can have many different parity check matrices, the matrix which defines a code as being LDPC must be assigned specifically. Any parity check matrix can be assigned for this purpose, and once a code is assigned an LDPC matrix it is considered by Magma to be an LDPC code (regardless of the density or other properties of the matrix). The matrix must be of sparse type (`MtrxSprs`).

`IsLDPC(C)`

Return true if C is an LDPC code (which is true if it has been assigned an LDPC matrix).

`AssignLDPCMatrix(~C, H)`

Given a sparse matrix H which is a parity check matrix of the code C, assign H as the LDPC matrix of C.

`LDPCMatrix(C)`

Given an LDPC code C, return the sparse matrix which has been assigned as its low density parity check matrix.

`LDPCDensity(C)`

Given an LDPC code C, return the density of the sparse matrix which has been assigned as its low density parity check matrix.

```
IsRegularLDPC(C)
```

Returns `true` if C is an LDPC code and has regular column and row weights. If true, the row and column weights are also returned.

```
TannerGraph(C)
```

For an LDPC code C, return its Tanner graph. If there are n variables and m checks, then the graph has $n+m$ nodes, the first n of which are the variable nodes.

```
LDPCGirth(C)
```

For an LDPC code C, return the girth of its Tanner graph.

```
LDPCEnsembleRate(v, c)
LDPCEnsembleRate(Sv, Sc)
```

Return the theoretical rate of LDPC codes from the ensemble described by the given inputs.

Example H160E2

In Magma, whether or not a code is considered LDPC is based solely on whether or not an LDPC matrix has been assigned. This example shows that any code can be made to be considered LDPC, although a random parity check matrix without low density will perform very badly using LDPC decoding.

```
> C := RandomLinearCode(GF(2),100,50);
> IsLDPC(C);
false
> H := SparseMatrix(ParityCheckMatrix(C));
> H;
Sparse matrix with 50 rows and 100 columns over GF(2)
> AssignLDPCMatrix(~C, H);
> IsLDPC(C);
true
> LDPCDensity(C);
0.253400000000000014122036873232
```

The density of the parity check matrices of LDPC codes is much lower than that of randomly generated codes.

```
> C1 := RegularLDPCEnsemble(100,3,6);
> C1:Minimal;
[100, 50] Linear Code over GF(2)
> LDPCDensity(C1);
0.0599999999999999977795539507497
```

160.1.3 LDPC Decoding and Simulation

The impressive performance of LDPC codes lies in their iterative decoding algorithm. Magma provides facilities to decode using LDPC codes, as well as simulating transmission over a binary symmetric or white Gaussian noise channels.

The binary symmetric channel transmits binary values and is defined by $p < 0.5$. Each individual bit independently sustains a "bit-flip" error with probability p.

The Gaussian channel is analog, transmitting real-values, and is defined by a standard deviation σ. Binary values are mapped to -1 and 1 before being transmitted. Each value independently sustains an errors which are normally distributed about 0 with standard deviation σ.

`LDPCDecode(C, v)`

`Channel`	MonStgElt	*Default :* "*BinarySymmetric*"
`p`	RngReSubElt	*Default :* 0.1
`StdDev`	RngReSubElt	*Default :* 0.25
`Iterations`	RngIntElt	*Default :* `Dimension(C)`

For an LDPC code C and a received vector v, decode v to a codeword of C using the LDPC iterative decoding algorithm.

The nature of the channel from which v is received is described by the variable argument `Channel`, which can either be the `BinarySymmetric` channel or the `Gaussian` channel. Errors on the binary symmetric channel is described by the argument `p`, while on the Gaussian channel they are described by `StdDev`.

The vector v must be over a ring corresponding to the channel which is selected. For the binary symmetric channel v must be a binary vector over F_2, while for the Gaussian channel it must be real-valued.

Since the decoding algorithm is iterative and does not necessarily terminate on its own, a maximum number of iterations needs to be specified using the argument `Iterations`. The default value is much larger than would normally be used in practice, giving maximum error-correcting performance (at possibly some cost to efficiency).

Example H160E3

Errors in the binary symmetric channel are just bit flips.

```
> n := 500;
> C := RegularLDPCEnsemble(n, 4, 8);
> e := 5;
> Errs := {};
> repeat Include(~Errs, Random(1,n)); until #Errs eq e;
> v := Random(C);
> ev := AmbientSpace(C)![ (i in Errs) select 1 else 0 : i in [1..n]];
> rec_vec := v + ev;
> time res := LDPCDecode(C, rec_vec : Channel:="BinarySymmetric", p:=0.2);
Time: 0.000
```

```
> res eq v;
true
```

Example H160E4

For the Gaussian channel binary vectors are considered to be transmitted as sequences of the values 1 and -1. Errors are normally distributed with a standard deviation defined by the channel.
To simulate a Gaussian channel requires obtaining normally distributed errors. This can be done (discretely) by generating a multiset of possible errors.

```
> sigma := 0.5;
> MaxE := 3.0;
> N := 100;
> V := [ MaxE*(i/N)     : i in [-N div 2..N div 2]];
> E := [ 0.5*(1+Erf(x/(sigma*Sqrt(2)))) : x in V ];
> Dist := {* V[i]^^Round(1000*(E[i]-E[i-1])) : i in [2..#V]*};
```

A codeword of an LDPC code needs to be mapped into the real domain.

```
> n := 500;
> C := RegularLDPCEnsemble(n, 4, 8);
> v := Random(C);
> R := RealField();
> RS := RSpace(R, n);
> vR := RS ! [ IsOne(v[i]) select 1 else -1 : i in [1..n]];
```

Normally distributed errors are then introduced, and the received vector decoded.

```
> for i in [1..n] do
>    vR[i] +:= Random(Dist);
> end for;
> time res := LDPCDecode(C, vR : Channel:="Gaussian", StdDev:=sigma);
Time: 0.000
> res eq v;
true
```

`LDPCSimulate(C, N)`

`Channel`	MonStgElt	*Default :* "*BinarySymmetric*"
`p`	RngReSubElt	*Default :* 0.1
`StdDev`	RngReSubElt	*Default :* 0.25
`Iterations`	RngIntElt	*Default :* `Dimension(C)`

For an LDPC code C, simulate N transmissions across the given channel and return the accumulated bit error rate and word error rate.

The variable arguments are as described for the function `LDPCDecode`. The channel which is described controls not only the way the decoding algorithm functions, but also the nature of the errors introduced during the simulation.

Example H160E5

The more noise that is introduced into the channel the error rate increases. Note that the bit error rate (the first return value) is always lower than the word error rate (the second return value).

```
> C := RegularLDPCEnsemble(200, 3, 6);
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 0.7);
0.0011824999999999995739519143001 0.00619999999999999978211873141731
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 0.75);
0.00749499999999999992617016886243 0.041000000000000017208456881690
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 0.8);
0.03372200000000000019735324485737 0.15970000000000000008615330671091
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 0.85);
0.08561749999999999991606713933834 0.37080000000000000018474111129763
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 0.9);
0.16279099999999999991265653420669 0.64049999999999999958255614274094
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 0.95);
0.23765749999999999993756105709508 0.84009999999999999957900342906214
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 1.0);
0.29652650000000000026169288958044 0.94479999999999999973177011725056
```

160.1.4 Density Evolution

The asymptotic performance of ensembles of LDPC codes can be determined using *density evolution*. An ensemble of LDPC codes (either regular or irregular) is defined by a pair of degree distributions, corresponding to the degrees at the variable and check nodes of the Tanner graph.

Over a specific channel, the critical parameter which defines the asymptotic performance of a given ensemble is its *threshold*, which is a value of the channel parameter (i.e., the probability of error p for the binary symmetric channel, and the standard deviation σ for the Gaussian channel). When the channel parameter is less than the threshold, asymptotically a code from the ensemble will decode with an error probability tending to zero. However, at any channel parameter above the threshold there will be a non-vanishing finite error probability.

Determining the threshold of an ensemble over the binary symmetric channel is relatively trivial, however over the real-valued Gaussian channel it can involve extensive computations. The speed depends heavily on the granularity of the discretization which is used, though this also affects the accuracy of the result.

The default settings of the Magma implementation use a reasonably coarse discretization, emphasizing speed over accuracy. These (still quite accurate) approximate results can then be used to help reduce the workload of calculations over finer discretizations if more accuracy is required.

```
LDPCBinarySymmetricThreshold(v, c)
```

```
LDPCBinarySymmetricThreshold(Sv, Sc)
```

`Precision` RngReSubElt *Default :* 0.00005

Determines the threshold of the described ensemble of LDPC codes over the binary symmetric channel, which is the critical value of the channel parameter above which there is a non-vanishing error probability (asymptotically). The ensemble can either be defined by two integers for (v, c)-regular LDPC codes, or by two density distributions Sv and Sc, which are sequences of non-negative real numbers. The density distributions do not ned to be normalized, though the first entry (corresponding to weight 1 nodes in the Tanner graph) should always be zero.

The computation proceeds by establishing lower and upper bounds on the threshold, then narrowing this range by repeatedly performing density evolution on the midpoint. The argument `Precision` controls the precision to which the threshold is desired.

```
DensityEvolutionBinarySymmetric(v, c, p)
```

```
DensityEvolutionBinarySymmetric(Sv, Sc, p)
```

Perform density evolution on the binary symmetric channel using channel parameter p and determine the asymptotic behaviour for the given LDPC ensemble. The return value is boolean, where `true` indicates that p is below the threshold and the ensemble has error probability asymptotically tending to zero.

Example H160E6

Density evolution on the binary symmetric channel is not computationally intensive.

```
> time LDPCBinarySymmetricThreshold(3, 6);
0.0394140625000000000000000000000
Time: 0.010
> time LDPCBinarySymmetricThreshold(4, 8);
0.0473925781250000000000000000001
Time: 0.110
> time LDPCBinarySymmetricThreshold(4, 10);
0.0368359375000000000000000000000
Time: 0.090
```

```
LDPCGaussianThreshold(v, c)
LDPCGaussianThreshold(Sv, Sc)
```

`Lower`	RngReSubElt	*Default* : 0
`Upper`	RngReSubElt	*Default* : ∞
`Points`	RngIntElt	*Default* : 500
`MaxLLR`	RngReSubElt	*Default* : 25
`MaxIterations`	RngIntElt	*Default* : ∞
`QuickCheck`	BoolElt	*Default* : `true`
`Precision`	RngReSubElt	*Default* : 0.00005

Determines the threshold of the described ensemble of LDPC codes over the Gaussian channel, which is the critical value of the standard deviation above which there is a non-vanishing error probability (asymptotically). The ensemble can either be defined by two integers for (v, c)-regular LDPC codes, or by two density distributions Sv and Sc, which are sequences of non-negative real numbers. The density distributions do not ned to be normalized, though the first entry (corresponding to weight 1 nodes in the Tanner graph) should always be zero.

The computation proceeds by establishing lower and upper bounds on the threshold, then narrowing this range by repeatedly performing density evolution on the midpoint. If the threshold is approximately known then manually setting tight `Lower` and `Upper` bounds can reduce the length of the calculation.

The speed with which these evolutions are computed depends on how fine the discretization is, controlled by the variable argument `Points`. If the threshold is needed to high levels of accuracy then an initial computation with fewer points is recommended to get a reduced searched range. The specific meaning of eachvariable argument is described below.

`Lower` and `Upper` are real-valued bounds on the threshold, which (if tight) can help to reduce the search range and speed up the threshold determination. The validity of an input bound is verified before the search begins, and an error is returned if it is incorrect.

`Points` and `MaxLLR` define the discretized basis of log likelihood ratios on which density evolution is performed, and have integer and real values resp. Specifically, the probability mass function is defined on the range $[-\texttt{MaxLLR}, \ldots, \texttt{MaxLLR}]$ on $2 * Points + 1$ discretized points.

`MaxIterations` allows the user to set a finite limit of iterations that a density evolution should perform in determining the asymptotic behaviour at each channel parameter. Although this may help reduce the time of a computation, it should be kept in mind that the result may not be valid.

`QuickCheck` defines the method by which the asymptotic behaviour at each channel parameter is identified. If set to `false`, then the probability density must evolve all the way to within an infinitesimal value of unity. When set to `true`, if the rate of change of the probability density is seen to be successively increasing then the asymptotic behaviour is assumed to go to unity. Empirically this method seems to give accurate results and so the default behaviour is `true`, however it has no theoretical justification.

`Precision` is a real-valued parameter defining the precision to which the threshold should be determined.

Setting the verbose mode `Code` prints out the bounds on the threshold as subsequent density evolutions narrow the search range.

```
DensityEvolutionGaussian(v, c, σ)
DensityEvolutionGaussian(Sv, Sc, σ)
```

`Points`	RngIntElt	*Default :* 500
`MaxLLR`	RngReSubElt	*Default :* 25
`MaxIterations`	RngIntElt	*Default :* ∞
`QuickCheck`	BoolElt	*Default :* `true`

Perform density evolution on the Gaussian channel using standard deviation σ and determine the asymptotic behaviour for the given LDPC ensemble. The return value is boolean, where `true` indicates that σ is below the threshold and the ensemble has error probability asymptotically tending to zero.

See the description of `LDPCGaussianThreshold` for a description of the variable arguments.

```
GoodLDPCEnsemble(i)
```

Access a small database of density distributions defining good irregular LDPC ensembles. Returned is the published threshold of the ensemble over the Gaussian channel, along with the variable and check degree distributions. The input i is a non-negative integer, which indexes the database (in no particular order).

Example H160E7

Since performing density evolution on a large number of discrete points is time consuming, it is normally better to first get an estimate with an easier computation.

In this example a published value of the threshold of an ensemble (obtained using a different implementation) can be compared to the outputs from different levels of discretization.

```
> thresh, Sv, Sc := GoodLDPCEnsemble(5);
> R4 := RealField(4);
> [R4| x : x in Sv];
[ 0.0000, 0.3001, 0.2839, 0.0000, 0.0000, 0.0000, 0.0000, 0.4159 ]
> [R4| x : x in Sc];
[ 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.2292, 0.7708 ]
> thresh;
0.949700000000000000000000000000
> time approx1 := LDPCGaussianThreshold(Sv, Sc:
>         Points := 500, Precision := 0.001);
Time: 19.400
> approx1;
```

```
0.960976562499999964472863211995
> time approx2 := LDPCGaussianThreshold(Sv, Sc:
>       Points := 3000,
>       Lower := approx1-0.02, Upper := approx1+0.02);
Time: 873.560
> approx2;
0.951210937499999964472863211995
```

This estimate can now be used to narrow the search range of accurate density evolution. For the very long calculation the verbose mode is used to keep track of the progress of the calculation.

```
> SetVerbose("Code", true);
> time approx3 := LDPCGaussianThreshold(Sv, Sc:
>        Points := 5000,
>        Lower := approx2-0.005, Upper := approx2+0.0005);
Determining the mapping matrix...
                        ...mapping matrix obtained  19.10s
Threshold Determination for LDPC code ensemble:
c:  (6):0.229190 (7):0.770810
v:  (2):0.300130 (3):0.283950 (8):0.415920
will be found to precision 0.000050
Max LLR is 25.000000 distributed across 5000 points
Asymptotic behaviour determination is: fast
Beginning search with lb = 0.946211, ub = 0.951711
New Bounds: lb = 0.948961, ub = 0.951711  114.41s
New Bounds: lb = 0.950336, ub = 0.951711  367.19s
New Bounds: lb = 0.950336, ub = 0.951023  553.95s
New Bounds: lb = 0.950336, ub = 0.950680  814.35s
New Bounds: lb = 0.950508, ub = 0.950680  1261.46s
New Bounds: lb = 0.950508, ub = 0.950594  1557.46s
New Bounds: lb = 0.950508, ub = 0.950551  1891.52s
Time: 2136.150
```

The thresholds given in the database are published values taken from other implementations, and so are not guaranteed to match up exactly with the values obtained using Magma.

161 LINEAR CODES OVER FINITE RINGS

Chapter 161
LINEAR CODES OVER FINITE RINGS

161.1 Introduction

This chapter describes those functions which are applicable to linear codes over finite rings. Magma currently supports the basic facilities for codes over integer residue rings and galois rings, including cyclic codes, constructions, complete weight enumerators and decoding. Much additional functionality specific to codes defined over Z_4, the integers modulo 4, can be found in Chapter 162.

For modules defined over rings with zero divisors, it is of course not possible to talk about the concept of dimension (the modules are not free). But in Magma each code over such a ring has a *unique* generator matrix corresponding to the *Howell form.* The number of rows k in this unique 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 that the rank of the generator matrix is always well-defined and unique (based on the Smith form which is well-defined over PIRs), but k may sometimes be larger than the rank.

Without a concept of dimension, codes over finite rings are referenced by their cardinality. A code C is called an (n, M, d) *code* if it has length n, cardinality M and minimum Hamming weight d.

The reader is referred to [Wan97] as a general reference on codes over Galois rings, especially linear codes over Z_4.

In this chapter, as for codes over finite fields, the term “code” will refer to a linear code, unless otherwise specified.

161.2 Constructions

161.2.1 General Linear Codes

```
LinearCode< R, n | L >
```

Create a code as a subspace of the R-space $V = R^{(n)}$ which is generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) An element of V.

(b) A set or sequence of elements of V.

(c) A sequence of n elements of R, defining an element of V.

(d) A set or sequence of sequences of type (c).

(e) A subspace of V.

(f) A set or sequence of subspaces of V.

`LinearCode(U)`

Let V be the R-space $R^{(n)}$ and suppose that U is a subspace of V. The effect of this function is to define the linear code C corresponding to the subspace U.

`LinearCode(A)`

Given a $k \times n$ matrix A over the ring R, construct the linear code generated by the rows of A. Note that it is not assumed that the rank of A is k. The effect of this constructor is otherwise identical to that described above.

`PermutationCode(u, G)`

Given a finite permutation group G of degree n, and a vector u belonging to the n-dimensional vector space V over the ring R, construct the code C corresponding to the subspace of V spanned by the set of vectors obtained by applying the permutations of G to the vector u.

Example H161E1

The octacode O_8 over $\mathbf{Z}_4$ [Wan97, Ex. 1.3] can be defined as follows:

```
> Z4 := IntegerRing(4);
> O8 := LinearCode<Z4, 8 |
>     [1,0,0,0,3,1,2,1],
>     [0,1,0,0,1,2,3,1],
>     [0,0,1,0,3,3,3,2],
>     [0,0,0,1,2,3,1,1]>;
> O8;
[8, 4, 4] Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 1 2 1]
[0 1 0 0 1 2 3 1]
[0 0 1 0 3 3 3 2]
[0 0 0 1 2 3 1 1]
```

Alternatively, if we want to see the code as a subspace of $R^{(8)}$, where $R = \mathbf{Z}_4$, we could proceed as follows:

```
> O8 := LinearCode(sub<RSpace(Z4, 8) |
>     [1,0,0,0,3,1,2,1],
>     [0,1,0,0,1,2,3,1],
>     [0,0,1,0,3,3,3,2],
>     [0,0,0,1,2,3,1,1]>);
```

Example H161E2

We define a code by constructing a matrix over GR$(4, 3)$, and using its rowspace to generate the code:

```
> R<w> := GaloisRing(4,3);
> S := [1, 1, 0, w^2, w, w + 2, 2*w^2, 2*w^2 + w + 3];
> G := Matrix(R, 2, 4, S);
> G;
[            1             1             0           w^2]
[            w         w + 2         2*w^2 2*w^2 + w + 3]
> C := LinearCode(G);
> C;
(4, 512, 3) Linear Code over GaloisRing(2, 2, 3)
Generator matrix:
[          1           1           0         w^2]
[          0           2       2*w^2 2*w^2 + 2*w]
> #C;
512
```

Example H161E3

We define G to be a permutation group of degree 7 and construct the code C as the $\mathbf{Z}_4$-code generated by applying the permutations of G to a certain vector:

```
> G := PSL(3, 2);
> G;
Permutation group G of degree 7
    (1, 4)(6, 7)
    (1, 3, 2)(4, 7, 5)
> Z4 := IntegerRing(4);
> V := RSpace(Z4, 7);
> u := V ! [1, 0, 0, 1, 0, 1, 1];
> C := PermutationCode(u, G);
> C;
[7, 6, 2] Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 1 0 1 1]
[0 1 0 1 1 1 0]
[0 0 1 0 1 1 1]
[0 0 0 2 0 0 2]
[0 0 0 0 2 0 2]
[0 0 0 0 0 2 2]
```

161.2.2 Simple Linear Codes

`ZeroCode(R, n)`

Given a ring R and positive integer n, return the $(n, 0, n)$ code consisting of only the zero code word. By convention the minimum weight of the zero code is n).

`RepetitionCode(R, n)`

Given a ring R and positive integer n, return the length n code with minimum Hamming weight n, generated by the all-ones vector.

`ZeroSumCode(R, n)`

Given a ring R and positive integer n, return the length n code over R such that for all codewords $(c_1, c_2, \ldots, c_n)$ we have $\sum_i c_i = 0$.

`UniverseCode(R, n)`

Given a ring R and positive integer n, return the length n code with minimum Hamming weight 1, consisting of all possible codewords.

`RandomLinearCode(R, n, k)`

Given a finite ring R and positive integers n and k, such that $0 < k \leq n$, the function returns a random linear code of length n over R with k generators.

Example H161E4

The repetition and zero sum codes are dual over all rings.

```
> R := Integers(9);
> C1 := RepetitionCode(R, 5);
> C1;
(5, 9, 5) Linear Code over IntegerRing(9)
Generator matrix:
[1 1 1 1 1]
> C2 := ZeroSumCode(R, 5);
> C2;
(5, 6561, 2) Linear Code over IntegerRing(9)
Generator matrix:
[1 0 0 0 8]
[0 1 0 0 8]
[0 0 1 0 8]
[0 0 0 1 8]
> C1 eq Dual(C2);
true
```

161.2.3 General Cyclic Codes

Cyclic codes form an important family of linear codes over all rings. A cyclic code is one which is generated by all of the cyclic shifts of a given codeword:

$$(c_0, c_1, \dots, c_{n-1}, c_n), (c_n, c_0, \dots, c_{n-2}, c_{n-1}), \dots, (c_1, c_2, \dots, c_n, c_0)$$

Using the correspondence $(c_0, c_1, \dots, c_n) \iff c_0 + c_1x + \cdots + c_nx^n$, the cyclic codes of length n over the ring R are in one-to-one correspondence with the principal ideals of $R[x]/(x^n - 1)R[x]$.

```
CyclicCode(u)
```

Given a vector u belonging to the R-space $R^{(n)}$, construct the length n cyclic code generated by the right cyclic shifts of the vector u.

```
CyclicCode(n, g)
```

Let R be a ring. Given a positive integer n and a univariate polynomial $g(x) \in R[x]$, construct the length n cyclic code generated by $g(x)$.

```
CyclotomicFactors(R, n)
```

Given a Galois ring R (which is possibly an integer residue ring with a prime power modulus), and a positive integer n which is coprime to the characteristic of R, return a factorisation of $x^n - 1$ over R.

Note that since factorisation is not necessarily unique over R, the factorisation returned is the one obtained by first factoring over the residue field of R and then performing Hensel lifting.

Example H161E5

We construct some cyclic codes over $\mathbf{Z}_4$ by factorizing $x^n - 1$ over $\mathbf{Z}_4$ for $n = 7, 23$ and using some of the irreducible factors found.

```
> Z4 := IntegerRing(4);
> P<x> := PolynomialRing(Z4);
> n := 7; L := CyclotomicFactors(Z4, n); L;
[
    x + 3,
    x^3 + 2*x^2 + x + 3,
    x^3 + 3*x^2 + 2*x + 3
]
> CyclicCode(n, L[1]);
[7, 6, 2] Cyclic Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 0 3]
[0 1 0 0 0 0 3]
[0 0 1 0 0 0 3]
[0 0 0 1 0 0 3]
```

```
[0 0 0 0 1 0 3]
[0 0 0 0 0 1 3]
> CyclicCode(n, L[2]);
[7, 4, 3] Cyclic Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 1 2]
[0 1 0 0 2 1 1]
[0 0 1 0 1 1 3]
[0 0 0 1 3 2 3]
> CyclicCode(n, L[3]);
[7, 4, 3] Cyclic Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 2 3]
[0 1 0 0 3 1 1]
[0 0 1 0 1 1 2]
[0 0 0 1 2 1 3]
> n := 23; L := CyclotomicFactors(Z4, n); L;
[
    x + 3,
    x^11 + 2*x^10 + 3*x^9 + 3*x^7 + 3*x^6 + 3*x^5 + 2*x^4 + x + 3,
    x^11 + 3*x^10 + 2*x^7 + x^6 + x^5 + x^4 + x^2 + 2*x + 3
]
> CyclicCode(n, L[2]);
[23, 12] Cyclic Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 3 1 0 0 2 3 3 3 0 3 2]
[0 1 0 0 0 0 0 0 0 0 0 0 2 1 1 0 0 0 1 1 3 2 3]
[0 0 1 0 0 0 0 0 0 0 0 0 3 3 1 1 2 3 3 0 1 2 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 3 3 1 1 2 3 3 0 1 2]
[0 0 0 0 1 0 0 0 0 0 0 0 2 2 3 3 1 3 0 1 3 2 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 1 2 3 1 2 0 1 1 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 2 3 1 2 0 1 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 2 3 1 2 0 1 1]
[0 0 0 0 0 0 0 0 1 0 0 0 1 3 0 1 3 3 0 2 2 1 3]
[0 0 0 0 0 0 0 0 0 1 0 0 3 2 3 0 3 2 2 3 2 1 3]
[0 0 0 0 0 0 0 0 0 0 1 0 3 0 2 3 2 2 1 1 3 1 3]
[0 0 0 0 0 0 0 0 0 0 0 1 3 0 0 2 1 1 1 0 1 2 3]
```

Example H161E6

We create a cyclic code of length 5 over GR$(4, 2)$.

```
> R<w> := GR(4,2);
> P<x> := PolynomialRing(R);
> g := CyclotomicFactors(R, 5)[2];
> g;
x^2 + (3*w + 2)*x + 1
> C := CyclicCode(5, g);
> C;
(5, 4096, 3) Cyclic Code over GaloisRing(2, 2, 2)
Generator matrix:
[      1       0       0       1 3*w + 2]
[      0       1       0   w + 2   w + 2]
[      0       0       1 3*w + 2       1]
```

161.3 Invariants

`#C`

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

`C . i`

`Name(C, i)`

Given a code C and a positive integer i, return the i-th generator of C.

`Alphabet(C)`

The underlying ring (or alphabet) R of the code C.

`AmbientSpace(C)`

The ambient space of the code C, i.e., the generic R-space V in which C is contained.

`Basis(C)`

The basis of the linear code C, returned as a sequence of elements of C.

`Generators(C)`

The generators for the linear code C, returned as a set.

`GeneratorMatrix(C)`

The generator matrix for the linear code C. This gives a unique canonical generating set for the code.

`Generic(C)`

Given a length n code C over a ring R, return the generic $(n, \#R^n, 1)$ code in which C is contained.

`Length(C)`

Given a code C, return the block length n of C.

`PseudoDimension(C)`

`NumberOfGenerators(C)`

`Ngens(C)`

The number of generators (which equals the pseudo-dimension k) of the linear code C.

`ParityCheckMatrix(C)`

The parity check matrix for the code C, which can be defined as the canonical generator matrix of the dual of C.

`Random(C)`

A random codeword of the code C.

`RSpace(C)`

Given a length n linear code C, defined as a subspace U of the n-dimensional space V, return U as a subspace of V with basis corresponding to the rows of the generator matrix for C.

`InformationRate(C)`

Given a code C over a ring with cardinality q, return the information rate of C, that is, the ratio $Log_q(\#C)/n$.

161.4 Subcodes

161.4.1 The Subcode Constructor

`sub< C | L >`

Given a length n linear code C over R, construct the subcode of C, generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) An element of C;

(b) A set or sequence of elements of C;

(c) A sequence of n elements of R, defining an element of C;

(d) A set or sequence of sequences of type (c);

(e) A subcode of C;

(f) A set or sequence of subcodes of C.

`Subcode(C, t)`

Given a length n linear code C with k generators and an integer t, $1 \leq t < k$, return a subcode of C of pseudo-dimension t.

`Subcode(C, S)`

Given a length n linear code C with k generators and a set S of integers, each of which lies in the range $[1, k]$, return the subcode of C generated by the basis elements whose positions appear in S.

Example H161E7

We construct a subcode of a code over a Galois ring by multiplying each of its generators by a zero divisor.

```
> R<w> := GR(4,2);
> C := RandomLinearCode(R, 4, 2);
> C;
(4, 256, 3) Linear Code over GaloisRing(2, 2, 2)
Generator matrix:
[      1       0   w + 1 3*w + 2]
[      0       1 3*w + 1       1]
> #C;
256
>
> C1 := sub< C | 2*C.1, 2*C.2 >;
> C1;
(4, 16, 3) Linear Code over GaloisRing(2, 2, 2)
Generator matrix:
[      2       0 2*w + 2     2*w]
[      0       2 2*w + 2       2]
> #C1;
16
```

161.5 Boolean Predicates

For the following operators, C and D are codes defined as a subset (or subspace) of the R-space V.

`u in C`

Return `true` if and only if the vector u of V belongs to the code C.

`u notin C`

Return `true` if and only if the vector u of V does not belong to the code C.

`C subset D`

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

`C notsubset D`

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

`C eq D`

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

`C ne D`

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

`IsCyclic(C)`

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

`IsSelfDual(C)`

Return `true` if and only if the linear code C is self-dual (or self-orthogonal) (i.e., C equals the dual of C).

`IsSelfOrthogonal(C)`

Return `true` if and only if the linear code C is self-orthogonal; that is, return whether C is contained in the dual of C.

`IsProjective(C)`

Returns `true` if and only if the (non-quantum) code C is projective.

`IsZero(u)`

Return `true` if and only if the codeword u is the zero vector.

Example H161E8

We consider an [8, 7] linear code K_8 over Z_4 and examine some of its properties.

```
> Z4 := IntegerRing(4);
> K8 := LinearCode< Z4, 8 |
>     [1,1,1,1,1,1,1,1],
>     [0,2,0,0,0,0,0,2],
>     [0,0,2,0,0,0,0,2],
>     [0,0,0,2,0,0,0,2],
>     [0,0,0,0,2,0,0,2],
>     [0,0,0,0,0,2,0,2],
>     [0,0,0,0,0,0,2,2]>;
> K8;
[8, 7, 2] Linear Code over IntegerRing(4)
Generator matrix:
[1 1 1 1 1 1 1 1]
[0 2 0 0 0 0 0 2]
[0 0 2 0 0 0 0 2]
[0 0 0 2 0 0 0 2]
[0 0 0 0 2 0 0 2]
[0 0 0 0 0 2 0 2]
[0 0 0 0 0 0 2 2]
> IsCyclic(K8);
```

```
true
> IsSelfDual(K8);
true
> K8 eq Dual(K8);
true
```

161.6 New Codes from Old

The operations described here produce a new code by modifying in some way the code words of a given code.

161.6.1 Sum, Intersection and Dual

For the following operators, C and D are codes defined as subsets (or subspaces) of the same R-space V.

`C + D`

The (vector space) sum of the linear codes C and D, where C and D are contained in the same R-space V.

`C meet D`

The intersection of the linear codes C and D, where C and D are contained in the same R-space V.

`Dual(C)`

The dual D of the linear code C. The dual consists of all codewords in the R-space V which are orthogonal to all codewords of C.

