cvec

Compact vectors over finite fields

2.7.6

6 August 2022

Max Neunhöffer

Max Neunhöffer
Email: max@9hoeffer.de
Homepage: http://www-groups.mcs.st-and.ac.uk/~neunhoef
Address: Gustav-Freytag-Straße 40
50354 Hürth
Germany

Copyright

Contents

Chapter 1

Introduction

1.1 Philosophy

This package implements a complete infrastructure for vectors over finite fields. The basic idea is, that one can store an element of a finite field using only a few bits rather than a full machine word. Therefore one can pack more than one finite field element in a machine word. This approach not only saves memory but also allows fast arithmetic.

Contrary to other implementations this package uses long word instructions for arithmetic and other operations rather than table lookups, because modern microprocessor designs seem to support faster memory access in this way and memory access is the main limiting factor for computations over finite fields. This approach also allows for bigger finite fields with more than 256 elements. For a more detailed descriptions of this design see Chapter 3.

The main purpose of this implementation is to use it in MeatAxe-like applications, that is, working with matrices consisting of compressed vectors over finite fields, doing linear algebra calculations like nullspaces, spinning of vectors, multiplying and inverting matrices and the like. Another purpose could be matrix group calculations. Usually in such computations, the base field does not change too often and not many different lengths of vectors occur. This implementation is optimized with these applications in mind and might not be very efficient for other purposes like using compressed vectors as coefficient lists of polynomials.

Another important point is that compressed vectors in this package do not even try to behave exactly like GAP lists. To the contrary, they disallow many operations that are possible for GAP lists for example changing their length or assigning arbitrary GAP objects to positions in the vector. The reason for this is that the chosen data structure does not allow to assign anything but elements of the one base field to positions in the vector and the option to change the representation “on the fly” is not desirable in most applications. On the other hand one can be relatively sure not to “lose compression” along the way.

1.2 Overview over this manual

Chapter 2 describes the installation of this package. Chapter 3 describes the basic design and all the data structures used in this package, including the external representation of matrices on storage. These descriptions might be very valuable to understand the behaviour of the implementation and various performance issues. Those and other performance issues are covered in Chapter 7, where you mainly find hints on how to tune your own programs that use this package. Chapters 4 to 5 describe the available functionality for vectors and matrices respectively. Chapter 8 describes, how the functionality in this package is or is not usable in connection with the GAP library. Finally, Chapter 9 shows instructive examples for the usage of this package.

1.3 Feedback

For bug reports, feature requests and suggestions, please use our issue tracker.

Chapter 2

Installation of the cvec-Package

To get the newest version of this GAP 4 package download one of the archive files

- `cvec-x.x.tar.gz`
- `cvec-x.x.tar.bz2`
- `cvec-x.x.zip`

and unpack it using

```
gunzip cvec-x.x.tar.gz; tar xvf cvec-x.x.tar
```

or

```
bzip2 -d cvec-x.x.tar.bz2; tar xvf cvec-x.x.tar
```

or

```
unzip -x cvec-x.x.zip
```

respectively.

Do this in a directory called "`pkg`", preferably (but not necessarily) in the "`pkg`" subdirectory of your GAP 4 installation. It creates a subdirectory called "`cvec`".

The package will not work without the following compilation step.

To compile the C part of the package do (in the `pkg` directory)

```
cd cvec
./configure
make
```

If you installed the package in another "`pkg`" directory than the standard "`pkg`" directory in your GAP 4 installation, then you have to do two things. Firstly during compilation you have to use the option `-with-gaproot=PATH` of the `configure` script where "PATH" is a path to the main GAP root directory (if not given the default "`../..`" is assumed).

Secondly you have to specify the path to the directory containing your "`pkg`" directory to GAP's list of directories. This can be done by starting GAP with the "`-l`" command line option followed by the name of the directory and a semicolon. Then your directory is prepended to the list of directories searched. Otherwise the package is not found by GAP. Of course, you can add this option to your GAP startup script.

Chapter 3

The Data Structures

This chapter describes all the data structures used in this package. We start with a section on finite fields and what is already there in the GAP kernel and library. Then we describe compressed vectors and compressed matrices.

3.1 Finite field elements

Throughout the package, elements in the field $GF(p)$ of p elements are represented by numbers $0, 1, \dots, p-1$ and arithmetic is just the standard arithmetic in the integers modulo p.

Bigger finite fields are done by calculating in the polynomial ring $GF(p)[x]$ in one indeterminate x modulo a certain irreducible polynomial. By convention, we use the so-called “Conway polynomials” (see `http://www.math.rwth-aachen.de:8001/~Frank.Luebeck/data/ConwayPol/index.html`) for this purpose, because they provide a standard way of embedding finite fields into their extension fields. Because Conway polynomials are monic, we can store coset representatives by storing polynomials of degree less than d, where d is the degree of the finite field over its prime field.

As of this writing, GAP has two implementation of finite field elements built into its kernel and library, the first of which stores finite field elements by storing the discrete logarithm with respect to a primitive root of the field. Although this is nice and efficient for small finite fields, it becomes unhandy for larger finite fields, because one needs a lookup table of length p^d for doing efficient addition. This implementation thus is limited to fields with less than or equal to 65536 elements. The other implementation using polynomial arithmetic modulo the Conway polynomial is used for fields with more than 65536 elements. For prime fields of characteristic p with more than that elements, there is an implementation using integers modulo p.

3.2 Compressed Vectors in Memory

3.2.1 Packing of prime field elements

For this section, we assume that the base field is $GF(p^d)$, the finite field with p^d elements, where p is a prime and d is a positive integer. This is realized as a field extension of degree d over the prime field $GF(p)$ using the Conway polynomial $c \in GF(p)[x]$. We always represent field elements of $GF(p^d)$ by polynomials $a = \sum_{i=0}^{d-1} a_i x^i$ where the coefficients a_i are in $GF(p)$. Because c is monic, this gives a one-to-one correspondence between polynomials and finite field elements.

The memory layout for compressed vectors is determined by two important constants, depending only on p and the word length of the machine. The word length is 4 bytes on 32bit machines (for example on the i386 architecture) and 8 bytes on 64bit machines (for example on the AMD64 architecture). More concretely, a “`Word`” is an `unsigned long int` in C and the length of a `Word` is `sizeof(unsigned long int)`.

Those constants are `bitsperel` (bits per prime field element) and `elsperword` (prime field elements per `Word`). `bitsperel` is 1 for $p = 2$ and otherwise the smallest integer, such that $2^b itsperel > 2 \cdot p - 1$. This means, that we can store the numbers from 0 to $2 \cdot p - 1$ in `bitsperel` bits. `elsperword` is $32/bitsperel$ rounded down to the next integer and multiplied by 2 if the length of a `Word` is 8 bytes. Note that we thus store as many prime field elements as possible into one `Word`, however, on 64bit machines we store only twice as much as would fit into 32bit, even if we could pack one more into a `Word`. This has technical reasons to make conversion between different architectures more efficient.

These definitions imply that we can put `elsperword` prime field elements into one `Word`. We do this by using the `bitsperel` least significant bits in the `Word` for the first prime field element, using the next `bitsperel` bits for the next prime field element and so on. Here is an example that shows how the 6 finite field elements $0, 1, 2, 3, 4, 5$ of $GF(11)$ are stored in that order in one 32bit `Word`. `bitsperel` is here 4, because $2^4 < 2 \cdot 11 - 1 = 21 < 2^5$. Therefore `elsperword` is 5 on a 32bit machine.

Example
```
most significant xx|.....|.....|.....|.....|.....|..... least significant
                 00|00101|00100|00011|00010|00001|00000
                   |    5|    4|    3|    2|    1|    0
```

Here is another example that shows how the 20 finite field elements $0, 1, 2, 0, 0, 0, 1, 1, 1, 2, 2, 2, 0, 1, 2, 2, 1, 0, 2, 2$ of $GF(3)$ are stored in that order in one 64bit `Word`. `bitsperel` is here 3, because $2^2 < 2 \cdot 3 - 1 = 5 < 2^3$. Therefore `elsperword` is 20 on a 32bit machine. Remember, that we only put twice as many elements in one 64bit `Word` than we could in one 32bit `Word`!

Example
```
xxxx..!..!..!..!..!..!..!..!..!..!..!..!..!..!..!..!..!..!..!..!
0000010010000000101001000100001001001000100100100000000000010001000
      2  2  0  1  2  2  1  0  2  2  2  1  1  1  0  0  0  2  1  0
```

(The exclamation marks mark the right hand side of the prime field elements.)

