GUAVA

A GAP4 Package

Version 1.6

by

Jasper Cramwinckel, Erik Roijackers, Reinald Baart, Eric Minkes

Translated from the GAP 3 version by

Lea Ruscio

Currently maintained by

W. D. Joyner

email: wdj@usna.edu

May 2002

Contents

1 About GUAVA

GUAVA is a GAP 4 package for computing with codes. Except for the automorphism group and isomorphism testing functions, which make use of J.S. Leon's partition backtrack programs, GUAVA is written in the GAP language. Several algorithms that need the speed were integrated in the GAP kernel. Please send your bug reports to the email address: `gap-trouble@dcs.st-and.ac.uk`

GUAVA is primarily designed for the construction and analysis of codes. The functions can be divided into three subcategories:

Construction of codes:
GUAVA can construct **unrestricted**, **linear** and **cyclic codes**. Information about the code is stored in a record, together with operations applicable to the code.

Manipulations of codes:
Manipulation transforms one code into another, or constructs a new code from two codes. The new code can profit from the data in the record of the old code(s), so in these cases calculation time decreases.

Computations of information about codes:
GUAVA can calculate important data of codes very fast. The results are stored in the code record.

1.1 Acknowledgements

GUAVA was written by Jasper Cramwinckel, Erik Roijackers, and Reinald Baart. as a final project during their study of Mathematics at the Delft University of Technology, department of Pure Mathematics, and in Aachen, at Lehrstuhl D fuer Mathematik.

In version 1.3, new functions were added by Eric Minkes, also from Delft University of Technology.

JC,ER and RB would like to thank the GAP people at the RWTH Aachen for their support, A.E. Brouwer for his advice and J. Simonis for his supervision.

The GAP 4 version of GUAVA was created by Lea Ruscio and is maintained by David Joyner.

1.2 Installing GUAVA

To install GUAVA (as a GAP4 Package) unpack the archive file in a directory in the `pkg` hierarchy of your version of GAP4. (This might be the `pkg` directory of the GAP4 home directory; it is however also possible to keep an additional `pkg` directory in your private directories, see section 74.2 of the GAP4 reference manual for details on how to do this.)

After unpacking GUAVA the GAP-only part of GUAVA is installed. The parts of GUAVA depending on J. Leon's backtrack programs package (for computing automorphism groups) are only available in a UNIX environment, where you should proceed as follows:

Go to the newly created `guava` directory and call `./configure` *path* where *path* is the path to the GAP home directory. So for example, if you install the package in the main `pkg` directory call

```
./configure ../..
```

This will fetch the architecture type for which GAP has been compiled last and create a `Makefile`. Now call

```
make
```

to compile the binary and to install it in the appropriate place.

This completes the installation of GUAVA for a single architecture. If you use this installation of GUAVA on different hardware platforms you will have to compile the binary for each platform separately. This is done by calling `configure` and `make` for the package anew immediately after compiling GAP itself for the respective architecture. If your version of GAP is already compiled (and has last been compiled on the same architecture) you do not need to compile GAP again; it is sufficient to call the `configure` script in the GAP home directory.

1.3 Loading GUAVA

After starting up GAP, the GUAVA package needs to be loaded. Load GUAVA by typing at the GAP prompt:

```
gap> RequirePackage( "guava" );
```

If GUAVA isn't already in memory, it is loaded and its beautiful banner is displayed.

If you are a frequent user of GUAVA, you might consider putting this line in your `.gaprc` file.

2 Codewords

A **codeword** is basically just a vector of finite field elements. In GUAVA, a codeword is a record, with this base vector as its most important element.

Codewords have been implemented in GUAVA mainly because of their easy interfacing with the user. The user can input codewords in different formats, and output information is formatted in a readable way.

Codewords work together with codes (see 3), although many operations are available on codewords themselves.

The first section describes how codewords are constructed (see 2.1.2 and 2.1.3).

Sections 2.2 and 2.3 describe the arithmetic operations applicable to codewords.

Section 2.4 describe functions that convert codewords back to vectors or polynomials (see 2.4.1 and 2.4.2).

Section 2.5 describe functions that change the way a codeword is displayed (see 2.5.1 and 2.5.2).

Finally, Section 2.6 describes a function to generate a null word (see 2.6.1) and some functions for extarcting properties of codewords (see 2.6.2, 2.6.3 and 2.6.4).

2.1 Construction of Codewords

1 ► `Codeword(` *obj* `[,` *n*`] [,` *F*`] )`

`Codeword` returns a codeword or a list of codewords constructed from *obj*. The object *obj* can be a vector, a string, a polynomial or a codeword. It may also be a list of those (even a mixed list).

If a number n is specified, all constructed codewords have length n. This is the only way to make sure that all elements of *obj* are converted to codewords of the same length. Elements of *obj* that are longer than n are reduced in length by cutting of the last positions. Elements of *obj* that are shorter than n are lengthened by adding zeros at the end. If no n is specified, each constructed codeword is handled individually.

If a Galois field F is specified, all codewords are constructed over this field. This is the only way to make sure that all elements of obj are converted to the same field F (otherwise they are converted one by one). Note that all elements of *obj* must have elements over F or over `Integers`. Converting from one Galois field to another is not allowed. If no F is specified, vectors or strings with integer elements will be converted to the smallest Galois field possible.

Note that a significant speed increase is achieved if F is specified, even when all elements of *obj* already have elements over F.

Every vector in *obj* can be a finite field vector over F or a vector over `Integers`. In the last case, it is converted to F or, if omitted, to the smallest Galois field possible.

Every string in *obj* must be a string of numbers, without spaces, commas or any other characters. These numbers must be from 0 to 9. The string is converted to a codeword over F or, if F is omitted, over the smallest Galois field possible. Note that since all numbers in the string are interpreted as one-digit numbers, Galois fields of size larger than 10 are not properly represented when using strings.

Every polynomial in *obj* is converted to a codeword of length n or, if omitted, of a length dictated by the degree of the polynomial. If F is specified, a polynomial in *obj* must be over F.

Every element of *obj* that is already a codeword is changed to a codeword of length n. If no n was specified, the codeword doesn't change. If F is specified, the codeword must have base field F.

```
gap> c := Codeword([0,1,1,1,0]);
[ 0 1 1 1 0 ]
gap> VectorCodeword( c );
[ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> c2 := Codeword([0,1,1,1,0], GF(3));
[ 0 1 1 1 0 ]
gap> VectorCodeword( c2 );
[ 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0, 0*Z(3) ]
gap> Codeword([c, c2, "0110"]);
[ [ 0 1 1 1 0 ], [ 0 1 1 1 0 ], [ 0 1 1 0 ] ]
gap> p := UnivariatePolynomial(GF(2), [Z(2)^0, 0*Z(2), Z(2)^0]);
Z(2)^0+x_1^2
gap> Codeword(p);
x^2 + 1
```

2 ► `Codeword(` *obj*`,` *C* `)`

In this format, the elements of *obj* are converted to elements of the same vector space as the elements of a code C. This is the same as calling `Codeword` with the word length of C (which is n) and the field of C (which is F).

```
gap> C := WholeSpaceCode(7,GF(5));
a cyclic [7,7,1]0 whole space code over GF(5)
gap> Codeword(["0220110", [1,1,1]], C);
[ [ 0 2 2 0 1 1 0 ], [ 1 1 1 0 0 0 0 ] ]
gap> Codeword(["0220110", [1,1,1]], 7, GF(5));
[ [ 0 2 2 0 1 1 0 ], [ 1 1 1 0 0 0 0 ] ]
```

3 ► `IsCodeword(` *obj* `)`

`IsCodeword` returns `true` if *obj*, which can be an object of arbitrary type, is of the codeword type and `false` otherwise. The function will signal an error if *obj* is an unbound variable.

```
gap> IsCodeword(1);
false
gap> IsCodeword(ReedMullerCode(2,3));
false
gap> IsCodeword("11111");
false
gap> IsCodeword(Codeword("11111"));
true
```

2.2 Comparisons of Codewords

1 ► c_1 `=` c_2
► c_1 `<>` c_2

The equality operator `=` evaluates to `true` if the codewords c_1 and c_2 are equal, and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the codewords c_1 and c_2 are not equal, and to `false` otherwise.

Note that codewords are equal if and only if their base vectors are equal. Whether they are represented as a vector or polynomial has nothing to do with the comparison.

Comparing codewords with objects of other types is not recommended, although it is possible. If c_2 is the codeword, the other object c_1 is first converted to a codeword, after which comparison is possible. This way, a codeword can be compared with a vector, polynomial, or string. If c_1 is the codeword, then problems may arise if c_2 is a polynomial. In that case, the comparison always yields a `false`, because the polynomial comparison is called (see `Comparisons of Polynomials`).

```
gap> P := UnivariatePolynomial(GF(2), Z(2)*[1,0,0,1]);
Z(2)^0+x_1^3
gap> c := Codeword(P, GF(2));
x^3 + 1
gap> P = c;        # codeword operation
true
gap> c2 := Codeword("1001", GF(2));
[ 1 0 0 1 ]
gap> c = c2;
true
```

2.3 Arithmetic Operations for Codewords

The following operations are always available for codewords. The operands must have a common base field, and must have the same length. No implicit conversions are performed.

1 ► c_1 `+` c_2

The operator `+` evaluates to the sum of the codewords c_1 and c_2.

2 ► c_1 `-` c_2

The operator `-` evaluates to the difference of the codewords c_1 and c_2.

3 ► C `+` c
► c `+` C

The operator `+` evaluates to the coset code of code C after adding a codeword c to all codewords. See 5.1.18.

In general, the operations just described can also be performed on vectors, strings or polynomials, although this is not recommended. The vector, string or polynomial is first converted to a codeword, after which the normal operation is performed. For this to go right, make sure that at least one of the operands is a codeword. Further more, it will not work when the right operand is a polynomial. In that case, the polynomial operations ('FiniteFieldPolynomialOps') are called, instead of the codeword operations ('CodewordOps').

Some other code-oriented operations with codewords are described in 3.2.

2.4 Functions that Convert Codewords to Vectors or Polynomials

1 ► `VectorCodeword(` *obj* `)`

Here *obj* can be a code word or a list of code words. This function retyurns the corresponding vectors over a finite field.

```
gap> a := Codeword("011011");; VectorCodeword(a);
[ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0 ]
```

2 ► `PolyCodeword(` *obj* `)`

`PolyCodeword` returns a polynomial or a list of polynomials over a Galois field, converted from *obj*. The object *obj* can be a code word, or a list of codewords.

```
gap> a := Codeword("011011");; PolyCodeword(a);
x_1+x_1^2+x_1^4+x_1^5
```

2.5 Functions that Change the Display Form of a Codeword

1 ▶ `TreatAsVector( obj )`

`TreatAsVector` adapts the codewords in *obj* to make sure they are printed as vectors. *obj* may be a codeword or a list of codewords. Elements of *obj* that are not codewords are ignored. After this function is called, the codewords will be treated as vectors. The vector representation is obtained by using the coefficient list of the polynomial.

Note that this only changes the way a codeword is printed. `TreatAsVector` returns nothing, it is called only for its side effect. The function `VectorCodeword` converts codewords to vectors (see 2.4.1).

```
gap> B := BinaryGolayCode();
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> c := CodewordNr(B, 4);
x^22 + x^20 + x^17 + x^14 + x^13 + x^12 + x^11 + x^10
gap> TreatAsVector(c);
gap> c;
[ 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 0 1 0 1 ]
```

2 ▶ `TreatAsPoly( obj )`

`TreatAsPoly` adapts the codewords in *obj* to make sure they are printed as polynomials. *obj* may be a codeword or a list of codewords. Elements of *obj* that are not codewords are ignored. After this function is called, the codewords will be treated as polynomials. The finite field vector that defines the codeword is used as a coefficient list of the polynomial representation, where the first element of the vector is the coefficient of degree zero, the second element is the coefficient of degree one, etc, until the last element, which is the coefficient of highest degree.

Note that this only changes the way a codeword is printed. `TreatAsPoly` returns nothing, it is called only for its side effect. The function `PolyCodeword` converts codewords to polynomials (see 2.4.2).

```
gap> a := Codeword("00001",GF(2));
[ 0 0 0 0 1 ]
gap> TreatAsPoly(a); a;
x^4
gap> b := NullWord(6,GF(4));
[ 0 0 0 0 0 0 ]
gap> TreatAsPoly(b); b;
0
```

2.6 Other Codeword Functions

1 ▶ `NullWord( n )`
▶ `NullWord( n, F )`
▶ `NullWord( C )`

`NullWord` returns a codeword of length n over the field F of only zeros. The default for F is `GF(2)`. n must be greater then zero. If only a code C is specified, `NullWord` will return a null word with the word length and the Galois field of C.

```
gap> NullWord(8);
[ 0 0 0 0 0 0 0 0 ]
gap> Codeword("0000") = NullWord(4);
true
gap> NullWord(5,GF(16));
[ 0 0 0 0 0 ]
gap> NullWord(ExtendedTernaryGolayCode());
[ 0 0 0 0 0 0 0 0 0 0 0 0 ]
```

2 ► `DistanceCodeword(` c_1, c_2 `)`

`DistanceCodeword` returns the Hamming distance from c_1 to c_2. Both variables must be codewords with equal word length over the same Galois field. The Hamming distance between two words is the number of places in which they differ. As a result, `DistanceCodeword` always returns an integer between zero and the word length of the codewords.

```
gap> a := Codeword([0, 1, 2, 0, 1, 2]);; b := NullWord(6, GF(3));;
gap> DistanceCodeword(a, b);
4
gap> DistanceCodeword(b, a);
4
gap> DistanceCodeword(a, a);
0
```

3 ► `Support(` c `)`

`Support` returns a set of integers indicating the positions of the non-zero entries in a codeword c.

```
gap> a := Codeword("012320023002");; Support(a);
[ 2, 3, 4, 5, 8, 9, 12 ]
gap> Support(NullWord(7));
[  ]
```

The support of a list with codewords can be calculated by taking the union of the individual supports. The weight of the support is the length of the set.

```
gap> L := Codeword(["000000", "101010", "222000"], GF(3));;
gap> S := Union(List(L, i -> Support(i)));
[ 1, 2, 3, 5 ]
gap> Length(S);
4
```

4 ► `WeightCodeword(` c `)`

`WeightCodeword` returns the weight of a codeword c, the number of non-zero entries in c. As a result, `WeightCodeword` always returns an integer between zero and the word length of the codeword.

```
gap> WeightCodeword(Codeword("22222"));
5
gap> WeightCodeword(NullWord(3));
0
gap> C := HammingCode(3);
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> Minimum(List(AsSSortedList(C){[2..Size(C)]}, WeightCodeword ) );
3
```

3 Codes

A **code** basically is nothing more than a set of **codewords**. We call these the **elements** of the code. A codeword is a sequence of elements of a finite field `GF(q)` where q is a prime power. Depending on the type of code, a codeword can be interpreted as a vector or as a polynomial. This is explained in more detail in Chapter 2.

In GUAVA, codes can be defined by their elements (this will be called an **unrestricted code**), by a generator matrix (a **linear code**) or by a generator polynomial (a **cyclic code**).

Any code can be defined by its elements. If you like, you can give the code a name.

```
gap> C := ElementsCode(["1100", "1010", "0001"], "example code",
>                       GF(2) );
a (4,3,1..4)2..4 example code over GF(2)
```

An (n,M,d) code is a code with **word length** n, **size** M and **minimum distance** d. If the minimum distance has not yet been calculated, the lower bound and upper bound are printed. So

```
a (4,3,1..4)2..4 code over GF(2)
```

means a binary unrestricted code of length 4, with 3 elements and the minimum distance is greater than or equal to 1 and less than or equal to 4 and the **covering radius** is greater than or equal to 2 and less than or equal to 4.

```
gap> MinimumDistance(C);
2
gap> C;
a (4,3,2)2..4 example code over GF(2)
```

If the set of elements is a linear subspace of $GF(q)^n$, the code is called **linear**. If a code is linear, it can be defined by its **generator matrix** or **parity check matrix**. The generator matrix is a basis for the elements of a code, the parity check matrix is a basis for the nullspace of the code.

```
gap> G := GeneratorMatCode([[1,0,1],[0,1,2]], "demo code", GF(3) );
a linear [3,2,1..2]1 demo code over GF(3)
```

So a linear $[n,k,d]r$ code is a code with **word length** n, **dimension** k, **minimum distance** d and **covering radius** r.

If the code is linear and all cyclic shifts of its elements are again codewords, the code is called **cyclic**. A cyclic code is defined by its **generator polynomial** or **check polynomial**. All elements are multiples of the generator polynomial modulo a polynomial $x^n - 1$ where n is the word length of the code. Multiplying a code element with the check polynomial yields zero (modulo the polynomial $x^n - 1$).

```
gap> G := GeneratorPolCode(Indeterminate(GF(2))+Z(2)^0, 7, GF(2) );
a cyclic [7,6,1..2]1 code defined by generator polynomial over GF(2)
```

It is possible that GUAVA does not know that an unrestricted code is linear. This situation occurs for example when a code is generated from a list of elements with the function `ElementsCode` (see 4.1.1). By calling the function `IsLinearCode` (see 3.3.2), GUAVA tests if the code can be represented by a generator matrix. If so, the code record and the operations are converted accordingly.

```
gap> L := Z(2)*[ [0,0,0], [1,0,0], [0,1,1], [1,1,1] ];;
gap> C := ElementsCode( L, GF(2) );
a (3,4,1..3)1 user defined unrestricted code over GF(2)
# so far, {\GUAVA} does not know what kind of code this is
gap> IsLinearCode( C );
true                       # it is linear
gap> C;
a linear [3,2,1]1 user defined unrestricted code over GF(2)
```

Of course the same holds for unrestricted codes that in fact are cyclic, or codes, defined by a generator matrix, that in fact are cyclic.

Codes are printed simply by giving a small description of their parameters, the word length, size or dimension and minimum distance, followed by a short description and the base field of the code. The function `Display` gives a more detailed description, showing the construction history of the code.

GUAVA doesn't place much emphasis on the actual encoding and decoding processes; some algorithms have been included though. Encoding works simply by multiplying an information vector with a code, decoding is done by the function `Decode`. For more information about encoding and decoding, see sections 3.2 and 3.10.1.

```
gap> R := ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> w := [ 1, 0, 1, 1 ] * R;
[ 1 0 0 1 1 0 0 1 ]
gap> Decode( R, w );
[ 1 0 1 1 ]
gap> Decode( R, w + "10000000" ); # One error at the first position
[ 1 0 1 1 ]                       # Corrected by Guava
```

Sections 3.1 and 3.2 describe the operations that are available for codes.

Section 3.3 describe the functions that tests whether an object is a code and what kind of code it is (see 3.3.1, 3.3.2 and 3.3.3) and various other boolean functions for codes.

Section 3.4 describe functions about equivalence and isomorphism of codes (see 3.4.1, 3.4.2 and 3.4.3).

Section 3.5 describes functions that work on **domains** (see Chapter 30 in the GAP Reference Manual).

Section 3.6 describes functions for printing and displaying codes.

Section 3.7 describes functions that return the matrices and polynomials that define a code (see 3.7.1, 3.7.2, 3.7.3, 3.7.4, 3.7.5).

Section 3.8 describes functions that return the basic parameters of codes (see 3.8.1, 3.8.2 and 3.8.4).

Section 3.9 describes functions that return distance and weight distributions (see 3.9.1, 3.9.2, 3.9.3 and 3.9.4).

Section 3.10 describes functions that are related to decoding (see 3.10.1, 3.10.2, 3.10.3 and 3.10.4).

In Chapters 4 and 5 we follow on, describing functions that generate and manipulate codes.

3.1 Comparisons of Codes

1 ► *C_1* `=` *C_2*
► *C_1* `<>` *C_2*

The equality operator `=` evaluates to `true` if the codes *C_1* and *C_2* are equal, and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the codes *C_1* and *C_2* are not equal, and to `false` otherwise.

Note that codes are equal if and only if their elements are equal. Codes can also be compared with objects of other types. Of course they are never equal.

```
gap> M := [ [0, 0], [1, 0], [0, 1], [1, 1] ];;
gap> C1 := ElementsCode( M, GF(2) );
a (2,4,1..2)0 user defined unrestricted code over GF(2)
gap> M = C1;
false
gap> C2 := GeneratorMatCode( [ [1, 0], [0, 1] ], GF(2) );
a linear [2,2,1]0 code defined by generator matrix over GF(2)
gap> C1 = C2;
true
gap> ReedMullerCode( 1, 3 ) = HadamardCode( 8 );
true
gap> WholeSpaceCode( 5, GF(4) ) = WholeSpaceCode( 5, GF(2) );
false
```

Another way of comparing codes is `IsEquivalent`, which checks if two codes are equivalent (see 3.4.1).

3.2 Operations for Codes

1 ► *C_1* `+` *C_2*

The operator `+` evaluates to the direct sum of the codes *C_1* and *C_2*. See 5.2.1.

2 ► C `+` c
► c `+` C

The operator `+` evaluates to the coset code of code C after adding c to all elements of C. See 5.1.18.

3 ► *C_1* `*` *C_2*

The operator `*` evaluates to the direct product of the codes *C_1* and *C_2*. See 5.2.3.

4 ► x `*` C

The operator `*` evaluates to the element of C belonging to information word x. x may be a vector, polynomial, string or codeword or a list of those. This is the way to do encoding in GUAVA. C must be linear, because in GUAVA, encoding by multiplication is only defined for linear codes. If C is a cyclic code, this multiplication is the same as multiplying an information polynomial x by the generator polynomial of C (except for the result not being a codeword type). If C is a linear code, it is equal to the multiplication of an information vector x by the generator matrix of C (again, the result then is not a codeword type).

