The C MeatAxe
Release

User's Guide
and
Technical Reference

Michael Ringe[1]

29.07.98

[1]Current address: Lehrstuhl D für Mathematik, RWTH Aachen, 52056 Aachen, Germany
E-mail: mringe@math.rwth-aachen.de

Contents

Chapter 1

User's guide

1.1 Overview

The MeatAxe is a set of programs for working with matrices over finite fields. Its primary purpose is the calculation of modular character tables, although it can be used for other purposes, such as investigating subgroup structure, module structure etc. Indeed, there is a set of programs (see section 1.6) to compute automatically the submodule lattice of a given module.

Each of the programs is self-contained, reading its input from files, and writing its output to files. To make the MeatAxe usable, therefore, it is necessary to write operating system commands to run the various programs. This documentation is primarily for the programs, and further documentation is necessary for the various implementations in differing operating environments.

The primitive objects are of two types: matrices and permutations. Permutation objects can be handled, but not as smoothly as you might expect. For example, it is hoped that programs such as split (ZSP) and multiply (ZMU) will be able to work with mixed types, but at present ZSP is restricted to matrices only, and ZMU can multiply matrix $\cdot$ permutation, but not vice versa.

Remarks on the C implementation

Most of the programs (including their documentation) in this package are based on the FORTRAN code written by Richard A. Parker. The translation from FORTRAN to C was done in 1989 and since then both versions have developed independently. The user interface of the C programs has been changed significantly (e.g., command line arguments were introduced) and some new programs have been added which are not available in the FORTRAN MeatAxe.

The objective of this suite of programs is research, so that where simplicity and power are in contention, power is selected. Nevertheless, I have tried to make the programs as straightforward and user-friendly as possible. Moreover, the source code is freely available, so be free to change any parts you are annoyed with. Refer to the programmer's guide (see chapter 2) for more information about the MeatAxe internals.

Supported platforms

The MeatAxe should run on most UNIX platforms including Linux, NetBSD, Ultrix, Solaris and HP-UX, and on Windows NT. Other platforms as MS-DOS, VM/CMS and VMS are not officially supported, but it should not be too difficult to compile and run the programs on those systems.

Users of the MeatAxe on a non-UNIX platform should be aware that their operating system may pose some additional restrictions on the programs:

1. The length and syntax of file names may be restricted.

2. CPU time information may not be avalable or incorrect.

1.2 Using the MeatAxe programs

This section contains some general information about the MeatAxe programs and how to use them.

1.2.1 Setting up the environment

It is recommended that you include the MeatAxe installation directory, where all the programs are installed, in your path. You may also define the following environment variables which are recognized by the MeatAxe programs:

Name	Meaning
`MTXLIB`	Name of the MeatAxe library directory. You need to define this variable only if you want to use a library directory different from the default library (see section 1.2.4). This can be overridden by the `-L` option.
`MTXBIN`	Name of the MeatAxe installation directory. This is necessary if the programs have been moved into another directory without recompiling and you want to use the automatic table file generation feature (see section 1.2.2). The value of this variable can be overridden by the `-B` option.

1.2.2 Arithmetic tables

All programs use lookup tables for row operations and finite field arithmetic. These tables are read from a file named

```
pXXX.zzz
```

where `XXX` is the field order. For example, tables for GF(25) are read from the file `p025.zzz`. If the table file does not exist, the program will try to execute the `maketab` program to generate the file. If that fails, too, the program stops. Table files may also be placed in the library directory (see below) because the programs look there before they try to generate a new file.

1.2.3 File types

All MeatAxe programs operate on files — normally, no interactive i/o is required during the run. There are two kinds of files:

- Binary (internal format) data files. Programs usually read and write data in binary format. For a description of the file format, see section 3.1.2. Default names are G1, G2, ... for input files and P1, P2, ... for output files.
- Text files, used by some programs as ZCV and ZPR. The format of text files is described with the ZCV program (see section 1.4.2).

Typically, a program reads its input from G1,...,Gn and writes its output to P$n+1$,...,Pm. For example, the ZTE (tensor matrices) program reads two matrices from G1 and G2 and writes their tensor product to P3. To find out which files are used by a specific program, look up the program description in section 1.3ff.

You may override the default names by specifying new file names on the command line. For example, to multiply two matrices in `mat1` and `mat2` and write the result to `result` call ZMU with 3 arguments as follows:

```
zmu mat1 mat2 result
```

1.2.4 The library directory

When given a file name, all programs look for the file in the current directory first. If the file is not found there, some programs try to find it in the library directory, provided a library has been defined. This extended search applies, for example, to ZEV and ZCV input files and also to the arithmetic table files.

The library is usually defined at compile-time by assigning a value to the symbol `MTXLIB`. This definition may be overridden at run-time by defining the environment variable `MTXLIB`. In either case `MTXLIB` must be a complete directory name including a trailing separator character (e.g., '/' for UNIX). When looking for files in the library, the programs simply append the file name to the value of `MTXLIB` in order to build the complete file name.

1.2.5 Command line syntax and common options

All MeatAxe programs expect one or more arguments, ususally the names of input and output files. Arguments may be preceeded by program options which always start with a minus sign. Options without arguments can be grouped. For example, instead of

```
chop -G -V -V module
```

you can write

```
chop -GVV module
```

Some options require an additional argument which must be separated from the option by a space. It is not recommended to group options with arguments. Here is an example:

```
chop -g 3 -T 3600 -GVV module
```

The following list contains the common options which are are recognized by all MeatAxe programs. These options are not mentioned in the program the program descriptions below.

- `-help` Print a help screen.
- `-Q` Quiet, do not print any message except errors and results.
- `-V` Verbose, print more messages. Repeat this option (e.g., `-VVV`) to see more detailed information.
- `-B` Used to specify a different binary directory. This overrides both the directory name compiled into the programs and the value of —MTXBIN—.
- `-L` Used to specify a different library directory. This overrides both the directory name compiled into the programs and the value of —MTXLIB—.
- `-o` *options* This option is used to set various options for testing and debugging the MeatAxe. Even if documented in the program descriptions, `-o` options may be changed or removed in the next release, so you should not rely on them.

The following options are not supported by all programs, but their meaning is always the same.

- `-g` Set the number of generators. This option must be followed by a positive integer.
- `-G` Generate output in GAP format.

1.3 Utility Programs

1.3.1 MAKETAB and BIGMKTAB

USAGE

`maketab` *fieldorder*
`bigmktab` *fieldorder*

Description

These programs generate lookup tables for fast field arithmetic. Tables are stored in a file (one file for each field) which is loaded by the ZZZ module at run-time. *fieldorder* must be a positive integer (a prime power, actually) less than or equal to 256 for MAKETAB, or 65536 for BIGMKTAB.

MAKETAB creates tables for the small field version, up to GF(256). The output file is named `pXXX.zzz`, where XXX is the field order. The size of this file is approximately 130 KBytes and does not depend on the field order.

BIGMKTAB creates tables for the large field version. The output file is named `pXXXXX.zzz`, where XXXXX is the field order. This file is usually (i.e., for comparable field orders) much smaller than the 'zzz' file, but its size increases linearly with q.

The table file is needed whenever a program performs finite field operations, i.e., when working with matrices. The programs look for the table file first in the current directory and then, if it is not there, in the library directory (see section 1.2.4). Thus, table files for the most commonly used fields can be kept in the library, and additional table files can be created in the current directory when they are needed. Usually, the name of the library directory is defined at compile-time, but this may be changed at any time by defining the environment variable `MTXLIB`.

1.3.2 ZZZTEST: Library test

Usage

`zzztest [-b] [-t]` *field*

Description

This program tests the finite field arithmetic and other functions of the MeatAxe library. There must be exactly one argument, *field*, specifying the field order. If the `-b` option is used, the program performs a benchmark and prints some information about the computer's speed. `-t` prints the addition and multiplication tables for the finite field.

1.4 File conversion and manipulation

1.4.1 ZCF: Change format

Usage

`zcf` q `[`*Input Result*`]`

Description

This program converts between various data types. Currently there are two kinds of conversions available:

1. If *Input* is a matrix, the field is changed to GF(q). The result is a matrix of the same size over GF(q).
2. If *Input* is a permutation of degree n, it is converted into the corresponding $(n \times n)$ permutation matrix over GF(q).

The result is written to *Result*. Default file names are G1 for input and P2 for output.

If *Input* is a matrix it can be restricted to a subfield or embedded into a larger field. In the first case, of course, the entries must already be in the subfield. The conversion is done in two steps.

First, all entries of the matrix are converted to integers. Then, they are mapped to the new field and reassembled into rows. The result is written out row by row.

In case of permutations the output matrix is generated row by row by inserting ones at the positions specified by the permutation.

LIMITS

If the input is a matrix, the whole matrix must fit into memory. Additionally, the program needs $n \cdot m \cdot s$ bytes of memory, where m and n are the dimensions of the input matrix and $s = 1$ for the small arithmetic version and $s = 2$ for the big version. In case of permutations, the input permutation and one row of the output file must fit into memory.

1.4.2 ZCV: Convert from text to binary format

USAGE

$$\texttt{zcv [}\left\{\begin{array}{c}\textit{Textfile}\\ \texttt{=}\end{array}\right\}\texttt{[}\textit{BinFile}\texttt{]]}$$

DESCRIPTION

This program reads a text file, and outputs a matrix or permutation in internal format depending on the input to *BinFile* (default: P2). If the input file name is '=', input is read from stdin. If no arguments are given, T1 is used as input file and output goes to P2. Here are some examples:

Command	Conversion
`zcv`	T1 → P2
`zcv txt`	Txt → P2
`zcv txt z1`	Txt → z1
`zcv = z1`	standard input → z1

To convert text from a file called '=' you must use input redirection:

```
zcv = binfile < =
```

Another use of this program is, together with ZPR, to convert between various types of object. See the documentation of ZPR for examples.

Text file format The text file is interpreted line by line. Empty lines are ignored completetly. If the line contains one or more '#', which may occur at any position, the first '#' and all remaining characters on this line are ingored.

Each object, for example a matrix or a permutation, consists of a one line header followed by the data. The header may be one of the following:

`matrix field=`f `rows=`m `cols=`n"	A matrix in $\mathrm{GF}(f)^{m \times n}$
`permutation degree=`m"	A permutation on m points
`polynomial field=`f `degree=`m"	A polynomial $p \in \mathrm{GF}(f)[x]$ of degree m.

The header may be given in a different format, for example

```
MeatAxeFileInfo := "matrix field=5 rows=100 cols=100";
```

Other header form,ats are supported for compatibility with older versions of the MeatAxe.

After an old type 1 header, the format of the data follwing the header is fixed. There must be at most 80 characters per line, lines must be filled as much as possible, and each row of the matrix starts with a new input line. There are no blanks allowed to separate numbers. In all other modes, the data part consists of a sequence of integers in free format, separated by any combination of blanks, tabs, or newlines.

1.4.3 ZPR: Print a matrix or permutation

USAGE

`zpr [-G] [-s] [-r` *Range*`] [`*Datafile* `[`*Textfile*`]]`

DESCRIPTION

This program reads a matrix or permutation(s) from *Datafile* (G1 by default), and outputs the object(s) therein to *Textfile* in a format suitable for reading by a human, or by the ZCV program. If *Datafile* but no *Textfile* is specified, output is written to the standard output by default. However, if ZPR is invoked without any arguments input is read from G1 and output is written to T1. This is for compatibility with the FORTRAN MeatAxe.

There are several options to change the output format:

- `-G` GAP output.
- `-s` Print only a summary (file header fields).
- `-r` Select an output range i.e., a range of rows within a matrix or a range of permutations if the input file is a list of permutations. By default ZPR prints all rows of a matrix or all permutations in a file, respectively. *Range* may be a single number or an interval in the form *First*–*Last* as in the following examples:

Option	Meaning for	
	Matrices	Permutations
`-r5`	Row 5	5th Permutation
`-r5-10`	Rows 5–10	5th – 10th Permutation

If neither `-c` nor `-G` is specified, ZPR uses the standard output format described below.

The program starts by inspecting the input file to see what sort of thing it is, and a suitable output format is chosen as described in the table below. The next step is to write out a 'header' record. This has 4 integers in it. The first integer (2 digits) describes which format is in use. The other 3 (6 digits each) describe the object. For a matrix, they are field size, number of rows, number of columns. For a permutation they are '1', number of points, and number of permutations.

Description of object	mode	FL	NOR	NOC
Matrix over $GF(q), q < 10$	01	field	no. of rows	no. of columns
Matrix, $10<q<100$	03	field	no. of rows	no. columns
Matrix, $100<q< 10^6$	04	field	no. of rows	no. of columns
Permutation	12	1	no. points	no. of permutations

This format exactly matches the input format of the ZCV program. Hence these programs can be used freely to convert between internal and readable formats, and it is recommended that archive material be kept in the readable format, since this is more likely to be readable in a few years time.

Another use of this pair of programs is, with a little use of an editor, to convert between various types of objects. For example, to convert a permutation to a matrix, use ZPR and then edit the type from 12 to 2. You may also convert between the 'small' and 'big' arithmetic versions. In order to do this, you need two versions of both ZPR and ZCV, one compiled with the big and one with the small ZZZ module. Let these programs be called **`bigzpr`**, **`smallzpr`**, **`bigzcv`** and **`smallzcv`**. Then, conversion from small to big version is done with

```
smallzpr matrix.small | bigzcv - matrix.big
```

and from big to small version with

```
bigzpr matrix.big | smallzcv - matrix.small
```

SEE ALSO

ZCV

1.4.4 ZCT: Cut

USAGE

`zct` *Rows*`[:`*Columns*`] [`*Input* `[`*Result*`]]`

DESCRIPTION

This program cuts a piece, specified by *Rows* and *Columns*, out of the file *Input*, and writes the piece to *Result*. *Input* may be a matrix or a set of permutations. If no file names are specified, input is read from G1 and the result goes out to P2. If only *Result* is omitted, the output file name is formed by appending "1" to *Input*.

Both *Rows* and *Columns* are lists of positive integers or ranges (e.g., '13-25') separated by commas. If *Input* is a matrix, the corresponding rows and columns are cut, and the resulting rectangular pieces are combined to one rectangular matrix which is written to *Result*. If the *Columns* list is omitted, all columns of the selected rows are cut.