Note that different architectures store their `Words` in a different byte order in memory (“endianess”). We do *not* specify how the data is distributed into bytes here! All access is via `Words` and their arithmetic (shifting, addition, multiplication, etc.). See Section 3.4 for a discussion of this with respect to our external representation.

3.2.2 Extension Fields

Now that we have seen how prime field elements are packed into `Words`, we proceed to the description how compressed vectors are stored over arbitrary extension fields:

Assume a compressed vector has length l with $l \geq 0$. If $d = 1$ (prime field), it just uses $elsperword/l$ `Words` (division rounded up to the next integer), where the first `Word` stores the leftmost `elsperword` field elements in the first `Word` as described above and so on. This means, that the very first field element is stored in the least significant bits of the first `Word`.

In the extension field case $GF(p^d)$, a vector of length l uses $(elsperword/l) * d$ Words (division rounded up to the next integer), where the first d Words store the leftmost elsperword field elements. The very first word stores all the constant coefficients of the polynomials representing the first elsperword field elements in their order from left to right, the second Word stores the coefficients of x^1 and so on until the d'th Word, which stores the coefficients of x^{d-1}. Unused entries behind the end of the actual vector data within the last Word has to be zero!.

The following example shows, how the 9 field elements x^2+x+1, x^2+2x+2, x^2+3x+3, x^2+4x+4, $2x^2+x$, $2x^2+3x+1$, $2x^2+4x+2$, $3x^2+1$, and $4x^2+x+3$ of $GF(5^3)$ occurring in a vector of length 9 in that order are stored on a 32bit machine:

Example
```
^^^ lower memory addresses ^^^
              ....|....|....|....|....|....|....|....  (least significant bit)
1st Word:  0001|0010|0001|0000|0100|0011|0010|0001|  (first
2nd Word:  0000|0100|0011|0001|0100|0011|0010|0001|     8 field
3rd Word:  0011|0010|0010|0010|0001|0001|0001|0001|        elements)
---------------------------------------------------
4th Word:  0000|0000|0000|0000|0000|0000|0000|0011|  (second
5th Word:  0000|0000|0000|0000|0000|0000|0000|0001|     8 field
6th Word:  0000|0000|0000|0000|0000|0000|0000|0100|        elements)
VVV higher memory addresses VVV
```

A “cvec” (one of our compressed vectors) is a GAP “Data object” (that is with TNUM equal to T_DATOBJ). The first machine word in its bag is a pointer to its type, from the second machine word on the Words containing the above data are stored. The bag is exactly long enough to hold the type pointer and the data Words.

3.2.3 How is information about the base field stored?

But how does the system know, over which field the vector is and how long it is? The type of a GAP object consists of 3 pieces: A family, a bit list (for the filters), and one GAP object for “defining data”. The additional information about the vector is stored in the third piece, the defining data, and is called a “cvecclass”.

A cvecclass is a positional object with 5 components:

Position	Name	Content
1	IDX_fieldinfo	a fieldinfo object, see below
2	IDX_len	the length of the vector as immediate GAP integer
3	IDX_wordlen	the number of Words used as immediate GAP integer
4	IDX_type	a GAP type used to create new vectors
5	IDX_GF	a GAP object for the base field
6	IDX_lens	a list holding lengths of vectors in cvecclasses for the same field
7	IDX_classes	a list holding cvecclasses for the same field with lengths as in entry number 6

Table: Components of a cvecclass

In position 5 we have just the GAP finite field object GF(p,d). The names appear as symbols in the code.

The field is described by the GAP object in position 1, a so-called fieldinfo object, which is described in the following table:

Position	Name	*Content*
1	`IDX_p`	p as an immediate GAP integer
2	`IDX_d`	d as an immediate GAP integer
3	`IDX_q`	$q = p^d$ as a GAP integer
4	`IDX_conway`	a GAP string containing the coefficients of the Conway Polynomial as unsigned int []
5	`IDX_bitsperel`	number of bits per element of the prime field (`bitsperel`)
6	`IDX_elsperword`	prime field elements per `Word` (`elsperword`)
7	`IDX_wordinfo`	a GAP string containing C data for internal use
8	`IDX_bestgrease`	the best grease level (see Section 5.8)
9	`IDX_greasetabl`	the length of a grease table using best grease
10	`IDX_filts`	a filter list for the creation of new vectors over this field
11	`IDX_tab1`	a table for $GF(q)$ to `[0..q-1]` conversion if $q \leq 65536$
12	`IDX_tab2`	a table for `[0..q-1]` to $GF(q)$ conversion if $q \leq 65536$
13	`IDX_size`	0 for $q \leq 65536$, otherwise 1 if q is a GAP immediate integer and 2 if not
14	`IDX_scafam`	the scalars family

Table: Components of a `fieldinfo`

Note that GAP has a family for all finite field elements of a given characteristic p, vectors over a finite field are then in the collections family of that family and matrices are in the collections family of the collections family of the scalars family. We use the same families in the same way for compressed vectors and matrices.

3.2.4 Limits that follow from the Data Format

The following limits come from the above mentioned data format or other internal restrictions (an "immediate integer" in GAP can take values between 2^{-28} and $(2^{28}) - 1$ inclusively on 32bit machines and between 2^{-60} and $(2^{60}) - 1$ on 64bit machines):

- The prime p must be an immediate integer.
- The degree d must be smaller than 1024 (this limit is arbitrarily chosen at the moment and could be increased easily).
- The Conway polynomial must be known to GAP.
- The length of a vector must be an immediate integer.

3.3 Compressed Matrices

The implementation of `cmats` (compressed matrices) is done mainly on the GAP level without using too many C parts. Only the time critical parts for some operations for matrices are done in the kernel.

A `cmat` object is a positional object with at least the following components:

Component name	Content
`len`	the number of rows, can be 0
`vecclass`	a `cvecclass` object describing the class of rows
`scaclass`	a `cscaclass` object holding a reference to `GF(p,d)`
`rows`	a list containing the rows of the matrix, which are `cvecs`
`greasehint`	the recommended greasing level

Table: Components of a `cmat` object

The `cvecclass` in the component `vecclass` determines the number of columns of the matrix by the length of the rows.

The length of the list in component `rows` is `len+1`, because the first position is equal to the integer 0 such that the type of the list `rows` can always be computed efficiently. The rows are then stored in positions 2 to `len+1`.

The component `greasehint` contains the greasing level for the next matrix multiplication, in which this matrix occurs as the factor on the right hand side (if greasing is worth the effort, see Section 5.8).

A `cmat` can be "pre-greased", which just means, that a certain number of linear combinations of its rows is already precomputed (see Section 5.8). In that case, the object is in the additional filter `HasGreaseTab` and the following components are bound additionally:

Component name	Content
`greaselev`	the grease level
`greasetab`	the grease table, a list of lists of `cvecs`
`greaseblo`	the number of greasing blocks
`spreadtab`	a lookup table for indexing the right linear combination

Table: Additional components of a `cmat` object when pre-greased

3.4 External Representation of Matrices on Storage

3.4.1 Byte ordering and word length

This section describes the external representation of matrices. A special data format is needed here, because of differences between computer architectures with respect to word length (32bit/64bit) and endianess. The term "endianess" refers to the fact, that different architectures store their long words in a different way in memory, namely they order the bytes that together make up a long word in different orders.

The external representation is the same as the internal format of a 32bit machine with little endianess, which means, that the lower significant bytes of a `Word` are stored in lower addresses. The reasons for this decision are firstly that 64bit machines can do the bit shifting to convert between internal and external representation easier using their wide registers, and secondly, that the nowadays most popular architectures i386 and AMD64 use both little endianess, such that conversion is only necessary on a minority of machines.

3.4.2 The header of a `cmat` file

The header of a `cmat` file consists of 5 words of 64bit each, that are stored in little endian byte order:

the magic value "`GAPCMat1`" as ASCII letters (8 bytes) in this order
the value of p to describe the base field
the value of d to describe the base field
the number of rows of the matrix
the number of columns of the matrix

Table: Header of a `cmat` file

After these 40 bytes follow the data words as described above using little endian 32bit `Words` as in the internal representation on 32bit machines.

Note that the decision to put not more than twice as many prime field elements into a 64bit `Word` than would fit into a 32bit `Word` makes the conversion between internal and external representation much easier to implement.

Chapter 4

Vectors

See Section 3.2 for a documentation of the data format of `cvecs` and its restrictions.

4.1 Creation

Note that many functions described later in this chapter implicitly create new `cvecs`, such that it is usually only in the beginning of a calculation necessary to create `cvecs` explicitly.

4.1.1 CVec

▷ `CVec(`*`arg`*`)` (operation)

Returns: a new mutable `cvec`

Creates a new `cvec`. See the method descriptions for details.