To decode, use the function `Decode` (see 3.10.1).

5 ► c `in` C

The `in` operator evaluates to `true` if C contains the codeword or list of codewords specified by c. Of course, c and C must have the same word lengths and base fields.

```
gap> C:= HammingCode( 2 );; eC:= AsSSortedList( C );
[ [ 0 0 0 ], [ 1 1 1 ] ]
gap> eC[2] in C;
true
gap> [ 0 ] in C;
false
```

6 ► IsSubset(C_1,C_2)

returns `true` if C_2 is a subcode of C_1, i.e. if C_1 contains at least all the elements of C_2.

```
gap> IsSubset( HammingCode(3), RepetitionCode( 7 ) );
true
gap> IsSubset( RepetitionCode( 7 ), HammingCode( 3 ) );
false
gap> IsSubset( WholeSpaceCode( 7 ), HammingCode( 3 ) );
true
```

3.3 Boolean Functions for Codes

1 ► IsCode(*obj*)

`IsCode` returns `true` if *obj*, which can be an object of arbitrary type, is a code and `false` otherwise. Will cause an error if *obj* is an unbound variable.

```
gap> IsCode( 1 );
false
gap> IsCode( ReedMullerCode( 2,3 ) );
true
```

2 ► IsLinearCode(*obj*)

`IsLinearCode` checks if object *obj* (not necessarily a code) is a linear code. If a code has already been marked as linear or cyclic, the function automatically returns `true`. Otherwise, the function checks if a basis G of the elements of *obj* exists that generates the elements of *obj*. If so, G is a generator matrix of *obj* and the function returns `true`. If not, the function returns `false`.

```
gap> C := ElementsCode( [ [0,0,0],[1,1,1] ], GF(2) );
a (3,2,1..3)1 user defined unrestricted code over GF(2)
gap> IsLinearCode( C );
true
gap> IsLinearCode( ElementsCode( [ [1,1,1] ], GF(2) ) );
false
gap> IsLinearCode( 1 );
false
```

3 ► IsCyclicCode(*obj*)

`IsCyclicCode` checks if the object *obj* is a cyclic code. If a code has already been marked as cyclic, the function automatically returns `true`. Otherwise, the function checks if a polynomial g exists that generates the elements of *obj*. If so, g is a generator polynomial of *obj* and the function returns `true`. If not, the function returns `false`.

```
gap> C := ElementsCode( [ [0,0,0], [1,1,1] ], GF(2) );
a (3,2,1..3)1 user defined unrestricted code over GF(2)
gap> # GUAVA does not know the code is cyclic
gap> IsCyclicCode( C );      # this command tells GUAVA to find out
true
gap> IsCyclicCode( HammingCode( 4, GF(2) ) );
false
gap> IsCyclicCode( 1 );
false
```

4 ► `IsPerfectCode(` *C* `)`

`IsPerfectCode` returns `true` if C is a perfect code. For a code with odd minimum distance $d = 2t+1$, this is the case when every word of the vector space of C is at distance at most t from exactly one element of C. Codes with even minimum distance are never perfect.

In fact, a code that is not **trivial perfect** (the binary repetition codes of odd length, the codes consisting of one word, and the codes consisting of the whole vector space), and does not have the parameters of a Hamming- or Golay-code, cannot be perfect.

```
gap> H := HammingCode(2);
a linear [3,1,3]1 Hamming (2,2) code over GF(2)
gap> IsPerfectCode( H );
true
gap> IsPerfectCode( ElementsCode( [ [1,1,0], [0,0,1] ], GF(2) ) );
true
gap> IsPerfectCode( ReedSolomonCode( 6, 3 ) );
false
gap> IsPerfectCode(BinaryGolayCode());
true
```

5 ► `IsMDSCode(` *C* `)`

`IsMDSCode` returns true if C is a **Maximum Distance Seperable code**, or MDS code for short. A linear $[n,k,d]$-code of length n, dimension k and minimum distance d is an MDS code if $k = n - d + 1$, in other words if C meets the Singleton bound (see 6.1.1). An unrestricted (n,M,d) code is called MDS if $k = n - d + 1$, with k equal to the largest integer less than or equal to the logarithm of M with base q, the size of the base field of C.

Well known MDS codes include the repetition codes, the whole space codes, the even weight codes (these are the only **binary** MDS Codes) and the Reed-Solomon codes.

```
gap> C1 := ReedSolomonCode( 6, 3 );
a cyclic [6,4,3]2 Reed-Solomon code over GF(7)
gap> IsMDSCode( C1 );
true    # 6-3+1 = 4
gap> IsMDSCode( QRCode( 23, GF(2) ) );
false
```

6 ► `IsSelfDualCode(` *C* `)`

`IsSelfDualCode` returns `true` if C is self-dual, i.e. when C is equal to its dual code (see also 5.1.16). If a code is self-dual, it automatically is self-orthogonal (see 3.3.7).

If C is a non-linear code, it cannot be self-dual, so `false` is returned. A linear code can only be self-dual when its dimension k is equal to the redundancy r.

```
gap> IsSelfDualCode( ExtendedBinaryGolayCode() );
true
gap> C := ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> DualCode( C ) = C;
true
```

7 ► `IsSelfOrthogonalCode(` C `)`

`IsSelfOrthogonalCode` returns `true` if C is **self-orthogonal**. A code is self-orthogonal if every element of C is orthogonal to all elements of C, including itself. In the linear case, this simply means that the generator matrix of C multiplied with its transpose yields a null matrix.

In addition, a code is **self-dual** if it contains all vectors that its elements are orthogonal to (see 3.3.6).

```
gap> R := ReedMullerCode(1,4);
a linear [16,5,8]6 Reed-Muller (1,4) code over GF(2)
gap> IsSelfOrthogonalCode(R);
true
gap> IsSelfDualCode(R);
false
```

3.4 Equivalence and Isomorphism of Codes

1 ► `IsEquivalent(` C_1, C_2 `)`

`IsEquivalent` returns true if C_1 and C_2 are equivalent codes. This is the case if C_1 can be obtained from C_2 by carrying out column permutations. GUAVA only handles binary codes. The external program `desauto` from **J.S. Leon** is used to compute the isomorphism between the codes. If C_1 and C_2 are equal, they are also equivalent.

Note that the algorithm is **very** slow for non-linear codes.

```
gap> x:= Indeterminate( GF(2) );; pol:= x^3+x+1;
Z(2)^0+x_1+x_1^3
gap> H := GeneratorPolCode( pol, 7, GF(2));
a cyclic [7,4,1..3]1 code defined by generator polynomial over GF(2)
gap> H = HammingCode(3, GF(2));
false
gap> IsEquivalent(H, HammingCode(3, GF(2)));
true                         # H is equivalent to a Hamming code
gap> CodeIsomorphism(H, HammingCode(3, GF(2)));
(3,4)(5,6,7)
```

2 ► `CodeIsomorphism(` C_1, C_2 `)`

If the two codes C_1 and C_2 are equivalent codes (see 3.4.1), `CodeIsomorphism` returns the permutation that transforms C_1 into C_2. If the codes are not equivalent, it returns `false`.

```
gap> x:= Indeterminate( GF(2) );; pol:= x^3+x+1;
Z(2)^0+x_1+x_1^3
gap> H := GeneratorPolCode( pol, 7, GF(2));
a cyclic [7,4,1..3]1 code defined by generator polynomial over GF(2)
gap> CodeIsomorphism(H, HammingCode(3, GF(2)));
(3,4)(5,6,7)
gap> PermutedCode(H, (3,4)(5,6,7)) = HammingCode(3, GF(2));
true
```

3 ► `AutomorphismGroup(` C `)`

`AutomorphismGroup` returns the **automorphism group** of a binary code C. This is the largest permutation group of degree n such that each permutation applied to the columns of C again yields C. GUAVA uses the external program `desauto` from **J.S. Leon** to compute the automorphism group. The function `PermutedCode` permutes the columns of a code (see 5.1.5).

```
gap> R := RepetitionCode(7,GF(2));
a cyclic [7,1,7]3 repetition code over GF(2)
gap> AutomorphismGroup(R);
Sym( [ 1 .. 7 ] )
                        # every permutation keeps R identical
gap> C := CordaroWagnerCode(7);
a linear [7,2,4]3 Cordaro-Wagner code over GF(2)
gap> AsSSortedList(C);
[ [ 0 0 0 0 0 0 0 ], [ 0 0 1 1 1 1 1 ], [ 1 1 0 0 0 1 1 ], [ 1 1 1 1 1 0 0 ] ]
gap> AutomorphismGroup(C);
Group([ (3,4), (4,5), (1,6)(2,7), (1,2), (6,7) ])
gap> C2 :=  PermutedCode(C, (1,6)(2,7));
a linear [7,2,4]3 permuted code
gap> AsSSortedList(C2);
[ [ 0 0 0 0 0 0 0 ], [ 0 0 1 1 1 1 1 ], [ 1 1 0 0 0 1 1 ], [ 1 1 1 1 1 0 0 ] ]
gap> C2 = C;
true
```

3.5 Domain Functions for Codes

These are some GAP functions that work on `Domains` in general. Their specific effect on `Codes` is explained here.

1 ► `IsFinite(` C `)`

`IsFinite` is an implementation of the GAP domain function `IsFinite`. It returns true for a code C.

```
gap> IsFinite( RepetitionCode( 1000, GF(11) ) );
true
```

2 ► `Size(` C `)`

`Size` returns the size of C, the number of elements of the code. If the code is linear, the size of the code is equal to q^k, where q is the size of the base field of C and k is the dimension.

```
gap> Size( RepetitionCode( 1000, GF(11) ) );
11
gap> Size( NordstromRobinsonCode() );
256
```

3 ► `LeftActingDomain(` C `)`

`LeftActingDomain` returns the base field of a code C. Each element of C consists of elements of this base field. If the base field is F, and the word length of the code is n, then the codewords are elements of F^n. If C is a cyclic code, its elements are interpreted as polynomials with coefficients over F.

```
gap> C1 := ElementsCode([[0,0,0], [1,0,1], [0,1,0]], GF(4));
a (3,3,1..3)2..3 user defined unrestricted code over GF(4)
gap> LeftActingDomain( C1 );
GF(2^2)
gap> LeftActingDomain( HammingCode( 3, GF(9) ) );
GF(3^2)
```

4 ▶ `Dimension(` C `)`

`Dimension` returns the parameter k of C, the dimension of the code, or the number of information symbols in each codeword. The dimension is not defined for non-linear codes; `Dimension` then returns an error.

```
gap> Dimension( NullCode( 5, GF(5) ) );
0
gap> C := BCHCode( 15, 4, GF(4) );
a cyclic [15,9,5]3..4 BCH code, delta=5, b=1 over GF(4)
gap> Dimension( C );
9
gap> Size( C ) = Size( LeftActingDomain( C ) ) ^ Dimension( C );
true
```

5 ▶ `AsSSortedList(` C `)`

`AsSSortedList` returns a list of the elements of C. These elements are of the codeword type (see 2). Note that for large codes, generating the elements may be very time- and memory-consuming. For generating a specific element or a subset of the elements, use `CodewordNr` (see 3.5.6).

```
gap> C := ConferenceCode( 5 );
a (5,12,2)1..4 conference code over GF(2)
gap> AsSSortedList( C );
[ [ 0 0 0 0 0 ], [ 0 0 1 1 1 ], [ 0 1 0 1 1 ], [ 0 1 1 0 1 ], [ 0 1 1 1 0 ],
  [ 1 0 0 1 1 ], [ 1 0 1 0 1 ], [ 1 0 1 1 0 ], [ 1 1 0 0 1 ], [ 1 1 0 1 0 ],
  [ 1 1 1 0 0 ], [ 1 1 1 1 1 ] ]
gap> CodewordNr( C, [ 1, 2 ] );
[ [ 0 0 0 0 0 ], [ 0 0 1 1 1 ] ]
```

6 ▶ `CodewordNr(` C`,` *list* `)`

`CodewordNr` returns a list of codewords of C. *list* may be a list of integers or a single integer. For each integer of *list*, the corresponding codeword of C is returned. The correspondence of a number i with a codeword is determined as follows: if a list of elements of C is available, the i^{th} element is taken. Otherwise, it is calculated by multiplication of the i^{th} information vector by the generator matrix or generator polynomial, where the information vectors are ordered lexicographically.

So `CodewordNr(`C`,` i`)` is equal to `AsSSortedList(`C`)[`i`]`. The latter function first calculates the set of all the elements of C and then returns the i^{th} element of that set, whereas the former only calculates the i^{th} codeword.

```
gap> R := ReedSolomonCode(2,2);
a cyclic [2,1,2]1 Reed-Solomon code over GF(3)
gap> AsSSortedList(R);
[ [ 0 0 ], [ 1 1 ], [ 2 2 ] ]
gap> CodewordNr(R, [1,3]);
[ [ 0 0 ], [ 2 2 ] ]
gap> C := HadamardCode( 16 );
a (16,32,8)5..6 Hadamard code of order 16 over GF(2)
gap> AsSSortedList(C)[17] = CodewordNr( C, 17 );
true
```

3.6 Printing and Displaying Codes

1 ► `Print(` *C* `)`

`Print` prints information about C. This is the same as typing the identifier C at the GAP-prompt.

If the argument is an unrestricted code, information in the form

```
a (<n>,<M>,<d>)<r> ... code over GF(q)
```

is printed, where n is the word length, M the number of elements of the code, d the minimum distance and r the covering radius.

If the argument is a linear code, information in the form

```
a linear [<n>,<k>,<d>]<r> ... code over GF(q)
```

is printed, where n is the word length, k the dimension of the code, d the minimum distance and r the covering radius.

In all cases, if d is not yet known, it is displayed in the form

```
<lowerbound>,..,<upperbound>
```

and if r is not yet known, it is displayed in the same way.

The function `Display` gives more information. See 3.6.3.

```
gap> C1 := ExtendedCode( HammingCode( 3, GF(2) ) );
a linear [8,4,4]2 extended code
gap> Print( "This is ", NordstromRobinsonCode(), ". \n");
This is a (16,256,6)4 Nordstrom-Robinson code over GF(2).
```

2 ► `String(` *C* `)`

`String` returns information about C in a string. This function is used by `Print` (see `Print`).

3 ► `Display(` *C* `)`

`Display` prints the method of construction of code C. With this history, in most cases an equal or equivalent code can be reconstructed. If C is an unmanipulated code, the result is equal to output of the function `Print` (see 3.6.1).

```
gap> Display( RepetitionCode( 6, GF(3) ) );
a cyclic [6,1,6]4 repetition code over GF(3)
gap> C1 := ExtendedCode( HammingCode(2) );;
gap> C2 := PuncturedCode( ReedMullerCode( 2, 3 ) );;
gap> Display( LengthenedCode( UUVCode( C1, C2 ) ) );
a linear [12,8,2]2..4 code, lengthened with 1 column(s) of
a linear [11,8,1]1..2 U U+V construction code of
U: a linear [4,1,4]2 extended code of
   a linear [3,1,3]1 Hamming (2,2) code over GF(2)
V: a linear [7,7,1]0 punctured code of
   a cyclic [8,7,2]1 Reed-Muller (2,3) code over GF(2)
```

3.7 Generating (Check) Matrices and Polynomials

1 ▶ `GeneratorMat( C )`

`GeneratorMat` returns a generator matrix of C. The code consists of all linear combinations of the rows of this matrix.

If until now no generator matrix of C was determined, it is computed from either the parity check matrix, the generator polynomial, the check polynomial or the elements (if possible), whichever is available.

If C is a non-linear code, the function returns an error.

```
gap> GeneratorMat( HammingCode( 3, GF(2) ) );
[ <an immutable GF2 vector of length 7>, <an immutable GF2 vector of length 7>
    , <an immutable GF2 vector of length 7>,
  <an immutable GF2 vector of length 7> ]
gap> GeneratorMat( RepetitionCode( 5, GF(25) ) );
[ [ Z(5)^0, Z(5)^0, Z(5)^0, Z(5)^0, Z(5)^0 ] ]
gap> GeneratorMat( NullCode( 14, GF(4) ) );
[  ]
```

2 ▶ `CheckMat( C )`

`CheckMat` returns a parity check matrix of C. The code consists of all words orthogonal to each of the rows of this matrix. The transpose of the matrix is a right inverse of the generator matrix. The parity check matrix is computed from either the generator matrix, the generator polynomial, the check polynomial or the elements of C (if possible), whichever is available.

If C is a non-linear code, the function returns an error.

```
gap> CheckMat( HammingCode(3, GF(2) ) );
[ <an immutable GF2 vector of length 7>, <an immutable GF2 vector of length 7>
    , <an immutable GF2 vector of length 7> ]
gap> CheckMat( RepetitionCode( 5, GF(25) ) );
[ [ Z(5)^0, Z(5)^2, 0*Z(5), 0*Z(5), 0*Z(5) ],
  [ 0*Z(5), Z(5)^0, Z(5)^2, 0*Z(5), 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), Z(5)^0, Z(5)^2, 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0, Z(5)^2 ] ]
gap> CheckMat( WholeSpaceCode( 12, GF(4) ) );
[  ]
```

3 ▶ `GeneratorPol( C )`

`GeneratorPol` returns the generator polynomial of C. The code consists of all multiples of the generator polynomial modulo $x^n - 1$ where n is the word length of C. The generator polynomial is determined from either the check polynomial, the generator or check matrix or the elements of C (if possible), whichever is available.

If C is not a cyclic code, the function returns `false`.

```
gap> GeneratorPol(GeneratorMatCode([[1, 1, 0], [0, 1, 1]], GF(2)));
Z(2)^0+x_1
gap> GeneratorPol( WholeSpaceCode( 4, GF(2) ) );
Z(2)^0
gap> GeneratorPol( NullCode( 7, GF(3) ) );
-Z(3)^0+x_1^7
```

4 ▶ `CheckPol( C )`

`CheckPol` returns the check polynomial of C. The code consists of all polynomials f with $f * h = 0 (\bmod\ x^n - 1)$, where h is the check polynomial, and n is the word length of C. The check polynomial is computed from the generator polynomial, the generator or parity check matrix or the elements of C (if possible), whichever is available.

If C if not a cyclic code, the function returns an error.

```
gap> CheckPol(GeneratorMatCode([[1, 1, 0], [0, 1, 1]], GF(2)));
Z(2)^0+x_1+x_1^2
gap> CheckPol(WholeSpaceCode(4, GF(2)));
Z(2)^0+x_1^4
gap> CheckPol(NullCode(7,GF(3)));
Z(3)^0
```

5 ► `RootsOfCode(` C `)`

`RootsOfCode` returns a list of all zeros of the generator polynomial of a cyclic code C. These are finite field elements in the splitting field of the generator polynomial, $GF(q^m)$, m is the multiplicative order of the size of the base field of the code, modulo the word length.

The reverse proces, constructing a code from a set of roots, can be carried out by the function `RootsCode` (see 4.4.3).

```
gap> C1 := ReedSolomonCode( 16, 5 );
a cyclic [16,12,5]3..4 Reed-Solomon code over GF(17)
gap> RootsOfCode( C1 );
[ Z(17), Z(17)^2, Z(17)^3, Z(17)^4 ]
gap> C2 := RootsCode( 16, last );
a cyclic [16,12,5]3..4 code defined by roots over GF(17)
gap> C1 = C2;
true
```

3.8 Parameters of Codes

1 ► `WordLength(` C `)`

`WordLength` returns the parameter n of C, the word length of the elements. Elements of cyclic codes are polynomials of maximum degree $n-1$, as calculations are carried out modulo x^n-1.

```
gap> WordLength( NordstromRobinsonCode() );
16
gap> WordLength( PuncturedCode( WholeSpaceCode(7) ) );
6
gap> WordLength( UUVCode( WholeSpaceCode(7), RepetitionCode(7) ) );
14
```

2 ► `Redundancy(` C `)`

`Redundancy` returns the redundancy r of C, which is equal to the number of check symbols in each element. If C is not a linear code the redundancy is not defined and `Redundancy` returns an error.

If a linear code C has dimension k and word length n, it has redundancy $r=n-k$.

```
gap> C := TernaryGolayCode();
a cyclic [11,6,5]2 ternary Golay code over GF(3)
gap> Redundancy(C);
5
gap> Redundancy( DualCode(C) );
6
```

3 ► `MinimumDistance(` C `)`

`MinimumDistance` returns the minimum distance of C, the largest integer d with the property that every element of C has at least a Hamming distance d (see 2.6.2) to any other element of C. For linear codes, the minimum distance is equal to the minimum weight. This means that d is also the smallest positive value with $w[d+1] \neq 0$, where w is the weight distribution of C (see 3.9.1). For unrestricted codes, d is the smallest positive value with $w[d+1] \neq 0$, where w is the inner distribution of C (see 3.9.2).

For codes with only one element, the minimum distance is defined to be equal to the word length.

```
gap> C := MOLSCode(7);; MinimumDistance(C);
3
gap> WeightDistribution(C);
[ 1, 0, 0, 24, 24 ]
gap> MinimumDistance( WholeSpaceCode( 5, GF(3) ) );
1
gap> MinimumDistance( NullCode( 4, GF(2) ) );
4
gap> C := ConferenceCode(9);; MinimumDistance(C);
4
gap> InnerDistribution(C);
[ 1, 0, 0, 0, 63/5, 9/5, 18/5, 0, 9/10, 1/10 ]
```

4 ► `MinimumDistance(` C, w `)`

In this form, `MinimumDistance` returns the minimum distance of a codeword w to the code C, also called the **distance to** C. This is the smallest value d for which there is an element c of the code C which is at distance d from w. So d is also the minimum value for which $D[d+1] \neq 0$, where D is the distance distribution of w to C (see 3.9.4).