Here are some examples. Let *Input* be the 5 by 10 matrix

```
1 2 3 4 5 6 7 8 9 0
0 1 2 3 4 5 6 7 8 9
0 0 1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 0 0 0
9 8 7 6 5 4 3 2 1 0
```

Then, ZCT would produce the output shown below for several *Rows*:*Columns* lists:

Rows:*Columns*	*Result*

`1,4-5`	`1 2 3 4 5 6 7 8 9 0`
	`0 0 0 0 0 0 0 0 0 0`
	`9 8 7 6 5 4 3 2 1 0`

`1-3:8-10`	`8 9 0`
	`7 8 9`
	`6 7 8`

`1-2:2,5-7,9`	`2 5 6 7 9` `1 4 5 6 8`

`1-2,5:1-3,9-10`	`1 2 3 9 0`
	`0 1 2 8 9`
	`9 8 7 1 0`

The rows and columns which select the piece need not occur in ascending order, but the output depends on the ordering. For example, `zct 1,2` and `zct 2,1` both extract the first two rows of a matrix, but the second form will also permute the rows. Another example:

```
zct 3-4,1-2:3-4,1-2 inp out
```

would perform the following operation on a 4 by 4 matrix:

```
  inp       out
1 1 2 2   4 4 3 3
1 1 2 2   4 4 3 3
3 3 4 4   2 2 1 1
3 3 4 4   2 2 1 1
```

With permutations the program works in the same way as with matrices. Each permutation is treated as a row. The *Columns* list must be empty in this case, because ZCT can cut only entire permutations.

LIMITS

The number of entries in the *Rows* and *Columns* list must not be greater than 10. One row (or permutation, respectively) of the input file and the whole result of the cut must fit into memory.

SEE ALSO

ZPT

1.4.5 ZPT: Paste

USAGE

zpt [- $\left\{\begin{matrix}\texttt{r}\\ \texttt{R}\end{matrix}\right\}$ *#Rows*] [- $\left\{\begin{matrix}\texttt{c}\\ \texttt{C}\end{matrix}\right\}$ *#Cols*] *Out* *Inp*...

DESCRIPTION

This program reads matrices or permutations from *Inp*, pastes them together, and writes the result to *Out*. The way in which the pieces are pasted together is controlled by two parameters, *#Rows* and *#Cols*. For example,

```
zpt -r 2 -c 3 x aa ab ac ba bb bc
```

would paste together 6 matrices (`aa`...`bc`) in two rows and three columns like this:

aa	ab	ac
ba	bb	bc

The file name '`-`' is treated specially: No file is read in, and the corresponding piece of the output matrix is left empty. This can be used to calculate the direct sum of two representations. For example,

```
zpt -r 2 -c 2 A+B A - - B
```

creates the following matrix in block diagonal form:

A+B=

A	0
0	B

If `-R` and `-C` are used instead of `-r` and `-c`, ZPT will take *Inp* as 'base name'. This means, the actual input file names are formed by appending one or two numbers to the base name. Thus, the following commands are equivalent:

```
zpt -r 2 -c 2 x x11 x12 x21 x22
zpt -r 2 -C 2 x x1 x2
zpt -R 2 -C 2 x x
```

However,

```
zpt -R 2 -c 2 x x1 x2
```

is different, because `x1` and `x2` are now interpreted as base names of columns rather than rows. The latter command would paste together the same matrices as before, but with `x12` and `x21` exchanged.

With permutations the program works in the same way, but *#Cols* must be equal to 1. That means, permutations can only be concatenated.

If only one of *#Rows* and *#Cols* is specified, the other parameter is assumed to be one. If both parameters are omitted, *#Cols* is assumed to be one and *#Rows* is determined by th number of *Inp* arguments. This is useful, for example, to concatenate several files with permutations without having to count the number of files:

```
zpt Out File1 xxs test re233.1 re233.2 z1 z2
```

or for vectors:

```
zpt List Vector*
```

Together with ZMU, ZPT can be used to perform a piecewise multiplication of two matrices (see section 1.5.10).

LIMITS

Permutations are read and written one by one, so only one permutation must fit into memory. In the case of matrices, the memory requirement depends on the sizes of the pieces.

BUGS

Using `-R` and `-C` simultaneously with arguments greater than 10 will cause trouble because, for example, a file name like `file111` is ambiguous.

SEE ALSO

ZCT, ZMU

1.5 Main Programs

1.5.1 ZAD: Add two matrices

USAGE

zad *Mat1 Mat2 Result*

DESCRIPTION

This program reads two matrices from *Mat1* and *Mat2* and outputs their sum to *Result*. Both input files must be matrices over the same field with equal number of rows and columns. The matrices need not be square.

IMPLEMENTATION

The program reads one row at a time of each matrix, adds them together, and writes out the result.

1.5.2 ZCL: 'Clean' a matrix by a space

USAGE

zcl *Space Matrix CleanedMat RowOps*

DESCRIPTION

This program reads in two matrices from *Space* and *Matrix*, and 'cleans' *Matrix* with *Space*. It puts out two matrices to *CleanedMat* and *RowOps* such that

$$Matrix = RowOps \cdot Space + CleanedMat$$

and *CleanedMat* has as many zero columns as possible. The two input matrices must be over the same field and have the same number of columns. Also, *Space* must be in echelon form.

One use of this program is to calculate the action of a generator on an invariant subspace: Take the subspace in echelon form (as it is on output from ZSP), multiply it by a generator. Feed the original echelon form into *Space* and the result of the multiplication into *Matrix*. *CleanedMat* should be zero, and *RowOps* is the action of the generator on the invariant subspace. The action on the quotient of a given subspace can be calculated with the ZQT program (see section 1.5.14).

IMPLEMENTATION

The first matrix, *Space* is hold completely in memory. The second matrix, *Matrix* is then processed a row at a time. Row operations are performed to clear out the pivot points of the input row using the rows of the first input matrix. A row describing what was done is written out to *RowOps*, and the remnant (clean) row is output to *CleanedMat*.

LIMITS

The entire first matrix and one row of the second matrix must fit into memory.

The program does not perform a full check that *Space* is in echelon form.

1.5.3 ZCP: Characteristic and minimal polynomial of a matrix

USAGE

`zcp [-Gfm]` *Matrix*

DESCRIPTION

This program reads in a matrix, and calculates its characteristic or minimal polynomial. With no options, the characteristic polynomial is computed in a partially factored form (see below). The output can be changed using the following options:

- `-f` Complete factorization.
- `-m` Calculate the minimal polynomial.
- `-G` Output in GAP format.

Without `-G`, the output is in text format. Each line contains one factor (as defined below) of the characteristic or minimal polynomial. The `-G` option may be used to generate output which is readable by the GAP computer program. The output, then, is a sequence of sequences of finite field elements, representing the coefficients of the factors in ascending order.

IMPLEMENTATION

The characteristic polynomial of a matrix A is computed by constructing a sequence

$$0 = U_0 < U_1 < \ldots < U_n = V$$

of A-invariant subspaces U_i, where U_i/U_{i-1} is A-cyclic. In the i-th step, U_i is constructed by chosing a random vector $u \in V \setminus U_{i-1}$ and calculating $u, uA, uA^2, \ldots$ until some linear combination of these vectors falls into U_{i-1}. This yields a polynomial $p_i(x)$ with $up_i(A) \in U_{i-1}$. $p_i(x)$ is the characteristic polynomial of A on U_i/U_{i-1}, and the full characteristic polynomial of A is the product of all p_i's.

The algorithm for the minimal polynomial uses the same technique of constructing a sequence (U_i) of A-invariant subspaces. In the i-th step, images $uA, uA^2, \ldots$ of a seed vector u are calculated, until a linear combination of these vectors vanishes (this is the main difference to the algorithm above). This yields a polynomial $p_i(x)$ of minimal degree with $up_i(A) = 0$, and the minimal polynomial of A is the least common multiple of the p_i's.

LIMITS

There must be enough memory to hold three (four for the minimal polynomial) matrices of the same size as the input matix.

1.5.4 ZEF: Reduce to (semi) echelon form

Usage

`zef [-G]` *Matrix Result*

Description

This program reads in a matrix, performs a Gaussian elimination to put the matrix into semi-echelon form, and writes out the result.

A matrix is in semi-echelon form, if the first non-zero entry in each row entry is a 1, and all entries exactly below that 1 are zero. Here is an example:

```
00011231231234241212З
01201231231223123123З
000000001345121223233
102012330332333312212
001021230333323123123
000001230311212112121
```

A matrix is in (full) echelon form if the rows are further permuted so that the first non-zero entry in each row is later than the first non-zero entry in all previous rows. There is no real need to do this row permutation in the MeatAxe system, and the permutation is not done by this program.

At the end, the matrix may have fewer rows than it started with, since the rows may have been linearly dependent. The rows of the output matrix are always linearly independent and span the same space as the rows of the input matrix. A message

```
RANK nnn
```

is printed at the end of the run to notify the operator of the size of the output matrix. This program may be used to find the rank of matrix, being faster than the null-space program. There is no need for the input matrix to be square, and the output may also not be square.

Limits

The entire input matrix and the pivot table (4 bytes for each row) must fit into memory.

1.5.5 ZEV: Eigenvalues

Usage

`zev [-G]` *Matrix* `[`*Poly* `[`*Group*`]]`

Description

This program reads a matrix from *Matrix* and a list of polynomials from *Poly*. For each input polynomial, it evaluates that function of the input matrix, calculates the nullity, and puts out this nullity, divided by the degree, along with a text from the input. By default, polynomials are read from the standard input. The `-G` option produces output in GAP format.

The program was specifically designed to assist in the calculation of the Brauer characters of diagonalizable matrices, with the text giving the complex number which is the Brauer character of the companion matrix for that polynomial. Usually the polynomials have been prepared in a separate data file and are fed into ZEV by giving the file name or by redirecting its input. The preparation of the input polynomials is generally a time-consuming task if it is done by hand, but there are data files available for the most commonly used fields. These files should be located in the library directory (see section 1.2.4). They are distributed with this release of the C MeatAxe. If the user is familiar with the computer program system GAP, he will find it easy to create his own data files.

If the nullity is not a multiple of the degree, ZEV prints a warning message.

Polynomial file format The data file contains the polynomials in text form. Several polynomials can be comprised in a 'group', and the data file can contain any number of groups of polynomials. This allows several sets of polynomials to be kept in one data file (for example, all polynomials for a given field), the appropriate polynomials being selected trough the *Group* argument on the command line.

The file is read and interpreted line by line. There are three types of lines:

1. Comment lines, beginning with a '`#`'. These lines are simply ignored by ZEV, as are empty lines.

2. Group headers. Each line beginning with a non-space character is interpreted as the beginning of a new group of polynomials. Such lines contain only one text field, the name of the group (up to 1023 characters).

3. Lines beginning with a space are interpreted as polynomials. The format is

   `[space]`*Name Coefficients*

   where *Name* is any text (up to 1023 characters), and *Coefficients* are the coefficients of the polynomial (in free format). Note that the first character must be an ordinary space charcter, a TAB is not allowed! The coefficients must use the names as specified by the arithmetic — usually $0, 1, \ldots q-1$. The one exception is -1, which the program treats specially as '0'-'1' so that the cyclotomic polynomials can be used over *all* fields. The coefficients are in decreasing degree, starting with the coefficient of the highest power of x and continuing, ending up with the constant term.

Here is an example:

```
# Sample input file for ZEV
# Some polynomials over GF(5)
#
p11b11
  1         1 4
  b11       1 4 4 1 3 4
  -1-b11    1 2 4 1 1 4
p13c13
  1         1 4
  c13       1 3 0 3 1
  c13*3     1 1 4 1 1
  c13*9     1 2 1 2 1
```

This file contains 7 polynomials in two groups. The polynomial 'b11' in group 'p11b11' is $f(x) = x^5 + 4x^4 + 4x^3 + x^2 + 3x + 4$.

Output format There are two output formats. By default the nullities are printed in tabular form giving group, name, degree and multiplicity (i.e., nullity divided by degree) for each polynomial. If the `-G` option is given, ZEV prints an algebraic expression which can be read from GAP. Here is an example with an 8 by 8 matrix over GF(3), polynomials being read from `poly3` (`$` is the shell prompt):

```
$ zev mat poly3 p8i2
      p8i2                  1    1    1
      p8i2                 -1    1    4
      p8i2                  0    2    2
      p8i2                 i2    2    0
      p8i2                -i2    2    1
$ zev -G mat poly3 p8i2
MeatAxe.BrauerChar := 1*(1) + 4*(-1) + 2*(0) + 1*(-i2);
```

Note that `i2` does not appear in the expression because its coefficient is zero.

LIMITS

There must be enough memory to hold the input matrix and two more matrices of the same size. Lines in the polynomial input file must not be longer than 1023 characters. Group and polynomial names must be shorter than 1023 characters.

BUGS

It is not checked that the input file is a matrix. TAB characters at the beginning of a line are not interpreted as space.

1.5.6 ZFR: Frobenius automorphism

USAGE

`zfr` *Matrix Result*

DESCRIPTION

This program reads a matrix, applies the Frobenius automorphism $x \mapsto x^p$, where p is the characteristic of the field, to each entry and writes out the result. Default file names are G1 for input and P2 for output.

The program first calculates the characteristic by repeatedly adding 1 to itself until zero is reached. A message of the form

`CHARACTERISTIC IS` p

is printed to inform the user which p was used. Then the matrix is read row by row, the Frobenius automorphism is applied to each entry, and the resulting row is written out.

LIMITS

One row of the matrix must fit into memory.

1.5.7 ZIV: Invert a matrix or permutation

USAGE

`ziv` *Input Result*

DESCRIPTION

This program reads a matrix or a permutation from *Input* and writes its inverse out to *Result*.

IMPLEMENTATION

First the input file is inspected. If it is a matrix, it is read in and an identity matrix is made in memory. Inversion proceeds by row operations as follows: The original matrix is reduced to the identity matrix, and the same operations are done to the other matrix that started as the identity matrix. At the end of this process, the second matrix is the inverse of the original matrix, so is written out to *Result*.