Example H161E9

Verify some simple results from the sum and intersection of subcodes.

```
> R<w> := GR(9,2);
> P<x> := PolynomialRing(R);
> g := x^2 + 7*w*x + 1;
> C := CyclicCode(5, g);
> C;
(5, 43046721) Cyclic Code over GaloisRing(3, 2, 2)
Generator matrix:
[  1   0   0   1   w]
[  0   1   0 2*w 2*w]
[  0   0   1   w   1]
[  0   0   0   3   0]
[  0   0   0   0   3]
>
> C1 := sub< C | C.1 >;
> C1;
```

```
(5, 81, 3) Linear Code over GaloisRing(3, 2, 2)
Generator matrix:
[1 0 0 1 w]
> C2 := sub< C | C.4 >;
> C2;
(5, 9, 1) Linear Code over GaloisRing(3, 2, 2)
Generator matrix:
[0 0 0 3 0]
> C3 := sub< C | { C.1 , C.4} >;
> C3;
(5, 729, 1) Linear Code over GaloisRing(3, 2, 2)
Generator matrix:
[1 0 0 1 w]
[0 0 0 3 0]
> (C1 + C2) eq C3;
true
> (C1 meet C3) eq C1;
true
```

161.6.2 Standard Constructions

`DirectSum(C, D)`

Given a length n_1 code C and a length n_2 code D, both over the same ring R, construct the direct sum of C and D. The direct sum consists of all length $n_1 + n_2$ vectors $u|v$, where $u \in C$ and $v \in D$.

`DirectProduct(C, D)`

Given a length n_1 code C and a length n_2 code D, both over the same ring R, construct the direct product of C and D. The direct product has length $n_1 \cdot n_2$ and its generator matrix is the Kronecker product of the basis matrices of C and D.

`C1 cat C2`

Given codes $C1$ and $C2$, both defined over the same ring R, return the concatenation C of $C1$ and $C2$. If A and B are the generator matrices of $C1$ and $C2$, respectively, the concatenation of $C1$ and $C2$ is the code with generator matrix whose rows consist of each row of A concatenated with each row of B.

`ExtendCode(C)`

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

`ExtendCode(C, n)`

Return the code C extended n times.

`PadCode(C, n)`

Add n zeros to the end of each codeword of C.

`PlotkinSum(C, D)`

Given codes C and D both over the same ring R and of the same length n, construct the Plotkin sum of C and D. The Plotkin sum consists of all vectors $u|u+v$, $u \in C$ and $v \in D$.

`PunctureCode(C, i)`

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

`PunctureCode(C, S)`

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

`ShortenCode(C, i)`

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

`ShortenCode(C, S)`

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

Example H161E10

We combine codes in various ways and look at the length of the new code.

```
> R<w> := GR(8,2);
> C1 := RandomLinearCode(R, 4, 2);
> C2 := RandomLinearCode(R, 5, 3);
> Length(C1);
4
> Length(C2);
5
> C3 := DirectSum(C1, C2);
> Length(C3);
9
> C4 := DirectProduct(C1, C2);
> Length(C4);
20
> C5 := C1 cat C2;
> Length(C5);
9
```

Example H161E11

We note that, in general, puncturing a code over $\mathbf{Z}_4$ reduces the minimum Lee distance by 2.

```
> C := PreparataCode(3);
> C;
(8, 256, 4) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 1 2 1]
[0 1 0 0 2 1 1 3]
[0 0 1 0 1 1 3 2]
[0 0 0 1 3 2 3 3]
> MinimumLeeWeight(C);
6
> C1 := PunctureCode(C,8);
> C1;
(7, 256, 3) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 1 2]
[0 1 0 0 2 1 1]
[0 0 1 0 1 1 3]
[0 0 0 1 3 2 3]
> MinimumLeeWeight(C1);
4
```

161.7 Codeword Operations

161.7.1 Construction

`C ! [a_1, ..., a_n]`

`elt< C | a_1, ..., a_n >`

Given a code C which is defined as a subset of the R-space $R^{(n)}$, and elements $a_1, \ldots, a_n$ belonging to R, construct the codeword $(a_1, \ldots, a_n)$ of C. It is checked that the vector $(a_1, \ldots, a_n)$ is an element of C.

`C ! u`

Given a code C which is defined as a subset of the R-space $V = R^{(n)}$, and an element u belonging to V, create the codeword of C corresponding to u. The function will fail if u does not belong to C.

`C ! 0`

The zero word of the code C.

Example H161E12

We create some elements of a code over a finite ring.

```
> R<w> := GR(16,2);
> P<x> := PolynomialRing(R);
> L := CyclotomicFactors(R, 7);
> C := CyclicCode(7, L[2]);
> C ! [1, 2*w, 0, w+3, 7*w, 12*w+3, w+3];
(       1      2*w        0    w + 3      7*w 12*w + 3    w + 3)
> elt< C | 0, 3, 0, 2*w + 5, 6*w + 9, 4*w + 5, 14*w + 14 >;
(        0         3         0   2*w + 5   6*w + 9   4*w + 5 14*w + 14)
```

If the given vector does not lie in the given code then an error will result.

```
>  C ! [0,0,0,0,0,0,1];
>> C ! [0,0,0,0,0,0,1];
     ^
Runtime error in '!': Result is not in the given structure
>  elt< C | 1, 0, 1, 0, 1, 0, 1>;
>> elt< C | 1, 0, 1, 0, 1, 0, 1>;
      ^
Runtime error in elt< ... >: Result is not in the lhs of the constructor
```

161.7.2 Operations

`u + v`

Sum of the codewords u and v, where u and v belong to the same linear code C.

`-u`

Additive inverse of the codeword u belonging to the linear code C.

`u - v`

Difference of the codewords u and v, where u and v belong to the same linear code C.

`a * u`

Given an element a belonging to the ring R, and a codeword u belonging to the linear code C, return the codeword $a * u$.

`Weight(v)`

The Hamming weight of the codeword v, i.e., the number of non-zero components of v.

`Distance(u, v)`

The Hamming distance between the codewords u and v, where u and v belong to the same code C.

`Support(w)`

Given a word w belonging to the length n code C, return its support as a subset of the integer set $\{1..n\}$. The support of w consists of the coordinates at which w has non-zero entries.

`(u, v)`

`InnerProduct(u, v)`

Inner product of the vectors u and v with respect to the Euclidean norm, where u and v belong to the parent vector space of the code C.

`Coordinates(C, u)`

Given a length n linear 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 the alphabet of C so that $u = a_1 * C.1 + \ldots + a_k * C.k$.

`Normalize(u)`

Given an element u of a code defined over the ring R, return the normalization of u, which is the unique vector v such that $v = a \cdot u$ for some scalar $a \in R$ such that the first non-zero entry of v is the canonical associate in R of the first non-zero entry of u (v is zero if u is zero).

`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.

`Rotate(~u, k)`

Given a vector u, destructively rotate u by k coordinate positions.

`Parent(w)`

Given a word w belonging to the code C, return the ambient space V of C.

Example H161E13

Given a code over a finite ring, we explore various operations on its code words.

```
> R<w> := GR(4, 4);
> P<x> := PolynomialRing(R);
> g := x + 2*w^3 + 3*w^2 + w + 2;
> C := CyclicCode(3, g);
> C;
(3, 1048576) Cyclic Code over GaloisRing(2, 2, 4)
Generator matrix:
[          1           0     w^2 + w]
[          0           1 w^2 + w + 1]
[          0           0           2]
> u := C.1;
> v := C.2;
> u;
(      1       0 w^2 + w)
> v;
(          0           1 w^2 + w + 1)
> u + v;
(              1               1 2*w^2 + 2*w + 1)
> 2*u;
(          2           0 2*w^2 + 2*w)
> 4*u;
(0 0 0)
> Weight(u);
2
> Support(u);
{ 1, 3 }
```

161.7.3 Accessing Components of a Codeword

```
u[i]
```

Given a codeword u belonging to the code C defined over the ring R, return the i-th component of u (as an element of R).

```
u[i] := x;
```

Given an element u belonging to a subcode C of the full R-space $V = R^n$, a positive integer i, $1 \leq i \leq n$, and an element x of R, this function returns a vector in V which is u with its i-th component redefined to be x.

161.8 Weight Distributions

In the case of a linear code, weight and distance distributions are equivalent (in particular minimum weight and minimum distance are equivalent).

161.8.1 Hamming Weight

For an element $x \in \mathbf{R}$ for any finite ring R, the *Hamming weight* $w_H(x)$ is defined by:

$$w_H(x) = 0 \iff x = 0, \qquad w_H(x) = 1 \iff x \neq 0$$

The *Hamming weight* $w_H(v)$ of a vector $v \in R^n$ is defined to be the sum (in $\mathbf{Z}$) of the Hamming weights of its components.

The *Hamming weight* is often referred to as simply the *weight*.

`MinimumWeight(C)`

`MinimumDistance(C)`

Determine the minimum (Hamming) weight of the words belonging to the code C, which is also the minimum distance between any two codewords.

`WeightDistribution(C)`

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

`DualWeightDistribution(C)`

The (Hamming) weight distribution of the dual code of C. For more explanation, see `WeightDistribution`.

Example H161E14

We calculate the weight distribution of a cyclic code over the Galois ring of size 81.

```
> R<w> := GR(9,2);
> P<x> := PolynomialRing(R);
> L := CyclotomicFactors(R, 4);
> g := L[3] * L[4];
> g;
x^2 + (8*w + 7)*x + w + 1
> C := CyclicCode(4, g);
> C;
(4, 6561, 3) Cyclic Code over GaloisRing(3, 2, 2)
Generator matrix:
[      1       0   w + 1 8*w + 7]
[      0       1       w 8*w + 8]
> WeightDistribution(C);
[ <0, 1>, <3, 320>, <4, 6240> ]
```

161.9 Weight Enumerators

`CompleteWeightEnumerator(C)`

Let C be a code over a finite ring R of cardinality q, and suppose that the elements of R are ordered in some way. Then for a codeword $v \in C$ and the i-th element $a \in R$, let $s_i(v)$ denote the number of components of v equal to a.

This function returns the complete weight enumerator $\mathcal{W}_C(X_0, X_1, \dots, X_{q-1})$ of C, which is defined by:

$$\mathcal{W}_C(X_0, X_1, \dots, X_{q-1}) = \sum_{v \in C} {X_0}^{s_0(v)} {X_1}^{s_1(v)} \cdots {X_{q-1}}^{s_{q-1}(v)}.$$

See [Wan97, p. 9] for more information. The result will lie in a global multivariate polynomial ring over $\mathbf{Z}$ with q variables. The angle-bracket notation may be used to assign names to the indeterminates.

`WeightEnumerator(C)`

`HammingWeightEnumerator(C)`

Suppose C is a code over some finite ring R. This function returns the Hamming weight enumerator $\mathrm{Ham}_C(X, Y)$ of C, which is defined by:

$$\mathrm{Ham}_C(X, Y) = \sum_{v \in C} X^{n - w_H(v)} Y^{w_H(v)},$$

where $w_H(v)$ is the Hamming weight function. The result will lie in a global multivariate polynomial ring over $\mathbf{Z}$ with two variables. The angle-bracket notation may be used to assign names to the indeterminates.

Example H161E15

We compute the complete weight enumerator of a cyclic code over the Galois ring GR(4, 2).

```
> R<w> := GR(4,2);
> P<x> := PolynomialRing(R);
> L := CyclotomicFactors(R, 3);
> g := L[1];
> g;
x + 3
> C := CyclicCode(3, g);
> C;
(3, 256, 2) Cyclic Code over GaloisRing(2, 2, 2)
Generator matrix:
[1 0 3]
[0 1 3]
> CWE<[X]> := CompleteWeightEnumerator(C);
> CWE;
```

```
X[1]^3 + 6*X[1]*X[2]*X[4] + 3*X[1]*X[3]^2 + 6*X[1]*X[5]*X[13] +
    6*X[1]*X[6]*X[16] + 6*X[1]*X[7]*X[15] + 6*X[1]*X[8]*X[14] +
    3*X[1]*X[9]^2 + 6*X[1]*X[10]*X[12] + 3*X[1]*X[11]^2 +
    3*X[2]^2*X[3] + 6*X[2]*X[5]*X[16] + 6*X[2]*X[6]*X[15] +
    6*X[2]*X[7]*X[14] + 6*X[2]*X[8]*X[13] + 6*X[2]*X[9]*X[12] +
    6*X[2]*X[10]*X[11] + 3*X[3]*X[4]^2 + 6*X[3]*X[5]*X[15] +
    6*X[3]*X[6]*X[14] + 6*X[3]*X[7]*X[13] + 6*X[3]*X[8]*X[16] +
    6*X[3]*X[9]*X[11] + 3*X[3]*X[10]^2 + 3*X[3]*X[12]^2 +
    6*X[4]*X[5]*X[14] + 6*X[4]*X[6]*X[13] + 6*X[4]*X[7]*X[16] +
    6*X[4]*X[8]*X[15] + 6*X[4]*X[9]*X[10] + 6*X[4]*X[11]*X[12] +
    3*X[5]^2*X[9] + 6*X[5]*X[6]*X[12] + 6*X[5]*X[7]*X[11] +
    6*X[5]*X[8]*X[10] + 3*X[6]^2*X[11] + 6*X[6]*X[7]*X[10] +
    6*X[6]*X[8]*X[9] + 3*X[7]^2*X[9] + 6*X[7]*X[8]*X[12] +
    3*X[8]^2*X[11] + 3*X[9]*X[13]^2 + 6*X[9]*X[14]*X[16] +
    3*X[9]*X[15]^2 + 6*X[10]*X[13]*X[16] + 6*X[10]*X[14]*X[15] +
    6*X[11]*X[13]*X[15] + 3*X[11]*X[14]^2 + 3*X[11]*X[16]^2 +
    6*X[12]*X[13]*X[14] + 6*X[12]*X[15]*X[16]
```

161.10 Bibliography

[**Wan97**] Zhe-Xian Wan. *Quaternary Codes*, volume 8 of *Series on Applied Mathematics*. World Scientific, Singapore, 1997.

162 LINEAR CODES OVER THE INTEGER RESIDUE RING $\mathbf{Z}_4$

Chapter 162

LINEAR CODES OVER THE INTEGER RESIDUE RING $\mathbf{Z}_4$

162.1 Introduction

In Chapter 161 basic functions for working with codes over a finite ring are described. Because of the large amount of machinery developed specifically for $\mathbf{Z}_4$-codes, this chapter will be devoted to this special case. The functionality available for $\mathbf{Z}_4$-codes consists of the machinery described in chapter 161 together with the contents of this chapter.

This chapter includes constructions for some families of codes over $\mathbf{Z}_4$ (see sections 162.2.2 and 162.2.4), efficient functions for computing the rank and dimension of the kernel of any code over $\mathbf{Z}_4$ (Section 162.3.2), as well as general functions for computing coset representatives for a subcode in a code over $\mathbf{Z}_4$ (Section 162.3.3). In addition, there are functions for computing the permutation automorphism group for Hadamard and extended perfect codes over $\mathbf{Z}_4$, and their cardinal (Section 162.7). Finally, various algorithms for decoding codes over $\mathbf{Z}_4$ are also provided (Section 162.6).

Error correcting codes over $\mathbf{Z}_4$ are often referred to as *quaternary* codes. Important concepts when discussing quaternary codes are *Lee weight* and the *Gray map*, which maps linear codes over $\mathbf{Z}_4$ to (possibly non-linear) codes over $\mathbf{Z}_2$. Many good non-linear binary codes can be defined as the images of simple linear quaternary codes. A code over $\mathbf{Z}_4$ is a subgroup of $\mathbf{Z}_4^n$, so it is isomorphic to an abelian structure $\mathbf{Z}_2^\gamma \times \mathbf{Z}_4^\delta$ and we will say that it is of type $2^\gamma 4^\delta$, or simply that it has $2^{\gamma+2\delta}$ codewords. As general references on the available functions in Magma for codes over $\mathbf{Z}_4$, the reader is referred to [HKC$^+$94, Wan97].

For general references on the material in this chapter, the reader is referred to the bibliography included at the end of this chapter.

The machinery described in this chapter largely corresponds to Version 2.0 of the package *Codes over* $\mathbf{Z}_4$*: A* Magma *Package* which has been developed by Roland D. Barrolleta, Jaume Pernas, Jaume Pujol and Mercè Villanueva of the Combinatoric, Coding and Security Group (CCSG) at the Universitat Autònoma de Barcelona.

162.2 Constructions for $\mathbf{Z}_4$ Codes

162.2.1 The Gray Map

For an element $x \in \mathbf{Z}_4$, the *Gray map* $\phi : \mathbf{Z}_4 \to \mathbf{Z}_2^2$ is defined by:

$$0 \mapsto 00, \quad 1 \mapsto 01, \quad 2 \mapsto 11, \quad 3 \mapsto 10.$$

This map is extended to a map from $\mathbf{Z}_4^n$ onto $\mathbf{Z}_2^{2n}$ in the obvious way (by concatenating the images of each component). The resulting map is a weight- and distance-preserving map from $\mathbf{Z}_4^n$ (with Lee weight metric) to $\mathbf{Z}_2^{2n}$ (with Hamming weight metric). See [Wan97, Chapter 3] for more information (but note that that author uses a different ordering of the components of the image of a vector).

```
GrayMap(C)
```

Given a $\mathbf{Z}_4$-linear code C, this function returns the Gray map for C. This is the map ϕ from C to $\mathbf{F}_2^{2n}$, as defined above.

```
GrayMapImage(C)
```

Given a $\mathbf{Z}_4$-linear code C, this function returns the image of C under the Gray map as a sequence of vectors in $\mathbf{F}_2^{2n}$. As the resulting image may not be a $\mathbf{F}_2$-linear code, a sequence of vectors is returned rather than a code.

```
HasLinearGrayMapImage(C)
```

Given a $\mathbf{Z}_4$-linear code C, this function returns true if and only if the image of C under the Gray map is a $\mathbf{F}_2$-linear code. If so, the function also returns the image B as a $\mathbf{F}_2$-linear code, together with the bijection $\phi : C \to B$.

Example H162E1

Let $\phi(O_8)$ be the image of the octacode O_8 under the Gray map. This image is not a $\mathbf{F}_2$-linear code, but it is the non-linear $(8, 256, 6)$ Nordstrom-Robinson code [Wan97, Ex.3.4]. The statements below demonstrate that the Hamming weight distribution of the $\mathbf{F}_2$ image is identical to the Lee weight distribution of the linear $\mathbf{Z}_4$ code.

```
> Z4 := IntegerRing(4);
> O8 := LinearCode<Z4, 8 |
>     [1,0,0,0,3,1,2,1],
>     [0,1,0,0,1,2,3,1],
>     [0,0,1,0,3,3,3,2],
>     [0,0,0,1,2,3,1,1]>;
> HasLinearGrayMapImage(O8);
false
> NR := GrayMapImage(O8);
> #NR;
256
> LeeWeightDistribution(O8);
[ <0, 1>, <6, 112>, <8, 30>, <10, 112>, <16, 1> ]
> {* Weight(v): v in NR *};
```

```
{* 0, 16, 6^^112, 8^^30, 10^^112 *}
```

After defining the code K_8, the images of some codewords under the Gray map are found.

```
> Z4 := IntegerRing(4);
> K8 := LinearCode< Z4, 8 |
>     [1,1,1,1,1,1,1,1],
>     [0,2,0,0,0,0,0,2],
>     [0,0,2,0,0,0,0,2],
>     [0,0,0,2,0,0,0,2],
>     [0,0,0,0,2,0,0,2],
>     [0,0,0,0,0,2,0,2],
>     [0,0,0,0,0,0,2,2]>;
> f := GrayMap(K8);
> K8.1;
(1 1 1 1 1 1 1 1)
> f(K8.1);
(0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1)
> K8.2;
(0 2 0 0 0 0 0 2)
> f(K8.2);
(0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1)
```

The image of K_8 under the Gray map is a linear code over $\mathbf{F}_2$.

```
> l, B, g := HasLinearGrayMapImage(K8);
> l;
true
> B;
[16, 8, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 0]
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1]
[0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1]
[0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1]
[0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1]
> g(K8.1) in B;
true
```

162.2.2 Families of Codes over $\mathbf{Z}_4$

This section presents some standard constructions for $\mathbf{Z}_4$-linear codes. Further constructions will become available in the near future.

```
KerdockCode(m)
```

Given an integer $m \geq 2$, return the quaternary Kerdock code $K(m)$ of length $2^m - 1$ defined by a default primitive polynomial $h \in \mathbf{Z}_4[x]$ of degree m.

```
PreparataCode(m)
```

Given an integer $m \geq 2$, return the quaternary Preparata code $P(m)$ of length $2^m - 1$ defined by a default primitive polynomial $h \in \mathbf{Z}_4[x]$ of degree m.

```
ReedMullerCodeZ4(r, m)
```

Given an integer $m \geq 2$ and an integer r such that $0 \leq r \leq m$ this function returns the r-th order Reed-Muller code over $\mathbf{Z}_4$ of length 2^m.

```
GoethalsCode(m)
```

Given a positive integer m, where m must be an odd and greater than or equal to 3, return the Goethals code of length 2^m.

```
DelsarteGoethalsCode(m, delta)
```

Return the Delsarte-Goethals Code of length 2^m.

```
GoethalsDelsarteCode(m, delta)
```

Return the Goethals-Delsarte code of length 2^m

```
QRCodeZ4(p)
```

Given a prime number p such that 2 is a quadratic residue modulo p, return the quadratic residue code of length p over $\mathbf{Z}_4$.

```
GolayCodeZ4(e)
```

Return the Golay Code over $\mathbf{Z}_4$. If e is `true` then return the extended Golay Code

```
SimplexAlphaCodeZ4(k)
```

Return the simplex alpha code over $\mathbf{Z}_4$ of degree k.

```
SimplexBetaCodeZ4(k)
```

Return the simplex beta code over $\mathbf{Z}_4$ of degree k.

Example H162E2

The minimum Lee weights of some default Kerdock and Preparata codes are found.

```
> PreparataCode(3);
(8, 256, 4) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 1 2 1]
[0 1 0 0 2 1 1 3]
[0 0 1 0 1 1 3 2]
[0 0 0 1 3 2 3 3]
> MinimumLeeWeight($1);
6
> KerdockCode(4);
[16, 5, 8] Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 1 1 3 0 3 3 0 2 1 2 3]
[0 1 0 0 0 2 3 3 3 2 1 3 0 0 1 1]
[0 0 1 0 0 3 1 0 3 0 3 1 1 3 2 2]
[0 0 0 1 0 2 1 3 0 1 2 3 1 3 3 0]
[0 0 0 0 1 1 3 0 3 3 0 2 1 2 1 3]
> MinimumLeeWeight($1);
12
> KerdockCode(5);
(32, 4096, 16) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 0 3 3 3 2 0 3 2 2 0 3 0 1 0 1 3 1 1 0 3 1 2 3 2 2 3 3]
[0 1 0 0 0 0 3 2 2 1 2 3 1 0 2 3 3 1 1 1 0 0 2 1 3 0 3 1 1 0 1 2]
[0 0 1 0 0 0 1 0 3 0 1 3 1 3 0 3 3 2 1 0 2 3 3 2 2 2 2 0 3 3 1 3]
[0 0 0 1 0 0 1 2 1 1 0 2 1 3 3 1 3 2 2 0 1 1 2 3 3 1 0 3 2 1 0 0]
[0 0 0 0 1 0 0 1 2 1 1 0 2 1 3 3 1 3 2 2 0 1 1 2 3 3 1 0 3 2 1 0]
[0 0 0 0 0 1 1 1 2 0 1 2 2 0 1 0 3 0 3 1 3 3 0 1 3 2 1 2 2 1 3 1]
> MinimumLeeWeight($1);
28
```

`HadamardCodeZ4(δ, m)`

Given an integer $m \geq 1$ and an integer δ such that $1 \leq \delta \leq \lfloor (m+1)/2 \rfloor$, return a Hadamard code over $\mathbf{Z}_4$ of length 2^{m-1} and type $2^{\gamma}4^{\delta}$, where $\gamma = m + 1 - 2\delta$. Moreover, return a generator matrix with $\gamma + \delta$ rows constructed in a recursive way from the `Plotkin` and `BQPlotkin` constructions defined in Section 162.2.4.

A Hadamard code over $\mathbf{Z}_4$ of length 2^{m-1} is a code over $\mathbf{Z}_4$ 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.

ExtendedPerfectCodeZ4(δ, m)

Given an integer $m \geq 2$ and an integer δ such that $1 \leq \delta \leq \lfloor (m+1)/2 \rfloor$, return an extended perfect code over $\mathbf{Z}_4$ of length 2^{m-1}, such that its dual code is of type $2^\gamma 4^\delta$, where $\gamma = m + 1 - 2\delta$. Moreover, return a generator matrix constructed in a recursive way from the Plotkin and BQPlotkin constructions defined in Section 162.2.4.

An extended perfect code over $\mathbf{Z}_4$ of length 2^{m-1} is a code over $\mathbf{Z}_4$ 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 H162E3

Some codes over $\mathbf{Z}_4$ 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 Hadamard code C over $\mathbf{Z}_4$ of length 8 and type $2^1 4^2$ is defined. The matrix Gc is the quaternary matrix used to generate C and obtained by a recursive method from Plotkin and BQPlotkin constructions.

```
> C, Gc := HadamardCodeZ4(2,4);
> C;
((8, 4^2 2^1)) Linear Code over IntegerRing(4)
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]
> Gc;
[1 1 1 1 1 1 1 1]
[0 1 2 3 0 1 2 3]
[0 0 0 0 2 2 2 2]
> HasLinearGrayMapImage(C);
true [16, 5, 8] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 1 1 1 0 1 1 1 1 0 0 0]
[0 1 0 0 1 0 1 1 0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1 0 0 1 0 1 1 0 1]
[0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
Mapping from: CodeLinRng: C to [16, 5, 8] Linear Code over GF(2) given by a rule
```

Then, an extended perfect code D over $\mathbf{Z}_4$ of length 8 is defined, such that its dual code is of type $2^1 4^2$. The matrix Gd is the quaternary matrix which is used to generate D and obtained in a recursive way from Plotkin and BQPlotkin constructions. Note that the code D is the Kronecker dual code of C.

```
> D, Gd := ExtendedPerfectCodeZ4(2,4);
> D;
((8, 4^5 2^1)) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 1 0 0 1 3]
```

```
[0 1 0 1 0 0 2 2]
[0 0 1 1 0 0 1 1]
[0 0 0 2 0 0 0 2]
[0 0 0 0 1 0 3 2]
[0 0 0 0 0 1 2 3]
> Gd;
[1 1 1 1 1 1 1 1]
[0 1 2 3 0 1 2 3]
[0 0 1 1 0 0 1 1]
[0 0 0 2 0 0 0 2]
[0 0 0 0 1 1 1 1]
[0 0 0 0 0 1 2 3]
> DualKroneckerZ4(C) eq D;
true
```

`ReedMullerCodeZ4(r, m)`

`ReedMullerCodeQRMZ4(r, m)`

Given an integer $m \geq 2$ and an integer r such that $0 \leq r \leq m$, the r-th order Reed-Muller code over $\mathbf{Z}_4$ of length 2^m is returned.

The binary image under the modulo 2 map is the binary linear r-th order Reed-Muller code of length 2^m. For $r = 1$ and $r = m - 2$, the function returns the quaternary linear Kerdock and Preparata code, respectively.

`ReedMullerCodesLRMZ4(r, m)`

Given an integer $m \geq 1$ and an integer r such that $0 \leq r \leq m$, a set of r-th order Reed-Muller codes over $\mathbf{Z}_4$ of length 2^{m-1} is returned.

The binary image under the Gray map of any of these codes is a binary (not necessarily linear) code with the same parameters as the binary linear r-th order Reed-Muller code of length 2^m. Note that for these codes neither the usual inclusion nor duality properties of the binary linear Reed-Muller family are satisfied.

`ReedMullerCodeRMZ4(s, r, m)`

Given an integer $m \geq 1$, an integer r such that $0 \leq r \leq m$, and an integer s such that $0 \leq s \leq \lfloor (m-1)/2 \rfloor$, return a r-th order Reed-Muller code over $\mathbf{Z}_4$ of length 2^{m-1}, denoted by $RM_s(r, m)$, as well as the generator matrix used in the recursive construction.

The binary image under the Gray map is a binary (not necessarily linear) code with the same parameters as the binary linear r-th order Reed-Muller code of length 2^m. Note that the inclusion and duality properties are also satisfied, that is, the code $RM_s(r-1, m)$ is a subcode of $RM_s(r, m)$, $r > 0$, and the code $RM_s(r, m)$ is the Kronecker dual code of $RM_s(m-r-1, m)$, $r < m$.

Example H162E4

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. Note that $RM_1(1,4)$ and $RM_1(2,4)$ are the same as the ones given in Example H162E3 by `HadamardCodeZ4(2,4)` and `ExtendedPerfectCodeZ4(2,4)`, respectively.

```
> C1,G1 := ReedMullerCodeRMZ4(1,1,4);
> C2,G2 := ReedMullerCodeRMZ4(1,2,4);
> C1;
((8, 4^2 2^1)) Linear Code over IntegerRing(4)
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]
> C2;
((8, 4^5 2^1)) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 1 0 0 1 3]
[0 1 0 1 0 0 2 2]
[0 0 1 1 0 0 1 1]
[0 0 0 2 0 0 0 2]
[0 0 0 0 1 0 3 2]
[0 0 0 0 0 1 2 3]
> C1 subset C2;
true
> DualKroneckerZ4(C2) eq C1;
true
```

```
ReedMullerCodesRMZ4(s, m)
```

Let m be an integer $m \geq 1$, and s an integer such that $0 \leq s \leq \lfloor (m-1)/2 \rfloor$. This function returns a sequence containing the family of Reed-Muller codes over $\mathbf{Z}_4$ of length 2^{m-1}, that is, the codes $RM_s(r,m)$, for all $0 \leq r \leq m$.

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

$$RM_s(0,m) \subset RM_s(1,m) \subset \ldots \subset RM_s(m,m)$$

Example H162E5

The family of Reed-Muller codes over $\mathbf{Z}_4$ of length 2^2 given by $s=0$ is constructed.

```
> F := ReedMullerCodesRMZ4(0,3);
> F;
[((4, 4^0 2^1)) Cyclic Linear Code over IntegerRing(4)
Generator matrix:
[2 2 2 2],
((4, 4^1 2^2)) Cyclic Linear Code over IntegerRing(4)
Generator matrix:
[1 1 1 1]
[0 2 0 2]
[0 0 2 2],
((4, 4^3 2^1)) Cyclic Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 1]
[0 1 0 1]
[0 0 1 1]
[0 0 0 2],
((4, 4^4 2^0)) Cyclic Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]]
> F[1] subset F[2] and F[2] subset F[3] and F[3] subset F[4];
true
```

162.2.3 Derived Binary Codes

As well as the binary image of a quaternary code under the Gray map (see section 162.2.1), there are also two other associated canonical binary codes. They are known the *residue* and *torsion* codes, the former being a subcode of the latter.

From any binary code-subcode pair $C_1 \subset C_2$, a quaternary code C can be constructed such that the residue and torsion codes of C will be C_1 and C_2 respectively. Note that this quaternary code is not unique.

```
BinaryResidueCode(C)
```

Given a quaternary code C, return the binary code formed by taking each codeword in C modulo 2. This is known as the *binary residue code* of C.

```
BinaryTorsionCode(C)
```

Given a quaternary code C, return the binary code formed by the support of each codeword in C which is zero modulo 2. This is known as the *binary torsion code* of C.

```
Z4CodeFromBinaryChain(C1, C2)
```

Given binary code C_1 and C_2 such that $C_1 \subset C_2$, return a quaternary code such that its binary residue code is C_1 and its binary torsion code is C_2.