4.1.2 CVec (cvecclass)

▷ `CVec(`*`cvecclass`*`)` (method)

Returns: a new mutable `cvec`

`cvecclass` must be a `cvecclass`. Creates a new `cvec` containing only zeroes. For an explanation of the term `cvecclass` see Section 3.2 and `CVecClass` (4.1.12).

4.1.3 CVec (coefflcvecclass)

▷ `CVec(`*`coeffs, cvecclass`*`)` (method)

Returns: a new mutable `cvec`

This method takes a coefficient list and a `cvecclass` as arguments. Assume the vector will be over $GF(p,d)$ with $q = p^d$. For the coefficient list *`coeffs`* there exist several possibilities, partially depending on the base field size. The easiest way is to use GAP finite field elements, which will be put into the new vector in the same order. If $d = 1$, one can always use GAP immediate integers between 0 and $p-1$, the vector will then contain the corresponding cosets in $GF(p) = Z/pZ$. If q is small enough to be a GAP immediate integer, then one can use GAP immediate integers that are equal to the p-adic expansion using the coefficients of the representing polynomial as coefficients. That is, if an element in $GF(p,d)$ is represented by the polynomial $\sum_{i=0}^{d-1} a_i x^i$ with $a_i \in \{0,\dots,p-1\}$, then one has to put $\sum_{i=0}^{d-1} a_i p^i$ as integer into the coefficient list *`coeffs`*. If q is larger, then *`coeffs`* must be a list of lists of length d and contains for each field element of $GF(p,d)$ in the vector a list of its d

coefficients of the representing polynomial. For an explanation of the term `cvecclass` see Section 3.2 and `CVecClass` (4.1.12). Of course, the length of the list *coeffs* must match the length of the vector given in the `cvecclass`.

4.1.4 CVec (coefflpd)

▷ `CVec(coeffs, p, d)` (method)

Returns: a new mutable `cvec`

This method takes a coefficient list and two positive integers *p* and *d* as arguments. A new `cvec` over $GF(p,d)$ will be created. Let $q := p^d$.

For a description of the possible values of the coefficient list *coeffs* see the description of the method `CVec` (4.1.3).

4.1.5 CVec (compvec)

▷ `CVec(v)` (method)

Returns: a new `cvec`

v must be a GAP compressed vector either over $GF(2)$ or $GF(p,d)$ with $p^d \leq 256$. Creates a new `cvec` containing the same numbers as *v* over the field which the `Field` operation returns for *v*.

4.1.6 CVec (coefflff)

▷ `CVec(coeffs, f)` (method)

Returns: a new mutable `cvec`

This method takes a coefficient list and a finite field *f* as arguments. A new `cvec` over *f* will be created. Let $q :=$`Size(f)`.

For a description of the possible values of the coefficient list *coeffs* see the description of the method `CVec` (4.1.3).

After having encountered the concept of a `cvecclass` already a few times it is time to learn how to create one. The following operation is used first to create new `cvecclasses` and second to ask a `cvec` for its class. In addition, it is used for `cscas`.

4.1.7 CVecClass

▷ `CVecClass(arg)` (operation)

Returns: a `cvecclass`

Creates new `cvecclasses` and asks `cvecs` for their class. See the following method descriptions for details. For an explanation of the term `cvecclass` see Section 3.2.

4.1.8 CVecClass

▷ `CVecClass(p, d, l)` (method)

Returns: a `cvecclass`

All three arguments must be integers. The arguments *p* and *d* must be positive and describe the base field $GF(p,d)$. The third argument must be non-negative and the method returns the `cvecclass` of vectors over $GF(p,d)$ of length *l*.

For an explanation of the term `cvecclass` and its data structure see Section 3.2.

4.1.9 CVecClass

▷ `CVecClass(`*`v`*`)` (method)

Returns: a `cvecclass`

The argument *`v`* must be a `cvec`. The method returns the corresponding `cvecclass`. For an explanation of the term `cvecclass` and its data structure see Section 3.2.

4.1.10 CVecClass

▷ `CVecClass(`*`v, l`*`)` (method)

Returns: a `cvecclass`

The argument *`v`* must be a `cvec`. The method returns the corresponding `cvecclass` for vectors over the same field as *`v`* but with length *`l`*. This is much faster than producing the same object by giving p and d. For an explanation of the term `cvecclass` and its data structure see Section 3.2.

4.1.11 CVecClass

▷ `CVecClass(`*`c, l`*`)` (method)

Returns: a `cvecclass`

The argument *`c`* must be a `cvecclass`. The method returns the corresponding `cvecclass` for vectors over the same field as those in *`c`* but with length *`l`*. This is much faster than producing the same object by giving p and d. For an explanation of the term `cvecclass` and its data structure see Section 3.2.

4.1.12 CVecClass

▷ `CVecClass(`*`f, l`*`)` (method)

Returns: a `cvecclass`

The argument *`f`* must be a `finite field`. The method returns the corresponding `cvecclass` for vectors over the field *`f`* with length *`l`*. For an explanation of the term `cvecclass` and its data structure see Section 3.2.

4.1.13 ZeroSameMutability (cvec)

▷ `ZeroSameMutability(`*`v`*`)` (method)

Returns: the zero `cvec` in the same `cvecclass` as *`v`*

`v` must be a `cvec`.

4.1.14 ShallowCopy (cvec)

▷ `ShallowCopy(`*`v`*`)` (method)

Returns: a mutable copy of *`v`*

`v` must be a `cvec`.

4.1.15 Randomize (cvec)

▷ `Randomize(`*`v`*`)` (method)

▷ `Randomize(`*`v, rs`*`)` (method)

Returns: the vector *`v`*

`v` must be a `cvec` and `rs` must be a random source object if given. This method changes the vector `v` in place by (pseudo) random values in the field over which the vector lives. If a random source is given, the pseudo random numbers used are taken from this source, otherwise the global random source in the GAP library is taken.

4.2 Arithmetic

Of course, the standard arithmetic infix operations $+$, $-$ and $*$ (for vectors and scalars) work as expected by using the methods below. We start this section with a subsection on the usage of scalars in arithmetic operations involving vectors.

4.2.1 Handling of scalars in arithmetic operations

In all places (like in `\*`) where vectors and scalars occur, the following conventions apply to scalars:

GAP finite field elements can be used as scalars.

Integers between 0 and $p-1$ (inclusively) can always be used as scalars representing prime field elements via the isomorphism $GF(p) = Z/pZ$, also for extension fields. Larger integers than $p-1$, as long as they are GAP immediate integers, are interpreted as the p-adic expansion of the coefficient list of the polynomial representing the scalar. Note that this usage of immediate integers differs from the standard list arithmetic in GAP because multiplication with the integer n not necessarily means the same than n times addition! Larger integers than immediate integers are not supported.

4.2.2 \+ (cveccvec)

▷ `\+(v, w)` (method)

Returns: the sum $v + w$ as a new `cvec`

For two `cvecs` `v` and `w`. Works as expected.

4.2.3 \- (cveccvec)

▷ `\-(v, w)` (method)

Returns: the difference $v - w$ as a new `cvec`

For two `cvecs` `v` and `w`. Works as expected.

4.2.4 AdditiveInverseSameMutability (cvec)

▷ `AdditiveInverseSameMutability(v)` (method)

▷ `\-(v)` (method)

Returns: the additive inverse of `v` as a new `cvec`

For a `cvec` `v`. Works as expected.

4.2.5 AdditiveInverseMutable (cvec)

▷ `AdditiveInverseMutable(v)` (method)

Returns: the additive inverse of `v` as a new mutable `cvec`

For a `cvec` `v`. Works as expected.

4.2.6 * (cvecsca)

▷ `\*(v, s)` (method)

▷ `\*(s, v)` (method)

Returns: the scalar multiple `s·v`

For a `cvec` *v* and a scalar *s*. For the format of the scalar see 4.2.1. Works as expected.

4.2.7 AddRowVector (cveccvecsca)

▷ `AddRowVector(v, w, s)` (method)

▷ `AddRowVector(v, w, s, fr, to)` (method)

Returns: nothing

v and *w* must be `cvecs` over the same field with equal length, *s* a scalar (see Subsection 4.2.1) and *v* must be mutable. Adds `s·w` to *v* modifying *v*. If *fr* and *to* are given, they give a hint, that *w* is zero outside positions *fr* to *to* (inclusively). The method can, if it chooses so, save time by not computing outside that range. In fact, the implemented method restricts the operation to the `Words` involved.

If either *fr* or *to* is 0 it defaults to 1 and `Length(v)` respectively.