If the input file contains a permutation it is read in and its inverse is calculated point by point. I.e., if the input permutation maps i to k, the inverse permutation is set up to map k to i. The inverse permutation is then written out. If the input file contains more than one permutation, only the first one is inverted and written out. A message saying

`ZIV WARNING - ONLY 1 OF` n `PERMUTATIONS WILL BE INVERTED`

is printed in this case.

LIMITS

There must be enough memory to hold two matrices or two permutations, respectively.

1.5.8 ZKD: Condense a permutation

USAGE

`zkd` *q OrbSz Perm* ***Result***

DESCRIPTION

This program reads an orbit sizes file (*OrbSz*) and a permutation from *Perm*. It outputs the condensed form, i.e., a matrix over GF(()q) to *Result*. The field must be specified on the command line because the other input data is is all to do with permutations and the program would otherwise not know which field was intended. The orbit sizes are read from *OrbSz*. This file is an integer matrix with one row and n columns, where n is the number of orbits. Normally this file would be produced by the ZMO (make orbits) program.

The second input file, *Perm*, must be one, or more, permutations. Notice that only the first permutation is read in and condensed. If there are more than one permutation, the others are ignored. It is assumed, that the orbits are contiguous, i.e., if $l_1, l_2, \ldots, l_r$ are the orbit sizes, then the orbits are given by

$$\begin{aligned} O_i &= \{s_i, s_i+1, \ldots, s_i+l_i-1\} \qquad (1 \le i \le r) \\ \text{where } s_i &= 1 + \sum_{k=1}^{i-1} l_k \end{aligned}$$

This is normally achieved by conjugating the original group generators by the permutation that comes out of ZMO (see the ZMO description).

IMPLEMENTATION

The first step is to calculate the largest power (m) of the characteristic that divides any of the orbit sizes. ZKD assumes that this is the order of the Sylow-p subgroup of the condensation subgroup, but it prints out its findings with the message

`p-part taken has order` n

so the user can check it. If this is not the order of the Sylow-p subgroup of the condensation group, the program will not 'know', so will continue. Normally, however, the condensation subgroup K will have trivial Sylow-p subgroup, or at any rate the Sylow subgroup will have a regular orbit, and in this case at least the condensation is legitimate.

The output is a square matrix with one row and one column for each orbit of K. Abstractly, the condensation can be described as follows. Let G be a permutation group of degree n, F a field of characteristic p and $K \le G$ a p'-subgroup. Then, there is an idempotent

$$e = \frac{1}{|K|} \sum_{h \in K} h \in FG$$

associated to K. Now, let V be a FG-module, for example (as in this program) the natural permutation module $V = F^n$, where G acts by permuting the entries of vectors. Then, Ve is an $e(FG)e$-module, and for any $\pi \in G$, the condensed form is $e\pi e$, regarded as a linear map on Ve.

To calculate the action of $e\pi e$, let $(v_1, \ldots, v_n)$ be the standard basis such that $v_i\pi = v_{i\pi}$ for $\pi \in G$. As explained above, the K-orbits are assumed to be contiguous. Thus, a basis of Ve is given by the orbit sums

$$w_i = v_{s_i} + \ldots v_{s_i+l_i-1} \qquad (1 \le i \le r)$$

and with respect to this basis we have

$$w_i(e\pi e) = \sum_{k\in O_i} \frac{1}{l_{[k\pi]}} w_{[k\pi]}$$

where $[m]$ denotes the orbit to which point m belongs (i.e., $s_{[m]} \leq m < s_{[m]} + l_{[m]}$).

If K is not a p'-subgroup, e is no longer defined. However, the last formula can still be given a sense by replacing

$$\frac{1}{l_{[i\pi]}} \to \lambda_{[i\pi]} := \begin{cases} \frac{1}{l_{[i\pi]}/p^m} & p^m | l_{[i\pi]} \\ 0 & \text{else} \end{cases}$$

where m is the highest power of the characteristic which divides any of the orbit sizes. Thus, all but the orbits with maximal p-part are discarded, and the corresponding columns in the output matrix are zero.

1.5.9 ZMO: Make orbits under permutations

USAGE

`zmo [-s` *Seed*`]` *Perm1 Perm2 Orbits Sizes*

`zmo [-g` *#Perms*`] [-s` *Seed*`]` *Perms Orbits Sizes*

DESCRIPTION

This program reads two permutations from *Perm1* and *Perm2*, and finds the orbits under the permutations. The points are written to *Orbits* in an order such that the points in one orbit occur together. The orbit sizes are written to *Sizes*.

The —-g— may be used if the number of permutations is not two. In this case, *Perm* is treated as a base name, and one permutation is read from the files *Perms*`.1`, *Perms*`.2`, ... For example,

```
zmo -g 3 p orb orbs
```

reads three permutations from `p.1`, `p.3`, `p.3`. If any of the input files contains more than one permutation, only the first one is read.

IMPLEMENTATION

The orbits are calculated by a recursive technique. The first permutation is used to remember which points have already been output, by making the point number negative. A stack is kept of those points that have been found in the orbit but have not yet been multiplied by the generators. The outer loop consists of looking through the points and 'doing' that point if it is not already marked as done. The point is put onto the stack, and operation then continues by taking a point off the stack, and for each generator in turn, if its image is new, putting it on the stack and marking it as done. Points are output as they are multiplied by the generators.

By default, ZMO starts with point 1. A different starting point may be selected with the `-s` option.

The program produces 2 files of output. *Orbits* is a permutation-format list of all the points in the first orbit, followed by all the points in the second etc. This permutation can be used to conjugate permutations such that the orbits are contiguous. $X \cdot G \cdot X^{-1}$ is the relevant calculation.

The second file, *Sizes*, is an integer matrix with one row containing the orbit sizes. This file can be used as input to ZKD or ZUK. Since it is not known until the end how many orbits there will be, and this is needed at the head of the sizes file, the orbit sizes are held in memory.

Finally, a message is printed to tell the user to describe what happened. If there are at most 15 distinct orbit sizes, the program prints a list of

```
nnn Orbit(s) of size nnn
```

messages. If there are more than 15, then a message

```
nnn DISTINCT ORBIT SIZES
```

is produced. If there are more than 200 orbit sizes, the message will say

```
MORE THAN 200 DISTINCT ORBIT SIZES
```

LIMITS

The number of permutation must be less than 50. All permutations must fit into memory at the same time. The number of orbits must be less than or equal to 20000 (2000 for the PC version). Also the stack size is limited to 50000 positions (10000 for the PC version).

1.5.10 ZMU: Multiply

USAGE

`zmu [-r` *Row*`[.`*#Pieces*`]] [-c` *Col*`[.`*#Pieces*`]] [`*File1 File2 Result*`]`

DESCRIPTION

This program reads in two matrices or permutations from *File1* and *File2* and outputs their product (*File1***File2*) to *Result*. Default file names are G1, G2 and P3.

The input files must contain two 'compatible' objects, i.e., their product must be defined. Currently, ZMU can handle the following data types:

1. Both files are matrices over the same field, and the number of columns of *File1* equals the number of rows of *File2*. They are multiplied using the ordinary matrix product: Each row of the first matrix is multiplied by the second matrix giving one row of the result. This is written out before the next row is multiplied, so only one row of the product is in memory at a time.

2. A one by one matrix is treated as a single field element. It can be multiplied from both sides with any matrix over the same field.

3. Both input files are permutations of degree n_1 and n_2, respectively. The result, in this case, is a permutation of degree n :=max(n_1,n_2). If $n_1 \neq n_2$, the smaller permutation is extended to degree n by adding fixed points. Then, the usual product is calculated by mapping each point first by the left then by the right permutation and storing the image in the resulting permutation.

4. *File1* is a matrix, *File2* is a permutation and the degree of the permutation equals the number of columns of the matrix. The result is a matrix of the same size which is calculated from the input matrix by permuting the marks of each row in the following way: The i^{th} mark of the row is stored as the k^{th} mark of the result if point i maps to point k under the permutation. This can be written as
$$(v_1, \ldots, v_n) \cdot \pi = (v_{1\pi^{-1}}, \ldots, v_{n\pi^{-1}})$$
where $v_1, \ldots, v_n$ are the marks in one row and π is the permutation.

5. *File1* is a permutation of degree m, and *File2* is a m by n matrix. The result is again a m by n matrix which consists of the rows of the input matrix, rearranged according to the permutation. If the permutation maps i to k, then the k^{th} row of the input matrix becomes the i^{th} row of the output matrix. Here is an example:
$$(123) \cdot \begin{pmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{pmatrix} = \begin{pmatrix} 2 & 2 \\ 3 & 3 \\ 1 & 1 \end{pmatrix}$$

With these conventions, products between matrices and permutations are defined in a consistent way. The associative law $a(bc) = (ab)c$ holds whenever ab and bc are defined ($a, b, c =$ matrix or permutation). A permutation matrix created with ZCV or ZCF, if multiplied with another matrix, produces the same result as the original permutation.

In the case of two matrices, a piecewise multiplication can be performed using the `-r` and `-c` options. If one or both of these options are specified on the command line, ZMU will read only some rows of *File1* and/or some columns of *File2*. Multiplying the two pieces together yields a rectangular piece of the result. By default the result is divided into 4 pieces of (almost) equal size. To calculate the 4 pieces successively, type

```
zmu -r 1 -c 1 m1 m2 tmp11
zmu -r 1 -c 2 m1 m2 tmp12
zmu -r 2 -c 1 m1 m2 tmp21
zmu -r 2 -c 2 m1 m2 tmp22
```

The resulting matrices `tmpXX` can then be pasted together using ZPT:

```
zpt -R 2 -C 2 result tmp
```

This procedure can be used in a multi-processor environment where each piece of the result is computed on a separate machine.

By adding an additional parameter to `-r` and/or `-c` you can control the number of vertical or horizontal pieces. For example,

```
zmu -r 3.5
```

means: Cut *File1* horizontally into five pieces and use the third piece for multiplication. *#Pieces* must not be greater than the number of rows.

Bugs

If any of the input files contains more than one permutation, only the first one is read in and multiplied.

See Also

ZPT

1.5.11 ZNU: Matrix null-space

Usage

`znu` *Matrix* [*Nullspace*]

Description

This program reads in a matrix from *Matrix* and outputs a basis for its null-space in semi-echelon form to *Nullspace*. If the *Nullspace* argument is omitted the null-space is not written out, but its dimension is still displayed.

Notice that the input matrix does not need to be square. The matrix is read in, and the program generates the $n \times n$ identity matrix in memory where n is the number of rows. It then proceeds to do row operations to the input matrix, and the same operations to the other one. The input matrix is reduced to semi-echelon form, and whenever a zero vector is made, the corresponding row of the other matrix is marked for output.

When the echelon form process has been completed, the dimension of the null-space is reported as

```
NULLITY nnn
```

Then, the basis for the null-space is reduced to semi-echelon form and written out.

LIMITS

There must be enough memory for the input matrix and a square matrix with the same number of rows.

1.5.12 ZOR: Order of permutations or a matrix

USAGE

`zor [-q] [-m` *MaxOrder*`]` *Input*

DESCRIPTION

This program reads a file, containing either permutations, or a square matrix, and calculates the order(s).

IMPLEMENTATION

In any case the whole input file is read into memory. If it is a matrix, the order is found by calculating the orders on cyclic subspaces and taking the least common multiple. The algorithm works as follows:

1. Let $A \in F^{n\times n}$ be the given matrix and $V = F^n$ the space where A acts. Set $W := \{0\}$ (the trivial subspace) and $o := 1$.
2. Chose a vector $v \in V \setminus W$. Calculate the cyclic subspace $\langle v\rangle_A$ generated by v and the order o_v of A acting on $\langle v\rangle_A$. This is done by repeatedly multiplying the vector v with A until it returns to where it started.
3. Set $o := \mathrm{lcm}(o, o_v)$.
4. Set $W := W + \langle v\rangle_A$. If $W < V$, continue with step 2. Otherwise, o is the order of A.

Gaussian elimination is used to maintain a basis of W in semi-echelon form. At the end, the result is printed in the format

`ORDER IS` *o*

In order to avoid infinite loops, there is a limit o_v. If the vector does not return after 1000 multiplications the order is assumed to be infinite and the program stops with an error message. This happens also if the value of o exceeds 100000.

There are two options which can reduce the run-time of the program. With the `-m` option the user may specify a maximal order. If in the process described above o reaches this limit, ZOR will assume that o is the order. If o becomes greater than *MaxOrder*, the program stops with an error message. The second option, `-q` ('quick'), makes ZOR stop if the dimension of W reaches 1/10 of the dimension of the whole space. In this case, o may be less than the order of the matrix, so the result is printed as

`ORDER IS A MULTIPLE OF` *o*

If the input file contains permutations, each one is read in and its order is calculated as the least common multiple of the orbit sizes. The result is printed in the format

```
ELEMENT nn HAS ORDER nnn
```

The `-q` and `-m` options have no effect for permutations.

LIMITS

The whole matrix plus a second matrix of the same size must fit into memory. In the case of permutations, there must be enough memory for one permutation.

1.5.13 ZPC: Permutation chop

USAGE

zpc [-b] [*Gen1 Gen2 Seed* [*Sub1 Sub2* [*Quot1 Quot2*]]]

zpc [-b] -g *#Perm*[.*#Gen*] *Gen Seed Sub Quot*

DESCRIPTION

If invoked without any options, this program reads two permutations from *Gen1* and *Gen2* and a point from *Seed*. The orbit containing that point is made, and the action on the orbit is written to *Sub1* and *Sub2*. If any points remain, the action on these is written to *Quot1* and *Quot2*. Default file names are G1 and G2 (permutations), G3 (seed), P4 and P5 (action on the orbit), and P6 and P7 (action on the remaining points).

With the `-b` option, *Seed* is taken as a list of points defining one block of a block system for the permutation group, and the program computes the action of both the generators on the block system. Nothing is written to *Quot1* and *Quot2* in this case, even if the group generated by the two permutations does not act transitively on the block system.