Example H162E6

This example shows that the derived binary codes of the $\mathbf{Z}_4$ Golay code, are in fact equal to the binary Golay code.

```
> C := GolayCodeZ4(false);
> C;
(23, 4^12 2^0)) Cyclic Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 3 1 0 0 2 3 3 3 0 3 2]
[0 1 0 0 0 0 0 0 0 0 0 0 2 1 1 0 0 0 1 1 3 2 3]
[0 0 1 0 0 0 0 0 0 0 0 0 3 3 1 1 2 3 3 0 1 2 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 3 3 1 1 2 3 3 0 1 2]
[0 0 0 0 1 0 0 0 0 0 0 0 2 2 3 3 1 3 0 1 3 2 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 1 2 3 1 2 0 1 1 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 2 3 1 2 0 1 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 2 3 1 2 0 1 1]
[0 0 0 0 0 0 0 0 1 0 0 0 1 3 0 1 3 3 0 2 2 1 3]
[0 0 0 0 0 0 0 0 0 1 0 0 3 2 3 0 3 2 2 3 2 1 3]
[0 0 0 0 0 0 0 0 0 0 1 0 3 0 2 3 2 2 1 1 3 1 3]
[0 0 0 0 0 0 0 0 0 0 0 1 3 0 0 2 1 1 1 0 1 2 3]
>
> CRes := BinaryResidueCode(C);
> CTor := BinaryTorsionCode(C);
> CRes eq CTor;
true
> CRes:Minimal;
[23, 12, 7] Linear Code over GF(2)
> AreEq, _ := IsEquivalent( CRes, GolayCode(GF(2), false) );
> AreEq;
true
```

Note that the canonical code over $\mathbf{Z}_4$ corresponding to the derived binary codes $CRes$ and $CTor$ is different to the initial $\mathbf{Z}_4$ code C.

```
> C1 := Z4CodeFromBinaryChain(CRes, CTor);
> C1:Minimal;
(23, 16777216) Linear Code over IntegerRing(4)
> C eq C1;
false
```

162.2.4 New Codes from Old

The functions described in this section produce a new code over $\mathbf{Z}_4$ by modifying in some way the codewords of some given codes over $\mathbf{Z}_4$.

```
PlotkinSum(A, B)
```

Given matrices A and B both over the same ring and with the same number of columns, return the P_{AB} matrix over the same ring of A and B, where

$$P_{AB} = \begin{pmatrix} A & A \\ 0 & B \end{pmatrix}.$$

```
PlotkinSum(C, D)
```

Given codes C and D both over the same ring and of the same length, construct the Plotkin sum of C and D. The Plotkin sum consists of all vectors of the form $(u|u+v)$, where $u \in C$ and $v \in D$.

Note that the Plotkin sum is computed using generator matrices for C and D and the `PlotkinSum` function for matrices. Thus, this function returns the code over $\mathbf{Z}_4$ generated by the matrix P_{AB} defined above, where A and B are the generator matrices for C and D, respectively.

```
QuaternaryPlotkinSum(A, B)
```

Given two matrices A and B over $\mathbf{Z}_4$, both with the same number of columns, return the QP_{AB} matrix over $\mathbf{Z}_4$, where

$$QP_{AB} = \begin{pmatrix} A & A & A & A \\ 0 & B & 2B & 3B \end{pmatrix}.$$

```
QuaternaryPlotkinSum(C, D)
```

Given two codes C and D over $\mathbf{Z}_4$, both of the same length, construct the Quaternary Plotkin sum of C and D. The Quaternary Plotkin sum is a code over $\mathbf{Z}_4$ that consists of all vectors of the form $(u, u+v, u+2v, u+3v)$, where $u \in C$ and $v \in D$.

Note that the Quaternary Plotkin sum is computed using generator matrices of C and D and the `QuaternaryPlotkinSum` function for matrices, that is, this function returns the code over $\mathbf{Z}_4$ generated by the matrix QP_{AB} defined above, where A and B are generators matrices of C and D, respectively.

```
BQPlotkinSum(A, B, C)
```

Given three matrices A, B, and C over $\mathbf{Z}_4$, all with the same number of columns, return the BQP_{ABC} matrix over $\mathbf{Z}_4$, where

$$BQP_{ABC} = \begin{pmatrix} A & A & A & A \\ 0 & B' & 2B' & 3B' \\ 0 & 0 & \hat{B} & \hat{B} \\ 0 & 0 & 0 & C \end{pmatrix},$$

B' is obtained from B replacing the twos with ones in the rows of order two, and $\hat{B}$ is obtained from B removing the rows of order two.

`BQPlotkinSum(D, E, F)`

Given three codes D, E and F over $\mathbf{Z}_4$, all of the same length, construct the BQ Plotkin sum of D, E and F. Let Ge be a generator matrix for E of type $2^\gamma 4^\delta$. The code E' over $\mathbf{Z}_4$ is obtained from E by replacing the twos with ones in the γ rows of order two of Ge, and the code $\hat{E}$ over $\mathbf{Z}_4$ is obtained from E removing the γ rows of order two of Ge.

The BQ Plotkin sum is a code over $\mathbf{Z}_4$ that consists of all vectors of the form $(u, u+v', u+2v'+\hat{v}, u+3v'+\hat{v}+z)$, where $u \in Gd$, $v' \in Ge'$ $\hat{v} \in \hat{Ge}$, and $z \in Gf$, where Gd, Ge', $\hat{Ge}$ and Gf are generator matrices for D, E', $\hat{E}$ and F, respectively.

Note that the BQPlotkin sum is computed using generator matrices of D, E and F and the `BQPlotkinSum` function for matrices. However, this function does not necessarily return the same code over $\mathbf{Z}_4$ as that generated by the matrix QP_{ABC} defined above, where A, B and C are generators matrices of D, E and F, respectively, as shown in Example H162E7.

`DoublePlotkinSum(A, B, C, D)`

Given four matrices A, B, C, and D over $\mathbf{Z}_4$, all with the same number of columns, return the DP_{ABC} matrix over $\mathbf{Z}_4$, where

$$DP_{ABCD} = \begin{pmatrix} A & A & A & A \\ 0 & B & 2B & 3B \\ 0 & 0 & C & C \\ 0 & 0 & 0 & D \end{pmatrix}.$$

`DoublePlotkinSum(E, F, G, H)`

Given four codes E, F, G and H over $\mathbf{Z}_4$, all of the same length, construct the Double Plotkin sum of E, F, G and H. The Double Plotkin sum is a code over $\mathbf{Z}_4$ that consists of all vectors of the form $(u, u+v, u+2v+z, u+3v+z+t)$, where $u \in E$, $v \in F$, $z \in G$ and $t \in H$.

Note that the Double Plotkin sum is computed using generator matrices of E, F, G and H and the `DoublePlotkinSum` function for matrices, that is, this function returns the code over $\mathbf{Z}_4$ generated by the matrix DP_{ABCD} defined above, where A, B, C and D are generator matrices for E, F, G and H, respectively.

`DualKroneckerZ4(C)`

Given a code C over $\mathbf{Z}_4$ of length 2^m, return its Kronecker dual code. The Kronecker dual code of C is $C_\otimes^\perp = \{x \in \mathbf{Z}_4^{2^m} : x \cdot K_{2^m} \cdot y^t = 0, \forall y \in C\}$, where $K_{2^m} = \otimes_{j=1}^m K_2$, $K_2 = \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix}$ and $\otimes$ denotes the Kronecker product of matrices. Equivalently, K_{2^m} is a quaternary matrix of length 2^m with the vector $(1,3,3,1,3,1,1,3,\ldots)$ in the main diagonal and zeros elsewhere.

Example H162E7

The purpose of this example is to show that the codes over $\mathbf{Z}_4$ constructed from the BQPlotkinSum function for matrices are not necessarily the same as the ones constructed from the BQPlotkinSum function for codes.

```
> Z4:=IntegerRing(4);
> Ga:=Matrix(Z4,1,2,[1,1]);
> Gb:=Matrix(Z4,2,2,[1,2,0,2]);
> Gc:=Matrix(Z4,1,2,[2,2]);
> Ca:=LinearCode(Ga);
> Cb:=LinearCode(Gb);
> Cc:=LinearCode(Gc);
> C:=LinearCode(BQPlotkinSum(Ga,Gb,Gc));
> D:=BQPlotkinSum(Ca,Cb,Cc);
> C eq D;
false
```

Example H162E8

```
> Ga := GeneratorMatrix(ReedMullerCodeRMZ4(1,2,3));
> Gb := GeneratorMatrix(ReedMullerCodeRMZ4(1,1,3));
> Gc := GeneratorMatrix(ReedMullerCodeRMZ4(1,0,3));
> C := ReedMullerCodeRMZ4(1,2,4);
> Cp := LinearCode(PlotkinSum(Ga, Gb));
> C eq Cp;
true
> D := ReedMullerCodeRMZ4(2,2,5);
> Dp := LinearCode(BQPlotkinSum(Ga, Gb, Gc));
> D eq Dp;
true
```

162.3 Invariants

162.3.1 The Standard Form

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

$$\begin{pmatrix} I_{k_1} & A & B \\ 0 & 2I_{k_2} & 2C \end{pmatrix}$$

where I_{k_1} and I_{k_2} are the $k_1 \times k_1$ and $k_2 \times k_2$ identity matrices, respectively, A and C are $\mathbf{Z}_2$-matrices, and B is a $\mathbf{Z}_4$-matrix. Any $\mathbf{Z}_4$-linear code C is permutation-equivalent to a code S which is in standard form. Furthermore, the integers k_1 and k_2, defined above, are unique [Wan97, Prop. 1.1].

`StandardForm(C)`

This function, given any $\mathbf{Z}_4$-linear code C, returns a permutation-equivalent code S in standard form, together with the corresponding isomorphism from C onto S.

Example H162E9

The standard form is computed for a small $\mathbf{Z}_4$ code. Note that the number of rows in the generator matrix of the standard code may be less than that in the original code.

```
> Z4 := IntegerRing(4);
> C := LinearCode<Z4, 4 | [2,2,1,1], [0,2,0,2]>;
> C;
[4, 3, 2] Linear Code over IntegerRing(4)
Generator matrix:
[2 0 1 3]
[0 2 0 2]
[0 0 2 2]
> S, f := StandardForm(C);
> S;
[4, 2, 2] Linear Code over IntegerRing(4)
Generator matrix:
[1 1 2 2]
[0 2 2 0]
> #S;
8
> #C;
8
> f(C.1);
(1 3 0 2)
> f(C.2);
(0 2 2 0)
> f(C.3);
(2 2 0 0)
> S.1@@f;
(2 2 1 1)
> S.2@@f;
(0 2 0 2)
```

162.3.2 Structures Associated with the Gray Map

```
MinRowsGeneratorMatrix(C)
```

A generator matrix for the code C over $\mathbf{Z}_4$ of length n and type $2^\gamma 4^\delta$, 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.

```
SpanZ2CodeZ4(C)
```

Given a code C over $\mathbf{Z}_4$ of length n, return $S_C = \Phi^{-1}(S_{bin})$ as a code over $\mathbf{Z}_4$, and the linear span of C_{bin}, $S_{bin} = \langle C_{bin} \rangle$, as a binary linear code of length $2n$, where $C_{bin} = \Phi(C)$ and Φ is the Gray map.

```
KernelZ2CodeZ4(C)
```

Given a code C over $\mathbf{Z}_4$ of length n, return its kernel K_C as a subcode over $\mathbf{Z}_4$ of C, and $K_{bin} = \Phi(K_C)$ as a binary linear subcode of C_{bin} of length $2n$, 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 code C over $\mathbf{Z}_4$ of length n, 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 subcode over $\mathbf{Z}_4$. 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 code C over $\mathbf{Z}_4$, 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 code C over $\mathbf{Z}_4$, return the dimension of the Gray map image of its kernel K_C over $\mathbf{Z}_4$, 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 H162E10

```
> C := ReedMullerCodeRMZ4(0,3,5);
> DimensionOfKernelZ2(C);
20
> DimensionOfSpanZ2(C);
27
> K, Kb := KernelZ2CodeZ4(C);
> S, Sb := SpanZ2CodeZ4(C);
> K subset C;
true
> C subset S;
true
> Dimension(Kb) eq DimensionOfKernelZ2(C);
true
> Dimension(Sb) eq DimensionOfSpanZ2(C);
true
```

162.3.3 Coset Representatives

`CosetRepresentatives(C)`

Given a code C over $\mathbf{Z}_4$ of length n, with ambient space $V = \mathbf{Z}_4^n$, 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 set of coset representatives $\{c_0, c_1, \ldots, c_t\}$ satisfies the two 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 code C over $\mathbf{Z}_4$ of length n, and a subcode S over $\mathbf{Z}_4$ 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 set of coset representatives $\{c_0, c_1, \ldots, c_t\}$ satisfies the two 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 H162E11

```
> C := LinearCode<Integers(4), 4 | [[1,0,0,3],[0,1,1,3]]>;
> L := CosetRepresentatives(C);
> Set(RSpace(Integers(4),4)) eq {v+ci : v in Set(C), ci in L};
true
> K := KernelZ2CodeZ4(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
```

162.3.4 Information Space and Information Sets

InformationSpace(C)

Given a code C over $\mathbf{Z}_4$ of length n and type $2^\gamma 4^\delta$, return the $\mathbf{Z}_4$-submodule of $\mathbf{Z}_4^{\gamma+\delta}$ isomorphic to $\mathbf{Z}_2^\gamma \times \mathbf{Z}_4^\delta$ 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 H162E12

```
> C := LinearCode<Integers(4), 4 | [[2,0,0,2],[0,1,1,3]]>;
> R, V, f, fbin := InformationSpace(C);
> G := MinRowsGeneratorMatrix(C);
> (#R eq #C) and (#V eq #C);
true
> Set([f(i) : i in R]) eq Set(C);
true
> Set([i*G : i in R]) eq Set(C);
false
> i := R![2,3];
> c := f(i);
> c;
(2 3 3 3)
> c in C;
true
> i*G eq c;
false
> ibin := V![1,1,0];
> cbin := fbin(ibin);
> cbin;
(1 1 1 0 1 0 1 0)
> cbin in GrayMapImage(C);
true
> cbin eq GrayMap(C)(c);
true
```

`InformationSet(C)`

Given a code C over $\mathbf{Z}_4$ of length n and type $2^\gamma 4^\delta$, return an information set $I = [i_1, \ldots, i_{\gamma+\delta}] \subseteq \{1, \ldots, n\}$ for C such that the code C punctured on $\{1, \ldots, n\} \backslash \{i_{\gamma+1}, \ldots, i_{\gamma+\delta}\}$ is of type 4^δ, and the corresponding information set $\Phi(I) = [2i_1 - 1, \ldots, 2i_\gamma - 1, 2i_{\gamma+1} - 1, 2i_{\gamma+1}, \ldots, 2i_{\gamma+\delta} - 1, 2i_{\gamma+\delta}] \subseteq \{1, \ldots, 2n\}$ 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 code C over $\mathbf{Z}_4$ of length n and type $2^\gamma 4^\delta$ and a sequence $I \subseteq \{1, \ldots, n\}$ or $I \subseteq \{1, \ldots, 2n\}$, return `true` if and only if $I \subseteq \{1, \ldots, n\}$ is an information set for C. This function also returns another boolean, which is `true` if and only if $I \subseteq \{1, \ldots, 2n\}$ 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 H162E13

```
> C := HadamardCodeZ4(3,6);
> C;
((32, 4^3 2^1)) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 3 2 0 3 2 1 3 2 1 0 2 1 0 3 1 0 3 2 0 3 2 1 3 2 1 0 2 1 0 3]
[0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3]
[0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
> I, Ibin := InformationSet(C);
> I;
[ 16, 28, 31, 32 ]
> Ibin;
[ 31, 55, 56, 61, 62, 63, 64 ]
> #PunctureCode(C, {1..32} diff Set(I)) eq #C;
true
> Cbin := GrayMapImage(C);
> V := VectorSpace(GF(2), 7);
> #{V![c[i] : i in Ibin] : c in Cbin} eq #Cbin;
true
```

```
> IsInformationSet(C, I);
true false
> IsInformationSet(C, Ibin);
false true
> IsInformationSet(C, [1, 2, 5, 17]);
true false
> IsInformationSet(C, [1, 2, 3, 4, 9, 10, 33]);
false true
> D := LinearCode<Integers(4), 5 | [[2,0,0,2,0],[0,2,0,2,2],[0,0,2,2,0]]>;
> IsInformationSet(D, [1,3,5]);
true true
```

162.3.5 Syndrome Space and Coset Leaders

```
SyndromeSpace(C)
```

Given a code C over $\mathbf{Z}_4$ of length n and type $2^\gamma 4^\delta$, return the $\mathbf{Z}_4$-submodule of $\mathbf{Z}_4^{n-\delta}$ isomorphic to $\mathbf{Z}_2^\gamma \times \mathbf{Z}_4^{n-\gamma-\delta}$ such that the first γ coordinates are of order two, that is, the space of syndrome vectors for C. The function also returns the $(2n-2\delta-\gamma)$-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 dual code of C, produced by function `Dual(C)`.

```
Syndrome(u, C)
```

Given a code C over $\mathbf{Z}_4$ of length n and type $2^\gamma 4^\delta$, and a vector u from the ambient space $V = \mathbf{Z}_4^n$ or $V_2 = \mathbf{Z}_2^{2n}$, construct the syndrome of u relative to the code C. This will be an element of the syndrome space of C, considered as the $\mathbf{Z}_4$-submodule of $\mathbf{Z}_4^{n-\delta}$ isomorphic to $\mathbf{Z}_2^\gamma \times \mathbf{Z}_4^{n-\gamma-\delta}$ such that the first γ coordinates are of order two.

```
CosetLeaders(C)
```

Given a code C over $\mathbf{Z}_4$ of length n, with ambient space $V = \mathbf{Z}_4^n$, 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. 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 H162E14

```
> C := LinearCode<Integers(4), 4 | [[2,0,0,2],[0,1,1,3]]>;
> R, V, f, fbin := InformationSpace(C);
> Rs, Vs := SyndromeSpace(C);
> #R * #Rs eq 4^Length(C);
true
> #V * #Vs eq 4^Length(C);
true
> i := R![2,3];
> c := f(i);
> c;
(2 3 3 3)
> u := c;
> u[2] := u[2] + 3;
> u;
(2 2 3 3)
> s := Syndrome(u, C);
> s in Rs;
true
> H := Transpose(MinRowsGeneratorMatrix(Dual(C)));
> s eq u*H;
true
> L, mapCosetLeaders := CosetLeaders(C);
> ev := mapCosetLeaders(s);
> ev;
(0 3 0 0)
> ev in L;
true
> u - ev eq c;
true
```

162.3.6 Miscellaneous Functions

```
Correlation(v)
```

Let v be a codeword over $\mathbf{Z}_4$. Define $w_j = \#\{k : v[k] = j\}$ for $j = 0, \ldots, 3$. Then the *correlation* of v is the Gaussian integer $(w_0 - w_2) + i * (w_1 - w_3)$.

162.4 Weight Distributions

In the case of a linear code, weight and distance distributions are equivalent (in particular minimum weight and minimum distance are equivalent).

162.4.1 Hamming Weight

For an element $x \in R$ for any finite ring R, the *Hamming weight* $w_H(x)$ is defined by:

$$w_H(x) = 0 \iff x = 0, \qquad w_H(x) = 1 \iff x \neq 0$$

The *Hamming weight* $w_H(v)$ of a vector $v \in R^n$ is defined to be the sum (in $\mathbf{Z}$) of the Hamming weights of its components.

The *Hamming weight* is often referred to as simply the *weight.*

```
MinimumWeight(C)
```

```
MinimumDistance(C)
```

Determine the minimum (Hamming) weight of the words belonging to the code C, which is also the minimum distance between any two codewords.

```
WeightDistribution(C)
```

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

```
DualWeightDistribution(C)
```

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

Example H162E15

In this example, the weight distribution of a quadratic residue code over $\mathbf{Z}_4$ and its dual are computed.

```
> C := QRCodeZ4(17);
> C;
((17, 4^9 2^0)) Cyclic Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 0 0 0 0 1 1 3 0 3 0 3 1]
[0 1 0 0 0 0 0 0 0 3 0 2 3 1 3 1 2]
[0 0 1 0 0 0 0 0 0 2 1 2 2 1 1 1 3]
[0 0 0 1 0 0 0 0 0 1 3 0 2 1 1 0 2]
[0 0 0 0 1 0 0 0 0 2 3 1 0 0 1 3 2]
[0 0 0 0 0 1 0 0 0 2 0 1 1 2 0 3 1]
[0 0 0 0 0 0 1 0 0 3 1 1 1 2 2 1 2]
[0 0 0 0 0 0 0 1 0 2 1 3 1 3 2 0 3]
```

```
[0 0 0 0 0 0 0 0 1 1 3 0 3 0 3 1 1]
> WeightDistribution(C);
[ <0, 1>, <5, 34>, <6, 68>, <7, 748>, <8, 2567>, <9, 6817>, <10, 17612>,
  <11, 34340>, <12, 50014>, <13, 56168>, <14, 50728>, <15, 30872>,
  <16, 9826>, <17, 2349> ]
> DualWeightDistribution(C);
[ <0, 1>, <6, 68>, <8, 935>, <9, 1632>, <10, 4148>, <11, 8568>, <12, 12886>,
  <13, 14280>, <14, 11968>, <15, 7752>, <16, 2890>, <17, 408> ]
```

162.4.2 Lee Weight

For an element $x \in \mathbf{Z}_4$, the *Lee weight* $w_L(x)$ is defined by:

$$w_L(0) = 0, \quad w_L(1) = w_L(3) = 1, \quad w_L(2) = 2.$$

The *Lee weight* $w_L(v)$ of a vector $v \in \mathbf{Z}_4^n$ is defined to be the sum (in $\mathbf{Z}$) of the Lee weights of its components. See [Wan97, p. 16].

`LeeWeight(a)`

The Lee weight of the element $a \in \mathbf{Z}_4$.

`LeeWeight(v)`

The Lee weight of the codeword v.

`LeeDistance(u, v)`

The Lee distance between the codewords u and v, where u and v belong to the same code C. This is defined to be the Lee weight of $(u - v)$.

`MinimumLeeWeight(C)`

`MinimumLeeDistance(C)`

The minimum Lee weight of the code C.

`LeeWeightDistribution(C)`

The Lee weight distribution of the code C.

`DualLeeWeightDistribution(C)`

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

`WordsOfLeeWeight(C, w)`

`Cutoff` RngIntElt *Default :* ∞

Given a linear code C, return the set of all words of C having Lee weight w. If `Cutoff` is set to a non-negative integer c, then the algorithm will terminate after a total of c words have been found.

```
WordsOfBoundedLeeWeight(C, l, u)
```

`Cutoff` RngIntElt *Default :* ∞

Given a linear code C, return the set of all words of C having Lee weight between l and u, inclusive. If `Cutoff` is set to a non-negative integer c, then the algorithm will terminate after a total of c words have been found.

Example H162E16

We calculate the Lee weight distribution of a Reed Muller code over $\mathbf{Z}_4$ and enumerate all words of Lee weight 8.

```
> C := ReedMullerCodeZ4(1, 3);
> C;
(8, 256, 4) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 1 2 1]
[0 1 0 0 2 1 1 3]
[0 0 1 0 1 1 3 2]
[0 0 0 1 3 2 3 3]
> LeeWeightDistribution(C);
[ <0, 1>, <6, 112>, <8, 30>, <10, 112>, <16, 1> ]
> W := WordsOfLeeWeight(C, 8);
> #W;
30
```

162.4.3 Euclidean Weight

For an element $x \in \mathbf{Z}_4$, the *Euclidean weight* $w_E(x)$ is defined by:

$$w_E(0) = 0, \quad w_E(1) = w_E(3) = 1, \quad w_E(2) = 4.$$

The *Euclidean weight* $w_E(v)$ of a vector $v \in \mathbf{Z}_4^n$ is defined to be the sum (in $\mathbf{Z}$) of the Euclidean weights of its components. See [Wan97, p. 16].

```
EuclideanWeight(a)
```

The Euclidean weight of the element $a \in \mathbf{Z}4$.

```
EuclideanWeight(v)
```

The Euclidean weight of the $\mathbf{Z}_4$-codeword v.

```
EuclideanDistance(u, v)
```

The Euclidean distance between the $\mathbf{Z}_4$-codewords u and v, where u and v belong to the same code C. This is defined to be the Euclidean weight of $(u - v)$.

```
MinimumEuclideanWeight(C)
MinimumEuclideanDistance(C)
```

The minimum Euclidean weight of the $\mathbf{Z}_4$-code C.

```
EuclideanWeightDistribution(C)
```

The Euclidean weight distribution of the $\mathbf{Z}_4$-code C.

```
DualEuclideanWeightDistribution(C)
```

The Euclidean weight distribution of the dual of the $\mathbf{Z}_4$-code C.

Example H162E17

The Euclidean weight distribution is calculated for a quadratic residue code over $\mathbf{Z}_4$

```
> C := QRCodeZ4(17);
> C;
(17, 262144) Cyclic Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 0 0 0 0 1 1 3 0 3 0 3 1]
[0 1 0 0 0 0 0 0 0 3 0 2 3 1 3 1 2]
[0 0 1 0 0 0 0 0 0 2 1 2 2 1 1 1 3]
[0 0 0 1 0 0 0 0 0 1 3 0 2 1 1 0 2]
[0 0 0 0 1 0 0 0 0 2 3 1 0 0 1 3 2]
[0 0 0 0 0 1 0 0 0 2 0 1 1 2 0 3 1]
[0 0 0 0 0 0 1 0 0 3 1 1 1 2 2 1 2]
[0 0 0 0 0 0 0 1 0 2 1 3 1 3 2 0 3]
[0 0 0 0 0 0 0 0 1 1 3 0 3 0 3 1 1]
> EuclideanWeightDistribution(C);
[ <0, 1>, <7, 136>, <8, 170>, <9, 170>, <10, 408>, <11, 544>, <12, 986>,
<13, 1768>, <14, 3128>, <15, 5032>, <16, 6120>, <17, 6360>, <18, 8432>,
<19, 12512>, <20, 12682>, <21, 11152>, <22, 14416>, <23, 17680>, <24, 16048>,
<25, 15164>, <26, 17952>, <27, 16864>, <28, 13328>, <29, 14144>, <30, 14144>,
<31, 10064>, <32, 7837>, <33, 8024>, <34, 6800>, <35, 4896>, <36, 3485>,
<37, 2992>, <38, 2992>, <39, 1768>, <40, 510>, <41, 1258>, <42, 1224>,
<44, 238>, <45, 408>, <46, 136>, <47, 136>, <48, 34>, <68, 1> ]
```

162.5 Weight Enumerators

```
CompleteWeightEnumerator(C)
```

Let C be a code over a finite ring R of cardinality q, and suppose that the elements of R are ordered in some way. Then for a codeword $v \in C$ and the i-th element $a \in R$, let $s_i(v)$ denote the number of components of v equal to a.

This function returns the complete weight enumerator $\mathcal{W}_C(X_0, X_1, \ldots, X_{q-1})$ of C, which is defined by:

$$\mathcal{W}_C(X_0, X_1, \ldots, X_{q-1}) = \sum_{v \in C} {X_0}^{s_0(v)} {X_1}^{s_1(v)} \cdots {X_{q-1}}^{s_{q-1}(v)}.$$

See [Wan97, p. 9] for more information. The result will lie in a global multivariate polynomial ring over $\mathbf{Z}$ with q variables. The angle-bracket notation may be used to assign names to the indeterminates.

```
SymmetricWeightEnumerator(C)
```

Suppose C is a $\mathbf{Z}_4$-code. This function returns the symmetric weight enumerator $\mathrm{swe}_C(X_0, X_1, X_2)$ of C, which is defined by:

$$\mathrm{swe}_C(X_0, X_1, X_2) = \mathcal{W}_C(X_0, X_1, X_2, X_1),$$

where $\mathcal{W}_C$ is the complete weight enumerator, defined above. See [Wan97, p. 14] for more information. The result will lie in a global multivariate polynomial ring over $\mathbf{Z}$ with three variables. The angle-bracket notation may be used to assign names to the indeterminates.

```
WeightEnumerator(C)
```

```
HammingWeightEnumerator(C)
```

Suppose C is a code over some finite ring R. This function returns the Hamming weight enumerator $\mathrm{Ham}_C(X, Y)$ of C, which is defined by:

$$\mathrm{Ham}_C(X, Y) = \sum_{v \in C} X^{n - w_H(v)} Y^{w_H(v)},$$

where $w_H(v)$ is the Hamming weight function. The result will lie in a global multivariate polynomial ring over $\mathbf{Z}$ with two variables. The angle-bracket notation may be used to assign names to the indeterminates.

```
LeeWeightEnumerator(C)
```

Suppose C is a $\mathbf{Z}_4$-code. This function returns the Lee weight enumerator $\mathrm{Lee}_C(X, Y)$ of C, which is defined by:

$$\mathrm{Lee}_C(X, Y) = \sum_{v \in C} X^{2*n - w_L(v)} Y^{w_L(v)},$$

where $w_L(v)$ is the Lee weight function, defined in Section 162.4.2. The result will lie in a global multivariate polynomial ring over $\mathbf{Z}$ with two variables. The angle-bracket notation may be used to assign names to the indeterminates.

`EuclideanWeightEnumerator(C)`

Suppose C is a $\mathbf{Z}_4$-code. This function returns the Euclidean weight enumerator $\mathrm{Euclidean}_C(X, Y)$ of C, which is defined by:

$$\mathrm{Euclidean}_C(X, Y) = \sum_{v \in C} X^{4*n - w_E(v)} Y^{w_E(v)},$$

where $w_E(v)$ is the Euclidean weight function, defined in Section 162.4.3. The result will lie in a global multivariate polynomial ring over $\mathbf{Z}$ with two variables. The angle-bracket notation may be used to assign names to the indeterminates.

Example H162E18

Several different weight enumerators are calculated for the octacode. To ensure the polynomials print out nicely, names are assigned to the polynomial ring indeterminates in each case. These names will persist if further calls to these functions (over $\mathbf{Z}_4$) are made.

```
> Z4 := IntegerRing(4);
> O8 := LinearCode<Z4, 8 |
>     [1,0,0,0,3,1,2,1],
>     [0,1,0,0,1,2,3,1],
>     [0,0,1,0,3,3,3,2],
>     [0,0,0,1,2,3,1,1]>;
> #O8;
256
> CWE<X0,X1,X2,X3> := CompleteWeightEnumerator(O8);
> CWE;
X0^8 + 14*X0^4*X2^4 + 56*X0^3*X1^3*X2*X3 + 56*X0^3*X1*X2*X3^3 +
    56*X0*X1^3*X2^3*X3 + 56*X0*X1*X2^3*X3^3 + X1^8 + 14*X1^4*X3^4 +
    X2^8 + X3^8
> SWE<X0,X1,X2> := SymmetricWeightEnumerator(O8);
> SWE;
X0^8 + 14*X0^4*X2^4 + 112*X0^3*X1^4*X2 + 112*X0*X1^4*X2^3 + 16*X1^8 +
    X2^8
> HWE<X,Y> := HammingWeightEnumerator(O8);
> HWE;
X^8 + 14*X^4*Y^4 + 112*X^3*Y^5 + 112*X*Y^7 + 17*Y^8
> LeeWeightEnumerator(O8);
X^16 + 112*X^10*Y^6 + 30*X^8*Y^8 + 112*X^6*Y^10 + Y^16
> EuclideanWeightEnumerator(O8);
X^32 + 128*X^24*Y^8 + 126*X^16*Y^16 + Y^32
```

162.6 Decoding

This section describes functions for decoding vectors from the ambient space of a code over $\mathbf{Z}_4$, or the corresponding space over $\mathbf{Z}_2$ under the Gray map, using four different algorithms: coset decoding, syndrome decoding, lifted decoding and permutation decoding. The reader is referred to [FCPV08, FCPV10, VZP15] for more information on coset decoding; to [HKC$^+$94, MS78, Wan97] on syndrome decoding; to [BZ01, GV98] on lifted decoding; and to [BV16a, BV16b, BBFCV15] on permutation decoding.

