GUAVA

A GAP4 Package for computing with error-correcting codes

Version 3.23

26 November 2026

Jasper Cramwinckel
Erik Roijackers
Reinald Baart
Eric Minkes
Lea Ruscio
Robert L Miller
Tom Boothby
Cen ("CJ") Tjhai
David Joyner
Joe Fields (Maintainer)

Joe Fields (Maintainer) Email: fieldsj1@southernct.edu
Homepage: http://giam.southernct.edu/
Address: Mathematics Department,
Southern Connecticut State University,
New Haven, CT,
06515 USA.

Copyright

GUAVA: © The GUAVA Group: 1992-2003 Jasper Cramwinckel, Erik Roijackers, Reinald Baart, Eric Minkes, Lea Ruscio (for the tex version), Jeffrey Leon © 2004 David Joyner, Cen Tjhai, Jasper Cramwinckel, Erik Roijackers, Reinald Baart, Eric Minkes, Lea Ruscio. © 2007 Robert L Miller, Tom Boothby © 2009, 2012, 2016, 2018, 2022, 2025 Joe Fields

GUAVA is released under the GNU General Public License (GPL).

GUAVA is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

GUAVA is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with GUAVA; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details, see https://www.fsf.org/licenses/gpl.html.

For many years GUAVA has been released along with the "backtracking" C programs of J. Leon. In one of his *.c files the following statements occur: "Copyright (C) 1992 by Jeffrey S. Leon. This software may be used freely for educational and research purposes. Any other use requires permission from the author." The following should now be appended: "I, Jeffrey S. Leon, agree to license all the partition backtrack code which I have written under the GPL (www.fsf.org) as of this date, April 17, 2007."

GUAVA documentation: © Jasper Cramwinckel, Erik Roijackers, Reinald Baart, Eric Minkes, Lea Ruscio (for the tex version), Joe Fields, David Joyner, Cen Tjhai. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Acknowledgements

GUAVA was originally written by Jasper Cramwinckel, Erik Roijackers, and Reinald Baart in the early-to-mid 1990's as a final project during their study of Mathematics at the Delft University of Technology, Department of Pure Mathematics, under the direction of Professor Juriaan Simonis. This work was continued in Aachen, at Lehrstuhl D für 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 (versions 1.4 and 1.5) was created by Lea Ruscio and (from 2001, starting with version 1.6, to early 2009) was maintained by David Joyner, who (with the help of several students) added several new functions. Starting with version 2.7, the "best linear code" tables for binary codes have been updated. From 2009, starting with version 3.10, GUAVA has been maintained by Joe Fields. For further details, see the CHANGES.md file in the GUAVA directory, also available at https://gap-packages.github.io/guava/CHANGES.guava.

This documentation was prepared with the GAPDoc package of Frank Lübeck and Max Neunhöffer. The conversion from TeX to GAPDoc's XML was done by David Joyner in 2004.

Please send bug reports, suggestions and other comments about GUAVA to support@gap-system.org. Currently known bugs and suggested GUAVA projects are listed in the Issues section of the GUAVA GitHub page: https://github.com/gap-packages/guava/issues. Older releases and further history can be found on the GUAVA web page https://gap-packages.github.io/guava/.

Contributors: Other than the authors listed on the title page, the following people have contributed code to the GUAVA project: Alexander Hulpke, Steve Linton, Frank Lübeck, Aron Foster, Wayne Irons, Clifton (Clipper) Lennon, Jason McGowan, Shuhong Gao, Greg Gamble and Jeffrey S. Leon.

For documentation on Leon's programs, see the `src/leon/doc` subdirectory of GUAVA.

Contents

1 Introduction **6**
- 1.1 Introduction to the GUAVA package . . . 6
- 1.2 Installing GUAVA . . . 6
- 1.3 Loading GUAVA . . . 7

2 A First Tutorial in GUAVA **8**
- 2.1 Working with codewords . . . 8
- 2.2 Calculations with codes . . . 9

3 Codewords **11**
- 3.1 Construction of Codewords . . . 12
- 3.2 Comparisons of Codewords . . . 14
- 3.3 Arithmetic Operations for Codewords . . . 15
- 3.4 Functions that Convert Codewords to Vectors or Polynomials . . . 16
- 3.5 Functions that Change the Display Form of a Codeword . . . 17
- 3.6 Other Codeword Functions . . . 18

4 Codes **20**
- 4.1 Comparisons of Codes . . . 22
- 4.2 Operations for Codes . . . 23
- 4.3 Boolean Functions for Codes . . . 24
- 4.4 Equivalence and Isomorphism of Codes . . . 30
- 4.5 Domain Functions for Codes . . . 32
- 4.6 Printing and Displaying Codes . . . 34
- 4.7 Generating (Check) Matrices and Polynomials . . . 36
- 4.8 Parameters of Codes . . . 38
- 4.9 Distributions . . . 45
- 4.10 Decoding Functions . . . 47

5 Generating Codes **55**
- 5.1 Generating Unrestricted Codes . . . 55
- 5.2 Generating Linear Codes . . . 59
- 5.3 Gabidulin Codes . . . 67
- 5.4 Golay Codes . . . 68
- 5.5 Generating Cyclic Codes . . . 70
- 5.6 Evaluation Codes . . . 80

5.7 Algebraic geometric codes . . . 83
5.8 Low-Density Parity-Check Codes . . . 96

6 Manipulating Codes 98
6.1 Functions that Generate a New Code from a Given Code . . . 98
6.2 Functions that Generate a New Code from Two or More Given Codes . . . 108

7 Bounds on codes, special matrices and miscellaneous functions 116
7.1 Distance bounds on codes . . . 116
7.2 Covering radius bounds on codes . . . 122
7.3 Special matrices in GUAVA . . . 130
7.4 Some functions related to the norm of a code . . . 136
7.5 Miscellaneous functions . . . 138
7.6 Miscellaneous polynomial functions . . . 144

8 Coding theory functions in GAP 148
8.1 Distance functions . . . 148
8.2 Other functions . . . 151

9 GNU Free Documentation License 153

References 160

Index 161

Chapter 1

Introduction

1.1 Introduction to the GUAVA package

This is the manual of the GAP package GUAVA. GUAVA contains many functions that allow one to perform computations relevant to the theory of error-correcting codes. This version of GUAVA (3.23) requires GAP 4.11 or later.

The functions in GUAVA can be divided into three subcategories:

- Construction of codes: GUAVA can construct unrestricted, linear and cyclic codes. Information about the code, such as operations applicable to the code, is stored in a record-like data structure called a GAP object.
- Manipulations of codes: Manipulations transform one code into another, or construct 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 often decreases.
- Computations of information about codes: GUAVA can calculate important parameters of codes quickly. The results are stored in the codes' object components.

Except for the automorphism group and isomorphism testing functions, which make use of J.S. Leon's programs (see [Leo91] and the documentation in the 'src/leon' subdirectory of the 'guava' directory for some details), and `MinimumWeight` (4.8.5) function, GUAVA is written in the GAP language, and runs on any system supporting GAP 4.11 and above. Several algorithms that need the speed were integrated in the GAP kernel.

Good general references for error-correcting codes and the technical terms in this manual are MacWilliams and Sloane [MS83] and also Huffman and Pless [HP03].

1.2 Installing GUAVA

Recent versions of GAP come complete with all of the packages -- including GUAVA. Thus the following instructions are not usually applicable but may be needed in certain circumstances.

To install GUAVA unpack the archive file in a directory in the 'pkg' hierarchy of your version of GAP 4.

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-like environment, where you should proceed as follows: Go to the newly created 'guava' directory and call '`./configure /gappath`' where `/gappath` 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 binaries and install them in the appropriate place. For a Windows machine with CYGWIN installed - see https://www.cygwin.com/ - instructions for compiling Leon's binaries are likely to be similar to those above.

If it is not already installed, you should also install the GAP package SONATA. You can download this from the GAP website and unpack it in the 'pkg' subdirectory.

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 binaries for each platform separately.

1.3 Loading GUAVA

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

Example

```
gap> LoadPackage( "guava" );
true
```

If GUAVA isn't already in memory, it is loaded and the author information is displayed. If you are a frequent user of GUAVA, you might consider adding GUAVA to the "PackagesToLoad" preference in your gap initialization file. (Usually `gap.ini` which should be located in your home directory.) Type `GAPInfo.UserGapRoot;` at the GAP prompt to find the location of your initialization file. If none exists, the command `WriteGapIniFile();` will create a default `gap.ini` file which you can then modify.

Chapter 2

A First Tutorial in GUAVA

An error-correcting code is essentially just a subset of the set of all possible messages of a given length over some finite "alphabet."

In algebraic coding theory, the "alphabet" is usually some finite field (very often GF(2)) and frequently the error-correcting code is chosen to be a vector subspace of the space of all row vectors of some fixed length n. Such codes are known as *Linear Codes*, but, however a code is defined the point is to have a collection of "codewords" that are said to be "in the code" and any other word (row vectors that are *not* "in the code") will be assumed to be a codeword that has been mangled by the addition of noise.

When a message is received that is not a codeword, we ask ourselves the question "Which codeword is closest to this message I've received?" In other words we make the presumption that the received message is actually a codeword that has been changed in a relatively small number of positions -- and *we put them back the way they were supposed to be!*

That process is called "decoding." Developing codes that have efficient decoding algorithms is one of the central problems of algebraic coding theory.

2.1 Working with codewords

So let's play around a bit.

Start GAP in a terminal window, then issue the command

Example

```
gap> LoadPackage("guava");
true
```

GUAVA can construct codewords in a variety of ways. One of the most typical cases is for a codeword to consist of binary digits. In that case we say that "the code is over GF(2)" and codewords can be constructed as follows:

Example

```
gap> c1:=Codeword("101010101");
[ 1 0 1 0 1 0 1 0 1 ]
gap> v:=Z(2)*[1,1,1,1,1,1,1,1,1];
[ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ]
gap> c2:=Codeword(v);
[ 1 1 1 1 1 1 1 1 1 ]
gap> c3:=c1+c2;
[ 0 1 0 1 0 1 0 1 0 ]
```

```
gap> Weight(c1);
5
gap> Weight(c2);
9
gap> Weight(c3);
4
```

The previous excerpt from a GAP session shows that codewords can be constructed from quoted strings or from vectors whose entries lie in a finite field. We also see that codewords can be added together and that there is a function called `Weight` which (if it isn't obvious) tells us how many entries in a codeword are non-zero.

The *Hamming distance* is used extensively in coding theory. It tells us in how many positions two codewords differ. In GUAVA the Hamming distance is implemented by a function called `DistanceCodeword`.

Example

```
gap> DistanceCodeword(c1, c2);
4
```

Note that the Hamming distance between `c1` and `c2` happens to give the same value as the weight of their sum. This is no coincidence and has to do with the curious fact that in GF(2) adding and subtracting are the same thing.

A codeword can also be constructed using a polynomial. Indeed, the internal representation of a codeword requires either a polynomial or a vector. There are GUAVA functions that allow one to switch back and forth between the two representations.

Example

```
gap> x:=Indeterminate(GF(2), "x");;
gap> c4:=Codeword(x^7+x^2+x+1);
x^7 + x^2 + x + 1
gap> VectorCodeword(c4);
<an immutable GF2 vector of length 8>
gap> Display(last);
[ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ]
gap> c5:=Codeword([1,0,0,0,0,0,1]);
[ 1 0 0 0 0 0 1 ]
gap> PolyCodeword(c5);
x^6+Z(2)^0
```

2.2 Calculations with codes

A code is fundamentally just a collection of codewords. Sometimes a code is merely a *set* of codewords. Other times a code will be the vector space generated by some small set of codewords.

First let's build a code that is merely a set:

Example

```
gap> l:=["111000", "011100", "001110", "000111", "100011", "110001", "000000", "111111" ];;
gap> m:=Codeword(l,6,GF(2));
[ [ 1 1 1 0 0 0 ], [ 0 1 1 1 0 0 ], [ 0 0 1 1 1 0 ], [ 0 0 0 1 1 1 ],
  [ 1 0 0 0 1 1 ], [ 1 1 0 0 0 1 ], [ 0 0 0 0 0 0 ], [ 1 1 1 1 1 1 ] ]
gap> C1:=ElementsCode(m, GF(2));
a (6,8,1..6)2..3 user defined unrestricted code over GF(2)
```

```
gap> IsLinearCode(C1);
false
gap> WeightDistribution(C1);
[ 1, 0, 0, 6, 0, 0, 1 ]
```

In this example we first wrote out a list of strings, then converted them into codewords over GF(2). The call to `ElementsCode` constructs a code from a list of elements. It is possible that the set of codewords we used actually is a vector space, but the call to `IsLinearCode` says no. Finally the last function tells us that there are 6 codewords of weight 3, and one each of weights 0 and 6 in this code.

A very useful feature of GUAVA is the ability to construct random codes:

Example

```
gap> C:= RandomLinearCode(12,5,GF(2));
a  [12,5,?] randomly generated code over GF(2)
```

An error-correcting code's properties are fairly well captured by three numbers which traditionally are referred to using the letters n, k and d. We ask for a random code by specifying n (the wordlength), and k (the code's dimension) as well as the field which serves as the alphabet for the code.

One of the most salient features of a code (a feature that determines how good it will be at correcting errors) is its minimum weight, d. This is the smallest weight of any nonzero word in the code. If we wish to correct m errors we will need to have a minimum weight of at least $2m+1$.

Example

```
gap> MinimumWeight(C);
3
```

This particular code would be capable of correcting single bit errors.

Finally, one might be interested in the entire distribution of the weights of the words in a code. The weight distribution is a vector that tells us how many words there are in a code with each possible weight between 0 and n.

Example

```
gap> WeightDistribution(C);
[ 1, 0, 0, 2, 3, 6, 7, 6, 4, 2, 1, 0, 0 ]
```

Chapter 3

Codewords

Let $GF(q)$ denote a finite field with q (a prime power) elements. A *code* is a subset C of some finite-dimensional vector space V over $GF(q)$. The *length* of C is the dimension of V. Usually, $V = GF(q)^n$ and the length is the number of coordinate entries. When C is itself a vector space over $GF(q)$ then it is called a *linear code* and the *dimension* of C is its dimension as a vector space over $GF(q)$.

In GUAVA, a 'codeword' is a GAP record, with one of its components being an element in V. Likewise, a 'code' is a GAP record, with one of its components being a subset (or subspace with given basis, if C is linear) of V.

Example

```
gap> C:=RandomLinearCode(20,10,GF(4));
a  [20,10,?] randomly generated code over GF(4)
gap> c:=Random(C);
[ 1 0 a^2 a a 1 1 0 1 0 a^2 1 0 0 a a a 1 a 0 ]
gap> Set(NamesOfComponents(C));
[ "Basis", "Dimension", "GeneratorMat",
  "GeneratorsOfLeftOperatorAdditiveGroup", "LeftActingDomain",
  "NiceFreeLeftModule", "Representative", "WordLength", "ZeroImmutable",
  "lowerBoundMinimumDistance", "name", "upperBoundMinimumDistance" ]
gap> Set(NamesOfComponents(c));
[ "VectorCodeword", "WordLength", "treatAsPoly" ]
gap> c!.VectorCodeword;
< immutable compressed vector length 20 over GF(4) >
gap> Display(last);
[ Z(2)^0, 0*Z(2), Z(2^2)^2, Z(2^2), Z(2^2), Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0,
   0*Z(2), Z(2^2)^2, Z(2)^0, 0*Z(2), 0*Z(2), Z(2^2), Z(2^2), Z(2^2), Z(2)^0,
   Z(2^2), 0*Z(2) ]
gap> C!.Dimension;
10
```

Mathematically, a 'codeword' is an element of a code C, but in GUAVA the `Codeword` and `VectorCodeword` commands have implementations which do not check if the codeword belongs to C (i.e., are independent of the code itself). They exist primarily to make it easier for the user to construct the associated GAP record. Using these commands, one can enter into GAP both a codeword c (belonging to C) and a received word r (not belonging to C) using the same command. The user can input codewords in different formats (as strings, vectors, and polynomials), and output information is formatted in a readable way.

A codeword c in a linear code C arises in practice by an initial encoding of a 'block' message m, adding enough redundancy to recover m after c is transmitted via a 'noisy' communication medium. In GUAVA, for linear codes, the map $m \longmapsto c$ is computed using the command `c:=m*C` and recovering m from c is obtained by the command `InformationWord(C,c)`. These commands are explained more below.

Many operations are available on codewords themselves, although codewords also work together with codes (see chapter 4 on Codes).

The first section describes how codewords are constructed (see `Codeword` (3.1.1) and `IsCodeword` (3.1.3)). Sections 3.2 and 3.3 describe the arithmetic operations applicable to codewords. Section 3.4 describes functions that convert codewords back to vectors or polynomials (see `VectorCodeword` (3.4.1) and `PolyCodeword` (3.4.2)). Section 3.5 describes functions that change the way a codeword is displayed (see `TreatAsVector` (3.5.1) and `TreatAsPoly` (3.5.2)). Finally, Section 3.6 describes a function to generate a null word (see `NullWord` (3.6.1)) and some functions for extracting properties of codewords (see `DistanceCodeword` (3.6.2), `Support` (3.6.3) and `WeightCodeword` (3.6.4)).

3.1 Construction of Codewords

3.1.1 Codeword

▷ `Codeword(obj[, n][, F])` (function)

`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 off 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. In fact, no finite field of size larger than 11 arises in this fashion at all.

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*.

Example

```
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]);
x^2+Z(2)^0
gap> Codeword(p);
x^2 + 1
```

This command can also be called using the syntax `Codeword(obj,C)`. In this format, the elements of *obj* are converted to elements of the same ambient vector space as the elements of a code *C*. The command `Codeword(c,C)` is the same as calling `Codeword(c,n,F)`, where *n* is the word length of *C* and the *F* is the ground field of *C*.

Example

```
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 ] ]
gap> C:=RandomLinearCode(10,5,GF(3));
a  [10,5,?] randomly generated code over GF(3)
gap> Codeword("1000000000",C);
[ 1 0 0 0 0 0 0 0 0 0 ]
gap> Codeword("1000000000",10,GF(3));
[ 1 0 0 0 0 0 0 0 0 0 ]
```

3.1.2 CodewordNr

▷ `CodewordNr(C, list)` (function)

`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 ith element is taken. Otherwise, it is calculated by multiplication of the ith information vector by the generator matrix or generator polynomial, where the information vectors are ordered lexicographically. In particular, the returned codeword(s) could be a vector or a polynomial. So `CodewordNr(C, i)` is equal to `AsSSortedList(C)[i]`, described in the next chapter. The latter function first calculates the set of all the elements of C and then returns the ith element of that set, whereas the former only calculates the ith codeword.

Example

```
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> 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 ] ]
```

3.1.3 IsCodeword

▷ `IsCodeword(`*obj*`)` (function)

`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.

Example

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

3.2 Comparisons of Codewords

3.2.1 \= (for codewords)

▷ `\=(`*c1*`, `*c2*`)` (method)

The equality operator `c1 = c2` evaluates to 'true' if the codewords *c1* and *c2* are 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 *c2* is the codeword, the other object *c1* 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 *c1* is the codeword, then problems may arise if *c2* is a polynomial. In that case, the comparison always yields a 'false', because the polynomial comparison is called.

The equality operator is also denoted `EQ`, and `EQ(c1,c2)` is the same as `c1 = c2`. There is also an inequality operator, `< >`, or `not EQ`.

Example

```
gap> P := UnivariatePolynomial(GF(2), Z(2)*[1,0,0,1]);
x^3+Z(2)^0
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
gap> C:=HammingCode(3);
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> c1:=Random(C);
[ 1 1 1 0 0 0 0 ]
gap> c2:=Random(C);
[ 0 1 1 0 0 1 1 ]
gap> EQ(c1,c2);
false
gap> not EQ(c1,c2);
true
```

3.3 Arithmetic Operations for Codewords

3.3.1 \+ (for codewords)

▷ `\+(c1, c2)` (method)

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.

The operator + evaluates to the sum of the codewords *c1* and *c2*.

Example

```
gap> C:=RandomLinearCode(10,5,GF(3));
a  [10,5,?] randomly generated code over GF(3)
gap> c:=Random(C);
[ 0 2 2 2 0 1 2 2 2 1 ]
gap> Codeword(c+"2000000000");
[ 2 2 2 2 0 1 2 2 2 1 ]
gap> Codeword(c+"1000000000");
Error, <x> and <y> have different characteristic
```

The last command returns a GAP ERROR since the 'codeword' which GUAVA associates to "1000000000" belongs to $GF(2)$ and not $GF(3)$.

3.3.2 \- (for codewords)

▷ `\-(c1, c2)` (method)

Similar to addition: the operator - evaluates to the difference of the codewords *c1* and *c2*.

3.3.3 \+ (for codeword and code)

▷ `\+(v, C)` (method)

The operator `v+C` evaluates to the coset code of code `C` after adding a 'codeword' `v` to all codewords in `C`. Note that if $c \in C$ then mathematically $c + C = C$ but GUAVA only sees them equal as *sets*. See `CosetCode` (6.1.17).

Note that the command `C+v` returns the same output as the command `v+C`.

Example

```
gap> C:=RandomLinearCode(10,5);
a  [10,5,?] randomly generated code over GF(2)
gap> c:=Random(C);
[ 0 0 0 1 1 1 0 1 1 0 ]
gap> c+C;
<add. coset of a  [10,5,?] randomly generated code over GF(2)>
gap> c+C=C;
true
gap> IsLinearCode(c+C);
false
gap> v:=Codeword("100000000");
[ 1 0 0 0 0 0 0 0 0 ]
gap> v+C;
<add. coset of a  [10,5,?] randomly generated code over GF(2)>
gap> C=v+C;
false
gap> C := GeneratorMatCode( [ [1,0,0,0], [0,1,0,0] ], GF(2) );
a linear [4,2,1]1..2 code defined by generator matrix over GF(2)
gap> Elements(C);
[ [ 0 0 0 0 ], [ 0 1 0 0 ], [ 1 0 0 0 ], [ 1 1 0 0 ] ]
gap> v:=Codeword("0011");
[ 0 0 1 1 ]
gap> C+v;
<add. coset of a linear [4,2,1]1..2 code defined by generator matrix over GF(2)>
gap> Elements(C+v);
[ [ 0 0 1 1 ], [ 0 1 1 1 ], [ 1 0 1 1 ], [ 1 1 1 1 ] ]
```

In general, the operations just described can also be performed on codewords expressed as 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 4.2.

3.4 Functions that Convert Codewords to Vectors or Polynomials

3.4.1 VectorCodeword

▷ `VectorCodeword(`*obj*`)` (function)

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

Example

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

3.4.2 PolyCodeword

▷ `PolyCodeword(obj)` (function)

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

Example

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

3.5 Functions that Change the Display Form of a Codeword

3.5.1 TreatAsVector

▷ `TreatAsVector(obj)` (function)

`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 `VectorCodeword` (3.4.1)).

Example

```
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 ]
```

3.5.2 TreatAsPoly

▷ `TreatAsPoly(obj)` (function)

`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 `PolyCodeword` (3.4.2)).

Example

```
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
```

3.6 Other Codeword Functions

3.6.1 NullWord

▷ `NullWord(`*n*`, `*F*`)` (function)

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

Example

```
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 ]
```

3.6.2 DistanceCodeword

▷ `DistanceCodeword(`*c1*`, `*c2*`)` (function)

`DistanceCodeword` returns the Hamming distance from *c1* to *c2*. 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.

Example

```
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
```

3.6.3 Support

▷ `Support(c)` (function)

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

Example

```
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.

Example

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

3.6.4 WeightCodeword

▷ `WeightCodeword(c)` (function)

`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.

Example

```
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
```

Chapter 4

Codes

A *code* is a set of codewords (recall a codeword in GUAVA is simply a sequence of elements of a finite field $GF(q)$, where q is a prime power). We call these the *elements* of the code. 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 3.

In GUAVA, codes can be a set specified by its elements (this will be called an *unrestricted code*), by a generator matrix listing a set of basis elements (for a linear code) or by a generator polynomial (for a cyclic code).

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

Example

```
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 (except in the case where the code is a random linear codes, where these are not printed for efficiency reasons). 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.

Example

```
gap> C := ElementsCode(["1100", "1010", "0001"], "example code", GF(2) );
a (4,3,1..4)2..4 example code over GF(2)
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*. By definition, the rows of the generator matrix is a basis for the code (as a vector space over $GF(q)$). By definition, the rows of the parity check matrix is a basis for the dual space of the code,

$$C^* = \{v \in GF(q)^n \mid v \cdot c = 0, \ for\ all\ c \in C\}.$$

Example

```
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 codewords (regarded as n-tuples) are again codewords, the code is called *cyclic*. All elements of a cyclic code are multiples of the monic polynomial modulo a polynomial $x^n - 1$, where n is the word length of the code. Such a polynomial is called a *generator polynomial* The generator polynomial must divide $x^n - 1$ and its quotient is called a *check polynomial*. Multiplying a codeword in a cyclic code by the check polynomial yields zero (modulo the polynomial $x^n - 1$). In GUAVA, a cyclic code can be defined by either its generator polynomial or check polynomial.

Example

```
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 in fact linear. This situation occurs for example when a code is generated from a list of elements with the function `ElementsCode` (see `ElementsCode` (5.1.1)). By calling the function `IsLinearCode` (see `IsLinearCode` (4.3.4)), GUAVA tests if the code can be represented by a generator matrix. If so, the code record and the operations are converted accordingly.

Example

```
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
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 actually are cyclic.

Codes are printed simply by giving a small description of their parameters, the word length, size or dimension and perhaps the 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 functions `Decode` or `Decodeword`. For more information about encoding and decoding, see sections 4.2 and 4.10.1.

Example

```
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. Corrected by Guava.
[ 1 0 1 1 ]
```

Sections 4.1 and 4.2 describe the operations that are available for codes. Section 4.3 describes the functions that test whether an object is a code and what kind of code it is (see `IsCode`, `IsLinearCode` (4.3.4) and `IsCyclicCode`) and various other boolean functions for codes. Section 4.4 describes functions about equivalence and isomorphism of codes (see `IsEquivalent` (4.4.1), `CodeIsomorphism` (4.4.2) and `AutomorphismGroup` (4.4.3)). Section 4.5 describes functions that work on *domains* (see Chapter "Domains and their Elements" in the GAP Reference Manual). Section 4.6 describes functions for printing and displaying codes. Section 4.7 describes functions that return the matrices and polynomials that define a code (see `GeneratorMat` (4.7.1), `CheckMat` (4.7.2), `GeneratorPol` (4.7.3), `CheckPol` (4.7.4), `RootsOfCode` (4.7.5)). Section 4.8 describes functions that return the basic parameters of codes (see `WordLength` (4.8.1), `Redundancy` (4.8.2) and `MinimumDistance` (4.8.3)). Section 4.9 describes functions that return distance and weight distributions (see `WeightDistribution` (4.9.2), `InnerDistribution` (4.9.3), `OuterDistribution` (4.9.5) and `DistancesDistribution` (4.9.4)). Section 4.10 describes functions that are related to decoding (see `Decode` (4.10.1), `Decodeword` (4.10.2), `Syndrome` (4.10.8), `SyndromeTable` (4.10.9) and `StandardArray` (4.10.10)). In Chapters 5 and 6 which follow, we describe functions that generate and manipulate codes.

4.1 Comparisons of Codes

4.1.1 \= (for codes)

▷ `\=(C1, C2)` (method)

The equality operator `C1 = C2` evaluates to 'true' if the codes *C1* and *C2* are equal, and to 'false' otherwise.

The equality operator is also denoted `EQ`, and `Eq(C1,C2)` is the same as `C1 = C2`. There is also an inequality operator, `<>`, or `not EQ`.