A number of permutations different from 2 can be specified by using the `-g` option. *#Perm* is the number of permutations to chop, and *#Gen* specifies how many of these are sufficient to generate the whole permutation group. For example `-g 5.2` means 'chop 5 permutations, but assume that the first two generate the whole group'. *#Gen* may be omitted, the default being *#Perm*. Thus, instead of `-g 5.5` you may simply type `-g 5`. When the `-g` option is used there must be exactly 4 Arguments, 3 of which (*Gen*, *Sub* and *Quot*) are interpreted as 'base names'. For example,

```
zpc -g 3 z seed s q
```

would read three permutations from `z1`, `z2` and `z3`, and a point from `seed`. The actions on the orbit would be written to `s1`, `s2` and `s3`, and the actions on the remaining points to `q1`, `q2` and `q3`.

The program checks the input files to see if they contain reasonable data. Failure of any of these checks produces an error message and the program stops. If all goes well, the first point of the permutation in *Seed* is taken to be the first point in the orbit. The orbit is then calculated by applying the generators to all new points. The points of the orbit are renumbered, and, for each of the permutations, the action on the orbit is written out. If this orbit is all the points, the message

TRANSITIVE ON *n* POINTS

appears, and the program stops. If any points remain, these are also renumbered, and the action on the remaining points is written out, too (unless `-b` was used). In this case

INTRANSITIVE - 'SUB' *n* 'QUOT' *m*

is written to stdout, the 'SUB' being the orbit of the given point.

With the `-b` option, the orbit is calculated in the same way, but a whole block, i.e., a set of points, is read from *Seed* and taken as the starting 'point'. The points are then renumbered in such a way that the orbit containing the seed is $\{1, 2, \ldots, N\}$, where N is a multiple of the block size, and the blocks are contiguous. After this rearrangement, point number n belongs to block $\lfloor (n-1)/b \rfloor + 1$, where b is the block size. The action of the permutations on the blocks can then be calculated by simply dividing the point numbers by the block size. Of course, the block size must be a divisor of the permutation's degree. Otherwise the program stops with

```
ZPC ERROR - BLOCK SIZE DOES NOT DIVIDE DEGREE
```

It might turn out during the computation, that two points in one block are mapped to images in different blocks. This means that *Seed* was not a block of any block system for the given representation and the program will stop with the error message

```
ZPC ERROR - NOT A BLOCK SYSTEM
```

If all goes well, the user is informed with one of the messages

TRANSITIVE ON *n* BLOCKS
INTRANSITIVE - 'SUB' *n* 'QUOT' *m*

and the result is written to *Sub1* and *Sub2*. *Quot1* and *Quot2* are never written when `-b` is specified, even if the action on blocks is intransitive.

The use if this program is twofold. Firstly it can be used to obtain transitive constituents from intransitive permutation representations. Secondly, if the starting point is in some sense canonical, the resulting renumbered representation will be according to a 'standard' numbering of the points. It should be noticed that this comment only applies to the representation on the 'SUB' — that written to *Sub1* and *Sub2*. If this standard-base-like property of the output is not required, there is no particular reason to choose *Seed*, and point 1, or indeed one of the permutations can be used.

LIMITS

The number of permutations, specified with `-g`, must not be greater than 10. All permutations are held in memory at the same time.

1.5.14 ZQT: Clean and quotient

USAGE

zqt [-i] [*Subsp Matrix Quot*]

DESCRIPTION

This program reads in a subspace from *Subsp* and applies the canonical map to its quotient on the matrix *Matrix*, the result being written out to *Quot*. Default file names are G1 (subspace), G2 (matrix), and P3 (output).

Subsp should be a matrix in semi-echelon form, and the two input matrices must have the same field parameter and the same number of columns. If this is not the case the program stops with an error message. Otherwise the program reads in *Subsp*, builds a table of pivot columns and then proceeds, row by row, through *Matrix*. For each row, the significant entries are zeroized by adding the correct multiple of rows of *Subsp*. The insignificant columns are then extracted and written out to *Quot*. Hence

Subsp has A rows, B columns and is in echelon form,

Matrix has C rows, B columns and is otherwise arbitrary, and

Quot has C rows and B-A columns.

In other words: the program projects *Matrix* onto the B-A dimensional quotient space defined by *Subsp*.

If the `-i` option is used, ZQT calculates the action of *Matrix* on the quotient by *Subsp*. This is done by projecting the matrix as explained above, and taking only the 'insignificant' rows. Insignificant rows are defined by treating the pivot table as a table of *rows* rather than columns.

Example: Let `spc` be an invariant subspace and `z1` an algebra element (a square matrix). Then, after

```
zqt -i spc z1 q1
```

q1 contains the action of z1 on the quotient by spc.

Another, less obvious use of ZQT is to condense a matrix representation. First, find an element E of the group algebra with stable rank, i.e. rank(E*E) = rank(E). This can be found by taking any element F of the group algebra and raising it to higher powers until the rank stabilizes. We may then condense onto the kernel of E as follows

```
zef E X        X is the echelon form of Image(E)
znu E Y        Y is the kernel of E
zqt X Y Z      calculate the canonical projection of Y ...
ziv Z T        ... and adjust Y so that the canonical ...
zmu T Y Y1     ... projection of Y1 is the identity
zmu Y1 Z1 T1   calculate KZ1 = condensed Z1
zqt X T1 KZ1
zmu Y1 Z2 T1   calculate KZ2 = condensed Z2
zqt X T1 KZ2
```

Bugs

It is not completely checked that *Subspace* is in echelon form.

Limits

The first matrix (*Subspace*) and one row of *Matrix* must fit into memory.

1.5.15 ZSB: Standard basis

Usage

zsb *Matrix1 ...MatrixN Seed SBasis* [*Op*]

zsb [-g#*Gen*] *Matrix Seed SBasis* [*Op*]

Description

This program reads in two or more matrices from (*Matrix1 ... MatrixN*) and one or more vectors from *Seed*, and outputs a square non-singular matrix to *SBasis* whose production from the input is basis independent. Hence the program produces the 'standard basis'. In fact all the rows of the output matrix are images of the input vector, and each row consists of the first image of the vector that is independent of all the preceding vectors.

If there are more than two generators, the second form (with -g) must be used. In this case, *Matrix* is the base name and actual file names are built by appending '.1', '.2' and so on.

The last argument, *Op*, is optional. If it is present, it is taken as a file name and a record of all operations is written to this file. The format is an integer matrix with two columns and one row for each vector in *SBasis*. The first column is the source vector number. The second column specifies by which generator the source vector was multiplied.

The vector is taken as the top row of the matrix in *Seed*. The program starts with some checks — the input matrices must be square, and the same size and field as each other and the seed vectors. If these checks are passed, two bases for the same space are set up to contain the first seed vector. One is the (growing) output matrix, and the other is the same space, but in near-echelon form. The program proceeds by multiplying the vectors of the output matrix in turn by the generators. The resulting vector is tested for linear independence with the preceding vectors using the echelon form matrix. If the new vector is dependent, it is merely discarded and the next new vector is made. If it is independent, it is appended to the output matrix, and the echelon form matrix is also updated (by the remnants of the test for linear independence).

If the first seed vector spans a proper subspace (or if it is zero) the program reads from *Seed* until it find a vector which is not contained already in the subspace found so far. Then it repeats the procedure outlined above, appending the resulting vectors to the output matrix. For each seed vector a message is printed which tells the user the dimension of the invariant subspace:

`VECTOR` *nnnn* `SPANS` *dddd*

The program stops if either the maximal dimension is reached or if there are no more seed vectors. In the latter case the output is not a basis and a message saying

`ZSB WARNING: SPAN IS ONLY` *nnnn* `OF SPACE` *dddd*

is written to stderr. In case of success, the program checks for unused seed vectors in the input file. If there are vectors left, the message

`ZSB WARNING: ONLY` n `OF` m `SEED VECTORS WERE USED`

is printed. In any case the output matrix is written to *SBasis*.

This program is primarily designed to put a representation into a standard basis, so that equivalent representations are identical. This can be achieved by finding a matrix in the group algebra of nullity 1, and using the null vector and the two generators as input to this program. Calling the output matrix X, form X * G1 * inverse(X) to bring the generators into standard basis. If an equivalent representation is then found, and the same group algebra elements null vector is used in the same way, a proof of equivalence has been found, since both representations have been conjugated so they look identical.

Another use of this program is to find the bilinear form (if any) fixed by a representation. This is done by running ZSB twice, once on a representation and once on the transpose-inverse representation, calculating the matrix that conjugates the first to the second.

LIMITS

This program currently works only on matrices, although there are similar jobs that need to be done with permutation representations.

SEE ALSO

ZSC

1.5.16 ZSC: Spin up with script

USAGE

`zsb [-g`#*Gen*`]` *Gen* *Seed* *Op* `[`*Basis*`]`

DESCRIPTION

This program reads in two or more matrices from *Gen*`.1`, *Gen*`.2`, ..., a list of seed vectors from *Seed* and a 'script' (i.e., a list of operations) from *Op*. Then, ZSC applies the script to each seed vector and writes the output in a separate file (*Op*`.1`, *Op*`.1`, ...) for each vector.

Gen`.1`, ...,*Gen*.n must be square matrices over the same field. *Seed* must be a matrix over the same field with the same number of columns. *Op* must be an integer matrix with two columns.

The format of the *Op* file is described together with the ZSB program. The *Op* file must be produced with a cyclic seed vector, i.e., the first row must be $(1,0)$ and all other rows must be of the form (m,n) with $n \neq 0$.

The number of generators is 2 by default. This can be changed by using the `-g` option. The number of output files equals the number of seed vectors. If no *Out* argument is given, the output name defaults to *Seed*. For example,

```
zsc -g 3 gen seed op
```

reads threee generators fom `gen.1`, `gen.2` and `gen.3`, n seed vectors from `seed`, the script from `op`, and writes the output to `seed.0001`, `seed.0002`, ..., `seed.`n.

SEE ALSO

ZSB

1.5.17 ZSI: Sum and intersection

USAGE

```
zsi [-G] Space1 Space2 Sum Intersection
zsi -s [-G] Space1 Space2 Sum
```

DESCRIPTION

This program reads in two spaces from *Space1* and *Space2* and writes out their sum and intersection, in semi-echelon form, to *Sum* and *Intersection*, respectively. The second form, using the `-s` option, calculates only the sum.

The input files must be matrices over the same field and with the same number of columns. They need not be in (semi-)echelon form. Before the program writes out the result it prints one of the following messages:

```
SUM Dim1 INTERSECTION Dim2
SUM Dim1
```

LIMITS

In normal mode, there must be enough memory to hold two copies of each of the two spaces at the same time. If only the sum is calculated (using `-s`) the program assumes the worst case (n_0) for the dimension of the sum, and allocates an appropriate memory block. n_0 is given by the total number of rows in both input files or the number of columns, whatever is smaller.

1.5.18 ZSM: Small matrices package

USAGE

```
zsm [-g NGen] Command File(s)
```

DESCRIPTION

This program performs operations on matrices and — at least to some extent — on permutations. *File(s)* is a list of file names for input and/or output, depending on the operation. *Command* is any of the following

Command	Description
`pwr`	Power of a matrix or permutation
`ro`	Random orders of matrices or permutations
`fp`	Fingerprint (matrices only)
`mw`	Make word (matrices only)

Most commands take one or more additional arguments (see below). Available options are:

- `-g` Set number of generators (default is 2 generators). This option has no effect on the `pwr` command. See the description of the `ro` subcommand below for more information.
- `-G` Produces GAP output.

Power ZSM can raise a matrix or permutation to its n-th power. The syntax is

> `zsm pwr`*Num File Result*

where *Num* is a positive integer. *File* may be a square matrix or a permutation. The result is written to *Result*.

Random orders This command can be used to find the group generated by a given set of matrices or permutations. The syntax is

> `zsm ro[`*Num*`]` *Gen1 Gen2*
> `zsm -g`#*Gen* `ro[`*Num*`]` *Gen*

where *Num* is a positive integer. The first form reads two generators from *Gen1* and *Gen2* — both square matrices and permutations are allowed, but not mixed. If the `-g` option is used, *Gen* is treated as 'base name', and actual file names are built by appending '.1', '.2',... to the base name.

The program calculates *Num* (pseudo) random products of the generators and prints out their orders. If the number is omitted it defaults to 120.

Fingerprint The command

> `zsm fp` *Gen1 Gen2*
> `zsm -g` #*Gen* `fp` *Gen*

reads in two or more generators as explained above and calculates their fingerprint, i.e., the nullities of 6 'standard' elements in the algebra generated. Only matrices are allowed, not permutations.

More nullities can be calculated by specifying one or two additional arguments to the `fp` command as follows

> `zsm fp`*Num* `...`
> `zsm fp`*First*-*Last* `...`

With this command, the program calculates the first *Num* words (or words from *First* up to *Last*, respectively), and prints out their nullities. The numbering of algebra elements is described below.

Make Word The command

> `zsm mw`*Num Gen1 Gen2 Word* `[`*Nullsp*`]`
> `zsm -g` #*NGen* `mw`*Num Gen Word* `[`*Nullsp*`]`

reads two or more matrices and calculates the word number *Num* in the algebra generated by the matrices. The word is written to *Word* and, if an additional argument is specified, its null-space is written to *Nullsp*.

Enumeration of words is defined as follows. First, a fixed set $t_1, t_2, \ldots$ of monomials in the generators is calculated. The choice of these monomials depends only on the number n of generators. E.g., if there are two generators a and b, the first monomials are a, b, ab, b^2, ba, ab^2,... Any word generated by ZSM is a linear combination of the t_i's with coefficient 0 or 1. The number of a word is obtained by interpreting the sequence of coefficients as a number in base 2.

This scheme is the same as used in the lattice package (see section 1.6). Thus, the 'make word' function can be used, for example, to evaluate peak words (given by their number) on submodules or quotients.

To produce a multiple of the unit matrix, the syntax

> `zsm mw`*Num*`id` *Gen1 Gen2 Word*`]`

may be used. E.g., '`mw3id`' calculates $3 \cdot I$.

LIMITS

Contrary to the program name there is no explicit limit on the size of matrices or permutations.