162.6.1 Coset Decoding

`CosetDecode(C, u : ` *parameters*`)`

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

Given a code C over $\mathbf{Z}_4$ of length n, and a vector u from the ambient space $V = \mathbf{Z}_4^n$ or $V_2 = \mathbf{Z}_2^{2n}$, attempt to decode u with respect to C. 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 subcode over $\mathbf{Z}_4$, $[c_0, c_1, \ldots, c_t]$ 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 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 code C over $\mathbf{Z}_4$ of length n, and a sequence Q of vectors from the ambient space $V = \mathbf{Z}_4^n$ or $V_2 = \mathbf{Z}_2^{2n}$, attempt to decode the vectors of Q with respect to C. This function is similar to the function `CosetDecode(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 `CosetDecode(C, u)`.

Example H162E19

Starting with the Hadamard code C over $\mathbf{Z}_4$ of length 16 and type $2^0 4^3$, 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 := HadamardCodeZ4(3, 5);
> C;
((16, 4^3 2^0)) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 3 2 0 3 2 1 3 2 1 0 2 1 0 3]
[0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3]
[0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3]
> d := MinimumLeeDistance(C);
> t := Floor((d-1)/2);
> t;
7
> c := C ! [1,1,1,1,2,2,2,2,3,3,3,3,0,0,0,0];
> 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;
> c;
(1 1 1 1 2 2 2 2 3 3 3 3 0 0 0 0)
> u;
(1 1 1 1 0 2 2 2 3 3 3 0 3 0 0 2)
> grayMap := GrayMap(UniverseCode(Integers(4), Length(C)));
> grayMap(c-u);
(0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1)
> isDecoded, uDecoded := CosetDecode(C, u : MinWeightCode := d);
> isDecoded;
true
> uDecoded eq c;
true
```

162.6.2 Syndrome Decoding

`SyndromeDecode(C, u)`

Given a code C over $\mathbf{Z}_4$ of length n, and a vector u from the ambient space $V = \mathbf{Z}_4^n$ or $V_2 = \mathbf{Z}_2^{2n}$, attempt to decode u with respect to C. 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 code C over $\mathbf{Z}_4$ of length n, and a sequence Q of vectors from the ambient space $V = \mathbf{Z}_4^n$ or $V_2 = \mathbf{Z}_2^{2n}$, attempt to decode the vectors of Q with respect to C. This function is similar to the function `SyndromeDecode(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 `SyndromeDecode(C, u)`.

Example H162E20

The Hadamard code C over $\mathbf{Z}_4$ of length 8 and type $2^1 4^2$ 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 CosetLeaders function, to the syndrome to recover the original vector.

```
> C := HadamardCodeZ4(2, 4);
> C;
((8, 4^2 2^1, 8)) Linear Code over IntegerRing(4)
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]
> t := Floor((MinimumLeeDistance(C)-1)/2);
> t;
3
> R, V, f, fbin := InformationSpace(C);
> i := R![2,1,0];
> c := f(i);
> c;
(1 0 3 2 3 2 1 0)
> u := c;
> u[5] := u[5] + 3;
> u[7] := u[7] + 2;
> c;
(1 0 3 2 3 2 1 0)
> u;
(1 0 3 2 2 2 3 0)
> grayMap := GrayMap(UniverseCode(Integers(4), Length(C)));
> grayMap(c-u);
(0 0 0 0 0 0 0 0 1 0 0 1 1 0 0)
> isDecoded, uDecoded := SyndromeDecode(C, u);
> isDecoded;
true
> uDecoded eq c;
true
> L, mapCosetLeaders := CosetLeaders(C);
> ev := mapCosetLeaders(Syndrome(u, C));
> ev;
(0 0 0 0 3 0 2 0)
> u - ev eq c;
true
```

162.6.3 Lifted Decoding

`LiftedDecode(C, u : parameters)`

`AlgMethod` MonStgElt *Default : "Euclidean"*

Given a code C over $\mathbf{Z}_4$ of length n, and a vector u from the ambient space $V = \mathbf{Z}_4^n$ or $V_2 = \mathbf{Z}_2^{2n}$, attempt to decode u with respect to C. 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 (in the Euclidean case it may happen that u' is not in C because there are too many errors in u to correct).

The lifted decoding algorithm comprises lifting decoding algorithms for two binary linear codes C_0 and C_1, being the residue and torsion codes of C. Let t_0 and t_1 be the error-correcting capability of C_0 and C_1, respectively. Assume the received vector $u = c + e$, where $c \in C$ and $e \in V$ is the error vector. Then, the lifted decoding algorithm can correct all error vectors e such that $\tau_1 + \tau_3 \leq t_0$ and $\tau_2 + \tau_3 \leq t_1$, where τ_i is the number of occurrences of i in e.

In the decoding process, the function `Decode(C, u)` for linear codes is used. The available algorithms for linear codes are: syndrome decoding and a Euclidean algorithm, which operates on alternant codes (BCH, Goppa, and Reed–Solomon codes, etc.). If C_0 or C_1 is alternant, the Euclidean algorithm is used by default, but the syndrome algorithm will be used if the parameter `AlgMethod` is assigned the value "Syndrome". For non-alternant codes C_0 and C_1, only syndrome decoding is possible, so the parameter AlgMethod is not relevant.

```
LiftedDecode(C, Q : parameters)
```

AlgMethod MonStgElt *Default : "Euclidean"*

Given a code C over $\mathbf{Z}_4$ of length n, and a sequence Q of vectors from the ambient space $V = \mathbf{Z}_4^n$ or $V_2 = \mathbf{Z}_2^{2n}$, attempt to decode the vectors of Q with respect to C. This function is similar to the function LiftedDecode(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 parameter AlgMethod are the same as for LiftedDecode(C, u).

Example H162E21

The Hadamard code C over $\mathbf{Z}_4$ of length 8 and type $2^1 4^2$ is constructed. Then an information word is encoded using C, three errors are introduced into the codeword c and then c is recovered by using the lifted decoding algorithm.

```
> C := HadamardCodeZ4(2, 4);
> C;
((8, 4^2 2^1, 8)) Linear Code over IntegerRing(4)
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]
> d := MinimumLeeDistance(C);
> t := Floor((d-1)/2);
> t;
3
> C0 := BinaryResidueCode(C);
> C1 := BinaryTorsionCode(C);
> t0 := Floor((MinimumDistance(C0)-1)/2);
> t1 := Floor((MinimumDistance(C1)-1)/2);
> t0, t1;
1 1
```

Using the lifted decoding, it is possible to correct all error vectors e such that $\tau_1 + \tau_3 \leq t_0 = 1$ and $\tau_2 + \tau_3 \leq t_1 = 1$, where τ_i is the number of occurrences of i in e. The following statements show that it is not possible to correct the error vector $e = (00003020)$ since $\tau_2 + \tau_3 = 2 > 1$, but it is possible to correct the error vector $e = (00001020)$ since $\tau_1 + \tau_3 = 1 \leq 1$ and $\tau_2 + \tau_3 = 1 \leq 1$.

```
> R, V, f, fbin := InformationSpace(C);
> i := R![2,1,0];
> c := f(i);
> c;
(1 0 3 2 3 2 1 0)
> u := c;
> u[5] := u[5] + 3;
> u[7] := u[7] + 2;
> c;
(1 0 3 2 3 2 1 0)
> u;
(1 0 3 2 2 2 3 0)
> e := u - c;
> e;
(0 0 0 0 3 0 2 0)
> isDecoded, uDecoded := LiftedDecode(C, u);
> isDecoded;
true
> uDecoded eq c;
false
> u := c;
> u[5] := u[5] + 1;
> u[7] := u[7] + 2;
> c;
(1 0 3 2 3 2 1 0)
> u;
(1 0 3 2 0 2 3 0)
> e := u - c;
> e;
(0 0 0 0 1 0 2 0)
> isDecoded, uDecoded := LiftedDecode(C, u);
> isDecoded;
true
> uDecoded eq c;
true
```

162.6.4 Permutation Decoding

Let C be a code over $\mathbf{Z}_4$ of length n and type $2^\gamma 4^\delta$ and $C_{bin} = \Phi(C)$, where Φ is the Gray map. A subset $S \subseteq \mathrm{Sym}(2n)$ is an s-PD-set for C_{bin} with respect to a subset of coordinate positions $I \subseteq \{1, \ldots, 2n\}$ if S is a subset of the permutation automorphism group of C_{bin}, I is an information set for C_{bin}, and every s-set of coordinate positions in $\{1, \ldots, 2n\}$ is moved out of the information set I by at least one element of S, where $1 \leq s \leq t$ and t is the error-correcting capability of C_{bin}.

If $I = [i_1, \ldots, i_{\gamma+\delta}] \subseteq \{1, \ldots, n\}$ is an information set for C such that the code obtained by puncturing C at positions $\{1, \ldots, n\} \backslash \{i_{\gamma+1}, \ldots, i_{\gamma+\delta}\}$ is of type 4^δ, then $\Phi(I) = [2i_1 - 1, \ldots, 2i_\gamma - 1, 2i_{\gamma+1} - 1, 2i_{\gamma+1}, \ldots, 2i_{\gamma+\delta} - 1, 2i_{\gamma+\delta}]$ is an information set for C_{bin}. It is also easy to see that if S is a subset of the permutation automorphism group of C, that is, $S \subseteq \mathrm{PAut}(C) \subseteq \mathrm{Sym}(n)$, then $\Phi(S) = [\Phi(\tau) : \tau \in S] \subseteq \mathrm{PAut}(C_{bin}) \subseteq \mathrm{Sym}(2n)$, where

$$\Phi(\tau)(i) = \begin{cases} 2\tau(i/2), & \text{if } i \text{ is even,} \\ 2\tau((i+1)/2) - 1 & \text{if } i \text{ is odd.} \end{cases}$$

Given a subset of coordinate positions $I \subseteq \{1, \ldots, n\}$ and a subset $S \subseteq \mathrm{Sym}(n)$, in order to check that $\Phi(S)$ is an s-PD-set for C_{bin} with respect to $\Phi(I)$, it is enough to check that S is a subset of the permutation automorphism group of C, I is an information set for C, and every s-set of coordinate positions in $\{1, \ldots, n\}$ is moved out of the information set I by at least one element of S [BV16a, BV16b].

```
IsPermutationDecodeSet(C, I, S, s)
```

Given a code C over $\mathbf{Z}_4$ of length n and type $2^\gamma 4^\delta$, a sequence $I \subseteq \{1, \ldots, 2n\}$, a sequence S of elements in the symmetric group $\mathrm{Sym}(2n)$ of permutations on the set $\{1, \ldots, 2n\}$, and an integer $s \geq 1$, return `true` if and only if S is an s-PD-set for $C_{bin} = \Phi(C)$, where Φ is the Gray map, with respect to the information set I.

The arguments I and S can also be given as a sequence $I \subseteq \{1, \ldots, n\}$ and a sequence S of elements in the symmetric group $\mathrm{Sym}(n)$ of permutations on the set $\{1, \ldots, n\}$, respectively. In this case, the function returns `true` if and only if $\Phi(S)$ is an s-PD-set for $C_{bin} = \Phi(C)$ with respect to the information set $\Phi(I)$, where $\Phi(I)$ and $\Phi(S)$ are the sequences defined as above.

Depending on the length of the code C, its type, and the integer s, this function could take some time to compute whether S or $\Phi(S)$ is an s-PD-set for C_{bin} with respect to I or $\Phi(I)$, respectively. Specifically, if the function returns `true`, it is necessary to check $\sum_{i=1}^{s} \binom{|I|}{i} \cdot \binom{N-|I|}{s-i}$ s-sets, where $N = n$ and $|I| = \gamma + \delta$ when I is given as an information set for C, or $N = 2n$ and $|I| = \gamma + 2\delta$ when I is given as an information set for C_{bin}.

The verbose flag `IsPDSetFlag` is set to level 0 by default. If it is set to level 1, the total time used to check the condition is shown. Moreover, the reason why the function returns `false` is also shown, that is, whether I is not an information set, S is not a subset of the permutation automorphism group or S is not an s-PD-set. If it is set to level 2, the percentage of the computation process performed is also printed.

```
PermutationDecode(C, I, S, s, u)
```

The arguments for the intrinsic are as follows:

- C is a code over $\mathbf{Z}_4$ of length n and type $2^\gamma 4^\delta$;
- $I = [i_1, \ldots, i_{\gamma+\delta}] \subseteq \{1, \ldots, n\}$ is the information set for C given as a sequence of coordinate positions such that the code obtained by puncturing C at coordinate positions $\{1, \ldots, n\} \backslash \{i_{\gamma+1}, \ldots, i_{\gamma+1}, \ldots, i_{\gamma+\delta}\}$ is of type 4^δ;
- S is a sequence such that either S or $\Phi(S)$ is an s-PD-set for $C_{bin} = \Phi(C)$, where Φ is the Gray map, with respect to $\Phi(I)$;
- s is an integer such that $s \in \{1, \ldots, t\}$ where t is the error-correcting capability of C_{bin};
- u is a vector from the ambient space $V = \mathbf{Z}_4^n$ or $V_2 = \mathbf{Z}_2^{2n}$.

Given the above assumptions, the function attempts to decode u with respect to C. If the decoding algorithm succeeds in computing a vector $u' \in C$ as the decoded version of $u \in V$, then the function returns the values `true`, u' and $\Phi(u')$. If the decoding algorithm does not succeed in decoding u, then the function returns the values `false`, the zero vector in V and the zero vector in V_2.

Assume that the received vector $\Phi(u) = c + e$, where $u \in V$, $c \in C_{bin}$ and $e \in V_2$ is the error vector with at most t errors. The permutation decoding algorithm proceeds by moving all errors in the received vector $\Phi(u)$ out of the information positions. That is, the nonzero coordinates of e are moved out of the information set $\Phi(I)$ for C_{bin}, by using an automorphism of C_{bin}.

Note that $\Phi(I)$ and $\Phi(S)$ are the sequences defined as above. Moreover, the function does not check whether I is an information set for C, nor whether S or $\Phi(S)$ is an s-PD-set for C_{bin} with respect to $\Phi(I)$, nor that $s \leq t$.

```
PermutationDecode(C, I, S, s, Q)
```

Given

- a code C over $\mathbf{Z}_4$ of length n and type $2^\gamma 4^\delta$;
- an information set $I = [i_1, \ldots, i_{\gamma+\delta}] \subseteq \{1, \ldots, n\}$ for C as a sequence of coordinate positions, such that the code C punctured on $\{1, \ldots, n\} \backslash \{i_{\gamma+1}, \ldots, i_{\gamma+\delta}\}$ is of type 4^δ;
- a sequence S such that either S or $\Phi(S)$ is an s-PD-set for $C_{bin} = \Phi(C)$, where Φ is the Gray map, with respect to $\Phi(I)$;
- an integer $s \in \{1, \ldots, t\}$, where t is the error-correcting capability of C_{bin};
- and a sequence Q of vectors from the ambient space $V = \mathbf{Z}_4^n$ or $V_2 = \mathbf{Z}_2^{2n}$,

attempt to decode the vectors of Q with respect to C. This function is similar to the version of `PermutationDecode` that decodes a single vector except that 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 used by the single vector version of `PermutationDecode`.

Example H162E22

First the Hadamard code C over $\mathbf{Z}_4$ of length 32 and type $2^1 4^3$ is constructed. It is known that $I = [17, 1, 2, 5]$ is an information set for C and $S = \{\pi^i : 1 \leq i \leq 8\}$, where $\pi = (1, 24, 26, 15, 3, 22, 28, 13)\ (2, 23, 27, 14, 4, 21, 25, 16)\ (5, 11, 32, 20, 7, 9, 30, 18)\ (6, 10, 29, 19, 8, 12, 31, 17)$, is a subset of the permutation automorphism group of C such that $\Phi(S)$ is a 7-PD-set for $C_{bin} = \Phi(C)$ with respect to $\Phi(I)$. Then, choosing a codeword c of C, c is perturbed by the addition of an error vector to give a new vector u, and finally permutation decoding is applied to u to recover c.

```
> C := HadamardCodeZ4(3, 6);
> C;
((32, 4^3 2^1)) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 3 2 0 3 2 1 3 2 1 0 2 1 0 3 1 0 3 2 0 3 2 1 3 2 1 0 2 1 0 3]
[0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3]
[0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
```

```
> t := Floor((MinimumLeeDistance(C) - 1)/2);
> t;
15
> I := [17, 1, 2, 5];
> p := Sym(32)!(1, 24, 26, 15, 3, 22, 28, 13)(2, 23, 27, 14, 4, 21, 25, 16)
>                 (5, 11, 32, 20, 7, 9, 30, 18)(6, 10, 29, 19, 8, 12, 31,17);
> S := [ p^i : i in [1..8] ];
> SetVerbose("IsPDSetFlag", 2);
> IsPermutationDecodeSet(C, I, S, 7);
Checking whether I is an information set...
Checking whether S is in the permutation automorphism group...
Checking whether S is an s-PD-set...
10 %
20 %
30 %
40 %
50 %
60 %
70 %
80 %
90 %
Took 136.430 seconds (CPU time)
true
> SetVerbose("IsPDSetFlag", 0);
> c := C ! [1,2,3,0,0,1,2,3,3,0,1,2,2,3,0,1,3,0,1,2,2,3,0,1,1,2,3,0,0,1,2,3];
> c in C;
true
> u := c;
> u[1] := c[1] + 2;
> u[2] := c[2] + 2;
> u[3] := c[3] + 1;
> u[16] := c[16] + 3;
> u[27] := c[27] + 1;
> u in C;
false
> LeeDistance(u, c);
7
> grayMap := GrayMap(UniverseCode(Integers(4), Length(C)));
> cbin := grayMap(c);
> ubin := grayMap(u);
> Distance(ubin, cbin);
7
> isDecoded, uDecoded, ubinDecoded := PermutationDecode(C, I, S, 7, u);
> isDecoded;
true
> uDecoded eq c;
true
> ubinDecoded eq cbin;
```

```
true
> isDecoded, uDecoded, ubinDecoded := PermutationDecode(C, I, S, 7, ubin);
> isDecoded;
true
> uDecoded eq c;
true
> ubinDecoded eq cbin;
true
```

PDSetHadamardCodeZ4(δ, m)

AlgMethod MonStgElt *Default : "Deterministic"*

Given an integer $m \geq 5$, and an integer δ such that $3 \leq \delta \leq \lfloor (m+1)/2 \rfloor$, the Hadamard code C over $\mathbf{Z}_4$ of length $n = 2^{m-1}$ and type $2^\gamma 4^\delta$, where $\gamma = m+1-2\delta$, given by the function HadamardCodeZ4(δ, m), is considered. The function returns an information set $I = [i_1, \ldots, i_{\gamma+\delta}] \subseteq \{1, \ldots, n\}$ for C together with a subset S of the permutation automorphism group of C such that $\Phi(S)$ is an s-PD-set for $C_{bin} = \Phi(C)$ with respect to $\Phi(I)$, where Φ is the Gray map and $\Phi(I)$ and $\Phi(S)$ are defined above. The function also returns the information set $\Phi(I)$ and the s-PD-set $\Phi(S)$. For $m \geq 1$ and $1 \leq \delta \leq 2$, the Gray map image of C is linear and it is possible to find an s-PD-set for $C_{bin} = \Phi(C)$, for any $s \leq \lfloor 2^m/(m+1) \rfloor - 1$, by using the function PDSetHadamardCode(m).

The information sets I and $\Phi(I)$ are returned as sequences of $\gamma + \delta$ and $\gamma + 2\delta$ integers, giving the coordinate positions that correspond to the information sets for C and C_{bin}, respectively. The sets S and $\Phi(S)$ are also returned as sequences of elements in the symmetric groups Sym(n) and Sym($2n$) of permutations on the set $\{1, \ldots, n\}$ and $\{1, \ldots, 2n\}$, respectively.

A deterministic algorithm is used by default. In this case, the function returns the s-PD-set of size $s+1$ with $s = \lfloor (2^{2\delta-2} - \delta)/\delta \rfloor$, which is the maximum value of s when $\gamma = 0$, as described in [BV16a]. If the parameter AlgMethod is assigned the value "Nondeterministic", the function tries to improve the previous result by finding an s-PD-set of size $s+1$ such that $\lfloor (2^{2\delta-2} - \delta)/\delta \rfloor \leq s \leq \lfloor (2^{m-1} + \delta - m - 1)/(m+1-\delta) \rfloor$. In this case, the function starts from the maximum value of s and decreases it if the s-PD-set is not found after a specified time.

PDSetKerdockCodeZ4(m)

Given an integer $m \geq 4$ such that $2^m - 1$ is not a prime number, the Kerdock code C over $\mathbf{Z}_4$ of length $n = 2^m$ and type 4^{m+1}, given by the function KerdockCodeZ4(m) is considered. The function returns the information set $I = [1, \ldots, m+1]$ for C together with a subset S of the permutation automorphism group of C such that $\Phi(S)$ is an s-PD-set for $C_{bin} = \Phi(C)$ with respect to $\Phi(I)$, where Φ is the Gray map and $\Phi(I)$ and $\Phi(S)$ are defined above. The function also returns the information set $\Phi(I) = [1, \ldots, 2m+2]$ and the s-PD-set $\Phi(S)$. The size of the s-PD-set S is always $\lambda = s+1$, where λ is the greatest divisor of $2^m - 1$ such that $\lambda \leq 2^m/(m+1)$.

The information sets I and $\Phi(I)$ are returned as sequences of $m+1$ and $2m+2$ integers, giving the coordinate positions that correspond to the information sets for C and C_{bin}, respectively. The sets S and $\Phi(S)$ are also returned as sequences of elements in the symmetric groups Sym(n) and Sym($2n$) of permutations on the sets $\{1,\ldots,n\}$ and $\{1,\ldots,2n\}$, respectively. The s-PD-set S contains the $s+1$ permutations described in [BV16b].

Example H162E23

A 4-PD-set S of size 5 for the Hadamard code C over $\mathbf{Z}_4$ of length 16 and type $2^0 4^3$ is constructed. A check that it really is a 4-PD-set for C is then made. Note that $\lfloor (2^{2\delta-2}-\delta)/\delta \rfloor = 4$. Finally, a codeword c of C is selected, perturbed by an error vector e to give a vector u, to which permutation decoding is applied to recover c.

```
> C := HadamardCodeZ4(3, 5);
> I, S, Ibin, Sbin := PDSetHadamardCodeZ4(3, 5);
> s := #Sbin-1; s;
4
> s eq Floor((2^(2*3-2)-3)/3);
true
> IsPermutationDecodeSet(C, I, S, s);
true
> IsPermutationDecodeSet(C, Ibin, Sbin, s);
true
> c := C ! [3,2,1,0,1,0,3,2,3,2,1,0,1,0,3,2];
> R := UniverseCode(Integers(4), Length(C));
> u := R ! [2,3,2,0,1,0,3,2,3,2,1,0,1,0,3,3];
> u in C;
false
> LeeDistance(u, c);
4
> grayMap := GrayMap(R);
> cbin := grayMap(c);
> isDecoded, uDecoded, ubinDecoded := PermutationDecode(C, I, S, 4, u);
> isDecoded;
true
> uDecoded eq c;
true
> ubinDecoded eq cbin;
true
```

For the Hadamard code C over $\mathbf{Z}_4$ of length 32 and type $2^1 4^3$, a 4-PD-set of size 5 can be constructed either by using the deterministic method (by default), or by using a nondeterministic method to obtain an s-PD-set of size $s+1$ with $4 \leq s \leq 7$. In both cases, the given sets are checked for really being s-PD-sets for C.

```
> C := HadamardCodeZ4(3, 6);
> I, S, Ibin, Sbin := PDSetHadamardCodeZ4(3, 6);
> s := #Sbin-1; s;
```

```
4
> IsPermutationDecodeSet(C, I, S, s);
true
> I, S, Ibin, Sbin := PDSetHadamardCodeZ4(3, 6 : AlgMethod := "Nondeterministic");
> s := #Sbin-1; s;
6
> IsPermutationDecodeSet(C, I, S, s);
true
```

Finally, a 2-PD-set of size 3 is constructed for the Kerdock code of length 16 and type $2^0 4^5$, and formally checked for being a 2-PD-set for this code.

```
> C := KerdockCode(4);
> I, S, Ibin, Sbin := PDSetKerdockCodeZ4(4);
> IsPermutationDecodeSet(C, I, S, 2);
true
> IsPermutationDecodeSet(C, Ibin, Sbin, 2);
true
```

162.7 Automorphism Groups

PermutationGroupHadamardCodeZ4(δ, m)

PAutHadamardCodeZ4(δ, m)

Given an integer $m \geq 1$ and an integer δ such that $1 \leq \delta \leq \lfloor (m+1)/2 \rfloor$, this function returns the permutation group G of a Hadamard code over $\mathbf{Z}_4$ of length 2^{m-1} and type $2^\gamma 4^\delta$, where $\gamma = m + 1 - 2\delta$. The group G contains all permutations of the coordinates which preserve the code. Thus only permutation of coordinates is allowed, and the degree of G is always 2^{m-1}. Moreover, the generator matrix with $\gamma + \delta$ rows used to generate the code is returned. This matrix is constructed in a recursive way using the Plotkin and BQPlotkin constructions defined in Section 162.2.4.

PermutationGroupHadamardCodeZ4Order(δ, m)

PAutHadamardCodeZ4Order(δ, m)

Given an integer $m \geq 1$ and an integer δ such that $1 \leq \delta \leq \lfloor (m+1)/2 \rfloor$, return the order of the permutation group G of a Hadamard code over $\mathbf{Z}_4$ of length 2^{m-1} and type $2^\gamma 4^\delta$, where $\gamma = m + 1 - 2\delta$. The group G contains all permutations of the coordinates which preserve the code.

`PermutationGroupExtendedPerfectCodeZ4(δ, m)`

`PAutExtendedPerfectCodeZ4(δ, m)`

Given an integer $m \geq 2$ and an integer δ such that $1 \leq \delta \leq \lfloor (m+1)/2 \rfloor$, return the permutation group G of an extended perfect code over $\mathbf{Z}_4$ of length 2^{m-1}, such that its dual code is of type $2^{\gamma}4^{\delta}$, where $\gamma = m + 1 - 2\delta$. The group G contains all permutations of the coordinates which preserve the code. Thus only permutation of coordinates is allowed, and the degree of G is always 2^{m-1}. Moreover, the generator matrix with $\gamma + \delta$ rows used to generate the code is returned. This matrix is constructed in a recursive way using the Plotkin and BQPlotkin constructions defined in Section 162.2.4.

`PermutationGroupExtendedPerfectCodeZ4Order(δ, m)`

`PAutExtendedPerfectCodeZ4Order(δ, m)`

Given an integer $m \geq 2$ and an integer δ such that $1 \leq \delta \leq \lfloor (m+1)/2 \rfloor$, return the order of the permutation group G of an extended perfect code over $\mathbf{Z}_4$ of length 2^{m-1}, such that its dual code is of type $2^{\gamma}4^{\delta}$, where $\gamma = m + 1 - 2\delta$. The group G contains all permutations of the coordinates which preserve the code.

Example H162E24

```
> C := HadamardCodeZ4(2,4);
> PAut := PAutHadamardCodeZ4(2,4);
> PAut;
Permutation group PAut acting on a set of cardinality 8
    (1, 2)(3, 4)(5, 6)(7, 8)
    (2, 4)(6, 8)
    (5, 7)(6, 8)
    (1, 5)(3, 7)
> {p : p in Sym(8) | C^p eq C} eq Set(PAut);
true
> #PAut eq PAutHadamardCodeZ4Order(2,4);
true
> d := 2; m := 4; g := m+1-2*d;
> PAutHadamardCodeZ4Order(d, m) eq
>  #GL(d-1,Integers(4))*#GL(g,Integers(2))*2^g*4^((g+1)*(d-1));
true
> d := 4; m := 8; g := m+1-2*d;
> PAutHadamardCodeZ4Order(d, m) eq
>   #GL(d-1,Integers(4))*#GL(g,Integers(2))*2^g*4^((g+1)*(d-1));
true
> PAutHadamardCodeZ4(2,4) eq PAutExtendedPerfectCodeZ4(2,4);
true
```

162.8 Bibliography

[**BBFCV15**] J. J. Bernal, J. Borges, C. Fernández-Córboda, and M. Villanueva. Permutation decoding of $\mathbf{Z}_2\mathbf{Z}_4$-linear codes. *Des. Codes and Cryptogr.*, 76(2):269–277, 2015.

[**BV16a**] R. Barrolleta and M. Villanueva. Partial permutation decoding for binary linear and $\mathbf{Z}_4$-linear Hadamard codes. Submitted to *Designs, Codes and Cryptography*, arXiv:1512.01839, 2016.

[**BV16b**] R. Barrolleta and M. Villanueva. PD-sets for $\mathbf{Z}_4$-linear codes: Hadamard and Kerdock codes. *Proceedings of the IEEE International Symposium on Information Theory*, 2016.

[**BZ01**] N.S. Babu and K.H. Zimmermann. Decoding of linear codes over Galois rings. *IEEE Trans. on Information Theory*, 47(4):1599–1603, 2001.

[**FCPV08**] C. Fernández-Córdoba, J. Pujol, and M. Villanueva. *On rank and kernel of $\mathbf{Z}_4$-linear codes*, pages 46–55. Number 5228 in Lecture Notes in Computer Science. 2008.

[**FCPV10**] C. Fernández-Córdoba, J. Pujol, and M. Villanueva. $\mathbf{Z}_2\mathbf{Z}_4$-linear codes: rank and kernel. *Designs, Codes and Cryptography*, 56(1):43–59, 2010.

[**GV98**] M. Greferath and U. Velbinger. Efficient decoding of $\mathbf{Z}_{p^k}$-linear codes. *IEEE Trans. on Information Theory*, 44:1288–1291, 1998.

[**HKC$^+$94**] A.R. Hammons, P.V. Kumar, A.R. Calderbank, N.J.A. Sloane, and P. Solé. The $\mathbf{Z}_4$-linearity of kerdock, preparata, goethals and related codes. *IEEE Trans. on Information Theory*, 40:301–319, 1994.

[**MS78**] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes.* North Holland, New York, 1978.

[**VZP15**] M. Villanueva, F. Zeng, and J. Pujol. Efficient representation of binary nonlinear codes: constructions and minimum distance computation. *Designs, Codes and Cryptography*, 76(1):3–21, 2015.

[**Wan97**] Zhe-Xian Wan. *Quaternary Codes*, volume 8 of *Series on Applied Mathematics.* World Scientific, Singapore, 1997.

163 ADDITIVE CODES

Chapter 163
ADDITIVE CODES

163.1 Introduction

The concept of a linear codes over a finite field (see Chapter 158) can be generalized to the notion of an *additive* code. Given a finite field F and the space of all n-tuples of F, an additive code is a subset of $F^{(n)}$ which is a K-linear subspace for some subfield $K \subseteq F$.

Additive codes have become increasingly important recently due to their application to the construction of quantum error-correcting codes, though they are also of interest in their own right. A Magma package for quantum error-correcting codes is built on the machinery for additive codes.

In Magma an additive code has both an *alphabet* F and a *coefficient field* K, which is a subfield of F. An error-correcting code is considered to be defined by its wordset, so there may be several different ways of presenting a given code using different coefficient fields.

Since a given code may be presented over different coefficient rings, the *dimension* k of an additive code is defined relative to the alphabet of the code, $\#C = (\#F)^k$, leading to possibility of *fractional* dimensions. Consequently, the number of generators of an additive code will not equal its dimension, there being $[F : K]$ times as many generators. So a length n K-additive code over F has between zero and $n * [F : K]$ generators.

For example, consider the two length 3 vectors over F_4: $(1, 0, \omega^2), (0, \omega, 0)$. The linear code generated by these vectors consists of all scalar multiples and sums, resulting in a total of $4^2 = 16$ vectors. But the F_2-additive code generated by these two vectors contains *only their sums*, resulting in a total of $2^2 = 4$ vectors. These vectors are $(0, 0, 0), (1, 0, \omega^2), (0, \omega, 0), (1, \omega, \omega^2)$. The alphabet of this code is F_4, its coefficient field is F_2, it has 2 generators and is of dimension 1.

A length n, dimension k K-additive code over F with k_g generators is represented in Magma as an $[n, k : k_g]$ K-additive code over F. The concepts of weight, distance and their respective distributions and enumerators transfer directly from linear codes. An $[n, k : k_g, d]$ K-additive code over F is a K-linear subset of $F^{(n)}$ which has fractional dimension k, k_g generators and a minimum weight of d.

As a general rule, additive and linear codes may be used interchangeably. Indeed any linear code can be expressed as an additive code, using either its alphabet or any subfield as its coefficient field. So any linear code over a finite field, of type `CodeLinFld`, is in fact also an additive code, of type `CodeAdd`. The theory of purely linear codes is more general than that of additive codes so unfortunately not all operations are transferable.

163.2 Construction of Additive Codes

163.2.1 Construction of General Additive Codes

```
AdditiveCode< F, K, n | L >
```

Create the K-additive code in $F^{(n)}$ of length n which is generated by the elements specified by the list L, where K is a subfield of F and L is one or more items of the following types:

(a) An element of $F^{(n)}$;

(b) A set or sequence of elements of $F^{(n)}$;

(c) A sequence of n elements of F, defining an element of $F^{(n)}$;

(d) A set or sequence of sequences of type (c);

(e) A subcode of $F^{(n)}$;

```
AdditiveCode(G)
AdditiveCode(K, G)
```

Given a matrix G over a field F and a subfield K of F, return the K-additive code over F generated by the rows of G. If no coefficient field K is specified, then the prime field of F is used.

Example H163E1

Starting with two linearly independent vectors in $\mathbf{F}_4^{(3)}$, we compare the linear code over F_4 they generate with the corresponding F_2-additive code.