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

Example

```
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 `IsEquivalent` (4.4.1)). By the way, this called `CodeIsomorphism`. For the current version of GUAVA, unless one of the codes is unrestricted, this calls Leon's C program (which only works for binary linear codes and only on a unix/linux computer).

4.2 Operations for Codes

4.2.1 \+ (for codes)

▷ `\+(C1, C2)` (method)

The operator '+' evaluates to the direct sum of the codes *C1* and *C2*. See `DirectSumCode` (6.2.1).

Example

```
gap> C1:=RandomLinearCode(10,5);
a  [10,5,?] randomly generated code over GF(2)
gap> C2:=RandomLinearCode(9,4);
a  [9,4,?] randomly generated code over GF(2)
gap> C1+C2;
a linear [19,9,1..2]4..9 direct sum code
```

4.2.2 * (for codes)

▷ `\*(C1, C2)` (method)

The operator '*' evaluates to the direct product of the codes *C1* and *C2*. See `DirectProductCode` (6.2.3).

Example

```
gap> C1 := GeneratorMatCode( [ [1, 0,0,0], [0, 1,0,0] ], GF(2) );
a linear [4,2,1]1..2 code defined by generator matrix over GF(2)
gap> C2 := GeneratorMatCode( [ [0,0,1, 1], [0,0,0, 1] ], GF(2) );
a linear [4,2,1]1..2 code defined by generator matrix over GF(2)
gap> C1*C2;
a linear [16,4,1]4..12 direct product code
```

4.2.3 * (for message and code)

▷ `\*(m, C)` (method)

The operator `m*C` evaluates to the element of *C* belonging to information word ('message') *m*. Here *m* 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 *m* by the generator polynomial of *C*. If *C* is a linear code, it is equal to the multiplication of an information vector *m* by a generator matrix of *C*.

To invert this, use the function `InformationWord` (see `InformationWord` (4.2.4), which simply calls the function `Decode`).

Example

```
gap> C := GeneratorMatCode( [ [1, 0,0,0], [0, 1,0,0] ], GF(2) );
a linear [4,2,1]1..2 code defined by generator matrix over GF(2)
gap> m:=Codeword("11");
[ 1 1 ]
gap> m*C;
[ 1 1 0 0 ]
```

4.2.4 InformationWord

▷ `InformationWord(C, c)` (function)

Here *C* is a linear code and *c* is a codeword in it. The command `InformationWord` returns the message word (or 'information digits') *m* satisfying `c=m*C`. This command simply calls `Decode`, provided `c in C` is true. Otherwise, it returns an error.

To invert this, use the encoding function `*` (see `\*` (4.2.3)).

Example

```
gap> C:=HammingCode(3);
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> c:=Random(C);
[ 1 1 0 1 0 0 1 ]
gap> InformationWord(C,c);
[ 0 0 0 1 ]
gap> c:=Codeword("1111100");
[ 1 1 1 1 1 0 0 ]
gap> InformationWord(C,c);
Error, ERROR: codeword must belong to code
gap> C:=NordstromRobinsonCode();
a (16,256,6)4 Nordstrom-Robinson code over GF(2)
gap> c:=Random(C);
[ 1 0 0 1 0 0 0 0 0 1 1 1 1 0 1 1 ]
gap> InformationWord(C,c);
"ERROR: code must be linear"
```

4.3 Boolean Functions for Codes

4.3.1 in

▷ `in(c, C)` (function)

The command `c in C` 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.

Example

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

4.3.2 IsSubset

▷ `IsSubset(C1, C2)` (function)

The command `IsSubset(C1,C2)` returns 'true' if *C2* is a subcode of *C1*, i.e. if *C1* contains all the elements of *C2*.

Example

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

4.3.3 IsCode

▷ `IsCode(obj)` (function)

`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.

Example

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

4.3.4 IsLinearCode

▷ `IsLinearCode(obj)` (function)

`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 recorded as a generator matrix of *obj* and the function returns 'true'. If not, the function returns 'false'.

Example

```
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
```

4.3.5 IsCyclicCode

▷ `IsCyclicCode(obj)` (function)

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 recorded as a generator polynomial of *obj* and the function returns 'true'. If not, the function returns 'false'.

Example

```
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
```

4.3.6 IsPerfectCode

▷ IsPerfectCode(*C*) (function)

IsPerfectCode(C) returns 'true' if *C* is a perfect code. If $C \subset GF(q)^n$ then, by definition, this means that for some positive integer t, the space $GF(q)^n$ is covered by non-overlapping spheres of (Hamming) radius t centered at the codewords in *C*. 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 "trivially 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 (see section 1.12 in [HP03]).

Example

```
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
```

4.3.7 IsMDSCode

▷ IsMDSCode(*C*) (function)

IsMDSCode(C) returns true if *C* is a maximum distance separable (MDS) code. 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 UpperBoundSingleton (7.1.1)). 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.

Example

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

4.3.8 IsSelfDualCode

▷ `IsSelfDualCode(C)` (function)

`IsSelfDualCode(C)` returns 'true' if `C` is self-dual, i.e. when `C` is equal to its dual code (see also `DualCode` (6.1.14)). A code is self-dual if it contains all vectors that its elements are orthogonal to. If a code is self-dual, it automatically is self-orthogonal (see `IsSelfOrthogonalCode` (4.3.9)).

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

Example

```
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
```

4.3.9 IsSelfOrthogonalCode

▷ `IsSelfOrthogonalCode(C)` (function)

`IsSelfOrthogonalCode(C)` 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.)

Example

```
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
```

4.3.10 IsDoublyEvenCode

▷ `IsDoublyEvenCode(C)` (function)

`IsDoublyEvenCode(C)` returns 'true' if C is a binary linear code which has codewords of weight divisible by 4 only. According to [HP03], a doubly-even code is self-orthogonal and every row in its generator matrix has weight that is divisible by 4.

Example

```
gap> C:=BinaryGolayCode();
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> WeightDistribution(C);
[ 1, 0, 0, 0, 0, 0, 0, 253, 506, 0, 0, 1288, 1288, 0, 0, 506, 253, 0, 0, 0,
  0, 0, 0, 1 ]
gap> IsDoublyEvenCode(C);
false
gap> C:=ExtendedCode(C);
a linear [24,12,8]4 extended code
gap> WeightDistribution(C);
[ 1, 0, 0, 0, 0, 0, 0, 0, 759, 0, 0, 0, 2576, 0, 0, 0, 759, 0, 0, 0, 0, 0, 0,
  0, 1 ]
gap> IsDoublyEvenCode(C);
true
```

4.3.11 IsSinglyEvenCode

▷ `IsSinglyEvenCode(C)` (function)

`IsSinglyEvenCode(C)` returns 'true' if C is a binary self-orthogonal linear code which is not doubly-even. In other words, C is a binary self-orthogonal code which has codewords of even weight.

Example

```
gap> x:=Indeterminate(GF(2),"x");
x
gap> C:=QuasiCyclicCode( [x^0, 1+x^3+x^5+x^6+x^7], 11, GF(2) );
a linear [22,11,1..6]4..7 quasi-cyclic code over GF(2)
gap> IsSelfDualCode(C);  # self-dual is a restriction of self-orthogonal
true
gap> IsDoublyEvenCode(C);
false
gap> IsSinglyEvenCode(C);
true
```

4.3.12 IsEvenCode

▷ `IsEvenCode(C)` (function)

`IsEvenCode(C)` returns 'true' if C is a binary linear code which has codewords of even weight--regardless whether or not it is self-orthogonal.

Example

```
gap> C:=BinaryGolayCode();
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> IsSelfOrthogonalCode(C);
false
```

```
gap> IsEvenCode(C);
false
gap> C:=ExtendedCode(C);
a linear [24,12,8]4 extended code
gap> IsSelfOrthogonalCode(C);
true
gap> IsEvenCode(C);
true
gap> C:=ExtendedCode(QRCode(17,GF(2)));
a linear [18,9,6]3..5 extended code
gap> IsSelfOrthogonalCode(C);
false
gap> IsEvenCode(C);
true
```

4.3.13 IsSelfComplementaryCode

▷ `IsSelfComplementaryCode(C)` (function)

`IsSelfComplementaryCode` returns 'true' if

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

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

Example

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

4.3.14 IsAffineCode

▷ `IsAffineCode(C)` (function)

`IsAffineCode` returns 'true' if C 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.

Example

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

4.3.15 IsAlmostAffineCode

▷ `IsAlmostAffineCode(C)` (function)

`IsAlmostAffineCode` returns 'true' if `C` is an almost affine code. A code is called *almost affine* if the size of any punctured code of `C` 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.

Example

```
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
```

4.4 Equivalence and Isomorphism of Codes

4.4.1 IsEquivalent

▷ `IsEquivalent(C1, C2)` (function)

We say that `C1` is *permutation equivalent* to `C2` if `C1` can be obtained from `C2` by carrying out column permutations. `IsEquivalent` returns true if `C1` and `C2` are equivalent codes. At this time, `IsEquivalent` only handles *binary* codes. (The external unix/linux program DESAUTO from J. S. Leon is called by `IsEquivalent`.) Of course, if `C1` and `C2` are equal, they are also equivalent.

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

More generally, we say that `C1` is *equivalent* to `C2` if `C1` can be obtained from `C2` by carrying out column permutations and a permutation of the alphabet.

Example

```
gap> x:= Indeterminate( GF(2), "x" );; pol:= x^3+x+1;
x^3+x+Z(2)^0
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

# verify that H is equivalent to a Hamming code
gap> IsEquivalent(H, HammingCode(3, GF(2)));
true
gap> CodeIsomorphism(H, HammingCode(3, GF(2)));
(3,4)(5,6,7)
```

4.4.2 CodeIsomorphism

▷ `CodeIsomorphism(C1, C2)` (function)

If the two codes *C1* and *C2* are permutation equivalent codes (see IsEquivalent (4.4.1)), CodeIsomorphism returns the permutation that transforms *C1* into *C2*. If the codes are not equivalent, it returns 'false'.

At this time, IsEquivalent only computes isomorphisms between *binary* codes on a linux/unix computer (since it calls Leon's C program DESAUTO).

Example

```
gap> x:= Indeterminate( GF(2), "x" );; pol:= x^3+x+1;
x^3+x+Z(2)^0
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
```

4.4.3 AutomorphismGroup

▷ AutomorphismGroup(*C*) (function)

AutomorphismGroup returns the automorphism group of a linear code *C*. For a binary code, the automorphism group is the largest permutation group of degree *n* such that each permutation applied to the columns of *C* again yields *C*. GUAVA calls the external program DESAUTO written by J. S. Leon, if it exists, to compute the automorphism group. If Leon's program is not compiled on the system (and in the default directory) then it calls instead the much slower program PermutationAutomorphismGroup.

See Leon [Leo82] for a more precise description of the method, and the guava/src/leon/doc subdirectory for details about Leon's C programs.

The function PermutedCode permutes the columns of a code (see PermutedCode (6.1.4)).

Example

```
gap> R := RepetitionCode(7,GF(2));
a cyclic [7,1,7]3 repetition code over GF(2)

# verify that every permutation keeps R identical, that is, the
# automorphism group is Sym(7)
gap> AutomorphismGroup(R);
Sym( [ 1 .. 7 ] )
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
```

4.4.4 PermutationAutomorphismGroup

▷ `PermutationAutomorphismGroup(C)` (function)

`PermutationAutomorphismGroup` returns the permutation automorphism group of a linear code `C`. This is the largest permutation group of degree n such that each permutation applied to the columns of `C` again yields `C`. It is written in GAP, so is much slower than `AutomorphismGroup`.

When `C` is binary `PermutationAutomorphismGroup` does *not* call `AutomorphismGroup`, even though they agree mathematically in that case. This way `PermutationAutomorphismGroup` can be called on any platform which runs GAP.

The older name for this command, `PermutationGroup`, will become obsolete in the next version of GAP.

Example

```
gap> R := RepetitionCode(3,GF(3));
a cyclic [3,1,3]2 repetition code over GF(3)
gap> G:=PermutationAutomorphismGroup(R);;
gap> G=SymmetricGroup(3);
true
```

4.5 Domain Functions for Codes

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

4.5.1 IsFinite

▷ `IsFinite(C)` (function)

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

Example

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

4.5.2 Size

▷ `Size(C)` (function)

`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.

Example

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

4.5.3 LeftActingDomain

▷ `LeftActingDomain(C)` (function)

`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.

Example

```
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)
```

4.5.4 Dimension

▷ `Dimension(C)` (function)

`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.

Example

```
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
```

4.5.5 AsSSortedList

▷ `AsSSortedList(C)` (function)

`AsSSortedList` (as strictly sorted list) returns an immutable, duplicate free list of the elements of `C`. For a finite field $GF(q)$ generated by powers of $Z(q)$, the ordering on

$$GF(q) = \{0, Z(q)^0, Z(q), Z(q)^2, \ldots Z(q)^{q-2}\}$$

is that determined by the exponents i. These elements are of the type codeword (see `Codeword` (3.1.1)). 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 `CodewordNr` (3.1.2)).

Example

```
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 ] ]
```

4.6 Printing and Displaying Codes

4.6.1 Print

▷ `Print(C)` (function)

`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.

Except for codes produced by `RandomLinearCode`, 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. For certain ranges of n, the values of *lowerbound* and *upperbound* are obtained from tables.

The function `Display` gives more information. See `Display` (4.6.3).

Example

```
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).
```

4.6.2 String

▷ `String(C)` (function)

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

Example

```
gap> x:= Indeterminate( GF(3), "x" );; pol:= x^2+1;
x^2+Z(3)^0
gap> Factors(pol);
[ x^2+Z(3)^0 ]
gap> H := GeneratorPolCode( pol, 8, GF(3));
```

```
a cyclic [8,6,1..2]1..2 code defined by generator polynomial over GF(3)
gap> String(H);
"a cyclic [8,6,1..2]1..2 code defined by generator polynomial over GF(3)"
```

4.6.3 Display

▷ Display(*C*) (function)

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 Print (4.6.1)).

Example

```
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)
```

4.6.4 DisplayBoundsInfo

▷ DisplayBoundsInfo(*bds*) (function)

DisplayBoundsInfo prints the method of construction of the code *C* indicated in bds:= BoundsMinimumDistance(n, k, GF(q)).

Example

```
gap> bounds := BoundsMinimumDistance( 20, 17, GF(4) );;
gap> DisplayBoundsInfo(bounds);
an optimal linear [20,17,d] code over GF(4) has d=3
--------------------------------------------------------------------------------
Lb(20,17)=3, by shortening of:
Lb(21,18)=3, by applying construction B to a [81,77,3] code
Lb(81,77)=3, by shortening of:
Lb(85,81)=3, reference: Ham
--------------------------------------------------------------------------------
Ub(20,17)=3, by considering shortening to:
Ub(7,4)=3, by considering puncturing to:
Ub(6,4)=2, by construction B applied to:
Ub(2,1)=2, repetition code
--------------------------------------------------------------------------------
Reference Ham:
%T this reference is unknown, for more info
%T contact A.E. Brouwer (aeb@cwi.nl)
```

4.7 Generating (Check) Matrices and Polynomials

4.7.1 GeneratorMat

▷ `GeneratorMat(C)` (function)

`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.

Example

```
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> Display(last);
 1 1 1 . . . .
 1 . . 1 1 . .
 . 1 . 1 . 1 .
 1 1 . 1 . . 1
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) ) );
[  ]
```

4.7.2 CheckMat

▷ `CheckMat(C)` (function)

`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.

Example

```
gap> CheckMat( HammingCode(3, GF(2) ) );
[ [ 0*Z(2), 0*Z(2), 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), Z(2)^0, Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ] ]
gap> Display(last);
 . . . 1 1 1 1
 . 1 1 . . 1 1
 1 . 1 . 1 . 1
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) ) );
[  ]
```

4.7.3 GeneratorPol

▷ `GeneratorPol(C)` (function)

`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'.

Example

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

4.7.4 CheckPol

▷ `CheckPol(C)` (function)

`CheckPol` returns the check polynomial of C. The code consists of all polynomials f with

$$f \cdot h \equiv 0 \pmod{x^n - 1},$$

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.

Example

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

4.7.5 RootsOfCode

▷ `RootsOfCode(C)` (function)

`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 process, constructing a code from a set of roots, can be carried out by the function RootsCode (see RootsCode (5.5.3)).

Example

```
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
```

4.8 Parameters of Codes

4.8.1 WordLength

▷ WordLength(*C*) (function)

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.

Example

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

4.8.2 Redundancy

▷ Redundancy(*C*) (function)

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 $r = n-k$.

Example

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

4.8.3 MinimumDistance

▷ MinimumDistance(*C*) (function)

`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 `DistanceCodeword` (3.6.2)) 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 $w = (w[1], w[2], ..., w[n])$ is the weight distribution of `C` (see `WeightDistribution` (4.9.2)). For unrestricted codes, d is the smallest positive value with $w[d+1] \neq 0$, where w is the inner distribution of `C` (see `InnerDistribution` (4.9.3)).

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

For linear codes `C`, the algorithm used is the following: After replacing `C` by a permutation equivalent `C'`, one may assume the generator matrix has the following form $G = (I_k | A)$, for some $k \times (n-k)$ matrix A. If $A = 0$ then return $d(C) = 1$. Next, find the minimum distance of the code spanned by the rows of A. Call this distance $d(A)$. Note that $d(A)$ is equal to the Hamming distance $d(v, 0)$ where v is some proper linear combination of i distinct rows of A. Return $d(C) = d(A) + i$, where i is as in the previous step.

This command may also be called using the syntax `MinimumDistance(C, w)`. In this form, `MinimumDistance` returns the minimum distance of a codeword `w` to the code `C`, also called the *distance from* `w` *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 `DistancesDistribution` (4.9.4)).

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).

Example

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

See also the GUAVA commands relating to bounds on the minimum distance in section 7.1.

4.8.4 MinimumDistanceLeon

▷ `MinimumDistanceLeon(C)` (function)

`MinimumDistanceLeon` returns the "probable" minimum distance d_{Leon} of a linear binary code `C`, using an implementation of Leon's probabilistic polynomial time algorithm. Briefly: Let `C` be a linear code of dimension k over $GF(q)$ as above. The algorithm has input parameters s and ρ, where s is an integer between 2 and $n-k$, and ρ is an integer between 2 and k.

- Find a generator matrix G of C.
- Randomly permute the columns of G.
- Perform Gaussian elimination on the permuted matrix to obtain a new matrix of the following form:

  $$G = (I_k \,|\, Z \,|\, B)$$

  with Z a $k \times s$ matrix. If (Z,B) is the zero matrix then return 1 for the minimum distance. If $Z = 0$ but not B then either choose another permutation of the rows of `C` or return 'method fails'.
- Search Z for at most ρ rows that lead to codewords of weight less than ρ.
- For these codewords, compute the weight of the whole word in `C`. Return this weight.

(See for example J. S. Leon, [Leo88] for more details.) Sometimes (as is the case in GUAVA) this probabilistic algorithm is repeated several times and the most commonly occurring value is taken. (This function actually has two optional arguments: `p` and `num`. The command `MinimumDistanceLeon(C,p,num)` allows a bit more flexibility for the user - see the GAP code in codeops.gi for details.)

Example

```
gap> C:=RandomLinearCode(50,22,GF(2));
a  [50,22,?] randomly generated code over GF(2)
gap> MinimumDistanceLeon(C);
8
gap> MinimumDistance(C);
8
```

4.8.5 MinimumWeight

▷ `MinimumWeight(C)` (function)

`MinimumWeight` returns the minimum Hamming weight of a linear code `C`. At the moment, this function works for binary and ternary codes only. The `MinimumWeight` function relies on an external executable program which is written in C language. As a consequence, the execution time of `MinimumWeight` function is faster than that of `MinimumDistance` (4.8.3) function.

The `MinimumWeight` function implements Chen's [Che69] algorithm if `C` is cyclic, and Zimmermann's [Zim96] algorithm if `C` is a general linear code. This function has a built-in check on the constraints of the minimum weight codewords. For example, for a self-orthogonal code over GF(3), the minimum weight codewords have weight that is divisible by 3, i.e. 0 mod 3 congruence. Similarly, self-orthogonal codes over GF(2) have codeword weights that are divisible by 4 and even codes over GF(2) have codewords weights that are divisible by 2. By taking these constraints into account, in many cases, the execution time may be significantly reduced. Consider the minimum Hamming weight enumeration of the $[151,45]$ binary cyclic code (second example below). This cyclic code is self-orthogonal, so the weight of all codewords is divisible by 4. Without considering this constraint,

the computation will finish at information weight 10, rather than 9. We can see that, this 0 mod 4 constraint on the codeword weights, has allowed us to avoid enumeration of $b(45,10) = 3,190,187,286$ additional codewords, where $b(n,k) = n!/((n-k)!k!)$ is the binomial coefficient of integers n and k.

Note that the C source code for this minimum weight computation has not yet been optimised, especially the code for GF(3), and there are chances to improve the speed of this function. Your contributions are most welcomed.

If you find any bugs on this function, please report it to ctjhai@plymouth.ac.uk.

Example

```
gap> # Extended ternary quadratic residue code of length 48
gap> n := 47;;
gap> x := Indeterminate(GF(3), "x");;
gap> F := Factors(x^n-1);;
gap> v := NullVector(n,GF(3));;
gap> v := v + ShallowCopy(VectorCodeword( Codeword(F[2]) ));;
gap> G := CirculantMatrix(24, v);;
gap> for i in [1..Size(G)] do Add(G[i], Sum(G[i])); od;;
gap> C := GeneratorMatCodeNC(G, GF(3));
a  [48,24,?] randomly generated code over GF(3)
gap> MinimumWeight(C);
15
gap> # Binary cyclic code [151,45,36]
gap> n := 151;;
gap> x := Indeterminate(GF(2), "x");;
gap> F := Factors(x^n-1);;
gap> C := CheckPolCode(F[2]*F[3]*F[3]*F[4], n, GF(2));
a cyclic [151,45,1..50]31..75 code defined by check polynomial over GF(2)
gap> MinimumWeight(C);
36
```

4.8.6 DecreaseMinimumDistanceUpperBound

▷ `DecreaseMinimumDistanceUpperBound(C, t, m)` (function)

`DecreaseMinimumDistanceUpperBound` is an implementation of the algorithm for the minimum distance of a linear binary code `C` by Leon [Leo88]. This algorithm tries to find codewords with small minimum weights. The parameter `t` is at least 1 and less than the dimension of `C`. The best results are obtained if it is close to the dimension of the code. The parameter `m` 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. (This was implemented before `MinimumDistanceLeon` but independently. The older manual had given the command incorrectly, so the command was only found after reading all the *.*gi* files in the GUAVA library. Though both `MinimumDistance` and `MinimumDistanceLeon` often run much faster than `DecreaseMinimumDistanceUpperBound`, `DecreaseMinimumDistanceUpperBound` appears to be more accurate than `MinimumDistanceLeon`.)

Example

```
gap> Reset(GlobalMersenneTwister, 1);; # make the example reproducible
gap> C:=RandomLinearCode(5,2,GF(2));
a  [5,2,?] randomly generated code over GF(2)
```

```
gap> DecreaseMinimumDistanceUpperBound(C,1,4);
rec( mindist := 1, word := [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] )
gap> MinimumDistance(C);
1
gap> C:=RandomLinearCode(8,4,GF(2));
a  [8,4,?] randomly generated code over GF(2)
gap> DecreaseMinimumDistanceUpperBound(C,3,4);
rec( mindist := 3,
  word := [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ] )
gap> MinimumDistance(C);
3
```

4.8.7 MinimumDistanceRandom

▷ `MinimumDistanceRandom(C, num, s)` (function)

`MinimumDistanceRandom` returns an upper bound for the minimum distance d_{random} of a linear binary code `C`, using a probabilistic polynomial time algorithm. Briefly: Let `C` be a linear code of dimension k over $GF(q)$ as above. The algorithm has input parameters *num* and s, where s is an integer between 2 and $n-1$, and *num* is an integer greater than or equal to 1.

- Find a generator matrix G of C.
- Randomly permute the columns of G, written G_p..
- $$G = (A, B)$$

  with A a $k \times s$ matrix. If A is the zero matrix then return 'method fails'.
- Search A for at most 5 rows that lead to codewords, in the code C_A with generator matrix A, of minimum weight.
- For these codewords, use the associated linear combination to compute the weight of the whole word in `C`. Return this weight and codeword.

This probabilistic algorithm is repeated *num* times (with different random permutations of the rows of G each time) and the weight and codeword of the lowest occurring weight is taken.

Example

```
gap> Reset(GlobalMersenneTwister, 1);; # make the example reproducible
gap> C:=RandomLinearCode(60,20,GF(2));
a  [60,20,?] randomly generated code over GF(2)
gap> #mindist(C); #time;
gap> #mindistleon(C,10,30); #time; #doesn't work well
gap> a:=MinimumDistanceRandom(C,10,30); #time; # done 10 times -with fastest time!!

 This is a probabilistic algorithm which may return the wrong answer.
[ 13, [ 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 1 1 0 0 1 1 0 0 1
    0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 ] ]
gap> a[2] in C;
true
```

```
gap> b:=DecreaseMinimumDistanceUpperBound(C,10,1); #time; #only done once!
rec( mindist := 13,
  word := [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2),
      0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2),
      Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2),
      0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2),
      0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2),
      0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0,
      0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ] )
gap> Codeword(b!.word) in C;
true
gap> MinimumDistance(C);
13
gap> c:=MinimumDistanceLeon(C);
12
gap> C:=RandomLinearCode(30,10,GF(3));
a  [30,10,?] randomly generated code over GF(3)
gap> a:=MinimumDistanceRandom(C,10,10);

 This is a probabilistic algorithm which may return the wrong answer.
[ 13, [ 0 0 0 0 0 0 0 1 0 2 0 0 0 2 1 2 2 1 0 1 0 1 1 1 0 1 0 0 2 0 ] ]
gap> a[2] in C;
true
gap> MinimumDistance(C);
9
gap> c:=MinimumDistanceLeon(C);
Code must be binary. Quitting.
0
gap> a:=MinimumDistanceRandom(C,1,29);

 This is a probabilistic algorithm which may return the wrong answer.
[ 9, [ 0 0 0 0 0 0 1 0 0 0 1 0 1 2 0 0 0 0 1 0 1 0 0 2 0 2 0 0 2 0 ] ]
```

4.8.8 CoveringRadius

▷ `CoveringRadius(C)` (function)

`CoveringRadius` returns the *covering radius* of a linear code `C`. This is the smallest number r with the property that each element v of the ambient vector space of `C` has at most a distance r to the code `C`. So for each vector v there must be an element c of `C` with $d(v,c) \leq r$. The smallest covering radius of any $[n,k]$ binary linear code is denoted $t(n,k)$. A binary linear code with reasonable small covering radius is called a *covering code*.

If `C` is a perfect code (see `IsPerfectCode` (4.3.6)), the covering radius is equal to t, the number of errors the code can correct, where $d = 2t+1$, with d the minimum distance of `C` (see `MinimumDistance` (4.8.3)).

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 BoundsCoveringRadius (7.2.1)), is 1, then the value in

```
C.boundsCoveringRadius
```

is returned. Otherwise, the function

```
C.operations.CoveringRadius
```

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

The algorithm used to compute the covering radius is the following. First, CosetLeadersMatFFE is used to compute the list of coset leaders (which returns a codeword in each coset of $GF(q)^n/C$ of minimum weight). Then WeightVecFFE is used to compute the weight of each of these coset leaders. The program returns the maximum of these weights.

Example

```
gap> H := RandomLinearCode(10, 5, GF(2));
a  [10,5,?] randomly generated code over GF(2)
gap> CoveringRadius(H);
3
gap> H := HammingCode(4, GF(2));; IsPerfectCode(H);
true
gap> CoveringRadius(H); # Hamming codes have minimum distance 3
1
gap> CoveringRadius(ReedSolomonCode(7,4));
3
gap> CoveringRadius( BCHCode( 17, 3, GF(2) ) );
3
gap> CoveringRadius( HammingCode( 5, GF(2) ) );
1
gap> C := ReedMullerCode( 1, 9 );;
gap> CoveringRadius( C );
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 ]
```

See also the GUAVA commands relating to bounds on the minimum distance in section 7.2.

4.8.9 SetCoveringRadius

▷ SetCoveringRadius(*C*, *intlist*) (function)

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.

Example

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

4.9 Distributions

4.9.1 MinimumWeightWords

▷ `MinimumWeightWords(C)` (function)

`MinimumWeightWords` returns the list of minimum weight codewords of C.
This algorithm is written in GAP and is rather slow. It is only suitable for small codes.
This does not call the very fast function `MinimumWeight` (see `MinimumWeight` (4.8.5)).

Example

```
gap> C:=HammingCode(3,GF(2));
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> MinimumWeightWords(C);
[ [ 1 0 0 0 0 1 1 ], [ 0 1 0 1 0 1 0 ], [ 0 1 0 0 1 0 1 ], [ 1 0 0 1 1 0 0 ],
  [ 0 0 1 0 1 1 0 ], [ 0 0 1 1 0 0 1 ], [ 1 1 1 0 0 0 0 ] ]
```

4.9.2 WeightDistribution

▷ `WeightDistribution(C)` (function)

`WeightDistribution` returns the weight distribution of C, as a vector. The ith 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 `InnerDistribution` (4.9.3)). If w is the weight distribution of a linear code C, it must have the zero codeword, so $w[1] = 1$ (one word of weight 0).

Some codes, such as the Hamming codes, have precomputed weight distributions. For others, the program WeightDistribution calls the GAP program `DistancesDistributionMatFFEVecFFE`, which is written in C. See also `CodeWeightEnumerator`.

Example

```
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 ]
```

4.9.3 InnerDistribution

▷ `InnerDistribution(C)` (function)

`InnerDistribution` returns the inner distribution of `C`. The ith 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 `WeightDistribution` (4.9.2)).

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 `MinimumDistance` (4.8.3).

Example

```
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 ]
```

4.9.4 DistancesDistribution

▷ `DistancesDistribution(C, w)` (function)

`DistancesDistribution` returns the distribution of the distances of all elements of `C` to a codeword `w` in the same vector space. The ith 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 `MinimumDistance` (4.8.3)).

Example

```
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); # distance to H
5
```

4.9.5 OuterDistribution

▷ `OuterDistribution(C)` (function)

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 pairs, the first coordinate being an element of $GF(q)^n$ (this is a codeword type) and the second coordinate being a 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 `DistancesDistribution` (4.9.4)) can be used to calculate one entry of the list.

Example

```
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
```

4.10 Decoding Functions

4.10.1 Decode

▷ `Decode(C, r)` (function)

`Decode` decodes `r` (a 'received word') with respect to code `C` and returns the 'message word' (i.e., the information digits associated to the codeword $c \in C$ closest to `r`). Here `r` can be a GUAVA codeword or a list of codewords. First, possible errors in `r` are corrected, then the codeword is decoded to an *information codeword m* (and not an element of `C`). If the code record has a field 'specialDecoder', this special algorithm is used to decode the vector. Hamming codes, BCH codes, cyclic codes, and generalized Reed-Solomon have such a special algorithm. (The algorithm used for BCH codes is the Sugiyama algorithm described, for example, in section 5.4.3 of [HP03]. A special decoder has also being written for the generalized Reed-Solomon code using the interpolation algorithm. For cyclic codes, the error-trapping algorithm is used.) If `C` is linear and no special decoder field has been set then syndrome decoding is used. Otherwise (when `C` is non-linear), the nearest neighbor decoding algorithm is used (which is very slow).

A special decoder can be created by defining a function

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

The function uses the arguments `C` (the code record itself) and `r` (a vector of the codeword type) to decode `r` to an information vector. A normal decoder would take a codeword `r` of the same word length and field as `C`, and would return an information vector 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.

Encoding is done by multiplying the information vector with the code (see 4.2).

Example

```
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("0011010")); # one error corrected
[ 1 0 1 0 ]
gap> C!.SpecialDecoder := function(C, c)
> return NullWord(Dimension(C));
> end;
function( C, c ) ... end
gap> Decode(C, c); # new decoder always returns null word
[ 0 0 0 0 ]
```

4.10.2 Decodeword

▷ `Decodeword(C, r)` (function)

`Decodeword` decodes `r` (a 'received word') with respect to code `C` and returns the codeword $c \in C$ closest to `r`. Here `r` can be a GUAVA codeword or a list of codewords. If the code record has a field 'specialDecoder', this special algorithm is used to decode the vector. Hamming codes, generalized Reed-Solomon codes, BCH codes and cyclic codes have such a special algorithm. (The algorithm used for BCH codes is the Sugiyama algorithm described, for example, in section 5.4.3 of [HP03]. The algorithm used for generalized Reed-Solomon codes is the "interpolation algorithm" described for example in chapter 5 of [JH04]. For cyclic codes, the error-trapping algorithm is used.) If `C` is linear and no special decoder field has been set then syndrome decoding is used. Otherwise, when `C` is non-linear, the nearest neighbor algorithm has been implemented (which should only be used for small-sized codes).

Example

```
gap> Reset(GlobalMersenneTwister, 1);; # make the example reproducible
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> Decodeword(C, c);                     # decoding
[ 1 0 1 1 0 1 0 ]
gap> R:=PolynomialRing(GF(11),["t"]);
GF(11)[t]
gap> P:=List([1,3,4,5,7],i->Z(11)^i);
[ Z(11), Z(11)^3, Z(11)^4, Z(11)^5, Z(11)^7 ]
gap> C:=GeneralizedReedSolomonCode(P,3,R);
a linear [5,3,1..3]2  generalized Reed-Solomon code over GF(11)
gap> MinimumDistance(C);
3
gap> c:=Random(C);
[ 10 7 7 5 3 ]
gap> v:=Codeword([ 10, 7, 7, 5, 2 ], GF(11));
[ 10 7 7 5 2 ]
gap> GeneralizedReedSolomonDecoderGao(C,v);
[ 10 7 7 5 3 ]
gap> Decodeword(C,v); # calls the special interpolation decoder
[ 10 7 7 5 3 ]
gap> G:=GeneratorMat(C);;
gap> Display(G);
  1  1  1  1  1
  2  8  5 10  7
  4  9  3  1  5
gap> C1:=GeneratorMatCode(G,GF(11));
a linear [5,3,1..3]2 code defined by generator matrix over GF(11)
gap> Decodeword(C1,v); # calls syndrome decoding
[ 10 7 7 5 3 ]
```

4.10.3 GeneralizedReedSolomonDecoderGao

▷ `GeneralizedReedSolomonDecoderGao(C, r)` (function)

`GeneralizedReedSolomonDecoderGao` decodes r (a 'received word') to a codeword $c \in C$ in a generalized Reed-Solomon code C (see `GeneralizedReedSolomonCode` (5.6.2)), closest to r. Here r must be a GUAVA codeword. If the code record does not have name 'generalized Reed-Solomon code' then an error is returned. Otherwise, the Gao decoder [Gao03] is used to compute c.

For long codes, this method is faster in practice than the interpolation method used in `Decodeword`.

Example