SEE ALSO

CHOP, PWKOND

1.5.19 ZSP: Spin up and split

USAGE

$$\texttt{zsp [}\mathit{Options}\texttt{] [}\mathtt{\mathit{Gen1\ Gen2\ Seed}}\texttt{ [}\left\{\begin{matrix}\mathit{Sub1\ Sub2\ Quot1\ Quot2}\\ \mathit{Space}\end{matrix}\right\}\texttt{]]}$$

$$\texttt{zsp -g } \#\mathit{Mat}\texttt{[.}\#\mathit{Gen}\texttt{] [}\mathit{Options}\texttt{] [}\mathit{Gen\ Seed}\texttt{ [}\left\{\begin{matrix}\mathit{Sub\ Quot}\\ \mathit{Space}\end{matrix}\right\}\texttt{]]}$$

DESCRIPTION

This program reads in two or more matrices or permutations and a space (from *Seed*), and tries to find a subspace which is invariant under the action of the matrices. If a proper invariant subspace is found, the program can either write out this space or 'split' the representation, i.e., calculate the action of the matrices on both the subspace and its quotient.

In order to find an invariant subspace the program starts with one or more 'seed' vectors taken from *Seed* and calculates the smallest invariant subspace containing these vectors ('spin-up'). This is done by retaining always a semi-echelon form of the space found so far, and multiplying the rows of this by all the generators, appending the (Gaussian-eliminated) result to the space if it is not already in, and continuing until all vectors in the semi-echelon form have been treated. If at any stage the dimension reaches the dimension of the entire space or the limit specified via `-d`, the process is aborted, and the program continues with the next seed vector(s).

This program is rather complex, because it can perform several different tasks. What the program does exactly depends on the setting of command line options. Available options are:

`-g` Set number of matrices (if not 2)
`-s` Select the 'seed method' (see below)
`-n` Spin up and write out the invariant subspace, but do not split
`-d` Set a limit on the dimension of the invariant subspace
`-t` Quick (and unsecure) mode. See below for details.
`-o` No output to files, messages only
`-v` Verbose.

All but the `-o` and `-n` options take one or more arguments. They are described in more detail below.

Two or more generators There are two ways of invoking ZSP. Without the `-g` option, it is assumed that the group has two generators. Matrices and seed space are read from *Gen1*, *Gen2* and *Seed*, respectively, or from G1, G2, G3 by default. The output matrices, unless `-n` is used, are written to *Sub1*, *Sub2*, *Quot1*, *Quot2*. Default output file names are P4, P5, P6 and P7, respectively.

If there are more than two generators, the second form must be used. *#Mat* is the number of matrices, an integer greater or equal to 2. In this case all arguments except *Seed* are understood as 'base names', i.e., ZSP will append a number to the argument — '1' for the first matrix, '2' for the second and so on — to build the file name. For example,

```
zsp -g 3 mat sd subsp quotsp
```

reads three matrices from `mat1, mat2, mat3`, seed vectors from `sd`, and writes the action on the subspace to `subsp1`, `subsp2`, `subsp3` and the action on the quotient space to `quotsp1`, `quotsp2`, `quotsp3`.

#Gen is an optional second argument, which must be separated from *#Mat* by a dot. *#Gen* must be an integer with $2 \leq \#Gen \leq \#Mat$. If used, ZSP assumes that the first *#Gen* matrices are sufficient to generate the whole group. For example, `-g 5.3` means 'split 5 matrices, but use only the first three to spin up the seed vector'. By default, all matrices are assumed to be generators, so `-g 5.5` is equivalent to `-g 5`.

Working with permutations The generators may be permutations which act on vectors as usual (see the ZMU documentation). However, matrices and permutations must not be mixed. If an input file contains more than one permutation, only the first one is read in, and a warning message is printed telling the user that the rest of the file has been ignored. A further restriction is that only the action on the subspace, not on the quotient, is calculated.

Seed modes With the `-s` option the user controls how the program creates seed vectors from the *Seed* space. There are three methods available:

`-sg`[*Num*] Generate seed vectors by forming linear combinations of the vectors read from *Seed*. This is the default. An additional integer number, may be appended to select a single vector. Notice that there must be no space between `-sg` and the number.

`-sr`[*Num*] Read seed vectors one by one from *Seed* and spin up each vector individually. Again, a single vector can be selected by appending a number to the option.

`-ss` Read a whole space from *Seed* and calculate its closure under the operation of the matrices.

In the default mode (`-sg`) ZSP will try each vector in the linear span of the rows of *Seed* modulo scalar multiples. Each linear combination is assigned a number in the following way: Let $v_1,\ldots,v_n$ be the rows in *Seed*, and

$$k = k_n q^{n-1} + k_{n-1} q^{n-2} + \ldots + k_2 q + k_1 \qquad (0 \leq k_i < q)$$

an integer. Then, the linear combination corresponding to k is

$$v(k) = k_n v_n + k_{n-1} v_{n-1} + \ldots + k_1 v_1$$

Where the coefficients k_i are identified with field elements (as defined by the ZZZ arithmetic module). In order to avoid scalar multiples, the first non-vanishing coefficient is required to be one.

Output Options The `-n` and `-o` options control the program's output. By default, seed vectors are spinned up until an invariant subspace is found or there are no more seed vectors. In the first case, the action of all matrices on subspace and quotient is calculated, and a message saying

`VECTOR` *Num* `SPLITS: SUBSPACE` *SDim* `QUOTIENT` *QDim*

is printed. Otherwise the message is

```
SPANS WHOLE SPACE
```

or

```
EVERY SEED VECTOR SPANS WHOLE SPACE
```

depending on the seed method.

The `-n` option can be used if only the invariant subspace is needed, but not the action of the matrices. Instead of *SubX* and *QuotX* there is one single argument, *Space*, which is taken as a base name for the invariant subspaces generated. The actual file name is built by appending a 4-digit number to *Space* which identifies the corresponding seed vector. By default, subspaces are written to `V`*nnnn*. For each seed vector, a message of the form

```
VECTOR n SPINS UP TO Dim1 QUOTIENT Dim2
```

is printed. Note that there is no output, if a vector spins up to the whole space. If the `-ss` option is used, there is only one output file, and no number is appended to *Space*.

With the `-o` option no output files are generated, but the same messages as above are printed.

Setting a dimension limit If it is known that there are no proper invariant subspaces of dimension greater than *Maxdim*, you can tell this to ZSP with

-d *Maxdim*

Then, if during the spin-up process the subspace dimension reaches *Maxdim*, ZSP assumes that the vector spans the whole space and stops with

```
SPANS WHOLE SPACE
```

Quick mode If the `-t` option is used, ZSP runs in 'quick mode'. For example,

```
zsp -t 5 -n gen1 gen2 seed
```

spins up vectors from `seed` under the gererators `gen1` and `gen2`. If, during the spin-up process, 5 vectors in sequence have their images under the generators already in the subspace, the space found so far is considered invariant. Note that this assumption may be valid in many cases, but in general it is not! In the above example, a non-invariant subspace may be written out with no warning. Trying to split with a noninvariant subspace (i.e., if you don't use the `-n` option) will produce a warning message, though.

Input files The matrices must be square, of the same size, and over the same field. There is no requirement for the matrices to be non-singular, and they often are not (e.g., if they are condensed matrices).

Seed must have the same number of columns and be over the same field as the matrices. The rows of *Seed* should be linearly independent but need not be in echelon form. It is not checked that the rows of *Seed* form a basis, but if they are linearly dependent the program becomes ineffective and will probably produce the following error message:

```
ZSP ERROR - CANNOT SPIN UP ZERO VECTOR
```

EXAMPLES

Let `spc` be a subspace and `m1`, `m2`, ... be matrices. To calculate the closure of *spc* under `m1` and `m2` type

```
zsp -ss m1 m2 spc closure
```

The same, but for 5 matrices `m1`, ..., `m5` is done with

```
zsp -ss -g 5 m spc closure
```

Another use of this program is in Norton's criterion of irreducibility. To prove that a given module is irreducible, find an element in the algebra with small nullity (≤ 3 is preferable). Let `z1` and `z2` be the action of the generators and `a` the algebra element. If the module is irreducible, this can be proved as follows (`$` is the shell prompt):

`$ znu a nsp`	Calculate the null space...
`NULLITY 3`	it has dimension 3
`$ zsp z1 z2 nsp s1 s2 q1 q2`	Try to split...
`EVERY SEED VECTOR SPANS WHOLE SPACE`	it fails!
`$ ztr z1 z1t`	Transpose first generator
`$ ztr z2 z2t`	Transpose second generator
`$ ztr a at`	Transpose the algebra element
`$ znu at nsp`	Calculate the null space
`$ zsp -s1 z1t z2t nsp s1 s2 q1 q2`	Try to split...
`SPANS WHOLE SPACE`	The module is irreducible!

LIMITS

All matrices plus one extra matrix (holding the subspace) must fit into memory. Additional memory is required for the seed vectors and for the pivot table, but the latter grow only linearly with the dimension.

With `-sg`, the dimension of the seed space must be less than 10. The maximal number of seed vectors generated and used for splitting is $(q^n - 1)/(q - 1)$, where q is the field order and n is the dimension of the seed space. Thus, when working with large fields, the run time may become very long. E.g., if $q = 19$ and $n = 6$ there are 2613660 seed vectors.

The number of matrices must be less than or equal to 10.

MESSAGES

```
ZSP ERROR - CANNOT SPIN UP ZERO VECTOR
```

The seed vector for the spin-up process was zero. This error occurs if the seed space basis is linearly dependent or if a zero vector has been read from *Seed*.

```
ZSP ERROR - CANNOT MAKE THIS SEED VECTOR
```

The requested seed vector does not exist. If you are using `-sr`, the highest possible vector number is the number of rows in *Seed*. With `-sg`, the highest possible number is $2q^{n-1} - 1$, where q is the field order and n ist the number of rows in *Seed*.

1.5.20 ZSY: Symmetrized tensor product

USAGE

`zsy` *Mode* `[`*Inp* `[`*Out*`]]`

DESCRIPTION

This program reads a matrix or permutation from *Inp*, calculates its symmetrized tensor product according to *Mode*, and writes out the result to *Out*. Default file names are G1 for input and P2 for output.

The *Mode* argument specifies the tensor product to be taken and the kind of symmetrization to be performed. Currently there are 5 Modes available: `s2`, `e2`, `e3`, `e4` and `m3`.

Mode	Description	Size of result (m)
`s2`	Symmetric square	$n(n+1)/2$
`e2`	Antisymmetric square	$n(n-1)/2$
`e3`	Antisymmetric cube	$n(n-1)(n-2)/6$
`e4`	Antisymmetric fourth power	$n(n-1)(n-2)(n-3)/24$
`m3`	Tensor cube, mixed symmetry	$n(n-1)(n+1)/3$

The meaning of the last column in this table is as follows: If *Inp* is a square matrix of dimension n, the output is a square matrix of dimension m. If *Inp* is a permutation of degree n, the output is a permutation of degree m. Since the typical application of ZSY is to generate new representations from existing ones, it will usually be used with square matrices. However, the input is not required to be square, except for `m3`.

Permutations Currently, only modes S2, E2 and E3 are available for permutations. The result gives the operation of the input permutation on unordered pairs (E2, S2) or triples (E3) of points. More precisely, if the given permutation operates on $\{1, \ldots, n\}$, then:

$$\begin{aligned} \mathrm{E2} &= \text{operation on } \{(i,k) \,|\, 1 \le i < k \le n\} \\ \mathrm{S2} &= \text{operation on } \{(i,k) \,|\, 1 \le i \le k \le n\} \\ \mathrm{E3} &= \text{operation on } \{(i,k,l) \,|\, 1 \le i < k < l \le n\} \end{aligned}$$

In the output, pairs and triples are numbered lexicographically. For example, E2 uses the following order: (1,2), (1,3), (2,3), (1,4), …, ($n-1$,n). Notice that the symmetric square is never transitive but decomposes into the diagonal and the antisymmetric square.

Here are some examples:

$$\begin{aligned} \pi &= (1\ 5\ 4\ 3\ 2) \\ E2(\pi) &= (1\ 7\ 10\ 6\ 3)(2\ 8\ 4\ 9\ 5) \\ S2(\pi) &= (1\ 15\ 10\ 6\ 3)(2\ 11\ 14\ 9\ 5)(7\ 14\ 8\ 4\ 12) \\ E3(\pi) &= (1\ 5\ 8\ 10\ 4)(2\ 6\ 9\ 3\ 7) \end{aligned}$$

Matrices The r-th exterior power (E2, E3, E4) has as its entries the determinants of $r \times r$ submatrices of the input. Rows and columns are ordered lexicographically, which is equivalent to taking the following basis in the tensor product:

E2: $v_i \wedge v_j$ with $1 \le i < j \le n$
E3: $v_i \wedge v_j \wedge v_k$ with $1 \le i < j < k \le n$
E4: $v_i \wedge v_j \wedge v_k \wedge v_l$ with $1 \le i < j < k < l \le n$

The ordering of basis vectors is lexicographical, for example (E2): $v_1 \wedge v_2$, $v_1 \wedge v_3$, …, $v_1 \wedge v_n$, $v_2 \wedge v_3$, $v_2 \wedge v_4$, …, $v_3 \wedge v_n$, …, $v_{n-1} \wedge v_n$.

The symmetric square of a matrix with r rows and c columns is a matrix with $r(r+1)/2$ rows and $c*(c+1)/2$ columns, with entries given by the formulae

	$c(c-1)/2$	c
$r*(r-1)/2$	$ad+bc$	ac
r	$2ab$	a^2

where the upper left is the $r(r-1)/2$ by $c(c-1)/2$ matrix of permanents. The program orders both the rows and the columns in lexicographical order, i.e. v_1v_2, v_1v_3, …, v_1v_n, v_2v_3, v_2v_4, …, v_2v_n, v_3v_4, …, $v_{n-1}v_n$, v_1v_1, v_2v_2,…, v_nv_n with the assumption that $v_iv_j = v_jv_i$, i.e. the action is on quadratic polynomials.

The symmetric square is, in general, irreducible except in characteristic 2. In that case there is a copy of the Frobenius square as an invariant submodule, as can be seen from the $2ab$ in the above formulae. Invariant subspaces in characteristic 2 correspond to special groups (i.e. groups of the form $2^n \times 2^m$) on which the group given acts on the quotient 2^n.