Note that w must be an element of the same vector space as the elements of C. w does not necessarily belong to the code (if it does, the minimum distance is zero).

```
gap> C := MOLSCode(7);; w := CodewordNr( C, 17 );
[ 3 3 6 2 ]
gap> MinimumDistance( C, w );
0
gap> C := RemovedElementsCode( C, w );; MinimumDistance( C, w );
3                           # so w no longer belongs to C
```

3.9 Distributions

1 ► `WeightDistribution(` C `)`

`WeightDistribution` returns the weight distribution of C, as a vector. The i^{th} element of this vector contains the number of elements of C with weight $i-1$. For linear codes, the weight distribution is equal to the inner distribution (see 3.9.2).

Suppose w is the weight distribution of C. If C is linear, it must have the zero codeword, so $w[1] = 1$ (one word of weight 0).

```
gap> WeightDistribution( ConferenceCode(9) );
[ 1, 0, 0, 0, 0, 18, 0, 0, 0, 1 ]
gap> WeightDistribution( RepetitionCode( 7, GF(4) ) );
[ 1, 0, 0, 0, 0, 0, 0, 3 ]
gap> WeightDistribution( WholeSpaceCode( 5, GF(2) ) );
[ 1, 5, 10, 10, 5, 1 ]
```

2► `InnerDistribution(` C `)`

`InnerDistribution` returns the inner distribution of C. The i^{th} element of the vector contains the average number of elements of C at distance $i-1$ to an element of C. For linear codes, the inner distribution is equal to the weight distribution (see 3.9.1).

Suppose w is the inner distribution of C. Then $w[1] = 1$, because each element of C has exactly one element at distance zero (the element itself). The minimum distance of C is the smallest value $d > 0$ with $w[d+1] \neq 0$, because a distance between zero and d never occurs. See 3.8.4.

```
gap> InnerDistribution( ConferenceCode(9) );
[ 1, 0, 0, 0, 63/5, 9/5, 18/5, 0, 9/10, 1/10 ]
gap> InnerDistribution( RepetitionCode( 7, GF(4) ) );
[ 1, 0, 0, 0, 0, 0, 0, 3 ]
```

3► `OuterDistribution(` C `)`

The function `OuterDistribution` returns a list of length q^n, where q is the size of the base field of C and n is the word length. The elements of the list consist of an element of $(GF(q))^n$ (this is a codeword type) and the distribution of distances to the code (a list of integers). This table is **very** large, and for $n > 20$ it will not fit in the memory of most computers. The function `DistancesDistribution` (see 3.9.4) can be used to calculate one entry of the list.

```
gap> C := RepetitionCode( 3, GF(2) );
a cyclic [3,1,3]1 repetition code over GF(2)
gap> OD := OuterDistribution(C);
[ [ [ 0 0 0 ], [ 1, 0, 0, 1 ] ], [ [ 1 1 1 ], [ 1, 0, 0, 1 ] ],
  [ [ 0 0 1 ], [ 0, 1, 1, 0 ] ], [ [ 1 1 0 ], [ 0, 1, 1, 0 ] ],
  [ [ 1 0 0 ], [ 0, 1, 1, 0 ] ], [ [ 0 1 1 ], [ 0, 1, 1, 0 ] ],
  [ [ 0 1 0 ], [ 0, 1, 1, 0 ] ], [ [ 1 0 1 ], [ 0, 1, 1, 0 ] ] ]
gap> WeightDistribution(C) = OD[1][2];
true
gap> DistancesDistribution( C, Codeword("110") ) = OD[4][2];
true
```

4► `DistancesDistribution(` C, w `)`

`DistancesDistribution` returns a distribution of the distances of all elements of C to a codeword w in the same vector space. The i^{th} element of the distance distribution is the number of codewords of C that have distance $i-1$ to w. The smallest value d with $w[d+1] \neq 0$ is defined as the **distance to** C (see 3.8.4).

```
gap> H := HadamardCode(20);
a (20,40,10)6..8 Hadamard code of order 20 over GF(2)
gap> c := Codeword("10110101101010010101", H);
[ 1 0 1 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 0 1 ]
gap> DistancesDistribution(H, c);
[ 0, 0, 0, 0, 0, 1, 0, 7, 0, 12, 0, 12, 0, 7, 0, 1, 0, 0, 0, 0, 0 ]
gap> MinimumDistance(H, c);
5                           # distance to H
```

3.10 Decoding Functions

1 ▶ `Decode(` *C* `,` *c* `)`

`Decode` decodes *c* with respect to code *C*. *c* is a codeword or a list of codewords. First, possible errors in *c* are corrected, then the codeword is decoded to an information codeword *x*. If the code record has a field `specialDecoder`, this special algorithm is used to decode the vector. Hamming codes and BCH codes have such a special algorithm. Otherwise, syndrome decoding is used. Encoding is done by multiplying the information vector with the code (see 3.2).

A special decoder can be created by defining a function

```
C!.SpecialDecoder := function(C, c) ... end;
```

The function uses the arguments *C*, the code record itself, and *c*, a vector of the codeword type, to decode *c* to an information word. A normal decoder would take a codeword *c* of the same word length and field as *C*, and would return a information word of length *k*, the dimension of *C*. The user is not restricted to these normal demands though, and can for instance define a decoder for non-linear codes.

```
gap> C := HammingCode(3);
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> c := "1010"*C;                     # encoding
[ 1 0 1 1 0 1 0 ]
gap> Decode(C, c);                      # decoding
[ 1 0 1 0 ]
gap> Decode(C, Codeword("0010101"));
[ 1 1 0 1 ]                             # one error corrected
gap> C!.SpecialDecoder := function(C, c)
> return NullWord(Dimension(C));
> end;
function ( C, c ) ... end
gap> Decode(C, c);
[ 0 0 0 0 ]           # new decoder always returns null word
```

2 ▶ `Syndrome(` *C* `,` *c* `)`

`Syndrome` returns the syndrome of word *c* with respect to a code *C*. *c* is a word of the vector space of *C*. If *c* is an element of *C*, the syndrome is a zero vector. The syndrome can be used for looking up an error vector in the syndrome table (see 3.10.3) that is needed to correct an error in *c*.

A syndrome is not defined for non-linear codes. `Syndrome` then returns an error.

```
gap> C := HammingCode(4);
a linear [15,11,3]1 Hamming (4,2) code over GF(2)
gap> v := CodewordNr( C, 7 );
[ 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 ]
gap> Syndrome( C, v );
[ 0 0 0 0 ]
gap> Syndrome( C, Codeword( "000000001100111" ) );
[ 1 1 1 1 ]
gap> Syndrome( C, Codeword( "000000000000001" ) );
[ 1 1 1 1 ]    # the same syndrome: both codewords are in the same
               # coset of C
```

3 ▶ `SyndromeTable(` *C* `)`

`SyndromeTable` returns a **syndrome table** of a linear code *C*, consisting of two columns. The first column consists of the error vectors that correspond to the syndrome vectors in the second column. These vectors both are of the codeword type. After calculating the syndrome of a word c with `Syndrome` (see 3.10.2), the error vector needed to correct c can be found in the syndrome table. Subtracting this vector from c yields an element of C. To make the search for the syndrome as fast as possible, the syndrome table is sorted according to the syndrome vectors.

```
gap> H := HammingCode(2);
a linear [3,1,3]1 Hamming (2,2) code over GF(2)
gap> SyndromeTable(H);
[ [ [ 0 0 0 ], [ 0 0 ] ], [ [ 1 0 0 ], [ 0 1 ] ],
  [ [ 0 1 0 ], [ 1 0 ] ], [ [ 0 0 1 ], [ 1 1 ] ] ]
gap> c := Codeword("101");
[ 1 0 1 ]
gap> c in H;
false          # c is not an element of H
gap> Syndrome(H,c);
[ 1 0 ]        # according to the syndrome table,
               # the error vector [ 0 1 0 ] belongs to this syndrome
gap> c - Codeword("010") in H;
true           # so the corrected codeword is
               # [ 1 0 1 ] - [ 0 1 0 ] = [ 1 1 1 ],
               # this is an element of H
```

4 ► `StandardArray(` C `)`

`StandardArray` returns the standard array of a code C. This is a matrix with elements of the codeword type. It has q^r rows and q^k columns, where q is the size of the base field of C, r is the redundancy of C, and k is the dimension of C. The first row contains all the elements of C. Each other row contains words that do not belong to the code, with in the first column their syndrome vector (see 3.10.2).

A non-linear code does not have a standard array. `StandardArray` then returns an error.

Note that calculating a standard array can be very time- and memory- consuming.

```
gap> StandardArray(RepetitionCode(3));
[ [ [ 0 0 0 ], [ 1 1 1 ] ], [ [ 0 0 1 ], [ 1 1 0 ] ],
  [ [ 0 1 0 ], [ 1 0 1 ] ], [ [ 1 0 0 ], [ 0 1 1 ] ] ]
```

4 Generating Codes

In this chapter we describe functions for generating codes.

Section 4.1 describes function for generating unrestricted codes.

Section 4.2 describes function for generating linear codes.

Finally, Section 4.4 describes functions for generating cyclic codes.

4.1 Generating Unrestricted Codes

In this section we start with functions that creating code from user defined matrices or special matrices (see 4.1.1, 4.1.3, 4.1.4 and 4.1.6). These codes are unrestricted codes; they may later be discovered to be linear or cyclic.

The next functions generate random codes (see 4.1.7) and the Nordstrom-Robinson code (see 4.1.8), respectively.

Finally, we describe two functions for generating Greedy codes. These are codes that contructed by gathering codewords from a space (see 4.1.9 and 4.1.10).

1 ▶ `ElementsCode(` *L* `[,` *Name* `],` *F* `)`

`ElementsCode` creates an unrestricted code of the list of elements L, in the field F. L must be a list of vectors, strings, polynomials or codewords. *Name* can contain a short description of the code.

If L contains a codeword more than once, it is removed from the list and a GAP set is returned.

```
gap> M := Z(3)^0 * [ [1, 0, 1, 1], [2, 2, 0, 0], [0, 1, 2, 2] ];;
gap> C := ElementsCode( M, "example code", GF(3) );
a (4,3,1..4)2 example code over GF(3)
gap> MinimumDistance( C );
4
gap> AsSSortedList( C );
[ [ 0 1 2 2 ], [ 1 0 1 1 ], [ 2 2 0 0 ] ]
```

2 ▶ `HadamardCode(` *H* `,` *t* `)`
▶ `HadamardCode(` *H* `)`

In the first form `HadamardCode` returns a Hadamard code from the Hadamard matrix H, of the t^{th} kind. In the second form, $t = 3$ is used.

A Hadamard matrix is a square matrix H with $H * H^T = -n * I_n$, where n is the size of H. The entries of H are either 1 or -1.

The matrix H is first transformed into a binary matrix A_n (by replacing the 1s by 0s and the -1s by 1s).

The first kind ($t = 1$) is created by using the rows of A_n as elements, after deleting the first column. This is a $(n-1, n, n/2)$ code. We use this code for creating the Hadamard code of the second kind ($t = 2$), by adding all the complements of the already existing codewords. This results in a $(n-1, 2n, n/2-1)$ code. The third code ($t = 3$) is created by using the rows of A_n (without cutting a column) and their complements as elements. This way, we have an $(n, 2n, n/2)$ code. The returned code is generally an unrestricted code, but for $n = 2^r$, the code is linear.

```
gap> H4 := [[1,1,1,1],[1,-1,1,-1],[1,1,-1,-1],[1,-1,-1,1]];;
gap> HadamardCode( H4, 1 );
a (3,4,2)1 Hadamard code of order 4 over GF(2)
gap> HadamardCode( H4, 2 );
a (3,8,1)0 Hadamard code of order 4 over GF(2)
gap> HadamardCode( H4 );
a (4,8,2)1 Hadamard code of order 4 over GF(2)
```

3 ▶ `HadamardCode(` n, t `)`
▶ `HadamardCode(` n `)`

In the first form `HadamardCode` returns a Hadamard code with parameter n of the t^{th} kind. In the second form, $t = 3$ is used.

When called in these forms, `HadamardCode` first creates a Hadamard matrix (see 6.2.4), of size n and then follows the same procedure as described above. Therefore the same restrictions with respect to n as for Hadamard matrices hold.

```
gap> C1 := HadamardCode( 4 );
a (4,8,2)1 Hadamard code of order 4 over GF(2)
gap> C1 = HadamardCode( H4 );
true
```

4 ▶ `ConferenceCode(` H `)`

`ConferenceCode` returns a code of length $n-1$ constructed from a symmetric **conference matrix** H. H must be a symmetric matrix of order n, which satisfies $H * H^T = ((n-1) * I$. $n = 2(\bmod 4)$. The rows of $1/2(H+I+J), 1/2(-H+I+J)$, plus the zero and all-ones vectors form the elements of a binary non-linear $(n-1, 2*n, 1/2*(n-2))$ code.

```
gap> H6 := [[0,1,1,1,1,1],[1,0,1,-1,-1,1],[1,1,0,1,-1,-1],
> [1,-1,1,0,1,-1],[1,-1,-1,1,0,1],[1,1,-1,-1,1,0]];;
gap> C1 := ConferenceCode( H6 );
a (5,12,2)1..4 conference code over GF(2)
gap> IsLinearCode( C1 );
false
```

5 ▶ `ConferenceCode(` n `)`

GUAVA constructs a symmetric conference matrix of order $n+1$ ($n = 1(\bmod 4)$) and uses the rows of that matrix, plus the zero and all-ones vectors, to construct a binary non-linear $(n, 2*(n+1), 1/2*(n-1))$ code.

```
gap> C2 := ConferenceCode( 5 );
a (5,12,2)1..4 conference code over GF(2)
gap> AsSSortedList( C2 );
[ [ 0 0 0 0 0 ], [ 0 0 1 1 1 ], [ 0 1 0 1 1 ], [ 0 1 1 0 1 ], [ 0 1 1 1 0 ],
  [ 1 0 0 1 1 ], [ 1 0 1 0 1 ], [ 1 0 1 1 0 ], [ 1 1 0 0 1 ], [ 1 1 0 1 0 ],
  [ 1 1 1 0 0 ], [ 1 1 1 1 1 ] ]
```

6 ▶ `MOLSCode(` n, q `)`
▶ `MOLSCode(` q `)`

`MOLSCode` returns an $(n, q^2, n-1)$ code over `GF(`q`)`. The code is created from $n-2$ **Mutually Orthogonal Latin Squares** (MOLS) of size $q*q$. The default for n is 4. GUAVA can construct a MOLS code for $n-2 \leq q$; q must be a prime power, $q > 2$. If there are no $n-2$ MOLS, an error is signalled.

Since each of the $n-2$ MOLS is a $q*q$ matrix, we can create a code of size q^2 by listing in each code element the entries that are in the same position in each of the MOLS. We precede each of these lists with the two coordinates that specify this position, making the word length become n.

The MOLS codes are MDS codes (see 3.3.5).

```
gap> C1 := MOLSCode( 6, 5 );
a (6,25,5)3..4 code generated by 4 MOLS of order 5 over GF(5)
gap> mols := List( [1 .. WordLength(C1) - 2 ], function( nr )
>       local ls, el;
>       ls := NullMat( Size(LeftActingDomain(C1)), Size(LeftActingDomain(C1)) );
>       for el in VectorCodeword( AsSSortedList( C1 ) ) do
>          ls[IntFFE(el[1])+1][IntFFE(el[2])+1] := el[nr + 2];
>       od;
>       return ls;
>    end );;
gap> AreMOLS( mols );
true
gap> C2 := MOLSCode( 11 );
a (4,121,3)2 code generated by 2 MOLS of order 11 over GF(11)
```

7 ► `RandomCode( n, M, F )`

`RandomCode` returns a random unrestricted code of size M with word length n over F. M must be less than or equal to the number of elements in the space $GF(q)^n$.

The function `RandomLinearCode` returns a random linear code (see 4.2.11).

```
gap> C1 := RandomCode( 6, 10, GF(8) );
a (6,10,1..6)4..6 random unrestricted code over GF(8)
gap> MinimumDistance(C1);
3
gap> C2 := RandomCode( 6, 10, GF(8) );
a (6,10,1..6)4..6 random unrestricted code over GF(8)
gap> C1 = C2;
false
```

8 ► `NordstromRobinsonCode()`

`NordstromRobinsonCode` returns a Nordstrom-Robinson code, the best code with word length $n = 16$ and minimum distance $d = 6$ over `GF(2)`. This is a non-linear $(16, 256, 6)$ code.

```
gap> C := NordstromRobinsonCode();
a (16,256,6)4 Nordstrom-Robinson code over GF(2)
gap> OptimalityCode( C );
0
```

9 ► `GreedyCode( L, d, F )`

`GreedyCode` returns a Greedy code with design distance d over F. The code is constructed using the Greedy algorithm on the list of vectors L. This algorithm checks each vector in L and adds it to the code if its distance to the current code is greater than or equal to d. It is obvious that the resulting code has a minimum distance of at least d.

Note that Greedy codes are often linear codes.

The function `LexiCode` creates a Greedy code from a basis instead of an enumerated list (see 4.1.10).

```
gap> C1 := GreedyCode( Tuples( AsSSortedList( GF(2) ), 5 ), 3, GF(2) );
a (5,4,3..5)2 Greedy code, user defined basis over GF(2)
gap> C2 := GreedyCode( Permuted( Tuples( AsSSortedList( GF(2) ), 5 ),
>                         (1,4) ), 3, GF(2) );
a (5,4,3..5)2 Greedy code, user defined basis over GF(2)
gap> C1 = C2;
false
```

10 ▶ `LexiCode(` *n*, *d*, *F* `)`

In this format, `Lexicode` returns a Lexicode with word length n, design distance d over F. The code is constructed using the Greedy algorithm on the lexicographically ordered list of all vectors of length n over F. Every time a vector is found that has a distance to the current code of at least d, it is added to the code. This results, obviously, in a code with minimum distance greater than or equal to d.

```
gap> C := LexiCode( 4, 3, GF(5) );
a (4,17,3..4)2..4 lexicode over GF(5)
```

11 ▶ `LexiCode(` *B*, *d*, *F* `)`

When called in this format, `LexiCode` uses the basis B instead of the standard basis. B is a matrix of vectors over F. The code is constructed using the Greedy algorithm on the list of vectors spanned by B, ordered lexicographically with respect to B.

```
gap> B := [ [Z(2)^0, 0*Z(2), 0*Z(2)], [Z(2)^0, Z(2)^0, 0*Z(2)] ];;
gap> C := LexiCode( B, 2, GF(2) );
a linear [3,1,2]1..2 lexicode over GF(2)
```

Note that binary Lexicodes are always linear.

The function `GreedyCode` creates a Greedy code that is not restricted to a lexicographical order (see 4.1.9).

4.2 Generating Linear Codes

In this section we describe functions for constructing linear codes. A linear code always has a generator or check matrix.

The first two functions generate linear codes from the generator matrix (4.2.1) or check matrix (4.2.2). All linear codes can be constructed with these functions.

The next functions we describe generate some well known codes, like Hamming codes (4.2.3), Reed-Muller codes (4.2.4) and the extended Golay codes (4.3.2 and 4.3.4).

A large and powerful family of codes are alternant codes. They are obtained by a small modification of the parity check matrix of a BCH code (see 4.2.5, 4.2.6, 4.2.8 and 4.2.9).

Finally, we describe a function for generating random linear codes (see 4.2.11).

1 ▶ `GeneratorMatCode(` *G* `[,` *Name* `],` *F* `)`

`GeneratorMatCode` returns a linear code with generator matrix G. G must be a matrix over Galois field F. *Name* can contain a short description of the code. The generator matrix is the basis of the elements of the code. The resulting code has word length n, dimension k if G is a $k * n$-matrix. If $GF(q)$ is the field of the code, the size of the code will be q^k.

If the generator matrix does not have full row rank, the linearly dependent rows are removed. This is done by the function `BaseMat` (see 24.10.1) and results in an equal code. The generator matrix can be retrieved with the function `GeneratorMat` (see 3.7.1).

```
gap> G := Z(3)^0 * [[1,0,1,2,0],[0,1,2,1,1],[0,0,1,2,1]];;
gap> C1 := GeneratorMatCode( G, GF(3) );
a linear [5,3,1..2]1..2 code defined by generator matrix over GF(3)
gap> C2 := GeneratorMatCode( IdentityMat( 5, GF(2) ), GF(2) );
a linear [5,5,1]0 code defined by generator matrix over GF(2)
gap> GeneratorMatCode( List( AsSSortedList( NordstromRobinsonCode() ),
> x -> VectorCodeword( x ) ), GF( 2 ) );
a linear [16,11,1..4]2 code defined by generator matrix over GF(2)
# This is the smallest linear code that contains the N-R code
```

2 ▶ `CheckMatCode(` *H* `[,` *Name* `],` *F* `)`

`CheckMatCode` returns a linear code with check matrix H. H must be a matrix over Galois field F. *Name* can contain a short description of the code. The parity check matrix is the transposed of the nullmatrix of the generator matrix of the code. Therefore, $c * H^T = 0$ where c is an element of the code. If H is a $r * n$-matrix, the code has word length n, redundancy r and dimension $n - r$.