```
gap> Reset(GlobalMersenneTwister, 1);; # make the example reproducible
gap> R:=PolynomialRing(GF(11),["t"]);
GF(11)[t]
gap> P:=List([1,3,4,5,7],i->Z(11)^i);
[ Z(11), Z(11)^3, Z(11)^4, Z(11)^5, Z(11)^7 ]
gap> C:=GeneralizedReedSolomonCode(P,3,R);
a linear [5,3,1..3]2  generalized Reed-Solomon code over GF(11)
gap> MinimumDistance(C);
3
gap> c:=Random(C);
[ 10 7 7 5 3 ]
gap> v:=Codeword([ 10, 7, 7, 5, 7 ], GF(11));
[ 10 7 7 5 7 ]
gap> GeneralizedReedSolomonDecoderGao(C,v);
[ 10 7 7 5 3 ]
```

4.10.4 GeneralizedReedSolomonListDecoder

▷ `GeneralizedReedSolomonListDecoder(C, r, tau)` (function)

`GeneralizedReedSolomonListDecoder` implements Sudans list-decoding algorithm (see section 12.1 of [JH04]) for "low rate" Reed-Solomon codes. It returns the list of all codewords in C which are a distance of at most `tau` from r (a 'received word'). C must be a generalized Reed-Solomon code C (see `GeneralizedReedSolomonCode` (5.6.2)) and r must be a GUAVA codeword.

Example

```
gap> F:=GF(16);
GF(2^4)
gap> a:=PrimitiveRoot(F);; b:=a^7;; b^4+b^3+1;
0*Z(2)
gap> Pts:=List([0..14],i->b^i);
[ Z(2)^0, Z(2^4)^7, Z(2^4)^14, Z(2^4)^6, Z(2^4)^13, Z(2^2), Z(2^4)^12,
  Z(2^4)^4, Z(2^4)^11, Z(2^4)^3, Z(2^2)^2, Z(2^4)^2, Z(2^4)^9, Z(2^4),
  Z(2^4)^8 ]
gap> x:=X(F);;
gap> R1:=PolynomialRing(F,[x]);;
gap> vars:=IndeterminatesOfPolynomialRing(R1);;
gap> y:=X(F,vars);;
gap> R2:=PolynomialRing(F,[x,y]);;
gap> C:=GeneralizedReedSolomonCode(Pts,3,R1);
a linear [15,3,1..13]10..12  generalized Reed-Solomon code over GF(16)
```

```
gap> MinimumDistance(C); ## 6 error correcting
13
gap> z:=Zero(F);;
gap> r:=[z,z,z,z,z,z,z,z,b^6,b^2,b^5,b^14,b,b^7,b^11];;
gap> r:=Codeword(r);
[ 0 0 0 0 0 0 0 0 a^12 a^14 a^5 a^8 a^7 a^4 a^2 ]
gap> cs:=GeneralizedReedSolomonListDecoder(C,r,2);
[ [ 0 a^9 a^3 a^13 a^6 a^10 a^11 a a^12 a^14 a^5 a^8 a^7 a^4 a^2 ],
  [ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ] ]
gap> c1:=cs[1]; c1 in C;
[ 0 a^9 a^3 a^13 a^6 a^10 a^11 a a^12 a^14 a^5 a^8 a^7 a^4 a^2 ]
true
gap> c2:=cs[2]; c2 in C;
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
true
gap> WeightCodeword(c1-r);
7
gap> WeightCodeword(c2-r);
7
```

4.10.5 BitFlipDecoder

▷ `BitFlipDecoder(C, r)` (function)

The iterative decoding method `BitFlipDecoder` must only be applied to LDPC codes. For more information on LDPC codes, refer to Section 5.8. For these codes, `BitFlipDecoder` decodes very quickly. (Warning: it can give wildly wrong results for arbitrary binary linear codes.) The bit flipping algorithm is described for example in Chapter 13 of [JH04].

Example

```
gap> Reset(GlobalMersenneTwister, 1);; # make the example reproducible
gap> C:=HammingCode(4,GF(2));
a linear [15,11,3]1 Hamming (4,2) code over GF(2)
gap> c:=Random(C);
[ 0 0 0 1 1 1 0 1 0 0 0 0 0 0 1 ]
gap> v:=List(c);
[ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2),
  0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ]
gap> v[1]:=Z(2)+v[1]; # flip 1st bit of c to create an error
Z(2)^0
gap> v:=Codeword(v);
[ 1 0 0 1 1 1 0 1 0 0 0 0 0 0 1 ]
gap> BitFlipDecoder(C,v);
[ 0 0 0 1 1 1 0 1 0 0 0 0 0 0 1 ]
```

4.10.6 NearestNeighborGRSDecodewords

▷ `NearestNeighborGRSDecodewords(C, v, dist)` (function)

`NearestNeighborGRSDecodewords` finds all generalized Reed-Solomon codewords within distance *dist* from *v* *and* the associated polynomial, using "brute force". Input: *v* is a received vector

(a GUAVA codeword), `C` is a GRS code, `dist` > 0 is the distance from `v` to search in `C`. Output: a list of pairs $[c, f(x)]$, where $wt(c-v) \leq dist - 1$ and $c = (f(x_1), ..., f(x_n))$.

Example

```
gap> F:=GF(16);
GF(2^4)
gap> a:=PrimitiveRoot(F);; b:=a^7; b^4+b^3+1;
Z(2^4)^7
0*Z(2)
gap> Pts:=List([0..14],i->b^i);
[ Z(2)^0, Z(2^4)^7, Z(2^4)^14, Z(2^4)^6, Z(2^4)^13, Z(2^2), Z(2^4)^12,
  Z(2^4)^4, Z(2^4)^11, Z(2^4)^3, Z(2^2)^2, Z(2^4)^2, Z(2^4)^9, Z(2^4),
  Z(2^4)^8 ]
gap> x:=X(F,"x");;
gap> R1:=PolynomialRing(F,[x]);;
gap> vars:=IndeterminatesOfPolynomialRing(R1);;
gap> y:=X(F,vars);;
gap> R2:=PolynomialRing(F,[x,y]);;
gap> C:=GeneralizedReedSolomonCode(Pts,3,R1);
a linear [15,3,1..13]10..12  generalized Reed-Solomon code over GF(16)
gap> MinimumDistance(C); # 6 error correcting
13
gap> z:=Zero(F);
0*Z(2)
gap> r:=[z,z,z,z,z,z,z,z,b^6,b^2,b^5,b^14,b,b^7,b^11];; # 7 errors
gap> r:=Codeword(r);
[ 0 0 0 0 0 0 0 0 a^12 a^14 a^5 a^8 a^7 a^4 a^2 ]
gap> cs:=NearestNeighborGRSDecodewords(C,r,7);
[ [ [ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ], 0*Z(2) ],
  [ [ 0 a^9 a^3 a^13 a^6 a^10 a^11 a a^12 a^14 a^5 a^8 a^7 a^4 a^2 ],
      x+Z(2)^0 ] ]
```

4.10.7 NearestNeighborDecodewords

▷ `NearestNeighborDecodewords(C, v, dist)` (function)

`NearestNeighborDecodewords` finds all codewords in a linear code `C` within distance `dist` from `v`, using "brute force". Input: `v` is a received vector (a GUAVA codeword), `C` is a linear code, `dist` > 0 is the distance from `v` to search in `C`. Output: a list of $c \in C$, where $wt(c-v) \leq dist - 1$.

Example

```
gap> F:=GF(16);
GF(2^4)
gap> a:=PrimitiveRoot(F);; b:=a^7; b^4+b^3+1;
Z(2^4)^7
0*Z(2)
gap> Pts:=List([0..14],i->b^i);
[ Z(2)^0, Z(2^4)^7, Z(2^4)^14, Z(2^4)^6, Z(2^4)^13, Z(2^2), Z(2^4)^12,
  Z(2^4)^4, Z(2^4)^11, Z(2^4)^3, Z(2^2)^2, Z(2^4)^2, Z(2^4)^9, Z(2^4),
  Z(2^4)^8 ]
gap> x:=X(F,"x");;
gap> R1:=PolynomialRing(F,[x]);;
gap> vars:=IndeterminatesOfPolynomialRing(R1);;
```

```
gap> y:=X(F,vars);;
gap> R2:=PolynomialRing(F,[x,y]);;
gap> C:=GeneralizedReedSolomonCode(Pts,3,R1);
a linear [15,3,1..13]10..12  generalized Reed-Solomon code over GF(16)
gap> MinimumDistance(C);
13
gap> z:=Zero(F);
0*Z(2)
gap> r:=[z,z,z,z,z,z,z,z,b^6,b^2,b^5,b^14,b,b^7,b^11];;
gap> r:=Codeword(r);
[ 0 0 0 0 0 0 0 0 a^12 a^14 a^5 a^8 a^7 a^4 a^2 ]
gap> cs:=NearestNeighborDecodewords(C,r,7);
[ [ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ],
  [ 0 a^9 a^3 a^13 a^6 a^10 a^11 a a^12 a^14 a^5 a^8 a^7 a^4 a^2 ] ]
```

4.10.8 Syndrome

▷ `Syndrome(C, v)` (function)

`Syndrome` returns the syndrome of word v with respect to a linear code C. v is a codeword in the ambient vector space of C. If v 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 `SyndromeTable` (4.10.9)) that is needed to correct an error in v.

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

Example

```
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 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" ) ); # the same syndrome: both codewords are in the
[ 1 1 1 1 ]
```

4.10.9 SyndromeTable

▷ `SyndromeTable(C)` (function)

`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 v with `Syndrome` (see `Syndrome` (4.10.8)), the error vector needed to correct v can be found in the syndrome table. Subtracting this vector from v 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.

Example

```
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;  # c is not an element of H
false
gap> Syndrome(H,c); # according to the syndrome table, the error vector [ 0 1 0 ] belongs to this
[ 1 0 ]
gap> c - Codeword("010") in H; # so the corrected codeword is [ 1 0 1 ] - [ 0 1 0 ] = [ 1 1 1 ],
true
```

4.10.10 StandardArray

▷ `StandardArray(C)` (function)

`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`, $r = n - k$ 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 `Syndrome` (4.10.8)).

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.

Example

```
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 ] ] ]
```

4.10.11 PermutationDecode

▷ `PermutationDecode(C, v)` (function)

`PermutationDecode` performs permutation decoding when possible and returns original vector and prints 'fail' when not possible.

This uses `AutomorphismGroup` in the binary case, and (the slower) `PermutationAutomorphismGroup` otherwise, to compute the permutation automorphism group P of `C`. The algorithm runs through the elements p of P checking if the weight of $H(p \cdot v)$ is less than $(d-1)/2$. If it is then the vector $p \cdot v$ is used to decode v: assuming `C` is in standard form then $c = p^{-1}Em$ is the decoded word, where m is the information digits part of $p \cdot v$. If no such p exists then "fail" is returned. See, for example, section 10.2 of Huffman and Pless [HP03] for more details.

Example

```
gap> Reset(GlobalMersenneTwister, 1);; # make the example reproducible
gap> C0:=HammingCode(3,GF(2));
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> G0:=GeneratorMat(C0);;
gap> G := List(G0, ShallowCopy);;
gap> PutStandardForm(G);
()
gap> Display(G);
```

```
 1 . . . . 1 1
 . 1 . . 1 . 1
 . . 1 . 1 1 .
 . . . 1 1 1 1
gap> H0:=CheckMat(C0);;
gap> Display(H0);
 . . . 1 1 1 1
 . 1 1 . . 1 1
 1 . 1 . 1 . 1
gap> c0:=Random(C0);
[ 0 0 0 1 1 1 1 ]
gap> v01:=c0[1]+Z(2)^2;;
gap> v1:=List(c0, ShallowCopy);;
gap> v1[1]:=v01;;
gap> v1:=Codeword(v1);
[ 1 0 0 1 1 1 1 ]
gap> c1:=PermutationDecode(C0,v1);
[ 0 0 0 1 1 1 1 ]
gap> c1=c0;
true
```

4.10.12 PermutationDecodeNC

▷ `PermutationDecodeNC(`*C*`, `*v*`, `*P*`)` (function)

Same as `PermutationDecode` except that one may enter the permutation automorphism group *P* in as an argument, saving time. Here *P* is a subgroup of the symmetric group on *n* letters, where *n* is the word length of *C*.

Chapter 5

Generating Codes

In this chapter we describe functions for generating codes.

Section 5.1 describes functions for generating unrestricted codes.

Section 5.2 describes functions for generating linear codes.

Section 5.3 describes functions for constructing certain covering codes, such as the Gabidulin codes.

Section 5.4 describes functions for constructing the Golay codes.

Section 5.5 describes functions for generating cyclic codes.

Section 5.6 describes functions for generating codes as the image of an evaluation map applied to a space of functions. For example, generalized Reed-Solomon codes and toric codes are described there.

Section 5.7 describes functions for generating algebraic geometry codes.

Section 5.8 describes functions for constructing low-density parity-check (LDPC) codes.

5.1 Generating Unrestricted Codes

In this section we start with functions that creating code from user defined matrices or special matrices (see `ElementsCode` (5.1.1), `HadamardCode` (5.1.2), `ConferenceCode` (5.1.3) and `MOLSCode` (5.1.4)). These codes are unrestricted codes; they may later be discovered to be linear or cyclic.

The next functions generate random codes (see `RandomCode` (5.1.5)) and the Nordstrom-Robinson code (see `NordstromRobinsonCode` (5.1.6)), respectively.

Finally, we describe two functions for generating Greedy codes. These are codes that constructed by gathering codewords from a space (see `GreedyCode` (5.1.7) and `LexiCode` (5.1.8)).

5.1.1 ElementsCode

▷ `ElementsCode(`*`L[, name]`*`, F)` (function)

`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.

Example

```
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 ] ]
```

5.1.2 HadamardCode

▷ `HadamardCode(H[, t])` (function)

The four forms this command can take are `HadamardCode(H,t)`, `HadamardCode(H)`, `HadamardCode(n,t)`, and `HadamardCode(n)`.

In the case when the arguments `H` and `t` are both given, `HadamardCode` returns a Hadamard code of the tth kind from the Hadamard matrix `H` In case only `H` is given, $t = 3$ is used.

By definition, a Hadamard matrix is a square matrix `H` with $H \cdot H^T = -n \cdot I_n$, 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 1's by 0's and the -1's by 1s).

The Hadamard matrix of 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 kind* ($t = 3$) 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.

The command `HadamardCode(n,t)` returns a Hadamard code with parameter `n` of the tth kind. For the command `HadamardCode(n)`, $t = 3$ is used.

When called in these forms, `HadamardCode` first creates a Hadamard matrix (see `HadamardMat` (7.3.4)), 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.

Example

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

5.1.3 ConferenceCode

▷ `ConferenceCode(H)` (function)

`ConferenceCode` returns a code of length $n-1$ constructed from a symmetric 'conference matrix' H. A *conference matrix* H is a symmetric matrix of order n, which satisfies $H \cdot H^T = ((n-1) \cdot I$, with $n \equiv 2 \pmod 4$. The rows of $\frac{1}{2}(H+I+J)$, $\frac{1}{2}(-H+I+J)$, plus the zero and all-ones vectors form the elements of a binary non-linear $(n-1, 2n, (n-2)/2)$ code.

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

Example

```
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
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 ] ]
```

5.1.4 MOLSCode

▷ `MOLSCode([n, ]q)` (function)

`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 $q \times q$. The default for `n` is 4. GUAVA can construct a MOLS code for $n-2 \leq q$. Here `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 $q \times q$ 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 `IsMDSCode` (4.3.7)).

Example

```
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)
```

5.1.5 RandomCode

▷ `RandomCode(n, M, F)` (function)

`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 `RandomLinearCode` (5.2.12)).

Example

```
gap> C1 := RandomCode( 6, 10, GF(8) );
a (6,10,1..6)4..6 random unrestricted code over GF(8)
gap> MinimumDistance(C1);
2
gap> C2 := RandomCode( 6, 10, GF(8) );
a (6,10,1..6)4..6 random unrestricted code over GF(8)
gap> C1 = C2;
false
```

5.1.6 NordstromRobinsonCode

▷ `NordstromRobinsonCode()` (function)

`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.

Example

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

5.1.7 GreedyCode

▷ `GreedyCode(L, d, F)` (function)

`GreedyCode` returns a Greedy code with design distance `d` over the finite field `F`. The code is constructed using the greedy algorithm on the list of vectors `L`. (The greedy 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`.

Greedy codes are often linear codes.

The function `LexiCode` creates a greedy code from a basis instead of an enumerated list (see `LexiCode` (5.1.8)).

Example

```
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
```

5.1.8 LexiCode

▷ `LexiCode(n, d, F)` (function)

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`.

Another syntax which one can use is `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`.

Note that binary lexicodes are always linear.

Example

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

The function `GreedyCode` creates a greedy code that is not restricted to a lexicographical order (see `GreedyCode` (5.1.7)).

5.2 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 (`GeneratorMatCode` (5.2.1)) or check matrix (`CheckMatCode` (5.2.3)). All linear codes can be constructed with these functions.

The next functions we describe generate some well-known codes, like Hamming codes (`HammingCode` (5.2.4)), Reed-Muller codes (`ReedMullerCode` (5.2.5)) and the extended Golay codes (`ExtendedBinaryGolayCode` (5.4.2) and `ExtendedTernaryGolayCode` (5.4.4)).

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 `AlternantCode` (5.2.6), `GoppaCode` (5.2.7), `GeneralizedSrivastavaCode` (5.2.8) and `SrivastavaCode` (5.2.9)).

Finally, we describe a function for generating random linear codes (see `RandomLinearCode` (5.2.12)).

5.2.1 GeneratorMatCode

▷ `GeneratorMatCode(G[, name], F)` (function)

GeneratorMatCode returns a linear code with generator matrix *G*. *G* must be a matrix over finite 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 $k \times n$-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 GAP function BaseMat and results in an equal code. The generator matrix can be retrieved with the function GeneratorMat (see GeneratorMat (4.7.1)).

Example

```
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 ) ); # This is the smallest linear code that contains the N-R
a linear [16,11,1..4]2 code defined by generator matrix over GF(2)
```

5.2.2 CheckMatCodeMutable

▷ CheckMatCodeMutable(*H[, name], F*) (function)

CheckMatCodeMutable is the same as CheckMatCode except that the check matrix and generator matrix are mutable.

5.2.3 CheckMatCode

▷ CheckMatCode(*H[, name], F*) (function)

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, $c \cdot H^T = 0$ where c is an element of the code. If *H* is a $r \times n$-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 GAP function BaseMat. and results in an equal code. The check matrix can be retrieved with the function CheckMat (see CheckMat (4.7.2)).

Example

```
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)
```

5.2.4 HammingCode

▷ `HammingCode(r, F)` (function)

`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 `Decode` (4.10.1)) 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.

Example

```
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)
```

5.2.5 ReedMullerCode

▷ `ReedMullerCode(r, k)` (function)

`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} (see for example, section 1.10 in [HP03]). By definition, the rth order binary Reed-Muller code of length $n=2^m$, for $0 \leq r \leq m$, is the set of all vectors f, where f is a Boolean function which is a polynomial of degree at most r.

Example

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

See `GeneralizedReedMullerCode` (5.6.3) for a more general construction.

5.2.6 AlternantCode

▷ `AlternantCode(r, Y[, alpha], F)` (function)

`AlternantCode` returns an 'alternant code', with parameters `r`, `Y` and `alpha` (optional). `F` denotes the (finite) base field. Here, `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 the form $H=([a_i^j y_i])$, where $0 \leq j \leq r-1$, $1 \leq i \leq n$, and where $[...]$ is as in `VerticalConversionFieldMat` (7.3.9)). If no `alpha` is specified, the vector $[1,a,a^2,..,a^{n-1}]$ is used, where a is a primitive element of a Galois field `F`.

Example

```
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)
```

5.2.7 GoppaCode

▷ `GoppaCode(G, L)` (function)

`GoppaCode` returns a Goppa code `C` from Goppa polynomial g, having coefficients in a Galois Field $GF(q)$. `L` must be a list of elements in $GF(q)$, that are not roots of g. The word length of the code is equal to the length of `L`. The parity check matrix has the form $H = ([a_i^j/G(a_i)])_{0 \le j \le deg(g)-1,\ a_i \in L}$, where $a_i \in L$ and $[...]$ is as in `VerticalConversionFieldMat` (7.3.9), so H has entries in $GF(q)$, $q = p^m$. It is known that $d(C) \ge deg(g)+1$, with a better bound in the binary case provided g has no multiple roots. See Huffman and Pless [HP03] section 13.2.2, and MacWilliams and Sloane [MS83] section 12.3, for more details.

One can also call `GoppaCode` using the syntax `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.

This is a special case of an alternant code.

Example

```
gap> x:=Indeterminate(GF(8),"x");
x
gap> L:=Elements(GF(8));
[ 0*Z(2), Z(2)^0, Z(2^3), Z(2^3)^2, Z(2^3)^3, Z(2^3)^4, Z(2^3)^5, Z(2^3)^6 ]
gap> g:=x^2+x+1;
x^2+x+Z(2)^0
gap> C:=GoppaCode(g,L);
a linear [8,2,5]3 classical Goppa code over GF(2)
gap> xx := Indeterminate( GF(2), "xx" );;
gap> gg := xx^2 + xx + 1;; L := AsSSortedList( GF(8) );;
gap> C1 := GoppaCode( gg, L );
a linear [8,2,5]3 classical Goppa code over GF(2)
gap> y := Indeterminate( GF(2), "y" );;
gap> h := y^2 + y + 1;;
gap> C2 := GoppaCode( h, 8 );
a linear [8,2,5]3 classical Goppa code over GF(2)
gap> C1=C2;
true
gap> C=C1;
true
```

5.2.8 GeneralizedSrivastavaCode

▷ `GeneralizedSrivastavaCode(a, w, z[, t], F)` (function)

`GeneralizedSrivastavaCode` returns a generalized Srivastava code with parameters `a`, `w`, `z`, `t`. $a = \{a_1,...,a_n\}$ and $w = \{w_1,...,w_s\}$ 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: $d \ge st+1$. The check matrix of this code is the form

$$H = ([\frac{z_i}{(a_i - w_j)^k}]),$$

$1 \le k \le t$, where $[...]$ is as in `VerticalConversionFieldMat` (7.3.9). We use this definition of H to define the code. The default for `t` is 1. The original Srivastava codes (see `SrivastavaCode` (5.2.9)) are a special case $t = 1$, $z_i = a_i^\mu$, for some μ.

Example

```
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)
```

5.2.9 SrivastavaCode

▷ `SrivastavaCode(a, w[, mu], F)` (function)

SrivastavaCode returns a Srivastava code with parameters `a`, `w` (and optionally `mu`). $a = \{a_1,...,a_n\}$ and $w = \{w_1,...,w_s\}$ 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, in which $z_i = a_i^{mu}$ for some `mu` and $t = 1$.

J. N. Srivastava introduced this code in 1967, though his work was not published. See Helgert [Hel72] for more details on the properties of this code. Related reference: G. Roelofsen, ON GOPPA AND GENERALIZED SRIVASTAVA CODES PhD thesis, Dept. Math. and Comp. Sci., Eindhoven Univ. of Technology, the Netherlands, 1982.

Example

```
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 );     # Always true if F is a prime field
true
```

5.2.10 CordaroWagnerCode

▷ `CordaroWagnerCode(n)` (function)

`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 `RepetitionCode` (5.5.13)).

Example

```
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 ] ]
```

5.2.11 FerreroDesignCode

▷ `FerreroDesignCode(P, m)` (function)

Requires the GAP package SONATA

A group K together with a group of automorphism H of K such that the semidirect product KH is a Frobenius group with complement H is called a Ferrero pair (K,H) in SONATA. Take a Frobenius

$(G,+)$ group with kernel K and complement H. Consider the design D with point set K and block set $\{a^H+b \mid a,b \in K, a \neq 0\}$. Here a^H denotes the orbit of a under conjugation by elements of H. Every planar near-ring design of type "*" can be obtained in this way from groups. These designs (from a Frobenius kernel of order v and a Frobenius complement of order k) have $v(v-1)/k$ distinct blocks and they are all of size k. Moreover each of the v points occurs in exactly $v-1$ distinct blocks. Hence the rows and the columns of the incidence matrix M of the design are always of constant weight.

`FerreroDesignCode` constructs binary linear code arising from the incidence matrix of a design associated to a "Ferrero pair" arising from a fixed-point-free (fpf) automorphism groups and Frobenius group.

INPUT: P is a list of prime powers describing an abelian group G. $m > 0$ is an integer such that G admits a cyclic fpf automorphism group of size m. This means that for all $q = p^k \in P$, OrderMod(p, m) must divide q (see the SONATA documentation for `FpfAutomorphismGroupsCyclic`).

OUTPUT: The binary linear code whose generator matrix is the incidence matrix of a design associated to a "Ferrero pair" arising from the fixed-point-free (fpf) automorphism group of G. The pair (H,K) is called a Ferraro pair and the semidirect product KH is a Frobenius group with complement H.

AUTHORS: Peter Mayr and David Joyner

Example

```
gap> G:=AbelianGroup([5,5] );
<pc group of size 25 with 2 generators>
gap> FpfAutomorphismGroupsMaxSize( G );
[ 24, 2 ]
gap> L:=FpfAutomorphismGroupsCyclic( [5,5], 3 );
[ [ [ f1, f2 ] -> [ f1*f2^2, f1*f2^3 ] ],
  <pc group of size 25 with 2 generators> ]
gap> D := DesignFromFerreroPair( L[2], Group(L[1][1]), "*" );
<a 2 - ( 25, 3, 2 ) nearring generated design>
gap> M:=IncidenceMat( D );; Length(M); Length(TransposedMat(M));
25
200
gap> C1:=GeneratorMatCode(M*Z(2),GF(2));
a linear [200,25,1..24]62..100 code defined by generator matrix over GF(2)
gap> MinimumDistance(C1);
24
gap> C2:=FerreroDesignCode( [5,5],3);
a linear [200,25,1..24]62..100 code defined by generator matrix over GF(2)
gap> C1=C2;
true
```

5.2.12 RandomLinearCode

▷ `RandomLinearCode(n, k, F)` (function)

`RandomLinearCode` returns a random linear code with word length `n`, dimension `k` over field `F`. The method used is to first construct a $k \times n$ matrix of the block form (I,A), where I is a $k \times k$ identity matrix and A is a $k \times (n-k)$ matrix constructed using `Random(F)` repeatedly. Then the columns are permuted using a randomly selected element of `SymmetricGroup(n)`.

To create a random unrestricted code, use `RandomCode` (see `RandomCode` (5.1.5)).

Example

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

The method GUAVA chooses to output the result of a `RandomLinearCode` command is different than other codes. For example, the bounds on the minimum distance is not displayed. However, you can use the `Display` command to print this information. This new display method was added in version 1.9 to speed up the command (if n is about 80 and k about 40, for example, the time it took to look up and/or calculate the bounds on the minimum distance was too long).

5.2.13 OptimalityCode

▷ `OptimalityCode(`*C*`)` (function)

`OptimalityCode` returns the difference between the smallest known upper bound and the actual size of the code. Note that the value of the function `UpperBound` is not always equal to the actual upper bound $A(n,d)$ thus the result may not be equal to 0 even if the code is optimal!

`OptimalityLinearCode` is similar but applies only to linear codes.

5.2.14 BestKnownLinearCode

▷ `BestKnownLinearCode(`*n*`, `*k*`, `*F*`)` (function)

`BestKnownLinearCode` returns the best known (as of 11 May 2006) linear code of length *n*, dimension *k* over field *F*. The function uses the tables described in section `BoundsMinimumDistance` (7.1.13) to construct this code.

This command can also be called using the syntax `BestKnownLinearCode( rec )`, where *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 `BoundsMinimumDistance` (7.1.13)), if the bounds are already calculated.

Example

```
gap> C1 := BestKnownLinearCode( 23, 12, GF(2) );
a linear [23,12,7]3 punctured code
gap> C1 = BinaryGolayCode();  # it's constructed differently
false
gap> C1 := BestKnownLinearCode( 23, 12, GF(2) );
a linear [23,12,7]3 punctured code
gap> G1 := MutableCopyMat(GeneratorMat(C1));;
gap> PutStandardForm(G1);
()
gap> Display(G1);
 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 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> C2 := BinaryGolayCode();
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> G2 := MutableCopyMat(GeneratorMat(C2));;
gap> PutStandardForm(G2);
()
gap> Display(G2); ## Despite their generator matrices are different, they are equivalent codes, s
 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 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> IsEquivalent(C1,C2);
true
gap> CodeIsomorphism(C1,C2);
(4,14,6,12,5)(7,17,18,11,19)(8,22,13,21,16)(10,23,15,20)
gap> Display( BestKnownLinearCode( 81, 77, GF(4) ) );
a linear [81,77,3]2..3 shortened code of
a linear [85,81,3]1 Hamming (4,4) code over GF(4)
gap> C:=BestKnownLinearCode(174,72);
a linear [174,72,31..36]26..87 code defined by generator matrix over GF(2)
gap> bounds := BoundsMinimumDistance( 81, 77, GF(4) );
rec(
  construction :=
    [ <Operation "ShortenedCode">,
      [ [ <Operation "HammingCode">, [ 4, 4 ] ], [ 1, 2, 3, 4 ] ] ], k := 77,
  lowerBound := 3,
  lowerBoundExplanation := [ "Lb(81,77)=3, by shortening of:",
      "Lb(85,81)=3, reference: Ham" ], n := 81, q := 4,
  references :=
    rec(
      Ham := [ "%T this reference is unknown, for more info",
          "%T contact A.E. Brouwer (aeb@cwi.nl)" ],
      cap := [ "%T this reference is unknown, for more info",
          "%T contact A.E. Brouwer (aeb@cwi.nl)" ] ), upperBound := 3,
  upperBoundExplanation := [ "Ub(81,77)=3, by considering shortening to:",
      "Ub(18,14)=3, reference: cap" ] )
gap> C := BestKnownLinearCode( bounds );
a linear [81,77,3]2..3 shortened code
```

```
gap> C = BestKnownLinearCode(81, 77, GF(4) );
true
```

5.3 Gabidulin Codes

These five binary, linear codes are derived from an article by Gabidulin, Davydov and Tombak [GDT91]. All these 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.

Like the Hamming codes, these codes have fixed minimum distance and covering radius, but can be arbitrarily long.

5.3.1 GabidulinCode

▷ `GabidulinCode(m, w1, w2)` (function)

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

5.3.2 EnlargedGabidulinCode

▷ `EnlargedGabidulinCode(m, w1, w2, e)` (function)

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

5.3.3 DavydovCode

▷ `DavydovCode(r, v, ei, ej)` (function)

`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.

5.3.4 TombakCode

▷ `TombakCode(m, e, beta, gamma, w1, w2)` (function)

`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 $GF(2^{m-3})$.

5.3.5 EnlargedTombakCode

▷ `EnlargedTombakCode(m, e, beta, gamma, w1, w2, u)` (function)

`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 $GF(2^{m-1})$.

Example