The mixed symmetry component of the tensor cube (M3) is calculated with respect to the basis

$$\begin{aligned} w_{ikl} &:= v_i \otimes v_k \otimes v_l + v_l \otimes v_k \otimes v_i - v_k \otimes v_i \otimes v_l - v_k \otimes v_l \otimes v_i \\ \text{where } i &= 1, \ldots, n-1 \end{aligned}$$

$$\begin{aligned} k &= i+1,\ldots,n \\ l &= i,\ldots,n \end{aligned}$$

Here are some examples:

$$E2\begin{pmatrix} 1 & 2 & 1 & 3 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 1 & 3 & 6 & 2 \\ 0 & 1 & 0 & 2 & 0 & 4 \\ 6 & 5 & 6 & 5 & 1 & 4 \end{pmatrix} \pmod 7$$

$$E3\begin{pmatrix} 1 & 0 & 2 & 0 & 2 \\ 1 & 1 & 2 & 1 & 2 \\ 3 & 3 & 2 & 3 & 2 \\ 1 & 2 & 3 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 4 & 3 & 4 & 0 & 3 & 2 & 1 & 3 & 4 \\ 1 & 2 & 2 & 3 & 2 & 3 & 1 & 3 & 4 & 2 \\ 4 & 0 & 4 & 0 & 2 & 0 & 4 & 2 & 1 & 3 \end{pmatrix} \pmod 5$$

$$S2\begin{pmatrix} 1 & 2 & 1 & 3 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 1 & 5 & 5 & 7 & 0 & 2 & 2 & 3 \\ 4 & 3 & 6 & 6 & 12 & 9 & 1 & 4 & 2 & 9 \\ 1 & 2 & 1 & 6 & 5 & 8 & 0 & 2 & 4 & 3 \\ 4 & 2 & 6 & 4 & 12 & 6 & 1 & 4 & 1 & 9 \\ 0 & 0 & 0 & 4 & 2 & 4 & 0 & 1 & 4 & 1 \\ 4 & 4 & 6 & 8 & 12 & 12 & 1 & 4 & 4 & 9 \end{pmatrix} \pmod{13}$$

BUGS

If the input file contains more than one permutation, only the first permutation is read in and processed.

LIMITS

If the input is a matrix, the whole input matrix and one row of the result must fit into memory. In case of permutations both the input and the result must fit into memory.

1.5.21 ZTC: Trace

USAGE

`ztc [-G] [`*Input*`]`

DESCRIPTION

This program reads a matrix or permutation from *Input* (or from G1, if no file name is given), calculates its trace and outputs this to the user.

If the input file is a matrix, it is read row by row and the diagonal entries are added up. Note that the matrix need not be square. The result is printed in the form

`TRACE IS` n

where it n is an integer representing the trace in GF(q). See section 3.1.3 for details. In particular, elements of the prime field are represented by $0, 1, \ldots, p-1$.

Permutations are read one by one from the input file, and their trace, i.e., the number of fixed points is calculated. The result is printed in the form

`ELEMENT` i `HAS TRACE` n

The `-G` option produces GAP output.

1.5.22 ZTE: Tensor product of matrices or permutations

USAGE

`zte` [*A* *B* *Result*]

DESCRIPTION

This program reads in two matrices or permutations from A and B and writes out their tensor product to *Result*. If no file names are specified, input is read from G1 and G2, and the result goes out to P3.

The program first reads in the input files. If they are matrices, their tensor product is calculated row by row and written to the output file. The tensor product of a $m \times n$ matrix A with a $m' \times n'$ matrix B is a $mm' \times nn'$ matrix C the entries of which are formed by multiplying the entries of A by the entries of B. The standard ordering is used, i.e., the entries of A are used to direct the placing of matrix B in the result:

$$C_{(i-1)m'+k,(j-1)n'+l} = A_{i,j}B_{k,l}$$

An example may make this clearer:

$$\begin{array}{ccccc} G1 & & G2 & & P3 \\ \begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 0 \end{pmatrix} & \otimes & \begin{pmatrix} 2 & 0 \\ 1 & 3 \end{pmatrix} & = & \begin{pmatrix} 2 & 0 & 4 & 0 & 6 & 0 \\ 1 & 3 & 2 & 6 & 3 & 9 \\ 2 & 0 & 2 & 0 & 0 & 0 \\ 1 & 3 & 1 & 3 & 0 & 0 \end{pmatrix} \end{array}$$

If the input is two permutations on n and n' points, respectively, then the result is a permutation on nn' points, which describes the action of (A, B) on ordered pairs of points. This action is defined in the obvious way: (i, k) maps to (iA, kB). In the output, pairs are represented as numbers using the lexicographic ordering $(1, 1)$, …, $(1, n')$, $(2, 1)$, …, (n, n').

LIMITS

Both input files and, in the case of matrices, one row of the output matrix must fit into memory at the same time.

1.5.23 ZTM: Tensor multiply

USAGE

`ztm` *A* *B* *Vectors* *Result*

DESCRIPTION

This program reads two matrices from A and B, a list of vectors from *Vectors*, and calculates the image of the vectors under $A \otimes B$. The result is written to *Result*. Thus, A and B must be square matrices of dimension m and n over the same field, *Vectors* must be a matrix over the same field with mn columns and an arbitrary number of rows, and *Result* has the same size as *Vectors*. This calculation could be done with the ZMU and ZTE programs, but using ZTM avoids the memory-consuming calculation of $A \otimes B$.

IMPLEMENTATION

Let m, n be the dimenstions of A and B, respectively. For each input vector $v \in F^{1\times mn}$, the program computes the $m \times n$ matrix $\hat{v}$ with

$$\hat{v}_{ik} = v_{(i-1)n+k} \qquad (1 \le i \le m,\ 1 \le k \le n)$$

Then, $v(A \otimes B)$ can be computed by ordinary matrix operations:

$$v(\widehat{A \otimes B}) = A^{tr} \cdot \hat{v} \cdot B$$

Limits

Both matrices and one vector must fit into memory at the same time.

1.5.24 ZTR: Transpose a matrix

Usage

`ztr` *Matrix Result*

Description

This program reads in a matrix, calculates its transpose in memory and writes out the result.

Limits

The whole matrix plus one additional row must fit into memory.

1.5.25 ZTS: Tensor Spin

Usage

`zts [-g` *NumGen*`]` *M N Seed Result*

Description

This program is similar to ZSP, but works on the tensor product of two modules, $M \otimes N$. ZTS spins up one or more vectors, and calculates a matrix representation corresponding to the invariant subspace. The program does not use the matrix representation of the generators on $M \otimes N$, which would be too large in many cases. This program is used, for example, to spin up vectors that have been uncondensed with TUC.

The action of the generators on both M and N must be given as square matrices. These matrices are read from M`.1`, M`.2`,..., and N`.1`, N`.2`,..., respectively. You can use the `-g` option to specify the number of generators. The default is two generators. Seed vectors are read from *Seed*. They must be given with respect to the lexicographically ordered basis explained below.

Implementation

Let $B = (b_1, \dots, b_m)$ be a basis of M, $C = (c_1, \dots, c_n)$ a basis of N, and denote by $B \otimes C$ the lexicographically ordered basis $(b_1 \otimes c_1, b_1 \otimes c_2, \dots b_m \otimes c_n)$. For $v \in M \otimes N$, the coordinate row $m(v, B \otimes C)$ has mn entries which can be arranged as a $m \times n$ matrix (top to bottom, left to right). Let $M(B, v, C)$ denote this matrix. Then

$$M(B, va, C) = m(B, a|_M, B)^{\mathrm{tr}} M(B, v, C) m(C, a|_N, C) \quad \text{for all } a \in A,\ v \in M \otimes N$$

Using this relation, we can calculate the image of any vector $v \in M \otimes N$ under an algebra element a, and thus spin up a vector, without using the matrix representation of a on $v \in M \otimes N$.

See Also

TUC, ZSP

1.5.26 ZUK: Uncondense

USAGE

`zuk` *Space OrbSz Result*

DESCRIPTION

This program reads a matrix which is assumed to be a condensed space of a permutation representation whose orbit sizes are in the file *OrbSz*. The vectors of *Space* are elongated so as to lie in the original permutation space and written out to the file *Result*.

OrbSz must be an orbit sizes file, i.e., an integer matrix with one row and the same number of columns as *Space*.

If these checks are passed, the program reads the orbit sizes into memory, and then proceeds row by row through the input matrix using the orbit sizes to dictate how often to repeat each entry of the input vector.

Here is an example:

$$\mathit{Space} = \begin{pmatrix} 2 & 0 & 4 \\ 1 & 3 & 2 \\ 2 & 0 & 2 \end{pmatrix}, \quad \mathit{OrbSz} = (2,4,3), \quad \mathit{Result} = \begin{pmatrix} 2 & 2 & 0 & 0 & 0 & 0 & 4 & 4 & 4 \\ 1 & 1 & 3 & 3 & 3 & 3 & 2 & 2 & 2 \\ 2 & 2 & 0 & 0 & 0 & 0 & 2 & 2 & 2 \end{pmatrix}$$

1.5.27 ZVP: Vector permute

USAGE

`zvp [`*Options*`]` *Mat1 Mat2 Seed Perm1 Perm2* `[`*Orbit*`]`
`zvp -g` *#Mat* `[`*Options*`]` *Mat Seed Perm* `[`*Orbit*`]`

DESCRIPTION

This program reads two or more matrices and one or more vectors from *Seed*, and finds the orbit of the vector under the matrices. The action of the matrices on the orbit is written out in permutation form.

Mat1 and *Mat2* must be square matrices of equal size over the same field. *Seed* must be a matrix over the same field with an equal number of columns.

If used without the `-g` option, ZVP reads two matrices from *Mat1* and *Mat2*, and output goes to *Perm1* and *Perm2*. A number of matrices greater than 2 can be specified with the `-g` option. In this case, *Mat* and *Perm* are taken as base names, i.e., ZVP reads matrices from *Mat*1, *Mat*2,... and writes permutations to *Perm*1, *Perm*2, ...

Available options are:

Option	Meaning
`-g`	Select number of matrices (see above).
`-n`	No output to files, messages only.
`-p`	'Projective' mode. In this mode ZVP operates on 1-dimensional subspaces rather than vectors. All vectors in the orbit file will be normalized (i.e., their first non-zero entry is equal to one).
`-v`	Write the orbit to *Orbit*.
`-l` *Limit*	Set the maximal orbit size. ZVP will not try to calculate orbits greater than this limit. The default value for the maximal orbit size is 100000.
`-sg`[*Num*]	Take *Seed* as a basis and produce seed vectors from this basis by linear combination. This is also the default. *Num* is the first vector to be used as seed vector. Here the numbering of seed vectors is the same as in the ZSP program. Note that there must be no space between `-sg` and *Num*.

`-sr`[*Num*] Read seed vectors one by one from *Seed*, do not make linear combinations.

The `-l` option can be used to allow for orbits longer than 100000 vectors, or to save memory if the orbit is known to be short. Notice that `-sg` and `-sr` work the same way as with ZSP.

After initializing everything and reading the input files the program enters the main loop. The next seed vector is read in, or generated, and the orbit is set up initially to contain only this vector. Then, the complete orbit is calculated by applying the two matrices to vectors in the orbit until no new vectors appear, or until the limit is reached. In the latter case, the program prints a message

`ORBIT` n `HAS MORE THAN` m `VECTORS`

and proceeds with the next seed vector. If an orbit has been found, the action of the two matrices on the orbit is written out as two permutations and the user is informed with the message

`ORBIT` n `HAS` m `VECTORS`

If the `-v` option was used, also the orbit is written out.

LIMITS

The matrices, seed vectors, and all vectors in the orbit must fit into memory.

1.6 The Lattice Package

1.6.1 Introduction

The MeatAxe module lattice package is a collection of programs which extends the set of standard MeatAxe programs. Its purpose is to calculate all submodules of a given module.

It is assumed that the algebra is generated by two elements. These generators must be available as two matrices in the files `X.1` and `X.2`, say (of course, `X` can be any other name). At the end of the calculation there will be a file `X.out` containing a list of all submodules of the given module.

The whole process consists of 7 steps each of which is done by a separate program. For example, given the two generators in `X.1` and `X.2`, one would execute the following commands:

```
chop X
pwkond X
mkcycl X
mkinc X
mkdotl X
mksub X
```

Instead of typing these lines separately you may want to create a shell script which calls all programs automatically. If all programs have been run without errors you will find the results in the following files:

`X.out`	A text file containing the submodule lattice. This is probably the only output file you want to look at. The files listed below are more difficult to read.
`X.cfinfo`	A list of all composition factors including dimension, multiplicity, number of mountains and number of dotted lines.
`X.dot`	The dotted lines (binary format).
`X.inc`	The incidence matrix (binary format).

Only the `.out` and `.cfinfo` files are intended as output to the user. The other files contain important information, too, but they are in binary format and cannot be read directly. Besides, there will be a lot of other files which have been created during the computation. See the following sections for an explanation of these files.

1.6.2 CHOP: Find irreducible constituents

USAGE

`chop [-G] [-g` *NGen*`] [-s` *Start*`] [-n` *MaxNul*`] [-d` *MaxDeg*`]]` *Name*

DESCRIPTION

This program reads two matrices from *Name*`.1` and *Name*`.2` describing the action of two generators on the module, and calculates a composition series of the module. For each composition factor it writes out two generators to *CFName*`.1`, *CFName*`.2`. *CFName* is the name of the composition factor, which is built by appending the dimension and a letter to the module name. For example, `X10a.1` is the first generator of the first composition factor of dimension 10 of the module `X`. If a second, inequivalent composition factor of dimension 10 is found, it will be named `X10b` and so on. CHOP also creates a file called *Name*`.cfinfo` containing a list of all composition factors. This file is used by all subsequent programs.

Available options are:

`-help` Print a help screen.

`-G` Generate output in GAP format.

`-g` *NGen* Set number of generators, if different from two.

`-s` *Start* Start calculating words at position *Start* (see description below).

`-n` *MaxNul* Controls the treatment of null-spaces appearing in the chopping process. Only null-spaces with a dimension less than or equal to this limit are searched completely.

`-d` *MaxDeg* Sets a limit for the degrees of polynomials used in the chopping process.

IMPLEMENTATION

CHOP repeatedly splits a module into submodule and quotient until it arrives at the irreducible constituents. Thus, it finds a composition series. The program assumes that the algebra generated by the input matrices contains the unit matrix.