If the check matrix does not have full row rank, the linearly dependent rows are removed. This is done by the function `BaseMat` (see 24.10.1) and results in an equal code. The check matrix can be retrieved with the function `CheckMat` (see 3.7.2).

```
gap> G := Z(3)^0 * [[1,0,1,2,0],[0,1,2,1,1],[0,0,1,2,1]];;
gap> C1 := CheckMatCode( G, GF(3) );
a linear [5,2,1..2]2..3 code defined by check matrix over GF(3)
gap> CheckMat(C1);
[ [ Z(3)^0, 0*Z(3), Z(3)^0, Z(3), 0*Z(3) ],
  [ 0*Z(3), Z(3)^0, Z(3), Z(3)^0, Z(3)^0 ],
  [ 0*Z(3), 0*Z(3), Z(3)^0, Z(3), Z(3)^0 ] ]
gap> C2 := CheckMatCode( IdentityMat( 5, GF(2) ), GF(2) );
a cyclic [5,0,5]5 code defined by check matrix over GF(2)
```

3 ▶ `HammingCode(` *r*`,` *F* `)`

`HammingCode` returns a Hamming code with redundancy r over F. A Hamming code is a single-error-correcting code. The parity check matrix of a Hamming code has all nonzero vectors of length r in its columns, except for a multiplication factor. The decoding algorithm of the Hamming code (see 3.10.1) makes use of this property.

If q is the size of its field F, the returned Hamming code is a linear $[(q^r - 1)/(q - 1), (q^r - 1)/(q - 1) - r, 3]$ code.

```
gap> C1 := HammingCode( 4, GF(2) );
a linear [15,11,3]1 Hamming (4,2) code over GF(2)
gap> C2 := HammingCode( 3, GF(9) );
a linear [91,88,3]1 Hamming (3,9) code over GF(9)
```

4 ▶ `ReedMullerCode(` *r*`,` *k* `)`

`ReedMullerCode` returns a binary **Reed-Muller code** $R(r, k)$ with dimension k and order r. This is a code with length 2^k and minimum distance 2^{k-r}. By definition, the r^{th} order binary Reed-Muller code of length $n = 2^m$, for $0 \le r \le m$, is the set of all vectors f, where f is a Boolean function which is a polynomial of degree at most r.

```
gap> ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
```

5 ► `AlternantCode(` r, Y, F `)`
► `AlternantCode(` r, Y, *alpha*, F `)`

`AlternantCode` returns an **alternant code**, with parameters r, Y and *alpha* (optional). r is the design redundancy of the code. Y and *alpha* are both vectors of length n from which the parity check matrix is constructed. The check matrix has entries of the form $a_i^j y_i$. If no *alpha* is specified, the vector $[1, a, a^2, \cdots, a^{n-1}]$ is used, where a is a primitive element of a Galois field F.

```
gap> Y := [ 1, 1, 1, 1, 1, 1, 1];; a := PrimitiveUnityRoot( 2, 7 );;
gap> alpha := List( [0..6], i -> a^i );;
gap> C := AlternantCode( 2, Y, alpha, GF(8) );
a linear [7,3,3..4]3..4 alternant code over GF(8)
```

6 ► `GoppaCode(` G, L `)`

`GoppaCode` returns a Goppa code from Goppa polynomial G, having coefficients in a Galois Field $GF(q^m)$. L must be a list of elements in $GF(q^m)$, that are not roots of G. The word length of the code is equal to the length of L. The parity check matrix contains entries of the form $a_i^j G(a_i)$, a_i in L. The function `VerticalConversionFieldMat` converts this matrix to a matrix with entries in $GF(q)$ (see 6.2.9).

```
gap> x := Indeterminate( GF(2), "x" );;
gap> G := x^2 + x + 1;; L := AsSSortedList( GF(8) );;
gap> C := GoppaCode( G, L );
a linear [8,2,5]3 Goppa code over GF(2)
```

7 ► `GoppaCode(` G, n `)`

When called with parameter n, GUAVA constructs a list L of length n, such that no element of L is a root of G.

```
gap> x := Indeterminate( GF(2), "x" );;
gap> G := x^2 + x + 1;;
gap> C := GoppaCode( G, 8 );
a linear [8,2,5]3 Goppa code over GF(2)
```

8 ► `GeneralizedSrivastavaCode(` a, w, z, F `)`
► `GeneralizedSrivastavaCode(` a, w, z, t, F `)`

`GeneralizedSrivastavaCode` returns a generalized Srivastava code with parameters a, w, z, t. $a = a_1, \cdots, a_n$ and $w = w_1, \cdots, w_s$ are lists of $n + s$ distinct elements of $F = GF(q^m)$, z is a list of length n of nonzero elements of $GF(q^m)$. The parameter t determines the designed distance: $d \geq st+1$. The parity check matrix of this code has entries of the form

$$\frac{z_i}{(a_i - w_l)^k}$$

`VerticalConversionFieldMat` converts this matrix to a matrix with entries in $GF(q)$ (see 6.2.9). The default for t is 1. The original Srivastava codes (see 4.2.9) are a special case $t = 1$, $z_i = a_i^\mu$ for some μ.

```
gap> a := Filtered( AsSSortedList( GF(2^6) ), e -> e in GF(2^3) );;
gap> w := [ Z(2^6) ];; z := List( [1..8], e -> 1 );;
gap> C := GeneralizedSrivastavaCode( a, w, z, 1, GF(64) );
a linear [8,2,2..5]3..4 generalized Srivastava code over GF(2)
```

9 ▶ `SrivastavaCode(` a, w, F `)`
▶ `SrivastavaCode(` a, w, mu, F `)`

`SrivastavaCode` returns a Srivastava code with parameters a, w, mu. $a = a_1, \cdots, a_n$ and $w = w_1, \cdots, w_s$ are lists of $n+s$ distinct elements of $F = GF(q^m)$. The default for mu is 1. The Srivastava code is a generalized Srivastava code (see 4.2.8), in which $z\_i = a_i^{mu}$ for some mu and $t = 1$.

```
gap> a := AsSSortedList( GF(11) ){[2..8]};;
gap> w := AsSSortedList( GF(11) ){[9..10]};;
gap> C := SrivastavaCode( a, w, 2, GF(11) );
a linear [7,5,3]2 Srivastava code over GF(11)
gap> IsMDSCode( C );
true    # Always true if F is a prime field
```

10 ▶ `CordaroWagnerCode(` n `)`

`CordaroWagnerCode` returns a binary Cordaro-Wagner code. This is a code of length n and dimension 2 having the best possible minimum distance d. This code is just a little bit less trivial than `RepetitionCode` (see 4.4.11).

```
gap> C := CordaroWagnerCode( 11 );
a linear [11,2,7]5 Cordaro-Wagner code over GF(2)
gap> AsSSortedList(C);
[ [ 0 0 0 0 0 0 0 0 0 0 0 ], [ 0 0 0 0 1 1 1 1 1 1 1 ],
  [ 1 1 1 1 0 0 0 1 1 1 1 ], [ 1 1 1 1 1 1 1 0 0 0 0 ] ]
```

11 ▶ `RandomLinearCode(` n, k , F `)`

`RandomLinearCode` returns a random linear code with word length n, dimension k over field F.

To create a random unrestricted code, use `RandomCode` (see 4.1.7).

```
gap> C := RandomLinearCode( 15, 4, GF(3) );
a linear [15,4,1..6]6..10 random linear code over GF(3)
```

12 ▶ `BestKnownLinearCode(` n, k , F `)`

`BestKnownLinearCode` returns the best known linear code of length n, dimension k over field F. The function uses the tables described in section 6.1.12 to construct this code.

```
gap> C1 := BestKnownLinearCode( 23, 12, GF(2) );
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> C1 = BinaryGolayCode();
true
gap> Display( BestKnownLinearCode( 8, 4, GF(4) ) );
a linear [8,4,4]2..3 U U+V construction code of
U: a cyclic [4,3,2]1 dual code of
   a cyclic [4,1,4]3 repetition code over GF(4)
V: a cyclic [4,1,4]3 repetition code over GF(4)
gap> C := BestKnownLinearCode(131,47);
a linear [131,47,28..32]23..68 shortened code
```

13 ► `BestKnownLinearCode( rec )`

In this form, *rec* must be a record containing the fields `lowerBound`, `upperBound` and `construction`. It uses the information in this field to construct a code. This form is meant to be used together with the function `BoundsMinimumDistance` (see 6.1.12), if the bounds are already calculated.

```
gap> bounds := BoundsMinimumDistance( 20, 17, GF(4) );
rec( n := 20, k := 17, q := 4,
  references := rec( HM := [ "%T this reference is unknown, for more info",
          "%T contact A.E. Brouwer (aeb@cwi.nl)" ] ),
  construction := [ <Operation "ShortenedCode">,
      [ [ <Operation "HammingCode">, [ 3, 4 ] ], [ 1 ] ] ], lowerBound := 3,
  lowerBoundExplanation := [ "Lb(20,17)=3, by shortening of:",
      "Lb(21,18)=3, reference: HM" ], upperBound := 3,
  upperBoundExplanation :=
    [ "Ub(20,17)=3, otherwise construction B would contradict:",
      "Ub(3,1)=3, repetition code" ] )
gap> C := BestKnownLinearCode( bounds );
a linear [20,17,3]2 shortened code
gap> C = BestKnownLinearCode( 20, 17, GF(4) );
true
```

4.3 Golay Codes

1 ► `BinaryGolayCode()`

`BinaryGolayCode` returns a binary Golay code. This is a perfect [23,12,7] code. It is also cyclic, and has generator polynomial $g(x) = 1 + x^2 + x^4 + x^5 + x^6 + x^{10} + x^{11}$. Extending it results in an extended Golay code (see 4.3.2). There's also the ternary Golay code (see 4.3.3).

```
gap> BinaryGolayCode();
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> ExtendedBinaryGolayCode() = ExtendedCode(BinaryGolayCode());
true
gap> IsPerfectCode(BinaryGolayCode());
true
```

2 ► `ExtendedBinaryGolayCode( )`

`ExtendedBinaryGolayCode` returns an extended binary Golay code. This is a $[24, 12, 8]$ code. Puncturing in the last position results in a perfect binary Golay code (see 4.3.1). The code is self-dual (see 3.3.6).

```
gap> C := ExtendedBinaryGolayCode();
a linear [24,12,8]4 extended binary Golay code over GF(2)
gap> P := PuncturedCode(C);
a linear [23,12,7]3 punctured code
gap> P = BinaryGolayCode();
true
```

3 ► `TernaryGolayCode()`

`TernaryGolayCode` returns a ternary Golay code. This is a perfect [11,6,5] code. It is also cyclic, and has generator polynomial $g(x) = 2 + x^2 + 2x^3 + x^4 + x^5$. Extending it results in an extended Golay code (see 4.3.4). There's also the binary Golay code (see 4.3.1).

```
gap> TernaryGolayCode();
a cyclic [11,6,5]2 ternary Golay code over GF(3)
gap> ExtendedTernaryGolayCode() = ExtendedCode(TernaryGolayCode());
true
```

4 ► `ExtendedTernaryGolayCode( )`

`ExtendedTernaryGolayCode` returns an extended ternary Golay code. This is a $[12, 6, 6]$ code. Puncturing this code results in a perfect ternary Golay code (see 4.3.3). The code is self-dual (see 3.3.6).

```
gap> C := ExtendedTernaryGolayCode();
a linear [12,6,6]3 extended ternary Golay code over GF(3)
gap> P := PuncturedCode(C);
a linear [11,6,5]2 punctured code
gap> P = TernaryGolayCode();
true
```

4.4 Generating Cyclic Codes

The elements of a cyclic code C are all multiples of a polynomial $g(x)$, where calculations are carried out modulo $x^n - 1$. Therefore, the elements always have a degree less than n. A cyclic code is an ideal in the ring of polynomials modulo $x^n - 1$. The polynomial $g(x)$ is called the **generator polynomial** of C. This is the unique monic polynomial of least degree that generates C. It is a divisor of the polynomial $x^n - 1$.

The **check polynomial** is the polynomial $h(x)$ with $g(x) * h(x) = x^n - 1$. Therefore it is also a divisor of $x^n - 1$. The check polynomial has the property that $c(x) * h(x) = 0(\mathrm{mod}\ (x^n - 1))$ for every codeword $c(x)$.

The first two functions described generate cyclic codes from a given generator or check polynomial. All cyclic codes can be constructed using these functions.

Next are described the two cyclic Golay codes (see 4.3.1 and 4.3.3).

Then functions that generate cyclic codes from a prescribed set of roots of the generator polynomial are described, including the BCH codes (see 4.4.3, 4.4.5, 4.4.6 and 4.4.7).

Finally we describe the trivial codes (see 4.4.9, 4.4.10, 4.4.11), and `CyclicCodes` (4.4.12).

1 ► `GeneratorPolCode(` *g*`,` *n* `[,` *Name* `],` *F* `)`

`GeneratorPolCode` creates a cyclic code with a generator polynomial g, word length n, over F. *Name* can contain a short description of the code.

If g is not a divisor of $x^n - 1$, it cannot be a generator polynomial. In that case, a code is created with generator polynomial $gcd(g, x^n - 1)$, i.e. the greatest common divisor of g and $x^n - 1$. This is a valid generator polynomial that generates the ideal (g). See 4.4.

```
gap> x:= Indeterminate( GF(2) );; P:= x^2+1;
Z(2)^0+x^2
gap> G := GeneratorPolCode(P, 7, GF(2));
a cyclic [7,6,1..2]1 code defined by generator polynomial over GF(2)
gap> GeneratorPol( G );
Z(2)^0+x
gap> G2 := GeneratorPolCode( x+1, 7, GF(2));
a cyclic [7,6,1..2]1 code defined by generator polynomial over GF(2)
gap> GeneratorPol( G2 );
Z(2)^0+x
```

2 ► `CheckPolCode(` *h*`,` *n* `[,` *Name* `],` *F* `)`

`CheckPolCode` creates a cyclic code with a check polynomial h, word length n, over F. *Name* can contain a short description of the code.

If h is not a divisor of $x^n - 1$, it cannot be a check polynomial. In that case, a code is created with check polynomial $gcd(h, x^n - 1)$, i.e. the greatest common divisor of h and $x^n - 1$. This is a valid check polynomial that yields the same elements as the ideal (h). See 4.4.

```
gap>  x:= Indeterminate( GF(3) );; P:= x^2+2;
-Z(3)^0+x_1^2
gap> H := CheckPolCode(P, 7, GF(3));
a cyclic [7,1,7]4 code defined by check polynomial over GF(3)
gap> CheckPol(H);
-Z(3)^0+x_1
gap> Gcd(P, X(GF(3))^7-1);
-Z(3)^0+x_1
```

3 ► `RootsCode(` *n*`,` *list* `)`

This is the generalization of the BCH, Reed-Solomon and quadratic residue codes (see 4.4.5, 4.4.6 and 4.4.7). The user can give a length of the code n and a prescribed set of zeros. The argument *list* must be a valid list of primitive n^{th} roots of unity in a splitting field $GF(q^m)$. The resulting code will be over the field $GF(q)$. The function will return the largest possible cyclic code for which the list *list* is a subset of the roots of the code. From this list, GUAVA calculates the entire set of roots.

```
gap> a := PrimitiveUnityRoot( 3, 14 );
Z(3^6)^52
gap> C1 := RootsCode( 14, [ a^0, a, a^3 ] );
a cyclic [14,7,3..6]3..7 code defined by roots over GF(3)
gap> MinimumDistance( C1 );
4
gap> b := PrimitiveUnityRoot( 2, 15 );
Z(2^4)
gap> C2 := RootsCode( 15, [ b, b^2, b^3, b^4 ] );
a cyclic [15,7,5]3..5 code defined by roots over GF(2)
gap> C2 = BCHCode( 15, 5, GF(2) );
true
```

4 ► `RootsCode(` *n*`,` *list*`,` *F* `)`

In this second form, the second argument is a list of integers, ranging from 0 to n-1. The resulting code will be over a field F. GUAVA calculates a primitive n^{th} root of unity, α, in the extension field of F. It uses the set of the powers of α in the list as a prescribed set of zeros.

```
gap> C := RootsCode( 4, [ 1, 2 ], GF(5) );
a cyclic [4,2,3]2 code defined by roots over GF(5)
gap> RootsOfCode( C );
[ Z(5), Z(5)^2 ]
gap> C = ReedSolomonCode( 4, 3 );
true
```

5 ► `BCHCode(` *n*`,` *d* `,` *F* `)`
► `BCHCode(` *n*`,` *b*`,` *d*`,` *F* `)`

The function `BCHCode` returns a **Bose-Chaudhuri-Hockenghem code** (or BCH code for short). This is the largest possible cyclic code of length n over field F, whose generator polynomial has zeros

$$a^b, a^{b+1}, \cdots, a^{b+d-2},$$

where a is a primitive n^{th} root of unity in the splitting field $GF(q^m)$, b is an integer > 1 and m is the multiplicative order of q modulo n. Default value for b is 1. The length n of the code and the size q of the field must be relatively prime. The generator polynomial is equal to the product of the minimal polynomials of $X^b, X^{b+1}, \cdots, X^{b+d-2}$.

Special cases are $b = 1$ (resulting codes are called **narrow-sense** BCH codes), and $n = q^m - 1$ (known as **primitive** BCH codes). GUAVA calculates the largest value of $\hat{d}$for which the BCH code with designed distance $\acute{d}$coincides with the BCH code with designed distance d. This distance is called the **Bose distance** of the code. The true minimum distance of the code is greater than or equal to the Bose distance.

Printed are the designed distance (to be precise, the Bose distance) `delta`, and the starting power `b`.

```
gap> C1 := BCHCode( 15, 3, 5, GF(2) );
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> DesignedDistance( C1 );
7
gap> C2 := BCHCode( 23, 2, GF(2) );
a cyclic [23,12,5..7]3 BCH code, delta=5, b=1 over GF(2)
gap> DesignedDistance( C2 );
5
gap> MinimumDistance(C2);
7
```

6 ▶ `ReedSolomonCode(` *n*`,` *d* `)`

`ReedSolomonCode` returns a **Reed-Solomon code** of length n, designed distance d. This code is a primitive narrow-sense BCH code over the field $GF(q)$, where $q = n + 1$. The dimension of an RS code is $n - d + 1$. According to the Singleton bound (see 6.1.1) the dimension cannot be greater than this, so the true minimum distance of an RS code is equal to d and the code is maximum distance separable (see 3.3.5).

```
gap> C1 := ReedSolomonCode( 3, 2 );
a cyclic [3,2,2]1 Reed-Solomon code over GF(4)
gap> C2 := ReedSolomonCode( 4, 3 );
a cyclic [4,2,3]2 Reed-Solomon code over GF(5)
gap> RootsOfCode( C2 );
[ Z(5), Z(5)^2 ]
gap> IsMDSCode(C2);
true
```

7 ▶ `QRCode(` *n*`,` *F* `)`

`QRCode` returns a quadratic residue code. If F is a field $GF(q)$, then q must be a quadratic residue modulo n. That is, an x exists with $x^2 = q (\mathrm{mod}\ n)$. Both n and q must be primes. Its generator polynomial is the product of the polynomials $x - a^i$. a is a primitive n^{th} root of unity, and i is an integer in the set of quadratic residues modulo n.

```
gap> C1 := QRCode( 7, GF(2) );
a cyclic [7,4,3]1 quadratic residue code over GF(2)
gap> IsEquivalent( C1, HammingCode( 3, GF(2) ) );
true
gap> C2 := QRCode( 11, GF(3) );
a cyclic [11,6,4..5]2 quadratic residue code over GF(3)
gap> C2 = TernaryGolayCode();
true
```

8 ▶ `FireCode(` *G*`,` *b* `)`

`FireCode` constructs a (binary) Fire code. G is a primitive polynomial of degree m, factor of $x^r - 1$. b an integer $0 \leq b \leq m$ not divisible by r, that determines the burst length of a single error burst that can be corrected. The argument G can be a polynomial with base ring $GF(2)$, or a list of coefficients in $GF(2)$. The generator polynomial is defined as the product of G and $x^{2b-1}+1$.

```
gap> x:= Indeterminate( GF(2) );; G:= x^3+x^2+1;
Z(2)^0+x^2+x^3
gap> Factors( G );
[ Z(2)^0+x^2+x^3 ]
gap> C := FireCode( G, 3 );
a cyclic [35,27,1..4]2..6 3 burst error correcting fire code over GF(2)
gap> MinimumDistance( C );
4     # Still it can correct bursts of length 3
```

9 ► `WholeSpaceCode(` n, F `)`

`WholeSpaceCode` returns the cyclic whole space code of length n over F. This code consists of all polynomials of degree less than n and coefficients in F.

```
gap> C := WholeSpaceCode( 5, GF(3) );
a cyclic [5,5,1]0 whole space code over GF(3)
```

10 ► `NullCode(` n, F `)`

`NullCode` returns the zero-dimensional nullcode with length n over F. This code has only one word: the all zero word. It is cyclic though!

```
gap> C := NullCode( 5, GF(3) );
a cyclic [5,0,5]5 nullcode over GF(3)
gap> AsSSortedList( C );
[ [ 0 0 0 0 0 ] ]
```

11 ► `RepetitionCode(` n, F `)`

`RepetitionCode` returns the cyclic repetition code of length n over F. The code has as many elements as F, because each codeword consists of a repetition of one of these elements.

```
gap> C := RepetitionCode( 3, GF(5) );
a cyclic [3,1,3]2 repetition code over GF(5)
gap> AsSSortedList( C );
[ [ 0 0 0 ], [ 1 1 1 ], [ 2 2 2 ], [ 4 4 4 ], [ 3 3 3 ] ]
gap> IsPerfectCode( C );
false
gap> IsMDSCode( C );
true
```

12 ► `CyclicCodes(` n, F `)`

`CyclicCodes` returns a list of all cyclic codes of length n over F. It constructs all possible generator polynomials from the factors of $x^n - 1$. Each combination of these factors yields a generator polynomial after multiplication.