```
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)
```

5.4 Golay Codes

" The Golay code is probably the most important of all codes for both practical and theoretical reasons. " ([MS83], pg. 64). Though born in Switzerland, M. J. E. Golay (1902-1989) worked for the US Army Labs for most of his career. For more information on his life, see his obit in the June 1990 IEEE Information Society Newsletter.

5.4.1 BinaryGolayCode

▷ `BinaryGolayCode()` (function)

`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 `ExtendedBinaryGolayCode` (5.4.2)). There's also the ternary Golay code (see `TernaryGolayCode` (5.4.3)).

Example

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

5.4.2 ExtendedBinaryGolayCode

▷ `ExtendedBinaryGolayCode()` (function)

`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 `BinaryGolayCode` (5.4.1)). The code is self-dual.

Example

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

5.4.3 TernaryGolayCode

▷ `TernaryGolayCode()` (function)

`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 `ExtendedTernaryGolayCode` (5.4.4)). There's also the binary Golay code (see `BinaryGolayCode` (5.4.1)).

Example

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

5.4.4 ExtendedTernaryGolayCode

▷ `ExtendedTernaryGolayCode()` (function)

`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 `TernaryGolayCode` (5.4.3)). The code is self-dual.

Example

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

5.5 Generating Cyclic Codes

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

$$c(x)h(x) \equiv 0 \pmod{x^n - 1},$$

for every codeword $c(x) \in C$.

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

Two of the Golay codes already described are cyclic (see `BinaryGolayCode` (5.4.1) and `TernaryGolayCode` (5.4.3)). For example, the GUAVA record for a binary Golay code contains the generator polynomial:

Example

```
gap> C := BinaryGolayCode();
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> Set(NamesOfComponents(C));
[ "Dimension", "GeneratorMat", "GeneratorPol",
  "GeneratorsOfLeftOperatorAdditiveGroup", "LeftActingDomain",
  "MinimumWeightOfGenerators", "Redundancy", "Size", "SpecialDecoder",
  "UpperBoundOptimalMinimumDistance", "WeightDistribution", "WordLength",
  "boundsCoveringRadius", "lowerBoundMinimumDistance", "name",
  "upperBoundMinimumDistance" ]
gap> C!.GeneratorPol;
x^11+x^10+x^6+x^5+x^4+x^2+Z(2)^0
```

Then functions that generate cyclic codes from a prescribed set of roots of the generator polynomial are described, including the BCH codes (see `RootsCode` (5.5.3), `BCHCode` (5.5.4), `ReedSolomonCode` (5.5.5) and `QRCode` (5.5.7)).

Finally we describe the trivial codes (see `WholeSpaceCode` (5.5.11), `NullCode` (5.5.12), `RepetitionCode` (5.5.13)), and the command `CyclicCodes` which lists all cyclic codes (`CyclicCodes` (5.5.14)).

5.5.1 GeneratorPolCode

▷ `GeneratorPolCode(`*g, n[, name], F*`)` (function)

`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 `Generating Cyclic Codes` (5.5).

Example

```
gap> x:= Indeterminate( GF(2), "x" );; P:= x^2+1;
x^2+Z(2)^0
```

```
gap> C1 := GeneratorPolCode(P, 7, GF(2));
a cyclic [7,6,1..2]1 code defined by generator polynomial over GF(2)
gap> GeneratorPol( C1 );
x+Z(2)^0
gap> C2 := GeneratorPolCode( x+1, 7, GF(2));
a cyclic [7,6,1..2]1 code defined by generator polynomial over GF(2)
gap> GeneratorPol( C2 );
x+Z(2)^0
```

5.5.2 CheckPolCode

▷ `CheckPolCode(h, n[, name], F)` (function)

`CheckPolCode` creates a cyclic code with a check polynomial `h`, word length `n`, over `F`. `name` can contain a short description of the code (as a string).

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 5.5.

Example

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

5.5.3 RootsCode

▷ `RootsCode(n, list)` (function)

This is the generalization of the BCH, Reed-Solomon and quadratic residue codes (see `BCHCode` (5.5.4), `ReedSolomonCode` (5.5.5) and `QRCode` (5.5.7)). 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 nth 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.

This command can also be called with the syntax `RootsCode( n, list, q )`. 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 $GF(q)$. GUAVA calculates a primitive n^{th} root of unity, α, in the extension field of $GF(q)$. It uses the set of the powers of α in the list as a prescribed set of zeros.

Example

```
gap> a := PrimitiveUnityRoot( 3, 14 );
Z(3^6)^52
gap> C1 := RootsCode( 14, [ a^0, a, a^3 ] );
a cyclic [14,7,4..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
gap> C3 := RootsCode( 4, [ 1, 2 ], 5 );
a cyclic [4,2,3]2 code defined by roots over GF(5)
gap> RootsOfCode( C3 );
[ Z(5), Z(5)^2 ]
gap> C3 = ReedSolomonCode( 4, 3 );
true
```

5.5.4 BCHCode

▷ `BCHCode(n[, b], delta, F)` (function)

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

$$a^b, a^{b+1}, ..., a^{b+delta-2},$$

where a is a primitive n^{th} root of unity in the splitting field $GF(q^m)$, `b` is an integer $0 \leq b \leq n-delta+1$ and m is the multiplicative order of q modulo `n`. (The integers $\{b, ..., b+delta-2\}$ typically lie in the range $\{1, ..., n-1\}$.) Default value for `b` is 1, though the algorithm allows $b = 0$. The length `n` of the code and the size q of the field must be relatively prime. The generator polynomial is equal to the least common multiple of the minimal polynomials of

$$a^b, a^{b+1}, ..., a^{b+delta-2}.$$

The set of zeroes of the generator polynomial is equal to the union of the sets

$$\{a^x \mid x \in C_k\},$$

where C_k is the kth cyclotomic coset of q modulo n and $b \leq k \leq b+delta-2$ (see `CyclotomicCosets` (7.5.12)).

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 d for which the BCH code with designed distance d coincides with the BCH code with designed distance `delta`. This distance d 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) d, and the starting power b.

The Sugiyama decoding algorithm has been implemented for this code (see `Decode` (4.10.1)).

Example

```
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
```

See RootsCode (5.5.3) for a more general construction.

5.5.5 ReedSolomonCode

▷ ReedSolomonCode(*n*, *d*) (function)

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 UpperBoundSingleton (7.1.1)) 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 IsMDSCode (4.3.7)).

Example

```
gap> C1 := ReedSolomonCode( 3, 2 );
a cyclic [3,2,2]1 Reed-Solomon code over GF(4)
gap> IsCyclicCode(C1);
true
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
```

See GeneralizedReedSolomonCode (5.6.2) for a more general construction.

5.5.6 ExtendedReedSolomonCode

▷ ExtendedReedSolomonCode(*n*, *d*) (function)

ExtendedReedSolomonCode creates a Reed-Solomon code of length $n-1$ with designed distance $d-1$ and then returns the code which is extended by adding an overall parity-check symbol. The motivation for creating this function is calling ExtendedCode (6.1.1) function over a Reed-Solomon code will take considerably long time.

Example

```
gap> C := ExtendedReedSolomonCode(17, 13);
a linear [17,5,13]9..12 extended Reed Solomon code over GF(17)
gap> IsMDSCode(C);
true
```

5.5.7 QRCode

▷ QRCode(n, F) (function)

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 $x^2 \equiv q \pmod{n}$. 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.

Example

```
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> IsCyclicCode(C1);
true
gap> IsCyclicCode(HammingCode( 3, GF(2) ));
false
gap> C2 := QRCode( 11, GF(3) );
a cyclic [11,6,4..5]2 quadratic residue code over GF(3)
gap> C2 = TernaryGolayCode();
true
gap> Q1 := QRCode( 7, GF(2));
a cyclic [7,4,3]1 quadratic residue code over GF(2)
gap> P1:=AutomorphismGroup(Q1); IdGroup(P1);
Group([ (1,2)(5,7), (2,3)(4,7), (2,4)(5,6), (3,5)(6,7), (3,7)(5,6) ])
[ 168, 42 ]
```

5.5.8 QQRCodeNC

▷ QQRCodeNC(p) (function)

QQRCodeNC is the same as QQRCode, except that it uses GeneratorMatCodeNC instead of GeneratorMatCode.

5.5.9 QQRCode

▷ QQRCode(p) (function)

QQRCode returns a quasi-quadratic residue code, as defined by Proposition 2.2 in Bazzi-Mittel [BM06]. The parameter p must be a prime. Its generator matrix has the block form $G = (Q,N)$. Here Q is a $p \times p$ circulant matrix whose top row is $(0,x_1,...,x_{p-1})$, where $x_i = 1$ if and only if i is a quadratic residue mod p, and N is a $p \times p$ circulant matrix whose top row is $(0,y_1,...,y_{p-1})$, where $x_i + y_i = 1$ for all i. (In fact, this matrix can be recovered as the component DoublyCirculant of the code.)

Example

```
gap> C1 := QQRCode( 7);
a linear [14,7,1..4]3..5 code defined by generator matrix over GF(2)
gap> G1:=GeneratorMat(C1);;
gap> Display(G1);
 . 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> Display(C1!.DoublyCirculant);
 . 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> MinimumDistance(C1);
4
gap> C2 := QQRCode( 29); MinimumDistance(C2);
a linear [58,28,1..14]8..29 code defined by generator matrix over GF(2)
12
gap> Aut2:=AutomorphismGroup(C2); IdGroup(Aut2);
<permutation group of size 812 with 4 generators>
[ 812, 7 ]
```

5.5.10 FireCode

▷ `FireCode(`*g*`, `*b*`)` (function)

`FireCode` constructs a (binary) Fire code. *g* is a primitive polynomial of degree *m*, and a 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 of the code is defined as the product of *g* and $x^{2b-1}+1$.

Here is the general definition of 'Fire code', named after P. Fire, who introduced these codes in 1959 in order to correct burst errors. First, a definition. If $F = GF(q)$ and $f \in F[x]$ then we say f has *order* e if $f(x)|(x^e - 1)$. A *Fire code* is a cyclic code over F with generator polynomial $g(x) = (x^{2t-1} - 1)p(x)$, where $p(x)$ does not divide $x^{2t-1} - 1$ and satisfies $deg(p(x)) \geq t$. The length of such a code is the order of $g(x)$. Non-binary Fire codes have not been implemented.

.

Example

```
gap> x:= Indeterminate( GF(2), "x" );; G:= x^3+x^2+1;
x^3+x^2+Z(2)^0
gap> Factors( G );
[ x^3+x^2+Z(2)^0 ]
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 );     # Still it can correct bursts of length 3
4
```

5.5.11 WholeSpaceCode

▷ `WholeSpaceCode(n, F)` (function)

`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.

Example

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

5.5.12 NullCode

▷ `NullCode(n, F)` (function)

`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!

Example

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

5.5.13 RepetitionCode

▷ `RepetitionCode(n, F)` (function)

`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.

Example

```
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
```

5.5.14 CyclicCodes

▷ `CyclicCodes(n, F)` (function)

`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.

Example

```
gap> CyclicCodes(3,GF(3));
[ a cyclic [3,3,1]0 enumerated code over GF(3),
```

```
  a cyclic [3,2,1..2]1 enumerated code over GF(3),
  a cyclic [3,1,3]2 enumerated code over GF(3),
  a cyclic [3,0,3]3 enumerated code over GF(3) ]
```

5.5.15 NrCyclicCodes

▷ `NrCyclicCodes(n, F)` (function)

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

Example

```
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;     # This code is not cyclic
false
```

5.5.16 QuasiCyclicCode

▷ `QuasiCyclicCode(G, s, F)` (function)

`QuasiCyclicCode( G, k, F )` generates a rate $1/m$ quasi-cyclic code over field F. The input G is a list of univariate polynomials and m is the cardinality of this list. Note that m must be at least 2. The input s is the size of each circulant and it may not necessarily be the same as the code dimension k, i.e. $k \leq s$.

There also exists another version, `QuasiCyclicCode( G, s )` which produces quasi-cyclic codes over F_2 only. Here the parameter s holds the same definition and the input G is a list of integers, where each integer is an octal representation of a binary univariate polynomial.

Example

```
gap> #
gap> # This example show the case for k = s
gap> #
gap> L1 := PolyCodeword( Codeword("10000000000", GF(4)) );
Z(2)^0
gap> L2 := PolyCodeword( Codeword("12223201000", GF(4)) );
x^7+Z(2^2)*x^5+Z(2^2)^2*x^4+Z(2^2)*x^3+Z(2^2)*x^2+Z(2^2)*x+Z(2)^0
gap> L3 := PolyCodeword( Codeword("31111220110", GF(4)) );
x^9+x^8+Z(2^2)*x^6+Z(2^2)*x^5+x^4+x^3+x^2+x+Z(2^2)^2
```

```
gap> L4 := PolyCodeword( Codeword("13320333010", GF(4)) );
x^9+Z(2^2)^2*x^7+Z(2^2)^2*x^6+Z(2^2)^2*x^5+Z(2^2)*x^3+Z(2^2)^2*x^2+Z(2^2)^2*x+\
Z(2)^0
gap> L5 := PolyCodeword( Codeword("20102211100", GF(4)) );
x^8+x^7+x^6+Z(2^2)*x^5+Z(2^2)*x^4+x^2+Z(2^2)
gap> C := QuasiCyclicCode( [L1, L2, L3, L4, L5], 11, GF(4) );
a linear [55,11,1..32]24..41 quasi-cyclic code over GF(4)
gap> MinimumDistance(C);
29
gap> Display(C);
a linear [55,11,29]24..41 quasi-cyclic code over GF(4)
gap> #
gap> # This example show the case for k < s
gap> #
gap> L1 := PolyCodeword( Codeword("02212201220120211002000",GF(3)) );
-x^19+x^16+x^15-x^14-x^12+x^11-x^9-x^8+x^7-x^5-x^4+x^3-x^2-x
gap> L2 := PolyCodeword( Codeword("00221100200120220001110",GF(3)) );
x^21+x^20+x^19-x^15-x^14-x^12+x^11-x^8+x^5+x^4-x^3-x^2
gap> L3 := PolyCodeword( Codeword("22021011202221111020021",GF(3)) );
x^22-x^21-x^18+x^16+x^15+x^14+x^13-x^12-x^11-x^10-x^8+x^7+x^6+x^4-x^3-x-Z(3)^0
gap> C := QuasiCyclicCode( [L1, L2, L3], 23, GF(3) );
a linear [69,12,1..36]27..46 quasi-cyclic code over GF(3)
gap> MinimumDistance(C);
34
gap> Display(C);
a linear [69,12,34]27..46 quasi-cyclic code over GF(3)
gap> #
gap> # This example show the binary case using octal representation
gap> #
gap> L1 := 001;;   # 0 000 001
gap> L2 := 013;;   # 0 001 011
gap> L3 := 015;;   # 0 001 101
gap> L4 := 077;;   # 0 111 111
gap> C := QuasiCyclicCode( [L1, L2, L3, L4], 7 );
a linear [28,7,1..12]8..14 quasi-cyclic code over GF(2)
gap> MinimumDistance(C);
12
gap> Display(C);
a linear [28,7,12]8..14 quasi-cyclic code over GF(2)
```

5.5.17 CyclicMDSCode

▷ `CyclicMDSCode(q, m, k)` (function)

Given the input parameters q, m and k, this function returns a $[q^m+1, k, q^m-k+2]$ cyclic MDS code over GF(q^m). If q^m is even, any value of k can be used, otherwise only odd value of k is accepted.

Example

```
gap> C:=CyclicMDSCode(2,6,24);
a cyclic [65,24,42]31..41 MDS code over GF(64)
gap> IsMDSCode(C);
true
```

```
gap> C:=CyclicMDSCode(5,3,77);
a cyclic [126,77,50]35..49 MDS code over GF(125)
gap> IsMDSCode(C);
true
gap> C:=CyclicMDSCode(3,3,25);
a cyclic [28,25,4]2..3 MDS code over GF(27)
gap> GeneratorPol(C);
x^3+Z(3^3)^7*x^2+Z(3^3)^20*x-Z(3)^0
```

5.5.18 FourNegacirculantSelfDualCode

▷ `FourNegacirculantSelfDualCode(ax, bx, k)` (function)

A four-negacirculant self-dual code has a generator matrix G of the following form

```
    -                     -
    |         |  A  |  B  |
G = |  I_2k   |-----+-----|
    |         | -B^T| A^T |
    -                     -
```

where $AA^T + BB^T = -I_k$ and A, B and their transposed are all $k \times k$ negacirculant matrices. The generator matrix G returns a $[2k,k,d]_q$ self-dual code over GF(q). For discussion on four-negacirculant self-dual codes, refer to [HHKK07].

The input parameters `ax` and `bx` are the defining polynomials over GF(q) of negacirculant matrices A and B respectively. The last parameter k is the dimension of the code.

Example

```
gap> ax:=PolyCodeword(Codeword("1200200", GF(3)));
-x^4-x+Z(3)^0
gap> bx:=PolyCodeword(Codeword("2020221", GF(3)));
x^6-x^5-x^4-x^2-Z(3)^0
gap> C:=FourNegacirculantSelfDualCode(ax, bx, 14);;
gap> MinimumDistance(C);;
gap> CoveringRadius(C);;
gap> IsSelfDualCode(C);
true
gap> Display(C);
a linear [28,14,9]7 four-negacirculant self-dual code over GF(3)
gap> Display( GeneratorMat(C) );
 1 . . . . . . . . . . . . . 1 2 . . 2 . . 2 . 2 . 2 2 1
 . 1 . . . . . . . . . . . . . 1 2 . . 2 . 2 2 . 2 . 2 2
 . . 1 . . . . . . . . . . . . . 1 2 . . 2 1 2 2 . 2 . 2
 . . . 1 . . . . . . . . . . . 1 . . 1 2 . . 1 1 2 2 . 2 .
 . . . . 1 . . . . . . . . . . . 1 . . 1 2 . . 1 1 2 2 . 2
 . . . . . 1 . . . . . . . . . . . 1 . . 1 2 1 . 1 1 2 2 .
 . . . . . 1 . . . . . . . . . 1 . . 1 . . 1 . 1 . 1 1 2 2
 . . . . . 1 . . . . . . . 1 1 2 2 . 2 . 1 . . 1 . . 1
 . . . . . 1 . . . . . . . 1 1 2 2 . 2 2 1 . . 1 . .
 . . . . . 1 . . . . 1 . 1 1 2 2 . . 2 1 . . 1 .
```

```
  . . . . . . . . . . 1 . . . . 1 . 1 1 2 2 . . 2 1 . . 1
  . . . . . . . . . . . 1 . . 1 . 1 . 1 1 2 2 . . 2 1 . .
  . . . . . . . . . . . . 1 . 1 1 . 1 . 1 1 . 2 . . 2 1 .
  . . . . . . . . . . . . . 1 2 1 1 . 1 . 1 . . 2 . . 2 1
gap> ax:=PolyCodeword(Codeword("013131000", GF(7)));
x_1^5+Z(7)*x_1^4+x_1^3+Z(7)*x_1^2+x_1
gap> bx:=PolyCodeword(Codeword("425435030", GF(7)));
Z(7)*x_1^7+Z(7)^5*x_1^5+Z(7)*x_1^4+Z(7)^4*x_1^3+Z(7)^5*x_1^2+Z(7)^2*x_1+Z(7)^4
gap> C:=FourNegacirculantSelfDualCodeNC(ax, bx, 18);
a linear [36,18,1..13]0..36 four-negacirculant self-dual code over GF(7)
gap> IsSelfDualCode(C);
true
```

5.5.19 FourNegacirculantSelfDualCodeNC

▷ `FourNegacirculantSelfDualCodeNC(ax, bx, k)` (function)

This function is the same as `FourNegacirculantSelfDualCode`, except this version is faster as it does not estimate the minimum distance and covering radius of the code.

5.6 Evaluation Codes

5.6.1 EvaluationCode

▷ `EvaluationCode(P, L, R)` (function)

Input: F is a finite field, L is a list of rational functions in $R = F[x_1,...,x_r]$, P is a list of n points in F^r at which all of the functions in L are defined.
Output: The 'evaluation code' C, which is the image of the evaluation map

$$Eval_P : span(L) \to F^n,$$

given by $f \longmapsto (f(p_1),...,f(p_n))$, where $P = \{p_1,...,p_n\}$ and $f \in L$. The generator matrix of C is $G = (f_i(p_j))_{f_i \in L, p_j \in P}$.

This command returns a "record" object `C` with several extra components (type `NamesOfComponents(C)` to see them all): `C!.EvaluationMat` (not the same as the generator matrix in general), `C!.points` (namely P), `C!.basis` (namely L), and `C!.ring` (namely R).

Example

```
gap> F:=GF(11);;
gap> R := PolynomialRing(F,2);;
gap> indets := IndeterminatesOfPolynomialRing(R);;
gap> x:=indets[1];; y:=indets[2];;
gap> L:=[x^2*y,x*y,x^5,x^4,x^3,x^2,x,x^0];;
gap> Pts:=[ [ Z(11)^9, Z(11) ], [ Z(11)^8, Z(11) ], [ Z(11)^7, 0*Z(11) ],
   [ Z(11)^6, 0*Z(11) ], [ Z(11)^5, 0*Z(11) ], [ Z(11)^4, 0*Z(11) ],
   [ Z(11)^3, Z(11) ], [ Z(11)^2, 0*Z(11) ], [ Z(11), 0*Z(11) ],
   [ Z(11)^0, 0*Z(11) ], [ 0*Z(11), Z(11) ] ];;
gap> C:=EvaluationCode(Pts,L,R);
a linear [11,8,1..3]2..3  evaluation code over GF(11)
```

```
gap> MinimumDistance(C);
3
```

5.6.2 GeneralizedReedSolomonCode

▷ `GeneralizedReedSolomonCode(P, k, R)` (function)

Input: R=F[x], where F is a finite field, k is a positive integer, P is a list of n points in F.
Output: The C which is the image of the evaluation map

$$Eval_P : F[x]_k \to F^n,$$

given by $f \longmapsto (f(p_1),...,f(p_n))$, where $P = \{p_1,...,p_n\} \subset F$ and f ranges over the space $F[x]_k$ of all polynomials of degree less than k.

This command returns a "record" object `C` with several extra components (type `NamesOfComponents(C)` to see them all): `C!.points` (namely P), `C!.degree` (namely k), and `C!.ring` (namely R).

This code can be decoded using `Decodeword`, which applies the special decoder method (the interpolation method), or using `GeneralizedReedSolomonDecoderGao` which applies an algorithm of S. Gao (see `GeneralizedReedSolomonDecoderGao` (4.10.3)). This code has a special decoder record which implements the interpolation algorithm described in section 5.2 of Justesen and Hoholdt [JH04]. See `Decode` (4.10.1) and `Decodeword` (4.10.2) for more details.

The weighted version has implemented with the option `GeneralizedReedSolomonCode(P,k,R,wts)`, where $wts = [v_1,...,v_n]$ is a sequence of n non-zero elements from the base field F of R. See also the generalized Reed--Solomon code $GRS_k(P,V)$ described in [MS83], p.303.

The list-decoding algorithm of Sudan-Guruswami (described in section 12.1 of [JH04]) has been implemented for generalized Reed-Solomon codes. See `GeneralizedReedSolomonListDecoder` (4.10.4).

Example

```
gap> R:=PolynomialRing(GF(11),["t"]);
GF(11)[t]
gap> P:=List([1,3,4,5,7],i->Z(11)^i);
[ Z(11), Z(11)^3, Z(11)^4, Z(11)^5, Z(11)^7 ]
gap> C:=GeneralizedReedSolomonCode(P,3,R);
a linear [5,3,1..3]2  generalized Reed-Solomon code over GF(11)
gap> MinimumDistance(C);
3
gap> V:=[Z(11)^0,Z(11)^0,Z(11)^0,Z(11)^0,Z(11)];
[ Z(11)^0, Z(11)^0, Z(11)^0, Z(11)^0, Z(11) ]
gap> C:=GeneralizedReedSolomonCode(P,3,R,V);
a linear [5,3,1..3]2  weighted generalized Reed-Solomon code over GF(11)
gap> MinimumDistance(C);
3
```

See `EvaluationCode` (5.6.1) for a more general construction.

5.6.3 GeneralizedReedMullerCode

▷ `GeneralizedReedMullerCode(Pts, r, F)` (function)

`GeneralizedReedMullerCode` returns a 'Reed-Muller code' C with length $|Pts|$ and order r. One considers (a) a basis of monomials for the vector space over $F = GF(q)$ of all polynomials in $F[x_1,...,x_d]$ of degree at most r, and (b) a set Pts of points in F^d. The generator matrix of the associated *Reed-Muller code* C is $G = (f(p))_{f \in B, p \in Pts}$. This code C is constructed using the command `GeneralizedReedMullerCode(Pts,r,F)`. When Pts is the set of all q^d points in F^d then the command `GeneralizedReedMuller(d,r,F)` yields the code. When Pts is the set of all $(q-1)^d$ points with no coordinate equal to 0 then this is can be constructed using the `ToricCode` command (as a special case).

This command returns a "record" object `C` with several extra components (type `NamesOfComponents(C)` to see them all): `C!.points` (namely Pts) and `C!.degree` (namely r).

Example

```
gap> Pts:=ToricPoints(2,GF(5));
[ [ Z(5)^0, Z(5)^0 ], [ Z(5)^0, Z(5) ], [ Z(5)^0, Z(5)^2 ],
  [ Z(5)^0, Z(5)^3 ], [ Z(5), Z(5)^0 ], [ Z(5), Z(5) ], [ Z(5), Z(5)^2 ],
  [ Z(5), Z(5)^3 ], [ Z(5)^2, Z(5)^0 ], [ Z(5)^2, Z(5) ], [ Z(5)^2, Z(5)^2 ],
  [ Z(5)^2, Z(5)^3 ], [ Z(5)^3, Z(5)^0 ], [ Z(5)^3, Z(5) ],
  [ Z(5)^3, Z(5)^2 ], [ Z(5)^3, Z(5)^3 ] ]
gap> C:=GeneralizedReedMullerCode(Pts,2,GF(5));
a linear [16,6,1..8]6..10  generalized Reed-Muller code over GF(5)
```

See `EvaluationCode` (5.6.1) for a more general construction.

5.6.4 ToricPoints

▷ `ToricPoints(n, F)` (function)

`ToricPoints(n,F)` returns the points in $(F^\times)^n$.

Example

```
gap> ToricPoints(2,GF(5));
[ [ Z(5)^0, Z(5)^0 ], [ Z(5)^0, Z(5) ], [ Z(5)^0, Z(5)^2 ],
  [ Z(5)^0, Z(5)^3 ], [ Z(5), Z(5)^0 ], [ Z(5), Z(5) ], [ Z(5), Z(5)^2 ],
  [ Z(5), Z(5)^3 ], [ Z(5)^2, Z(5)^0 ], [ Z(5)^2, Z(5) ], [ Z(5)^2, Z(5)^2 ],
  [ Z(5)^2, Z(5)^3 ], [ Z(5)^3, Z(5)^0 ], [ Z(5)^3, Z(5) ],
  [ Z(5)^3, Z(5)^2 ], [ Z(5)^3, Z(5)^3 ] ]
```

5.6.5 ToricCode

▷ `ToricCode(L, F)` (function)

This function returns the toric codes as in D. Joyner [Joy04] (see also J. P. Hansen [Han00]). This is a truncated (generalized) Reed-Muller code. Here L is a list of integral vectors and F is the finite field. The size of F must be different from 2.

This command returns a record object `C` with an extra component (type `NamesOfComponents(C)` to see them all): `C!.exponents` (namely L).

Example

```
gap> C:=ToricCode([[1,0],[3,4]],GF(3));
a linear [4,1,4]2  toric code over GF(3)
gap> Display(GeneratorMat(C));
 1 1 2 2
gap> Elements(C);
[ [ 0 0 0 0 ], [ 1 1 2 2 ], [ 2 2 1 1 ] ]
```

See `EvaluationCode` (5.6.1) for a more general construction.

5.7 Algebraic geometric codes

Certain GUAVA functions related to algebraic geometric codes are described in this section.

5.7.1 AffineCurve

▷ `AffineCurve(`*poly, ring*`)` (function)

This function simply defines the data structure of an affine plane curve. In GUAVA, an affine curve is a record *crv* having two components: a polynomial *poly*, accessed in GUAVA by *crv.polynomial*, and a polynomial ring over a field F in two variables *ring*, accessed in GUAVA by *crv.ring*, containing *poly*. You use this function to define a curve in GUAVA.

For example, for the ring, one could take $\mathbb{Q}[x,y]$, and for the polynomial one could take $f(x,y) = x^2+y^2-1$. For the affine line, simply taking $\mathbb{Q}[x,y]$ for the ring and $f(x,y) = y$ for the polynomial.

(Not sure if F needs to be a field in fact ...)

To compute its degree, simply use the `DegreeMultivariatePolynomial` (7.6.2) command.

Example

```
gap> F:=GF(11);;
gap> R2:=PolynomialRing(F,["x","y"]);
GF(11)[x,y]
gap> vars:=IndeterminatesOfPolynomialRing(R2);;
gap> x:=vars[1];; y:=vars[2];;
gap> poly:=y;; crvP1:=AffineCurve(poly,R2);
rec( polynomial := y, ring := GF(11)[x,y] )
gap> degree_crv:=DegreeMultivariatePolynomial(poly,R2);
1
gap> poly:=y^2-x*(x^2-1);; ell_crv:=AffineCurve(poly,R2);
rec( polynomial := -x^3+y^2+x, ring := GF(11)[x,y] )
gap> degree_crv:=DegreeMultivariatePolynomial(poly,R2);
3
gap> poly:=x^2+y^2-1;; circle:=AffineCurve(poly,R2);
rec( polynomial := x^2+y^2-Z(11)^0, ring := GF(11)[x,y] )
gap> degree_crv:=DegreeMultivariatePolynomial(poly,R2);
2
gap> q:=3;;
gap> F:=GF(q^2);;
gap> R:=PolynomialRing(F,["x","y"]);;
gap> vars:=IndeterminatesOfPolynomialRing(R);
[ x, y ]
```

```
gap> x:=vars[1];; y:=vars[2];;
gap> crv:=AffineCurve(y^q+y-x^(q+1),R);
rec( polynomial := -x^4+y^3+y, ring := GF(3^2)[x,y] )
```

In GAP, a *point* on a curve defined by $f(x,y) = 0$ is simply a list `[a,b]` of elements of F satisfying this polynomial equation.

5.7.2 AffinePointsOnCurve

▷ `AffinePointsOnCurve(f, R, E)` (function)

`AffinePointsOnCurve(f,R,E)` returns the points $(x,y) \in E^2$ satisying $f(x,y) = 0$, where `f` is an element of $R = F[x,y]$.

Example