In order to split a given module or to prove its irreducibility the algorithm needs an element of the algebra with a non-trivial but low-dimensional kernel. Such elements are searched by taking linear combinations of certain products of the generators. See the description of the ZSM program for more details on the word generator.

For each element A generated in this way, the program calculates its characteristic polynomial and examines the irreducible factors. If $p(x)$ is an irreducible factor, $p(A)$ has a non-trivial kernel. Then, one vector of the kernel is chosen and the smallest submodule containing this vector is calculated. If the vector lies in a proper submodule, the action of the generators on this submodule as well as on the quotient are calculated and the same procedure is applied recursively to both submodule and quotient.

To avoid expensive matrix multiplications in the calculation of $p(A)$, there is a limit on the degree of $p(x)$. This limit can be set with the `-m` option and defaults to 5.

If a module cannot be split by the program, it may be irreducible. In order to prove this, CHOP uses Norton's criterion. This requires, however, to find an algebra element with a small kernel, because up to scalar multiples each vector in the kernel must be examined to see whether it spins up to the whole module. For this reason a 'nullity threshold' m is maintained by the program. Initially m is set to 3, but the user may choose a different value at run-time by using the `-m` *Init* option. Each algebra element a with $\dim(\ker a) \leq m$ is used for the Norton test.

The option `-s` *Start* may be used to make CHOP start with a word different from 1.

Algebra elements with trivial kernel are useless for the algorithm, so an attempt is made to avoid unnecessary computation of such elements. Once an element is known to have a trivial kernel on a given module M, the program will mark it as invertible and ignore it for all constituents of M.

If a constituent is irreducible but not absolutely irreducible, the nullity of any element in the algebra will be a multiple of $[E : F]$, where F is the ground field and E the splitting field. This situation is recognized by calculating the greatest common divisor d of all nullities which occur during the search. In order to prove that the splitting field degree is equal to d, the following method is used: Take a word with nullity d and two vectors v_1, v_2 in its null-space. Use these vectors as seeds for a standard basis algorithm. If the resulting representations are different, $[E : F]$ is less than d, and the word is discarded. Otherwise, the linear map which transforms one standard basis into the other is an endomorphism e of the module. If v_1, under the action of e, spins up to the whole null space, then $[E : F] = d$. Otherwise, take a third vector not in the span and repeat the procedure above. Again, this yields an endomorphism, or it turns out that $[E : F] < d$. These steps are repeated until a word with Nullity $[E : F]$ is found.

SEE ALSO

CFCOMP

1.6.3 PWKOND: Peak word condensation

USAGE

`pwkond [-t] [-n] [-p] [-e` *List*`]` *Name*

DESCRIPTION

For each irreducible constituent V_i, this program finds a 'peak word' and calculates the corresponding condensed module. A peak word for V_i is an algebra element a_i with the property

$$\begin{aligned} \dim\ker(a_i)|_{V_i} &= n \\ (a_i)|_{V_j} &= 0 \quad \text{for } i \neq j \end{aligned}$$

where $n = [E : F]$ is the degree of the splitting field extension. The `-e` option can be used to exclude 'bad' words from the search (see below). *List* is a list of integers or ranges of integers, for example

```
-e 57,82-112,289
```

The `-n` option disables the condensation. If this option is used, the program stops after the peak words have been found. If the `-t` option is specified, PWKOND transforms the generators of all irreducible constituents to the standard basis defined by the peak word.

For each composition factor there are several output files. If, for example, one composition factor is `X10a`, PWKOND would produce the following files:

`X10a.std.1` First generator in standard basis (only with -t)
`X10a.std.2` Second generator in standard basis (only with -t)
`X10a.op` Standard basis operations (see ZSB for details)
`X10a.1k` Condensed first generator
`X10a.2k` Condensed second generator
`X10a.np` Condensed peak word. This is a nilpotent matrix.
`X10a.im` Image of the peak word.
`X10a.k` Kernel of the peak word.

The `.cfinfo` file is written each time a peak word is found. So, if the program does not terminate or dies unexpectedly the information about the peak words found so far is not lost.

Implementation

Internally, a peak word is represented by a pair $(n, p(x))$ where n is the canonical number of the word[1], and $p(x)$ is a polynomial. The peak word represented by this pair is $p(W_n)$, W_n being the n-th word. Without `-p`, PWKOND considers only linear polynomials. If the `-p` option is used, PWKOND can find polynomials of any degree.

Whenever a peak word is found, the generalized condensation is calculated as follows: The peak-word, given by its canonical number, is evaluated as a matrix acting on V and then repeatedly raised to higher powers until the nullity stabilizes. The stable nullity equals the multiplicity k of the constituent times the degree $[E : F]$ of the splitting field extension. Having a power w^N of the peakword with stable nullity, the condensation onto its kernel, i.e., the projection of V onto $V/\mathrm{im}(w^N)$, is determined in the same way as it is also implemented in the ZQT program.

1.6.4 MKCYCL: Make cyclic submodules

Usage

`mkcycl` *Name*

Description

For each of the condensed modules a representative of each cyclic submodule is found. This is done by spinning up each vector, up to scalar multiples, and maintaining a list of different cyclic submodules. As the dimension of the module grows, the number of vectors to spin up quickly becomes very large. This poses an upper limit on the dimension of condensed modules, i.e., on the multiplicity of irreducible constituents. Over GF(2), for example, a 16-dimensional condensed module requires about 20 hours of CPU time on a workstation.

The output is, for each composition factor, a list of vectors (in matrix form) which generate all cyclic submodules. Its file name is *CFName*`.v`, for example `X10a.v`.

A second limit concerns the number of cyclic submodules. Usually there are much less cyclic submodules, invariant under both generators, as 1-spaces. Sometimes, however, it may happen that the peak word found in the second step is 'bad' in the sense that the condensed generators commute. In such a case one finds a large number of cyclic submodules and the following steps will probably take too much time. For this reason, the peak word program has an option to exclude one or more specified peak words from the search. So, if the peak word turns out to be 'bad', the user can try another one.

1.6.5 MKINC: Mountains and incidence matrix

Usage

`mkinc [-G]` *Name*

Description

This program runs in two steps. During the first step, all cyclic cyclic submodules found by `mkcycl` are uncondensed, giving the local submodules ('mountains') of the original module. Then, each local submodule is projected back to the condensed module, and all cyclic vectors which are contained in the image are found. At the end of this step, there is a list of local submodules and, for each local submodule, a list of cyclic subspaces in the condensed module. This information is written to *Name*`.mnt`.

In the second step, MKINC computes the incidence relation between local submodules. The result is a matrix which contains a '1' for each incidence. This matrix is written to the file *Name*`.inc`.

[1] See the description of the ZSM program for details.

Note that the whole calculation of step 2 is done in the condensed modules. This is possible because incidences between local submodules do not change if they are condensed. Usually this saves a lot of both memory and CPU time because one does not have to keep all mountains simultaneously, and the condensed modules have a smaller dimension.

Options

Option	**Meaning**
`-G`	Produce output readable by GAP.

1.6.6 MKDOTL: Dotted lines

USAGE

`mkdotl [-G]` *Name*

DESCRIPTION

This program calculates a set of dotted lines between the local submodules. More precisely, it computes one dotted line for each submodule with head $\cong S \oplus S$, S irreducible. It can be shown that this set of dotted lines is sufficient to determine the complete submodule lattice as described by Benson and Conway.

Input for this program are the incidence matrix calculated by MKINC and the cyclic submodules from `mkcycl`. Again, the whole calculation takes place in the condensed modules, so there is no need to uncondense the cyclic submodules.

It is known that all dotted lines have length $q + 1$, where q is the order of the splitting field. This information is used by the program to determine if a dotted line is complete.

A list of all dotted lines is written to *Name*`.dot`.

Options

Option	**Meaning**
`-G`	Produce output readable by GAP.

1.6.7 MKSUB: Find submodules

USAGE

`mksub [-b] [-o`*Format*`]` *Name*

DESCRIPTION

This program calculates the submodule lattice using the output generated by MKINC and MKDOTL. In this final step no matrix operations are involved. Instead the program works with bit strings representing the incidences and dotted-lines. The lattice may be decomposed into blocks using the `-b` option (see below).

Submodules are calculated generation by generation, the first generation consisting of all submodules generated by one local submodule. In the n^{th} generation all submodules generated by a submodule of generation n plus one local submodule are calculated. If no more submodules appear, the algorithm terminates.

Output is written to three text files and one binary file. The first file, *Name*`.out`, contains a list of irreducible constituents, the incidence matrix, the dimensions of all local submodules, a list of dotted lines, a list of all submodules, the radical and socle series, and a list of all local submodules ('mountains'). The second output file is *Name*`.lat`. It contains the lattice as a list in GAP format. This list contains, for each submodule, its dimension, maximal submodules and isomorphism types of simple factors are given (see the example below). The third output file, *Name*.gra, contains a description of the submodule lattice together with some additional information. This file is read by the MKGRAPH program to produce a graphical representation of the lattice. A further output file, *Name*.sub contains the submodule lattice is binary format. This file is read by GENMOD (see below).

Here is an example:

File m11.out:

```
Irreducibles:
    Type   Mult   SF   Mountains           Dotted lines
    1a     4      1    5 (0-4)             1 (0-0)
    10a    2      1    2 (5-6)             0
    44a    2      1    3 (7-9)             1 (1-1)

Mountains:
    No     Dim    Maximal Submountains
    0      112    2 6 7 9
    1      46     8
    2      56     5 8
    3      12     5
    4      1
    5      11     4
    6      66     1 5
    7      56     3
    8      45     4
    9      56     3

Incidence matrix:
      0: ++++++++++
      1: .+..+...+.
      2: ..+.++..+.
      3: ...+++....
      4: ....+.....
      5: ....++....
      6: .+..+++.+.
      7: ...+++.+..
      8: ....+...+.
      9: ...+++...+

Dotted lines:
    .+++......
    .......+++

Submodules:
    No    Dim  Flags  Ident                          Max
    0     0     R     ..........
    1     1    MRS    ....+.....                     0
    2     11   M      ....++....                     1
    3     12   M      ...+++....                     2
    4     45   M      ....+...+.                     1
    5     55    RS    ....++..+.                     2,4
    6     46   M      .+..+...+.                     4
    7     56   M      ...+++.+..                     3
    8     56          .+..++..+.                     5,6
```

```
    9     56   M      ...+++...+                        3
    10    56          ...+++..+.                        3,5
    11    66   M      .+..+++.+.                        8
    12    56   M      ..+.++..+.                        5
    13    57    RS    .+++++..+.                        8,10,12
    14    100         ...+++.+++                        7,9,10
    15    67          .++++++.+.                        11,13
    16    101         .+++++.+++                        13,14
    17    111   RS    .+++++++++                        15,16
    18    112  M S    ++++++++++                        17

Radical series:
    Layer 1: Dim=111   1a
    Layer 2: Dim=57    10a 44a
    Layer 3: Dim=55    1a 1a
    Layer 4: Dim=1     10a 44a
    Layer 5: Dim=0     1a
```

File m11.gra:

```
19
.r.  0
mrs  1 0 0
m..  1 1 1
m..  1 2 0
m..  1 1 2
.rs  2 2 2 4 1
m..  1 4 0
m..  1 3 2
...  2 5 0 6 1
m..  1 3 2
...  2 3 2 5 0
m..  1 8 1
m..  1 5 0
.rs  3 8 0 10 0 12 0
...  3 7 2 9 2 10 2
...  2 11 0 13 1
...  2 13 2 14 0
.rs  2 15 2 16 1
m.s  1 17 0
```

Blocks If the `-b` option is used, MKSUB tries to decompose the lattice into 'blocks'. By definition, a block is a set of one or more composition factors which is closed under incidences of local submodules. For a decomposition into blocks to be possible, there must be direct summands with no common irreducible constituent. If a decomposition exists, the whole lattice can be reconstructed from its blocks by forming direct sums.

Output files are created separately for each block, and a number is appended to the name. For example, if the representation is called `psl277` and the lattice decomposes into 3 blocks, `mksub -b psl211` creates 9 output files:

```
psl211.out.1 psl211.lat.1 psl211.gra.1
psl211.out.2 psl211.lat.2 psl211.gra.2
psl211.out.3 psl211.lat.3 psl211.gra.3
```

Changing the output format The default output as shown above may be changed by using the `-o` option. *Format* is any combination of `m`, `d`, `i`, `e`, `s`, `r`, `s`. Each letter corresponds to a certain piece of output:

`m`	Mountains
`d`	Dotted lines
`i`	Incidence matrix
`e`	`.lat` and `.gra` files
`s`	Submodules
`r`	Radical series
`o`	Socle series

1.6.8 GENMOD: Make submodules

USAGE

`mksub [-m]` *Name Number*

DESCRIPTION

This program makes a submodule, i.e., it takes the mountains contained in that submodule and spins them up. The result is written to *Name*`.s`*Number*.

If the `-m` option is used, *Number* is treated as the number of a mountain. This mountain is generated and written to *Name*`.m`*Number*.

1.6.9 CFCOMP: Compare constituents

USAGE

`cfcomp` M U ...

DESCRIPTION

After a module M has been chopped, you can use this program to determine if a given module U is an irreducible constituent of M. If yes, the program finds out which of the constituents $S_1, \ldots, S_l$ of M is isomorphic to U. The group action on U can be given with respect to any basis. U need not be irreducible, but using the program with a reducible U makes little sense.

The program need at least two arguments. The first argument is the name of the chopped module, M, and the second argument is the name of U. If the algebra has r generators — this number is found in the M`.cfinfo` file — the action of the generators on S is read from *Irred*`.1`,..., *Irred*`.`*r*. You can specify more than one U on the command line.

IMPLEMENTATION

The program uses the identifying words computed by CHOP. For each constituent S_i which hase the same dimension as U, the identifying word is evaluated on U, and a vector from its kernel is used to produce the standard basis. Then, the generators are transformed to standard basis and compared with the generators on S_i.

SEE ALSO

CHOP

1.7 The Tensor Condensation Package

Given two kG-modules M and N, and a subgroup $H \leq G$ such that $\mathrm{char}(k)$ does not divide $|H|$, the tensor condensation programs calculate a matrix representation on the condensed module $(M \otimes N)e$, where $e_H := \frac{1}{|H|} \sum_{h \in H} h$.