13 ► `NrCyclicCodes(` n, F `)`

The function `NrCyclicCodes` calculates the number of cyclic codes of length n over field F.

```
gap> NrCyclicCodes( 23, GF(2) );
8
gap> codelist := CyclicCodes( 23, GF(2) );
[ a cyclic [23,23,1]0 enumerated code over GF(2),
  a cyclic [23,22,1..2]1 enumerated code over GF(2),
  a cyclic [23,11,1..8]4..7 enumerated code over GF(2),
  a cyclic [23,0,23]23 enumerated code over GF(2),
  a cyclic [23,11,1..8]4..7 enumerated code over GF(2),
  a cyclic [23,12,1..7]3 enumerated code over GF(2),
  a cyclic [23,1,23]11 enumerated code over GF(2),
  a cyclic [23,12,1..7]3 enumerated code over GF(2) ]
gap> BinaryGolayCode() in codelist;
true
gap> RepetitionCode( 23, GF(2) ) in codelist;
true
gap> CordaroWagnerCode( 23 ) in codelist;
false    # This code is not cyclic
```

5 Manipulating Codes

In this chapter we describe several functions GUAVA uses to manipulate codes. Some of the best codes are obtained by starting with for example a BCH code, and manipulating it.

In some cases, it is faster to perform calculations with a manipulated code than to use the original code. For example, if the dimension of the code is larger than half the word length, it is generally faster to compute the weight distribution by first calculating the weight distribution of the dual code than by directly calculating the weight distribution of the original code. The size of the dual code is smaller in these cases.

Because GUAVA keeps all information in a code record, in some cases the information can be preserved after manipulations. Therefore, computations do not always have to start from scratch.

In Section 5.1, we describe functions that take a code with certain parameters, modify it in some way and return a different code (see 5.1.1, 5.1.2, 5.1.4, 5.1.5, 5.1.6, 5.1.7, 5.1.9, 5.1.10, 5.1.11, 5.1.13, 5.1.14, 5.1.15, 5.1.16, 5.1.17, 5.1.19, 5.1.21 and 5.1.18).

In Section 5.2, we describe functions that generate a new code out of two codes (see 5.2.1, 5.2.2, 5.2.3, 5.2.4 and 5.2.5).

5.1 Functions that Generate a New Code from a Given Code

1 ► `ExtendedCode(` C `[,` i `] )`

`ExtendedCode` **extends** the code C i times and returns the result. i is equal to 1 by default. Extending is done by adding a parity check bit after the last coordinate. The coordinates of all codewords now add up to zero. In the binary case, each codeword has even weight.

The word length increases by i. The size of the code remains the same. In the binary case, the minimum distance increases by one if it was odd. In other cases, that is not always true.

A cyclic code in general is no longer cyclic after extending.

```
gap> C1 := HammingCode( 3, GF(2) );
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> C2 := ExtendedCode( C1 );
a linear [8,4,4]2 extended code
gap> IsEquivalent( C2, ReedMullerCode( 1, 3 ) );
true
gap> List( AsSSortedList( C2 ), WeightCodeword );
[ 0, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 8 ]
gap> C3 := EvenWeightSubcode( C1 );
a linear [7,3,4]2..3 even weight subcode
```

To undo extending, call `PuncturedCode` (see 5.1.2). The function `EvenWeightSubcode` (see 5.1.4) also returns a related code with only even weights, but without changing its word length.

2 ▶ `PuncturedCode( C )`

`PuncturedCode` **punctures** C in the last column, and returns the result. Puncturing is done simply by cutting off the last column from each codeword. This means the word length decreases by one. The minimum distance in general also decrease by one.

3 ▶ `PuncturedCode( C, L )`

`PuncturedCode` punctures C in the columns specified by L, a list of integers. All columns specified by L are omitted from each codeword. If l is the length of L (so the number of removed columns), the word length decreases by l. The minimum distance can also decrease by l or less.

Puncturing a cyclic code in general results in a non-cyclic code. If the code is punctured in all the columns where a word of minimal weight is unequal to zero, the dimension of the resulting code decreases.

```
gap> C1 := BCHCode( 15, 5, GF(2) );
a cyclic [15,7,5]3..5 BCH code, delta=5, b=1 over GF(2)
gap> C2 := PuncturedCode( C1 );
a linear [14,7,4]3..5 punctured code
gap> ExtendedCode( C2 ) = C1;
false
gap> PuncturedCode( C1, [1,2,3,4,5,6,7] );
a linear [8,7,1]1 punctured code
gap> PuncturedCode( WholeSpaceCode( 4, GF(5) ) );
a linear [3,3,1]0 punctured code  # The dimension decreased from 4 to 3
```

`ExtendedCode` extends the code again (see 5.1.1) although in general this does not result in the old code.

4 ▶ `EvenWeightSubcode( C )`

`EvenWeightSubcode` returns the **even weight subcode** of C, consisting of all codewords of C with even weight. If C is a linear code and contains words of odd weight, the resulting code has a dimension of one less. The minimum distance always increases with one if it was odd. If C is a binary cyclic code, and $g(x)$ is its generator polynomial, the even weight subcode either has generator polynomial $g(x)$ (if $g(x)$ is divisible by $x-1$) or $g(x)*(x-1)$ (if no factor $x-1$ was present in $g(x)$). So the even weight subcode is again cyclic.

Of course, if all codewords of C are already of even weight, the returned code is equal to C.

```
gap> C1 := EvenWeightSubcode( BCHCode( 8, 4, GF(3) ) );
an (8,33,4..8)3..8 even weight subcode
gap> List( AsSSortedList( C1 ), WeightCodeword );
[ 0, 4, 4, 4, 4, 4, 4, 6, 4, 4, 4, 4, 6, 4, 4, 6, 4, 4, 8, 6, 4, 6, 8, 4, 4,
  4, 6, 4, 6, 8, 4, 6, 8 ]
gap> EvenWeightSubcode( ReedMullerCode( 1, 3 ) );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
```

`ExtendedCode` also returns a related code of only even weights, but without reducing its dimension (see 5.1.1).

5 ▶ `PermutedCode( C, L )`

`PermutedCode` returns C after column permutations. L is the permutation to be executed on the columns of C. If C is cyclic, the result in general is no longer cyclic. If a permutation results in the same code as C, this permutation belongs to the **automorphism group** of C (see 3.4.3). In any case, the returned code is **equivalent** to C (see 3.4.1).

```
gap> C1 := PuncturedCode( ReedMullerCode( 1, 4 ) );
a linear [15,5,7]5 punctured code
gap> C2 := BCHCode( 15, 7, GF(2) );
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> C2 = C1;
false
gap> p := CodeIsomorphism( C1, C2 );
( 2, 4,14, 9,13, 7,11,10, 6, 8,12, 5)
gap> C3 := PermutedCode( C1, p );
a linear [15,5,7]5 permuted code
gap> C2 = C3;
true
```

6 ► `ExpurgatedCode(` *C*, *L* `)`

`ExpurgatedCode` **expurgates** code *C* by throwing away codewords in list *L*. *C* must be a linear code. *L* must be a list of codeword input. The generator matrix of the new code no longer is a basis for the codewords specified by *L*. Since the returned code is still linear, it is very likely that, besides the words of *L*, more codewords of *C* are no longer in the new code.

```
gap> C1 := HammingCode( 4 );; WeightDistribution( C1 );
[ 1, 0, 0, 35, 105, 168, 280, 435, 435, 280, 168, 105, 35, 0, 0, 1 ]
gap> L := Filtered( AsSSortedList(C1), i -> WeightCodeword(i) = 3 );;
gap> C2 := ExpurgatedCode( C1, L );
a linear [15,4,3..4]5..11 code, expurgated with 7 word(s)
gap> WeightDistribution( C2 );
[ 1, 0, 0, 0, 14, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 ]
```

This function does not work on non-linear codes. For removing words from a non-linear code, use `RemovedElementsCode` (see 5.1.9). For expurgating a code of all words of odd weight, use `EvenWeightSubcode` (see 5.1.4).

7 ► `AugmentedCode(` *C*, *L* `)`

`AugmentedCode` returns *C* after **augmenting**. *C* must be a linear code, *L* must be a list of codeword input. The generator matrix of the new code is a basis for the codewords specified by *L* as well as the words that were already in code *C*. Note that the new code in general will consist of more words than only the codewords of *C* and the words *L*. The returned code is also a linear code.

```
gap> C31 := ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> C32 := AugmentedCode(C31,["00000011","00000101","00010001"]);
a linear [8,7,1..2]1 code, augmented with 3 word(s)
gap> C32 = ReedMullerCode( 2, 3 );
true
```

8 ► `AugmentedCode(` *C* `)`

When called without a list of codewords, `AugmentedCode` returns *C* after adding the all-ones vector to the generator matrix. *C* must be a linear code. If the all-ones vector was already in the code, nothing happens and a copy of the argument is returned. If *C* is a binary code which does not contain the all-ones vector, the complement of all codewords is added.

```
gap> C1 := CordaroWagnerCode(6);
a linear [6,2,4]2..3 Cordaro-Wagner code over GF(2)
gap> Codeword( [0,0,1,1,1,1] ) in C1;
true
gap> C2 := AugmentedCode( C1 );
a linear [6,3,1..2]2..3 code, augmented with 1 word(s)
gap> Codeword( [1,1,0,0,0,0] ) in C2;
true
```

The function `AddedElementsCode` adds elements to the codewords instead of adding them to the basis (see 5.1.10).

9 ▶ `RemovedElementsCode(` C`,` L `)`

`RemovedElementsCode` returns code C after removing a list of codewords L from its elements. L must be a list of codeword input. The result is an unrestricted code.

```
gap> C1 := HammingCode( 4 );; WeightDistribution( C1 );
[ 1, 0, 0, 35, 105, 168, 280, 435, 435, 280, 168, 105, 35, 0, 0, 1 ]
gap> L := Filtered( AsSSortedList(C1), i -> WeightCodeword(i) = 3 );;
gap> C2 := RemovedElementsCode( C1, L );
a (15,2013,3..15)2..15 code with 35 word(s) removed
gap> WeightDistribution( C2 );
[ 1, 0, 0, 0, 105, 168, 280, 435, 435, 280, 168, 105, 35, 0, 0, 1 ]
gap> MinimumDistance( C2 );
3        # C2 is not linear, so the minimum weight does not have to
         # be equal to the minimum distance
```

Adding elements to a code is done by the function `AddedElementsCode` (see 5.1.10). To remove codewords from the base of a linear code, use `ExpurgatedCode` (see 5.1.6).

10 ▶ `AddedElementsCode(` C`,` L `)`

`AddedElementsCode` returns code C after adding a list of codewords L to its elements. L must be a list of codeword input. The result is an unrestricted code.

```
gap> C1 := NullCode( 6, GF(2) );
a cyclic [6,0,6]6 nullcode over GF(2)
gap> C2 := AddedElementsCode( C1, [ "111111" ] );
a (6,2,1..6)3 code with 1 word(s) added
gap> IsCyclicCode( C2 );
true
gap> C3 := AddedElementsCode( C2, [ "101010", "010101" ] );
a (6,4,1..6)2 code with 2 word(s) added
gap> IsCyclicCode( C3 );
true
```

To remove elements from a code, use `RemovedElementsCode` (see 5.1.9). To add elements to the base of a linear code, use `AugmentedCode` (see 5.1.7).

11 ▶ `ShortenedCode(` C `)`

`ShortenedCode` returns code C shortened by taking a cross section. If C is a linear code, this is done by removing all codewords that start with a non-zero entry, after which the first column is cut off. If C was a $[n, k, d]$ code, the shortened code generally is a $[n-1, k-1, d]$ code. It is possible that the dimension remains the same; it is also possible that the minimum distance increases.

```
gap> C1 := HammingCode( 4 );
a linear [15,11,3]1 Hamming (4,2) code over GF(2)
gap> C2 := ShortenedCode( C1 );
a linear [14,10,3]2 shortened code
```

If C is a non-linear code, `ShortenedCode` first checks which finite field element occurs most often in the first column of the codewords. The codewords not starting with this element are removed from the code, after which the first column is cut off. The resulting shortened code has at least the same minimum distance as C.

```
gap> C1 := ElementsCode( ["1000", "1101", "0011" ], GF(2) );
a (4,3,1..4)2 user defined unrestricted code over GF(2)
gap> MinimumDistance( C1 );
2
gap> C2 := ShortenedCode( C1 );
a (3,2,2..3)1..2 shortened code
gap> AsSSortedList( C2 );
[ [ 0 0 0 ], [ 1 0 1 ] ]
```

12 ► `ShortenedCode(` C`,` L `)`

When called in this format, `ShortenedCode` repeats the shortening process on each of the columns specified by L. L therefore is a list of integers. The column numbers in L are the numbers as they are **before** the shortening process. If L has l entries, the returned code has a word length of l positions shorter than C.

```
gap> C1 := HammingCode( 5, GF(2) );
a linear [31,26,3]1 Hamming (5,2) code over GF(2)
gap> C2 := ShortenedCode( C1, [ 1, 2, 3 ] );
a linear [28,23,3]2 shortened code
gap> OptimalityLinearCode( C2 );
0
```

The function `LengthenedCode` lengthens the code again (only for linear codes), see 5.1.13. In general, this is not exactly the inverse function.

13 ► `LengthenedCode(` C `[,` i `] )`

`LengthenedCode` returns code C lengthened. C must be a linear code. First, the all-ones vector is added to the generator matrix (see 5.1.7). If the all-ones vector was already a codeword, nothing happens to the code. Then, the code is extended i times (see 5.1.1). i is equal to 1 by default. If C was an $[n,k]$ code, the new code generally is a $[n+i,k+1]$ code.

```
gap> C1 := CordaroWagnerCode( 5 );
a linear [5,2,3]2 Cordaro-Wagner code over GF(2)
gap> C2 := LengthenedCode( C1 );
a linear [6,3,2]2..3 code, lengthened with 1 column(s)
```

`ShortenedCode` shortens the code, see 5.1.11. In general, this is not exactly the inverse function.

14 ► `ResidueCode(` C `[,` w `] )`

The function `ResidueCode` takes a codeword c of C of weight w (if w is omitted, a codeword of minimal weight is used). C must be a linear code and w must be greater than zero. It removes this word and all its linear combinations from the code and then punctures the code in the coordinates where c is unequal to zero. The resulting code is an $[n-w, k-1, d-\lfloor w*(q-1)/q \rfloor]$ code.

```
gap> C1 := BCHCode( 15, 7 );
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> C2 := ResidueCode( C1 );
a linear [8,4,4]2 residue code
gap> c := Codeword( [ 0,0,0,1,0,0,1,1,0,1,0,1,1,1,1 ], C1);;
gap> C3 := ResidueCode( C1, c );
a linear [7,4,3]1 residue code
```

15 ► `ConstructionBCode(` C `)`

The function `ConstructionBCode` takes a binary linear code C and calculates the minimum distance of the dual of C (see 5.1.16). It then removes the columns of the parity check matrix of C where a codeword of the dual code of minimal weight has coordinates unequal to zero. the resulting matrix is a parity check matrix for an $[n-dd, k-dd+1, \geq d]$ code, where dd is the minimum distance of the dual of C.

```
gap> C1 := ReedMullerCode( 2, 5 );
a linear [32,16,8]6 Reed-Muller (2,5) code over GF(2)
gap> C2 := ConstructionBCode( C1 );
a linear [24,9,8]5..10 Construction B (8 coordinates)
gap> BoundsMinimumDistance( 24, 9, GF(2) );
rec( n := 24, k := 9, q := 2, references := rec(  ),
  construction := [ <Operation "UUVCode">,
      [ [ <Operation "UUVCode">, [ [ <Operation "DualCode">,
                      [ [ <Operation "RepetitionCode">, [ 6, 2 ] ] ] ],
                  [ <Operation "CordaroWagnerCode">, [ 6 ] ] ] ],
          [ <Operation "CordaroWagnerCode">, [ 12 ] ] ] ], lowerBound := 8,
  lowerBoundExplanation := [ "Lb(24,9)=8, u u+v construction of C1 and C2:",
      "Lb(12,7)=4, u u+v construction of C1 and C2:",
      "Lb(6,5)=2, dual of the repetition code",
      "Lb(6,2)=4, Cordaro-Wagner code", "Lb(12,2)=8, Cordaro-Wagner code" ],
  upperBound := 8,
  upperBoundExplanation := [ "Ub(24,9)=8, otherwise construction B would contr\
adict:", "Ub(18,4)=8, Griesmer bound" ] )
# so C2 is optimal
```

16 ► `DualCode(` C `)`

`DualCode` returns the dual code of C. The dual code consists of all codewords that are orthogonal to the codewords of C. If C is a linear code with generator matrix G, the dual code has parity check matrix G (or if C has parity check matrix H, the dual code has generator matrix H). So if C is a linear [n, k] code, the dual code of C is a linear [n, n-k] code. If C is a cyclic code with generator polynomial $g(x)$, the dual code has the reciprocal polynomial of $g(x)$ as check polynomial.

The dual code is always a linear code, even if C is non-linear.

If a code C is equal to its dual code, it is called **self-dual**.

```
gap> R := ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> RD := DualCode( R );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> R = RD;
true
gap> N := WholeSpaceCode( 7, GF(4) );
a cyclic [7,7,1]0 whole space code over GF(4)
gap> DualCode( N ) = NullCode( 7, GF(4) );
```

```
true
```

17 ▶ `ConversionFieldCode( C )`

`ConversionFieldCode` returns code C after converting its field. If the field of C is `GF(`q^m`)`, the returned code has field `GF(`q`)`. Each symbol of every codeword is replaced by a concatenation of m symbols from `GF(`q`)`. If C is an (n, M, d_1) code, the returned code is a $(n * m, M, d_2)$ code, where $d_2 > d_1$.

See also 6.2.10.

```
gap> R := RepetitionCode( 4, GF(4) );
a cyclic [4,1,4]3 repetition code over GF(4)
gap> R2 := ConversionFieldCode( R );
a linear [8,2,4]3..4 code, converted to basefield GF(2)
gap> Size( R ) = Size( R2 );
true
gap> GeneratorMat( R );
[ [ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ] ]
gap> GeneratorMat( R2 );
[ [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ] ]
```

18 ▶ `CosetCode( C, w )`

`CosetCode` returns the coset of a code C with respect to word w. w must be of the codeword type. Then, w is added to each codeword of C, yielding the elements of the new code. If C is linear and w is an element of C, the new code is equal to C, otherwise the new code is an unrestricted code.

Generating a coset is also possible by simply adding the word w to C. See 3.2.

```
gap> H := HammingCode(3, GF(2));
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> c := Codeword("1011011");; c in H;
false
gap> C := CosetCode(H, c);
a (7,16,3)1 coset code
gap> List(AsSSortedList(C), el-> Syndrome(H, el));
[ [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ],
  [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ],
  [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ] ]
# All elements of the coset have the same syndrome in H
```

19 ▶ `ConstantWeightSubcode( C, w )`

`ConstantWeightSubcode` returns the subcode of C that only has codewords of weight w. The resulting code is a non-linear code, because it does not contain the all-zero vector.

```
gap> N := NordstromRobinsonCode();; WeightDistribution(N);
[ 1, 0, 0, 0, 0, 0, 112, 0, 30, 0, 112, 0, 0, 0, 0, 0, 1 ]
gap> C := ConstantWeightSubcode(N, 8);
a (16,30,6..16)5..8 code with codewords of weight 8
gap> WeightDistribution(C);
[ 0, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0, 0, 0, 0, 0, 0, 0 ]
```

20 ▶ `ConstantWeightSubcode( C )`

In this format, `ConstantWeightSubcode` returns the subcode of C consisting of all minimum weight codewords of C.

```
gap> eg := ExtendedTernaryGolayCode();; WeightDistribution(eg);
[ 1, 0, 0, 0, 0, 0, 264, 0, 0, 440, 0, 0, 24 ]
gap> C := ConstantWeightSubcode(eg);
a (12,264,6..12)3..6 code with codewords of weight 6
gap> WeightDistribution(C);
[ 0, 0, 0, 0, 0, 0, 264, 0, 0, 0, 0, 0, 0 ]
```

21 ► `StandardFormCode(` *C* `)`

`StandardFormCode` returns C after putting it in standard form. If C is a non-linear code, this means the elements are organized using lexicographical order. This means they form a legal GAP `Set`.

If C is a linear code, the generator matrix and parity check matrix are put in standard form. The generator matrix then has an identity matrix in its left part, the parity check matrix has an identity matrix in its right part. Although GUAVA always puts both matrices in a standard form using `BaseMat`, this never alters the code. `StandardFormCode` even applies column permutations if unavoidable, and thereby changes the code. The column permutations are recorded in the construction history of the new code (see 3.6.3). C and the new code are of course equivalent.

If C is a cyclic code, its generator matrix cannot be put in the usual upper triangular form, because then it would be inconsistent with the generator polynomial. The reason is that generating the elements from the generator matrix would result in a different order than generating the elements from the generator polynomial. This is an unwanted effect, and therefore `StandardFormCode` just returns a copy of C for cyclic codes.

```
gap> G := GeneratorMatCode( Z(2) * [ [0,1,1,0], [0,1,0,1], [0,0,1,1] ],
> "random form code", GF(2) );
a linear [4,2,1..2]1..2 random form code over GF(2)
gap> Codeword( GeneratorMat( G ) );
[ [ 0 1 0 1 ], [ 0 0 1 1 ] ]
gap> Codeword( GeneratorMat( StandardFormCode( G ) ) );
[ [ 1 0 0 1 ], [ 0 1 0 1 ] ]
```

5.2 Functions that Generate a New Code from Two Given Codes

1 ► `DirectSumCode(` *C_1*, *C_2* `)`

`DirectSumCode` returns the direct sum of codes C_1 and C_2. The direct sum code consists of every codeword of C_1 concatenated by every codeword of C_2. Therefore, if C_i was a (n_i, M_i, d_i) code, the result is a $(n_1 + n_2, M_1 * M_2, min(d_1, d_2))$ code.