```
gap> F:=GF(11);;
gap> R := PolynomialRing(F,["x","y"]);
GF(11)[x,y]
gap> indets := IndeterminatesOfPolynomialRing(R);;
gap> x:=indets[1];; y:=indets[2];;
gap> P:=AffinePointsOnCurve(y^2-x^11+x,R,F);
[ [ Z(11)^9, 0*Z(11) ], [ Z(11)^8, 0*Z(11) ], [ Z(11)^7, 0*Z(11) ],
  [ Z(11)^6, 0*Z(11) ], [ Z(11)^5, 0*Z(11) ], [ Z(11)^4, 0*Z(11) ],
  [ Z(11)^3, 0*Z(11) ], [ Z(11)^2, 0*Z(11) ], [ Z(11), 0*Z(11) ],
  [ Z(11)^0, 0*Z(11) ], [ 0*Z(11), 0*Z(11) ] ]
```

5.7.3 GenusCurve

▷ `GenusCurve(crv)` (function)

If `crv` represents $f(x,y) = 0$, where f is a polynomial of degree d, then this function simply returns $(d-1)(d-2)/2$. At the present, the function does not check if the curve is singular (in which case the result may be false).

Example

```
gap> q:=4;;
gap> F:=GF(q^2);;
gap> a:=Indeterminate(F, "a");;
gap> R1:=PolynomialRing(F,[a]);;
gap> var1:=IndeterminatesOfPolynomialRing(R1);;
gap> b:=Indeterminate(F, "b");;
gap> R2:=PolynomialRing(F,[a,b]);;
gap> var2:=IndeterminatesOfPolynomialRing(R2);;
gap> crv:=AffineCurve(b^q+b-a^(q+1),R2);
rec( polynomial := a^5+b^4+b, ring := GF(2^4)[a,b] )
gap> GenusCurve(crv);
6
```

5.7.4 GOrbitPoint

▷ `GOrbitPoint (G, P)` (function)

`P` must be a point in projective space $\mathbb{P}^n(F)$, `G` must be a finite subgroup of $GL(n+1,F)$, This function returns all (representatives of projective) points in the orbit $G \cdot P$.

The example below computes the orbit of the automorphism group on the Klein quartic over the field $GF(43)$ on the "point at infinity".

Example

```
gap> R:= PolynomialRing( GF(43), 3 );;
gap> vars:= IndeterminatesOfPolynomialRing(R);;
gap> x:= vars[1];; y:= vars[2];; z:= vars[3];;
gap> zz:=Z(43)^6;
Z(43)^6
gap> zzz:=Z(43);
Z(43)
gap> rho1:=zz^0*[[zz^4,0,0],[0,zz^2,0],[0,0,zz]];
[ [ Z(43)^24, 0*Z(43), 0*Z(43) ], [ 0*Z(43), Z(43)^12, 0*Z(43) ],
  [ 0*Z(43), 0*Z(43), Z(43)^6 ] ]
gap> rho2:=zz^0*[[0,1,0],[0,0,1],[1,0,0]];
[ [ 0*Z(43), Z(43)^0, 0*Z(43) ], [ 0*Z(43), 0*Z(43), Z(43)^0 ],
  [ Z(43)^0, 0*Z(43), 0*Z(43) ] ]
gap> rho3:=(-1)*[[(zz-zz^6 )/zzz^7,( zz^2-zz^5 )/ zzz^7, ( zz^4-zz^3 )/ zzz^7],
>             [( zz^2-zz^5 )/ zzz^7, ( zz^4-zz^3 )/ zzz^7, ( zz-zz^6 )/ zzz^7],
>             [( zz^4-zz^3 )/ zzz^7, ( zz-zz^6 )/ zzz^7, ( zz^2-zz^5 )/ zzz^7]];
[ [ Z(43)^9, Z(43)^28, Z(43)^12 ], [ Z(43)^28, Z(43)^12, Z(43)^9 ],
  [ Z(43)^12, Z(43)^9, Z(43)^28 ] ]
gap> G:=Group([rho1,rho2,rho3]);; #PSL(2,7)
gap> Size(G);
168
gap> P:=[1,0,0]*zzz^0;
[ Z(43)^0, 0*Z(43), 0*Z(43) ]
gap> O:=GOrbitPoint(G,P);
[ [ 0*Z(43), 0*Z(43), Z(43)^0 ], [ 0*Z(43), Z(43)^0, 0*Z(43) ],
  [ Z(43)^0, 0*Z(43), 0*Z(43) ], [ Z(43)^0, Z(43), Z(43)^39 ],
  [ Z(43)^0, Z(43)^2, Z(43)^29 ], [ Z(43)^0, Z(43)^3, Z(43)^4 ],
  [ Z(43)^0, Z(43)^7, Z(43)^27 ], [ Z(43)^0, Z(43)^8, Z(43)^17 ],
  [ Z(43)^0, Z(43)^9, Z(43)^34 ], [ Z(43)^0, Z(43)^13, Z(43)^15 ],
  [ Z(43)^0, Z(43)^14, Z(43)^5 ], [ Z(43)^0, Z(43)^15, Z(43)^22 ],
  [ Z(43)^0, Z(43)^19, Z(43)^3 ], [ Z(43)^0, Z(43)^20, Z(43)^35 ],
  [ Z(43)^0, Z(43)^21, Z(43)^10 ], [ Z(43)^0, Z(43)^25, Z(43)^33 ],
  [ Z(43)^0, Z(43)^26, Z(43)^23 ], [ Z(43)^0, Z(43)^27, Z(43)^40 ],
  [ Z(43)^0, Z(43)^31, Z(43)^21 ], [ Z(43)^0, Z(43)^32, Z(43)^11 ],
  [ Z(43)^0, Z(43)^33, Z(43)^28 ], [ Z(43)^0, Z(43)^37, Z(43)^9 ],
  [ Z(43)^0, Z(43)^38, Z(43)^41 ], [ Z(43)^0, Z(43)^39, Z(43)^16 ] ]
gap> Length(O);
24
```

Informally, a *divisor* on a curve is a formal integer linear combination of points on the curve, $D = m_1P_1 + ... + m_kP_k$, where the m_i are integers (the "multiplicity" of P_i in D) and P_i are (F-rational) points on the affine plane curve. In other words, a divisor is an element of the free abelian group generated by the F-rational affine points on the curve. The *support* of a divisor D is simply the set of points which occurs in the sum defining D with non-zero "multiplicity". The data structure for a divisor on an affine plane curve is a record having the following components:

- the coefficients (the integer weights of the points in the support),

- the support,
- the curve, itself a record which has components: polynomial and polynomial ring.

5.7.5 DivisorOnAffineCurve

▷ `DivisorOnAffineCurve(cdiv, sdiv, crv)` (function)

This is the command you use to define a divisor in GUAVA. Of course, `crv` is the curve on which the divisor lives, `cdiv` is the list of coefficients (or "multiplicities"), `sdiv` is the list of points on `crv` in the support.

Example

```
gap> q:=5;;
gap> F:=GF(q);;
gap> R:=PolynomialRing(F,["x","y"]);;
gap> vars:=IndeterminatesOfPolynomialRing(R);;
gap> x:=vars[1];; y:=vars[2];;
gap> crv:=AffineCurve(y^3-x^3-x-1,R);
rec( polynomial := -x^3+y^3-x-Z(5)^0, ring := GF(5)[x,y] )
gap> Pts:=AffinePointsOnCurve(crv!.polynomial,R,F);;
gap> supp:=Z(5)^0*[[0,1],[1,2]];; IsSubset(Pts, supp);
true
gap> D:=DivisorOnAffineCurve([1,-1],supp,crv);
rec( coeffs := [ 1, -1 ],
  curve := rec( polynomial := -x^3+y^3-x-Z(5)^0, ring := GF(5)[x,y] ),
  support := [ [ 0*Z(5), Z(5)^0 ], [ Z(5)^0, Z(5) ] ] )
```

5.7.6 DivisorAddition

▷ `DivisorAddition (D1, D2)` (function)

If $D_1 = m_1P_1 + ... + m_kP_k$ and $D_2 = n_1P_1 + ... + n_kP_k$ are divisors then $D_1 + D_2 = (m_1 + n_1)P_1 + ... + (m_k + n_k)P_k$.

5.7.7 DivisorDegree

▷ `DivisorDegree (D)` (function)

If $D = m_1P_1 + ... + m_kP_k$ is a divisor then the *degree* is $m_1 + ... + m_k$.

5.7.8 DivisorNegate

▷ `DivisorNegate (D)` (function)

Self-explanatory.

5.7.9 DivisorIsZero

▷ `DivisorIsZero (D)` (function)

Self-explanatory.

5.7.10 DivisorsEqual

▷ `DivisorsEqual (D1, D2)` (function)

Self-explanatory.

5.7.11 DivisorGCD

▷ `DivisorGCD (D1, D2)` (function)

If $m = p_1^{e_1} \ldots p_k^{e_k}$ and $n = p_1^{f_1} \ldots p_k^{f_k}$ are two integers then their greatest common divisor is $GCD(m,n) = p_1^{min(e_1,f_1)} \ldots p_k^{min(e_k,f_k)}$. A similar definition works for two divisors on a curve. If $D_1 = e_1P_1 + \ldots + e_kP_k$ and $D_2 n = f_1P_1 + \ldots + f_kP_k$ are two divisors on a curve then their *greatest common divisor* is $GCD(m,n) = min(e_1,f_1)P_1 + \ldots + min(e_k,f_k)P_k$. This function computes this quantity.

5.7.12 DivisorLCM

▷ `DivisorLCM (D1, D2)` (function)

If $m = p_1^{e_1} \ldots p_k^{e_k}$ and $n = p_1^{f_1} \ldots p_k^{f_k}$ are two integers then their least common multiple is $LCM(m,n) = p_1^{max(e_1,f_1)} \ldots p_k^{max(e_k,f_k)}$. A similar definition works for two divisors on a curve. If $D_1 = e_1P_1 + \ldots + e_kP_k$ and $D_2 = f_1P_1 + \ldots + f_kP_k$ are two divisors on a curve then their *least common multiple* is $LCM(m,n) = max(e_1,f_1)P_1 + \ldots + max(e_k,f_k)P_k$. This function computes this quantity.

Example

```
gap> F:=GF(11);
GF(11)
gap> R1:=PolynomialRing(F,["a"]);;
gap> var1:=IndeterminatesOfPolynomialRing(R1);; a:=var1[1];;
gap> b:=X(F,"b",var1);
b
gap> var2:=Concatenation(var1,[b]);
[ a, b ]
gap> R2:=PolynomialRing(F,var2);
GF(11)[a,b]
gap> crvP1:=AffineCurve(b,R2);
rec( polynomial := b, ring := GF(11)[a,b] )
gap> div1:=DivisorOnAffineCurve([1,2,3,4],[Z(11)^2,Z(11)^3,Z(11)^7,Z(11)],crvP1);
rec( coeffs := [ 1, 2, 3, 4 ],
  curve := rec( polynomial := b, ring := GF(11)[a,b] ),
  support := [ Z(11)^2, Z(11)^3, Z(11)^7, Z(11) ] )
gap> DivisorDegree(div1);
10
gap> div2:=DivisorOnAffineCurve([1,2,3,4],[Z(11),Z(11)^2,Z(11)^3,Z(11)^4],crvP1);
rec( coeffs := [ 1, 2, 3, 4 ],
```

```
  curve := rec( polynomial := b, ring := GF(11)[a,b] ),
  support := [ Z(11), Z(11)^2, Z(11)^3, Z(11)^4 ] )
gap> DivisorDegree(div2);
10
gap> div3:=DivisorAddition(div1,div2);
rec( coeffs := [ 5, 3, 5, 4, 3 ],
  curve := rec( polynomial := b, ring := GF(11)[a,b] ),
  support := [ Z(11), Z(11)^2, Z(11)^3, Z(11)^4, Z(11)^7 ] )
gap> DivisorDegree(div3);
20
gap> DivisorIsEffective(div1);
true
gap> DivisorIsEffective(div2);
true
gap> ndiv1:=DivisorNegate(div1);
rec( coeffs := [ -1, -2, -3, -4 ],
  curve := rec( polynomial := b, ring := GF(11)[a,b] ),
  support := [ Z(11)^2, Z(11)^3, Z(11)^7, Z(11) ] )
gap> zdiv:=DivisorAddition(div1,ndiv1);
rec( coeffs := [ 0, 0, 0, 0 ],
  curve := rec( polynomial := b, ring := GF(11)[a,b] ),
  support := [ Z(11), Z(11)^2, Z(11)^3, Z(11)^7 ] )
gap> DivisorIsZero(zdiv);
true
gap> div_gcd:=DivisorGCD(div1,div2);
rec( coeffs := [ 1, 1, 2, 0, 0 ],
  curve := rec( polynomial := b, ring := GF(11)[a,b] ),
  support := [ Z(11), Z(11)^2, Z(11)^3, Z(11)^4, Z(11)^7 ] )
gap> div_lcm:=DivisorLCM(div1,div2);
rec( coeffs := [ 4, 2, 3, 4, 3 ],
  curve := rec( polynomial := b, ring := GF(11)[a,b] ),
  support := [ Z(11), Z(11)^2, Z(11)^3, Z(11)^4, Z(11)^7 ] )
gap> DivisorDegree(div_gcd);
4
gap> DivisorDegree(div_lcm);
16
gap> DivisorEqual(div3,DivisorAddition(div_gcd,div_lcm));
true
```

Let G denote a finite subgroup of $PGL(2,F)$ and let D denote a divisor on the projective line $\mathbb{P}^1(F)$. If G leaves D unchanged (it may permute the points in the support of D but must preserve their sum in D) then the Riemann-Roch space $L(D)$ is a G-module. The commands in this section help explore the G-module structure of $L(D)$ in the case then the ground field F is finite.

5.7.13 RiemannRochSpaceBasisFunctionP1

▷ `RiemannRochSpaceBasisFunctionP1 (P, k, R2)` (function)

Input: `R2` is a polynomial ring in two variables, say $F[x,y]$; `P` is an element of the base field, say F; `k` is an integer. Output: $1/(x-P)^k$

5.7.14 DivisorOfRationalFunctionP1

▷ DivisorOfRationalFunctionP1 (*f*, *R*) (function)

Here $R = F[x,y]$ is a polynomial ring in the variables x, y and f is a rational function of x. Simply returns the principal divisor on $\mathbb{P}^1$ associated to f.

Example

```
gap> F:=GF(11);
GF(11)
gap> R1:=PolynomialRing(F,["a"]);;
gap> var1:=IndeterminatesOfPolynomialRing(R1);; a:=var1[1];;
gap> b:=X(F,"b",var1);
b
gap> var2:=Concatenation(var1,[b]);
[ a, b ]
gap> R2:=PolynomialRing(F,var2);
GF(11)[a,b]
gap> pt:=Z(11);
Z(11)
gap> f:=RiemannRochSpaceBasisFunctionP1(pt,2,R2);
(Z(11)^0)/(a^2+Z(11)^7*a+Z(11)^2)
gap> Df:=DivisorOfRationalFunctionP1(f,R2);
rec( coeffs := [ -2 ], curve := rec( polynomial := a, ring := GF(11)[a,b] ),
  support := [ Z(11) ] )
gap> Df.support;
[ Z(11) ]
gap> F:=GF(11);;
gap> R:=PolynomialRing(F,["x","y"]);;
gap> vars:=IndeterminatesOfPolynomialRing(R);;
gap> a:=vars[1];;
gap> b:=vars[2];;
gap> f:=(a^4+Z(11)^6*a^3-a^2+Z(11)^7*a+Z(11)^0)/(a^4+Z(11)*a^2+Z(11)^7*a+Z(11));;
gap> divf:=DivisorOfRationalFunctionP1(f,R);
rec( coeffs := [ 3, 1 ], curve := rec( polynomial := x, ring := GF(11)[x,y] ),
  support := [ Z(11), Z(11)^7 ] )
gap> denf:=DenominatorOfRationalFunction(f); RootsOfUPol(denf);
x^4+Z(11)*x^2+Z(11)^7*x+Z(11)
[  ]
gap> numf:=NumeratorOfRationalFunction(f); RootsOfUPol(numf);
x^4+Z(11)^6*x^3-x^2+Z(11)^7*x+Z(11)^0
[ Z(11)^7, Z(11), Z(11), Z(11) ]
```

5.7.15 RiemannRochSpaceBasisP1

▷ RiemannRochSpaceBasisP1 (*D*) (function)

This returns the basis of the Riemann-Roch space $L(D)$ associated to the divisor `D` on the projective line $\mathbb{P}^1$.

Example

```
gap> F:=GF(11);
GF(11)
gap> R1:=PolynomialRing(F,["a"]);;
```

```
gap> var1:=IndeterminatesOfPolynomialRing(R1);; a:=var1[1];;
gap> b:=X(F,"b",var1);
b
gap> var2:=Concatenation(var1,[b]);
[ a, b ]
gap> R2:=PolynomialRing(F,var2);
GF(11)[a,b]
gap> crvP1:=AffineCurve(b,R2);
rec( polynomial := b, ring := GF(11)[a,b] )
gap> D:=DivisorOnAffineCurve([1,2,3,4],[Z(11)^2,Z(11)^3,Z(11)^7,Z(11)],crvP1);
rec( coeffs := [ 1, 2, 3, 4 ],
  curve := rec( polynomial := b, ring := GF(11)[a,b] ),
  support := [ Z(11)^2, Z(11)^3, Z(11)^7, Z(11) ] )
gap> B:=RiemannRochSpaceBasisP1(D);
[ (Z(11)^0)/(a+Z(11)^7), (Z(11)^0)/(a+Z(11)^8),
  (Z(11)^0)/(a^2+Z(11)^9*a+Z(11)^6), (Z(11)^0)/(a+Z(11)^2),
  (Z(11)^0)/(a^2+Z(11)^3*a+Z(11)^4), (Z(11)^0)/(a^3+a^2+Z(11)^2*a+Z(11)^6),
  (Z(11)^0)/(a+Z(11)^6), (Z(11)^0)/(a^2+Z(11)^7*a+Z(11)^2),
  (Z(11)^0)/(a^3+Z(11)^4*a^2+a+Z(11)^8),
  (Z(11)^0)/(a^4+Z(11)^8*a^3+Z(11)*a^2+a+Z(11)^4), Z(11)^0 ]
gap> DivisorOfRationalFunctionP1(B[1],R2).support;
[ Z(11)^2 ]
gap> DivisorOfRationalFunctionP1(B[2],R2).support;
[ Z(11)^3 ]
gap> DivisorOfRationalFunctionP1(B[3],R2).support;
[ Z(11)^3 ]
gap> DivisorOfRationalFunctionP1(B[4],R2).support;
[ Z(11)^7 ]
gap> DivisorOfRationalFunctionP1(B[5],R2).support;
[ Z(11)^7 ]
gap> DivisorOfRationalFunctionP1(B[6],R2).support;
[ Z(11)^7 ]
gap> DivisorOfRationalFunctionP1(B[7],R2).support;
[ Z(11) ]
gap> DivisorOfRationalFunctionP1(B[8],R2).support;
[ Z(11) ]
gap> DivisorOfRationalFunctionP1(B[9],R2).support;
[ Z(11) ]
gap> DivisorOfRationalFunctionP1(B[10],R2).support;
[ Z(11) ]
gap> DivisorOfRationalFunctionP1(B[11],R2).support;
[  ]
```

5.7.16 MoebiusTransformation

▷ `MoebiusTransformation (A, R)` (function)

The arguments are a 2×2 matrix A with entries in a field F and a polynomial ring $R$of one variable, say $F[x]$. This function returns the linear fractional transformatio associated to A. These transformations can be composed with each other using GAP's `Value` command.

5.7.17 ActionMoebiusTransformationOnFunction

▷ `ActionMoebiusTransformationOnFunction (A, f, R2)` (function)

The arguments are a 2×2 matrix A with entries in a field F, a rational function `f` of one variable, say in $F(x)$, and a polynomial ring `R2`, say $F[x,y]$. This function simply returns the composition of the function `f` with the Möbius transformation of `A`.

5.7.18 ActionMoebiusTransformationOnDivisorP1

▷ `ActionMoebiusTransformationOnDivisorP1 (A, D)` (function)

A Möbius transformation may be regarded as an automorphism of the projective line $\mathbb{P}^1$. This function simply returns the image of the divisor `D` under the Möbius transformation defined by `A`, provided that `IsActionMoebiusTransformationOnDivisorDefinedP1(A,D)` returns true.

5.7.19 IsActionMoebiusTransformationOnDivisorDefinedP1

▷ `IsActionMoebiusTransformationOnDivisorDefinedP1 (A, D)` (function)

Returns true of none of the points in the support of the divisor `D` is the pole of the Möbius transformation.

Example

```
gap> F:=GF(11);
GF(11)
gap> R1:=PolynomialRing(F,["a"]);;
gap> var1:=IndeterminatesOfPolynomialRing(R1);; a:=var1[1];;
gap> b:=X(F,"b",var1);
b
gap> var2:=Concatenation(var1,[b]);
[ a, b ]
gap> R2:=PolynomialRing(F,var2);
GF(11)[a,b]
gap> crvP1:=AffineCurve(b,R2);
rec( polynomial := b, ring := GF(11)[a,b] )
gap> D:=DivisorOnAffineCurve([1,2,3,4],[Z(11)^2,Z(11)^3,Z(11)^7,Z(11)],crvP1);
rec( coeffs := [ 1, 2, 3, 4 ],
  curve := rec( polynomial := b, ring := GF(11)[a,b] ),
  support := [ Z(11)^2, Z(11)^3, Z(11)^7, Z(11) ] )
gap> A:=Z(11)^0*[[1,2],[1,4]];
[ [ Z(11)^0, Z(11) ], [ Z(11)^0, Z(11)^2 ] ]
gap> ActionMoebiusTransformationOnDivisorDefinedP1(A,D);
false
gap> A:=Z(11)^0*[[1,2],[3,4]];
[ [ Z(11)^0, Z(11) ], [ Z(11)^8, Z(11)^2 ] ]
gap> ActionMoebiusTransformationOnDivisorDefinedP1(A,D);
true
gap> ActionMoebiusTransformationOnDivisorP1(A,D);
rec( coeffs := [ 1, 2, 3, 4 ],
  curve := rec( polynomial := b, ring := GF(11)[a,b] ),
  support := [ Z(11)^5, Z(11)^6, Z(11)^8, Z(11)^7 ] )
```

```
gap> f:=MoebiusTransformation(A,R1);
(a+Z(11))/(Z(11)^8*a+Z(11)^2)
gap> ActionMoebiusTransformationOnFunction(A,f,R1);
-Z(11)^0+Z(11)^3*a^-1
```

5.7.20 DivisorAutomorphismGroupP1

▷ `DivisorAutomorphismGroupP1 (D)` (function)

Input: A divisor `D` on $\mathbb{P}^1(F)$, where F is a finite field. Output: A subgroup $Aut(D) \subset Aut(\mathbb{P}^1)$ preserving `D`.

Very slow.

Example

```
gap> F:=GF(11);
GF(11)
gap> R1:=PolynomialRing(F,["a"]);;
gap> var1:=IndeterminatesOfPolynomialRing(R1);; a:=var1[1];;
gap> b:=X(F,"b",var1);
b
gap> var2:=Concatenation(var1,[b]);
[ a, b ]
gap> R2:=PolynomialRing(F,var2);
GF(11)[a,b]
gap> crvP1:=AffineCurve(b,R2);
rec( polynomial := b, ring := GF(11)[a,b] )
gap> D:=DivisorOnAffineCurve([1,2,3,4],[Z(11)^2,Z(11)^3,Z(11)^7,Z(11)],crvP1);
rec( coeffs := [ 1, 2, 3, 4 ],
  curve := rec( polynomial := b, ring := GF(11)[a,b] ),
  support := [ Z(11)^2, Z(11)^3, Z(11)^7, Z(11) ] )
gap> agp:=DivisorAutomorphismGroupP1(D);;
gap> IdGroup(agp);
[ 10, 2 ]
```

5.7.21 MatrixRepresentationOnRiemannRochSpaceP1

▷ `MatrixRepresentationOnRiemannRochSpaceP1 (g, D)` (function)

Input: An element g in G, a subgroup of $Aut(D) \subset Aut(\mathbb{P}^1)$, and a divisor `D` on $\mathbb{P}^1(F)$, where F is a finite field. Output: a $d \times d$ matrix, where $d = dim L(D)$, representing the action of g on $L(D)$.

Note: g sends $L(D)$ to $r \cdot L(D)$, where r is a polynomial of degree 1 depending on g and `D`.

Also very slow.

The GAP command `BrauerCharacterValue` can be used to "lift" the eigenvalues of this matrix to the complex numbers.

Example

```
gap> F:=GF(11);
GF(11)
gap> R1:=PolynomialRing(F,["a"]);;
gap> var1:=IndeterminatesOfPolynomialRing(R1);; a:=var1[1];;
gap> b:=X(F,"b",var1);
b
```

```
gap> var2:=Concatenation(var1,[b]);
[ a, b ]
gap> R2:=PolynomialRing(F,var2);
GF(11)[a,b]
gap> crvP1:=AffineCurve(b,R2);
rec( polynomial := b, ring := GF(11)[a,b] )
gap> D:=DivisorOnAffineCurve([1,1,1,4],[Z(11)^2,Z(11)^3,Z(11)^7,Z(11)],crvP1);
rec( coeffs := [ 1, 1, 1, 4 ],
  curve := rec( polynomial := b, ring := GF(11)[a,b] ),
  support := [ Z(11)^2, Z(11)^3, Z(11)^7, Z(11) ] )
gap> agp:=DivisorAutomorphismGroupP1(D);;
gap> IdGroup(agp);
[ 20, 5 ]
gap> # a scalar matrix acts as the identity on P^1, so take one that is not
gap> g:=First(AsSortedList(agp), m -> m[1][2] <> Zero(F));
[ [ Z(11)^0, Z(11)^5 ], [ Z(11)^6, Z(11)^5 ] ]
gap> rho:=MatrixRepresentationOnRiemannRochSpaceP1(g,D);
[ [ 0*Z(11), 0*Z(11), Z(11), 0*Z(11), 0*Z(11), 0*Z(11), 0*Z(11), Z(11)^0 ],
  [ 0*Z(11), Z(11)^5, 0*Z(11), 0*Z(11), 0*Z(11), 0*Z(11), 0*Z(11), Z(11)^7 ],
  [ Z(11)^9, 0*Z(11), 0*Z(11), 0*Z(11), 0*Z(11), 0*Z(11), 0*Z(11), Z(11)^4 ],
  [ 0*Z(11), 0*Z(11), 0*Z(11), Z(11)^5, 0*Z(11), 0*Z(11), 0*Z(11), Z(11)^2 ],
  [ 0*Z(11), 0*Z(11), 0*Z(11), Z(11)^8, Z(11)^0, 0*Z(11), 0*Z(11), Z(11)^4 ],
  [ 0*Z(11), 0*Z(11), 0*Z(11), Z(11)^7, Z(11)^0, Z(11)^5, 0*Z(11), Z(11)^6 ],
  [ 0*Z(11), 0*Z(11), 0*Z(11), Z(11)^3, Z(11)^3, Z(11)^9, Z(11)^0, Z(11)^8 ],
  [ 0*Z(11), 0*Z(11), 0*Z(11), 0*Z(11), 0*Z(11), 0*Z(11), 0*Z(11), Z(11)^0 ] ]
gap> Display(rho);
  .  .  2  .  .  .  .  1
  . 10  .  .  .  .  .  7
  6  .  .  .  .  .  .  5
  .  .  . 10  .  .  .  4
  .  .  .  3  1  .  .  5
  .  .  .  7  1 10  .  9
  .  .  .  8  8  6  1  3
  .  .  .  .  .  .  .  1
```

5.7.22 GoppaCodeClassical

▷ `GoppaCodeClassical(`*`div`*`, `*`pts`*`)` (function)

Input: A divisor *div* on the projective line $\mathbb{P}^1(F)$ over a finite field F and a list *pts* of points $\{P_1,\ldots,P_n\} \subset F$ disjoint from the support of *div*.
Output: The classical (evaluation) Goppa code associated to this data. This is the code

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

Example

```
gap> F:=GF(11);;
gap> R2:=PolynomialRing(F,["x","y"]);;
gap> vars:=IndeterminatesOfPolynomialRing(R2);;
gap> a:=vars[1];;b:=vars[2];;
gap> cdiv:=[ 1, 2, -1, -2 ];
[ 1, 2, -1, -2 ]
```

```
gap> sdiv:=[ Z(11)^2, Z(11)^3, Z(11)^6, Z(11)^9 ];
[ Z(11)^2, Z(11)^3, Z(11)^6, Z(11)^9 ]
gap> crv:=rec(polynomial:=b,ring:=R2);
rec( polynomial := y, ring := GF(11)[x,y] )
gap> div:=DivisorOnAffineCurve(cdiv,sdiv,crv);
rec( coeffs := [ 1, 2, -1, -2 ],
  curve := rec( polynomial := y, ring := GF(11)[x,y] ),
  support := [ Z(11)^2, Z(11)^3, Z(11)^6, Z(11)^9 ] )
gap> pts:=Difference(Elements(GF(11)),div.support);
[ 0*Z(11), Z(11)^0, Z(11), Z(11)^4, Z(11)^5, Z(11)^7, Z(11)^8 ]
gap> C:=GoppaCodeClassical(div,pts);
a linear [7,2,1..6]4..5 code defined by generator matrix over GF(11)
gap> MinimumDistance(C);
6
```

5.7.23 EvaluationBivariateCode

▷ EvaluationBivariateCode(*pts, L, crv*) (function)

Input: pts is a set of affine points on crv, L is a list of rational functions on crv.
Output: The evaluation code associated to the points in pts and functions in L, but specifically for affine plane curves and this function checks if points are "bad" (if so removes them from the list pts automatically). A point is “bad” if either it does not lie on the set of non-singular F-rational points (places of degree 1) on the curve.

Very similar to EvaluationCode (see EvaluationCode (5.6.1) for a more general construction).

5.7.24 EvaluationBivariateCodeNC

▷ EvaluationBivariateCodeNC(*pts, L, crv*) (function)

As in EvaluationBivariateCode but does not check if the points are “bad”.

Input: pts is a set of affine points on crv, L is a list of rational functions on crv.
Output: The evaluation code associated to the points in pts and functions in L.

Example

```
gap> q:=4;;
gap> F:=GF(q^2);;
gap> R:=PolynomialRing(F,["x","y"]);;
gap> vars:=IndeterminatesOfPolynomialRing(R);;
gap> x:=vars[1];; y:=vars[2];;
gap> crv:=AffineCurve(y^q+y-x^(q+1),R);
rec( polynomial := x^5+y^4+y, ring := GF(2^4)[x,y] )
gap> L:=[ x^0, x, x^2*y^-1 ];
[ Z(2)^0, x, x^2/y ]
gap> Pts:=AffinePointsOnCurve(crv.polynomial,crv.ring,F);;
gap> C1:=EvaluationBivariateCode(Pts,L,crv);


 Automatically removed the following 'bad' points (either a pole or not on the\
 curve):
[ [ 0*Z(2), 0*Z(2) ] ]
```