4.2.8 MultVector (cvecsca)

▷ `MultVector(v, s)` (method)

▷ `MultVector(v, s, fr, to)` (method)

Returns: nothing

v must be a mutable `cvec` and *s* a scalar (see Subsection 4.2.1). Multiplies *v* by *s* modifying *v*. If *fr* and *to* are given, they give a hint, that *v* is zero outside positions *fr* to *to* (inclusively). The method can, if it chooses so, save time by not computing outside that range. In fact, the implemented method restricts the operation to the `Words` involved.

If either *fr* or *to* is 0 it defaults to 1 and `Length(v)` respectively.

4.2.9 ScalarProduct (cveccvec)

▷ `ScalarProduct(v, w)` (method)

Returns: the scalar product

Both *v* and *w* must be `cvecs` over the same field with equal length. The function returns the scalar product of the two vectors. Note that there is a very efficient method for prime fields with $p < 65536$. In the other cases the method taken is not extremely fast.

4.2.10 ZeroMutable (cvec)

▷ `ZeroMutable(v)` (method)

Returns: a mutable copy of the zero `cvec` in the same `cvecclass` as *v*

v must be a `cvec`.

4.2.11 ZeroVector (cvec)

▷ `ZeroVector(l, v)` (method)

Returns: a mutable copy of the zero `cvec` over the same field than *v* but with length *l* *v* must be a cvec and *l* a GAP integer.

4.3 Element access and slicing

cvecs behave to some extend like lists with respect to element access and slicing. However, they allow only actions that can be implemented efficiently and that do not change their length. In addition there is a highly optimised function to copy contiguous sections of cvecs into another cvec.

4.3.1 ELM_LIST

▷ ELM_LIST(*v*, *pos*) (method)

Returns: a finite field element

This is a method for (reading) element access of vectors. *v* must be a *cvec* and *pos* must be a positive integer not greater than the length of *v*. The finite field element at position *pos* in *v* is returned.

Note that the list access syntax "*v*[*pos*]" triggers a call to the ELM_LIST (5.2.3) operation.

4.3.2 ASS_LIST

▷ ASS_LIST(*v*, *pos*, *s*) (method)

Returns: nothing

This is a method for (writing) element access of vectors. *v* must be a *cvec* and *pos* must be a positive integer not greater than the length of *v*. *s* must be a finite field element or an integer. The finite field element at position *pos* in *v* is set to *s*.

The scalar *s* is treated exactly as described in Subsection 4.2.1.

Note that the list access syntax "*v*[*pos*] := *s*" triggers a call to the ASS_LIST (5.2.4) operation.

4.3.3 ELMS_LIST

▷ ELMS_LIST(*v*, *l*) (method)

Returns: a cvec

This is a method for (reading) slice access of vectors. *v* must be a *cvec* and *l* must be a range object or a list of positive integers not greater than the length of *v*. In both cases the list of numbers must be contiguous list of valid positions in the vector. A new cvec over the same field as *v* and with the same length as *l* is created and returned. The finite field element at i positions *l* in *v* are copied into the new vector.

Note that the list slice access syntax "*v*{*l*}" triggers a call to the ELMS_LIST (5.2.5) operation.

Note that there intentionally is no write slice access to cvecs, because in most cases this would lead to code that unnecessarily copies data around in an expensive way. Please use the following function instead:

4.3.4 CVEC_Slice

▷ CVEC_Slice(*src*, *dst*, *srcpos*, *len*, *dstpos*) (function)

Returns: nothing

`src` and `dst` must be `cvec`s over the same field. The field elements from positions `srcpos` to `srcpos` + `len` − 1 in `src` are copied to positions from `dstpos` in `dst`. Of course all positions must be within the vectors.

Note that this functions is quite efficient, however, the ranges are checked. If you want to avoid this, use `CVEC_SLICE` instead (with same calling convention), but do not complain later if crashes occur in case of illegal positions used.

4.3.5 CopySubVector

▷ `CopySubVector(src, dst, srcposs, dstposs)` (method)

Returns: nothing

Implements the operation `CopySubVector` (**Reference: CopySubVector**) for `cvecs` `src` and `dst`, that is, `srcposs` and `dstposs` must be ranges or plain lists of integers of equal length such that all numbers contained lie between 1 and the length of `src` and `dst` respectively. The result is undefined if `src` and `dst` are the same objects. The method is particularly efficient if both `srcposs` and `dstposs` are ranges with increment 1.

4.3.6 CVEC_Concatenation

▷ `CVEC_Concatenation(arg)` (method)

Returns: a new `cvec` by concatenating all arguments

This function provides concatenation of `cvec`s over the same base field. The result is a new `cvec`. A variable number of `cvec`s over the same field can be given.

4.4 Comparison of Vectors and other information

4.4.1 =

▷ `=(v, w)` (method)

Returns: `true` or `false`

Returns `true` if the `cvecs` `v` and `w` are equal. The vectors must be over the same field and must have equal length.

4.4.2 LT

▷ `LT(v, w)` (method)

Returns: `true` or `false`

Returns `true` if the `cvec` `v` is smaller than `w`. The vectors must be over the same field and must have equal length. The order implemented is very efficient but is *not* compatible with lexicographic ordering of lists of finite field elements! This method should therefore only be used for binary search purposes. Note that the operation `LT` is the same as \<.

4.4.3 IsZero

▷ `IsZero(v)` (method)

Returns: `true` or `false`

Returns `true` if the `cvec` `v` is equal to zero, meaning that all entries are equal to zero.

4.4.4 PositionNonZero

▷ `PositionNonZero(`*v*`)` (method)

Returns: a positive integer

Returns the index of the first entry in the `cvec` *v* that is not equal to zero. If the vector is equal to zero, then its `Length` plus one is returned.

4.4.5 PositionLastNonZero

▷ `PositionLastNonZero(`*v*`)` (method)

Returns: a non-negative integer

Returns the index of the last entry in the `cvec` *v* that is not equal to zero. If the vector is equal to zero, then 0 is returned.

4.4.6 Length

▷ `Length(`*v*`)` (method)

Returns: a non-negative integer

Returns the length of the `cvec` *v*.

4.5 Changing representation, Unpacking

4.5.1 Unpack (cvec)

▷ `Unpack(`*v*`)` (method)

Returns: a list of finite field elements

This operation unpacks a `cvec` *v*. A new plain list containing the corresponding numbers as GAP finite field elements. Note that the vector *v* remains unchanged.

4.5.2 IntegerRep (cvec)

▷ `IntegerRep(`*v*`)` (method)

Returns: a list of integers or of lists of integers

This operation unpacks a `cvec` *v* into its integer representation. A new plain list containing the corresponding numbers of the vector is returned. The integer representation of a finite field element is either one integer (containing the p-adic expansion of the polynomial representative of the residue class modulo the Conway polynomial, if the field has less or equal to 65536 elements. For larger finite fields each field element is represented as a list of d integers between 0 and $p-1$, where d is the degree of the finite field over its prime field. Note that the vector *v* remains unchanged.

4.5.3 NumberFFVector (cvec)

▷ `NumberFFVector(`*v*`, `*sz*`)` (method)

Returns: an integer

This implements the library operation `NumberFFVector` (**Reference: NumberFFVector**). Note that only the case that *sz* is the number of elements in the base field of *v* is implemented. There is an inverse operation called `CVecNumber` (4.5.4).

4.5.4 CVecNumber

▷ `CVecNumber(nr, cl)` (operation)
▷ `CVecNumber(nr, p, d, l)` (operation)

Returns: a `cvec`

This is the inverse operation to `NumberFFVector` (**Reference: NumberFFVector**). Of course, the base field of the vector and its length has to be specified, either by giving a `cvecclass` *cl* or the parameters *p*, *d*, and *l*. For both cases corresponding methods are available.

4.6 Access to the base field

4.6.1 BaseDomain (cvec)

▷ `BaseDomain(v)` (method)

Returns: the base field of *v*

For a `cvec` *v* this returns the GAP object `GF(p,d)`.

4.6.2 BaseField (cvec)

▷ `BaseField(v)` (method)

Returns: the base field of *v*

For a `cvec` *v* this returns the GAP object `GF(p,d)`.

4.6.3 Characteristic (cvec)

▷ `Characteristic(v)` (method)

Returns: the characteristic of the base field of *v*

Returns the characteristic of the base field of *v* (see `BaseField` (4.6.2)).

4.6.4 DegreeFFE (cvec)

▷ `DegreeFFE(v)` (method)

Returns: the degree of the base field of *v* over its prime field

Returns the degree of the base field of *v* over its prime field (see `BaseField` (4.6.2)).