```
> F<w> := GF(4);
> G := Matrix(F, 2, 3, [1,0,w^2,0,w,0]);
> G;
[  1   0 w^2]
[  0   w   0]
> C1 := LinearCode(G);
> C2 := AdditiveCode(GF(2), G);
> #C1;
16
> #C2;
4
> C2 subset C1;
true
```

The codewords of C_2 are arise only through addition of the generators: scalar multiplication is not permitted.

```
> { v : v in C2 };
{
    (  1   w w^2),
    (  0   0   0),
    (  1   0 w^2),
    (  0   w   0)
}
```

Example H163E2

We define an $\mathbf{F}_2$-additive code over $\mathbf{F}_8$ by constructing a random matrix and considering the code generated by its rows. Note that the number of generators exceeds the length of the code.

```
> K<w> := GF(8);
> M := KMatrixSpace(K, 5, 4);
> C := AdditiveCode(GF(2), Random(M));
> C;
[4, 1 2/3 : 5] GF(2)-Additive Code over GF(2^3)
Generator matrix:
[  1   1 w^2   0]
[  w w^2   w   1]
[w^2 w^2 w^2   1]
[  0 w^4 w^4 w^5]
[  0   0   1   0]
> WeightDistribution(C);
[ <0, 1>, <1, 1>, <2, 2>, <3, 9>, <4, 19> ]
> C;
[4, 1 2/3 : 5, 1] GF(2)-Additive Code over GF(2^3)
Generator matrix:
[  1   1 w^2   0]
[  w w^2   w   1]
[w^2 w^2 w^2   1]
[  0 w^4 w^4 w^5]
[  0   0   1   0]
```

`AdditiveCode(K, C)`

Given a code (linear or additive) C over some finite field F, and a subfield K of F such that the wordset of C forms a K-linear subspace, then return C as a K-additive code.

Example H163E3

Any linear code can be regarded as an additive code with respect to a subfield of its alphabet.

```
> C := RandomLinearCode(GF(4), 8, 3);
> C:Minimal;
[8, 3, 4] Linear Code over GF(2^2)
> A1 := AdditiveCode(GF(4), C);
> A1:Minimal;
[8, 3 : 3, 4] GF(2^2)-Additive Code over GF(2^2)
> { v : v in C } eq {v : v in A1 };
true
```

```
>
> A2 := AdditiveCode(GF(2), C);
> A2:Minimal;
[8, 3 : 6, 4] GF(2)-Additive Code over GF(2^2)
> { v : v in C } eq {v : v in A2 };
true
```

Example H163E4

A K-additive code over F can be viewed as an E-additive code for any subfield $E \subseteq K$.

```
> C4 := RandomAdditiveCode(GF(16), GF(4), 8, 5);
> C4:Minimal;
[8, 2 1/2 : 5] GF(2^2)-Additive Code over GF(2^4)
>
> C2 := AdditiveCode(GF(2), C4);
> C2:Minimal;
[8, 2 1/2 : 10] GF(2)-Additive Code over GF(2^4)
> { v : v in C2 } eq {v : v in C4 };
true
```

But for any E such that $K \subset E \subseteq F$ we can create an E-additive code if and only if the wordset is in fact an E-linear subspace.

```
> C2:Minimal;
[8, 2 1/2 : 10] GF(2)-Additive Code over GF(2^4)
> A1 := AdditiveCode(GF(4), C2);
> A1 eq C4;
true
>  A2 := AdditiveCode(GF(16), C2);
>> A2 := AdditiveCode(GF(16), C2);
                     ^

Runtime error in 'AdditiveCode': Code is not additive over given field
```

163.2.2 Some Trivial Additive Codes

`AdditiveZeroCode(F, K, n)`

Given a field F and subfield $K \subseteq F$ along with a positive integer n, return the $[n, 0, n]$ code consisting of only the zero code word, (where the minimum weight is by convention equal to n).

`AdditiveRepetitionCode(F, K, n)`

Given a field F and subfield $K \subseteq F$ along with a positive integer n, return the $[n, 1, n]$ code consisting of all repeating codewords.

```
AdditiveZeroSumCode(F, K, n)
```

Given a field F and subfield $K \subseteq F$ along with a positive integer n, return the $[n, n-1, 2]$ K-additive code over F such that for all codewords $(c_1, c_2, \ldots, c_n)$, we have $\sum_i c_i = 0$.

```
AdditiveUniverseCode(F, K, n)
```

Given a field F and subfield $K \subseteq F$ along with a positive integer n, return the $[n, n, 1]$ K-additive code over F consisting of all possible codewords.

```
RandomAdditiveCode(F, K, n, k)
```

Given a field F and subfield $K \subseteq F$ along with positive integers n and k, such that $0 < k \leq n * [F : K]$, and k, return a random K-additive code of length n and k generators over the field F.

Example H163E5

Over any finite field chain $K \subseteq F$, the zero code of length n is contained in every code of length n, and similarly every code of length n is contained in the universe code of length n.

```
> F := GF(9);
> K := GF(3);
> U := AdditiveUniverseCode(F, K, 5);
> Z := AdditiveZeroCode(F, K, 5);
> R := RandomAdditiveCode(F, K, 5, 2);
> (Z subset R) and (R subset U);
true
```

163.3 Invariants of an Additive Code

163.3.1 The Ambient Space and Alphabet

A length n additive code has an alphabet F and coefficient field $K \subseteq F$. The code consists of codewords which are a K-linear subspace of $F^{(n)}$.

```
Alphabet(C)
Field(C)
```

The underlying field (or alphabet) of the codewords of the additive code C. A length n additive code with alphabet F consists of codewords from $F^{(n)}$.

```
CoefficientField(C)
```

The field over which the codewords of the additive code C are considered linear. This will be a subfield of the alphabet of C.

`AmbientSpace(C)`

The ambient space of the additive code C, i.e. the generic R-space V in which C is contained.

`Generic(C)`

Given a length n additive code C, return the generic $[n, n, 1]$ code in which C is contained.

Example H163E6

A code can often be represented using several different coefficient fields.

```
> F<w> := GF(5,4);
> K := GF(5,2);
> C := RandomAdditiveCode(F, K, 12, 5);
> C:Minimal;
[12, 2 1/2 : 5] GF(5^2)-Additive Code over GF(5^4)
> #C;
9765625
> Alphabet(C);
Finite field of size 5^4
> CoefficientField(C);
Finite field of size 5^2
>
> C1 := AdditiveCode(GF(5), C);
> C1:Minimal;
[12, 2 1/2 : 10] GF(5)-Additive Code over GF(5^4)
> #C1;
9765625
> Alphabet(C1);
Finite field of size 5^4
> CoefficientField(C1);
Finite field of size 5
```

163.3.2 Basic Numerical Invariants

`Length(C)`

Return the block length n of an additive code C.

`Dimension(C)`

The (rational) dimension k of C. If the alphabet of C is F, then the dimension is defined by the equation $\#C = (\#F)^k$.

Note that since any basis of the additive code C is relative to the coefficient field K, this dimension is not necessarily equal to the number of generators of C and is not even necessarily integral.

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

The number of generators of the additive code C. Note that if the coefficient ring of C is not the same as its alphabet then this will be different from the dimension of C.

```
#C
```

Given an additive code C, return the number of codewords belonging to C.

```
InformationRate(C)
```

The information rate of the $[n, k]$ code C. This is the ratio k/n.

163.3.3 The Code Space

```
GeneratorMatrix(C)
```
```
BasisMatrix(C)
```

The generator matrix for an $[n, k(k_g)]$ K-additive code C over F is a $k_g \times n$ matrix over F, whose k_g rows form a basis for C when considered as vectors over K.

```
Basis(C)
```
```
Generators(C)
```

A basis for the K-additive code C, returned as a sequence of codewords over the alphabet of C, which generate the code over K.

```
C . i
```

Given an $[n, k(k_g)]$ K-additive code C and a positive integer i, $1 \leq i \leq k_g$, return the i-th element of the current basis of C over K.

163.3.4 The Dual Space

```
Dual(C)
```

The code that is dual to the code C. For an additive code C, this is the nullspace with respect to the trace inner product of the coefficient field.

```
ParityCheckMatrix(C)
```

The parity check matrix for the code C, returned as an element of $\mathrm{Hom}(V, U)$.

163.4 Operations on Codewords

163.4.1 Construction of a Codeword

C ! [$\mathtt{a}_1$, ..., $\mathtt{a}_n$]

elt< C | $\mathtt{a}_1$, ..., $\mathtt{a}_n$ >

Given a length n additive code C with alphabet F, then the codewords of C lie in $F^{(n)}$. Given elements $a_1, \ldots, a_n$ belonging to F, construct the codeword $(a_1, \ldots, a_n)$ of C. A check is made that the vector $(a_1, \ldots, a_n)$ is an element of C.

```
C ! u
```

Given an additive code C which is defined as a subset of the F-space $V = F^{(n)}$, and an element u belonging to V, create the codeword of C corresponding to u. The function will fail if u does not belong to C.

```
C ! 0
```

The zero word of the additive code C.

```
Random(C)
```

A random codeword of the additive code C.

163.4.2 Arithmetic Operations on Codewords

```
u + v
```

Sum of the codewords u and v, where u and v belong to the same linear code C.

```
-u
```

Additive inverse of the codeword u belonging to the linear code C.

```
u - v
```

Difference of the codewords u and v, where u and v belong to the same linear code C.

```
a * u
```

Given an element a belonging to the alphabet F, and a codeword u belonging to the additive code C, return the codeword $a * u$.

```
Normalize(u)
```

Normalize a codeword u of an additive code C, returning a scalar multiple of u such that its first non-zero entry is 1.

163.4.3 Distance and Weight

```
Distance(u, v)
```

The Hamming distance between the codewords u and v, where u and v belong to the same additive code C.

```
Weight(u)
```

The Hamming weight of the codeword u, i.e., the number of non-zero components of u.

163.4.4 Vector Space and Related Operations

```
(u, v)
```

```
InnerProduct(u, v)
```

Inner product of the vectors u and v with respect to the Euclidean norm, where u and v belong to the parent vector space of the code C.

```
TraceInnerProduct(K, u, v)
```

Given vectors u and v defined over a finite field L and a subfield K of L, this function returns the trace of the inner product of the vectors u and v with respect to K.

```
Support(w)
```

Given a word w belonging to the $[n, k]$ code C, return its support as a subset of the integer set $\{1..n\}$. The support of w consists of the coordinates at which w has non-zero entries.

```
Coordinates(C, u)
```

Given an $[n, k : k_g]$ K-additive code C and a codeword u of C return the coordinates of u with respect to the current basis of C. The coordinates of u are returned as a sequence $Q = [a_1, \ldots, a_{k_g}]$ of elements from K such that $u = a_1 {*} C.1 + \ldots + a_{k_g} {*} C.k_g$.

```
Parent(w)
```

Given a word w belonging to the code C, return the ambient space V of C.

```
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.

```
Rotate(~u, k)
```

Given a vector u, destructively rotate u by k coordinate positions.

```
Trace(u, S)
```

```
Trace(u)
```

Given a vector u with components in K, and a subfield S of K, construct the vector with components in S obtained from u by taking the trace of each component with respect to S. If S is omitted, it is taken to be the prime field of K.

163.4.5 Predicates for Codewords

```
u eq v
```

The function returns `true` if and only if the codewords u and v belonging to the same additive code are equal.

```
u ne v
```

The function returns `true` if and only if the codewords u and v belonging to the same additive code are not equal.

```
IsZero(u)
```

The function returns `true` if and only if the codeword u is the zero vector.

163.4.6 Accessing Components of a Codeword

```
u[i]
```

Given a codeword u belonging to the code C defined over the ring R, return the i-th component of u (as an element of R).

```
u[i] := x;
```

Given an element u belonging to a subcode C of the full R-space $V = R^n$, a positive integer i, $1 \leq i \leq n$, and an element x of R, this function returns a vector in V which is u with its i-th component redefined to be x.

163.5 Subcodes

163.5.1 The Subcode Constructor

```
sub< C | L >
```

Given a K-additive linear code C over F, construct the subcode of C, generated (over K) by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) An element of C;

(b) A set or sequence of elements of C;

(c) A sequence of n elements of F, defining an element of C;

(d) A set or sequence of sequences of type (c);

(e) A subcode of C;

```
Subcode(C, k)
```

Given an additive code C and an integer k, where k is less than the number of generators of C, then return a subcode of C with k generators.

`Subcode(C, S)`

Suppose C is an additive code and S is a set of positive integers, each of which is less than the number of generators of C. The function returns the subcode of C generated by the generators of C indexed by S.

`SubcodeBetweenCode(C1, C2, k)`

Given an additive code C_1 and a subcode C_2 of C_1, return a subcode of C_1 with k generators containing C_2.

`SubcodeWordsOfWeight(C, w)`

Given a length n additive code C and an integer which lies in the range $[1, n]$, return the subcode of C generated by those words of C of weight w.

`SubcodeWordsOfWeight(C, S)`

Given a length n additive code C and a set S of integers, each of which lies in the range $[1, n]$, return the subcode of C generated by those words of C whose weights lie in S.

Example H163E7

We give an example of how `SubcodeBetweenCode` may be used to create a code nested in between a subcode pair.

```
> F<w> := GF(8);
> C1 := AdditiveRepetitionCode(F, GF(2), 6);
> C1;
[6, 1 : 3, 6] GF(2)-Additive Code over GF(2^3)
Generator matrix:
[  1   1   1   1   1   1]
[  w   w   w   w   w   w]
[w^2 w^2 w^2 w^2 w^2 w^2]
> C3 := AdditiveZeroSumCode(F, GF(2), 6);
> C3;
[6, 5 : 15, 2] GF(2)-Additive Code over GF(2^3)
Generator matrix:
[  1   0   0   0   0   1]
[  w   0   0   0   0   w]
[w^2   0   0   0   0 w^2]
[  0   1   0   0   0   1]
[  0   w   0   0   0   w]
[  0 w^2   0   0   0 w^2]
[  0   0   1   0   0   1]
[  0   0   w   0   0   w]
[  0   0 w^2   0   0 w^2]
[  0   0   0   1   0   1]
[  0   0   0   w   0   w]
[  0   0   0 w^2   0 w^2]
[  0   0   0   0   1   1]
[  0   0   0   0   w   w]
[  0   0   0   0 w^2 w^2]
> C1 subset C3;
true
> C2 := SubcodeBetweenCode(C3, C1, 11);
> C2;
[6, 3 2/3 : 11] GF(2)-Additive Code over GF(2^3)
Generator matrix:
[  1   0   0   0   1   0]
[  w   0   0   0   w   0]
[w^2   0   0 w^2 w^2 w^2]
[  0   1   0   0   0   1]
[  0   w   0   0   0   w]
[  0 w^2   0   0   0 w^2]
[  0   0   1   0   0   1]
[  0   0   w   0   0   w]
[  0   0 w^2   0   0 w^2]
[  0   0   0   1   0   1]
[  0   0   0   w   0   w]
> (C1 subset C2) and (C2 subset C3);
true
```

163.5.2 Sum, Intersection and Dual

For the following operators, C and D are additive codes defined as subsets (or subspaces) of the same R-space F^n.

C + D

Given two additive codes which have the same length, which are defined over the same alphabet, and which have the same coefficient ring F, return the sum of these two codes with respect to F.

C meet D

The intersection of the additive codes C and D.

Dual(C)

The code that is dual to the code C. For an additive code C, this is the code generated by the nullspace of C, relative to the trace inner product.

163.5.3 Membership and Equality

```
u in C
```

Return `true` if and only if the vector u of V belongs to the additive code C, where V is the generic vector space containing C.

```
u notin C
```

Return `true` if and only if the vector u does not belong to the additive code C, where V is the generic vector space containing C.

```
C subset D
```

Return `true` if and only if the wordset of the code C is a subset of the wordset of the code D. (Either code may possibly be additive).

```
C notsubset D
```

Return `true` if and only if the wordset of the code C is not a subset of the wordset of the code D. (Either code may possibly be additive).

```
C eq D
```

Return `true` if and only if the codes C and D have the same wordsets. (Either code may possibly be additive).

```
C ne D
```

Return `true` if and only if the codes C and D have different wordsets. (Either code may possibly be additive).

163.6 Properties of Codes

For the following operators, C and D are codes defined as a subset (or subspace) of the vector space V.

```
IsSelfDual(C)
```

Return `true` if and only if the linear code C is self-dual (or self-orthogonal) (i.e. C equals the dual of C).

```
IsSelfOrthogonal(C)
```

Return `true` if and only if the linear code C is self-orthogonal (i.e. C is contained in the dual of C).

```
IsPerfect(C)
```

Return `true` if and only if the linear code C is perfect; that is, if and only if the cardinality of C is equal to the size of the sphere packing bound of C.

```
IsProjective(C)
```

Returns `true` if and only if the (non-quantum) code C is projective over its alphabet.

```
IsAdditiveProjective(C)
```

Returns `true` if and only if the additive code C is projective over its coefficient field. It is possible that some of the columns may not be independent with respect to the alphabet of the code.

163.7 The Weight Distribution

163.7.1 The Minimum Weight

An adaptation of the minimum weight algorithm for linear codes (see Section 158.8.1) has been developed for additive codes by Markus Grassl and Greg White.

From a user's perspective, the description given in 158.8.1 is sufficient to understand the additive case. The algorithm is still new, and has yet to be optimised to its full potential.

`MinimumWeight(C:` *parameters*`)`

`MinimumDistance(C:` *parameters*`)`

`RankLowerBound`	RngIntElt	*Default :* 0
`MaximumTime`	RngReSubElt	*Default :* ∞

Determine the minimum weight of the words belonging to the code C, which is also the minimum distance between any two codewords. The parameter `RankLowerBound` sets a minimum rank on the information sets used in the calculation, while the parameter `MaximumTime` sets a time limit (in seconds of "user time") after which the calculation is aborted.

By setting the verbose flag `"Code"`, information about the progress of the computation can be printed. An example to demonstrate the interpretation of the verbose output follows:

```
> SetVerbose("Code", true);
> SetSeed(1);
> MinimumWeight(RandomAdditiveCode(GF(4),GF(2),82,39));
GF(2)-Additive code over GF(4) of length 82 with 39 generators.
Is not cyclic
Lower Bound: 1, Upper Bound: 64
Constructed 5 distinct generator matrices
Total Ranks:      21  20  20  20  20
Relative Ranks:  21  20  20  20   1
Starting search for low weight codewords... 0.020
     Discarding non-contributing rank 1 matrix
Enumerating using 1 generator at a time:
     New codeword identified of weight 48, time 0.020
     New codeword identified of weight 46, time 0.020
     New codeword identified of weight 44, time 0.020
     New codeword identified of weight 42, time 0.020
```

```
   Completed Matrix  1:  lower =  5, upper = 42. Time so far: 0.030
     New codeword identified of weight 41, time 0.030
   Completed Matrix  2:  lower =  6, upper = 41. Time so far: 0.030
   Completed Matrix  3:  lower =  7, upper = 41. Time so far: 0.040
     New codeword identified of weight 40, time 0.040
   Completed Matrix  4:  lower =  8, upper = 40. Time so far: 0.040
Enumerating using 2 generators at a time:
     New codeword identified of weight 37, time 0.050
   Completed Matrix  1:  lower =  9, upper = 37. Time so far: 0.050
   Completed Matrix  2:  lower = 10, upper = 37. Time so far: 0.050
   Completed Matrix  3:  lower = 11, upper = 37. Time so far: 0.060
     New codeword identified of weight 36, time 0.060
   Completed Matrix  4:  lower = 12, upper = 36. Time so far: 0.060
Enumerating using 3 generators at a time:
     New codeword identified of weight 34, time 0.070
   Completed Matrix  1:  lower = 13, upper = 34. Time so far: 0.070
     New codeword identified of weight 33, time 0.080
   Completed Matrix  2:  lower = 14, upper = 33. Time so far: 0.090
   Completed Matrix  3:  lower = 15, upper = 33. Time so far: 0.100
   Completed Matrix  4:  lower = 16, upper = 33. Time so far: 0.110
Enumerating using 4 generators at a time:
     New codeword identified of weight 32, time 0.120
   Completed Matrix  1:  lower = 17, upper = 32. Time so far: 0.170
   Completed Matrix  2:  lower = 18, upper = 32. Time so far: 0.250
   Completed Matrix  3:  lower = 19, upper = 32. Time so far: 0.320
   Completed Matrix  4:  lower = 20, upper = 32. Time so far: 0.390
Termination predicted with 7 generators at matrix 4
Enumerating using 5 generators at a time:
   Completed Matrix  1:  lower = 21, upper = 32. Time so far: 0.960
   Completed Matrix  2:  lower = 22, upper = 32. Time so far: 1.570
   Completed Matrix  3:  lower = 23, upper = 32. Time so far: 2.160
   Completed Matrix  4:  lower = 24, upper = 32. Time so far: 2.750
Termination predicted at 118 s (1 m 57 s) with 7 generators at matrix 4
Enumerating using 6 generators at a time:
   Completed Matrix  1:  lower = 25, upper = 32. Time so far: 6.680
   Completed Matrix  2:  lower = 26, upper = 32. Time so far: 10.969
   Completed Matrix  3:  lower = 27, upper = 32. Time so far: 15.149
   Completed Matrix  4:  lower = 28, upper = 32. Time so far: 19.440
Termination predicted at 114 s (1 m 54 s) with 7 generators at matrix 4
Enumerating using 7 generators at a time:
   Completed Matrix  1:  lower = 29, upper = 32. Time so far: 41.739
   Completed Matrix  2:  lower = 30, upper = 32. Time so far: 66.239
   Completed Matrix  3:  lower = 31, upper = 32. Time so far: 90.780
   Completed Matrix  4:  lower = 32, upper = 32. Time so far: 115.510
```

```
Final Results: lower = 32, upper = 32, Total time: 115.510
32
```

Verbose output can be invaluable in the case of lengthy minimum weight calculations.

The algorithm constructs different (equivalent) generator matrices, each of which has pivots in different column positions of the code, called its *information set*. The *relative rank* of a generator matrix is the size of its information set independent of the previously constructed matrices.

When enumerating all generators taken r at a time, once r exceeds the difference between the total rank of a matrix, and its relative rank, the lower bound on the minimum weight will be incremented by 1 for that step.

The upper bound on the minimum weight is determined by the minimum weight of codewords that are enumerated. As soon as these bounds become equal, the computation is complete.

Example H163E8

We illustrate the much greater efficiency of the minimum weight algorithm compared to computing the full weight distribution.

```
> SetVerbose("Code",true);
> C := RandomAdditiveCode(GF(9),GF(3),39,25);
> MinimumWeight(C);
GF(3)-Additive code over GF(9) of length 39 with 25 generators. Is not cyclic
Lower Bound: 1, Upper Bound: 28
Constructed 4 distinct generator matrices
Total Ranks:      13  13  13  13
Relative Ranks:  13  13  12   1
Starting search for low weight codewords... 0.009
     Discarding non-contributing rank 1 matrix
Enumerating using 1 generator at a time:
     New codeword identified of weight 25, time 0.009
     New codeword identified of weight 23, time 0.009
     New codeword identified of weight 21, time 0.009
   Completed Matrix  1:  lower =  3, upper = 21. Time so far: 0.009
   Completed Matrix  2:  lower =  4, upper = 21. Time so far: 0.009
   Completed Matrix  3:  lower =  5, upper = 21. Time so far: 0.009
Enumerating using 2 generators at a time:
     New codeword identified of weight 20, time 0.009
     New codeword identified of weight 19, time 0.009
     New codeword identified of weight 18, time 0.009
   Completed Matrix  1:  lower =  6, upper = 18. Time so far: 0.009
   Completed Matrix  2:  lower =  7, upper = 18. Time so far: 0.019
   Completed Matrix  3:  lower =  8, upper = 18. Time so far: 0.019
Enumerating using 3 generators at a time:
     New codeword identified of weight 17, time 0.070
   Completed Matrix  1:  lower =  9, upper = 17. Time so far: 0.089
   Completed Matrix  2:  lower = 10, upper = 17. Time so far: 0.149
   Completed Matrix  3:  lower = 11, upper = 17. Time so far: 0.210
Enumerating using 4 generators at a time:
   Completed Matrix  1:  lower = 12, upper = 17. Time so far: 1.379
   Completed Matrix  2:  lower = 13, upper = 17. Time so far: 2.539
   Completed Matrix  3:  lower = 14, upper = 17. Time so far: 3.719
Termination predicted at 49 s with 5 generators at matrix 3
Enumerating using 5 generators at a time:
   Completed Matrix  1:  lower = 15, upper = 17. Time so far: 19.409
   Completed Matrix  2:  lower = 16, upper = 17. Time so far: 35.019
   Completed Matrix  3:  lower = 17, upper = 17. Time so far: 50.649
Final Results: lower = 17, upper = 17, Total time: 50.649
17
> time WeightDistribution(C);
[ <0, 1>, <17, 2>, <18, 58>, <19, 496>, <20, 4000>, <21, 29608>, <22, 194760>,
<23, 1146680>, <24, 6126884>, <25, 29400612>, <26, 126624092>, <27, 487889854>,
<28, 1672552654>, <29, 5075315756>, <30, 13534236754>, <31, 31434430104>, <32,
62869109200>, <33, 106686382216>, <34, 150616653852>, <35, 172132748756>, <36,
153007413552>, <37, 99247655566>, <38, 41788710876>, <39, 8571983110> ]
Time: 224142.820
```

163.7.2 The Weight Distribution

```
WeightDistribution(C)
```

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

```
DualWeightDistribution(C)
```

The weight distribution of the code which is dual to the additive code C (see `WeightDistribution`).

163.7.3 The Weight Enumerator

```
WeightEnumerator(C)
```

The (Hamming) weight enumerator $W_C(x, y)$ for the additive code C. The weight enumerator is defined by

$$W_C(x, y) = \sum_{u \in C} x^{n-wt(u)} y^{wt(u)}.$$

```
CompleteWeightEnumerator(C)
```

The complete weight enumerator $\mathcal{W}_C(z_0, \ldots, z_{q-1})$ for the additive code C where q is the size of the alphabet E of C. Let the q elements of E be denoted by $\omega_0, \ldots, \omega_{q-1}$.

If E is a prime field, we let ω_i be i (i.e. take the natural representation of each number). If E is a non-prime field, we let ω_0 be the zero element of E and let ω_i be α^{i-1} for $i = 1 \ldots q-1$ where α is the primitive element of E. Now for a codeword u of C, let $s_i(u)$ be the number of components of u equal to ω_i. The complete weight enumerator is defined by

$$\mathcal{W}_C(z_0, \ldots, z_{q-1}) = \sum_{u \in C} {z_0}^{s_0(u)} \ldots {z_{q-1}}^{s_{q-1}(u)}.$$

```
CompleteWeightEnumerator(C, u)
```

The complete weight enumerator $\mathcal{W}_{C+u}(z_0, \ldots, z_{q-1})$ for the coset $C + u$, where u is an element of the generic vector space for the code C.

163.7.4 The MacWilliams Transform

```
MacWilliamsTransform(n, k, q, W)
```

Let C be a hypothetical $[n, k]$ linear code over a finite field of cardinality q. Let W be the weight distribution of C (in the form as returned by the function `WeightDistribution`). This function applies the MacWilliams transform to W to obtain the weight distribution W' of the dual code of C. The transform is a combinatorial algorithm based on n, k, q and W alone. Thus C itself need not exist—the function simply works with the sequence of integer pairs supplied by the user. Furthermore, if W is not the weight distribution of an actual code, the result W' will be meaningless and even negative weights may be returned.

163.7.5 Words

The functions in this section only apply to codes over finite fields.

`Words(C, w: ` *parameters*`)`

`Cutoff`	RngIntElt	*Default :* ∞
`StoreWords`	BoolElt	*Default :* `true`

Given a linear code C defined over a finite field, return the set of all words of C having weight w. If `Cutoff` is set to a non-negative integer c, then the algorithm will terminate after a total of c words have been found. If `StoreWords` is `true` then the words generated will be stored internally.

```
NumberOfWords(C, w)
```

Given a linear code C defined over a finite field, return the number of words of C having weight w.

```
WordsOfBoundedWeight(C, l, u: parameters)
```

`Cutoff`	RngIntElt	*Default :* ∞
`StoreWords`	BoolElt	*Default :* `true`

Given a linear code C defined over a finite field, return the set of all words of C having weight between l and u, inclusive. If `Cutoff` is set to a non-negative integer c, then the algorithm will terminate after a total of c words have been found. If `StoreWords` is `true` then any words of a *single weight* generated will be stored internally.

163.8 Families of Linear Codes

163.8.1 Cyclic Codes

While cyclic linear codes are always generated by a single generating polynomial (vector), this is not the case for additive codes. Cyclic additive codes may be created in Magma using either a single generator, or a sequence of generators.

In the important case of $GF(2)$-additive vectors over $GF4$, all cyclic codes can be described in terms of two generators with one generator taken over $GF(4)$ and the other over $GF(2)$. A special function is provided for this construction.

```
AdditiveCyclicCode(v)
AdditiveCyclicCode(K, v)
AdditiveCyclicCode(Q)
AdditiveCyclicCode(K, Q)
```

Given either a single vector v or sequence of vectors Q over some finite field F, return the K-additive code over F generated by all shifts of the inputs. The field K must be a subfield of F and if it is the prime subfield of F, it may be omitted.

```
AdditiveCyclicCode(n, f)
AdditiveCyclicCode(K, n, f)
AdditiveCyclicCode(n, Q)
AdditiveCyclicCode(K, n, Q)
```

Given either a single polynomial f or sequence Q of polynomials over some finite field F, return the K-additive code of length n over F generated by all shifts of the inputs. The field K must be a subfield of F, and if it is the prime subfield of F, it may be omitted.

```
AdditiveCyclicCode(v4, v2)
```

Given two vectors of equal length n, where v_4 is over $GF(4)$ and v_2 is over $GF(2)$, return the F_2–additive code generated by all of their cyclic shifts. Note that for the case of $GF(2)$-additive codes over $GF(4)$, two generators suffice to generate any such code.

```
AdditiveCyclicCode(n, f4, f2)
```

Given two polynomials f_4 and f_2, where f_4 is over $GF(4)$ and f_2 is over $GF(2)$, return the F_2–additive code of length n generated by all of their cyclic shifts. The degree of the polynomials f_4 and f_2 must not exceed $n-1$. Note that for the case of $GF(2)$-additive codes over $GF(4)$, two generators suffice to generate any such code.