If both C_1 and C_2 are linear codes, the result is also a linear code. If one of them is non-linear, the direct sum is non-linear too. In general, a direct sum code is not cyclic.

Performing a direct sum can also be done by adding two codes (see Sevction 3.2). Another often used method is the "u, u+v"-construction, described in 5.2.2.

```
gap> C1 := ElementsCode( [ [1,0], [4,5] ], GF(7) );;
gap> C2 := ElementsCode( [ [0,0,0], [3,3,3] ], GF(7) );;
gap> D := DirectSumCode(C1, C2);;
gap> AsSSortedList(D);
[ [ 1 0 0 0 0 ], [ 1 0 3 3 3 ], [ 4 5 0 0 0 ], [ 4 5 3 3 3 ] ]
gap> D = C1 + C2;   # addition = direct sum
true
```

2 ► `UUVCode(` *C_1*, *C_2* `)`

`UUVCode` returns the so-called $(u|u + v)$ construction applied to C_1 and C_2. The resulting code consists of every codeword u of C_1 concatenated by the sum of u and every codeword v of C_2. If C_1 and C_2

have different word lengths, sufficient zeros are added to the shorter code to make this sum possible. If C_i is a (n_i, M_i, d_i) code, the result is a $(n_1 + max(n_1, n_2), M_1 * M_2, min(2 * d_1, d_2))$ code.

If both C_1 and C_2 are linear codes, the result is also a linear code. If one of them is non-linear, the UUV sum is non-linear too. In general, a UUV sum code is not cyclic.

The function `DirectSumCode` returns another sum of codes (see 5.2.1).

```
gap> C1 := EvenWeightSubcode(WholeSpaceCode(4, GF(2)));
a cyclic [4,3,2]1 even weight subcode
gap> C2 := RepetitionCode(4, GF(2));
a cyclic [4,1,4]2 repetition code over GF(2)
gap> R := UUVCode(C1, C2);
a linear [8,4,4]2 U U+V construction code
gap> R = ReedMullerCode(1,3);
true
```

3 ► `DirectProductCode(` C_1, C_2 `)`

`DirectProductCode` returns the direct product of codes C_1 and C_2. Both must be linear codes. Suppose C_i has generator matrix G_i. The direct product of C_1 and C_2 then has the Kronecker product of G_1 and G_2 as the generator matrix (see `KroneckerProduct`).

If C_i is a $[n_i, k_i, d_i]$ code, the direct product then is a $[n_1 * n_2, k_1 * k_2, d_1 * d_2]$ code.

```
gap> L1 := LexiCode(10, 4, GF(2));
a linear [10,5,4]2..4 lexicode over GF(2)
gap> L2 := LexiCode(8, 3, GF(2));
a linear [8,4,3]2..3 lexicode over GF(2)
gap> D := DirectProductCode(L1, L2);
a linear [80,20,12]20..45 direct product code
```

4 ► `IntersectionCode(` C_1, C_2 `)`

`IntersectionCode` returns the intersection of codes C_1 and C_2. This code consists of all codewords that are both in C_1 and C_2. If both codes are linear, the result is also linear. If both are cyclic, the result is also cyclic.

```
gap> C := CyclicCodes(7, GF(2));
[ a cyclic [7,7,1]0 enumerated code over GF(2),
  a cyclic [7,6,1..2]1 enumerated code over GF(2),
  a cyclic [7,3,1..4]2..3 enumerated code over GF(2),
  a cyclic [7,0,7]7 enumerated code over GF(2),
  a cyclic [7,3,1..4]2..3 enumerated code over GF(2),
  a cyclic [7,4,1..3]1 enumerated code over GF(2),
  a cyclic [7,1,7]3 enumerated code over GF(2),
  a cyclic [7,4,1..3]1 enumerated code over GF(2) ]
gap> IntersectionCode(C[6], C[8]) = C[7];
true
```

5 ► `UnionCode(` C_1, C_2 `)`

`UnionCode` returns the union of codes C_1 and C_2. This code consists of the union of all codewords of C_1 and C_2 and all linear combinations. Therefore this function works only for linear codes. The function `AddedElementsCode` can be used for non-linear codes, or if the resulting code should not include linear combinations. See 5.1.10. If both arguments are cyclic, the result is also cyclic.

```
gap> G := GeneratorMatCode([[1,0,1],[0,1,1]]*Z(2)^0, GF(2));
a linear [3,2,1..2]1 code defined by generator matrix over GF(2)
gap> H := GeneratorMatCode([[1,1,1]]*Z(2)^0, GF(2));
a linear [3,1,3]1 code defined by generator matrix over GF(2)
gap> U := UnionCode(G, H);
a linear [3,3,1]0 union code
gap> c := Codeword("010");; c in G;
false
gap> c in H;
false
gap> c in U;
true
```

6 Bounds on Codes, Special Matrices and Miscellaneous Functions

In this chapter we describe functions that determine bounds on the size and minimum distance of codes (Section 6.1), functions that work with special matrices GUAVA needs for several codes (see Section 6.2), and constructing codes or performing calculations with codes (see Section 6.3).

6.1 Bounds on codes

This section describes the functions that calculate estimates for upper bounds on the size and minimum distance of codes. Several algorithms are known to compute a largest number of words a code can have with given length and minimum distance. It is important however to understand that in some cases the true upper bound is unknown. A code which has a size equal to the calculated upper bound may not have been found. However, codes that have a larger size do not exist.

A second way to obtain bounds is a table. In GUAVA, an extensive table is implemented for linear codes over GF(2), GF(3) and GF(4). It contains bounds on the minimum distance for given word length and dimension. For binary codes, it contains entries for word length less than or equal to 257. For codes over $GF(3)$ and $GF(4)$, it contains entries for word length less than or equal to 130.

Firstly, we describe functions that compute specific upper bounds on the code size (see 6.1.1, 6.1.2, 6.1.3, 6.1.4, 6.1.5 and 6.1.6).

Next we describe a function that computes GUAVA's best upper bound on the code size (see 6.1.7).

Then we describe two functions that compute a lower and upper bound on the minimum distance of a code (see 6.1.8 and 6.1.10).

Finally, we describe a function that returns a lower and upper bound on the minimum distance with given parameters and a description of how the bounds were obtained (see 6.1.12).

1 ▶ `UpperBoundSingleton(` n, d, q `)`

`UpperBoundSingleton` returns the Singleton bound for a code of length n, minimum distance d over a field of size q. This bound is based on the shortening of codes. By shortening an (n,M,d) code $d-1$ times, an $(n-d+1,M,1)$ code results, with $M \leq q^{n-d+1}$ (see 5.1.11). Thus

$$M \leq q^{n-d+1}$$

Codes that meet this bound are called **maximum distance separable** (see 3.3.5).

```
gap> UpperBoundSingleton(4, 3, 5);
25
gap> C := ReedSolomonCode(4,3);; Size(C);
25
gap> IsMDSCode(C);
true
```

2 ► `UpperBoundHamming(` n, d, q `)`

The Hamming bound (also known as **sphere packing bound**) returns an upper bound on the size of a code of length n, minimum distance d, over a field of size q. The Hamming bound is obtained by dividing the contents of the entire space $GF(q)^n$ by the contents of a ball with radius $\lfloor (d-1)/2 \rfloor$. As all these balls are disjoint, they can never contain more than the whole vector space.

$$M \leq \frac{q^n}{V(n,e)}$$

where M is the maxmimum number of codewords and $V(n,e)$ is equal to the contents of a ball of radius e (see 6.3.1). This bound is useful for small values of d. Codes for which equality holds are called **perfect** (see 3.3.4).

```
gap> UpperBoundHamming( 15, 3, 2 );
2048
gap> C := HammingCode( 4, GF(2) );
a linear [15,11,3]1 Hamming (4,2) code over GF(2)
gap> Size( C );
2048
```

3 ► `UpperBoundJohnson(` n, d `)`

The Johnson bound is an improved version of the Hamming bound (see 6.1.2). In addition to the Hamming bound, it takes into account the elements of the space outside the balls of radius e around the elements of the code. The Johnson bound only works for binary codes.

```
gap> UpperBoundJohnson( 13, 5 );
77
gap> UpperBoundHamming( 13, 5, 2);
89   # in this case the Johnson bound is better
```

4 ► `UpperBoundPlotkin(` n, d, q `)`

The function `UpperBoundPlotkin` calculates the sum of the distances of all ordered pairs of different codewords. It is based on the fact that the minimum distance is at most equal to the average distance. It is a good bound if the weights of the codewords do not differ much. It results in:

$$M \leq \frac{d}{d-(1-1/q)n}$$

M is the maximum number of codewords. In this case, d must be larger than $(1-1/q)n$, but by shortening the code, the case $d < (1-1/q)n$ is covered.

```
gap> UpperBoundPlotkin( 15, 7, 2 );
32
gap> C := BCHCode( 15, 7, GF(2) );
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> Size(C);
32
gap> WeightDistribution(C);
[ 1, 0, 0, 0, 0, 0, 0, 15, 15, 0, 0, 0, 0, 0, 0, 1 ]
```

5 ► `UpperBoundElias(` n, d, q `)`

The Elias bound is an improvement of the Plotkin bound (see 6.1.4) for large codes. Subcodes are used to decrease the size of the code, in this case the subcode of all codewords within a certain ball. This bound is useful for large codes with relatively small minimum distances.

```
gap> UpperBoundPlotkin( 16, 3, 2 );
12288
gap> UpperBoundElias( 16, 3, 2 );
10280
```

6 ► `UpperBoundGriesmer(` n, d, q `)`

The Griesmer bound is valid only for linear codes. It is obtained by counting the number of equal symbols in each row of the generator matrix of the code. By omitting the coordinates in which all rows have a zero, a smaller code results. The Griesmer bound is obtained by repeating this proces until a trivial code is left in the end.

```
gap> UpperBoundGriesmer( 13, 5, 2 );
64
gap> UpperBoundGriesmer( 18, 9, 2 );
8        # the maximum number of words for a linear code is 8
gap> Size( PuncturedCode( HadamardCode( 20, 1 ) ) );
20       # this non-linear code has 20 elements
```

7 ► `UpperBound(` n, d, q `)`

`UpperBound` returns the best known upper bound $A(n,d)$ for the size of a code of length n, minimum distance d over a field of size q. The function `UpperBound` first checks for trivial cases (like $d = 1$ or $n = d$) and if the value is in the built-in table. Then it calculates the minimum value of the upper bound using the methods of Singleton (see 6.1.1), Hamming (see 6.1.2), Johnson (see 6.1.3), Plotkin (see 6.1.4) and Elias (see 6.1.5). If the code is binary, $A(n, 2*l-1) = A(n+1, 2*l)$, so the `UpperBound` takes the minimum of the values obtained from all methods for the parameters $(n, 2*l-1)$ and $(n+1, 2*l)$.

```
gap> UpperBound( 10, 3, 2 );
85
gap> UpperBound( 25, 9, 8 );
1211778792827540
```

8 ► `LowerBoundMinimumDistance(` C `)`

In this form, `LowerBoundMinimumDistance` returns a lower bound for the minimum distance of code C.

```
gap> C := BCHCode( 45, 7 );
a cyclic [45,23,7..9]6..16 BCH code, delta=7, b=1 over GF(2)
gap> LowerBoundMinimumDistance( C );
7     # designed distance is lower bound for minimum distance
```

9 ► `LowerBoundMinimumDistance(` n, k, F `)`

In this form, `LowerBoundMinimumDistance` returns a lower bound for the minimum distance of the best known linear code of length n, dimension k over field F. It uses the mechanism explained in section 6.1.12.

```
gap> LowerBoundMinimumDistance( 45, 23, GF(2) );
10
```

10 ► `UpperBoundMinimumDistance(` C `)`

In this form, `UpperBoundMinimumDistance` returns an upper bound for the minimum distance of code C. For unrestricted codes, it just returns the word length. For linear codes, it takes the minimum of the possibly known value from the method of construction, the weight of the generators, and the value from the table (see 6.1.12).

```
gap> C := BCHCode( 45, 7 );;
gap> UpperBoundMinimumDistance( C );
9
```

11 ► `UpperBoundMinimumDistance(` n, k, F `)`

In this form, `UpperBoundMinimumDistance` returns an upper bound for the minimum distance of the best known linear code of length n, dimension k over field F. It uses the mechanism explained in section 6.1.12.

```
gap> UpperBoundMinimumDistance( 45, 23, GF(2) );
11
```

12 ► `BoundsMinimumDistance(` n, k, F `)`

The function `BoundsMinimumDistance` calculates a lower and upper bound for the minimum distance of an optimal linear code with word length n, dimension k over field F. The function returns a record with the two bounds and an explenation for each bound. The function `Display` can be used to show the explanations.

The values for the lower and upper bound are obtained from a table. GUAVA has tables containing lower and upper bounds for $q = 2 (n \leq 257), 3$ and $4 (n \leq 130)$. These tables were derived from the table of Brouwer & Verhoeff. For codes over other fields and for larger word lengths, trivial bounds are used.

The resulting record can be used in the function `BestKnownLinearCode` (see 4.2.12) to construct a code with minimum distance equal to the lower bound.

```
gap> bounds := BoundsMinimumDistance( 7, 3 );; DisplayBoundsInfo( bounds );
an optimal linear [7,3,d] code over GF(2) has d=4
--------------------------------------------------------------------------------
Lb(7,3)=4, by shortening of:
Lb(8,4)=4, u u+v construction of C1 and C2:
Lb(4,3)=2, dual of the repetition code
Lb(4,1)=4, repetition code
--------------------------------------------------------------------------------
Ub(7,3)=4, Griesmer bound
# The lower bound is equal to the upper bound, so a code with
# these parameters is optimal.
gap> C := BestKnownLinearCode( bounds );; Display( C );
a linear [7,3,4]2..3 shortened code of
```

```
a linear [8,4,4]2 U U+V construction code of
U: a cyclic [4,3,2]1 dual code of
   a cyclic [4,1,4]2 repetition code over GF(2)
V: a cyclic [4,1,4]2 repetition code over GF(2)
```

6.2 Special matrices in GUAVA

This section explains functions that work with special matrices GUAVA needs for several codes.

Firstly, we describe some matrix generating functions (see 6.2.1, 6.2.2, 6.2.3, 6.2.4 and 6.2.5).

Next we describe two functions regarding a standard form of matrices (see 6.2.6 and 6.2.7).

Then we describe functions that return a matrix after a manipulation (see 6.2.8, 6.2.9 and 6.2.10).

Finally, we describe functions that do some tests on matrices (see 6.2.11 and 6.2.12).

1 ▶ `KrawtchoukMat(` n `,` q `)`

`KrawtchoukMat` returns the $n+1$ by $n+1$ matrix $K = (k_{ij})$ defined by $k_{ij} = K_i(j)$ for $i, j = 0, \cdots, n$. $K_i(j)$ is the Krawtchouk number (see 6.3.2). n must be a positive integer and q a prime power. The Krawtchouk matrix is used in the **MacWilliams identities**, defining the relation between the weight distribution of a code of length n over a field of size q, and its dual code. Each call to `KrawtchoukMat` returns a new matrix, so it is safe to modify the result.

```
gap> PrintArray( KrawtchoukMat( 3, 2 ) );
[ [   1,   1,   1,   1 ],
  [   3,   1,  -1,  -3 ],
  [   3,  -1,  -1,   3 ],
  [   1,  -1,   1,  -1 ] ]
gap> C := HammingCode( 3 );; a := WeightDistribution( C );
[ 1, 0, 0, 7, 7, 0, 0, 1 ]
gap> n := WordLength( C );; q := Size( LeftActingDomain( C ) );;
gap> k := Dimension( C );;
gap> q^( -k ) * KrawtchoukMat( n, q ) * a;
[ 1, 0, 0, 0, 7, 0, 0, 0 ]
gap> WeightDistribution( DualCode( C ) );
[ 1, 0, 0, 0, 7, 0, 0, 0 ]
```

2 ▶ `GrayMat(` n`,` F `)`

`GrayMat` returns a list of all different vectors (see `Vectors`) of length n over the field F, using Gray ordening. n must be a positive integer. This order has the property that subsequent vectors differ in exactly one coordinate. The first vector is always the null vector. Each call to `GrayMat` returns a new matrix, so it is safe to modify the result.

```
gap> GrayMat(3);
[ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ Z(2)^0, Z(2)^0, Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ] ]
gap> G := GrayMat( 4, GF(4) );; Length(G);
256          # the length of a GrayMat is always $q^n$
gap> G[101] - G[100];
[ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ]
```

3 ► `SylvesterMat( n )`

`SylvesterMat` returns the n by n Sylvester matrix of order n. This is a special case of the Hadamard matrices (see 6.2.4). For this construction, n must be a power of 2. Each call to `SylvesterMat` returns a new matrix, so it is safe to modify the result.

```
gap> PrintArray(SylvesterMat(2));
[ [   1,   1 ],
  [   1,  -1 ] ]
gap> PrintArray( SylvesterMat(4) );
[ [   1,   1,   1,   1 ],
  [   1,  -1,   1,  -1 ],
  [   1,   1,  -1,  -1 ],
  [   1,  -1,  -1,   1 ] ]
```

4 ► `HadamardMat( n )`

`HadamardMat` returns a Hadamard matrix of order n. This is an n by n matrix with the property that the matrix multiplied by its transpose returns n times the identity matrix. This is only possible for $n = 1, n = 2$ or in cases where n is a multiple of 4. If the matrix does not exist or is not known, `HadamardMat` returns an error. A large number of construction methods is known to create these matrices for different orders. `HadamardMat` makes use of two construction methods (among which the Sylvester construction (see 6.2.3)). These methods cover most of the possible Hadamard matrices, although some special algorithms have not been implemented yet. The following orders less than 100 do not have an implementation for a Hadamard matrix in GUAVA:$28, 36, 52, 76, 92\cdot$

```
gap> C := HadamardMat(8);; PrintArray(C);
[ [   1,   1,   1,   1,   1,   1,   1,   1 ],
  [   1,  -1,   1,  -1,   1,  -1,   1,  -1 ],
  [   1,   1,  -1,  -1,   1,   1,  -1,  -1 ],
  [   1,  -1,  -1,   1,   1,  -1,  -1,   1 ],
  [   1,   1,   1,   1,  -1,  -1,  -1,  -1 ],
  [   1,  -1,   1,  -1,  -1,   1,  -1,   1 ],
  [   1,   1,  -1,  -1,  -1,  -1,   1,   1 ],
  [   1,  -1,  -1,   1,  -1,   1,   1,  -1 ] ]
gap> C * TransposedMat(C) = 8 * IdentityMat( 8, 8 );
true
```

5 ► `MOLS( q )`
► `MOLS( q, n )`

`MOLS` returns a list of n **Mutually Orthogonal Latin Squares** (**MOLS**). A **Latin square** of order q is a q by q matrix whose entries are from a set F_q of q distinct symbols (GUAVA uses the integers from 0 to q) such that each row and each column of the matrix contains each symbol exactly once.

A set of Latin squares is a set of MOLS if and only if for each pair of Latin squares in this set, every ordered pair of elements that are in the same position in these matrices occurs exactly once.

n must be less than q. If n is omitted, two MOLS are returned. If q is not a prime power, at most 2 MOLS can be created. For all values of q with $q > 2$ and $q \neq 6$, a list of MOLS can be constructed. GUAVA however does not yet construct MOLS for $q \bmod 4 = 2$. If it is not possible to construct n MOLS, the function returns `false`.

MOLS are used to create q-ary codes (see 4.1.6).

```
gap> M := MOLS( 4, 3 );;PrintArray( M[1] );
[ [  0,  1,  2,  3 ],
  [  1,  0,  3,  2 ],
  [  2,  3,  0,  1 ],
  [  3,  2,  1,  0 ] ]
gap> PrintArray( M[2] );
[ [  0,  2,  3,  1 ],
  [  1,  3,  2,  0 ],
  [  2,  0,  1,  3 ],
  [  3,  1,  0,  2 ] ]
gap> PrintArray( M[3] );
[ [  0,  3,  1,  2 ],
  [  1,  2,  0,  3 ],
  [  2,  1,  3,  0 ],
  [  3,  0,  2,  1 ] ]
gap> MOLS( 12, 3 );
false
```

6 ► `PutStandardForm(` *M* `)`
► `PutStandardForm(` *M*, *idleft* `)`

`PutStandardForm` puts a matrix M in standard form, and returns the permutation needed to do so. *idleft* is a boolean that sets the position of the identity matrix in M. If *idleft* is set to `true`, the identity matrix is put in the left side of M. Otherwise, it is put at the right side. The default for *idleft* is `true`.