4.7 Hashing techniques for `cvecs`

4.7.1 CVEC_HashFunctionForCVecs

▷ `CVEC_HashFunctionForCVecs(v, data)` (function)

Returns: an integer hash value

This is a hash function usable for the `ChooseHashFunction` (**orb: ChooseHashFunction**) framework. It takes as arguments a `cvec` *v* and a list *data* of length 2. The first entry of *data* is the length of the hash table used and the second entry is the number of bytes looked at in the `cvec`.

4.7.2 ChooseHashFunction

▷ `ChooseHashFunction(`*`v, hashlen`*`)` (method)

Returns: a record describing a hash function

Chooses a hash function to store `cvecs` like *`v`* in a hash table of length *`hashlen`*. The return value is a record with components `func` and `data` bound to `CVEC_HashFunctionForCVecs` (4.7.1) and a list of length 2 to be given as a second argument to `CVEC_HashFunctionForCVecs` (4.7.1). This allows to use `cvecs` in the `ChooseHashFunction` (**orb: ChooseHashFunction**) framework.

Chapter 5

Matrices

A compressed matrix (a `cmat`) behaves very much like a list of `cvecs`. However, it insists on having only `cvecs` of the same length and over the same base field as its elements, and it insists on being a list without holes. Apart from these restrictions, you can use all the standard list operations with `cmats` (see Section 5.2.

In the rest of this chapter, we document all methods for matrices for the sake of completeness. If they behave exactly as is to be expected by the already defined operation no further explanation is given.

5.1 Creation

The basic operation to create new `cmats` is `CMat`, for which a variety of methods is available:

5.1.1 CMat

▷ `CMat(`*`l, cl, dochecks`*`)` (method)

▷ `CMat(`*`l, cl`*`)` (method)

Returns: a new `cmat`

A new `cmat` is created with rows being in the `cvecclass` *`cl`*. All elements of the list *`l`* must be `cvecs` in that class. The boolean flag *`dochecks`* indicates, whether this should be checked or not. If the flag is omitted, checks are performed. Note that *`l`* may be the empty list.

5.1.2 CMat

▷ `CMat(`*`l, dochecks`*`)` (method)

▷ `CMat(`*`l`*`)` (method)

Returns: a new `cmat`

A new `cmat` is created with rows being in the `cvecclass` of the vectors in *`l`*. All elements of the list *`l`* must be `cvecs` in the same class. The boolean flag *`dochecks`* indicates, whether this should be checked or not. If the flag is omitted, checks are performed. Note that *`l`* may not be the empty list.

5.1.3 CMat

▷ `CMat(`*`l, v`*`)` (method)

Returns: a new `cmat`

A new `cmat` is created with rows being in the `cvecclass` of the `cvec` *`v`*. All elements of the list *`l`* must be `cvec`s in the that same class. This is checked. Note that *`l`* may be the empty list.

5.1.4 CMat

▷ `CMat(`*`m`*`)` (method)

Returns: a new `cmat`

Creates a new `cmat` which is equal to *`m`*, which must be a compressed matrix in the filter `IsGF2MatrixRep` or the filter `Is8BitMatrixRep`.

There are some methods to create `cmat`s of special form:

5.1.5 CVEC_ZeroMat

▷ `CVEC_ZeroMat(`*`rows, cl`*`)` (function)

▷ `CVEC_ZeroMat(`*`rows, cols, p, d`*`)` (function)

Returns: a new `cmat`

Creates a zero matrix with *`rows`* rows and *`cols`* columns over the field `GF(`*`p`*`,`*`d`*`)`. If a `cvecclass` *`cl`* is given, the number of columns and the field follow from that.

5.1.6 CVEC_IdentityMat

▷ `CVEC_IdentityMat(`*`cl`*`)` (function)

▷ `CVEC_IdentityMat(`*`n, p, d`*`)` (function)

Returns: a new `cmat`

Creates an identity matrix with *`n`* rows and columns over the field `GF(`*`p`*`,`*`d`*`)`. If a `cvecclass` *`cl`* is given, the number of columns and the field follow from that.

5.1.7 CVEC_RandomMat

▷ `CVEC_RandomMat(`*`rows, cl`*`)` (function)

▷ `CVEC_RandomMat(`*`rows, cols, p, d`*`)` (function)

Returns: a new `cmat`

Creates a random matrix with *`rows`* rows and *`cols`* columns over the field `GF(`*`p`*`,`*`d`*`)`. If a `cvecclass` *`cl`* is given, the number of columns and the field follow from that. Note that this is not particularly efficient.

5.1.8 MutableCopyMat

▷ `MutableCopyMat(`*`m`*`)` (method)

Returns: a mutable copy of *`m`*

Creates a mutable copy of the `cmat` *`m`*.

5.1.9 Matrix

▷ `Matrix(`*`vectorlist, vector`*`)` (method)

▷ `MatrixNC(`*`vectorlist, vector`*`)` (method)

Returns: a new mutable `cmat`

Returns a new `cmat` containing the vectors in *`vectorlist`* as rows. The elements in *`vectorlist`* must be vectors of the same length as the sample vector *`vector`* and must live over the same base field. The sample vector is always necessary to be able to use the method selection. The *`vectorlist`* may be empty. The NC method does not check the inputs.

5.2 Matrices as lists

In this section, arguments named *`m`* and *`n`* are `cmats` and *`v`* and *`w`* are `cvecs` that fit into the corresponding matrices. *`pos`* is an integer between 1 and `Length(m)` if it applies to the matrix *`m`*.

5.2.1 Add

▷ `Add(m, v[, pos])` (method)

Returns: nothing

Behaves exactly as expected. Note that one can only add `cvecs` of the right length and over the right field.

5.2.2 Remove

▷ `Remove(m[, pos])` (method)

Returns: a `cvec`

Behaves exactly as expected. No holes can be made.

5.2.3 ELM_LIST

▷ `ELM_LIST(m, pos)` (method)

Returns: a `cvec`

Behaves exactly as expected. Note that this method is triggered when one uses the (reading) syntax "`m[pos]`".

5.2.4 ASS_LIST

▷ `ASS_LIST(m, pos, v)` (method)

Returns: nothing

Behaves exactly as expected. Note that one can only assign to positions such that the resulting matrix has no holes. This method is triggered when one uses the (assignment) syntax "`m[pos] :=` ".

5.2.5 ELMS_LIST

▷ `ELMS_LIST(m, poss)` (method)

Returns: a sub `cmat`

Behaves exactly as expected: A new matrix containing a subset of the rows is returned. Note that the row vectors are the same GAP objects as the corresponding rows of *`m`*. This operation is triggered by the expression *`m{poss}`*.

5.2.6 ASSS_LIST

▷ `ASSS_LIST(`*`m, poss, vals`*`)` (method)

Returns: nothing

Behaves exactly as expected. Of course all values in *`vals`* must be `cvecs` over the correct field and the `cmat` *`m`* must be a dense list afterwards. This operation is triggered by the statement *`m`*`{`*`poss`*`} :=` *`vals`*.

5.2.7 Length

▷ `Length(`*`m`*`)` (method)

Returns: the number of rows of the `cmat` *`m`*

Behaves exactly as expected.

5.2.8 ShallowCopy

▷ `ShallowCopy(`*`m`*`)` (method)

Returns: a new matrix containing the same rows than the `cmat` *`m`*

Behaves exactly as expected. Note that the rows of the result are the very same GAP objects than the rows of the `cmat` *`m`*.

5.2.9 Collected

▷ `Collected(`*`m`*`)` (method)

Returns: the same as the collected list of the rows of *`m`*

Behaves exactly as expected. Just uses the standard `Collected` (**Reference: Collected**) on the list of rows.

5.2.10 DuplicateFreeList

▷ `DuplicateFreeList(`*`m`*`)` (method)

Returns: a new mutable `cmat` containing the rows of *`m`* with duplicates removed

Behaves exactly as expected. Just uses the standard `DuplicateFreeList` (**Reference: DuplicateFreeList**) on the list of rows.

5.2.11 Append

▷ `Append(`*`m, n`*`)` (method)

Returns: nothing

Behaves exactly as expected. Of course, the `cmats` *`m`* and *`n`* must be over the same field and have the same number of columns. Note that the rows of *`n`* themselves (and no copies) will be put into the matrix *`m`*.

5.2.12 Filtered

▷ `Filtered(`*`m, f`*`)` (method)

Returns: a new `cmat` containing some of the rows of *`m`*

Behaves exactly as expected. The function *`f`* will be called for each row of *`m`*.

5.2.13 Unbind

▷ `Unbind(m, f)` (method)

Returns: nothing

Behaves exactly as expected. Of course, only the last bound row may be unbound.