163.8.2 Quasicyclic Codes

Quasicyclic codes are a generalisation of cyclic codes. In Magma quasicyclic codes consist of horizontally joined cyclic blocks.

```
AdditiveQuasiCyclicCode(n, Q)
```

```
AdditiveQuasiCyclicCode(K, n, Q)
```

Given an integer n, and a sequence Q of polynomials over some finite field F, return the K–additive quasicyclic code, whose cyclic blocks are generated by the polynomials in Q. The field K must be a subfield of F, and if it is the prime subfield of F, it may be omitted.

```
AdditiveQuasiCyclicCode(n, Q, h)
```

```
AdditiveQuasiCyclicCode(K, n, Q, h)
```

Given an integer n, and a sequence Q of polynomials over some finite field F, and an integer h, then return the K–additive quasicyclic code, whose cyclic blocks are generated by the polynomials in Q and stacked 2-dimensionally of height h. The field K must be a subfield of F, and if it is the prime subfield of F, it may be omitted.

```
AdditiveQuasiCyclicCode(Q)
```

```
AdditiveQuasiCyclicCode(K, Q)
```

Given a sequence Q of vectors over some finite field F, then return the K–additive quasicyclic code, whose cyclic blocks are generated by the vectors in Q. The field K must be a subfield of F, and if it is the prime subfield of F, it may be omitted.

```
AdditiveQuasiCyclicCode(Q, h)
```

```
AdditiveQuasiCyclicCode(K, Q, h)
```

Given a sequence Q of vectors over some finite field F, and an integer h, then return the K–additive quasicyclic code, whose cyclic blocks are generated by the vectors in Q and stacked 2-dimensionally of height h. The field K must be a subfield of F, and if it is the prime subfield of F, it may be omitted.

163.9 New Codes from Old

The operations described here produce a new code by modifying in some way the codewords of a given code.

163.9.1 Standard Constructions

```
AugmentCode(C)
```

Construct a new additive code by including the all-ones vector with the words of the additive code C.

```
CodeComplement(C, S)
```

Given a subcode S of the code C, return a code C' such that $C = S + C'$. Both C and S must be defined over the same field.

```
DirectSum(C, D)
```

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

```
DirectSum(Q)
```

Given a sequence of codes $Q = [C_1, \ldots, C_r]$, all defined over the same field F, construct the direct sum of the C_i.

```
DirectProduct(C, D)
```

Given an $[n_1, k_1]$ code C and an $[n_2, k_2]$ code D, both over the same ring R, construct the direct product of C and D. The direct product has length $n_1 {\cdot} n_2$ and its generator matrix is the Kronecker product of the basis matrices of C and D.

```
ExtendCode(C)
```

Given an $[n, k, d]$ additive code C, form a new code C' from C by adding the appropriate extra coordinate to each vector of C such that the sum of the coordinates of the extended vector is zero.

```
ExtendCode(C, n)
```

Return the code obtained by extending the code C extended n times.

```
PadCode(C, n)
```

Add n zeros to the end of each codeword of the code C.

```
PlotkinSum(C1, C2)
```

Given codes C_1 and C_2 defined over the same alphabet, return the code consisting of all vectors of the form $u|u + v$, where $u \in C1$ and $v \in C2$. Zeros are appended where needed to make up any length differences in the two codes.

```
PlotkinSum(C1, C2, C3: parameters)
```

a FldFinElt *Default :* -1

Given three codes C_1, C_2 and C_3 defined over the same alphabet K, return the code consisting of all vectors of the form $u|u+a*v|u+v+w$, where $u \in C1$, $v \in C2$ and $w \in C3$. The default value of the multiplier a is a primitive element of K. Zeros are appended where needed to ensure that every codeword has the same length.

```
PunctureCode(C, i)
```

Given an $[n,k]$ code C, and an integer i, $1 \le i \le n$, construct a new code C' by deleting the i-th coordinate from each code word of C.

```
PunctureCode(C, S)
```

Given an $[n,k]$ code C and a set S of distinct integers $\{i_1, \cdots, \mathrm{i}_r\}$ each of which lies in the range $[1,n]$, construct a new code C' by deleting the components $i_1, \cdots, i_r$ from each code word of C.

```
ShortenCode(C, i)
```

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

```
ShortenCode(C, S)
```

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

163.9.2 Combining Codes

```
C1 cat C2
```

Given codes $C1$ and $C2$, both defined over the same field K, return the concatenation C of $C1$ and $C2$. The generators of the resultant code are the concatenations of the generators of $C1$ and $C2$.

```
Juxtaposition(C1, C2)
```

Given an $[n_1,k,d_1]$ code $C1$ and an $[n_2,k,d_2]$ code $C2$ of the same dimension, where both codes are defined over the same field K, this function returns a $[n_1+n_2,k,\ge d_1+d_2]$ code whose generator matrix is `HorizontalJoin(A, B)`, where A and B are the generator matrices for codes $C1$ and $C2$, respectively.

163.10 Automorphism Group

```
AutomorphismGroup(C)
```

The automorphism group of the additive code C. Currently, this function is only available for additive codes over $GF(4)$.

```
PermutationGroup(C)
```

The subgroup of the automorphism group of the additive code C consisting of permutations of the coordinates. Currently, this function is only available for additive codes over $GF(4)$.

164 QUANTUM CODES

Chapter 164
QUANTUM CODES

164.1 Introduction

Interest in quantum computing has grown rapidly following the discovery by Peter Shor in 1994 of a polynomial-time algorithm for integer factorization [Sho94]. In a classical computer a sequence of N binary digits defines one specific configuration among the 2^N possible values. However, in a quantum computer a collection of N "qubits" has a state function (in 'ket' notation) $|\psi\rangle$ in a *Hilbert space*, which can be in a superposition of all 2^N possible values

$$|\psi\rangle = \sum_{\mathbf{v}\in\mathbf{Z}_2^N} \alpha_{\mathbf{v}}|\mathbf{v}\rangle, \qquad \alpha_{\mathbf{v}} \in \mathbf{C}, \quad \sum_{\mathbf{v}} |\alpha_{\mathbf{v}}|^2 = 1.$$

A basic theorem in quantum information theory states that it is impossible to clone a quantum state. Since this implies that it is not possible to copy quantum information, it was initially believed that error-correction would be impossible on a quantum computer. However, in 1995 Shor showed that it *was* possible to encode quantum information in such a way that errors can be corrected, assuming an error model in which errors occur independently in distinct qubits [Sho95].

Following this discovery, a class of quantum error-correcting codes known as *stabilizer codes* were developed. In [CRSS98] (which is the major reference for this chapter of the Magma Handbook), it was shown that stabilizer codes can be represented in terms of additive codes over finite fields (see chapter 163 for a description of additive codes). This remarkable result reduces the problem of constructing fault-tolerant encodings on a continuous Hilbert space to that of constructing certain discrete codes, allowing the use of many of the tools developed in classical coding theory.

The current Magma package for quantum codes deals exclusively with finite field representations of stabilizer codes. It is important to keep in mind that, although a quantum code is *represented* by a code over a finite field, an actual quantum code is in fact a totally different object. The full theory behind quantum stabilizer codes will not be described here, for that the reader should consult the main reference [CRSS98]. A brief synopsis will outline how the finite field representation of a stabilizer code is to be interpreted, and the specifics of this representation in Magma.

Many of the conventions and functions for classical error-correcting code types in Magma can be ambiguous in the context of quantum codes. For this reason the handbook should be carefully consulted before assuming that any particular aspect of a quantum code follows naturally from classical coding theory definitions.

The reduction of the problem of continuous errors on a Hilbert space to a problem employing a discrete finite field representation is achieved by confining attention to a finite *error group*. An element of the error group, acting on the N qubits, is expressed as a combination of bit flip errors, designated by the operator X, and phase shift errors, designated by the operator Z (as well as an overall phase factor that will be ignored here):

$$X(\mathbf{a})Z(\mathbf{b})|\mathbf{v}\rangle = (-1)^{\mathbf{v}\cdot\mathbf{b}}|\mathbf{v}+\mathbf{a}\rangle$$

The error group is given by the set $\{X(\mathbf{a})Z(\mathbf{b}) : \mathbf{a},\mathbf{b}\in\mathbf{Z}_2^n\}$ and its elements can be written as length $2N$ binary vectors $(\mathbf{a}|\mathbf{b})$. An error represented by such a vector in Magma is said to be in *extended format* which is distinct from the default representation. A more common (and practical) representation is as the element $\mathbf{w}$ of $F_4^{(N)}$ given by $\mathbf{w}=\mathbf{a}+\omega\mathbf{b}$, where ω is a primitive element of $GF(4)$. This representation is referred to as the *compact format*, and is the default format used in Magma for quantum codes. Note that this is slightly different to the representation $\widehat{\mathbf{w}}=\omega\mathbf{a}+\overline{\omega}\mathbf{b}$ used in [CRSS98] for binary quantum codes, but they are equivalent: $\mathbf{w}=\overline{\omega}*\widehat{\mathbf{w}}$.

The Magma package also supports non-binary quantum codes, which are obtained by generalizing from the binary case in a natural way. For quantum codes based on qubits over $GF(q)$, the compact format in $GF(q^2)$ will be $\mathbf{w}=\mathbf{a}+\lambda\mathbf{b}$, where λ is a fixed element returned by the function `QuantumBasisElement`$(GF(q^2))$.

A symplectic inner product is defined on the group of errors, in its representation as a set of $GF(q)$-vectors. For vectors in extended format this is defined by

$$(\mathbf{a}_1|\mathbf{b}_1)*(\mathbf{a}_2|\mathbf{b}_2)=\mathbf{a}_1\cdot\mathbf{b}_2-\mathbf{a}_2\cdot\mathbf{b}_1$$

In compact format (over $GF(4)$) the equivalent inner product is defined by

$$\mathbf{w}_1*\mathbf{w}_2=\mathrm{Trace}(\mathbf{w}_1\cdot\overline{\mathbf{w}}_2).$$

Since the commutator of two errors is given by

$$\begin{aligned}&\Big[\big(X(\mathbf{a}_1)Z(\mathbf{b}_1)\big)\big(X(\mathbf{a}_2)Z(\mathbf{b}_2)\big)-\big(X(\mathbf{a}_2)Z(\mathbf{b}_2)\big)\big(X(\mathbf{a}_1)Z(\mathbf{b}_1)\big)\Big]|\mathbf{v}\rangle\\ &\qquad=(-1)^{\mathbf{v}\cdot\mathbf{b}_2+(\mathbf{v}+\mathbf{a}_2)\cdot\mathbf{b}_1}|\mathbf{v}+\mathbf{a}_1+\mathbf{a}_2\rangle+(-1)^{\mathbf{v}\cdot\mathbf{b}_1+(\mathbf{v}+\mathbf{a}_1)\cdot\mathbf{b}_2}|\mathbf{v}+\mathbf{a}_1+\mathbf{a}_2\rangle\\ &\qquad=\Big[(-1)^{\mathbf{a}_2\cdot\mathbf{b}_1}-(-1)^{\mathbf{a}_1\cdot\mathbf{b}_2}\Big](-1)^{\mathbf{v}\cdot(\mathbf{b}_1-\mathbf{b}_2)}|\mathbf{v}+\mathbf{a}_1+\mathbf{a}_2\rangle\\ &\qquad=\Big[1-\delta_{\mathbf{a}_1\cdot\mathbf{b}_2,\,\mathbf{a}_2\cdot\mathbf{b}_1}\Big]\cdots\end{aligned}$$

then clearly errors will commute if and only if their finite field representations are orthogonal with respect to the symplectic inner product.

A quantum stabilizer code is defined by an abelian subgroup of the error group. In the context of its finite field representation this translates to a self-orthogonal additive code under the symplectic inner product. So a quantum stabilizer code Q is defined by a symplectic self-orthogonal additive code S, which is (with some redundancy) termed the stabilizer code of Q.

The error-correcting capability of a code is determined by the set of errors which can not be detected. For classical linear codes these undetectable errors are precisely the non-zero codewords of the code, while for a quantum code, the undetectable errors are given by the set $S^{\perp}\backslash S$, where $S^{\perp}$ is the symplectic dual of S.

The most important measure of the ability of a quantum code to correct errors is its *minimum weight*, that is, is the minimum of the weights of the words of $S^{\perp}\backslash S$. An exception to this definition occurs in the case of quantum codes having dimension zero, which are defined by symplectic self-dual stabilizer codes. These are termed "self-dual quantum codes" and are defined to have a minimum weight equal to the (classical) minimum weight of their stabilizer code.

164.2 Constructing Quantum Codes

A quantum code of length n over $GF(q)$ is defined in terms of a symplectic self-orthogonal stabilizer code, which is given either as a length n additive code over $GF(q^2)$ (compact format) or as a length $2n$ additive code over $GF(q)$ (extended format). If Q is a quantum code with generator matrix G_1 in compact format, and generator matrix $G_2 = (A|B)$ in extended format, then

$$G_1 = A + \lambda B,$$

where λ is a fixed element returned by the function `QuantumBasisElement`($GF(q^2)$). By default MAGMA assumes the compact format. However, the extended format can be flagged by setting the variable argument `ExtendedFormat` to `true`.

An $[n, k]$ symplectic self-orthogonal linear code over $GF(q^2)$ will generate an $[[n, n/2 - k]]$ quantum stabilizer code. A (compact format) additive symplectic self-orthogonal code C over $GF(q^2)$ will give a quantum code of the same length and "dimension" $\log_q(N)$, where N is the number of code words in C.

164.2.1 Construction of General Quantum Codes

```
QuantumCode(S)
```

`ExtendedFormat` BOOLELT *Default :* `false`

Given an additive code S which is self-orthogonal with respect to the symplectic inner product, return the quantum code defined by S. By default, S is interpreted as being in compact format, that is, a length n additive code over $GF(q^2)$. If `ExtendedFormat` is set *true*, then S is interpreted as being in extended format, that is, a length $2n$ additive code over $GF(q)$.

Example H164E1

A linear code over $GF(4)$ that is *even* is symplectic self-orthogonal. Note that when a quantum code is printed in MAGMA, an *additive* stabilizer matrix over $GF(q^2)$ is displayed.

```
> F<w> := GF(4);
> M := Matrix(F, 2, 6, [1,0,0,1,w,w, 0,1,0,w,w,1]);
> C := LinearCode(M);
> C;
[6, 2, 4] Linear Code over GF(2^2)
Generator matrix:
[  1   0   0   1   w   w]
[  0   1   0   w   w   1]
> IsEven(C);
true
> IsSymplecticSelfOrthogonal(C);
true
> Q := QuantumCode(C);
> Q;
[[6, 2]] Quantum code over GF(2^2), stabilized by:
[  1   0   0   1   w   w]
[  w   0   0   w w^2 w^2]
[  0   1   0   w   w   1]
[  0   w   0 w^2 w^2   w]
> MinimumWeight(Q);
1
> Q;
[[6, 2, 1]] Quantum code over GF(2^2), stabilized by:
[  1   0   0   1   w   w]
[  w   0   0   w w^2 w^2]
[  0   1   0   w   w   1]
[  0   w   0 w^2 w^2   w]
```

Example H164E2

Any stabilizer code used to construct a quantum code, may be expressed in either compact or extended format. The length 6 quaternary additive code S in the previous example (H164E1) is equivalent to a length 12 binary additive code in extended format.
Note that the code will still be displayed in compact format.

```
> F<w> := GF(4);
> C := LinearCode<GF(2), 12 |
>      [ 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1 ],
>      [ 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0 ],
>      [ 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0 ],
>      [ 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1 ] >;
> C;
[12, 4, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 1 1 1 0 0 1 1]
[0 1 0 0 0 1 0 0 0 1 1 0]
[0 0 0 1 0 1 1 1 0 0 0 0]
[0 0 0 0 1 1 1 0 0 1 1 1]
> Q := QuantumCode(C : ExtendedFormat := true);
> Q;
```

```
[[6, 2]] Quantum code over GF(2^2), stabilized by:
[  1   0   0   1   w   w]
[  w   0   0   w w^2 w^2]
[  0   1   0   w   w   1]
[  0   w   0 w^2 w^2   w]
```

Example H164E3

Any self-orthogonal code which has a rate of 1/2 must be self-dual, and gives rise to a dimension zero quantum code (which is also termed self-dual). In this example we construct the hexacode, which is the unique extremal length 6 self-dual quantum code of minimum weight 4.

```
> F<w> := GF(4);
> M := Matrix(F, 3, 6, [0,0,1,1,1,1, 0,1,0,1,w,w^2, 1,0,0,1,w^2,w]);
> C := LinearCode(M);
> C;
[6, 3, 4] Linear Code over GF(2^2)
Generator matrix:
[  1   0   0   1 w^2   w]
[  0   1   0   1   w w^2]
[  0   0   1   1   1   1]
> IsSymplecticSelfOrthogonal(C);
true
> Q := QuantumCode(C);
> MinimumWeight(Q);
4
> Q;
[[6, 0, 4]] self-dual Quantum code over GF(2^2), stabilized by:
[  1   0   0   1 w^2   w]
[  w   0   0   w   1 w^2]
[  0   1   0   1   w w^2]
[  0   w   0   w w^2   1]
[  0   0   1   1   1   1]
[  0   0   w   w   w   w]
```

Example H164E4

Stabilizer codes neither have to be linear nor even and indeed any additive code which is symplectic self-orthogonal will generate a quantum code. The following code was randomly generated.

```
> F<w> := GF(4);
> M := Matrix(F, 3, 7, [1,w,w,w,0,0,1, w,0,1,0,w^2,0,1, 0,w^2,w,w^2,w,0,0]);
> C := AdditiveCode(GF(2),M);
> C;
[7, 1 1/2 : 3, 4] GF(2)-Additive Code over GF(2^2)
Generator matrix:
[  1   w   w   w   0   0   1]
[  w   0   1   0 w^2   0   1]
[  0 w^2   w w^2   w   0   0]
> IsSymplecticSelfOrthogonal(C);
true
```

The code C can be shown to be neither linear nor even: in fact it has the same number of even and odd codewords.

```
> IsLinear(C);
false
> {* Weight(v) mod 2 : v in C *};
{* 0^^4, 1^^4 *}
>
> Q := QuantumCode(C);
> MinimumWeight(Q);
1
> Q;
[[7, 4, 1]] Quantum code over GF(2^2), stabilized by:
[  1   w   w   w   0   0   1]
[  w   0   1   0 w^2   0   1]
[  0 w^2   w w^2   w   0   0]
```

QuantumCode(M)

ExtendedFormat BoolElt *Default :* false

Given a matrix M over $GF(q^2)$ for which the $GF(q)$ additive span of its rows is a self-orthogonal code S with respect to the symplectic inner product, return the quantum code defined by S. By default, M is interpreted as being in compact format, that is, a matrix whose rows are length n vectors over $GF(q^2)$. However, if ExtendedFormat is set true, then M will be interpreted as being in extended format, that is, a matrix whose rows are length $2n$ vectors over $GF(q)$.

Example H164E5

A quantum code can be constructed directly from an additive stabilizer matrix, thereby avoiding creation of the stabilizer code. The quantum code given in example H164E4 could have also been constructed as follows:

```
> F<w> := GF(4);
> M := Matrix(F, 3, 7, [1,w,w,w,0,0,1, w,0,1,0,w^2,0,1, 0,w^2,w,w^2,w,0,0]);
> Q := QuantumCode(M);
> Q;
[[7, 4]] Quantum code over GF(2^2), stabilized by:
[  1   w   w   w   0   0   1]
[  w   0   1   0 w^2   0   1]
[  0 w^2   w w^2   w   0   0]
```

```
QuantumCode(G)
```

Given a graph G, return the self-dual (dimension 0) quantum code defined by the adjacency matrix of G.

Example H164E6

The unique extremal $[[6,0,4]]$ hexacode can be defined in terms of a graph representing a 5–spoked wheel. The graph is specified by listing the edges comprising its circumference, followed by the spokes radiating out from the center.

```
> G := Graph<6 | {1,2},{2,3},{3,4},{4,5},{5,1}, <6, {1,2,3,4,5}> >;
> Q := QuantumCode(G);
> Q:Minimal;
[[6, 0]] self-dual Quantum code over GF(2^2)
> MinimumWeight(Q);
4
> Q:Minimal;
[[6, 0, 4]] self-dual Quantum code over GF(2^2)
```

Example H164E7

The unique extremal $[[12,0,6]]$ dodecacode can also be described by a graph with a nice mathematical structure. The graph construction is derived from the diagram given by Danielson in [Dan05]. In order to employ modular arithmetic, the graph vertices are numbered from 0 to 11.

```
> S := {@ i : i in [0 .. 11] @};
> G := Graph<S |
>     { {4*k+i,4*k+i+2}                         : i in [0..1], k in [0..2] },
>     { {4*k+i,4*k+(i+1) mod 4}                 : i in [0..3], k in [0..2] },
>     { {4*k+i,4*((k+1) mod 3)+(i+1) mod 4} : i in [0..3], k in [0..2] } >;
> Q := QuantumCode(G);
> MinimumWeight(Q);
6
> Q:Minimal;
[[12, 0, 6]] self-dual Quantum code over GF(2^2)
```

```
RandomQuantumCode(F, n, k)
```

Let F be a degree 2 extension of a finite field $GF(q)$. Given positive integers n and k such that $n \geq k$, this function returns a random $[[n,k]]$ quantum stabilizer code over F. The field F is assumed to be given in compact format.

Example H164E8

We construct a random $[[10,6]]$ quantum code over $GF(4)$.

```
> F<w> := GF(4);
> Q := RandomQuantumCode(F, 10, 6);
> Q;
[[10, 6]] Quantum code over GF(2^2), stabilized by:
[  w   0   0   w   1   1   w w^2   w w^2]
[  0   1   0   w   1 w^2 w^2   1   w   w]
[  0   w   1   1   1   0 w^2   0   0   0]
[  0   0   w   1   1   0   1 w^2   1   w]
```

```
Subcode(Q, k)
```

Given a quantum code Q of dimension $k_Q \geq k$ then return a subcode of Q of dimension k.

164.2.2 Construction of Special Quantum Codes

```
Hexacode()
```

Return the $[[6,0,4]]$ self-dual quantum hexacode.

```
Dodecacode()
```

Return the $[[12,0,6]]$ self-dual quantum dodecacode.

164.2.3 CSS Codes

```
CSSCode(C1, C2)
CalderbankShorSteaneCode(C1, C2)
```

Given two classical linear binary codes C_1 and C_2 of length n such that C_2 is a subcode of C_1, form a quantum code using the construction of Calderbank, Shor and Steane [CS96, Ste96a, Ste96b].

Example H164E9

Let C_1 denote the $[7,4,3]$ Hamming code and C_2 denote its dual. Observing that C_1 contains C_2, we may apply the CSS construction using C_1 and C_2 to obtain a $[[7,1,3]]$ code.

```
> F<w> := GF(4);
> C1 := HammingCode(GF(2), 3);
> C1;
[7, 4, 3] "Hamming code (r = 3)" Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 1 0]
[0 1 0 0 0 1 1]
[0 0 1 0 1 1 1]
[0 0 0 1 1 0 1]
> C2 := Dual(C1);
> C2;
[7, 3, 4] Cyclic Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0 1 1]
[0 1 0 1 1 1 0]
[0 0 1 0 1 1 1]
> C2 subset C1;
true
> Q := CSSCode(C1, C2);
> MinimumWeight(Q);
3
> Q;
[[7, 1, 3]] CSS Quantum code over GF(2^2), stabilised by:
[  1   0   0   1   0   1   1]
[  w   0   0   w   0   w   w]
[  0   1   0   1   1   1   0]
[  0   w   0   w   w   w   0]
[  0   0   1   0   1   1   1]
[  0   0   w   0   w   w   w]
```

164.2.4 Cyclic Quantum Codes

Cyclic quantum codes are those having cyclic stabilizer codes. Conditions are listed in [CRSS98] for generating polynomials which give rise to symplectic self-orthogonal stabilizer codes.

`QuantumCyclicCode(v)`

`QuantumCyclicCode(Q)`

`LinearSpan` BoolElt *Default :* `false`

Given either a single vector v or sequence of vectors Q defined over a finite field F, return the quantum code generated by the span of the cyclic shifts of the supplied vectors. The span must be self-orthogonal with respect to the symplectic inner product. By default, the additive span is taken over the prime field, but if the variable argument `LinearSpan` is set to `true`, then the linear span will be taken.

Example H164E10

A large number of good cyclic quantum codes exist. For example, the best known binary self-dual quantum code of length 15 is cyclic.

```
> F<w> := GF(4);
> v := VectorSpace(F, 15) ! [w,1,1,0,1,0,1,0,0,1,0,1,0,1,1];
> Q := QuantumCyclicCode(v);
> MinimumWeight(Q);
6
> Q:Minimal;
[[15, 0, 6]] self-dual Quantum code over GF(2^2)
```

```
QuantumCyclicCode(n, f)
```

```
QuantumCyclicCode(n, Q)
```

`LinearSpan` BoolElt *Default :* `false`

Let n be a positive integer. Given either a single polynomial f or a sequence of polynomials Q over some finite field F, return the quantum code of length n generated by the additive span of their cyclic shifts. The additive span must be symplectic self-orthogonal. By default, the additive span is taken over the prime field, but if the variable argument `LinearSpan` is set to `true`, then the linear span will be taken.

Example H164E11

Since classical cyclic codes correspond to factors of cyclotomic polynomials it is frequently convenient to specify a cyclic code in terms of polynomials. Here we construct the best known binary quantum codes with parameters $[[23, 12, 4]]$ and $[[25, 0, 8]]$.

```
> F<w> := GF(4);
> P<x> := PolynomialRing(F);
> f := x^16 + x^15 + x^13 + w*x^12 + x^11 + w*x^10 + x^9 + x^8 + w^2*x^7 +
>      x^6 + x^5 + w*x^4 + w^2*x^3 + w*x^2 + w^2*x + w^2;
> Q := QuantumCyclicCode(23, f);
> MinimumWeight(Q);
4
> Q:Minimal;
[[23, 12, 4]] Quantum code over GF(2^2)
>
> f := x^12 + x^11 + x^10 + x^8 + w*x^6 + x^4 + x^2 + x + 1;
> Q := QuantumCyclicCode(25, f);
> MinimumWeight(Q);
```

```
8
> Q:Minimal;
[[25, 0, 8]] self-dual Quantum code over GF(2^2)
```

```
QuantumCyclicCode(v4, v2)
```

In the important case of $GF(2)$-additive codes over $GF(4)$, any cyclic code can be specified by two generators. Given vectors $v4$ and $v2$ both of length n, where v_4 is over $GF(4)$ and v_2 is over $GF(2)$, this function returns the length n quantum code generated by the additive span of their cyclic shifts. This span must be self-orthogonal with respect to the symplectic inner product.

Example H164E12

Any cyclic binary quantum code of length n is determined by a cyclic stabilizer code, which can be defined uniquely in terms of an n-dimensional vector over $GF(4)$ together with an n-dimensional vector over $GF(2)$. We construct the best known $[[21, 0, 8]]$ and $[[21, 5, 6]]$ cyclic binary quantum codes.

```
> F<w> := GF(4);
> v4 := RSpace(F, 21) ! [w^2,w^2,1,w,0,0,1,1,1,1,0,1,0,1,1,0,1,1,0,0,0];
> v2 := RSpace(GF(2),21) ! [1,0,1,1,1,0,0,1,0,1,1,1,0,0,1,0,1,1,1,0,0];
> Q := QuantumCyclicCode(v4,v2);
> MinimumWeight(Q);
8
> Q:Minimal;
[[21, 0, 8]] self-dual Quantum code over GF(2^2)
>
> v4 := RSpace(F, 21) ! [w,0,w^2,w^2,w,w^2,w^2,0,w,1,0,0,1,0,0,0,0,1,0,0,1];
> v2 := RSpace(GF(2), 21) ! [1,0,1,1,1,0,0,1,0,1,1,1,0,0,1,0,1,1,1,0,0];
> Q := QuantumCyclicCode(v4,v2);
> MinimumWeight(Q);
6
> Q:Minimal;
[[21, 5, 6]] Quantum code over GF(2^2)
```

164.2.5 Quasi-Cyclic Quantum Codes

Quasi-cyclic quantum codes are those having quasi-cyclic stabilizer codes.

```
QuantumQuasiCyclicCode(n, Q)
```

`LinearSpan` BoolElt *Default :* `false`

Given an integer n, and a sequence Q of polynomials over some finite field F, let S be the quasi-cyclic classical code generated by the span of the set of vectors formed by concatenating cyclic blocks generated by the polynomials in Q. Assuming that S is self-orthogonal with respect to the symplectic inner product, this function returns the quasi-cyclic quantum code with stabiliser code S. If the span of the vectors is not symplectic self-orthogonal, an error will be flagged.

By default the additive span is taken over the prime field, but if the variable argument `LinearSpan` is set to `true`, then the linear span will be taken.

```
QuantumQuasiCyclicCode(Q)
```

`LinearSpan` BoolElt *Default :* `false`

Given a sequence Q of vectors, return the quantum code whose additive stabilizer matrix is constructed from the length n cyclic blocks generated by the cyclic shifts of the vectors in Q. If the variable argument `LinearSpan` is set to `true`, then the linear span of the shifts will be used, else the additive span will be used (default).

Example H164E13

Most quasi-cyclic quantum codes currently known are linear, since this is where most research on quasi-cyclic codes has been focused. In this example we construct the best known quasi-cyclic binary quantum codes with parameters $[[14, 0, 6]]$ and $[[18, 6, 5]]$.

```
> F<w> := GF(4);
> V7 := VectorSpace(F, 7);
> v1 := V7 ! [1,0,0,0,0,0,0];
> v2 := V7 ! [w^2,1,w^2,w,0,0,w];
> Q := QuantumQuasiCyclicCode([v1, v2] : LinearSpan := true);
> MinimumWeight(Q);
6
> Q:Minimal;
[[14, 0, 6]] self-dual Quantum code over GF(2^2)
>
> V6 := VectorSpace(F, 6);
> v1 := V6 ! [1,1,0,0,0,0];
> v2 := V6 ! [1,0,1,w^2,0,0];
> v3 := V6 ! [1,1,w,1,w,0];
> Q := QuantumQuasiCyclicCode([v1, v2, v3] : LinearSpan := true);
> MinimumWeight(Q);
5
> Q:Minimal;
[[18, 6, 5]] Quantum code over GF(2^2)
```

164.3 Access Functions

```
QuantumBasisElement(F)
```

Given a degree 2 extension field $F = GF(q^2)$, return the element $\lambda \in F$ which acts to connect the extended and compact formats. For a vector $(\mathbf{a}|\mathbf{b})$ in extended format, the corresponding compact format of this vector will be $\mathbf{w} = \mathbf{a} + \lambda\mathbf{b}$.

```
StabilizerCode(Q)
StabiliserCode(Q)
```

`ExtendedFormat` BoolElt *Default :* `false`

The additive stabiliser code S which defines the quantum code Q. By default S is returned in the compact format of a length n code over $GF(q^2)$, but if `ExtendedFormat` is set to `true`, then it will be returned in extended format as a length $2n$ code over $GF(q)$.

```
StabilizerMatrix(Q)
StabiliserMatrix(Q)
```

`ExtendedFormat` BoolElt *Default :* `false`

Given a quantum code Q return the additive stabiliser matrix M defining Q. By default M is returned in the compact format of a length n code over $GF(q^2)$, but if `ExtendedFormat` is set to `true`, then it will be returned in the extended format as a length $2n$ code over $GF(q)$.

```
NormalizerCode(Q)
NormaliserCode(Q)
```

`ExtendedFormat` BoolElt *Default :* `false`

The additive normalizer code N which defines the quantum code Q. By default N is returned in the compact format of a length n code over $GF(q^2)$, but if `ExtendedFormat` is set to `true`, then it will be returned in extended format as a length $2n$ code over $GF(q)$.

```
NormalizerMatrix(Q)
NormaliserMatrix(Q)
```

`ExtendedFormat` BoolElt *Default :* `false`

Given a quantum code Q return the additive normalizer matrix M defining Q. By default M is returned in the compact format of a length n code over $GF(q^2)$, but if `ExtendedFormat` is set to `true`, then it will be returned in the extended format as a length $2n$ code over $GF(q)$.