```
a linear [63,3,1..57]51..59  evaluation code over GF(16)
gap> P:=Difference(Pts,[[ 0*Z(2^4)^0, 0*Z(2)^0 ]]);;
gap> C2:=EvaluationBivariateCodeNC(P,L,crv);
a linear [63,3,1..56]51..59  evaluation code over GF(16)
gap> C3:=EvaluationCode(P,L,R);
a linear [63,3,1..56]51..59  evaluation code over GF(16)
gap> MinimumDistance(C1);
56
gap> MinimumDistance(C2);
56
gap> MinimumDistance(C3);
56
```

5.7.25 OnePointAGCode

▷ `OnePointAGCode(f, P, m, R)` (function)

Input: `f` is a polynomial in R=F[x,y], where `F` is a finite field, `m` is a positive integer (the multiplicity of the 'point at infinity' ∞ on the curve $f(x,y)=0$), `P` is a list of n points on the curve over F. Output: The C which is the image of the evaluation map

$$Eval_P : L(m\cdot\infty) \to F^n,$$

given by $f \longmapsto (f(p_1),...,f(p_n))$, where $p_i \in P$. Here $L(m\cdot\infty)$ denotes the Riemann-Roch space of the divisor $m\cdot\infty$ on the curve. This has a basis consisting of monomials $x^i y^j$, where (i,j) range over a polygon depending on m and $f(x,y)$. For more details on the Riemann-Roch space of the divisor $m\cdot\infty$ see Proposition III.10.5 in Stichtenoth [Sti93].

This command returns a "record" object `C` with several extra components (type `NamesOfComponents(C)` to see them all): `C!.points` (namely `P`), `C!.multiplicity` (namely `m`), `C!.curve` (namely `f`) and `C!.ring` (namely `R`).

Example

```
gap> F:=GF(11);;
gap> R := PolynomialRing(F,["x","y"]);
GF(11)[x,y]
gap> indets := IndeterminatesOfPolynomialRing(R);
[ x, y ]
gap> x:=indets[1];; y:=indets[2];;
gap> P:=AffinePointsOnCurve(y^2-x^11+x,R,F);;
gap> C:=OnePointAGCode(y^2-x^11+x,P,15,R);
a linear [11,8,1..4]2..3  one-point AG code over GF(11)
gap> MinimumDistance(C);
4
gap> Pts:=List([1,2,4,6,7,8,9,10,11],i->P[i]);;
gap> C:=OnePointAGCode(y^2-x^11+x,PT,10,R);
a linear [9,6,1..4]2..3 one-point AG code over GF(11)
gap> MinimumDistance(C);
4
```

See `EvaluationCode` (5.6.1) for a more general construction.

5.8 Low-Density Parity-Check Codes

Low-density parity-check (LDPC) codes form a class of linear block codes whose parity-check matrix--as the name implies, is sparse. LDPC codes were introduced by Robert Gallager in 1962 [Gal62] as his PhD work. Due to the decoding complexity for the technology back then, these codes were forgotten. Not until the late 1990s, these codes were rediscovered and research results have shown that LDPC codes can achieve near Shannon's capacity performance provided that their block length is long enough and soft-decision iterative decoder is employed. Note that the bit-flipping decoder (see `BitFlipDecoder`) is a hard-decision decoder and hence capacity achieving performance cannot be achieved despite having a large block length.

Based on the structure of their parity-check matrix, LDPC codes may be categorised into two classes:

- Regular LDPC codes

  This class of codes has a fixed number of non zeros per column and per row in their parity-check matrix. These codes are usually denoted as (n, j, k) codes where n is the block length, j is the number of non zeros per column in their parity-check matrix and k is the number of non zeros per row in their parity-check matrix.

- Irregular LDPC codes

  The irregular codes, on the other hand, do not have a fixed number of non zeros per column and row in their parity-check matrix. This class of codes are commonly represented by two polynomials which denote the distribution of the number of non zeros in the columns and rows respectively of their parity-check matrix.

5.8.1 QCLDPCCodeFromGroup

▷ `QCLDPCCodeFromGroup(m, j, k)` (function)

`QCLDPCCodeFromGroup` produces an (n, j, k) regular quasi-cyclic LDPC code over GF(2) of block length $n = mk$. The term quasi-cyclic in the context of LDPC codes typically refers to LDPC codes whose parity-check matrix H has the following form

```
      -                                                   -
      |  I_P(0,0)   |  I_P(0,1)   | ... |  I_P(0,k-1)   |
      |  I_P(1,0)   |  I_P(1,1)   | ... |  I_P(1,k-1)   |
H =   |       .     |       .     |  .  |       .       |,
      |       .     |       .     |  .  |       .       |
      | I_P(j-1,0)  | I_P(j-1,1)  | ... | I_P(j-1,k-1)  |
      -                                                   -
```

where $I_{P(s,t)}$ is an identity matrix of size $m \times m$ which has been shifted so that the 1 on the first row starts at position $P(s,t)$.

Let F be a multiplicative group of integers modulo m. If m is a prime, $F = \{0, 1, ..., m-1\}$, otherwise F contains a set of integers which are relatively prime to m. In both cases, the order of F is equal to $\phi(m)$. Let a and b be non zeros of F such that the orders of a and b are k and j respectively. Note that the integers a and b can always be found provided that k and j respectively divide $\phi(m)$.

Having obtain integers a and b, construct the following $j \times k$ matrix P so that the element at row s and column t is given by $P(s,t) = a^t b^s$, i.e.

```
     -                                                  -
     |     1     |      a     | . . . |       a^{k-1}    |
     |     b     |     ab     | . . . |      a^{k-1}b    |
P =  |     .     |     .      |   .   |          .       |.
     |     .     |     .      |   .   |          .       |
     | b^{j-1}   | ab^{j-1}   | . . . | a^{k-1}b^{j-1}   |
     -                                                  -
```

The parity-check matrix H of the LDPC code can be obtained by replacing each element of matrix P, i.e. $P(s,t)$, with an identity matrix $I_{P(s,t)}$ of size $m \times m$.

The code rate R of the constructed code is given by

$$R \geq 1 - \frac{j}{k}$$

where the sign $\geq$ is due to the possible existence of some non linearly independent rows in H. For more details, refer to the paper by Tanner et al [TSS+04].

Example

```
gap> C := QCLDPCCodeFromGroup(7,2,3);
a linear [21,8,1..6]5..10 low-density parity-check code over GF(2)
gap> MinimumWeight(C);
6
gap> # The quasi-cyclic structure is obvious from the check matrix
gap> Display( CheckMat(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 . . . . .
gap> # This is the famous [155,64,20] quasi-cyclic LDPC codes
gap> C := QCLDPCCodeFromGroup(31,3,5);
a linear [155,64,1..24]24..77 low-density parity-check code over GF(2)
gap> # An example using non prime m, it may take a while to construct this code
gap> C := QCLDPCCodeFromGroup(356,4,8);
a linear [2848,1436,1..120]312..1412 low-density parity-check code over GF(2)
```

Chapter 6

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 6.1, we describe functions that take a code with certain parameters, modify it in some way and return a different code (see `ExtendedCode` (6.1.1), `PuncturedCode` (6.1.2), `EvenWeightSubcode` (6.1.3), `PermutedCode` (6.1.4), `ExpurgatedCode` (6.1.5), `AugmentedCode` (6.1.6), `RemovedElementsCode` (6.1.7), `AddedElementsCode` (6.1.8), `ShortenedCode` (6.1.9), `LengthenedCode` (6.1.10), `ResidueCode` (6.1.12), `ConstructionBCode` (6.1.13), `DualCode` (6.1.14), `ConversionFieldCode` (6.1.15), `ConstantWeightSubcode` (6.1.18), `StandardFormCode` (6.1.19) and `CosetCode` (6.1.17)). In Section 6.2, we describe functions that generate a new code out of two codes (see `DirectSumCode` (6.2.1), `UUVCode` (6.2.2), `DirectProductCode` (6.2.3), `IntersectionCode` (6.2.4) and `UnionCode` (6.2.5)).

6.1 Functions that Generate a New Code from a Given Code

6.1.1 ExtendedCode

▷ `ExtendedCode(`*`C[, i]`*`)` (function)

`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.

Example

```
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 PuncturedCode (6.1.2)). The function EvenWeightSubcode (see EvenWeightSubcode (6.1.3)) also returns a related code with only even weights, but without changing its word length.

6.1.2 PuncturedCode

▷ PuncturedCode(*C*) (function)

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.

This command can also be called with the syntax PuncturedCode(C, L). In this case, 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.

Example

```
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) ) ); # The dimension decreases from 4 to 3
a linear [3,3,1]0 punctured code
```

ExtendedCode extends the code again (see ExtendedCode (6.1.1)), although in general this does not result in the old code.

6.1.3 EvenWeightSubcode

▷ EvenWeightSubcode(*C*) (function)

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 $g(x)\cdot(x-1)$ (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`.

Example

```
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 `ExtendedCode` (6.1.1)).

6.1.4 PermutedCode

▷ `PermutedCode(C, L)` (function)

`PermutedCode` returns `C` after column permutations. `L` (in GAP disjoint cycle notation) 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 `AutomorphismGroup` (4.4.3)). In any case, the returned code is equivalent to `C` (see `IsEquivalent` (4.4.1)).

Example

```
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 );
(2,13,7,10,8,3,5,4,14)(12,15)
gap> C3 := PermutedCode( C1, p );
a linear [15,5,7]5 permuted code
gap> C2 = C3;
true
```

6.1.5 ExpurgatedCode

▷ `ExpurgatedCode(C, L)` (function)

`ExpurgatedCode` expurgates the 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.

Example

```
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 RemovedElementsCode (6.1.7)). For expurgating a code of all words of odd weight, use 'EvenWeightSubcode' (see EvenWeightSubcode (6.1.3)).

6.1.6 AugmentedCode

▷ AugmentedCode(*C*, *L*) (function)

AugmentedCode returns *C* after augmenting. *C* must be a linear code, *L* must be a list of codeword inputs. 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.

This command can also be called with the syntax 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.

Example

```
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
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 AddedElementsCode (6.1.8)).

6.1.7 RemovedElementsCode

▷ RemovedElementsCode(*C*, *L*) (function)

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.

Example

```
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 );     # C2 is not linear, so the minimum weight does not have to be equ
3
```

Adding elements to a code is done by the function AddedElementsCode (see AddedElementsCode (6.1.8)). To remove codewords from the base of a linear code, use ExpurgatedCode (see ExpurgatedCode (6.1.5)).

6.1.8 AddedElementsCode

▷ AddedElementsCode(*C*, *L*) (function)

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.

Example

```
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 RemovedElementsCode (6.1.7)). To add elements to the base of a linear code, use AugmentedCode (see AugmentedCode (6.1.6)).

6.1.9 ShortenedCode

▷ ShortenedCode(*C*[, *L*]) (function)

ShortenedCode(C) returns the 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.

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*.

This command can also be called using the syntax 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*.

Example

```
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
gap> C3 := ElementsCode( ["1000", "1101", "0011" ], GF(2) );
a (4,3,1..4)2 user defined unrestricted code over GF(2)
gap> MinimumDistance( C3 );
2
gap> C4 := ShortenedCode( C3 );
a (3,2,2..3)1..2 shortened code
gap> AsSSortedList( C4 );
[ [ 0 0 0 ], [ 1 0 1 ] ]
gap> C5 := HammingCode( 5, GF(2) );
a linear [31,26,3]1 Hamming (5,2) code over GF(2)
gap> C6 := ShortenedCode( C5, [ 1, 2, 3 ] );
a linear [28,23,3]2 shortened code
gap> OptimalityLinearCode( C6 );
0
```

The function LengthenedCode lengthens the code again (only for linear codes), see LengthenedCode (6.1.10). In general, this is not exactly the inverse function.

6.1.10 LengthenedCode

▷ LengthenedCode(*C[, i]*) (function)

LengthenedCode(C) returns the code *C* lengthened. *C* must be a linear code. First, the all-ones vector is added to the generator matrix (see AugmentedCode (6.1.6)). If the all-ones vector was already a codeword, nothing happens to the code. Then, the code is extended *i* times (see ExtendedCode (6.1.1)). *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.

Example

```
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 ShortenedCode (6.1.9). In general, this is not exactly the inverse function.

6.1.11 SubCode

▷ SubCode(C[, s]) (function)

This function SubCode returns a subcode of C by taking the first $k-s$ rows of the generator matrix of C, where k is the dimension of C. The integer s may be omitted and in this case it is assumed as 1.

Example

```
gap> C := BCHCode(31,11);
a cyclic [31,11,11]7..11 BCH code, delta=11, b=1 over GF(2)
gap> S1:= SubCode(C);
a linear [31,10,11]7..13 subcode
gap> WeightDistribution(S1);
[ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 120, 190, 0, 0, 272, 255, 0, 0, 120, 66,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
gap> S2:= SubCode(C, 8);
a linear [31,3,11]14..20 subcode
gap> History(S2);
[ "a linear [31,3,11]14..20 subcode of",
  "a cyclic [31,11,11]7..11 BCH code, delta=11, b=1 over GF(2)" ]
gap> WeightDistribution(S2);
[ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0 ]
```

6.1.12 ResidueCode

▷ ResidueCode(C[, c]) (function)

The function ResidueCode takes a codeword c of C (if c is omitted, a codeword of minimal weight is used). 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. C must be a linear code and c must be non-zero. If c is not in then no change is made to C.

Example

```
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
```

6.1.13 ConstructionBCode

▷ ConstructionBCode(C) (function)

The function ConstructionBCode takes a binary linear code C and calculates the minimum distance of the dual of C (see DualCode (6.1.14)). 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.

Example

```
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> bounds := BoundsMinimumDistance( 24, 9, GF(2) );;
gap> [ bounds.lowerBound, bounds.upperBound ];  # so C2 is optimal
[ 8, 8 ]
```

6.1.14 DualCode

▷ DualCode(C) (function)

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*.

Example

```
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 dual code
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
```

6.1.15 ConversionFieldCode

▷ ConversionFieldCode(C) (function)

ConversionFieldCode returns the code obtained from 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 $(n\cdot m,M,d_2)$ code, where $d_2 > d_1$.

See also HorizontalConversionFieldMat (7.3.10).

Example

```
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 ] ]
```

6.1.16 TraceCode

▷ `TraceCode(C)` (function)

Input: `C` is a linear code defined over an extension E of `F` (F is the "base field")

Output: The linear code generated by $Tr_{E/F}(c)$, for all $c \in C$.

`TraceCode` returns the image of the code `C` under the trace map. If the field of `C` is $GF(q^m)$, the returned code has field $GF(q)$.

Very slow. It does not seem to be easy to related the parameters of the trace code to the original except in the "Galois closed" case.

Example

```
gap> C:=HammingCode(2,GF(4)); MinimumDistance(C);
a linear [5,3,3]1 Hamming (2,4) code over GF(4)
3
gap> trC:=TraceCode(C,GF(2)); MinimumDistance(trC);
a linear [5,4,1]1 user defined unrestricted code over GF(2)
1
```

6.1.17 CosetCode

▷ `CosetCode(C, w)` (function)

`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 4.2.

Example

```
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));  # All elements of the coset have the same syn
[ [ 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 ] ]
```

6.1.18 ConstantWeightSubcode

▷ `ConstantWeightSubcode(C, w)` (function)

`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.

This command also can be called with the syntax `ConstantWeightSubcode(C)` In this format, `ConstantWeightSubcode` returns the subcode of C consisting of all minimum weight codewords of C.

`ConstantWeightSubcode` first checks if Leon's binary `wtdist` exists on your computer (in the default directory). If it does, then this program is called. Otherwise, the constant weight subcode is computed using a GAP program which checks each codeword in C to see if it is of the desired weight.

Example

```
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 ]
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 ]
```

6.1.19 StandardFormCode

▷ `StandardFormCode(C)` (function)

`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 `Display` (4.6.3)). 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.

Example

```
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 1 0 ], [ 0 0 1 1 ] ]
gap> Codeword( GeneratorMat( StandardFormCode( G ) ) );
[ [ 1 0 0 1 ], [ 0 1 0 1 ] ]
```

6.1.20 PiecewiseConstantCode

▷ `PiecewiseConstantCode(part, wts[, F])` (function)

`PiecewiseConstantCode` returns a code with length $n = \sum n_i$, where `part`=$[n_1,\ldots,n_k]$. `wts` is a list of *constraints* $w = (w_1,...,w_k)$, each of length k, where $0 \leq w_i \leq n_i$. 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`, i.e., each c_i is a subword of length n_i) is in the resulting code if and only if, for some constraint $w \in$ `wts`, $\|c_i\| = w_i$ for all $1 \leq i \leq k$, where $\|...\|$ denotes the Hamming weight.

An example might make things clearer:

Example

```
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.

6.2 Functions that Generate a New Code from Two or More Given Codes

6.2.1 DirectSumCode

▷ `DirectSumCode(C1, C2)` (function)

`DirectSumCode` returns the direct sum of codes `C1` and `C2`. The direct sum code consists of every codeword of `C1` concatenated by every codeword of `C2`. Therefore, if `Ci` 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 `C1` and `C2` 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 Section 4.2). Another often used method is the 'u, u+v'-construction, described in `UUVCode` (6.2.2).

Example

```
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
```

6.2.2 UUVCode

▷ `UUVCode(C1, C2)` (function)

`UUVCode` returns the so-called $(u\|u+v)$ construction applied to `C1` and `C2`. The resulting code consists of every codeword u of `C1` concatenated by the sum of u and every codeword v of `C2`. If `C1` and `C2` have different word lengths, sufficient zeros are added to the shorter code to make this sum possible. If `Ci` is a (n_i, M_i, d_i) code, the result is an $(n_1 + max(n_1, n_2), M_1 \cdot M_2, min(2 \cdot d_1, d_2))$ code.

If both `C1` and `C2` 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 `DirectSumCode` (6.2.1)).

Example

```
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
```

6.2.3 DirectProductCode

▷ `DirectProductCode(C1, C2)` (function)

`DirectProductCode` returns the direct product of codes `C1` and `C2`. Both must be linear codes. Suppose `Ci` has generator matrix G_i. The direct product of `C1` and `C2` then has the Kronecker product of G_1 and G_2 as the generator matrix (see the GAP command `KroneckerProduct`).

If `Ci` is a $[n_i, k_i, d_i]$ code, the direct product then is an $[n_1 \cdot n_2, k_1 \cdot k_2, d_1 \cdot d_2]$ code.

Example

```
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
```

6.2.4 IntersectionCode

▷ `IntersectionCode(C1, C2)` (function)

`IntersectionCode` returns the intersection of codes `C1` and `C2`. This code consists of all codewords that are both in `C1` and `C2`. If both codes are linear, the result is also linear. If both are cyclic, the result is also cyclic.

Example

```
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
```

The *hull* of a linear code is the intersection of the code with its dual code. In other words, the hull of C is `IntersectionCode(C, DualCode(C))`.

6.2.5 UnionCode

▷ `UnionCode(C1, C2)` (function)

`UnionCode` returns the union of codes `C1` and `C2`. This code consists of the union of all codewords of `C1` and `C2` 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 `AddedElementsCode` (6.1.8). If both arguments are cyclic, the result is also cyclic.

Example

```
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
```

6.2.6 ExtendedDirectSumCode

▷ `ExtendedDirectSumCode(L, B, m)` (function)

The extended direct sum construction is described in section V of Graham and Sloane [GS85]. 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

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

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 $k = mk_L + (k_B - k_L)$.

Example

```
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
```

6.2.7 AmalgamatedDirectSumCode

▷ `AmalgamatedDirectSumCode(c1, c2[, check])` (function)

`AmalgamatedDirectSumCode` returns the amalgamated direct sum of the codes *c1* and *c2*. The amalgamated 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 $M \leq M_1 \cdot M_2/2$.

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 *c1* is a normal code (see `IsNormalCode` (7.4.5)) with the last coordinate acceptable (see `IsCoordinateAcceptable` (7.4.3)), and *c2* is a normal code with the first coordinate acceptable, then the covering radius of the new code is $r \leq r_1 + r_2$. 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.

Example

```
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 := AmalgamatedDirectSumCode( c, d );;
gap> MinimumDistance( f );;
gap> CoveringRadius( f );;
gap> f;
a linear [22,8,3]7 amalgamated direct sum code
```

6.2.8 BlockwiseDirectSumCode

▷ `BlockwiseDirectSumCode(C1, L1, C2, L2)` (function)

`BlockwiseDirectSumCode` returns a subcode of the direct sum of *C1* and *C2*. The fields of *C1* and *C2* must be same. The lists *L1* and *L2* are two equally long with elements from the ambient vector spaces of *C1* and *C2*, respectively, *or* *L1* and *L2* are two equally long lists containing codes. The union of the codes in *L1* and *L2* must be *C1* and *C2*, respectively.

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

$$bds = \bigcup_{1 \leq i \leq \ell} (C_1 + (L_1)_i) \oplus (C_2 + (L_2)_i),$$

where ℓ is the length of `L1` and `L2`, and $\oplus$ is the direct sum.

In the second case, it is defined as

$$bds = \bigcup_{1 \le i \le \ell} ((L_1)_i \oplus (L_2)_i).$$

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

Example

```
gap> C1 := HammingCode( 3, GF(2) );;
gap> C2 := EvenWeightSubcode( WholeSpaceCode( 6, GF(2) ) );;
gap> BlockwiseDirectSumCode( C1, [[ 0,0,0,0,0,0,0 ],[ 1,0,1,0,1,0,0 ]],
> C2, [[ 0,0,0,0,0,0 ],[ 1,0,1,0,1,0 ]] );
a (13,1024,1..13)1..2 blockwise direct sum code
```

6.2.9 ConstructionXCode

▷ `ConstructionXCode(C, A)` (function)

Consider a list of j linear codes of the same length N over the same field F, $C = \{C_1, C_2, \ldots, C_j\}$, where the parameter of the ith code is $C_i = [N, K_i, D_i]$ and $C_j \subset C_{j-1} \subset \ldots \subset C_2 \subset C_1$. Consider a list of $j-1$ auxiliary linear codes of the same field F, $A = \{A_1, A_2, \ldots, A_{j-1}\}$ where the parameter of the ith code A_i is $[n_i, k_i = (K_i - K_{i+1}), d_i]$, an $[n, K_1, d]$ linear code over field F can be constructed where $n = N + \sum_{i=1}^{j-1} n_i$, and $d = \min\{D_j, D_{j-1} + d_{j-1}, D_{j-2} + d_{j-2} + d_{j-1}, \ldots, D_1 + \sum_{i=1}^{j-1} d_i\}$.

For more information on Construction X, refer to [SRC72].

Example

```
gap> C1 := BCHCode(127, 43);
a cyclic [127,29,43]31..59 BCH code, delta=43, b=1 over GF(2)
gap> C2 := BCHCode(127, 47);
a cyclic [127,22,47..51]36..63 BCH code, delta=47, b=1 over GF(2)
gap> C3 := BCHCode(127, 55);
a cyclic [127,15,55]41..62 BCH code, delta=55, b=1 over GF(2)
gap> G1 := ShallowCopy( GeneratorMat(C2) );;
gap> Add(G1, GeneratorMat(C1)[23]);;
gap> C1 := GeneratorMatCode(G1, GF(2));
a linear [127,23,1..43]35..63 code defined by generator matrix over GF(2)
gap> MinimumDistance(C1);
43
gap> C := [ C1, C2, C3 ];
[ a linear [127,23,43]35..63 code defined by generator matrix over GF(2),
  a cyclic [127,22,47..51]36..63 BCH code, delta=47, b=1 over GF(2),
  a cyclic [127,15,55]41..62 BCH code, delta=55, b=1 over GF(2) ]
gap> IsSubset(C[1], C[2]);
true
gap> IsSubset(C[2], C[3]);
true
gap> A := [ RepetitionCode(4, GF(2)), EvenWeightSubcode( QRCode(17, GF(2)) ) ];
[ a cyclic [4,1,4]2 repetition code over GF(2),
  a cyclic [17,8,6]3..6 even weight subcode ]
gap> CX := ConstructionXCode(C, A);
a linear [148,23,53]43..74 Construction X code
gap> History(CX);
```

```
[ "a linear [148,23,53]43..74 Construction X code of",
  "Base codes: [ a cyclic [127,15,55]41..62 BCH code, delta=55, b=1 over GF(2)\
, a cyclic [127,22,47..51]36..63 BCH code, delta=47, b=1 over GF(2), a linear \
[127,23,43]35..63 code defined by generator matrix over GF(2) ]",
  "Auxiliary codes: [ a cyclic [4,1,4]2 repetition code over GF(2), a cyclic [\
17,8,6]3..6 even weight subcode ]" ]
```

6.2.10 ConstructionXXCode

▷ `ConstructionXXCode(C1, C2, C3, A1, A2)` (function)

Consider a set of linear codes over field F of the same length, n, $C_1 = [n,k_1,d_1]$, $C_2 = [n,k_2,d_2]$ and $C_3 = [n,k_3,d_3]$ such that $C_2 \subset C_1$, $C_3 \subset C_1$ and $C_4 = C_2 \cap C_3$. Given two auxiliary codes $A_1 = [n_1, k_1 - k_2, e_1]$ and $A_2 = [n_2, k_1 - k_3, e_2]$ over the same field F, there exists an $[n+n_1+n_2, k_1, d]$ linear code C_{XX} over field F, where $d = \min\{d_4, d_3+e_1, d_2+e_2, d_1+e_1+e_2\}$.

The codewords of C_{XX} can be partitioned into three sections $(v \parallel a \parallel b)$ where v has length n, a has length n_1 and b has length n_2. A codeword from Construction XX takes the following form:

- $(v \parallel 0 \parallel 0)$ if $v \in C_4$
- $(v \parallel a_1 \parallel 0)$ if $v \in C_3 \backslash C_4$
- $(v \parallel 0 \parallel a_2)$ if $v \in C_2 \backslash C_4$
- $(v \parallel a_1 \parallel a_2)$ otherwise

For more information on Construction XX, refer to [All84].

Example

```
gap> a := PrimitiveRoot(GF(32));
Z(2^5)
gap> f0 := MinimalPolynomial( GF(2), a^0 );
x+Z(2)^0
gap> f1 := MinimalPolynomial( GF(2), a^1 );
x^5+x^2+Z(2)^0
gap> f5 := MinimalPolynomial( GF(2), a^5 );
x^5+x^4+x^2+x+Z(2)^0
gap> C2 := CheckPolCode( f0 * f1, 31, GF(2) );; MinimumDistance(C2);; Display(C2);
a cyclic [31,6,15]10..13 code defined by check polynomial over GF(2)
gap> C3 := CheckPolCode( f0 * f5, 31, GF(2) );; MinimumDistance(C3);; Display(C3);
a cyclic [31,6,15]10..13 code defined by check polynomial over GF(2)
gap> C1 := UnionCode(C2, C3);; MinimumDistance(C1);; Display(C1);
a linear [31,11,11]7..11 union code of
U: a cyclic [31,6,15]10..13 code defined by check polynomial over GF(2)
V: a cyclic [31,6,15]10..13 code defined by check polynomial over GF(2)
gap> A1 := BestKnownLinearCode( 10, 5, GF(2) );
a linear [10,5,4]2..4 shortened code
gap> A2 := DualCode( RepetitionCode(6, GF(2)) );
a cyclic [6,5,2]1 dual code
gap> CXX:= ConstructionXXCode(C1, C2, C3, A1, A2 );
a linear [47,11,15..17]13..23 Construction XX code
gap> MinimumDistance(CXX);
17
```

```
gap> History(CXX);
[ "a linear [47,11,17]13..23 Construction XX code of",
  "C1: a cyclic [31,11,11]7..11 union code",
  "C2: a cyclic [31,6,15]10..13 code defined by check polynomial over GF(2)",
  "C3: a cyclic [31,6,15]10..13 code defined by check polynomial over GF(2)",
  "A1: a linear [10,5,4]2..4 shortened code",
  "A2: a cyclic [6,5,2]1 dual code" ]
```

6.2.11 BZCode

▷ BZCode(*O, I*) (function)

Given a set of outer codes of the same length $O_i = [N, K_i, D_i]$ over GF(q^{e_i}), where $i = 1, 2, \ldots, t$ and a set of inner codes of the same length $I_i = [n, k_i, d_i]$ over GF(q), BZCode returns a Blokh-Zyablov multilevel concatenated code with parameter $[n \times N, \sum_{i=1}^{t} e_i \times K_i, \min_{i=1,\ldots,t}\{d_i \times D_i\}]$ over GF(q).

Note that the set of inner codes must satisfy chain condition, i.e. $I_1 = [n, k_1, d_1] \subset I_2 = [n, k_2, d_2] \subset \ldots \subset I_t = [n, k_t, d_t]$ where $0 = k_0 < k_1 < k_2 < \ldots < k_t$. The dimension of the inner codes must satisfy the condition $e_i = k_i - k_{i-1}$, where GF(q^{e_i}) is the field of the ith outer code.

For more information on Blokh-Zyablov multilevel concatenated code, refer to [Bro98].

6.2.12 BZCodeNC

▷ BZCodeNC(*O, I*) (function)

This function is the same as BZCode, except this version is faster as it does not estimate the covering radius of the code. Users are encouraged to use this version unless you are working on very small codes.

Example

```
gap> #
gap> # Binary code
gap> #
gap> O := [ CyclicMDSCode(2,3,7), BestKnownLinearCode(9,5,GF(2)), CyclicMDSCode(2,3,4) ];
[ a cyclic [9,7,3]1 MDS code over GF(8), a linear [9,5,3]2..3 shortened code,
  a cyclic [9,4,6]4..5 MDS code over GF(8) ]
gap> A := ExtendedCode( HammingCode(3,GF(2)) );;
gap> I := [ SubCode(A), A, DualCode( RepetitionCode(8, GF(2)) ) ];
[ a linear [8,3,4]3..4 subcode, a linear [8,4,4]2 extended code,
  a cyclic [8,7,2]1 dual code ]
gap> C := BZCodeNC(O, I);
a linear [72,38,12]0..72 Blokh Zyablov concatenated code
gap> #
gap> # Non binary code
gap> #
gap> O2 := ExtendedCode(GoppaCode(ConwayPolynomial(5,2), Elements(GF(5))));;
gap> O3 := ExtendedCode(GoppaCode(ConwayPolynomial(5,3), Elements(GF(5))));;
gap> O1 := DualCode( O3 );;
gap> MinimumDistance(O1);; MinimumDistance(O2);; MinimumDistance(O3);;
gap> Cy := CyclicCodes(5, GF(5));;
gap> for i in [4, 5] do MinimumDistance(Cy[i]);; od;
gap> O  := [ O1, O2, O3 ];
```

```
[ a linear [6,4,3]1 dual code, a linear [6,3,4]2..3 extended code,
  a linear [6,2,5]3..4 extended code ]
gap> I  := [ Cy[5], Cy[4], Cy[3] ];
[ a cyclic [5,1,5]3..4 enumerated code over GF(5),
  a cyclic [5,2,4]2..3 enumerated code over GF(5),
  a cyclic [5,3,1..3]2 enumerated code over GF(5) ]
gap> C  := BZCodeNC( 0, I );
a linear [30,9,5..15]0..30 Blokh Zyablov concatenated code
gap> MinimumDistance(C);
15
gap> History(C);
[ "a linear [30,9,15]0..30 Blokh Zyablov concatenated code of",
  "Inner codes: [ a cyclic [5,1,5]3..4 enumerated code over GF(5), a cyclic [5\
,2,4]2..3 enumerated code over GF(5), a cyclic [5,3,1..3]2 enumerated code ove\
r GF(5) ]",
  "Outer codes: [ a linear [6,4,3]1 dual code, a linear [6,3,4]2..3 extended c\
ode, a linear [6,2,5]3..4 extended code ]" ]
```

Chapter 7

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 7.1), functions that determine bounds on the size and covering radius of codes (Section 7.2), functions that work with special matrices GUAVA needs for several codes (see Section 7.3), and constructing codes or performing calculations with codes (see Section 7.5).