5.3 Arithmetic

Of course, the standard arithmetic infix operations $+$, $-$ and $*$ (for vectors and scalars) work as expected by using the methods below. The comments on the usage of scalars in arithmetic operations involving vectors from Subsection 4.2.1 apply analogously.

5.3.1 \+ (cmatcmat)

▷ `\+(m, n)` (method)

Returns: the sum $m+n$ as a new `cmat`

For two `cmat`s *m* and *n*. Works as expected.

5.3.2 \- (cmatcmat)

▷ `\-(m, n)` (method)

Returns: the difference $m-n$ as a new `cmat`

For two `cmat`s *m* and *n*. Works as expected.

5.3.3 AdditiveInverseSameMutability (cmat)

▷ `AdditiveInverseSameMutability(m)` (method)

▷ `\-(m)` (method)

Returns: the additive inverse of *m* as a new `cmat`

For a `cmat` *m*. Works as expected.

5.3.4 AdditiveInverseMutable (cmat)

▷ `AdditiveInverseMutable(m)` (method)

Returns: the additive inverse of *m* as a new mutable `cmat`

For a `cmat` *m*. Works as expected.

5.3.5 * (cmatsca)

▷ `\*(m, s)` (method)

▷ `\*(s, m)` (method)

Returns: the scalar multiple $s \cdot m$

For a `cmat` *m* and a scalar *s*. For the format of the scalar see 4.2.1. Works as expected.

5.3.6 * (cveccmat)

▷ `\*(v, m)` (method)

Returns: the product $v \cdot m$

For a `cmat` *m* and a `cvec` *s* with the same length as the number of rows of *m*. Works as expected. Note that there is a very fast method for the case that *m* is pre-greased (see 5.8).

5.3.7 \^ (cveccmat)

▷ `\^(v, m)` (method)

Returns: the product $v \cdot m$

For a `cmat` *m* and a `cvec` *s* with the same length as the number of rows of *m*. Works as expected. Note that there is a very fast method for the case that *m* is pre-greased (see 5.8).

5.3.8 * (cmatcmat)

▷ `\*(m, n)` (method)

Returns: the product $m \cdot n$

Of course, the `cmat` *m* must have as many columns as the `cmat` *n* has rows. Works as expected. Note that there is a very fast method for the case that *n* is pre-greased (see 5.8).

5.3.9 ZeroSameMutability (cmat)

▷ `ZeroSameMutability(m)` (method)

Returns: the zero `cmat` over the same field and with the same dimensions as *m*

m must be a `cmat`.

5.3.10 ZeroMutable (cmat)

▷ `ZeroMutable(m)` (method)

Returns: a mutable copy of the zero `cmat` over the same field and with the same dimensions as *m*

m must be a `cmat`.

5.3.11 OneSameMutability (cmat)

▷ `OneSameMutability(m)` (method)

Returns: the identity `cmat` over the same field and with the same dimensions as *m*

m must be a square `cmat`.

5.3.12 OneMutable (cmat)

▷ `OneMutable(m)` (method)

Returns: a mutable copy of the identity `cmat` over the same field and with the same dimensions as *m*

m must be a square `cmat`.

5.3.13 InverseMutable

▷ `InverseMutable(`*m*`)` (method)

Returns: the multiplicative inverse of *m*

If the `cmat` is not square or not invertible then `fail` is returned. Behaves exactly as expected.

5.3.14 InverseSameMutability

▷ `InverseSameMutability(`*m*`)` (method)

Returns: the multiplicative inverse of *m*

If the `cmat` is not square or not invertible then `fail` is returned. Behaves exactly as expected.

5.3.15 TransposedMat

▷ `TransposedMat(`*m*`)` (method)

Returns: the transpose of *m*

Behaves exactly as expected.

5.3.16 KroneckerProduct

▷ `KroneckerProduct(`*m, n*`)` (method)

Returns: the Kronecker product of *m* and *n*

Behaves exactly as expected.

5.4 Comparison of matrices and other information

5.4.1 =

▷ `=(`*m, n*`)` (method)

Returns: `true` or `false`

Returns `true` if the `cmats` *m* and *n* are equal. The matrices must be over the same field and must have equal dimensions.

5.4.2 LT

▷ `LT(`*m, n*`)` (method)

Returns: `true` or `false`

Returns `true` if the `cmat` *m* is smaller than *n*. The matrices must be over the same field and must have equal dimensions. The method implements the lexicographic order and uses `LT` for the ordering of vectors. Note that the operation `LT` is the same as \<.

5.4.3 IsZero

▷ `IsZero(`*m*`)` (method)

Returns: `true` or `false`

Returns `true` if the `cmat` *m* is equal to zero, meaning that all entries are equal to zero.

5.4.4 IsOne

▷ `IsOne(m)` (method)

Returns: `true` or `false`

Returns `true` iff the `cmat` *m* is equal to the identity matrix.

5.4.5 IsDiagonalMat

▷ `IsDiagonalMat(m)` (method)

Returns: `true` or `false`

Returns `true` iff the `cmat` *m* is a diagonal matrix.

5.4.6 IsUpperTriangularMat

▷ `IsUpperTriangularMat(m)` (method)

Returns: `true` or `false`

Returns `true` iff the `cmat` *m* is an upper triangular matrix.

5.4.7 IsLowerTriangularMat

▷ `IsLowerTriangularMat(m)` (method)

Returns: `true` or `false`

Returns `true` iff the `cmat` *m* is a lower triangular matrix.

5.4.8 CVEC_HashFunctionForCMats

▷ `CVEC_HashFunctionForCMats(m, data)` (function)

Returns: an integer hash value

This is a hash function usable for the `ChooseHashFunction` (**orb: ChooseHashFunction**) framework. It takes as arguments a `cmat` *m* and a list *data* of length 2. The first entry of *data* is the length of the hash table used and the second entry is the number of bytes looked at in the `cvecs` in the matrices.

5.4.9 ChooseHashFunction

▷ `ChooseHashFunction(m, l)` (method)

Returns: a record with entries `func` and `data`.

Chooses a hash function to be used for `cmats` like *m* (that is, over the same field with the same number of columns) and for hash tables of length *l*. The hash function itself is stored in the `func` component of the resulting record. The hash function has to be called with two arguments: the first must be a matrix like *m* and the second must be the value of the `data` component of the resulting record.

5.5 Slicing and submatrices

As described in Section 5.2 you can use the slicing operator `\{\}` for read and write access of a subset of the rows of a `cmat`. However, the double slicing operator is not supported. The reason for this is twofold: First there is a technical issue that the double slicing operator cannot easily be overloaded in the GAP system. The second is, that very often the double slicing operator is used to copy a part of one matrix to another part of another matrix using double slicing on both sides of an assignment. This is quite inefficient because it creates an intermediate object, namely the submatrix which is extracted.

Therefore we have chosen to support submatrix access through two operations `ExtractSubMatrix` (5.5.1) and `CopySubMatrix` (5.5.2) described below.

5.5.1 ExtractSubMatrix

▷ `ExtractSubMatrix(`*m, rows, cols*`)` (operation)

Returns: a submatrix of *m*

This operation extracts the submatrix of the matrix *m* consisting of the rows described by the integer list (or range) *rows* and of the columns described by the integer list (or range) *cols*. This is thus equivalent to the usage *m*`{`*rows*`}{`*cols*`}`. Note that the latter does not work for `cmat`s, whereas a quite efficient method for `ExtractSubMatrix` is available for `cmat`s.

5.5.2 CopySubMatrix

▷ `CopySubMatrix(`*src, dst, srows, drows, scols, dcols*`)` (operation)

Returns: nothing

This operation extracts the submatrix of the matrix *src* consisting of the rows described by the integer list (or range) *srows* and of the columns described by the integer list (or range) *scols* and copies it into the submatrix of *dst* described by the integer lists (or ranges) *drows* and *dcols*. No intermediate object is created. This is thus equivalent to the usage *dst*`{`*drows*`}{`*dcols*`} :=` *src*`{`*srows*`}{`*scols*`}`. Note that the latter does not work for `cmat`s, whereas a quite efficient method for `CopySubMatrix` is available for `cmat`s.

5.6 Information about matrices

5.6.1 BaseField (cmat)

▷ `BaseField(`*m*`)` (method)

Returns: the base field of *m*

For a `cmat` *m* this returns the GAP object `GF(p,d)` corresponding to the base field of *m*. Note that this is a relatively fast lookup.

5.6.2 Characteristic (cmat)

▷ `Characteristic(`*m*`)` (method)

Returns: the characteristic of the base field of *m*

Returns the characteristic of the base field of *m* (see `BaseField` (5.6.1)).