The function `BaseMat` also returns a similar standard form, but does not apply column permutations. The rows of the matrix still span the same vector space after `BaseMat`, but after calling `PutStandardForm`, this is not necessarily true.

```
gap> M := Z(2)*[[1,0,0,1],[0,0,1,1]];; PrintArray(M);
[ [    Z(2),  0*Z(2),  0*Z(2),    Z(2) ],
  [  0*Z(2),  0*Z(2),    Z(2),    Z(2) ] ]
gap> PutStandardForm(M);                    # identity at the left side
(2,3)
gap> PrintArray(M);
[ [    Z(2),  0*Z(2),  0*Z(2),    Z(2) ],
  [  0*Z(2),    Z(2),  0*Z(2),    Z(2) ] ]
gap> PutStandardForm(M, false);             # identity at the right side
(1,4,3)
gap> PrintArray(M);
[ [  0*Z(2),    Z(2),    Z(2),  0*Z(2) ],
  [  0*Z(2),    Z(2),  0*Z(2),    Z(2) ] ]
```

7 ► `IsInStandardForm(` *M* `)`
► `IsInStandardForm(` *M*, *idleft* `)`

`IsInStandardForm` determines if M is in standard form. *idleft* is a boolean that indicates the position of the identity matrix in M. If *idleft* is `true`, `IsInStandardForm` checks if the identity matrix is at the left side of M, otherwise if it is at the right side. The default for *idleft* is `true`. The elements of M may be elements of any field. To put a matrix in standard form, use `PutStandardForm` (see 6.2.6).

```
gap> IsInStandardForm(IdentityMat(7, GF(2)));
true
gap> IsInStandardForm([[1, 1, 0], [1, 0, 1]], false);
true
gap> IsInStandardForm([[1, 3, 2, 7]]);
true
gap> IsInStandardForm(HadamardMat(4));
false
```

8 ► `PermutedCols(` M `,` P `)`

`PermutedCols` returns a matrix M with a permutation P applied to its columns.

```
gap> M := [[1,2,3,4],[1,2,3,4]];; PrintArray(M);
[ [  1,  2,  3,  4 ],
  [  1,  2,  3,  4 ] ]
gap> PrintArray(PermutedCols(M, (1,2,3)));
[ [  3,  1,  2,  4 ],
  [  3,  1,  2,  4 ] ]
```

9 ► `VerticalConversionFieldMat(` M `,` F `)`

`VerticalConversionFieldMat` returns the matrix M with its elements converted from a field $F = GF(q^m)$, q prime, to a field $GF(q)$. Each element is replaced by its representation over the latter field, placed vertically in the matrix.

If M is a k by n matrix, the result is a $k*m$ by n matrix, since each element of $GF(q^m)$ can be represented in $GF(q)$ using m elements.

```
gap> M := Z(9)*[[1,2],[2,1]];; PrintArray(M);
[ [    Z(3^2),  Z(3^2)^5 ],
  [  Z(3^2)^5,    Z(3^2) ] ]
gap> DefaultField( Flat(M) );
GF(3^2)
gap> VCFM := VerticalConversionFieldMat( M, GF(9) );; PrintArray(VCFM);
[ [  0*Z(3),  0*Z(3) ],
  [  Z(3)^0,    Z(3) ],
  [  0*Z(3),  0*Z(3) ],
  [    Z(3),  Z(3)^0 ] ]
gap> DefaultField( Flat(VCFM) );
GF(3)
```

A similar function is `HorizontalConversionFieldMat` (see 6.2.10).

10 ► `HorizontalConversionFieldMat(` M `,` F `)`

`HorizontalConversionFieldMat` returns the matrix M with its elements converted from a field $F = GF(q^m)$, q prime, to a field $GF(q)$. Each element is replaced by its representation over the latter field, placed horizontally in the matrix.

If M is a k by n matrix, the result is a $k*m$ by $n*m$ matrix. The new word length of the resulting code is equal to $n*m$, because each element of $GF(q^m)$ can be represented in $GF(q)$ using m elements. The new dimension is equal to $k*m$ because the new matrix should be a basis for the same number of vectors as the old one.

`ConversionFieldCode` uses horizontal conversion to convert a code (see 5.1.17).

```
gap> M := Z(9)*[[1,2],[2,1]];; PrintArray(M);
[ [    Z(3^2),  Z(3^2)^5 ],
  [  Z(3^2)^5,    Z(3^2) ] ]
gap> DefaultField( Flat(M) );
GF(3^2)
gap> HCFM := HorizontalConversionFieldMat(M, GF(9));; PrintArray(HCFM);
[ [  0*Z(3),  Z(3)^0,  0*Z(3),    Z(3) ],
  [  Z(3)^0,  Z(3)^0,    Z(3),    Z(3) ],
  [  0*Z(3),    Z(3),  0*Z(3),  Z(3)^0 ],
  [    Z(3),    Z(3),  Z(3)^0,  Z(3)^0 ] ]
gap> DefaultField( Flat(HCFM) );
GF(3)
```

A similar function is `VerticalConversionFieldMat` (see 6.2.9).

11 ► `IsLatinSquare(` M `)`

`IsLatinSquare` determines if a matrix M is a latin square. For a latin square of size n by n, each row and each column contains all the integers $1, \ldots, n$ exactly once.

```
gap> IsLatinSquare([[1,2],[2,1]]);
true
gap> IsLatinSquare([[1,2,3],[2,3,1],[1,3,2]]);
false
```

12 ► `AreMOLS(` L `)`

`AreMOLS` determines if L is a list of mutually orthogonal latin squares (MOLS). For each pair of latin squares in this list, the function checks if each ordered pair of elements that are in the same position in these matrices occurs exactly once. The function `MOLS` creates MOLS (see 6.2.5).

```
gap> M := MOLS(4,2);
[ [ [ 0, 1, 2, 3 ], [ 1, 0, 3, 2 ], [ 2, 3, 0, 1 ], [ 3, 2, 1, 0 ] ],
  [ [ 0, 2, 3, 1 ], [ 1, 3, 2, 0 ], [ 2, 0, 1, 3 ], [ 3, 1, 0, 2 ] ] ]
gap> AreMOLS(M);
true
```

6.3 Miscellaneous functions

In this section we describe several functions GUAVA uses for constructing codes or performing calculations with codes.

1 ► `SphereContent(` n, t, F `)`

`SphereContent` returns the content of a ball of radius t around an arbitrary element of the vectorspace F^n. This is the cardinality of the set of all elements of F^n that are at distance (see 2.6.2) less than or equal to t from an element of F^n.

In the context of codes, the function is used to determine if a code is perfect. A code is perfect if spheres of radius t around all codewords contain exactly the whole vectorspace, where t is the number of errors the code can correct.

```
gap> SphereContent( 15, 0, GF(2) );
1    # Only one word with distance 0, which is the word itself
gap> SphereContent( 11, 3, GF(4) );
4984
gap> C := HammingCode(5);
a linear [31,26,3]1 Hamming (5,2) code over GF(2)
#the minimum distance is 3, so the code can correct one error
gap> ( SphereContent( 31, 1, GF(2) ) * Size(C) ) = 2 ^ 31;
true
```

2 ► `Krawtchouk(` k, i, n, q `)`

`Krawtchouk` returns the Krawtchouk number $K_k(i)$. q must be a primepower, n must be a positive integer, k must be a non-negative integer less then or equal to n and i can be any integer. (See 6.2.1).

```
gap> Krawtchouk( 2, 0, 3, 2);
3
```

3 ► `PrimitiveUnityRoot(` F, n `)`

`PrimitiveUnityRoot` returns a **primitive nth root of unity** in an extension field of F. This is a finite field element a with the property $a^n = 1 \bmod n$, and n is the smallest integer such that this equality holds.

```
gap> PrimitiveUnityRoot( GF(2), 15 );
Z(2^4)
gap> last^15;
Z(2)^0
gap> PrimitiveUnityRoot( GF(8), 21 );
Z(2^6)^3
```

4 ► `ReciprocalPolynomial(` P `)`

`ReciprocalPolynomial` returns the **reciprocal** of polynomial P. This is a polynomial with coefficients of P in the reverse order. So if $P = a_0+a_1X+\cdots+a_nX^n$, the reciprocal polynomial is $P' = a_n+a_{n-1}X+\cdots+a_0X^n$.

```
gap> P := UnivariatePolynomial( GF(3), Z(3)^0 * [1,0,1,2] );
Z(3)^0+x_1^2-x_1^3
gap> RecP := ReciprocalPolynomial( P );
-Z(3)^0+x_1+x_1^3
gap> ReciprocalPolynomial( RecP ) = P;
true
```

5 ► `ReciprocalPolynomial(` P , n `)`

In this form, the number of coefficients of P is considered to be at least n (possibly with zero coefficients at the highest degrees). Therefore, the reciprocal polynomial Pálso has degree at least n.

```
gap> P := UnivariatePolynomial( GF(3), Z(3)^0 * [1,0,1,2] );
Z(3)^0+x_1^2-x_1^3
gap> ReciprocalPolynomial( P, 6 );
-x_1^3+x_1^4+x_1^6
```

In this form, the degree of P is considered to be at least n (if not, zero coefficients are added). Therefore, the reciprocal polynomial Pálso has degree at least n.

6 ► `CyclotomicCosets(` q, n `)`

`CyclotomicCosets` returns the cyclotomic cosets of q modulo n. q and n must be relatively prime. Each of the elements of the returned list is a list of integers that belong to one cyclotomic coset. Each coset contains all multiplications of the **coset representative** by q, modulo n. The coset representative is the smallest integer that isn't in the previous cosets.

```
gap> CyclotomicCosets( 2, 15 );
[ [ 0 ], [ 1, 2, 4, 8 ], [ 3, 6, 12, 9 ], [ 5, 10 ],
  [ 7, 14, 13, 11 ] ]
gap> CyclotomicCosets( 7, 6 );
[ [ 0 ], [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ] ]
```

7 ▶ `WeightHistogram(` C `)`
▶ `WeightHistogram(` C, h `)`

The function `WeightHistogram` plots a histogram of weights in code C. The maximum length of a column is h. Default value for h is 1/3 of the size of the screen. The number that appears at the top of the histogram is the maximum value of the list of weights.

```
gap> H := HammingCode(2, GF(5));
a linear [6,4,3]1 Hamming (2,5) code over GF(5)
gap> WeightDistribution(H);
[ 1, 0, 0, 80, 120, 264, 160 ]
gap> WeightHistogram(H);
264----------------
               *
               *
               *
               *
               *  *
            *  *  *
         *  *  *  *
         *  *  *  *
+--------+--+--+--+--
0  1  2  3  4  5  6
```

7 Extensions to GUAVA

In this chapter some extensions added in Version 1.3 to GUAVA will be discussed. The most important extensions are new code constructions and new algorithms and bounds for the covering radius. Another important function is the implementation of the algorithm of Leon for finding the minimum distance.

7.1 Some functions for the covering radius

Together with the new code constructions, the new functions for computing (the bounds of) the covering radius are the most important additions to GUAVA. These additions required a change in the fields of a code record. In previous versions of GUAVA, the covering radius field was an integer field, called `coveringRadius`. To allow the code-record to contain more information about the covering radius, this field has been replaced by a field called `boundsCoveringRadius`. This field contains a vector of possible values of the covering radius of the code. If the value of the covering radius is known, then the length of this vector is one.

This means that every instance of `coveringRadius` in the version 1.2 were changed to `boundsCoveringRadius`. There is also an advantage to this: if bounds for a specific type of code are known, they can be implemented (and they have been). This has been especially useful for the Reed-Muller codes.

Of course, the main covering radius function dispatcher, `CoveringRadius`, had to be changed to incorporate these changes. Previously, this dispatcher called `code.operations.CoveringRadius`. The problem these functions had was that they only worked if the redundancy was not too large. Another problem was that the algorithm for linear and cyclic codes was written in `C` (in the kernel of GAP). This did not allow the user to interrupt the function, except by pressing `ctrl-C` twice, which exits GAP altogether. For more information, check the section on the (new) `CoveringRadius` (7.1.1) function.

Perhaps the most interesting new covering radius function is `IncreaseCoveringRadiusLowerBound` (7.1.4). It uses a probabilistic algorithm that tries to find better lower bounds of the covering radius of a code. It works best when the dimension is low, thereby giving a sort of complement function to `CoveringRadius`. When the dimension is about half the length of a code, neither algorithm will work, although `IncreaseCoveringRadiusLowerBound` is specifically designed to avoid memory problems, unlike `CoveringRadius`.

The function `ExhaustiveSearchCoveringRadius` (7.1.5) tries to find the covering radius of a code by exhaustively searching the space in which the code lies for coset leaders.

A number of bounds for the covering radius in general have been implemented, including some well known bounds like the sphere-covering bound, the redundancy bound and the Delsarte bound. These function all start with `LowerBoundCoveringRadius` (see 7.1.8, 7.1.9, 7.1.10, 7.1.11, 7.1.12, 7.1.13, 7.1.14, 7.1.8) or `UpperBoundCoveringRadius` (sections 7.1.15, 7.1.16, 7.1.17, 7.1.18, 7.1.19).

The functions `GeneralLowerBoundCoveringRadius` (7.1.6) and `GeneralUpperBoundCoveringRadius` (7.1.7) try to find the best known bounds for a given code. `BoundsCoveringRadius` (7.1.2) uses these functions to return a vector of possible values for the covering radius.

To allow the user to enter values in the `.boundsCoveringRadius` record herself, the function `SetCoveringRadius` is provided.

1 ► `CoveringRadius(` *code* `)`

`CoveringRadius` is a function that already appeared in earlier versions of GUAVA, but it is changed to reflect the implementation of new functions for the covering radius.

If there exists a function called `SpecialCoveringRadius` in the `operations` field of the code, then this function will be called to compute the covering radius of the code. At the moment, no code-specific functions are implemented.

If the length of `BoundsCoveringRadius` (see 7.1.2), is 1, then the value in

```
code.boundsCoveringRadius
```

is returned. Otherwise, the function

```
code.operations.CoveringRadius
```

is executed, unless the redundancy of *code* is too large. In the last case, a warning is issued.

If you want to overrule this restriction, you might want to execute

```
code.operations.CoveringRadius
```

yourself. However, this algorithm might also issue a warning that it cannot be executed, but this warning is sometimes issued too late, resulting in GAP exiting with an memory error. If it does run, it can only be stopped by pressing `ctrl-C` twice, thereby quitting GAP. It will not enter the usual break-loop. Therefore it is recommended to save your work before trying `code.operations.CoveringRadius`.

```
gap> CoveringRadius( BCHCode( 17, 3, GF(2) ) );
3
gap> CoveringRadius( HammingCode( 5, GF(2) ) );
1
gap> code := ReedMullerCode( 1, 9 );;
gap> CoveringRadius( code );
CoveringRadius: warning, the covering radius of
this code cannot be computed straightforward.
Try to use IncreaseCoveringRadiusLowerBound( <code> ).
(see the manual for more details).
The covering radius of <code> lies in the interval:
[ 240 .. 248 ]
```

2 ► `BoundsCoveringRadius(` *code* `)`

`BoundsCoveringRadius` returns a list of integers. The first entry of this list is the maximum of some lower bounds for the covering radius of *code*, the last entry the minimum of some upper bounds of *code*.

If the covering radius of *code* is known, a list of length 1 is returned.

`BoundsCoveringRadius` makes use of the functions `GeneralLowerBoundCoveringRadius` and `GeneralUpperBoundCoveringRadius`.

```
gap> BoundsCoveringRadius( BCHCode( 17, 3, GF(2) ) );
[ 3 .. 4 ]
gap> BoundsCoveringRadius( HammingCode( 5, GF(2) ) );
[ 1 ]
```

3 ► `SetCoveringRadius(` *code*`,` *intlist* `)`

`SetCoveringRadius` enables the user to set the covering radius herself, instead of letting GUAVA compute it. If *intlist* is an integer, GUAVA will simply put it in the `boundsCoveringRadius` field. If it is a list of integers, however, it will intersect this list with the `boundsCoveringRadius` field, thus taking the best of both lists. If this would leave an empty list, the field is set to *intlist*.

Because some other computations use the covering radius of the code, it is important that the entered value is not wrong, otherwise new results may be invalid.

```
gap> code := BCHCode( 17, 3, GF(2) );;
gap> BoundsCoveringRadius( code );
[ 3 .. 4 ]
gap> SetCoveringRadius( code, [ 2 .. 3 ] );
gap> BoundsCoveringRadius( code );
[ [ 2 .. 3 ] ]
```

4 ► `IncreaseCoveringRadiusLowerBound(` *code* `[,` *stopdistance* `] [,` *startword* `] )`

`IncreaseCoveringRadiusLowerBound` tries to increase the lower bound of the covering radius of *code*. It does this by means of a probabilistic algorithm. This algorithm takes a random word in $GF(q)^n$ (or *startword* if it is specified), and, by changing random coordinates, tries to get as far from *code* as possible. If changing a coordinate finds a word that has a larger distance to the code than the previous one, the change is made permanent, and the algorithm starts all over again. If changing a coordinate does not find a coset leader that is further away from the code, then the change is made permanent with a chance of 1 in 100, if it gets the word closer to the code, or with a chance of 1 in 10, if the word stays at the same distance. Otherwise, the algorithm starts again with the same word as before.

If the algorithm did not allow changes that decrease the distance to the code, it might get stuck in a sub-optimal situation (the coset leader corresponding to such a situation (i.e. no coordinate of this coset leader can be changed in such a way that we get at a larger distance from the code) is called an orphan).

If the algorithm finds a word that has distance *stopdistance* to the code, it ends and returns that word, which can be used for further investigations.

The variable `InfoCoveringRadius` can be set to `Print` to print the maximum distance reached so far every 1000 runs. The alogrithm can be interrupted with `ctrl-C`, allowing the user to look at the word that is currently being examined (called `current`), or to change the chances that the new word is made permanent (these are called `staychance` and `downchance`). If one of these variables is i, then it corresponds with a i in 100 chance.

At the moment, the algorithm is only useful for codes with small dimension, where small means that the elements of the code fit in the memory. It works with larger codes, however, but when you use it for codes with large dimension, you should be **very** patient. If running the algorithm quits GAP (due to memory problems), you can change the global variable `CRMemSize` to a lower value. This might cause the algorithm to run slower, but without quitting GAP. The only way to find out the best value of `CRMemSize` is by experimenting.

5 ► `ExhaustiveSearchCoveringRadius(` *code* `)`

`ExhaustiveSearchCoveringRadius` does an exhaustive search to find the covering radius of *code*. Every time a coset leader of a coset with weight w is found, the function tries to find a coset leader of a coset with weight $w+1$. It does this by enumerating all words of weight $w+1$, and checking whether a word is a coset leader. The start weight is the current known lower bound on the covering radius.

6 ► `GeneralLowerBoundCoveringRadius(` *code* `)`

`GeneralLowerBoundCoveringRadius` returns a lower bound on the covering radius of *code*. It uses as many functions which names start with `LowerBoundCoveringRadius` as possible to find the best known lower bound (at least that GUAVA knows of) together with tables for the covering radius of binary linear codes with length not greater than 64.

7 ▶ `GeneralUpperBoundCoveringRadius(` *code* `)`

`GeneralUpperBoundCoveringRadius` returns an upper bound on the covering radius of *code*. It uses as many functions which names start with `UpperBoundCoveringRadius` as possible to find the best known upper bound (at least that GUAVA knows of).

8 ▶ `LowerBoundCoveringRadiusSphereCovering(` n, M `[,` F `], false )`
▶ `LowerBoundCoveringRadiusSphereCovering(` n, r `[,` F `] [, true ] )`

If the last argument of `LowerBoundCoveringRadiusSphereCovering` is `false`, then it returns a lower bound for the covering radius of a code of size M and length n. Otherwise, it returns a lower bound for the size of a code of length n and covering radius r.

F is the field over which the code is defined. If F is omitted, it is assumed that the code is over `GF(2)`.

The bound is computed according to the sphere covering bound:

$$MV_q(n,r) \geq q^n$$

where $V_q(n,r)$ is the size of a sphere of radius r in $\mathtt{GF}(q)^n$.

9 ▶ `LowerBoundCoveringRadiusVanWee1(` n, M `[,` F `], false )`
▶ `LowerBoundCoveringRadiusVanWee1(` n, r `[,` F `] [, true ] )`

If the last argument of `LowerBoundCoveringRadiusVanWee1` is `false`, then it returns a lower bound for the covering radius of a code of size M and length n. Otherwise, it returns a lower bound for the size of a code of length n and covering radius r.

F is the field over which the code is defined. If F is omitted, it is assumed that the code is over `GF(2)`.

The Van Wee bound is an improvement of the sphere covering bound:

$$M\left\{V_q(n,r) - \frac{\binom{n}{r}}{\lceil\frac{n-r}{r+1}\rceil}\left(\left\lceil\frac{n+1}{r+1}\right\rceil - \frac{n+1}{r+1}\right)\right\} \geq q^n$$

10 ▶ `LowerBoundCoveringRadiusVanWee2(` n, M`, false )`
▶ `LowerBoundCoveringRadiusVanWee2(` n, r `[, true ] )`

If the last argument of `LowerBoundCoveringRadiusVanWee2` is `false`, then it returns a lower bound for the covering radius of a code of size M and length n. Otherwise, it returns a lower bound for the size of a code of length n and covering radius r.

This bound only works for binary codes. It is based on the following inequality:

$$M\frac{\left(\left(V_2(n,2) - \frac{1}{2}(r+2)(r-1)\right) V_2(n,r) + \varepsilon V_2(n,r-2)\right)}{\left(V_2(n,2) - \frac{1}{2}(r+2)(r-1) + \varepsilon\right)} \geq 2^n,$$

where

$$\varepsilon = \binom{r+2}{2}\left\lceil\binom{n-r+1}{2}\Big/\binom{r+2}{2}\right\rceil - \binom{n-r+1}{2}.$$

11 ▶ `LowerBoundCoveringRadiusCountingExcess(` n, M`, false )`
▶ `LowerBoundCoveringRadiusCountingExcess(` n, r `[, true ] )`

If the last argument of `LowerBoundCoveringRadiusCountingExcess` is `false`, then it returns a lower bound for the covering radius of a code of size M and length n. Otherwise, it returns a lower bound for the size of a code of length n and covering radius r.

This bound only works for binary codes. It is based on the following inequality:

$$M\left(\rho V_2(n,r) + \varepsilon V_2(n,r-1)\right) \geq (\rho+\varepsilon)2^n,$$

where

$$\varepsilon = (r+1)\left\lceil \frac{n+1}{r+1} \right\rceil - (n+1)$$

and

$$\rho = \begin{cases} n-3+\frac{2}{n} & \text{if} \quad r=2 \\ n-r-1 & \text{if} \quad r \geq 3 \end{cases}$$

12 ▶ `LowerBoundCoveringRadiusEmbedded1(` n`,` M`, false )`
▶ `LowerBoundCoveringRadiusEmbedded1(` n`,` r `[, true ] )`

If the last argument of `LowerBoundCoveringRadiusEmbedded1` is `false`, then it returns a lower bound for the covering radius of a code of size M and length n. Otherwise, it returns a lower bound for the size of a code of length n and covering radius r.