7.1 Distance 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. It contains entries for word lengths less than or equal to 257, 243 and 256 for codes over $GF(2)$, $GF(3)$ and $GF(4)$ respectively. These entries were obtained from Brouwer's tables as of 11 May 2006. For the latest information, please see A. E. Brouwer's tables [Bro06] on the internet.

Firstly, we describe functions that compute specific upper bounds on the code size (see `UpperBoundSingleton` (7.1.1), `UpperBoundHamming` (7.1.2), `UpperBoundJohnson` (7.1.3), `UpperBoundPlotkin` (7.1.4), `UpperBoundElias` (7.1.5) and `UpperBoundGriesmer` (7.1.6)).

Next we describe a function that computes GUAVA's best upper bound on the code size (see `UpperBound` (7.1.8)).

Then we describe two functions that compute a lower and upper bound on the minimum distance of a code (see `LowerBoundMinimumDistance` (7.1.9) and `UpperBoundMinimumDistance` (7.1.12)).

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 `BoundsMinimumDistance` (7.1.13)).

7.1.1 UpperBoundSingleton

▷ `UpperBoundSingleton(n, d, q)` (function)

`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 `ShortenedCode` (6.1.9)). Thus

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

Codes that meet this bound are called *maximum distance separable* (see `IsMDSCode` (4.3.7)).

Example

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

7.1.2 UpperBoundHamming

▷ `UpperBoundHamming(n, d, q)` (function)

The Hamming bound (also known as the *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 maximum number of codewords and $V(n,e)$ is equal to the contents of a ball of radius e (see `SphereContent` (7.5.5)). This bound is useful for small values of `d`. Codes for which equality holds are called *perfect* (see `IsPerfectCode` (4.3.6)).

Example

```
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
```

7.1.3 UpperBoundJohnson

▷ `UpperBoundJohnson(n, d)` (function)

The Johnson bound is an improved version of the Hamming bound (see `UpperBoundHamming` (7.1.2)). 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.

Example

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

7.1.4 UpperBoundPlotkin

▷ `UpperBoundPlotkin(n, d, q)` (function)

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:

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

where 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 $d \langle (1-1/q)n$ is covered.

Example

```
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 ]
```

7.1.5 UpperBoundElias

▷ `UpperBoundElias(n, d, q)` (function)

The Elias bound is an improvement of the Plotkin bound (see `UpperBoundPlotkin` (7.1.4)) 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.

Example

```
gap> UpperBoundPlotkin( 16, 3, 2 );
12288
gap> UpperBoundElias( 16, 3, 2 );
10280
gap> UpperBoundElias( 20, 10, 3 );
16255
```

7.1.6 UpperBoundGriesmer

▷ `UpperBoundGriesmer(n, d, q)` (function)

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 process until a trivial code is left in the end.

Example

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

7.1.7 IsGriesmerCode

▷ `IsGriesmerCode(C)` (function)

`IsGriesmerCode` returns 'true' if a linear code `C` is a Griesmer code, and 'false' otherwise. A code is called *Griesmer* if its length satisfies

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

Example

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

7.1.8 UpperBound

▷ `UpperBound(n, d, q)` (function)

`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 `UpperBoundSingleton` (7.1.1)), Hamming (see `UpperBoundHamming` (7.1.2)), Johnson (see `UpperBoundJohnson` (7.1.3)), Plotkin (see `UpperBoundPlotkin` (7.1.4)) and Elias (see `UpperBoundElias` (7.1.5)). If the code is binary, $A(n,2\cdot\ell-1) = A(n+1,2\cdot\ell)$, so the `UpperBound` takes the minimum of the values obtained from all methods for the parameters $(n,2\cdot\ell-1)$ and $(n+1,2\cdot\ell)$.

Example

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

7.1.9 LowerBoundMinimumDistance

▷ `LowerBoundMinimumDistance(C)` (function)

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

This command can also be called using the syntax `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 7.1.13.

Example

```
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 );     # designed distance is lower bound for minimum distance
7
gap> LowerBoundMinimumDistance( 45, 23, GF(2) );
10
```

7.1.10 LowerBoundGilbertVarshamov

▷ `LowerBoundGilbertVarshamov(n, d, q)` (function)

This is the lower bound on the size of a linear code due (independently) to Gilbert and Varshamov. It says that for each `n` and `d`, there exists a linear code having length n and minimum distance d at least of size q^k, where k is the largest integer such that $q^k < q^n/\texttt{SphereContent}(n-1,d-2,GF(q))$.

Example

```
gap> LowerBoundGilbertVarshamov(24,8,2);
64
gap> LowerBoundGilbertVarshamov(7,3,2);
16
gap> LowerBoundMinimumDistance(7,4,2);
3
gap> LowerBoundGilbertVarshamov(3,3,2);
2
gap> LowerBoundMinimumDistance(3,3,2);
1
gap> LowerBoundGilbertVarshamov(25,10,2);
16
```

7.1.11 LowerBoundSpherePacking

▷ `LowerBoundSpherePacking(n, d, q)` (function)

This is the (weaker) Gilbert-Varshamov bound valid for unrestricted codes over an alphabet of size `q` (where `q` is an integer > 1). It says that for each `n` and `r`, there exists an unrestricted code at least of size $q^n/\texttt{SphereContent}(n,d,GF(q))$ minimum distance d.

Example

```
gap> LowerBoundSpherePacking(3,2,2);
2
gap> LowerBoundSpherePacking(3,3,2);
1
```

7.1.12 UpperBoundMinimumDistance

▷ `UpperBoundMinimumDistance(C)` (function)

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 least weight among the rows of a reduced generator matrix, and the value from the table (see 7.1.13).

This command can also be called using the syntax `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 7.1.13.

Example

```
gap> C := BCHCode( 45, 7 );;
gap> UpperBoundMinimumDistance( C );
9
gap> UpperBoundMinimumDistance( 45, 23, GF(2) );
10
```

7.1.13 BoundsMinimumDistance

▷ `BoundsMinimumDistance(n, k, F)` (function)

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 explanation 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 $q = 2(n \leq 257), 3(n \leq 243), 4(n \leq 256)$. (Current as of 11 May 2006.) These tables were derived from the table of Brouwer. (See [Bro06], https://www.codetables.de for the most recent data.) 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 `BestKnownLinearCode` (5.2.14)) to construct a code with minimum distance equal to the lower bound.

Example

```
gap> bounds := BoundsMinimumDistance( 7, 3 );; DisplayBoundsInfo( bounds ); # The lower bound is
an optimal linear [7,3,d] code over GF(2) has d=4
--------------------------------------------------------------------------------
Lb(7,3)=4, by shortening of:
Lb(8,4)=4, by the u|u+v construction applied to C1 [4,3,2] and C2 [4,1,4]:
Lb(4,3)=2, dual of the repetition code
```

```
Lb(4,1)=4, repetition code
--------------------------------------------------------------------------------
Ub(7,3)=4, by the Griesmer bound
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)
```

7.2 Covering radius bounds on codes

7.2.1 BoundsCoveringRadius

▷ `BoundsCoveringRadius(C)` (function)

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

If the covering radius of C is known, a list of length 1 is returned. `BoundsCoveringRadius` makes use of the functions `GeneralLowerBoundCoveringRadius` and `GeneralUpperBoundCoveringRadius`.

Example

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

7.2.2 IncreaseCoveringRadiusLowerBound

▷ `IncreaseCoveringRadiusLowerBound(C[, stopdist][, startword])` (function)

`IncreaseCoveringRadiusLowerBound` tries to increase the lower bound of the covering radius of C. 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 C 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 *stopdist* 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 algorithm 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.

Example

```
gap> C:=RandomLinearCode(10,5,GF(2));
a  [10,5,?] randomly generated code over GF(2)
gap> IncreaseCoveringRadiusLowerBound(C,10);
Number of runs: 1000  best distance so far: 3
Number of runs: 2000  best distance so far: 3
Number of changes: 100
Number of runs: 3000  best distance so far: 3
Number of runs: 4000  best distance so far: 3
Number of runs: 5000  best distance so far: 3
Number of runs: 6000  best distance so far: 3
Number of runs: 7000  best distance so far: 3
Number of changes: 200
Number of runs: 8000  best distance so far: 3
Number of runs: 9000  best distance so far: 3
Number of runs: 10000  best distance so far: 3
Number of changes: 300
Number of runs: 11000  best distance so far: 3
Number of runs: 12000  best distance so far: 3
Number of runs: 13000  best distance so far: 3
Number of changes: 400
Number of runs: 14000  best distance so far: 3
user interrupt at...
#
# used ctrl-c to break out of execution
#
... called from
IncreaseCoveringRadiusLowerBound( code, -1, current ) called from
 function( arguments ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> current;
[ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ]
brk>
gap> CoveringRadius(C);
3
```

7.2.3 ExhaustiveSearchCoveringRadius

▷ `ExhaustiveSearchCoveringRadius(C)` (function)

`ExhaustiveSearchCoveringRadius` does an exhaustive search to find the covering radius of `C`. 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.

Example

```
gap> C:=BestKnownLinearCode(10,5,GF(2));
a linear [10,5,4]2..4 shortened code
gap> ExhaustiveSearchCoveringRadius(C);
Trying 3 ...
[ 3, 4 ]
gap> CoveringRadius(C);
3
```

7.2.4 GeneralLowerBoundCoveringRadius

▷ `GeneralLowerBoundCoveringRadius(C)` (function)

`GeneralLowerBoundCoveringRadius` returns a lower bound on the covering radius of `C`. 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.

Example

```
gap> C:=BestKnownLinearCode(10,5,GF(2));
a linear [10,5,4]2..4 shortened code
gap> GeneralLowerBoundCoveringRadius(C);
2
gap> CoveringRadius(C);
3
```

7.2.5 GeneralUpperBoundCoveringRadius

▷ `GeneralUpperBoundCoveringRadius(C)` (function)

`GeneralUpperBoundCoveringRadius` returns an upper bound on the covering radius of `C`. 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).

Example

```
gap> C:=BestKnownLinearCode(10,5,GF(2));
a linear [10,5,4]2..4 shortened code
gap> GeneralUpperBoundCoveringRadius(C);
4
gap> CoveringRadius(C);
3
```

7.2.6 LowerBoundCoveringRadiusSphereCovering

▷ `LowerBoundCoveringRadiusSphereCovering(n, M[, F], false)` (function)

This command can also be called using the syntax `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:

$$M \cdot V_q(n,r) \geq q^n$$

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

Example

```
gap> C:=BestKnownLinearCode(10,5,GF(2));
a linear [10,5,4]2..4 shortened code
gap> Size(C);
32
gap> CoveringRadius(C);
3
gap> LowerBoundCoveringRadiusSphereCovering(10,32,GF(2),false);
2
gap> LowerBoundCoveringRadiusSphereCovering(10,3,GF(2),true);
6
```

7.2.7 LowerBoundCoveringRadiusVanWee1

▷ `LowerBoundCoveringRadiusVanWee1(n, M[, F], false)` (function)

This command can also be called using the syntax `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:

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

Example

```
gap> C:=BestKnownLinearCode(10,5,GF(2));
a linear [10,5,4]2..4 shortened code
gap> Size(C);
32
gap> CoveringRadius(C);
3
gap> LowerBoundCoveringRadiusVanWee1(10,32,GF(2),false);
2
gap> LowerBoundCoveringRadiusVanWee1(10,3,GF(2),true);
6
```

7.2.8 LowerBoundCoveringRadiusVanWee2

▷ `LowerBoundCoveringRadiusVanWee2(n, M, false)` (function)

This command can also be called using the syntax `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:

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

where

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

Example

```
gap> C:=BestKnownLinearCode(10,5,GF(2));
a linear [10,5,4]2..4 shortened code
gap> Size(C);
32
gap> CoveringRadius(C);
3
gap> LowerBoundCoveringRadiusVanWee2(10,32,false);
2
gap> LowerBoundCoveringRadiusVanWee2(10,3,true);
7
```

7.2.9 LowerBoundCoveringRadiusCountingExcess

▷ `LowerBoundCoveringRadiusCountingExcess(n, M, false)` (function)

This command can also be called with `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:

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

where

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

and

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

Example

```
gap> C:=BestKnownLinearCode(10,5,GF(2));
a linear [10,5,4]2..4 shortened code
```

```
gap> Size(C);
32
gap> CoveringRadius(C);
3
gap> LowerBoundCoveringRadiusCountingExcess(10,32,false);
0
gap> LowerBoundCoveringRadiusCountingExcess(10,3,true);
7
```

7.2.10 LowerBoundCoveringRadiusEmbedded1

▷ `LowerBoundCoveringRadiusEmbedded1(n, M, false)` (function)

This command can also be called with `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:

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

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.

Example

```
gap> C:=BestKnownLinearCode(10,5,GF(2));
a linear [10,5,4]2..4 shortened code
gap> Size(C);
32
gap> CoveringRadius(C);
3
gap> LowerBoundCoveringRadiusEmbedded1(10,32,false);
2
gap> LowerBoundCoveringRadiusEmbedded1(10,3,true);
7
```

7.2.11 LowerBoundCoveringRadiusEmbedded2

▷ `LowerBoundCoveringRadiusEmbedded2(n, M, false)` (function)

This command can also be called with `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:

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

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.

Example

```
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> CoveringRadius(C);
5
gap> LowerBoundCoveringRadiusEmbedded2(10,32,false);
2
gap> LowerBoundCoveringRadiusEmbedded2(10,3,true);
7
```

7.2.12 LowerBoundCoveringRadiusInduction

▷ LowerBoundCoveringRadiusInduction(*n*, *r*) (function)

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.

Example

```
gap> C:=BCHCode(15,7,GF(2));
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> CoveringRadius(C);
5
gap> LowerBoundCoveringRadiusInduction(15,6);
7
```

7.2.13 UpperBoundCoveringRadiusRedundancy

▷ UpperBoundCoveringRadiusRedundancy(*C*) (function)

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

Example

```
gap> C:=BCHCode(15,7,GF(2));
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> CoveringRadius(C);
5
gap> UpperBoundCoveringRadiusRedundancy(C);
10
```

7.2.14 UpperBoundCoveringRadiusDelsarte

▷ `UpperBoundCoveringRadiusDelsarte(C)` (function)

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

This is described in Theorem 11.3.3 of [HP03].

Example

```
gap> C:=BCHCode(15,7,GF(2));
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> CoveringRadius(C);
5
gap> UpperBoundCoveringRadiusDelsarte(C);
5
```

7.2.15 UpperBoundCoveringRadiusStrength

▷ `UpperBoundCoveringRadiusStrength(C)` (function)

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

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.

Example

```
gap> C:=BCHCode(15,7,GF(2));
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> CoveringRadius(C);
5
gap> UpperBoundCoveringRadiusStrength(C);
7
```

7.2.16 UpperBoundCoveringRadiusGriesmerLike

▷ `UpperBoundCoveringRadiusGriesmerLike(C)` (function)

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

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

Example

```
gap> C:=BCHCode(15,7,GF(2));
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> CoveringRadius(C);
5
gap> UpperBoundCoveringRadiusGriesmerLike(C);
6
```

7.2.17 UpperBoundCoveringRadiusCyclicCode

▷ `UpperBoundCoveringRadiusCyclicCode(C)` (function)

This function returns an upper bound for the covering radius of `C`, 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 `C`.

Example

```
gap> C:=CyclicCodes(15,GF(2))[3];
a cyclic [15,12,1..2]1..3 enumerated code over GF(2)
gap> CoveringRadius(C);
3
gap> UpperBoundCoveringRadiusCyclicCode(C);
3
```

7.3 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 `KrawtchoukMat` (7.3.1), `GrayMat` (7.3.2), `SylvesterMat` (7.3.3), `HadamardMat` (7.3.4) and `MOLS` (7.3.11)).

Next we describe two functions regarding a standard form of matrices (see `PutStandardForm` (7.3.6) and `IsInStandardForm` (7.3.7)).

Then we describe functions that return a matrix after a manipulation (see `PermutedCols` (7.3.8), `VerticalConversionFieldMat` (7.3.9) and `HorizontalConversionFieldMat` (7.3.10)).

Finally, we describe functions that do some tests on matrices (see `IsLatinSquare` (7.3.12) and `AreMOLS` (7.3.13)).

7.3.1 KrawtchoukMat

▷ `KrawtchoukMat(n, q)` (function)

`KrawtchoukMat` returns the $n+1$ by $n+1$ matrix $K=(k_{ij})$ defined by $k_{ij}=K_i(j)$ for $i,j=0,\ldots,n$. $K_i(j)$ is the Krawtchouk number (see `Krawtchouk` (7.5.6)). `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.

Example

```
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 ]
```

7.3.2 GrayMat

▷ `GrayMat(`*n*`, `*F*`)` (function)

`GrayMat` returns a list of all different vectors (see GAP's `Vectors` command) of length *n* over the field *F*, using Gray ordering. *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.

Example

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

7.3.3 SylvesterMat

▷ `SylvesterMat(`*n*`)` (function)

`SylvesterMat` returns the $n \times n$ Sylvester matrix of order *n*. This is a special case of the Hadamard matrices (see `HadamardMat` (7.3.4)). 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.

Example

```
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 ] ]
```

7.3.4 HadamardMat

▷ `HadamardMat(`*n*`)` (function)

`HadamardMat` returns a Hadamard matrix of order *n*. This is an $n \times n$ 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 (as of 1998), `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 (the Paley Type I and II constructions, and the Sylvester construction -- see `SylvesterMat` (7.3.3)). 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 yet have an implementation for a Hadamard matrix in GUAVA: 52, 92.

Example

```
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
```

7.3.5 VandermondeMat

▷ `VandermondeMat(`*X*`, a)` (function)

The function `VandermondeMat` returns the $(a+1) \times n$ matrix of powers x_i^j where *X* is a list of elements of a field, $X = \{x_1, ..., x_n\}$, and `a` is a non-negative integer.

Example

```
gap> M:=VandermondeMat([Z(5),Z(5)^2,Z(5)^0,Z(5)^3],2);
[ [ Z(5)^0, Z(5), Z(5)^2 ], [ Z(5)^0, Z(5)^2, Z(5)^0 ],
  [ Z(5)^0, Z(5)^0, Z(5)^0 ], [ Z(5)^0, Z(5)^3, Z(5)^2 ] ]
gap> Display(M);
 1 2 4
 1 4 1
 1 1 1
 1 3 4
```

7.3.6 PutStandardForm

▷ `PutStandardForm(`*M*`[, idleft])` (function)

We say that a $k \times n$ matrix is in *standard form* if it is equal to the block matrix $(I \mid A)$, for some $k \times (n-k)$ matrix A and where I is the $k \times k$ identity matrix. It follows from a basis result in linear algebra that, after a possible permutation of the columns, using elementary row operations, every matrix can be reduced to standard form. `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*. (The default for *idleft* is 'true'.) If *idleft* is set to 'true', the identity matrix is put on the left side of *M*. Otherwise, it is put at the right side. (This option is useful when putting a check

matrix of a code into standard form.) 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.

Example

```
gap> M := Z(2)*[[1,0,0,1],[0,0,1,1]];; PrintArray(M);
[ [  Z(2)^0,  0*Z(2),  0*Z(2),  Z(2)^0 ],
  [  0*Z(2),  0*Z(2),  Z(2)^0,  Z(2)^0 ] ]
gap> PutStandardForm(M);                  # identity at the left side
(2,3)
gap> PrintArray(M);
[ [  Z(2)^0,  0*Z(2),  0*Z(2),  Z(2)^0 ],
  [  0*Z(2),  Z(2)^0,  0*Z(2),  Z(2)^0 ] ]
gap> PutStandardForm(M, false);           # identity at the right side
(1,4,3)
gap> PrintArray(M);
[ [  0*Z(2),  Z(2)^0,  Z(2)^0,  0*Z(2) ],
  [  0*Z(2),  Z(2)^0,  0*Z(2),  Z(2)^0 ] ]
gap> C := BestKnownLinearCode( 23, 12, GF(2) );
a linear [23,12,7]3 punctured code
gap> G:=MutableCopyMat(GeneratorMat(C));;
gap> PutStandardForm(G);
()
gap> Display(G);
 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 . 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
```

7.3.7 IsInStandardForm

▷ IsInStandardForm(*M[, idleft]*) (function)

IsInStandardForm determines if *M* is in standard form. *idleft* is a boolean that indicates the position of the identity matrix in *M*, as in PutStandardForm (see PutStandardForm (7.3.6)). IsInStandardForm checks if the identity matrix is at the left side of *M*, otherwise if it is at the right side. The elements of *M* may be elements of any field.

Example

```
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
```

7.3.8 PermutedCols

▷ `PermutedCols(M, P)` (function)

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

Example

```
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 ] ]
```

7.3.9 VerticalConversionFieldMat

▷ `VerticalConversionFieldMat(M, F)` (function)

`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, using the $GF(p)$-vector space isomorphism

$$[...] : GF(q) \to GF(p)^m,$$

with $q = p^m$.

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

Example

```
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 `HorizontalConversionFieldMat` (7.3.10)).

7.3.10 HorizontalConversionFieldMat

▷ `HorizontalConversionFieldMat(M, F)` (function)

`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 $k \times n$ matrix, the result is a $k \times m \times n \cdot m$ matrix. The new word length of the resulting code is equal to $n \cdot m$, because each element of $GF(q^m)$ can be represented in $GF(q)$ using m elements. The new dimension is equal to $k \times m$ 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 `ConversionFieldCode` (6.1.15)).

Example

```
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 `VerticalConversionFieldMat` (7.3.9)).

7.3.11 MOLS

▷ `MOLS(q[, n])` (function)

`MOLS` returns a list of `n` *Mutually Orthogonal Latin Squares* (MOLS). A *Latin square* of order `q` is a $q \times q$ 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. However, GUAVA does not yet construct MOLS for $q \equiv 2 \pmod 4$. If it is not possible to construct `n` MOLS, the function returns 'false'.

MOLS are used to create `q`-ary codes (see `MOLSCode` (5.1.4)).

Example

```
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
```

7.3.12 IsLatinSquare

▷ IsLatinSquare(*M*) (function)

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

Example

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

7.3.13 AreMOLS

▷ AreMOLS(*L*) (function)

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 MOLS (7.3.11)).

Example

```
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
```

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. Typically, these are applied to binary linear codes. The definitions of this section were introduced in Graham and Sloane [GS85].

7.4.1 CoordinateNorm

▷ `CoordinateNorm(C, coord)` (function)

`CoordinateNorm` returns the norm of `C` with respect to coordinate `coord`. If $C_a = \{c \in C \mid c_{coord} = a\}$, then the norm of `C` 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.

Example

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

7.4.2 CodeNorm

▷ `CodeNorm(C)` (function)

`CodeNorm` returns the norm of `C`. 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.

Example

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

7.4.3 IsCoordinateAcceptable

▷ `IsCoordinateAcceptable(C, coord)` (function)

`IsCoordinateAcceptable` returns 'true' if coordinate `coord` of `C` 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.

Example

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

7.4.4 GeneralizedCodeNorm

▷ `GeneralizedCodeNorm(C, subcode1, subscode2, ..., subcodek)` (function)

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

Example

```
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
```

7.4.5 IsNormalCode

▷ `IsNormalCode(C)` (function)

`IsNormalCode` returns 'true' if C 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 (in particular, [GS85]) about normality for certain code parameters.

Example

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

7.5 Miscellaneous functions

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

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

7.5.1 CodeWeightEnumerator

▷ `CodeWeightEnumerator(C)` (function)

`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 C with weight i.

Example

```
gap> CodeWeightEnumerator( ElementsCode( [ [ 0,0,0 ], [ 0,0,1 ],
> [ 0,1,1 ], [ 1,1,1 ] ], GF(2) ) );
x_1^3+x_1^2+x_1+1
gap> CodeWeightEnumerator( HammingCode( 3, GF(2) ) );
x_1^7+7*x_1^4+7*x_1^3+1
```

7.5.2 CodeDistanceEnumerator

▷ `CodeDistanceEnumerator(C, w)` (function)

`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 `w`.

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

Example

```
gap> CodeDistanceEnumerator( HammingCode( 3, GF(2) ),[0,0,0,0,0,0,1] );
x_1^6+3*x_1^5+4*x_1^4+4*x_1^3+3*x_1^2+x_1
gap> CodeDistanceEnumerator( HammingCode( 3, GF(2) ),[1,1,1,1,1,1,1] );
x_1^7+7*x_1^4+7*x_1^3+1
```

7.5.3 CodeMacWilliamsTransform

▷ `CodeMacWilliamsTransform(C)` (function)

`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 `C`.

Example

```
gap> CodeMacWilliamsTransform( HammingCode( 3, GF(2) ) );
7*x_1^4+1
```

7.5.4 CodeDensity

▷ `CodeDensity(C)` (function)

`CodeDensity` returns the *density* of `C`. 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 $GF(q^n)$ (which may be computed using `SphereContent`), t is the covering radius of the code and n is the length of the code.

Example

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

7.5.5 SphereContent

▷ SphereContent(n, t, F) (function)

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 DistanceCodeword (3.6.2) 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 partition the whole ambient vector space, where t is the number of errors the code can correct.

Example

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

7.5.6 Krawtchouk

▷ Krawtchouk(k, i, n, q) (function)

Krawtchouk returns the Krawtchouk number $K_k(i)$. q must be a prime power, 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 KrawtchoukMat (7.3.1)). This number is the value at $x = i$ of the polynomial

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

where $b(v,u)=u!/(v!(v-u)!)$ is the binomial coefficient if u,v are integers. For more properties of these polynomials, see [MS83].

Example

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

7.5.7 PrimitiveUnityRoot

▷ PrimitiveUnityRoot(F, n) (function)

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

Example

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

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

7.5.8 PrimitivePolynomialsNr

▷ `PrimitivePolynomialsNr(n, F)` (function)

`PrimitivePolynomialsNr` returns the number of irreducible polynomials over $F = GF(q)$ of degree n with (maximum) period $q^n - 1$. (According to a theorem of S. Golomb, this is $\phi(p^n - 1)/n$.)

See also the GAP function `RandomPrimitivePolynomial`, `RandomPrimitivePolynomial` (8.2.2).

Example

```
gap> PrimitivePolynomialsNr(3,4);
12
```

7.5.9 IrreduciblePolynomialsNr

▷ `IrreduciblePolynomialsNr(n, F)` (function)

`PrimitivePolynomialsNr` returns the number of irreducible polynomials over $F = GF(q)$ of degree n.

Example

```
gap> IrreduciblePolynomialsNr(3,4);
20
```

7.5.10 MatrixRepresentationOfElement

▷ `MatrixRepresentationOfElement(a, F)` (function)

Here F is either a finite extension of the "base field" $GF(p)$ or of the rationals $\mathbb{Q}$, and $a \in F$. The command `MatrixRepresentationOfElement` returns a matrix representation of `a` over the base field.

If the element `a` is defined over the base field then it returns the corresponding 1×1 matrix.

Example

```
gap> a:=Z(4);
Z(2^2)
gap> M:=MatrixRepresentationOfElement(a,GF(4));; Display(M);
 . 1
 1 1
gap> a:=Z(2)^0;                  # this one lies in the base field
Z(2)^0
gap> M:=MatrixRepresentationOfElement(a,GF(4));; Display(M);
 1
```

7.5.11 ReciprocalPolynomial

▷ `ReciprocalPolynomial(P)` (function)

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

This command can also be called using the syntax `ReciprocalPolynomial( P , n )`. In this form, the number of coefficients of `P` is assumed to be less than or equal to $n+1$ (with zero coefficients added in the highest degrees, if necessary). Therefore, the reciprocal polynomial also has degree $n+1$.

Example

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

7.5.12 CyclotomicCosets

▷ `CyclotomicCosets(q, n)` (function)

`CyclotomicCosets` returns the cyclotomic cosets of $q \pmod n$. `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. A q-cyclotomic coset of $s \pmod n$ is a set of the form $\{s, sq, sq^2, ..., sq^{r-1}\}$, where r is the smallest positive integer such that $sq^r - s$ is $0 \pmod n$. In other words, each coset contains all multiplications of the coset representative by $q \pmod n$. The coset representative is the smallest integer that isn't in the previous cosets.

Example

```
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 ] ]
```

7.5.13 WeightHistogram

▷ `WeightHistogram(C[, h])` (function)

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.

Example

```
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
```

7.5.14 MultiplicityInList

▷ `MultiplicityInList(`*L*`, a)` (function)

This is a very simple list command which returns how many times a occurs in L. It returns 0 if a is not in L. (The GAP command `Collected` does not quite handle this "extreme" case.)

Example

```
gap> L:=[1,2,3,4,3,2,1,5,4,3,2,1];;
gap> MultiplicityInList(L,1);
3
gap> MultiplicityInList(L,6);
0
```

7.5.15 MostCommonInList

▷ `MostCommonInList(`*L*`)` (function)

Input: a list L
Output: an a in L which occurs at least as much as any other in L

Example

```
gap> L:=[1,2,3,4,3,2,1,5,4,3,2,1];;
gap> MostCommonInList(L);
1
```

7.5.16 RotateList

▷ `RotateList(`*L*`)` (function)

Input: a list L
Output: a list L' which is the cyclic rotation of L (to the right)

Example

```
gap> L:=[1,2,3,4];;
gap> RotateList(L);
[ 2, 3, 4, 1 ]
```

7.5.17 CirculantMatrix

▷ `CirculantMatrix(k, L)` (function)

Input: integer k, a list L of length n
Output: kxn matrix whose rows are cyclic rotations of the list L

Example

```
gap> k:=3;; L:=[1,2,3,4];;
gap> M:=CirculantMatrix(k,L);;
gap> Display(M);
[ [  1,  2,  3,  4 ],
  [  4,  1,  2,  3 ],
  [  3,  4,  1,  2 ] ]
```

7.6 Miscellaneous polynomial functions

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

7.6.1 MatrixTransformationOnMultivariatePolynomial

▷ `MatrixTransformationOnMultivariatePolynomial (A, f, R)` (function)

`A` is an $n \times n$ matrix with entries in a field F, `R` is a polynomial ring of n variables, say $F[x_1,...,x_n]$, and `f` is a polynomial in `R`. Returns the composition $f \circ A$.

7.6.2 DegreeMultivariatePolynomial

▷ `DegreeMultivariatePolynomial(f, R)` (function)

This command takes two arguments, `f`, a multivariate polynomial, and `R` a polynomial ring over a field F containing `f`, say $R = F[x_1,x_2,...,x_n]$. The output is simply the maximum degrees of all the monomials occurring in `f`.

This command can be used to compute the degree of an affine plane curve.

Example

```
gap> F:=GF(11);;
gap> R2:=PolynomialRing(F,2);;
gap> vars:=IndeterminatesOfPolynomialRing(R2);;
gap> x:=vars[1];; y:=vars[2];;
gap> poly:=y^2-x*(x^2-1);;
gap> DegreeMultivariatePolynomial(poly,R2);
3
```

7.6.3 DegreesMultivariatePolynomial

▷ `DegreesMultivariatePolynomial(f, R)` (function)

Returns a list of information about the multivariate polynomial *f*. Nice for other programs but mostly unreadable by GAP users.