5.6.3 DegreeFFE (cmat)

▷ `DegreeFFE(`*m*`)` (method)

Returns: the degree of the base field of *m* over its prime field

Returns the degree of the base field of *m* over its prime field (see `BaseField` (5.6.1)).

5.6.4 DefaultField (cmat)

▷ `DefaultField(`*m*`)` (method)

Returns: the base field of *m*

For a `cmat` *m* this returns the GAP object `GF(p,d)` corresponding to the base field of *m*. Note that this is a relatively fast lookup.

5.7 Input and output

5.7.1 CVEC_WriteMat

▷ `CVEC_WriteMat(`*f, m*`)` (method)

Returns: `true` or `fail`

f must be a file object from the IO package (see `IsFile` (**IO: IsFile**)) and *m* must be a `cmat`. The matrix *m* is written to the file *f*. Note that the format (see Section 3.4) is platform independent, such that matrices can be exchanged between different architectures. The result is `true` or `fail` depending on whether everything worked or an error occurred respectively.

5.7.2 CVEC_WriteMatToFile

▷ `CVEC_WriteMatToFile(`*fn, m*`)` (method)

Returns: `true` or `fail`

fn must be a string object containing a file name and *m* must be a `cmat`. The matrix *m* is written to the file with name *fn* on the local storage. Note that the format (see Section 3.4) is platform independent, such that matrices can be exchanged between different architectures. The result is `true` or `fail` depending on whether everything worked or an error occurred respectively.

5.7.3 CVEC_WriteMatsToFile

▷ `CVEC_WriteMatsToFile(`*fnpref, l*`)` (method)

Returns: `true` or `fail`

fnpref must be a string object containing a file name prefix and *m* must be a list of `cmats`. The matrices in *l* are written to the files with names determined by using the string *fnpref* and appending the natural numbers from 1 to the length of *l* on the local storage. Note that the format (see Section 3.4) is platform independent, such that matrices can be exchanged between different architectures. The result is `true` or `fail` depending on whether everything worked or an error occurred respectively.

5.7.4 CVEC_ReadMat

▷ `CVEC_ReadMat(`*f*`)` (method)

Returns: a `cmat` or `fail`

f must be a file object from the IO package (see `IsFile` (**IO: IsFile**)). A matrix is read from the file *f*. Note that the format (see Section 3.4) is platform independent, such that matrices can be exchanged between different architectures. The result is `fail` if an error occurred.

5.7.5 CVEC_ReadMatFromFile

▷ `CVEC_ReadMatFromFile(`*`fn`*`)` (method)

Returns: a `cmat` or `fail`

`fn` must be a string object containing a file name. A matrix is read from the file with name *`fn`* on the local storage. Note that the format (see Section 3.4) is platform independent, such that matrices can be exchanged between different architectures. The result is `fail` if an error occurred.

5.7.6 CVEC_ReadMatsFromFile

▷ `CVEC_ReadMatsFromFile(`*`fnpref`*`)` (method)

Returns: a list of `cmats` or `fail`

`fnpref` must be a string object containing a file name prefix. A list of matrices is read from the files with names determined by using the string *`fnpref`* and appending the natural numbers from 1 on from the local storage. The number of matrices read is determined by the highest number such that the corresponding filename exists in the filesystem. Note that the format (see Section 3.4) is platform independent, such that matrices can be exchanged between different architectures. The result is `fail` if an error occurred.

5.8 Grease

The basic idea behind the “grease” technique is that over a finite field there are only finitely many linear combinations of a fixed list of vectors. Thus, many operations including matrix multiplication can be speeded up by precomputing all possible linear combinations and then just looking up the right one. For the case of matrix multiplication this can for example gain a factor of about 4 over the field with 2 elements using “grease level8”, which means that for blocks of 8 rows all linear combinations are precomputed.

The **cvec** uses grease whenever appropriate automatically for example for matrix multiplication. However, occasionally the user has to take a conscious decision, which matrices to grease, because this of course uses more memory.

A `cmat` can be “pre-greased” with level l, which means that it is chopped into chunks of l rows and for each such chunk all possible linear combinations are precomputed and stored. This increases the memory used to store the matrix by a factor of q^l if the base field of the matrix has q elements. However, operations like vector matrix multiplication and matrix matrix multiplication (here the right hand side matrix must be greased!) are speeded up. As a rule of thumb the factor one can hope for is about $l \cdot (q-1)/q$. Note that for big matrices matrix multiplication does greasing on the fly anyway and therefore one cannot hope for such a big factor by pre-greasing.

5.8.1 GreaseMat

▷ `GreaseMat(`*`m, l`*`)` (operation)

▷ `GreaseMat(`*`m`*`)` (operation)

Returns: nothing

`m` must be a `cmat`. It is pregreased with level *`l`*. Without the argument *`l`* a grease level depending of the field size is chosen automatically. Note that the matrix will need much more memory when pregreased.

5.8.2 UnGreaseMat

▷ `UnGreaseMat(m)` (operation)

Returns: nothing

m must be a `cmat`. The pregreased information is deleted. This can save a lot of memory.

5.9 Everything else

5.9.1 Randomize (cmat)

▷ `Randomize(m)` (method)

▷ `Randomize(m, rs)` (method)

Returns: the matrix *m*

m must be a `cmat` and *rs* must be a random source object if given. This method changes the matrix *m* in place by (pseudo) random values in the field over which the matrix lives. If a random source is given, the pseudo random numbers used are taken from this source, otherwise the global random source in the GAP library is taken.

5.9.2 OverviewMat (cmat)

▷ `OverviewMat(m)` (function)

Returns: nothing

An ASCII art overview over the `cmat` *m* is printed. Stars indicate nonzero blocks in the matrix and dots zero blocks.

5.9.3 Unpack (cmat)

▷ `Unpack(m)` (method)

Returns: a list of lists of finite field elements

This operation unpacks a `cmat` *m*. A new plain list of plain lists containing the corresponding numbers as GAP finite field elements. Note that the matrix *m* remains unchanged.

Chapter 6

Linear Algebra routines

6.1 Semi echelonised bases and cleaning

A semi echelonised basis is a record with the following components: `vectors` is a list of vectors of equal length, optionally (and optimally) they can be wrapped to a matrix. `pivots` is a list of integers. The number *i* in position *j* indicates that the *j*th vector in `vectors` has a one in column number *i* and all vectors with a number bigger than *i* in `vectors` have a zero in column number *i*.

Note that for technical reasons another component `helper` is bound containing a `cvec` of length 1 over the same field.

Note further that the output of the GAP library operation `SemiEchelonMat` (**Reference: SemiEchelonMat**) is not compatible to this setup, because it contains a component `heads` that contains at position *i*, if nonzero, the number of the row for that the pivot element is in column *i*.

6.1.1 EmptySemiEchelonBasis

▷ `EmptySemiEchelonBasis(v)` (method)

Returns: a new mutable empty semi echelonised basis

The argument *v* must be a sample vector or matrix. If it is a matrix, then one of its rows is taken as sample vector.

6.1.2 Vectors

▷ `Vectors(b)` (operation)

Returns: a matrix

The argument *b* must be a semi echelonised basis. This operation returns a (mutable) matrix whose rows are the basis vectors.

6.1.3 Length (for a semi echelonised basis)

▷ `Length(b)` (operation)

Returns: an integer

The argument *b* must be a semi echelonised basis. This operation returns the number of vectors in the basis.

6.1.4 CleanRow

▷ `CleanRow(b, v, extend, dec)` (method)

Returns: `true` or `false`

This is the basic operation for cleaning with a semi echelonised basis *b*. The vector *v* is cleaned with the vectors in the semi echelonised basis. The function returns `true` if *v* lies in the span of *b* and false otherwise.

The boolean value *extend* indicates, whether the basis should be extended in the latter case by the newly cleaned vector.

The argument *dec* is either `fail` in which case it is not used or a vector over the same field as *v* that is at least as long as the number n of vectors in *b* (plus one if *extend* is `true`). In this case, the first n components of *dec* are changed such that $\sum_{i=1}^{n} dec_i b_i$. If *extend* is `true` and *v* is not contained in the span of the vectors in *b* and *dec* is a vector, then the first $n+1$ components of *dec* are changed such that $v = \sum_{i=1}^{n+1} dec_i b_i$.

6.1.5 SemiEchelonBasisMutable

▷ `SemiEchelonBasisMutable(b)` (method)

Returns: a semi echelonised basis

Turns the output *b* of `SemiEchelonMat` (**Reference: SemiEchelonMat**) into a valid semi echelonised basis in the sense of the cvec package. This means that the component `pivots` is calculated from the component `heads`. Use this function only if absolutely necessary. Instead, directly invoke `SemiEchelonBasisMutable` on the original matrix.