This bound only works for binary codes. It is based on the following inequality:

$$M\left(V_2(n,r) - \binom{2r}{r}\right) \geq 2^n - A(n,2r+1)\binom{2r}{r},$$

where $A(n,d)$ denotes the maximal cardinality of a (binary) code of length n and minimum distance d. The function `UpperBound` is used to compute this value.

Sometimes `LowerBoundCoveringRadiusEmbedded1` is better than `LowerBoundCoveringRadiusEmbedded2`, sometimes it is the other way around.

13 ▶ `LowerBoundCoveringRadiusEmbedded2(` n`,` M`, false )`
▶ `LowerBoundCoveringRadiusEmbedded2(` n`,` r `[, true ] )`

If the last argument of `LowerBoundCoveringRadiusEmbedded2` is `false`, then it returns a lower bound for the covering radius of a code of size M and length n. Otherwise, it returns a lower bound for the size of a code of length n and covering radius r.

This bound only works for binary codes. It is based on the following inequality:

$$M\left(V_2(n,r) - \frac{3}{2}\binom{2r}{r}\right) \geq 2^n - 2A(n,2r+1)\binom{2r}{r},$$

where $A(n,d)$ denotes the maximal cardinality of a (binary) code of length n and minimum distance d. The function `UpperBound` is used to compute this value.

Sometimes `LowerBoundCoveringRadiusEmbedded1` is better than `LowerBoundCoveringRadiusEmbedded2`, sometimes it is the other way around.

14 ▶ `LowerBoundCoveringRadiusInduction(` n`,` r `)`

`LowerBoundCoveringRadiusInduction` returns a lower bound for the size of a code with length n and covering radius r.

If $n = 2r+2$ and $r \geq 1$, the returned value is 4.

If $n = 2r+3$ and $r \geq 1$, the returned value is 7.

If $n = 2r + 4$ and $r \geq 4$, the returned value is 8.

Otherwise, 0 is returned.

15 ▶ `UpperBoundCoveringRadiusRedundancy(` *code* `)`

`UpperBoundCoveringRadiusRedundancy` returns the redundancy of *code* as an upper bound for the covering radius of *code*. *code* must be a linear code.

16 ▶ `UpperBoundCoveringRadiusDelsarte(` *code* `)`

`UpperBoundCoveringRadiusDelsarte` returns an upper bound for the covering radius of *code*. This upper-bound is equal to the **external distance** of *code*, this is the minimum distance of the dual code, if *code* is a linear code.

17 ▶ `UpperBoundCoveringRadiusStrength(` *code* `)`

`UpperBoundCoveringRadiusStrength` returns an upper bound for the covering radius of *code*.

First the code is punctured at the zero coordinates (i.e. the coordinates where all codewords have a zero). If the remaining code has strength 1 (i.e. each coordinate contains each element of the field an equal number of times), then it returns $\frac{q-1}{q}m + (n-m)$ (where q is the size of the field and m is the length of punctured code), otherwise it returns n. This bound works for all codes.

18 ▶ `UpperBoundCoveringRadiusGriesmerLike(` *code* `)`

This function returns an upper bound for the covering radius of *code*, which must be linear, in a Griesmer-like fashion. It returns

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

19 ▶ `UpperBoundCoveringRadiusCyclicCode(` *code* `)`

This function returns an upper bound for the covering radius of *code*, which must be a cyclic code. It returns

$$n - k + 1 - \left\lceil \frac{w(g(x))}{2} \right\rceil,$$

where $g(x)$ is the generator polynomial of *code*.

7.2 New code constructions

In this section we describe some new constructions for codes. The first constructions are variations on the direct sum construction, most of the time resulting in better codes than the direct sum.

The piecewise constant code construction stands on its own. Using this construction, some good codes have been obtained.

The last five constructions yield linear binary codes with fixed minimum distances and covering radii. These codes can be arbitrary long.

1 ▶ `ExtendedDirectSumCode(` *L*`,` *B*`,` *m* `)`

The extended direct sum construction is described in an article by Graham and Sloane. The resulting code consists of m copies of L, extended by repeating the codewords of B m times.

Suppose L is an $[n_L, k_L]r_L$ code, and B is an $[n_L, k_B]r_B$ code (non-linear codes are also permitted). The length of B must be equal to the length of L. The length of the new code is $n = mn_L$, the dimension (in the case of linear codes) is $k \leq mk_L + k_B$, and the covering radius is $r \leq \lfloor m\Psi(L,B) \rfloor$, with

$$\Psi(L,B) = \max_{u \in F_2^{n_L}} \frac{1}{2^{k_B}} \sum_{v \in B} \mathrm{d}(L, v+u)\cdot$$

However, this computation will not be executed, because it may be too time consuming for large codes.

If $L \subseteq B$, and L and B are linear codes, the last copy of L is omitted. In this case the dimension is $k = mk_L + (k_B - k_L)$.

```
gap> c := HammingCode( 3, GF(2) );
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> d := WholeSpaceCode( 7, GF(2) );
a cyclic [7,7,1]0 whole space code over GF(2)
gap> e := ExtendedDirectSumCode( c, d, 3 );
a linear [21,15,1..3]2 3-fold extended direct sum code
```

2 ► `AmalgatedDirectSumCode(` *c_1*, *c_2* `[,` *check* `] )`

`AmalgatedDirectSumCode` returns the amalgated direct sum of the codes *c_1* and *c_2*. The amalgated direct sum code consists of all codewords of the form $(u\,|\,0\,|\,v)$ if $(u\,|\,0) \in c_1$ and $(0\,|\,v) \in c_2$ and all codewords of the form $(u\,|\,1\,|\,v)$ if $(u\,|\,1) \in c_1$ and $(1\,|\,v) \in c_2$. The result is a code with length $n = n_1 + n_2 - 1$ and size $M <= M_1 \cdot M_2/2$.

If both codes are linear, they will first be standardized, with information symbols in the last and first coordinates of the first and second code, respectively.

If c_1 is a normal code with the last coordinate acceptable, and c_2 is a normal code with the first coordinate acceptable, then the covering radius of the new code is $r <= r_1 + r_2$. However, checking whether a code is normal or not is a lot of work, and almost all codes seem to be normal. Therefore, an option *check* can be supplied. If *check* is true, then the codes will be checked for normality. If *check* is false or omitted, then the codes will not be checked. In this case it is assumed that they are normal. Acceptability of the last and first coordinate of the first and second code, respectively, is in the last case also assumed to be done by the user.

```
gap> c := HammingCode( 3, GF(2) );
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> d := ReedMullerCode( 1, 4 );
a linear [16,5,8]6 Reed-Muller (1,4) code over GF(2)
gap> e := DirectSumCode( c, d );
a linear [23,9,3]7 direct sum code
gap> f := AmalgatedDirectSumCode( c, d );;
gap> MinimumDistance( f );;
gap> CoveringRadius( f );; # takes some time
gap> f;
a linear [22,8,3]7 amalgated direct sum code
```

3 ► `BlockwiseDirectSumCode(` *c_1*, *l_1*, *c_2*, *l_2* `)`

`BlockwiseDirectSumCode` returns a subcode of the direct sum of *c_1* and *c_2*. The fields of *c_1* and *c_2* should be same. *l_1* and *l_2* are two equally long lists with elements from the spaces where *c_1* and *c_2* are in, respectively, *or* *l_1* and *l_2* are two equally long lists containing codes. The union of the codes in *l_1* and *l_2* must be *c_1* and *c_2*, respectively.

In the first case, the blockwise direct sum code is defined as

$$bds = \bigcup_{1 \le i \le l} (c_1 + (l_1)_i) \oplus (c_2 + (l_2)_i),$$

where l is the length of l_1 and l_2, and $\oplus$ is the direct sum.

In the second case, it is defined as

$$bds = \bigcup_{1 \le i \le l} ((l_1)_i \oplus (l_2)_i).$$

The length of the new code is $n = n_1 + n_2$.

```
gap> c := HammingCode( 3, GF(2) );;
gap> d := EvenWeightSubcode( WholeSpaceCode( 6, GF(2) ) );;
gap> BlockwiseDirectSumCode( c, [[ 0,0,0,0,0,0,0 ],[ 1,0,1,0,1,0,0 ]],
> d, [[ 0,0,0,0,0,0 ],[ 1,0,1,0,1,0 ]] );
a (13,1024,1..13)1..2 blockwise direct sum code
```

4 ► `PiecewiseConstantCode(` *part*`,` *weights* `[,` *field* `] )`

`PiecewiseConstantCode` returns a code with length $n = \sum n_i$, where *part* $= [n_1, \ldots, n_k]$. *weights* is a list of constraints, each of length k. The default field is `GF(2)`.

A constraint is a list of integers, and a word $c = (c_1, \ldots, c_k)$ (according to *part*) is in the resulting code if and only if $|c_i| = w_i^{(l)}$ for all $1 \le i \le k$ for some constraint $w^{(l)} \in$ *constraints*.

An example might make things clearer:

```
gap> PiecewiseConstantCode( [ 2, 3 ],
> [ [ 0, 0 ], [ 0, 3 ], [ 1, 0 ], [ 2, 2 ] ],
> GF(2) );
a (5,7,1..5)1..5 piecewise constant code over GF(2)
gap> AsSSortedList(last);
[ [ 0 0 0 0 0 ], [ 0 0 1 1 1 ], [ 0 1 0 0 0 ], [ 1 0 0 0 0 ], [ 1 1 0 1 1 ],
  [ 1 1 1 0 1 ], [ 1 1 1 1 0 ] ]
```

The first constraint is satisfied by codeword 1, the second by codeword 2, the third by codewords 3 and 4, and the fourth by codewords 5, 6 and 7.

7.3 Gabidulin codes

These five codes are derived from an article by Gabidulin, Davydov and Tombak [GDT91]. These five codes are defined by check matrices. Exact definitions can be found in the article.

The Gabidulin code, the enlarged Gabidulin code, the Davydov code, the Tombak code, and the enlarged Tombak code, correspond with theorem 1, 2, 3, 4, and 5, respectively in the article.

These codes have fixed minimum distance and covering radius, but can be arbitrarily long. They are defined through check matrices.

1 ► `GabidulinCode(` *m*`,` *w1*`,` *w2* `)`

`GabidulinCode` yields a code of length $5 \cdot 2^{m-2} - 1$, redundancy $2m - 1$, minimum distance 3 and covering radius 2. *w1* and *w2* should be elements of `GF(`2^{m-2}`)`.

2 ► `EnlargedGabidulinCode(` *m*`,` *w1*`,` *w2*`,` *e* `)`

`EnlargedGabidulinCode` yields a code of length $7 \cdot 2^{m-2} - 2$, redundancy $2m$, minimum distance 3 and covering radius 2. *w1* and *w2* are elements of `GF(`2^{m-2}`)`. e is an element of `GF(`2^m`)`. The core of an enlarged Gabidulin code consists of a Gabidulin code.

3 ▶ `DavydovCode(` *r* , *v* , *ei* , *ej* `)`

`DavydovCode` yields a code of length $2^v + 2^{r-v} - 3$, redundancy r, minimum distance 4 and covering radius 2. v is an integer between 2 and $r - 2$. *ei* and *ej* are elements of `GF(`2^v`)` and `GF(`2^{r-v}`)`, respectively.

4 ▶ `TombakCode(` *m* , *e* , *beta* , *gamma* , *w1* , *w2* `)`

`TombakCode` yields a code of length $15 \cdot 2^{m-3} - 3$, redundancy $2m$, minimum distance 4 and covering radius 2. *e* is an element of `GF(`2^m`)`. *beta* and *gamma* are elements of `GF(`2^{m-1}`)`. *w1* and *w2* are elements of `GF(`2^{m-3}`)`.

5 ▶ `EnlargedTombakCode(` *m* , *e* , *beta* , *gamma* , *w1* , *w2* , *u* `)`

`EnlargedTombakCode` yields a code of length $23 \cdot 2^{m-4} - 3$, redundancy $2m - 1$, minimum distance 4 and covering radius 2. The parameters *m*, *e*, *beta*, *gamma*, *w1* and *w2* are defined as in `TombakCode`. *u* is an element of `GF(`2^{m-1}`)`. The code of an enlarged Tombak code consists of a Tombak code.

```
gap> GabidulinCode( 4, Z(4)^0, Z(4)^1 );
a linear [19,12,3]2 Gabidulin code (m=4) over GF(2)
gap> EnlargedGabidulinCode( 4, Z(4)^0, Z(4)^1, Z(16)^11 );
a linear [26,18,3]2 enlarged Gabidulin code (m=4) over GF(2)
gap> DavydovCode( 6, 3, Z(8)^1, Z(8)^5 );
a linear [13,7,4]2 Davydov code (r=6, v=3) over GF(2)
gap> TombakCode( 5, Z(32)^6, Z(16)^14, Z(16)^10, Z(4)^0, Z(4)^1 );
a linear [57,47,4]2 Tombak code (m=5) over GF(2)
gap> EnlargedTombakCode( 6, Z(32)^6, Z(16)^14, Z(16)^10,
> Z(4)^0, Z(4)^0, Z(32)^23 );
a linear [89,78,4]2 enlarged Tombak code (m=6) over GF(2)
```

7.4 Some functions related to the norm of a code

In this section, some functions that can be used to compute the norm of a code and to decide upon its normality are discussed.

1 ▶ `CoordinateNorm(` *code* , *coord* `)`

`CoordinateNorm` returns the norm of *code* with respect to coordinate *coord*. If $C_a = \{c \in code | c_{coord} = a\}$, then the norm of *code* with respect to *coord* is defined as

$$\max_{v \in GF(q)^n} \sum_{a=1}^{q} d(x, C_a),$$

with the convention that $d(x, C_a) = n$ if C_a is empty.

```
gap> CoordinateNorm( HammingCode( 3, GF(2) ), 3 );
3
```

2 ▶ `CodeNorm(` *code* `)`

`CodeNorm` returns the norm of *code*. The norm of a code is defined as the minimum of the norms for the respective coordinates of the code. In effect, for each coordinate `CoordinateNorm` is called, and the minimum of the calculated numbers is returned.

```
gap> CodeNorm( HammingCode( 3, GF(2) ) );
3
```

3 ▶ `IsCoordinateAcceptable(` *code*`,` *coord* `)`

`IsCoordinateAcceptable` returns `true` if coordinate *coord* of *code* is acceptable. A coordinate is called acceptable if the norm of the code with respect to that coordinate is not more than two times the covering radius of the code plus one.

```
gap> IsCoordinateAcceptable( HammingCode( 3, GF(2) ), 3 );
true
```

4 ▶ `GeneralizedCodeNorm(` *code*`,` *subcode1*`,` *subcode2*`, ...,` *subcodek* `)`

`GeneralizedCodeNorm` returns the k-norm of *code* with respect to k subcodes.

```
gap> c := RepetitionCode( 7, GF(2) );;
gap> ham := HammingCode( 3, GF(2) );;
gap> d := EvenWeightSubcode( ham );;
gap> e := ConstantWeightSubcode( ham, 3 );;
gap> GeneralizedCodeNorm( ham, c, d, e );
4
```

5 ▶ `IsNormalCode(` *code* `)`

`IsNormalCode` returns `true` if *code* is normal. A code is called normal if the norm of the code is not more than two times the covering radius of the code plus one. Almost all codes are normal, however some (non-linear) abnormal codes have been found.

Often, it is difficult to find out whether a code is normal, because it involves computing the covering radius. However, `IsNormalCode` uses much information from the literature about normality for certain code parameters.

```
gap> IsNormalCode( HammingCode( 3, GF(2) ) );
true
```

6 ▶ `DecreaseMinimumDistanceLowerBound(` *code*`,` *s*`,` *iterations* `)`

`DecreaseMinimumDistanceLowerBound` is an implementation of the algorithm for the minimum distance by Leon [Leo91]. This algorithm tries to find codewords with small minimum weights. The parameter s is described in the article, the best results are obtained if it is close to the dimension of the code. The parameter *iterations* gives the number of runs that the algorithm will perform.

The result returned is a record with two fields; the first, `mindist`, gives the lowest weight found, and `word` gives the corresponding codeword.

7.5 New miscellaneous functions

In this section, some new miscellaneous functions are described, including weight enumerators, the MacWilliams-transform and affinity and almost affinity of codes.

1 ▶ `CodeWeightEnumerator(` *code* `)`

`CodeWeightEnumerator` returns a polynomial of the following form:

$$f(x) = \sum_{i=0}^{n} A_i x^i,$$

where A_i is the number of codewords in *code* with weight i.

```
gap> CodeWeightEnumerator( ElementsCode( [ [ 0,0,0 ], [ 0,0,1 ],
> [ 0,1,1 ], [ 1,1,1 ] ], GF(2) ) );
x^3 + x^2 + x + 1
gap> CodeWeightEnumerator( HammingCode( 3, GF(2) ) );
x^7 + 7*x^4 + 7*x^3 + 1
```

2 ► `CodeDistanceEnumerator(` *code*, *word* `)`

`CodeDistanceEnumerator` returns a polynomial of the following form:

$$f(x) = \sum_{i=0}^{n} B_i x^i,$$

where B_i is the number of codewords with distance i to *word*.

If *word* is a codeword, then `CodeDistanceEnumerator` returns the same polynomial as `CodeWeightEnumerator`.

```
gap> CodeDistanceEnumerator( HammingCode( 3, GF(2) ),[0,0,0,0,0,0,1] );
x^6 + 3*x^5 + 4*x^4 + 4*x^3 + 3*x^2 + x
gap> CodeDistanceEnumerator( HammingCode( 3, GF(2) ),[1,1,1,1,1,1,1] );
x^7 + 7*x^4 + 7*x^3 + 1 # ‘[1,1,1,1,1,1,1]’ $\in$ ‘HammingCode( 3, GF(2 ) )’
```

3 ► `CodeMacWilliamsTransform(` *code* `)`

`CodeMacWilliamsTransform` returns a polynomial of the following form:

$$f(x) = \sum_{i=0}^{n} C_i x^i,$$

where C_i is the number of codewords with weight i in the dual code of *code*.

```
gap> CodeMacWilliamsTransform( HammingCode( 3, GF(2) ) );
7*x^4 + 1
```

4 ► `IsSelfComplementaryCode(` *code* `)`

`IsSelfComplementaryCode` returns `true` if

$$v \in code \Rightarrow 1 - v \in code,$$

where 1 is the all-one word of length n.

```
gap> IsSelfComplementaryCode( HammingCode( 3, GF(2) ) );
true
gap> IsSelfComplementaryCode( EvenWeightSubcode(
> HammingCode( 3, GF(2) ) ) );
false
```

5 ► `IsAffineCode(` *code* `)`

`IsAffineCode` returns `true` if *code* is an affine code. A code is called *affine* if it is an affine space. In other words, a code is affine if it is a coset of a linear code.

```
gap> IsAffineCode( HammingCode( 3, GF(2) ) );
true
gap> IsAffineCode( CosetCode( HammingCode( 3, GF(2) ),
> [ 1, 0, 0, 0, 0, 0, 0 ] ) );
true
gap> IsAffineCode( NordstromRobinsonCode() );
false
```

6► `IsAlmostAffineCode(` *code* `)`

`IsAlmostAffineCode` returns `true` if *code* is an almost affine code. A code is called *almost affine* if the size of any punctured code of *code* is q^r for some r, where q is the size of the alphabet of the code. Every affine code is also almost affine, and every code over `GF(2)` and `GF(3)` that is almost affine is also affine.

```
gap> code := ElementsCode( [ [0,0,0], [0,1,1], [0,2,2], [0,3,3],
>                            [1,0,1], [1,1,0], [1,2,3], [1,3,2],
>                            [2,0,2], [2,1,3], [2,2,0], [2,3,1],
>                            [3,0,3], [3,1,2], [3,2,1], [3,3,0] ],
>                            GF(4) );;
gap> IsAlmostAffineCode( code );
true
gap> IsAlmostAffineCode( NordstromRobinsonCode() );
false
```

7► `IsGriesmerCode(` *code* `)`

`IsGriesmerCode` returns `true` if *code*, which must be a linear code, is Griesmer code, and `false` otherwise.

A code is called Griesmer if its length satisfies

$$n = g[k,d] = \sum_{i=0}^{k-1} \lceil \frac{d}{q^i} \rceil.$$

```
gap> IsGriesmerCode( HammingCode( 3, GF(2) ) );
true
gap> IsGriesmerCode( BCHCode( 17, 2, GF(2) ) );
false
```

8► `CodeDensity(` *code* `)`

`CodeDensity` returns the *density* of *code*. The density of a code is defined as

$$\frac{M \cdot V_q(n,t)}{q^n},$$

where M is the size of the code, $V_q(n,t)$ is the size of a sphere of radius t in q^n, t is the covering radius of the code and n is the length of the code.

```
gap> CodeDensity( HammingCode( 3, GF(2) ) );
1
gap> CodeDensity( ReedMullerCode( 1, 4 ) );
14893/2048
```

Bibliography

[GDT91] Ernst M. Gabidulin, Alexander A. Davydov, and Leonid M. Tombak. Linear codes with covering radius 2 and other new covering codes. *IEEE Trans. Inform. Theory*, 37(1):219–224, 1991.

[Leo91] Jeffrey S. Leon. Permutation group algorithms based on partitions, I: theory and algorithms. *J. Symbolic Comput.*, 12:533–583, 1991.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "PermutationCharacter" comes before "permutation group".

V

W