164.3.1 Quantum Error Group

As described in the introduction to this chapter, vectors over a finite field used to describe a quantum stabilizer code actually represent elements of the corresponding quantum error group. For a p-ary N qubit system (where p is prime) this error group is the extra-special group with order 2^{2N+1} consisting of combinations of N bit-flip errors, N phase flip errors, and an overall phase shift. All groups in this section use a polycyclic group representation.

`QuantumErrorGroup(p, n)`

Return the abelian group representing all possible errors for a length n p-ary qubit system, which is an extra-special group of order p^{2n+1} with $2n+1$ generators. The generators correspond to the qubit-flip operators $X(i)$, the phase-flip operators $Z(i)$, and an overall phase multiplication W by the p-th root of unity. The generators appear in the order $X(1), Z(1), \ldots, X(n), Z(n), W$.

`QuantumBinaryErrorGroup(n)`

Return the abelian group representing all possible errors on a length n binary qubit system, which is an extra special group of order 2^{2n-1}.

Example H164E14

The image of a vector in the error group is easily obtained from its extended format representation. We illustrate the connection between symplectic orthogonality as a vector, and commutativity as an element of the error group.

```
> n := 5;
> VSn  := VectorSpace(GF(2), n);
> VS2n := VectorSpace(GF(2), 2*n);
> E := QuantumBinaryErrorGroup(n);
> BitFlips   := [E.i : i in [1..2*n] | IsOdd(i)  ];
> PhaseFlips := [E.i : i in [1..2*n] | IsEven(i) ];
```

We first take two vectors which are not orthogonal and show their images in the error group do not commute.

```
> v1a := VSn ! [0,1,1,0,1]; v1b := VSn ! [0,1,1,0,1];
> v1  := VS2n ! HorizontalJoin(v1a, v1b);
> v2a := VSn ! [1,0,1,1,0]; v2b := VSn ! [0,1,0,1,1];
> v2  := VS2n ! HorizontalJoin(v2a, v2b);
> SymplecticInnerProduct(v1,v2 : ExtendedFormat := true);
1
>
> e1 := &*[ BitFlips[i]   : i in Support(v1a) ] *
>       &*[ PhaseFlips[i] : i in Support(v1b) ];
> e2 := &*[ BitFlips[i]   : i in Support(v2a) ] *
>       &*[ PhaseFlips[i] : i in Support(v2b) ];
> e1*e2 eq e2*e1;
```

```
false
```

Next a pair of orthogonal vectors is shown to commute.

```
> v1a := VSn ! [1,1,0,1,0]; v1b := VSn ! [0,0,1,1,0];
> v1  := VS2n ! HorizontalJoin(v1a, v1b);
> v2a := VSn ! [0,1,1,1,0]; v2b := VSn ! [0,1,1,1,0];
> v2  := VS2n ! HorizontalJoin(v2a, v2b);
> SymplecticInnerProduct(v1,v2 : ExtendedFormat := true);
0
>
> e1 := &*[ BitFlips[i]   : i in Support(v1a) ] *
>       &*[ PhaseFlips[i] : i in Support(v1b) ];
> e2 := &*[ BitFlips[i]   : i in Support(v2a) ] *
>       &*[ PhaseFlips[i] : i in Support(v2b) ];
> e1*e2 eq e2*e1;
true
```

`QuantumErrorGroup(Q)`

For a quantum code Q of length n, return the group of all errors on n qubits. This is the full error group, the ambient space containing all possible errors.

`StabilizerGroup(Q)`

`StabiliserGroup(Q)`

Return the abelian group of errors that defines the quantum code Q, which is a subgroup of the group returned by `QuantumErrorGroup(Q)`.

`StabilizerGroup(Q, G)`

`StabiliserGroup(Q, G)`

Given a quantum code Q with error group G (an extra-special group), return the abelian group of errors of Q as a subgroup of G.

Example H164E15

The stabilizer group of any quantum stabilizer code over $GF(4)$ will be abelian.

```
> F<w> := GF(4);
> Q := RandomQuantumCode(F, 10, 6);
> G := StabilizerGroup(Q);
> IsAbelian(G);
true
```

Example H164E16

In order to make stabilizer groups from distinct codes compatible with one another, the groups must be created within the same super-structure. This is done by first creating a copy of the full error group, and then generating each instance of a stabilizer group as a subgroup.
In this example, the intersection of the stabilizer groups of two random codes is formed. An error group E which will be a common over group for the two stabilizer groups is first created.

```
> F<w> := GF(4);
> Q1 := RandomQuantumCode(F, 15, 8);
> Q2 := RandomQuantumCode(F, 15, 8);
>
> E := QuantumErrorGroup(Q1);
> S1 := StabilizerGroup(Q1, E);
> S2 := StabilizerGroup(Q2, E);
> #(S1 meet S2);
2
```

164.4 Inner Products and Duals

The functions described in this section use the symplectic inner product defined for quantum codes.

`SymplecticInnerProduct(v1, v2)`

`ExtendedFormat` BoolElt *Default :* `false`

Let $v1$ and $v2$ be two vectors belonging to the vector space $K^{(n)}$, where K is a finite field. This function returns the inner product of $v1$ and $v2$ with respect to the symplectic inner product. The symplectic inner product in extended format is defined by $(a|b)*(c|d) = ad-bc$, and its definition transfers naturally to the compact format.

For binary quantum codes whose compact format is over $GF(4)$, the symplectic inner product is given by Trace($v_1 \cdot \overline{v}_2$).

`SymplecticDual(C)`

`ExtendedFormat` BoolElt *Default :* `false`

The dual of the additive (or possibly linear) code C with respect to the symplectic inner product. By default, C is interpreted as being in the compact format (a length n code over $GF(q^2)$), but if `ExtendedFormat` is set to `true`, then it will be interpreted as being in extended format (a code of length $2n$ over $GF(q)$).

```
IsSymplecticSelfDual(C)
```

ExtendedFormat BoolElt *Default :* false

Return true if the code C is equal to its symplectic dual and false otherwise. By default, C is interpreted as being in the compact format (a length n code over $GF(q^2)$), but if ExtendedFormat is set to true, then it will be interpreted as being in extended format (a code of length $2n$ over $GF(q)$).

```
IsSymplecticSelfOrthogonal(C)
```

ExtendedFormat BoolElt *Default :* false

Return true if the code C is contained in its symplectic dual. By default, C is interpreted as being in the compact format (a length n code over $GF(q^2)$), but if ExtendedFormat is set to true, then it will be interpreted as being in extended format (a code of length $2n$ over $GF(q)$).

Example H164E17

Vectors which are symplectically orthogonal to one another can be used to construct symplectic self-orthogonal codes.

```
> F<w> := GF(4);
> V5 := VectorSpace(F, 5);
> v := V5 ! [1,0,w,0,1];
> w := V5 ! [w,1,0,w,w];
> SymplecticInnerProduct(v,w);
0
> C := AdditiveCode<F, GF(2), 5 | v, w>;
> C;
[5, 1 : 2] GF(2)-Additive Code over GF(2^2)
Generator matrix:
[  1   0   w   0   1]
[  w   1   0   w   w]
> D := SymplecticDual(C);
> D;
[5, 4 : 8] GF(2)-Additive Code over GF(2^2)
Generator matrix:
[  1   0   0   0   1]
[  w   0   0   0   w]
[  0   1   0   0   0]
[  0   w   0   0   1]
[  0   0   1   0   w]
[  0   0   w   0   0]
[  0   0   0   1   1]
[  0   0   0   w   0]
> C subset D;
true
> Q := QuantumCode(C);
> Q;
```

```
[[5, 3]] Quantum code over GF(2^2), stabilised by:
[   1   0   w   0   1]
[   w   1   0   w   w]
```

Example H164E18

Any vector over $GF(4)$ will be symplectically orthogonal to itself.

```
> V5 := VectorSpace(GF(4), 5);
> { SymplecticInnerProduct(v, v) : v in V5 };
{ 0 }
```

164.5 Weight Distribution and Minimum Weight

The weight distribution of a quantum code Q consists of three separate distributions:

- The weight distribution of the stabilizer code S.
- The weight distribution of the symplectic dual $S^{\perp}$ of S.
- The weight distribution of the codewords in $S^{\perp}\backslash S$. Note that this set is not a linear space.

For a quantum code Q with stabilizer code S, the weights of the undetectable errors are the weights of the codewords in $S^{\perp}\backslash S$.

For a quantum code to be considered *pure*, its minimum weight must be less than or equal to the weight of its stabilizer code.

```
WeightDistribution(Q)
```

Given a quantum code Q with stabiliser code S, return its weight distribution. Recall that the quantum weight distribution comprises the weight distributions of S, $S^{\perp}$ and $S^{\perp}\backslash S$. The function returns each distribution as a separate value. Each weight distribution is returned in the form of a sequence of tuples consisting of a weight and the number of code words of that weight.

Example H164E19

Looking at a small quantum code from the database of best known codes. Its first weight distribution is of its stabilizer code S, the second of its normalizer code $S^{\perp}$, and the final weight distribution is of those non-zero codewords in $S^{\perp}\backslash S$.

```
> F<w> := GF(4);
> Q := QECC(GF(4),6,3);
> Q;
[[6, 3, 2]] Quantum code over GF(2^2), stabilised by:
[   1   0   1   1   1   0]
[   w   w   w   w   w   w]
[   0   1   0   0   0   1]
```

```
> WD_S, WD_N, WD := WeightDistribution(Q);
> WD_S eq WeightDistribution(StabiliserCode(Q));
true
> WD_N eq WeightDistribution(NormaliserCode(Q));
true
> WD;
[ <2, 28>, <3, 56>, <4, 154>, <5, 168>, <6, 98> ]
```

`MinimumWeight(Q)`

Method	MonStgElt	*Default* : "*Auto*"
RankLowerBound	RngIntElt	*Default* : 0
MaximumTime	RngReSubElt	*Default* : ∞

For the quantum code Q with stabilizer code S, return the minimum weight of Q, which is the minimum weight of the codewords in $S^{\perp}\backslash S$. For self-dual quantum codes (those of dimension 0), the minimum weight is defined to be the minimum weight of S. The default algorithm is based on the minimum weight algorithm for classical linear codes which is described in detail in section 158.8.1. For a description of the algorithm and its variable argument parameters please consult the full description provided there. The minimum weight may alternatively be calculated by finding the complete weight distribution. This algorithm may be selected by setting the value of the variable argument `Method` to "`Distribution`".

Example H164E20

The verbose output can be used in long minimum weight calculations to estimate the remaining running time. The algorithm terminates once the lower bound reaches the upper bound. The example below finishes in a very short period of time.

```
> F<w> := GF(4);
> V5 := VectorSpace(F, 5);
> gens := [V5| [0,0,1,w,w], [0,1,1,w,1], [0,0,1,0,w^2],
>              [0,0,1,w,1], [0,0,1,0,1], [1,w,1,w,w^2],
>              [1,1,1,w^2,w], [0,1,w,1,w^2] ];
> Q := QuantumQuasiCyclicCode(gens : LinearSpan := true);
> Q:Minimal;
[[40, 30]] Quantum code over GF(2^2)
> SetVerbose("Code",true);
> MinimumWeight(Q);
Quantum GF(2)-Additive code over GF(4) of length 40 with 70 generators.
Lower Bound: 1, Upper Bound: 40
Constructed 2 distinct generator matrices
Total Ranks:     35  35
Relative Ranks:  35   5
Time Taken: 0.14
Starting search for low weight codewords... (reset timings)
```

```
Enumerating using 1 generator at a time:
     New codeword identified of weight 6, time 0.00
     New codeword identified of weight 4, time 0.00
     Discarding non-contributing rank 5 matrix
     New Total Ranks:     35
     New Relative Ranks:  35
   Completed Matrix  1:  lower =  2, upper =  4. Elapsed: 0.00s
Termination predicted with 3 generators at matrix 1
Enumerating using 2 generators at a time:
   Completed Matrix  1:  lower =  3, upper =  4. Elapsed: 0.00s
Termination predicted with 3 generators at matrix 1
predicting enumerating (1820) 60725 vectors (0.000000% of 40 40 code)
Enumerating using 3 generators at a time:
   Completed Matrix  1:  lower =  4, upper =  4. Elapsed: 0.03s
Final Results: lower = 4, upper = 4, Total time: 0.03
4
```

`IsPure(Q)`

Return `true` if Q is a *pure* quantum code. That is, if the minimum weight of Q is less than or equal to the minimum weight of its stabiliser code.

Example H164E21

Many good codes are impure, the purity of best known quantum codes of length 15 are investigated.

```
> F<w> := GF(4);
> n := 15;
> time {* IsPure(QECC(F, n, k)) : k in [1..n] *};
{* false^^10, true^^5 *}
Time: 0.410
```

164.6 New Codes From Old

```
DirectSum(Q1, Q2)
```

Given an $[[n_1, k_1, d_1]]$ quantum code Q_1, and an $[[n_2, k_2, d_2]]$ quantum code Q_2, return the $[[n_1+n_n, k_1+k_2, \min\{d_1, d_2\}]]$ quantum code which is their direct product.

```
ExtendCode(Q)
```

Given an $[[n, k, d]]$ quantum code Q, return the extended $[[n+1, k, d]]$ quantum code.

```
ExtendCode(Q, m)
```

Perform m extensions on the $[[n, k, d]]$ quantum code Q, returning the extended $[[n + m, k, d]]$ quantum code.

```
PunctureCode(Q, i)
```

Given a $[[n, k, d]]$ quantum code Q, and a coordinate position i, return the $[[n - 1, k, d' >= d - 1]]$ quantum code produced by puncturing at position i.

```
PunctureCode(Q, I)
```

Given a $[[n, k, d]]$ quantum code Q, and a set of coordinate positions I of size s, return the $[[n - s, k, d' >= d - s]]$ quantum code produced by puncturing at the positions in I.

```
ShortenCode(Q, i)
```

Given a $[[n, k, d]]$ quantum code Q, and a coordinate position i, return the $[[n - 1, k' >= k - 1, d' >= d]]$ quantum code produced by shortening at position i.

This process will not necessarily result in a valid (symplectic self-orthogonal) quantum code, and an error will be given if it fails.

```
ShortenCode(Q, I)
```

Given a $[[n, k, d]]$ quantum code Q, and a set of coordinate positions I of size s, return the $[[n - s, k' >= k - s, d' >= d]]$ quantum code produced by shortening at the positions in I.

This process will not necessarily result in a valid (symplectic self-orthogonal) quantum code, and an error will be given if it fails.

Example H164E22

Good quantum codes can be created by combining stabilizer codes, using methods which are not general enough to warrant a specific quantum code function. This example creates a $[[28, 8, 6]]$ quantum code from $[[14, 8, 3]]$ and $[[14, 0, 6]]$ quantum codes using a Plotkin sum. It relies on the stabilizer codes forming a subcode chain, as described in Theorem 12 in [CRSS98].

```
> F<w> := GF(4);
> V7 := VectorSpace(F, 7);
> v1 := V7 ! [1,0,0,0,0,0,0];
> v2 := V7 ! [w^2,1,w^2,w,0,0,w];
> Q1 := QuantumQuasiCyclicCode([v1, v2] : LinearSpan := true);
> _ := MinimumWeight(Q1);
> Q1:Minimal;
[[14, 0, 6]] self-dual Quantum code over GF(2^2)
>
> v1 := V7 ! [1,0,1,1,1,0,0];
> v2 := V7 ! [1,w^2,w,w,1,0,w^2];
> Q2 := QuantumQuasiCyclicCode([v1, v2] : LinearSpan := true);
> _ := MinimumWeight(Q2);
> Q2:Minimal;
[[14, 8, 3]] Quantum code over GF(2^2)
>
> S1 := StabilizerCode(Q1);
> S2 := StabilizerCode(Q2);
> S2 subset S1;
true
>
> S3 := PlotkinSum(SymplecticDual(S1), S2);
> Q3 := QuantumCode(S3);
> _ := MinimumWeight(Q3);
> Q3:Minimal;
[[28, 8, 6]] Quantum code over GF(2^2)
```

164.7 Best Known Quantum Codes

An $[[n, k]]$ quantum stabiliser code Q is said to be a *best known* $[[n, k]]$ *quantum code* (BKQC) if C has the highest minimum weight among all known $[[n, k]]$ quantum codes. The acronym QECC (Quantum Error Correcting Code) will be used to more easily distinguish from the best known linear codes database (BKLC).

Magma currently has a database for binary quantum codes, though it should be noted that these codes are considered to be over the alphabet $GF(4)$, not $GF(2)$. The database for codes over $GF(4)$ currently contains constructions of all best known quantum codes of length 35. This includes self-dual quantum codes up to length 35, which are stored in the database as dimension 0 quantum codes.

Quantum codes of length up to 12 are optimal, in the sense that their minimum weights meet the upper bound. Thus the user has access to 665 best-known binary quantum codes.

The MAGMA QECC database uses the tables of bounds and constructions compiled by Markus Grassl (Karlsruhe), available online at [Gra], which are based on the results in [CRSS98]. Good codes have also been contributed by Eric Rains and Zlatko Varbanov.

The user can display the method used to construct a particular QECC code through use of a verbose mode, triggered by the verbose flag `BestCode`. When it is set to `true`, all of the functions in this section will output the steps involved in each code they construct.

```
QECC(F, n, k)
BKQC(F, n, k)
BestKnownQuantumCode(F, n, k)
```

Given a finite field F, and positive integers n and k such that $k \leq n$, return an $[[n, k]]$ quantum code over F which has the largest minimum weight among all known $[[n, k]]$ quantum codes. A second boolean return value signals whether or not the desired code exists in the database.

The database currently exists for $GF(4)$ (which are in fact binary quantum codes) up to length 35.

Example H164E23

The weight distribution of a small best known quantum code is calculated, verifying its minimum weight. Note that the *larger* the dimension of a quantum code, the easier it is to calculate its weight distribution.

```
> F<w> := GF(4);
> Q := QECC(F,25,16);
> Q:Minimal;
[[25, 16, 3]] Quantum code over GF(2^2)
> time WD_S, WD_N, WD := WeightDistribution(Q);
Time: 0.010
> WD_S;
[ <0, 1>, <1, 2>, <2, 1>, <14, 4>, <15, 16>, <16, 38>, <17, 79>, <18, 126>, <19,
129>, <20, 77>, <21, 27>, <22, 9>, <23, 3> ]
> WD_N;
[ <0, 1>, <1, 2>, <2, 1>, <3, 399>, <4, 6527>, <5, 75363>, <6, 707543>, <7,
5404369>, <8, 34084490>, <9, 180107319>, <10, 804255370>, <11, 3052443894>, <12,
9883860222>, <13, 27348684334>, <14, 64649758926>, <15, 130286413858>, <16,
222912028997>, <17, 321704696752>, <18, 387985433701>, <19, 385943417035>, <20,
310898936275>, <21, 197566276671>, <22, 95232787563>, <23, 32688613821>, <24,
7109768160>, <25, 735493959> ]
> WD;
[ <3, 399>, <4, 6527>, <5, 75363>, <6, 707543>, <7, 5404369>, <8, 34084490>, <9,
180107319>, <10, 804255370>, <11, 3052443894>, <12, 9883860222>, <13,
27348684334>, <14, 64649758922>, <15, 130286413842>, <16, 222912028959>, <17,
321704696673>, <18, 387985433575>, <19, 385943416906>, <20, 310898936198>, <21,
197566276644>, <22, 95232787554>, <23, 32688613818>, <24, 7109768160>, <25,
735493959> ]
```

So the $[[25, 16]]$ code is impure, and has a minimum distance of 3.

Example H164E24

Unlike linear codes, dimension 0 quantum codes are non-trivial and are the subject of much study. These are the *self-dual* quantum codes, which form a special subclass of quantum stabilizer codes. It can be seen that a length n self-dual code is described by a $n \times n$ generator matrix, an indication of the non-triviality of its structure.

```
> F<w> := GF(4);
> C := QECC(GF(4),8, 0);
> C;
[[8, 0, 4]] self-dual Quantum code over GF(2^2), stabilised by:
[ 1   0   0   1   0   1   1   0]
[ w   0   0   w   0   w   w   0]
[ 0   1   0   1   0   1   0   1]
[ 0   w   0   w   0   w   0   w]
[ 0   0   1   1   0   0   1   1]
[ 0   0   w   w   0   0   w   w]
[ 0   0   0   0   1   1   1   1]
[ 0   0   0   0   w   w   w   w]
```

Example H164E25

The verbose flag `BestCode` will show the method by which the best code is constructed in the database. In this example the construction of a $[[25, 11, 4]]$ quantum code is described.

```
> SetVerbose("BestCode",true);
> F<w> := GF(4);
> Q := QECC(F,25,11);
Construction of a [[ 25 , 11 , 4 ]] Quantum Code:
[1]:  [[40, 30, 4]] Quantum code over GF(2^2)
        QuasiCyclicCode of length 40 stacked to height 2 with generating
        polynomials: 1,  w^2*x^4 + w*x^3 + w^2*x^2 + w*x + w^2,  x^4 + w^2*x^2 +
        w^2*x + w^2,  x^4 + w*x^3 + x^2,  w*x^4 + x^3 + w^2*x,  w*x^4 + x^3 +
        x^2 + x,  w^2*x^4 + x^2 + w*x,  x^4 + w^2*x^3 + w^2*x^2 + x + 1,  w,
        x^4 + w^2*x^3 + x^2 + w^2*x + 1,  w*x^4 + x^2 + x + 1,  w*x^4 + w^2*x^3
        + w*x^2,  w^2*x^4 + w*x^3 + x,  w^2*x^4 + w*x^3 + w*x^2 + w*x,  x^4 +
        w*x^2 + w^2*x,  w*x^4 + x^3 + x^2 + w*x + w
[2]:  [[21, 11, 4]] Quantum code over GF(2^2)
        Shortening of [1] at { 2, 3, 4, 6, 8, 9, 10, 12, 13, 14, 15, 16, 18, 19,
        21, 24, 28, 34, 37 }
[3]:  [[25, 11, 4]] Quantum code over GF(2^2)
        ExtendCode [2] by 4
> Q:Minimal;
```

```
[[25, 11, 4]] Quantum code over GF(2^2)
```

164.8 Best Known Bounds

Along with the database of best known quantum codes in the previous section, there is also a database of best known upper and lower bounds on the maximal possible minimum weights of quantum codes. The upper bounds are not currently known with much accuracy, while the lower bounds match the minimum weights of the best known quantum codes database.

```
QECCLowerBound(F, n, k)
```

Return the best known lower bound on the maximal minimum distance of $[[n, k]]$ quantum codes over F. The bounds are currently available for binary quantum codes (which corresponds to $F = GF(4)$) up to length 35.

```
QECCUpperBound(F, n, k)
```

Return the best known upper bound on the minimum distance of $[[n, k]]$ quantum codes over F. The bounds are currently available for binary quantum codes (which corresponds to $F = GF(4)$) up to length 35.

Example H164E26

The best known lower bound on the minimum weight will always correspond to the best known quantum code from the Magma database. In this example the first code is in fact optimal, while the second one does not meet the upper bound, and so there is a theoretical possibility of an improvement.

```
> F<w> := GF(4);
> Q1 := QECC(F, 20, 10);
> Q1:Minimal;
[[20, 10, 4]] Quantum code over GF(2^2)
> QECCLowerBound(F, 20, 10);
4
> QECCUpperBound(F, 20, 10);
4
>
> Q2 := QECC(F, 25, 13);
> Q2:Minimal;
[[25, 13, 4]] Quantum code over GF(2^2)
> QECCLowerBound(F, 25, 13);
4
> QECCUpperBound(F, 25, 13);
5
```

164.9 Automorphism Group

Automorphisms acting on a quantum code are a slight generalization of those which act on the underlying additive stabilizer code. Automorphisms consists of both a permutation action on the columns of a stabilizer code, combined with a monomial action on the individual columns which permute the values.

The automorphism group of a length n additive stabilizer code over $\mathbf{F}_4$ is a subgroup of $Z_3 \wr Sym(n)$ of order $3 * n!$. However the automorphism group of the quantum code it generates is a subgroup of $Sym(3) \wr Sym(n)$ of order $3! * n!$ because of the more general action on the values in the columns.

In Magma automorphisms are returned as permutations, either as length $3n$ permutations for the full monomial action on a code, or as length n permutations when the automorphism is restricted to only the permutation action on the columns.

```
AutomorphismGroup(Q)
```

The automorphism group of the quantum code Q. Currently this function only applies to binary quantum codes.

```
PermutationGroup(Q)
```

The subgroup of the automorphism group of the quantum code Q consisting of those automorphisms which permute the coordinates of codewords. Currently this function only applies to binary quantum codes.

Example H164E27

The full automorphism group and its subgroup of coordinate permutations are calculated for the dodecacode.

```
> F<w> := GF(4);
> Q := Dodecacode();
> Q;
[[12, 0, 6]] self-dual Quantum code over GF(2^2), stabilised by:
[  1   0   0   0   0   0 w^2 w^2   0   w   1   w]
[  w   0   0   0   0   0   w   0   w   w   w   1]
[  0   1   0   0   0   0   1   0   1 w^2 w^2   1]
[  0   w   0   0   0   0   0   w   1   w   w   w]
[  0   0   1   0   0   0   0   1   1   1 w^2 w^2]
[  0   0   w   0   0   0 w^2   1 w^2   w   w   0]
[  0   0   0   1   0   0   w   w   1   1   0   1]
[  0   0   0   w   0   0   w   1   w w^2 w^2   0]
[  0   0   0   0   1   0   w   w   w   0   1   w]
[  0   0   0   0   w   0 w^2   1   1 w^2   0   w]
[  0   0   0   0   0   1 w^2 w^2 w^2   1   0 w^2]
[  0   0   0   0   0   w   w   1   0   1   w   w]
>
> AutomorphismGroup(Q);
Permutation group acting on a set of cardinality 36
Order = 648 = 2^3 * 3^4
```

```
    (1, 4, 32)(2, 5, 33)(3, 6, 31)(7, 13, 29)(8, 14, 30)(9, 15, 28)(10, 35, 22)
        (11, 36, 23)(12, 34, 24)(16, 19, 26)(17, 20, 27)(18, 21, 25)
    (4, 23, 8, 29, 10, 20, 18, 36, 32)(5, 24, 9, 30, 11, 21, 16, 34, 33)
        (6, 22, 7, 28, 12, 19, 17, 35, 31)(13, 14, 15)(25, 27, 26)
    (7, 35)(8, 36)(9, 34)(10, 20)(11, 21)(12, 19)(13, 26)(14, 27)(15, 25)
        (16, 30)(17, 28)(18, 29)(22, 31)(23, 32)(24, 33)
    (4, 29, 18)(5, 30, 16)(6, 28, 17)(7, 19, 31)(8, 20, 32)(9, 21, 33)
        (10, 36, 23)(11, 34, 24)(12, 35, 22)
> PermutationGroup(Q);
Permutation group acting on a set of cardinality 12
    (1, 7, 9, 3, 5, 11)(2, 8, 10, 4, 6, 12)
    (1, 2)(3, 4)(5, 10)(6, 9)(7, 12)(8, 11)
    (2, 4)(5, 9)(6, 12)(7, 11)(8, 10)
```

Example H164E28

The automorphism group for a quantum code is larger than that of its stabilizer code. In this example that is shown for the Hexacode.

```
> F<w> := GF(4);
> Q := Hexacode();
> Q:Minimal;
[[6, 0, 4]] self-dual Quantum code over GF(2^2)
> A_Q := AutomorphismGroup(Q);
> A_Q;
Permutation group A_Q acting on a set of cardinality 18
Order = 2160 = 2^4 * 3^3 * 5
    (1, 4)(2, 6)(3, 5)(7, 8)(10, 12)(13, 14)(17, 18)
    (2, 3)(5, 6)(7, 8)(10, 18)(11, 16)(12, 17)(13, 14)
    (4, 7)(5, 8)(6, 9)(13, 17)(14, 16)(15, 18)
    (7, 13)(8, 14)(9, 15)(10, 17)(11, 16)(12, 18)
    (7, 12)(8, 10)(9, 11)(13, 18)(14, 17)(15, 16)
> S := StabilizerCode(Q);
> A_S := AutomorphismGroup(S);
> A_S;
Permutation group A_S acting on a set of cardinality 18
Order = 180 = 2^2 * 3^2 * 5
    (1, 4)(2, 5)(3, 6)(7, 13)(8, 14)(9, 15)
    (4, 7, 12)(5, 8, 10)(6, 9, 11)(13, 15, 14)(16, 18, 17)
    (4, 6, 5)(7, 14, 11)(8, 15, 12)(9, 13, 10)(16, 18, 17)
> A_S subset A_Q;
true
```

164.10 Hilbert Spaces

In this first release, Magma offers a basic package for creating and computing with quantum Hilbert spaces. A Hilbert space in Magma can either be *densely* or *sparsely* represented, depending on how many qubits are required and how dense the desired quantum states will be. While a dense representation has a speed advantage in computations, the sparse representation uses less memory. Currently there are capabilities for doing basic unitary transformations and manipulations of quantum states.

In future versions, functionality will be added for more complex unitary transformations and measurements, allowing for a more general simulation of quantum computations. There will also be machinery for encoding quantum states using quantum error correcting codes, and testing their effectiveness by simulating a noisy quantum channel and decoding the results.

`HilbertSpace(F, n)`

`IsDense` BoolElt *Default :*

Given a complex field F and a positive integer n, return then quantum Hilbert Space on n qubits over F.

If the variable argument `IsDense` is set to either `true` or `false` then return a densely or sparsely represented quantum space respectively. If no value is set for `IsDense` then Magma will decide automatically.

`Field(H)`

Given a Hilbert space H, return the complex field over which the coefficients of states of H are defined.

`NumberOfQubits(H)`

`Nqubits(H)`

Given a Hilbert space H, return the number of qubits which comprises the space.

`Dimension(H)`

Given a Hilbert space H, return its dimension. This is 2^n, where n is the number of qubits of H.

`IsDenselyRepresented(H)`

Return `true` if the quantum Hilbert space H uses a dense representation.

`H1 eq H2`

Return `true` if the Hilbert spaces are equal.

`H1 ne H2`

Return `true` if the Hilbert spaces are not equal.

Example H164E29

A Hilbert space over 5 qubits will by default be a densely represented quantum space. It can however be manually chosen to use a sparse representation, it can be seen that these two space are not considered equal.

```
> F<i> := ComplexField(4);
> H := HilbertSpace(F, 5);
> H;
A densely represented Hilbert Space on 5 qubits to precision 4
> Dimension(H);
32
> IsDenselyRepresented(H);
true
>
> H1 := HilbertSpace(F, 5 : IsDense := false);
> H1;
A sparely represented Hilbert Space on 5 qubits to precision 4
> IsDenselyRepresented(H1);
false
> H eq H1;
false
```

164.10.1 Creation of Quantum States

`QuantumState(H, v)`

`QuantumState(H, v)`

Given a Hilbert space H and coefficients v (which can be either a dense or a sparse vector), of length equal to the dimension of H, then return the quantum state in H defined by v.