6.1.6 SemiEchelonBasisMutable

▷ `SemiEchelonBasisMutable(m)` (method)

Returns: a semi echelonised basis

The argument *m* must be a `cmat`. This method calculates a semi echelonised basis for the row space of *m*.

There are a number of similar operations the names of which are derived from `SemiEchelonBasisMutable` by appending single letters: The letters “P”, “T” and “X” are modifiers and there are operations for most of the 8 combinations of those letters being present or not respectively. Always give the present letters in alphabetical order.

The “X” indicates, that the input matrix may be destroyed, that is, the rows of *m* will be changed and the very same objects be used in the semi echelonised result basis.

The “T” indicates, that the transformation is calculated, that is, the resulting record `r` will have a component `coeffs`, such that `r.coeffs * m` is equal to `r.vectors` component of the semi echelonised result. Further, with given letter “T” there will be a component `relations` which is a basis for the (left) nullspace of *m*.

The “P” indicates, that a component `r.p` is calculated such that `r.p * r.vectors` is the original matrix *m*.

Currently the variants with “P” and “T” both present are not implement because they will probably not be very useful.

6.1.7 SemiEchelonBasisMutableX

▷ `SemiEchelonBasisMutableX(m)` (method)

Returns: a semi echelonised basis

Same as `SemiEchelonBasisMutable` (6.1.6) but destructive in *m*.

6.1.8 SemiEchelonBasisMutableT

▷ `SemiEchelonBasisMutableT(m)` (method)

Returns: a semi echelonised basis

Same as `SemiEchelonBasisMutable` (6.1.6) but in addition stores the transformation, see `SemiEchelonBasisMutable` (6.1.6).

6.1.9 SemiEchelonBasisMutableTX

▷ `SemiEchelonBasisMutableTX(m)` (method)

Returns: a semi echelonised basis

Same as `SemiEchelonBasisMutableT` (6.1.8) but destructive in *m*.

6.1.10 SemiEchelonBasisMutableP

▷ `SemiEchelonBasisMutableP(m)` (method)

Returns: a semi echelonised basis

Same as `SemiEchelonBasisMutable` (6.1.6) but in addition stores the inverse transformation, see `SemiEchelonBasisMutable` (6.1.6).

6.1.11 SemiEchelonBasisMutablePX

▷ `SemiEchelonBasisMutablePX(m)` (method)

Returns: a semi echelonised basis

Same as `SemiEchelonBasisMutableP` (6.1.10) but destructive in *m*.

6.1.12 MutableNullspaceMat

▷ `MutableNullspaceMat(m)` (method)

Returns: a `cmat`

Returns a `cmat` the rows of which are a basis of the (left) nullspace of the `cmat` *m*. Internally, `SemiEchelonBasisMutableT` (6.1.8) is used and the component `relations` of the result is returned. The result is mutable, which is the reason for the introduction of a new operation besides `NullspaceMat` (**Reference: NullspaceMat**).

6.1.13 MutableNullspaceMatX

▷ `MutableNullspaceMatX(m)` (method)

Returns: a `cmat`

Same as `MutableNullspaceMat` (6.1.12) but destructive in *m*.

6.1.14 NullspaceMat

▷ `NullspaceMat(`*`m`*`)` (method)

Returns: an immutable `cmat`

Behaves exactly as expected. *`m`* must be a `cmat`.

6.1.15 NullspaceMatDestructive

▷ `NullspaceMatDestructive(`*`m`*`)` (method)

Returns: an immutable `cmat`

Behaves exactly as expected. *`m`* must be a `cmat`.

6.2 Characteristic and minimal polynomial

6.2.1 CharacteristicPolynomialOfMatrix

▷ `CharacteristicPolynomialOfMatrix(`*`m`*`)` (method)

▷ `CharacteristicPolynomialOfMatrix(`*`m, indetnr`*`)` (method)

Returns: a record

Calculates the characteristic polynomial of the `cmat` *`m`* over its base field. Because during the calculations the polynomial automatically comes as a product of some not necessarily irreducible factors, the result is returned in a record with two components. The `poly` component contains the full characteristic polynomial. The `factors` component contains a list of not necessarily irreducibly factors the product of which is the characteristic polynomial. If an indeterminate number is given as second argument, the polynomials are written in that indeterminate, otherwise in the first indeterminate over the base field.

6.2.2 FactorsOfCharacteristicPolynomial

▷ `FactorsOfCharacteristicPolynomial(`*`m`*`)` (method)

▷ `FactorsOfCharacteristicPolynomial(`*`m, indetnr`*`)` (method)

Returns: a list

Calculates the characteristic polynomial of the `cmat` *`m`* over its base field. The result is a list of irreducible factors of the characteristic polynomial of *`m`*, the product of which is the characteristic polynomial. Because during the calculations the polynomial automatically comes as a product of some not necessarily irreducible factors, this is more efficient than just calculating the characteristic polynomial and then factoring it. If an indeterminate number is given as second argument, the polynomials are written in that indeterminate, otherwise in the first indeterminate over the base field.

6.2.3 MinimalPolynomialOfMatrix

▷ `MinimalPolynomialOfMatrix(`*`m`*`)` (method)

▷ `MinimalPolynomialOfMatrix(`*`m, indetnr`*`)` (method)

Returns: a polynomial over the base field of *`m`*

Calculates the minimal polynomial of the `cmat` *`m`* over its base field. The polynomial is returned. If an indeterminate number is given as second argument, the polynomials are written in that indeterminate, otherwise in the first indeterminate over the base field.

6.2.4 CharAndMinimalPolynomialOfMatrix

▷ `CharAndMinimalPolynomialOfMatrix(m)` (method)
▷ `CharAndMinimalPolynomialOfMatrix(m, indetnr)` (method)

Returns: a record

Calculates the characteristic and minimal polynomial of the `cmat` *m* over its base field. Because during the calculation of the minimal polynomial the characteristic polynomial is calculated anyway, the result is returned in a record with five components: The `charpoly` component contains the full characteristic polynomial. The `irreds` component contains the set of irreducible factors of the characteristic polynomial as a list. The `mult` component contains a corresponding list of multiplicities, that is in position i is the multiplicity of the irreducible factor number i in the characteristic polynomial. The component `minpoly` contains the minimal polynomial and the component `multmin` the corresponding multiplicities of the irreducible factors of the characteristic polynomial in the minimal polynomial. If an indeterminate number is given as second argument, the polynomials are written in that indeterminate, otherwise in the first indeterminate over the base field.

6.2.5 MinimalPolynomialOfMatrixMC

▷ `MinimalPolynomialOfMatrixMC(m, prob[, indetnr])` (operation)

Returns: a record

This method calculates the characteristic and minimal polynomials of the row list matrix *m* over its base domain. It uses the Monte Carlo algorithm by Neunhoeffer and Praeger. The second argument *prob* is an upper bound for the error probability, it can be 0 in which case a deterministic verification is done. The optional argument *indetnr* is the number of the indeterminate over the base domain to be used to express polynomials.

The result is a record with the following components bound: The component `charpoly` is the characteristic polynomial which is guaranteed to be correct. The component `minpoly` is always a divisor of the minimal polynomial and usually is equal to it. See below for details. The component `irreds` is a sorted list of the irreducible factors of the characteristic polynomial. The component `mult` is a corresponding list of the same length containing the multiplicities of these irreducible factors in the characteristic polynomial. The component `minmult` is a corresponding list of the same length containing the multiplicities of these irreducible factors in the polynomial given in `minpoly`. The component `proof` is set to `true` if the result is proved to be correct, which can happen even if *prob* was non-zero (for example in the case of a cyclic matrix). The component `iscyclic` is set to `true` if and only if the minimal polynomial is equal to the characteristic polynomial. The component `prob` is set to the probability given in *prob*, if that was zero then it is set to $1/10000$ but `proof` will be `true`. The components `A`, `B` and `S` describe a base change for *m* to a sparse form which is obtained as a byproduct. `S` is a semi echelonised basis which was obtained by a spin-up computation with *m* and if Y is the sparse basis then $Y = A \cdot S$ and $B = A^{-1}$.

The given result for the minimal polynomial could be a proper divisor of the real minimal polynomial if *prob* was non-zero, however, the probability for this outcome is guaranteed to be less than or equal to *prob*. Note that it is always guaranteed that all irreducible factors of the minimal polynomial are actually present, it can only happen that the multiplicities are too small.

Chapter 7

Performance

Here comes text.

Chapter 8

Cooperation with the GAP library

Here comes text.

Chapter 9

Examples

Here comes text.

References

Index