Example

```
gap> F:=GF(11);;
gap> R2:=PolynomialRing(F,["x","y"]);;
gap> vars:=IndeterminatesOfPolynomialRing(R2);;
gap> x:=vars[1];; y:=vars[2];;
gap> poly:=y^2-x*(x^2-1);;
gap> DegreesMultivariatePolynomial(poly,R2);
[ [ [ x, x, 1 ], [ x, y, 0 ] ], [ [ y^2, x, 0 ], [ y^2, y, 2 ] ],
  [ [ x^3, x, 3 ], [ x^3, y, 0 ] ] ]
```

7.6.4 SolveLinearSystem

▷ SolveLinearSystem(*L*, *vars*) (function)

Input: *L* is a list of linear forms in the variables *vars*.

Output: the solution of the system, if its unique.

The procedure is straightforward: Find the associated matrix A, find the "constant vector" b, and solve $A * v = b$. No error checking is performed.

Related to the GAP command SolutionMat(A, b).

Example

```
gap> F:=GF(11);;
gap> R2:=PolynomialRing(F,2);;
gap> vars:=IndeterminatesOfPolynomialRing(R2);;
gap> x:=vars[1];; y:=vars[2];;
gap> f:=3*y-3*x+1;; g:=-5*y+2*x-7;;
gap> soln:=SolveLinearSystem([f,g],[x,y]);
[ Z(11)^3, Z(11)^2 ]
gap> Value(f,[x,y],soln); # checking okay
0*Z(11)
gap> Value(g,[x,y],soln); # checking okay
0*Z(11)
```

7.6.5 GuavaVersion

▷ GuavaVersion() (function)

Returns the current version of Guava. Same as guava_version().

Example

```
gap> GuavaVersion();
"3.23"
```

7.6.6 ZechLog

▷ ZechLog(*x*, *b*, *F*) (function)

Returns the Zech log of x to base b, ie the i such that $x+1=b^i$, so $y+z=y(1+z/y)=b^k$, where k=Log(y,b)+ZechLog(z/y,b) and b must be a primitive element of F.

Example

```
gap> F:=GF(11);; l := One(F);;
gap> ZechLog(2*l,8*l,F);
6
gap> 8*l+l;(2*l)^(-24);
Z(11)^6
Z(11)^6
```

7.6.7 CoefficientToPolynomial

▷ `CoefficientToPolynomial(`*coeffs*`, `*R*`)` (function)

The function `CoefficientToPolynomial` returns the degree $d-1$ polynomial $c_0 + c_1x + ... + c_{d-1}x^{d-1}$, where *coeffs* is a list of elements of a field, $coeffs = \{c_0, ..., c_{d-1}\}$, and *R* is a univariate polynomial ring.

Example

```
gap> F:=GF(11);;
gap> R1:=PolynomialRing(F,["a"]);;
gap> var1:=IndeterminatesOfPolynomialRing(R1);; a:=var1[1];;
gap> coeffs:=Z(11)^0*[1,2,3,4];
[ Z(11)^0, Z(11), Z(11)^8, Z(11)^2 ]
gap> CoefficientToPolynomial(coeffs,R1);
Z(11)^2*a^3+Z(11)^8*a^2+Z(11)*a+Z(11)^0
```

7.6.8 DegreesMonomialTerm

▷ `DegreesMonomialTerm(`*m*`, `*R*`)` (function)

The function `DegreesMonomialTerm` returns the list of degrees to which each variable in the multivariate polynomial ring *R* occurs in the monomial *m*, where *coeffs* is a list of elements of a field.

Example

```
gap> F:=GF(11);
GF(11)
gap> R1:=PolynomialRing(F,["a"]);;
gap> var1:=IndeterminatesOfPolynomialRing(R1);; a:=var1[1];;
gap> b:=X(F,"b",var1);
b
gap> var2:=Concatenation(var1,[b]);
[ a, b ]
gap> R2:=PolynomialRing(F,var2);;
gap> c:=X(F,"c",var2);
c
gap> var3:=Concatenation(var2,[c]);
[ a, b, c ]
gap> R3:=PolynomialRing(F,var3);
GF(11)[a,b,c]
gap> m:=b^3*c^7;
b^3*c^7
```

```
gap> DegreesMonomialTerm(m,R3);
[ 0, 3, 7 ]
```

7.6.9 DivisorsMultivariatePolynomial

▷ `DivisorsMultivariatePolynomial(`*f*`, `*R*`)` (function)

The function `DivisorsMultivariatePolynomial` returns the list of polynomial divisors of *f* in the multivariate polynomial ring *R* with coefficients in a field. This program uses a simple but slow algorithm (see Joachim von zur Gathen, Jürgen Gerhard, [vG03], exercise 16.10) which first converts the multivariate polynomial *f* to an associated univariate polynomial f^*, then `Factors` f^*, and finally converts these univariate factors back into the multivariate polynomial factors of *f*. Since `Factors` is non-deterministic, `DivisorsMultivariatePolynomial` is non-deterministic as well.

Example

```
gap> R2:=PolynomialRing(GF(3),["x1","x2"]);
GF(3)[x1,x2]
gap> vars:=IndeterminatesOfPolynomialRing(R2);
[ x1, x2 ]
gap> x2:=vars[2];
x2
gap> x1:=vars[1];
x1
gap> f:=x1^3+x2^3;;
gap> DivisorsMultivariatePolynomial(f,R2);
[ x1+x2, x1+x2, x1+x2 ]
```

Chapter 8

Coding theory functions in GAP

This chapter will recall from the GAP reference manual some of the GAP coding theory and finite field functions useful for coding theory. Some of these functions are partially written in C for speed. The main functions are

- `AClosestVectorCombinationsMatFFEVecFFE`,
- `AClosestVectorCombinationsMatFFEVecFFECoords`,
- `CosetLeadersMatFFE`,
- `DistancesDistributionMatFFEVecFFE`,
- `DistancesDistributionVecFFEsVecFFE`,
- `DistanceVecFFE` and `WeightVecFFE`,
- `ConwayPolynomial` and `IsCheapConwayPolynomial`,
- `IsPrimitivePolynomial`, and `RandomPrimitivePolynomial`.

However, the GAP command `PrimitivePolynomial` returns an integer primitive polynomial not the finite field kind.

8.1 Distance functions

8.1.1 AClosestVectorCombinationsMatFFEVecFFE

▷ `AClosestVectorCombinationsMatFFEVecFFE(`*mat, F, vec, r, st*`)` (function)

This command runs through the *F*-linear combinations of the vectors in the rows of the matrix *mat* that can be written as linear combinations of exactly *r* rows (that is without using zero as a coefficient) and returns a vector from these that is closest to the vector *vec*. The length of the rows of *mat* and the length of *vec* must be equal, and all elements must lie in *F*. The rows of *mat* must be linearly independent. If it finds a vector of distance at most *st*, which must be a nonnegative integer, then it stops immediately and returns this vector.

Example

```
gap> F:=GF(3);;
gap> x:= Indeterminate( F, "x" );; pol:= x^2+1;
x^2+Z(3)^0
gap> C := GeneratorPolCode(pol,8,F);
a cyclic [8,6,1..2]1..2 code defined by generator polynomial over GF(3)
gap> v:=Codeword("12101111");
[ 1 2 1 0 1 1 1 1 ]
gap> v:=VectorCodeword(v);
[ Z(3)^0, Z(3), Z(3)^0, 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0 ]
gap> G:=GeneratorMat(C);
[ [ Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0 ] ]
gap> AClosestVectorCombinationsMatFFEVecFFE(G,F,v,1,1);
[ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0 ]
```

8.1.2 AClosestVectorComb..MatFFEVecFFECoords

▷ AClosestVectorComb..MatFFEVecFFECoords(*mat*, *F*, *vec*, *r*, *st*) (function)

AClosestVectorCombinationsMatFFEVecFFECoords returns a two element list containing (a) the same closest vector as in AClosestVectorCombinationsMatFFEVecFFE, and (b) a vector v with exactly r non-zero entries, such that $v * mat$ is the closest vector.

Example

```
gap> F:=GF(3);;
gap> x:= Indeterminate( F, "x" );; pol:= x^2+1;
x^2+Z(3)^0
gap> C := GeneratorPolCode(pol,8,F);
a cyclic [8,6,1..2]1..2 code defined by generator polynomial over GF(3)
gap> v:=Codeword("12101111"); v:=VectorCodeword(v);;
[ 1 2 1 0 1 1 1 1 ]
gap> G:=GeneratorMat(C);;
gap> AClosestVectorCombinationsMatFFEVecFFECoords(G,F,v,1,1);
[ [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0 ],
  [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ]
```

8.1.3 DistancesDistributionMatFFEVecFFE

▷ DistancesDistributionMatFFEVecFFE(*mat*, *f*, *vec*) (function)

DistancesDistributionMatFFEVecFFE returns the distances distribution of the vector *vec* to the vectors in the vector space generated by the rows of the matrix *mat* over the finite field *f*. All vectors must have the same length, and all elements must lie in a common field. The distances distribution is a list d of length $Length(vec)+1$, such that the value $d[i]$ is the number of vectors in vecs that have distance $i+1$ to *vec*.

Example

```
gap> v:=[ Z(3)^0, Z(3), Z(3)^0, 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0 ];;
gap> vecs:=[ [ Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
>   [ 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
>   [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ],
>   [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
>   [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3) ],
>   [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0 ] ];;
gap> DistancesDistributionMatFFEVecFFE(vecs,GF(3),v);
[ 0, 4, 6, 60, 109, 216, 192, 112, 30 ]
```

8.1.4 DistancesDistributionVecFFEsVecFFE

▷ `DistancesDistributionVecFFEsVecFFE(vecs, vec)` (function)

`DistancesDistributionVecFFEsVecFFE` returns the distances distribution of the vector *vec* to the vectors in the list *vecs*. All vectors must have the same length, and all elements must lie in a common field. The distances distribution is a list d of length $Length(vec)+1$, such that the value $d[i]$ is the number of vectors in *vecs* that have distance $i+1$ to *vec*.

Example

```
gap> v:=[ Z(3)^0, Z(3), Z(3)^0, 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0 ];;
gap> vecs:=[ [ Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
>   [ 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
>   [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ],
>   [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
>   [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3) ],
>   [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0 ] ];;
gap> DistancesDistributionVecFFEsVecFFE(vecs,v);
[ 0, 0, 0, 0, 0, 4, 0, 1, 1 ]
```

8.1.5 WeightVecFFE

▷ `WeightVecFFE(vec)` (function)

`WeightVecFFE` returns the weight of the finite field vector *vec*, i.e. the number of nonzero entries.

Example

```
gap> v:=[ Z(3)^0, Z(3), Z(3)^0, 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0 ];;
gap> WeightVecFFE(v);
7
```

8.1.6 DistanceVecFFE

▷ `DistanceVecFFE(vec1, vec2)` (function)

The *Hamming metric* on $GF(q)^n$ is the function

$$dist((v_1,...,v_n),(w_1,...,w_n)) = |\{i \in [1..n] \mid v_i \neq w_i\}|.$$

This is also called the (Hamming) distance between $v = (v_1,...,v_n)$ and $w = (w_1,...,w_n)$. `DistanceVecFFE` returns the distance between the two vectors *vec1* and *vec2*, which must have the same length and whose elements must lie in a common field. The distance is the number of places where *vec1* and *vec2* differ.

Example

```
gap> v1:=[ Z(3)^0, Z(3), Z(3)^0, 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0 ];;
gap> v2:=[ Z(3), Z(3)^0, Z(3)^0, 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0 ];;
gap> DistanceVecFFE(v1,v2);
2
```

8.2 Other functions

We basically repeat, with minor variation, the material in the GAP manual or from Frank Luebeck's website http://www.math.rwth-aachen.de/~Frank.Luebeck/data/ConwayPol on Conway polynomials. The PRIME FIELDS: If $p \geq 2$ is a prime then $GF(p)$ denotes the field $\mathbb{Z}/p\mathbb{Z}$, with addition and multiplication performed mod p.

The PRIME POWER FIELDS: Suppose $q = p^r$ is a prime power, $r > 1$, and put $F = GF(p)$. Let $F[x]$ denote the ring of all polynomials over F and let $f(x)$ denote a monic irreducible polynomial in $F[x]$ of degree r. The quotient $E = F[x]/(f(x)) = F[x]/f(x)F[x]$ is a field with q elements. If $f(x)$ and E are related in this way, we say that $f(x)$ is the DEFINING POLYNOMIAL of E. Any defining polynomial factors completely into distinct linear factors over the field it defines.

For any finite field F, the multiplicative group of non-zero elements $F^\times$ is a cyclic group. An $\alpha \in F$ is called a PRIMITIVE ELEMENT if it is a generator of $F^\times$. A defining polynomial $f(x)$ of F is said to be PRIMITIVE if it has a root in F which is a primitive element.

8.2.1 ConwayPolynomial

▷ `ConwayPolynomial(p, n)` (function)

A standard notation for the elements of $GF(p)$ is given via the representatives $0,...,p-1$ of the cosets modulo p. We order these elements by $0 \langle 1 \langle 2 \langle ... \langle p-1$. We introduce an ordering of the polynomials of degree r over $GF(p)$. Let $g(x) = g_r x^r + ... + g_0$ and $h(x) = h_r x^r + ... + h_0$ (by convention, $g_i = h_i = 0$ for $i \rangle r$). Then we define $g \langle h$ if and only if there is an index k with $g_i = h_i$ for $i \rangle k$ and $(-1)^{r-k} g_k \langle (-1)^{r-k} h_k$.

The CONWAY POLYNOMIAL $f_{p,r}(x)$ for $GF(p^r)$ is the smallest polynomial of degree r with respect to this ordering such that:

- $f_{p,r}(x)$ is monic,
- $f_{p,r}(x)$ is primitive, that is, any zero is a generator of the (cyclic) multiplicative group of $GF(p^r)$,
- for each proper divisor m of r we have that $f_{p,m}(x^{(p^r-1)/(p^m-1)}) \equiv 0 \pmod{f_{p,r}(x)}$; that is, the $(p^r-1)/(p^m-1)$-th power of a zero of $f_{p,r}(x)$ is a zero of $f_{p,m}(x)$.

`ConwayPolynomial(p,n)` returns the polynomial $f_{p,r}(x)$ defined above.

`IsCheapConwayPolynomial(p,n)` returns true if `ConwayPolynomial( p, n )` will give a result in reasonable time. This is either the case when this polynomial is pre-computed, or if n, p are not too big.

8.2.2 RandomPrimitivePolynomial

▷ `RandomPrimitivePolynomial(F, n)` (function)

For a finite field `F` and a positive integer `n` this function returns a primitive polynomial of degree `n` over `F`, that is a zero of this polynomial has maximal multiplicative order $|F|^n - 1$.

`IsPrimitivePolynomial(f)` can be used to check if a univariate polynomial `f` is primitive or not.

Chapter 9

GNU Free Documentation License

GNU Free Documentation License Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc. 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding

them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in

parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation

License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See https://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

References

[All84] W. O. Alltop. A method for extending binary linear codes. *IEEE Trans. Inform. Theory*, 30(6):871-872, 1984. 113

[BM06] L. Bazzi and S. K. Mitter. Some randomized code constructions from group actions. *IEEE Trans. Inform. Theory*, 52(7):3210-3219, 2006. 74

[Bro98] A. E. Brouwer. Bounds on the size of linear codes. In V. S. Pless and W. C. Huffman, editors, *Handbook of Coding Theory*, page 295-461. Elsevier, Amsterdam, 1998. 114

[Bro06] A. E. Brouwer. Bounds on the minimum distance of linear codes. `https://www.codetables.de/`, 1997-2006. 116, 121

[Che69] C. L. Chen. *Some Results on Algebraically Structured Error-Correcting Codes*. Doctoral dissertation, University of Hawaii, Honolulu, USA, 1969. 40

[Gal62] R. Gallager. Low-density parity-check codes. *IRE Trans. Inf. Theor.*, IT-8:21-28, January 1962. 96

[Gao03] S. Gao. A new algorithm for decoding Reed-Solomon codes. In V. Bhargava, H. V. Poor, V. Tarokh, and S. Yoon, editors, *Communications, Information and Network Security*, page 55-68. Springer, Boston, MA, 2003. 49

[GDT91] E. Gabidulin, A. Davydov, and L. Tombak. Linear codes with covering radius 2 and other new covering codes. *IEEE Trans. Inform. Theory*, 37(1):219-224, 1991. 67

[GS85] R. Graham and N. Sloane. On the covering radius of codes. *IEEE Trans. Inform. Theory*, 31(3):385-401, 1985. 110, 136, 138

[Han00] J. P. Hansen. Toric surfaces and error-correcting codes. In J. Buchmann, T. Høholdt, H. Stichtenoth, and H. Tapia-Recillas, editors, *Coding Theory, Cryptography and Related Areas*, page 132-142. Springer, Berlin Heidelberg, 2000. 82

[Hel72] H. J. Helgert. Srivastava codes. *IEEE Trans. Inform. Theory*, 18(2):292-297, March 1972. 63

[HHKK07] M. Harada, W. Holzmann, H. Kharaghani, and M. Khorvash. Extremal ternary self-dual codes constructed from negacirculant matrices. *Graphs Combin.*, 23(4):401-417, 2007. 79

[HP03] W. C. Huffman and V. Pless. *Fundamentals of error-correcting codes*. Cambridge University Press, 2003. 6, 26, 28, 47, 48, 53, 61, 62, 129

[JH04] J. Justesen and T. Høholdt. *A course in Error-Correcting Codes*. EMS Textbooks in Mathematics. European Mathematical Society, 2004. 48, 49, 50, 81

[Joy04] D. Joyner. Toric codes over finite fields. *Appl. Algebra Engrg. Comm. Comput.*, 15(1):63-79, 2004. 82

[Leo82] J. S. Leon. Computing automorphism groups of error-correcting codes. *IEEE Trans. Inform. Theory*, 28(3):496-511, May 1982. 31

[Leo88] J. S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Trans. Inform. Theory*, 34(5):1354-1359, September 1988. 40, 41

[Leo91] J. S. Leon. Permutation group algorithms based on partitions, I: Theory and algorithms. *J. Symbolic Comput.*, 12(4-5):533-583, 1991. 6

[MS83] F. J. MacWilliams and N. J. A. Sloane. *The theory of error-correcting codes*, volume 16 of *North-Holland Mathematical Library*. North-Holland, Amsterdam, 1983. 6, 62, 68, 81, 140

[SRC72] N. Sloane, S. Reddy, and C. Chen. New binary codes. *IEEE Trans. Inform. Theory*, 18(4):503-510, July 1972. 112

[Sti93] H. Stichtenoth. *Algebraic Function Fields and Codes*, volume 254 of *Graduate Texts in Mathematics*. Springer, 1993. 95

[TSS+04] R. Tanner, D. Sridhara, A. Sridharan, T. Fuja, and D. Costello Jr. LDPC block and convolutional codes based on circulant matrices. *IEEE Trans. Inform. Theory*, 50(12):2966-2984, December 2004. 97

[vG03] J. von zur Gathen and J. Gerhard. *Modern computer algebra*. Cambridge University Press, 2003. 147

[Zim96] K. H. Zimmermann. Integral Hecke modules, integral generalized Reed-Muller codes, and linear codes. Technical Report 3-96, Technische Universität Hamburg-Harburg, Hamburg, Germany, 1996. 40

Index

$A(n,d)$, 119
$GF(p)$, 151
$GF(q)$, 151
$t(n,k)$, 43
*
  for codes, 23
  for message and code, 23
\+
  for codes, 23
  for codeword and code, 15
  for codewords, 15
\=
  for codes, 22
  for codewords, 14
\-
  for codewords, 15
< >, 14, 22

acceptable coordinate, 137
AClosestVectorComb..MatFFEVecFFE-Coords, 149
AClosestVectorCombinationsMatFFE-VecFFE, 148
ActionMoebiusTransformationOnDivisorP1, 91
ActionMoebiusTransformationOnFunction, 91
AddedElementsCode, 102
affine code, 29
AffineCurve, 83
AffinePointsOnCurve, 84
AlternantCode, 61
AmalgamatedDirectSumCode, 111
AreMOLS, 136
AsSSortedList, 33
AugmentedCode, 101
AutomorphismGroup, 31

BCHCode, 72
BestKnownLinearCode, 65
BinaryGolayCode, 68
BitFlipDecoder, 50
BlockwiseDirectSumCode, 111
Bose distance, 72
bound, Gilbert-Varshamov lower, 120
bound, sphere packing lower, 120
bounds, Elias, 118
bounds, Griesmer, 118
bounds, Hamming, 117
bounds, Johnson, 117
bounds, Plotkin, 118
bounds, Singleton, 116
bounds, sphere packing bound, 117
BoundsCoveringRadius, 122
BoundsMinimumDistance, 121
BZCode, 114
BZCodeNC, 114

check polynomial, 21, 70
CheckMat, 36
CheckMatCode, 60
CheckMatCodeMutable, 60
CheckPol, 37
CheckPolCode, 71
CirculantMatrix, 144
code, 20
code, (n,M,d), 20
code, $[n,k,d]r$, 21
code, AG, 83
code, alternant, 61
code, Bose-Chaudhuri-Hockenghem, 72
code, conference, 56
code, Cordaro-Wagner, 63
code, cyclic, 21
code, Davydov, 67
code, doubly-even, 28
code, element test, 24
code, elements of, 20

code, evaluation, 80
code, even, 28
code, Fire, 75
code, Gabidulin, 67
code, Golay (binary), 68
code, Golay (ternary), 69
code, Goppa (classical), 61
code, greedy, 58
code, Hadamard, 56
code, Hamming, 60
code, linear, 20
code, maximum distance separable, 27
code, Nordstrom-Robinson, 58
code, perfect, 26
code, Reed-Muller, 61
code, Reed-Solomon, 73
code, self-dual, 27
code, self-orthogonal, 27
code, singly-even, 28
code, Srivastava, 62
code, subcode, 25
code, Tombak, 67
code, toric, 82
code, unrestricted, 20
CodeDensity, 139
CodeDistanceEnumerator, 139
CodeIsomorphism, 30
CodeMacWilliamsTransform, 139
CodeNorm, 137
codes, addition, 23
codes, decoding, 24
codes, direct sum, 23
codes, encoding, 23
codes, product, 23
CodeWeightEnumerator, 138
Codeword, 12
CodewordNr, 13
codewords, addition, 15
codewords, cosets, 16
codewords, subtraction, 15
CoefficientToPolynomial, 146
conference matrix, 57
ConferenceCode, 56
ConstantWeightSubcode, 106
ConstructionBCode, 104
ConstructionXCode, 112
ConstructionXXCode, 113
ConversionFieldCode, 105
ConwayPolynomial, 151
CoordinateNorm, 137
CordaroWagnerCode, 63
coset, 16
CosetCode, 106
covering code, 43
CoveringRadius, 43
cyclic, 79
CyclicCodes, 76
CyclicMDSCode, 78
CyclotomicCosets, 142

DavydovCode, 67
Decode, 47
Decodeword, 48
DecreaseMinimumDistanceUpperBound, 41
defining polynomial, 151
degree, 86
DegreeMultivariatePolynomial, 144
DegreesMonomialTerm, 146
DegreesMultivariatePolynomial, 144
density of a code, 139
Dimension, 33
DirectProductCode, 109
DirectSumCode, 108
Display, 35
DisplayBoundsInfo, 35
distance, 46
DistanceCodeword, 18
DistancesDistribution, 46
DistancesDistributionMatFFEVecFFE, 149
DistancesDistributionVecFFEsVecFFE, 150
DistanceVecFFE, 150
divisor, 85
DivisorAddition , 86
DivisorAutomorphismGroupP1 , 92
DivisorDegree , 86
DivisorGCD , 87
DivisorIsZero , 87
DivisorLCM , 87
DivisorNegate , 86
DivisorOfRationalFunctionP1 , 89
DivisorOnAffineCurve, 86
DivisorsEqual , 87
DivisorsMultivariatePolynomial, 147

doubly-even, 27
DualCode, 105

ElementsCode, 55
encoder map, 23
EnlargedGabidulinCode, 67
EnlargedTombakCode, 67
equivalent codes, 30
EvaluationBivariateCode, 94
EvaluationBivariateCodeNC, 94
EvaluationCode, 80
even, 28
EvenWeightSubcode, 99
ExhaustiveSearchCoveringRadius, 123
ExpurgatedCode, 100
ExtendedBinaryGolayCode, 68
ExtendedCode, 98
ExtendedDirectSumCode, 110
ExtendedReedSolomonCode, 73
ExtendedTernaryGolayCode, 69
external distance, 128

FerreroDesignCode, 63
FireCode, 75
FourNegacirculantSelfDualCode, 79
FourNegacirculantSelfDualCodeNC, 80

GabidulinCode, 67
Gary code, 131
GeneralizedCodeNorm, 137
GeneralizedReedMullerCode, 82
GeneralizedReedSolomonCode, 81
GeneralizedReedSolomonDecoderGao, 49
GeneralizedReedSolomonListDecoder, 49
GeneralizedSrivastavaCode, 62
GeneralLowerBoundCoveringRadius, 124
GeneralUpperBoundCoveringRadius, 124
generator polynomial, 21, 70
GeneratorMat, 36
GeneratorMatCode, 59
GeneratorPol, 37
GeneratorPolCode, 70
GenusCurve, 84
GoppaCode, 62
GoppaCodeClassical, 93
GOrbitPoint , 84
GrayMat, 131

greatest common divisor, 87
GreedyCode, 58
Griesmer code, 119
GuavaVersion, 145

Hadamard matrix, 56, 131
HadamardCode, 56
HadamardMat, 131
Hamming metric, 150
HammingCode, 61
HorizontalConversionFieldMat, 135
hull, 110

in, 24
IncreaseCoveringRadiusLowerBound, 122
information bits, 24
InformationWord, 24
InnerDistribution, 45
IntersectionCode, 109
IrreduciblePolynomialsNr, 141
IsActionMoebiusTransformationOn-
DivisorDefinedP1 , 91
IsAffineCode, 29
IsAlmostAffineCode, 30
IsCheapConwayPolynomial, 151
IsCode, 25
IsCodeword, 14
IsCoordinateAcceptable, 137
IsCyclicCode, 25
IsDoublyEvenCode, 27
IsEquivalent, 30
IsEvenCode, 28
IsFinite, 32
IsGriesmerCode, 119
IsInStandardForm, 133
IsLatinSquare, 136
IsLinearCode, 25
IsMDSCode, 26
IsNormalCode, 138
IsPerfectCode, 26
IsPrimitivePolynomial, 152
IsSelfComplementaryCode, 29
IsSelfDualCode, 27
IsSelfOrthogonalCode, 27
IsSinglyEvenCode, 28
IsSubset, 25

Krawtchouk, 140

KrawtchoukMat, 130

Latin square, 135
LDPC, 96
least common multiple, 87
LeftActingDomain, 33
length, 20
LengthenedCode, 103
LexiCode, 59
linear code, 11
LowerBoundCoveringRadiusCounting-
Excess, 126
LowerBoundCoveringRadiusEmbedded1, 127
LowerBoundCoveringRadiusEmbedded2, 127
LowerBoundCoveringRadiusInduction, 128
LowerBoundCoveringRadiusSphere-
Covering, 124
LowerBoundCoveringRadiusVanWee1, 125
LowerBoundCoveringRadiusVanWee2, 126
LowerBoundGilbertVarshamov, 120
LowerBoundMinimumDistance, 120
LowerBoundSpherePacking, 120

MacWilliams transform, 139
MatrixRepresentationOfElement, 141
MatrixRepresentationOnRiemannRoch-
SpaceP1 , 92
MatrixTransformationOnMultivariate-
Polynomial , 144
maximum distance separable, 117
MDS, 27, 79
minimum distance, 20
MinimumDistance, 38
MinimumDistanceLeon, 39
MinimumDistanceRandom, 42
MinimumWeight, 40
MinimumWeightWords, 45
MoebiusTransformation , 90
MOLS, 135
MOLSCode, 57
MostCommonInList, 143
MultiplicityInList, 143
mutually orthogonal Latin squares, 135

NearestNeighborDecodewords, 51
NearestNeighborGRSDecodewords, 50
NordstromRobinsonCode, 58
norm of a code, 137
normal code, 138
not =, 14, 22
NrCyclicCodes, 77
NullCode, 76
NullWord, 18

OnePointAGCode, 95
OptimalityCode, 65
order of polynomial, 75
OuterDistribution, 46

Parity check, 98
parity check matrix, 20
perfect, 117
perfect code, 139
permutation equivalent codes, 30
PermutationAutomorphismGroup, 31
PermutationAutomorphismGroup, 32
PermutationDecode, 53
PermutationDecodeNC, 54
PermutedCode, 100
PermutedCols, 134
PiecewiseConstantCode, 108
point, 84
PolyCodeword, 17
primitive element, 151
PrimitivePolynomialsNr, 141
PrimitiveUnityRoot, 140
Print, 34
PuncturedCode, 99
PutStandardForm, 132

QCLDPCCodeFromGroup, 96
QQRCode, 74
QQRCodeNC, 74
QRCode, 74
QuasiCyclicCode, 77

RandomCode, 58
RandomLinearCode, 64
RandomPrimitivePolynomial, 152
reciprocal polynomial, 141
ReciprocalPolynomial, 142
Redundancy, 38
ReedMullerCode, 61
ReedSolomonCode, 73
RemovedElementsCode, 101

RepetitionCode, 76
ResidueCode, 104
RiemannRochSpaceBasisFunctionP1 , 88
RiemannRochSpaceBasisP1 , 89
RootsCode, 71
RootsOfCode, 37
RotateList, 143

self complementary code, 29
self-dual, 80, 105
self-orthogonal, 27
SetCoveringRadius, 44
ShortenedCode, 102
singly-even, 28
Size, 32
size, 20
SolveLinearSystem, 145
SphereContent, 140
SrivastavaCode, 63
standard form, 132
StandardArray, 53
StandardFormCode, 107
strength, 129
String, 34
SubCode, 104
Support, 19
support, 85
SylvesterMat, 131
Syndrome, 52
syndrome table, 52
SyndromeTable, 52

TernaryGolayCode, 69
TombakCode, 67
ToricCode, 82
ToricPoints, 82
TraceCode, 106
TreatAsPoly, 17
TreatAsVector, 17

UnionCode, 110
UpperBound, 119
UpperBoundCoveringRadiusCyclicCode, 130
UpperBoundCoveringRadiusDelsarte, 129
UpperBoundCoveringRadiusGriesmerLike, 129
UpperBoundCoveringRadiusRedundancy, 128
UpperBoundCoveringRadiusStrength, 129
UpperBoundElias, 118
UpperBoundGriesmer, 119
UpperBoundHamming, 117
UpperBoundJohnson, 117
UpperBoundMinimumDistance, 121
UpperBoundPlotkin, 118
UpperBoundSingleton, 117
UUVCode, 108

VandermondeMat, 132
VectorCodeword, 16
VerticalConversionFieldMat, 134

weight enumerator polynomial, 138
WeightCodeword, 19
WeightDistribution, 45
WeightHistogram, 142
WeightVecFFE, 150
WholeSpaceCode, 76
WordLength, 38

ZechLog, 145