`H ! i`

Return the i-th quantum basis state of the Hilbert space H. This corresponds to the basis state whose qubits giving a binary representation of i.

`H ! s`

Given a sequence s of binary values, whose length is equal to the number of qubits of the Hilbert space H, return the quantum basis state corresponding to s.

`SetPrintKetsInteger(b)`

Input is a boolean value b, which controls a global variable determining the way quantum states are printed. If set to `false` (which is the default) then values in basis kets will be printed as binary sequences such as $|1010\rangle$. If set to `true` then basis kets will be printed using integer values to represent the binary sequences, the previous example becoming $|5\rangle$.

Example H164E30

One way to create a quantum state is to specify each coefficient of the state with a vector of length equal to the dimension of the Hilbert space.

```
> F<i> := ComplexField(4);
> H := HilbertSpace(F, 4);
> KS := KSpace(F, Dimension(H));
> v := KS! [F| i,    1,    0, -i,
>              2,    0,    0, 1+i,
>              -i-1, -3*i, 7, 0.5,
>              2.5*i, 0,   0, 1.2];
> v;
(1.000*i 1.000 0.0000 -1.000*i 2.000 0.0000 0.0000 1.000 + 1.000*i
    -1.000 - 1.000*i -3.000*i 7.000 0.5000 2.500*i 0.0000 0.0000
    1.200)
> e := QuantumState(H, v);
> e;
1.000*i|0000> + |1000> - 1.000*i|1100> + 2.000|0010> + (1.000 +
1.000*i)|1110> - (1.000 + 1.000*i)|0001> - 3.000*i|1001> + 7.000|0101>
+ 0.5000|1101> + 2.500*i|0011> + 1.200|1111>
```

Example H164E31

Quantum states can be created by combining basis states, input as either integer values or binary sequences.

```
> F<i> := ComplexField(4);
> H := HilbertSpace(F, 12);
> Dimension(H);
4096
> e1 := H!1 + (1+i)*(H!76) - H!3000;
> e1;
|100000000000> + (1.000 + 1.000*i)|001100100000> - |000111011101>
> e2 := H![1,0,1,1,1,0,0,0,1,1,0,0] - H![1,1,0,1,0,0,0,0,1,1,0,1];
> e2;
|101110001100> - |110100001101>
```

By using the function SetPrintKetsInteger basis states can also be printed as either integer values of binary sequences.

```
> SetPrintKetsInteger(true);
> e1;
|1> + (1.000 + 1.000*i)|76> - |3000>
> e2;
|797> - |2827>
```

164.10.2 Manipulation of Quantum States

`a * e`

Given a complex scalar value a, multiply the coefficients of the quantum state e by a.

`-e`

Negate all coefficients of the quantum state e.

`e1 + e2`

`e1 - e2`

Addition and subtraction of the quantum states e_1 and e_2.

`Normalisation(e)`

`Normalisation(~e)`

`Normalization(e)`

`Normalization(~e)`

Normalize the coefficients of the quantum state e, giving an equivalent state whose normalization coefficient is equal to one. Available either as a procedure or a function.

`NormalisationCoefficient(e)`

`NormalizationCoefficient(e)`

Return the normalisation coefficient of the quantum state e

`e1 eq e2`

Return `true` if and only if the quantum states e_1 and e_2 are equal. States are still considered equal if they have different normalizations.

`e1 ne e2`

Return `true` if and only if the quantum states e_1 and e_2 are not equal. States are still considered equal if they have different normalizations.

Example H164E32

Although a quantum state can be expressed with any normalisation, in reality a quantum state occupies a ray in a Hilbert space. So two quantum states are still considered equal if they lie on the same ray.

```
> F<i> := ComplexField(8);
> H := HilbertSpace(F, 1);
> e := H!0 + H!1;
> e;
|0> + |1>
> NormalisationCoefficient(e);
2.0000000
> e1 := Normalisation(e);
> e1;
0.70710678|0> + 0.70710678|1>
> NormalisationCoefficient(e1);
0.99999999
> e eq e1;
true
```

164.10.3 Inner Product and Probabilities of Quantum States

`InnerProduct(e1, e2)`

Return the inner product of the quantum states e_1 and e_2.

`ProbabilityDistribution(e)`

Return the probability distribution of the quantum state as a vector over the reals.

`Probability(e, i)`

Return the probability of basis state i being returned as the result of a measurement on the quantum state e.

`Probability(e, v)`

Given a binary vector v of length equal to the number of qubits in the quantum state e, return the probability of basis state corresponding to v being returned as the result of a measurement on e.

`PrintProbabilityDistribution(e)`

Print the probability distribution of the quantum state.

`PrintSortedProbabilityDistribution(e)`

`Max`	RngIntElt	*Default :* ∞
`MinProbability`	RngIntElt	*Default :* 0

Print the probability distribution of the quantum state in sorted order, with the most probable states printed first.

If the variable argument `Max` is set to a positive integer, then it will denote the maximum number of basis states to be printed.

If the variable argument `MinProbability` is set to some integer between 1 and 100, then it will denote the minimum probability of any basis state to be printed. This is useful for investigating those basis states which will are the likely results of any measurement.

Example H164E33

From a quantum state it is possible to either access the full probability distribution, or the probabilities of individual basis states.

```
> F<i> := ComplexField(4);
> H := HilbertSpace(F, 3);
> e := -0.5*H!0 + 6*i*H!3 + 7*H!4 - (1+i)*H!7;
> ProbabilityDistribution(e);
(0.002865 0.0000 0.0000 0.4126 0.5616 0.0000 0.0000 0.02292)
> Probability(e, 0);
0.002865
> Probability(e, 1);
0.0000
```

It is also possible to print out the full probability distribution.

```
> PrintProbabilityDistribution(e);
Non-zero probabilities:
|000>:  0.2865%
|110>:  41.26%
|001>:  56.16%
|111>:  2.292%
```

Example H164E34

It is usually only those basis states with large probabilities that are of interest. With the function `PrintSortedProbabilitydistribution` these basis states can be identified.

```
> F<i> := ComplexField(4);
> H := HilbertSpace(F, 4);
> KS := KSpace(F, 2^4);
> v := KS! [F| i,   11,   0, -3*i,
>              2,   0,    0, 6+i,
>            -i-1, -3*i, 7, -0.5,
>            2.5*i, 0,   0, 9.2];
> e := QuantumState(H, v);
> e;
1.000*i|0000> + 11.00|1000> - 3.000*i|1100> + 2.000|0010> + (6.000 +
1.000*i)|1110> - (1.000 + 1.000*i)|0001> - 3.000*i|1001> + 7.000|0101>
- 0.5000|1101> + 2.500*i|0011> + 9.200|1111>
> PrintSortedProbabilityDistribution(e);
Non-zero probabilities:
|1000>:         37.45%
|1111>:         26.19%
|0101>:         15.16%
|1110>:         11.45%
|1100>:         2.785%
|1001>:         2.785%
|0011>:         1.934%
|0010>:         1.238%
|0001>:         0.6190%
|0000>:         0.3095%
|1101>:         0.07737%
```

A useful way to isolate the important basis states is to provide a minimum cutoff probability.

```
> PrintSortedProbabilityDistribution(e: MinProbability := 15);
Non-zero probabilities:
|1000>:         37.45%
|1111>:         26.19%
|0101>:         15.16%
Reached Minimum Percentage
```

Another way is to supply the maximum number basis states that should be printed. A combination of these methods can also be used

```
> PrintSortedProbabilityDistribution(e: Max := 6);
Non-zero probabilities:
|1000>:         37.45%
|1111>:         26.19%
|0101>:         15.16%
|1110>:         11.45%
|1100>:         2.785%
|1001>:         2.785%
Reached Maximum count
```

164.10.4 Unitary Transformations on Quantum States

In this first release Magma offers a small selection of unitary transformations on quantum states. In future versions this list will be expanded to include more complex operations.

```
BitFlip(e, k)
BitFlip(∼e, k)
```

Flip the value of the k-th qubit of the quantum state e.

```
BitFlip(e, B)
BitFlip(∼e, B)
```

Given a set of positive integers B, flip the value of the qubits of the quantum state e indexed by the entries in B.

```
PhaseFlip(e, k)
PhaseFlip(∼e, k)
```

Flip the phase on the k-th qubit of the quantum state e.

```
PhaseFlip(e, B)
PhaseFlip(∼e, B)
```

Given a set of positive integers B, flip the phase on the qubits of the quantum state e indexed by the entries in B.

```
ControlledNot(e, B, k)
ControlledNot(∼e, B, k)
```

Flip the k-th bit of the quantum state e if all bits contained in B are set to 1.

```
HadamardTrasformation(e)
HadamardTrasformation(∼e)
```

Perform a Hadamard transformation on the quantum state e, which must be densely represented.

Example H164E35

The behaviours of several of the available unitary transformations are displayed on a quantum state.

```
> F<i> := ComplexField(4);
> H := HilbertSpace(F, 4);
> e := H!0 + H!3 + H!6 + H!15;
> PhaseFlip(~e, 4); e;
|0000> + |1100> + |0110> - |1111>
> ControlledNot(~e, {1,2}, 4); e;
|0000> + |0110> - |1110> + |1101>
> BitFlip(~e, 2); e;
```

```
|0100> + |0010> - |1010> + |1001>
> ControlledNot(~e, {2}, 3); e;
|0010> - |1010> + |0110> + |1001>
```

164.11 Bibliography

[**CRSS98**] A. Robert Calderbank, Eric M. Rains, P. W. Shor, and Neil J. A. Sloane. Quantum error correction via codes over GF(4). *IEEE Trans. Inform. Theory*, 44(4): 1369–1387, 1998.

[**CS96**] A. R. Calderbank and P. W. Shor. Good quantum error-correcting codes exist. *Phys. Rev. A*, 54:2551–2577, 1996.

[**Dan05**] D. E. Danielsen. On-self dual quantum codes, graphs, and Boolean functions. Master's thesis, University of Bergen, 2005.

[**Gra**] Markus Grassl. Bounds on the minimum distance of quantum codes. URL:http://iaks-www.ira.uka.de/home/grassl/QECC/.

[**Sho94**] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science (Santa Fe, NM, 1994)*, pages 124–134. IEEE Comput. Soc. Press, Los Alamitos, CA, 1994.

[**Sho95**] P. W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, 52:2493–2496, 1995.

[**Ste96a**] A. M. Steane. Error correcting codes in quantum theory. *Phys. Rev. Lett.*, 77(5):793–797, 1996.

[**Ste96b**] Andrew Steane. Multiple-particle interference and quantum error correction. *Proc. Roy. Soc. London Ser. A*, 452(1954):2551–2577, 1996.

PART XXII
CRYPTOGRAPHY

165 PSEUDO-RANDOM BIT SEQUENCES

Chapter 165
PSEUDO-RANDOM BIT SEQUENCES

165.1 Introduction

Magma provides some tools for the creation and analysis of pseudo-random bit sequences. The universe of these sequences is generally $\mathbf{F}_2$. However, some functions, such as `BerlekampMassey`, may be applied to sequences defined over arbitrary finite fields.

165.2 Linear Feedback Shift Registers

For a linear feedback shift register (LFSR) of length L, initial state $s_0, \ldots, s_{L-1} \in \mathbf{F}_q$, and connection polynomial $C(D) = 1 + c_1 D + c_2 D^2 + \ldots + c_L D^L$ (also over $\mathbf{F}_q$), the j'th element of the sequence is computed as $s_j = -\sum_{i=1}^{L} c_i s_{j-i}$ for $j \geq L$.

```
LFSRSequence(C, S, t)
```

Computes the first t sequence elements of the LFSR with connection polynomial C and initial state the sequence S (thus, the length of the LFSR is assumed to be the length of S). C must be at least degree 1, its coefficients must come from the same finite field as the universe of S, and its constant coefficient must be 1. Also, the sequence S must have at least as many terms as the degree of C.

```
LFSRStep(C, S)
```

Computes the next state of the LFSR having connection polynomial C and current state the sequence S (thus, the length of the LFSR is assumed to be the length of S). C must be at least degree 1, its coefficients must come from the same finite field as the universe of S, and its constant coefficient must be 1. Also, the sequence S must have at least as many terms as the degree of C.

```
BerlekampMassey(S)
```

```
ConnectionPolynomial(S)
```

```
CharacteristicPolynomial(S)
```

Given a sequence S of elements from $\mathbf{F}_q$, return the connection polynomial $C(D)$ and the length L of a LFSR that generates the sequence S.

Note that it is possible that the `BerlekampMassey` will return a singular LFSR (i.e. the degree of $C(D)$ is less than L), and therefore one must be sure to use the first L elements of S to regenerate the sequence.

Example H165E1

We first create a sequence and then use `BerlekampMassey` to get the connection polynomial and its length:

```
> S:= [GF(2)| 1,1,0,1,0,1,1,1,0,0,1,0];
> C<D>, L := BerlekampMassey(S);
> C;
D^3 + D^2 + 1
> L;
5
```

Now create a new sequence T containing the first L elements of S, and reconstruct the sequence from $C(D)$ and T.

```
> T := S[1..L];
> LFSRSequence(C, T, #S);
[ 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0 ]
```

`ShrinkingGenerator(C1, S1, C2, S2, t)`

Outputs a sequence of t bits from the shrinking generator having connection polynomials C_1 and C_2 and initial states sequences S_1 and S_2 (thus, the lengths of the LFSRs are assumed to be the lengths of S_1 and S_2). Bits are represented as elements from $\mathbf{F}_2$. Polynomial coefficients and sequence elements must be from $\mathbf{F}_2$. The degrees of the connection polynomials must be at least 1 and their trailing coefficients must be 1. The number of elements in the initial states must be at least as large as the degrees of the corresponding connection polynomials.

165.3 Number Theoretic Bit Generators

`RandomSequenceRSA(b, t)`

Generates a sequence of t bits using the RSA pseudo-random bit generator with an RSA modulus of approximately b bits in length. The modulus n is computed by finding (pseudo-)random primes with the `RandomPrime` function. If $\gcd(\phi(n),\ 3)$ is 1, then the exponent 3 will be used. Otherwise, a (pseudo-)random exponent e is chosen so that $\gcd(\phi(n),\ e) = 1$. The seed is also chosen as a (pseudo-)random number modulo n. Bits are represented as elements of $\mathbf{F}_2$.

Example H165E2

The code below counts the number of 1's that appear in a sequence of 1000 bits generated from a 100-bit RSA modulus.

```
> Z := Integers();
> &+[ Z | b : b in RandomSequenceRSA(100, 1000) ];
497
```

`RandomSequenceRSA(n, e, s, t)`

Generates a sequence of t bits using the RSA pseudo-random bit generator with modulus n, exponent e, and seed value s. Bits are represented as elements from $\mathbf{F}_2$. The integer n must be larger than 1.

`RSAModulus(b)`

Returns an RSA Modulus n of b bits in length, and an exponent e such that `Gcd(EulerPhi(`n`),`e`)=1`. The resulting values can be used to generate random bits with the function `RandomSequenceRSA`. The argument b must be at least 16. *Warning*: RSA Moduli generated by Magma should not be used for real world cryptographic applications. Such applications require a "true random" source to seed the random number generator. Magma's method of seeding may not be sufficiently random to meet the requirements of cryptographic standards.

`RSAModulus(b, e)`

Returns an RSA Modulus n of b bits in length such that `Gcd(EulerPhi(`n`),`e`)=1`. The resulting value can be used with e for the exponent to generate random bits with the function `RandomSequenceRSA`. The argument b must be at least 16. The argument e must be odd and must also be in the range $1 < e < 2^b$. *Warning*: RSA Moduli generated by Magma should not be used for real world cryptographic applications. Such applications require a "true random" source to seed the random number generator. Magma's method of seeding may not be sufficiently random to meet the requirements of cryptographic standards.

`RandomSequenceBlumBlumShub(b, t)`

`BlumBlumShub(b, t)`

Generates a sequence of t bits using the Blum-Blum-Shub pseudo-random bit generator with a Blum-Blum-Shub modulus of approximately b bits in length. The modulus n is computed within Magma by finding (pseudo-)random primes with the `RandomPrime` function (the condition being that the primes are congruent to 3 mod 4). The seed is chosen as a (pseudo-)random number modulo n. Bits are represented as elements from $\mathbf{F}_2$. b must be at least 16.

`RandomSequenceBlumBlumShub(n, s, t)`

`BlumBlumShub(n, s, t)`

Generates a sequence of t bits using the Blum-Blum-Shub pseudo-random bit generator with modulus n and seed value s. Bits are represented as elements from $\mathbf{F}_2$. The argument n must be larger than 1 and gcd(s, n) must be 1.

```
BBSModulus(b)
```

```
BlumBlumShubModulus(b)
```

Returns a Blum-Blum-Shub Modulus b bits in length. The resulting value can be used to generate random bits with the function `RandomSequenceBlumBlumShub`. The argument b must be at least 16. *Warning*: Blum-Blum-Shub Moduli generated by Magma should not be used for real world cryptographic applications. Such applications require a "true random" source to seed the random number generator. Magma's method of seeding may not be sufficiently random to meet the requirements of cryptographic standards.

165.4 Correlation Functions

```
AutoCorrelation(S, t)
```

Computes the autocorrelation of a sequence S, where S must have universe $\mathbf{F}_2$. The autocorrelation is defined to be

$$C(t) = \sum_{i=1}^{L}(-1)^{S[i]+S[i+t]}$$

where L is the length of the sequence, and the values of $S[i+t]$ wrap around to the beginning of the sequence when $i+t > L$.

Example H165E3

It is well known that the LFSR's with maximal periods have nice autocorrelation properties. This is illustrated below.

```
> C<D> := PrimitivePolynomial (GF(2), 5);
> C;
D^5 + D^2 + 1
> s := [GF(2)|1,1,1,1,1];
> t := LFSRSequence(C, s, 31);
> t;
[ 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0,
1, 1, 0, 0, 0 ]
> AutoCorrelation (t, 2);
-1
```

```
CrossCorrelation(S1, S2, t)
```

Computes the crosscorrelation of two binary sequences S_1 and S_2, where S_1 and S_2 must each have universe $\mathbf{F}_2$, and they must have the same length L. The crosscorrelation is defined to be:

$$C(t) = \sum_{i=1}^{L} (-1)^{S_1[i]+S_2[i+t]}$$

and the values of $S_2[i + t]$ wrap around to the beginning of the sequence when $i + t > L$.

165.5 Decimation

```
Decimation(S, f, d)
```

Given a binary sequence S, and integers f and d, return the decimation of S. This is the sequence containing elements $S[f]$, $S[f + d]$, $S[f + 2d]$, ... where the indices in S are interpreted with wrap-around as integers between 1 and $\#S$.

```
Decimation(S, f, d, t)
```

Decimation of the sequence S. Returns a new sequence containing the first t elements of $S[f]$, $S[f + d]$, $S[f + 2d]$, ... where the indices in S are interpreted with wrap-around as integers between 1 and $\#S$.

Example H165E4

Given a primitive polynomial over $\mathbf{F}_q$, one can obtain another primitive polynomial by decimating an LFSR sequence obtained from the initial polynomial. This is demonstrated in the code below.

```
> K := GF(7);
> C<D> := PrimitivePolynomial(K, 2);
> C;
D^2 + 6*D + 3
```

In order to generate an LFSR sequence, we must first multiply this polynomial by a suitable constant so that the trailing coefficient becomes 1.

```
> C := C * Coefficient(C,0)^-1;
> C;
5*D^2 + 2*D + 1
```

We are now able to generate an LFSR sequence of length $7^2 - 1$. The initial state can be anything other than $[0, 0]$.

```
> t := LFSRSequence (C, [K| 1,1], 48);
> t;
[ 1, 1, 0, 2, 3, 5, 3, 4, 5, 5, 0, 3, 1, 4, 1, 6, 4, 4, 0, 1, 5, 6, 5, 2, 6, 6,
0, 5, 4, 2, 4, 3, 2, 2, 0, 4, 6, 3, 6, 1, 3, 3, 0, 6, 2, 1, 2, 5 ]
```

We decimate the sequence by a value d having the property $\gcd(d, 48) = 1$.

```
> t := Decimation(t, 1, 5);
> t;
[ 1, 5, 0, 6, 5, 6, 4, 4, 3, 1, 0, 4, 1, 4, 5, 5, 2, 3, 0, 5, 3, 5, 1, 1, 6, 2,
0, 1, 2, 1, 3, 3, 4, 6, 0, 3, 6, 3, 2, 2, 5, 4, 0, 2, 4, 2, 6, 6 ]
> B := BerlekampMassey(t);
> B;
3*D^2 + 5*D + 1
```

To get the corresponding primitive polynomial, we multiply by a constant to make it monic.

```
> B := B * Coefficient(B, 2)^-1;
> B;
D^2 + 4*D + 5
> IsPrimitive(B);
true
```

PART XXIII
OPTIMIZATION

166 LINEAR PROGRAMMING

Chapter 166
LINEAR PROGRAMMING

166.1 Introduction

A Linear Program in n variables $x_1, \cdots, x_n$ with m constraints of the form

$$\sum_{j=1}^{n} a_j x_j \ \leq \ c$$

(the relations in any of the constraints may also be $=$ or $\geq$) may be represented in matrix form as:

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \quad (\,REL\,) \quad \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$$

where $(\,REL\,)$ represents a componentwise relation between vectors, with each element $=$, $\leq$, or $\geq$.

Note that there is an additional implicit constraint, wherein all variables are assumed to be nonnegative.

We wish to find a solution $(\,x_i\,)$ that maximises (or minimises) the objective function:

$$\sum_{i=1}^{n} o_i x_i$$

Magma provides two methods for solving LP problems. The first is to set up suitable constraint matrices and then use an explicit LP solving function to solve the problem. The second involves creating an instance of the `LP` process, which is of category `LP`. Constraints are added and options set before calling `Solution` to get a solution to the problem.

All functions that actually solve an LP problem return a solution vector together with an integer code representing the state of the solution, provided by the lp_solve library. The codes are:

0 Optimal Solution

1 Failure

2 Infeasible problem

3 Unbounded problem

4 Failure

Magma supports LP problems over Integer, Rational, and Real rings. For Integer and Real problems, the solutions will be provided as Integer and Real vectors respectively. For LP problems provided in Rationals, the solution is a Real vector.

Linear programming in Magma is implemented using the lp_solve library written by Michel Berkelaar (michel@ics.ele.tue.nl). The library source may be found at `ftp://ftp.ics.ele.tue.nl/pub/lp_solve/`.

For further reference see [Naz87], [Chv83], [OH68] and [NW88].

166.2 Explicit LP Solving Functions

Each explicit LP solving function takes four arguments to represent an LP problem in n variables with m constraints:

1 `LHS` : $m \times n$ matrix, representing the left-hand-side coefficients of the m constraints.

2 `relations` : $m \times 1$ matrix over the same ring as `LHS`, representing the relations for each constraint, with a positive entry representing $\geq$, a zero entry representing $=$, and a negative entry representing $\leq$.

3 `RHS` : $m \times 1$ matrix over the same ring as `LHS`, representing the right-hand-side values of the m constraints.

4 `objective` : $1 \times n$ matrix over the same ring as `LHS`, representing the coefficients of the objective function to be optimised.

Each function returns a vector representing an optimal solution to the problem, and an integer indicating the state of the solution, as described in the introduction.

```
MaximalSolution(LHS, relations, RHS, objective)
```

The vector maximising the LP problem, with an integer describing the state of the solution.

```
MinimalSolution(LHS, relations, RHS, objective)
```

The vector minimising the LP problem, with an integer describing the state of the solution.

```
MaximalIntegerSolution(LHS, relations, RHS, objective)
```

The integer vector maximising the LP problem, with an integer describing the state of the solution.

```
MinimalIntegerSolution(LHS, relations, RHS, objective)
```

The integer vector minimising the LP problem, with an integer describing the state of the solution.

```
MaximalZeroOneSolution(LHS, relations, RHS, objective)
```

The vector with each entry either zero or one maximising the LP problem, with an integer describing the state of the solution.

```
MinimalZeroOneSolution(LHS, relations, RHS, objective)
```

The vector with each entry either zero or one minimising the LP problem, with an integer describing the state of the solution.

Example H166E1

We solve the LP maximising

$$F(x,y) = 8x + 2y \quad x, y \in \mathbf{R}$$

subject to the constraints

$$10x + 21y \leq 156$$

$$2x + y \leq 22$$

```
> R := RealField( );
> lhs := Matrix(R, 2, 2, [10, 21, 2, 1]);
> rhs := Matrix(R, 2, 1, [156, 22]);
> rel := Matrix(R, 2, 1, [-1, -1]); // negative values - less-or-equal relation
> obj := Matrix(R, 1, 2, [8, 15]);
> MaximalSolution(lhs, rel, rhs, obj);
[9.562500000000000000 2.875000000000000888]
0
```

Example H166E2

We find solutions to the LP maximising

$$F(x_1, \cdots, x_7) = 592x_1 + 381x_2 + 273x_3 + 55x_4 + 48x_5 + 37x_6 + 23x_7$$

subject to the constraint

$$3534x_1 + 2356x_2 + 2767x_3 + 589x_4 + 528x_5 + 451x_6 + 304x_7 \leq 119567$$

with $(x_1, \cdots, x_7)$ taking real values, integer values, and zero/one values.

```
> R := RealField( );
> lhs := Matrix(R, 1, 7, [3534, 2356, 2767, 589, 528, 451, 304]);
> rhs := Matrix(R, 1, 1, [119567]);
> rel := Matrix(R, 1, 1, [-1]);
> obj := Matrix(R, 1, 7, [592, 381, 273, 55, 48, 37, 23]);
> MaximalSolution(lhs, rel, rhs, obj);
[33.83333333333333570 0.E-92 0.E-92 0.E-92 0.E-92 0.E-92 0.E-92]
0
> MaximalIntegerSolution(lhs, rel, rhs, obj);
[33.00000000000000000 1.000000000000000000 0.E-92 1.000000000000000000 0.E-92
    0.E-92 0.E-92]
0
> MaximalZeroOneSolution(lhs, rel, rhs, obj);
[1.000000000000000000 1.000000000000000000 1.000000000000000000
    1.000000000000000000 1.000000000000000000 1.000000000000000000
    1.000000000000000000]
0
```

166.3 Creation of LP objects

`LPProcess(R, n)`

A Linear Program over the ring R in n variables.

Example H166E3

We create an LP representing a problem in 2 real variables:

```
> R := RealField( );
> L := LPProcess(R, 2);
> L;
LP <Real Field, 2 variables>
Minimising objective function: [0 0]
Subject to constraints:
Variables bounded above by: [ ]
Variables bounded below by: [ ]
Solving in integers for variables [ ]
```

166.4 Operations on LP objects

`AddConstraints(L, lhs, rhs)`

`Rel` MonStgElt *Default : "eq"*

Add some constraints to the LP problem L. All constraints will have the same relation, given by `Rel`, which may be set to `"eq"` for strict equality (the default), `"le"` for less-or-equal constraints, or `"ge"` for greater-or-equal constraints.

Constraints are of the form

$$\sum_{j=1}^{n} \mathtt{lhs}_{ij} \quad \mathtt{Rel} \quad \mathtt{rhs}_{i1}$$

where *lhs* and *rhs* are described in Section 166.2.

`NumberOfConstraints(L)`

The number of constraints in the LP problem L.

`NumberOfVariables(L)`

The number of variables in the LP problem L.

`EvaluateAt(L, p)`

Evaluate the objective function of the LP problem L at the point p given by a matrix.

```
Constraint(L, n)
```

The LHS, RHS and relation (-1 for $\leq$, 0 for $=$, 1 for $\geq$) of the n-th constraint of the LP problem L.

```
IntegerSolutionVariables(L)
```

Sequence of indices of the variables in the LP problem L to be solved in integers.

```
ObjectiveFunction(L)
```

The objective function of the LP problem L.

```
IsMaximisingFunction(L)
```

Returns `true` if the LP problem L is set to maximise its objective function, `false` if set to minimise.

```
RemoveConstraint(L, n)
```

Remove the n-th constraint from the LP problem L.

```
SetIntegerSolutionVariables(L, I, m)
```

Set the variables of the LP problem L indexed by elements of the sequence I to be solved in integers if m is `true`, or in the usual ring if `false`.

```
SetLowerBound(L, n, b)
```

Set the lower bound on the n-th variable in the LP problem L to b.

Note that for all LP problems in Magma there is an implicit constraint that all variables are ≥ 0. This constraint is overridden if a lower bound is specified by using this function (e.g., specifying a lower bound of -5 works as expected), but the lower bound can currently not be completely removed.

```
SetMaximiseFunction(L, m)
```

Set the LP problem L to maximise its objective function if m is `true`, or to minimise the objective function if m is false.

```
SetObjectiveFunction(L, F)
```

Set the objective function of the LP problem L to the matrix F.

```
SetUpperBound(L, n, b)
```

Set the upper bound on the n-th variable in the LP problem L to b.

```
Solution(L)
```

Solve the LP problem L; returns a point representing an optimal solution, and an integer representing the state of the solution.

```
UnsetBounds(L)
```

Remove any bounds on all variables in the LP problem L.

Note that this reactivates the implicit constraint that all variables are ≥ 0.

Example H166E4

We use an LP object to solve the LP maximising

$$F(x, y) = 3x + 13y$$

subject to constraints

$$2x + 9y <= 40$$

$$11x - 8y <= 82$$

```
> R := RealField( );
> L := LPProcess(R, 2);
> SetObjectiveFunction(L, Matrix(R, 1, 2, [3,13]));
> lhs := Matrix(R, 2, 2, [2, 9, 11, -8]);
> rhs := Matrix(R, 2, 1, [40, 82]);
> AddConstraints(L, lhs, rhs : Rel := "le");
> SetMaximiseFunction(L, true);
> L;
LP <Real Field, 2 variables>
Maximising objective function: [ 3 13]
Subject to constraints:
1 : [2 9] <= [40]
2 : [11 -8] <= [82]
Variables bounded above by: [ ]
Variables bounded below by: [ ]
Solving in integers for variables [ ]
> Solution(L);
[9.199999999999999289 2.400000000000000355]
0
```

Now, we place some bounds on y:

```
> SetUpperBound(L, 2, R!2);
> SetLowerBound(L, 2, R!1);
> Solution(L);
[8.909090909090908283 2.000000000000000000]
0
```

And find integer solutions:

```
> SetIntegerSolutionVariables(L, [1,2], true);
> Solution(L);
[8.000000000000000000 2.000000000000000000]
0
```

Now, removing the 2nd constraint:

```
> RemoveConstraint(L, 2);
> L;
LP <Real Field, 2 variables>
```

```
Maximising objective function: [ 3 13]
Subject to constraints:
1 : [2 9] <= [40]
Variables bounded above by: [ 2:2 ]
Variables bounded below by: [ 2:1 ]
Solving in integers for variables [ 1, 2 ]
> Solution(L);
[11.00000000000000000 2.000000000000000000]
0
```

And removing the restriction to Integer values for y,

```
> SetIntegerSolutionVariables(L, [2], false);
> Solution(L);
[15.00000000000000000 1.111111111111111160]
0
```

166.5 Bibliography

[**Chv83**] V. Chvatal. *Linear Programming.* W.H. Freeman and Company, 1983.

[**Naz87**] John Lawrence Nazareth. *Computer Solution of Linear Programs.* Oxford University Press, 1987.

[**NW88**] G.L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, Inc., 1988.

[**OH68**] W. Orchard-Hays. *Advanced linear–programming computing techniques.* McGraw–Hill, 1968.