1.7.1 TCPEAK: Tensor condensation

Usage

`tcpeak` M N M_H N_H $NGen$ $NGen_H$

Description

Given a group G and two FG-Modules M and N, this program calculates the condensation of $M \otimes N$ with respect to a subgroup $H \leq G$. $Ngen$ is the number of generators of G, and $Ngen_H$ is the number of generators of H. The program expects the following input files:

M.1,...,M.$NGen$	Group action on M
N.1,...,N.$NGen$	Group action on N
M_H.1,...,M_H.$NGen_H$	Action of H on M
N_H.1,...,N_H.$NGen_H$	Action of H on N

The output consists of $NGen$ matrices which describe the group action on the condensed module. These matrices are written to `result.1,...,result.`$NGen$.

Note that, even if $a_1, \ldots, a_r$ is a generating set of kG, the condensed elements $e_H a_1 e_H, \ldots, e_H a_r e_H$ may not generate the whole condensed Algebra $e(kG)e$.

Implementation

The tensor condensation procedure comprises four steps:

1. Run CHOP and PWKOND on both M_H and N_H.
2. Transform M and N to a semisimplicity basis using BASBYPEAK (see below).
3. Run PRECOND (see below) to calculate the condensation on tensor products of irreducible constituents.
4. Run TKOND (seee below) to calculate the condensation of $M \otimes N$.

1.7.2 BASBYPEAK: Make symmetry basis using peak words

Usage

`basbypeak` M $Basis$ [$StdGen$]

DESCRIPTION

Given a semisimple A-module V, this program computes a basis that reflects the module's semisimplicity. With respect to this basis, the matrix representation M of A on V has a the form

$$M(a) = \begin{pmatrix} S_1(a) & & & & & & & & \\ & \ddots & & & & & & & \\ & & S_1(a) & & & & & & \\ & & & S_2(a) & & & & & \\ & & & & \ddots & & & & \\ & & & & & S_2(a) & & & \\ & & & & & & \ddots & & \\ & & & & & & & S_l(a) & \\ & & & & & & & & \ddots & \\ & & & & & & & & & S_l(a) \end{pmatrix} \quad \text{for } a \in A$$

where the S_i are the matrix representations of the irreducible constituents of V computed by CHOP. Note that all blocks corresponding to the same constituent are equal, not only equivalent, and that the blocks occur in their 'natural' order (as defined by CHOP). This is essential for the following tensor condensation procedure (see below). The program depends on the peak words found by PWKOND.

M is the name of the module, and *Basis* is the name of the output file. If the third argument is present, the generators are transformed to the new basis and written to *StdGen*.1,...,*StdGen*.*n*.

1.7.3 PRECOND: Tensor condensation on consituents

USAGE

`precond` *Info* M_H N_H

DESCRIPTION

This program performs two tasks:

1. It compares the irreducible constituents of M_H and N_H, and finds all pairs (S_i, T_j) of constituents where $S_i \cong T_j^\star$.
2. For each pair (S, T) of constituents found in step 1, the program calculates the embedding of $(S \otimes T)e_H$ into $S \otimes T$ as an direct summand, and the corresponding projection of $S \otimes T$ onto $(S \otimes T)e_H$.

IMPLEMENTATION

Step 1, matching of constituents, is implemented in the same way as in CHOP and CFCOMP, i.e., by using the standard basis with respect to identifying words. Step 2 is based on two observations:

(A) $V \otimes V^* \cong \mathrm{Hom}_k(V, V)$, and $(S \otimes T)e_H \cong \mathrm{End}_{kH}(V)$ as kH-Modules.

(B) There is a natural, H-invariant non-degenerate scalar product on $\mathrm{Hom}_k(V, V)$, given by $\Gamma(\phi, \psi) = \mathrm{Trace}(\phi\psi)$.

From (A) it is clear that calculating the embedding of $(S \otimes T)e_H$ into $S \otimes T$ is equivalent to computing a basis of $\mathrm{End}_{kH}(V)$. The latter is easily accomplished using the peak word ov V. As a consequence of the second observation, there is a natural one-to-one correspondence between H-invariant linear forms on $\mathrm{Hom}_k(V, V)$ and $\mathrm{End}_{kH}(V)$, which is used to calculate the projection from $\mathrm{Hom}_k(V, V)$ on $\mathrm{End}_{kH}(V)$.

1.7.4 TKOND: Tensor condensation

Usage

`tkond -g` *NGen* M N M_H N_H *Info*

Description

This program completes the tensor condensation procedure. It calculates, for one or more elements $a \in A$, the action of $e_H a e_H$ on the condensed tensor product $(M \otimes N)e_H$.

1.7.5 TUC: Tensor Uncondense

Usage

`tuc` M_H N_H *Info* *Cond* *Uncond*

Description

This program uncondenses one or more vectors, i.e., it calculates the embedding of $(M \otimes N)e_H$ into $(M \otimes N)$. The input vectors must be given with respect to the basis defined by PRECOND. The uncondensed vetors ae calculated with respect to the symmetry basis produced by BASBYPEAK.

See Also

TCPEAK, ZTS

Chapter 2

Programmer's guide

2.1 Introduction

Since it was ported from FORTRAN to C in 1989 the MeatAxe has been freely available, including source code, under the terms of the GNU general public license. Users are encouraged to extend or modify the programs as needed. Indeed, most of the recent improvements and additions to the MeatAxe are the result of users' reports, requests, and experiments.

This documentation is intended to provide the information you need to write your own MeatAxe programs or to modify existing ones. Since the software is changing continuously the documentation is necessarily incomplete and not everywhere correct. So, even with this manual, you should be prepared to look in the source code, which is always the ultimate reference. Starting with release 2.3 I have tried to merge documentation and source code, which should help to keep the documentation more up-to-date.

Internally, the MeatAxe is built in several layers where each layer uses services provided by the lower layers.

1. **The kernel layer.**
2. **The object layer.** This layer encapsulates low-level functionality in 'objects'[1] like matrices, permutations, and polynomials. It also provides the elementary operations with these objects like, for example, matrix multiplication.
3. **The Algorithm layer.** This layer provides more complex operations which typically involve several objects. The spin-up and split functions are examples for this layer.
4. **The program layer.** The top layer comprises the MeatAxe programs and shell scripts like ZAD, ZMU, CHOP, ...

Throughout this documentation, layers 1 to 3 are referred to as the MeatAxe *library*. Indeed, these layers are implemented as a library which is built during the compilation of the MeatAxe.

2.1.1 Using the library: Compiling and Linking

To use the MeatAxe library in your programs you need two files:

`libmtx.a`	The library file. This file is generated during the compilation of the MeatAxe.
`meataxe.h`	The header file for all MeatAxe library functions. This file is part of the distribution.

You must include the header file in your program by putting the line

```
#include "meataxe.h"
```

[1] This does not mean that the MeatAxe is written in an OO language.

somewhere at the top of your program. If `meataxe.h` resides in the same directory as your program the compiler will find it there. Otherwise you must specify a complete path in the include statement or specify the directory on the command line, for example:

```
cc -c -I/usr/joe/mtx/src -o myprog.o myprog.c
```

Having compiled your program you must link it with the MeatAxe library. Here is an example:

```
cc -o myprog myprog.o libmtx.a
```

2.1.2 Documentation overview

The main part of the documentation is contained in the following two chapters. In chapter 3 library functions are grouped by category to give a quick overview and to explain the underlying concepts. Chapter 4 contains an alphabetical reference of all library functions.

Chapter 3

Library overview

3.1 The MeatAxe kernel

The kernel

- provides the finite field arithmetic and low-level vector operations, and
- provides a platform-independent interface to operating system services, including the interaction with the user.

There are two kernels available: one for small fields (up to 256) and one for larger fields (up to 2^{16}). The kernel is selected at compile time. Kernel functions are optimized for high speed but they lack some features of the higher layers levels like error checking and debugging capabilities. Most kernel services are simple, but there are some more complex functions like Gaussian elimination, which have been implemented in the kernel for performance reasons.

Most of the kernel functions have names beginning with 'z'. Kernel functions generally do not allocate memory dynamically and do not check their arguments. The kernel cannot operate simultaneously with different finite fields, because there is a global row size and a global field order which must be maintained by the higher layers.

Simple Data types The kernel defines two basic data types

`FEL` .... A finite field element
`PTR` .... A pointer to a vector (row). `PTR` may be defined as `FEL`, but this is not mandatory. However, it must be possible to dereference and to increment a `PTR`.

The kernel also defines two constants:

`F_ZERO` .... The zero element of the current field.
`F_ONE` .... The unit element of the current field.

Depending on which kernel you are using, `F_ZERO` and `F_ONE` need not be constants. They may be defined as variables or even function calls.

3.1.1 OS-dependent stuff

`SInit()` .... Initialize OS dependent stuff
`STimeUsed()` .... CPU time
`STimeLimit()` ...... Set CPU time limit
`SFOpen()` .... Open file
`SFSeek()` .... Set file pointer
`Smalloc()` .... Allocate memory
`Srealloc()` .... Resize memory block

3.1.2 File i/o functions

`zreadlong()` .... Read long integers
`zwritelong()` ...... Write long integers
`zreadvec()` .... Read rows
`zwritevec()` .... Write rows
`zseek()` .... Open file and read header
`zreadhdr()` .... Open file and read header
`zwritehdr()` .... Open file and write header
`zftogap()` .... Convert to GAP format

Binary file format Binary data files contain a sequence of objects. Each object consists of a 12-byte header, followed by the data part. The data is stored in the internal format (see section 3.1.3) with the following differences:

- Packed rows (vectors) are truncated to their actual length. For example, a row with 17 entries over GF(125) on a machine with `sizeof(long)` = 4 would occupy 20 bytes in memory but only 17 bytes in a data file.
- All integers, including the file header, are stored as 32-bit numbers with least significant byte first.

The object header consists of three integers. The first one, T, specifies which kind of object follows. The two other values, R and C, carry additional information about the object. Their exact meaning depends on the value of T.

T	Type of object	R	C
¿ 1	Matrix over GF(T)	no. of rows	no. of columns
-1	Permutation	no. of points	no. of permutations
-2	Polynomial	field	degree
-8	Integer matrix	no. of rows	no. of columns

3.1.3 Finite fields

`zchar` .... Characteristic
`zfl` .... Field order
`zgen` .... Field generator
`zadd()` .... Finite field addition
`zsub()` .... Finite field subtraction
`zmul()` .... Finite field multiplication
`zdiv()` .... Finite field division
`zinv()` .... Finite field inversion
`zneg()` .... Additive inverse
`zsetfield()` .... Set the field
`zembed()` .... Embed a subfield
`zrestrict()` .... Restrict to a subfield
`zitof()` .... Convert integer to field element
`zftoi()` .... Convert field element to integer
`FEL` .... Finite field element

Embedding of subfields In the MeatAxe there is a 'standard' generator for each finite field. The generator for the field currently in use is available in the `zgen` variable. Thus, if a and a' are the MeatAxe generators of GF(q) and GF(q'), respectively, and $q' = q^n$, there is a 'standard' embedding of GF(q) into GF(q') defined by $a \mapsto (a')^n$. However, field elements which are identified under this embedding are usually not represented by the same number. For this reason there are two functions, `zembed()` and `zrestrict()`, which provide the embedding of subfields into the current field. Note that the MeatAxe is not well suited for calculations involving different fields at the same time because the arithmetic uses lookup tables which are specific to each field

Here is a short example. The following code converts a vector over GF(3) to GF(27). The MeatAxe cannot handle two fields at the same time, so it is necessary to unpack the row over GF(3), change to GF(27) and pack the embedded elements into a new row.

```
PTR row1, row2;
FEL buf[10];
...
zsetlen(3,10);
for (i = 1; i <= 10; ++i) buf[i-1] = zextract(row1,i);
zsetlen(27,10);
for (i = 1; i <= 10; ++i) zinsert(row2,i,zembed(buf[i-1],3));
```

In the 'small' kernel, field elements of GF(q) are represented by the numbers $0, 1, \ldots, q-1$. The field is defined by its Conway polynomial $p(x)$, a polynomial of degree n over $\mathbb{Z}_p[x]$, where $q = p^n$. Thus, we have a one-to-one correspondence of field elements $a \in \mathrm{GF}(q)$ and polynomials $f_a(x) \in \mathbb{Z}_p[x]$ of degree less than n. By treating $\mathbb{Z}_p$ as a subset of $\mathbb{Z}$ — actually, on the computer, elements of $\mathbb{Z}_p$ are represented by integers — this is also a polynomial over $\mathbb{Z}$. Now, calculate $f_a(p)$ giving the number of the field element a. It follows that the elements of the prime field are represented by 0, 1, …, $p-1$. The number 0 represents the zero element, and 1 represents the unit element.

The big version uses always 2 bytes (an `unsigned short`) for field elements. Non-zero elements are stored as their logarithms with respect to a fixed generator, i.e. the unit element is represented by the number 0. The zero element is represented by the special value `0xFFFF`.

As a consequence of the different representations of field elements in the small and big version, there are some rules which should be respected by all programs:

- Never assign numbers to variables of type `FEL` or pass numbers to functions expecting an argument of type `FEL`.
- Never perform integer arithmetic on variables of type `FEL`.
- Never `printf` or `scanf` variables of type `FEL`.
- Never use the literals 0 or 1 where the zero and unit element of the field is intended. Instead you should use the `F_ZERO` and `F_ONE` constants define in `meataxe.h`.
- Do not cast an integer to `FEL` or vice versa. Instead use `zitof()` and `zftoi()`.

Converting between finite field elements and integers The MeatAxe defines a 'standard' correspondance between the elements of GF(q) and the set $\{0, 1, \ldots, q\}$. For prime fields, the mapping is defined by assigning the integer 1 to the unit element of the field. For non-prime fields, each $a \in \mathrm{GF}(p^n)$ is represented – modulo the ideal generated by the Conway polynomial of degree n – by a unique polynomial $f_a(x) \in \mathrm{GF}(p)[x]$ with $\deg(f_a) < n$. Using the standard embedding of GF(p) into $\mathbb{Z}$ we can consider f_a as a polynomial over $\mathbb{Z}$. Then, the number assigned to a if $f_a(p)$.

The mapping between GF(q) and $\{0, 1, \ldots, q\}$ is provided by two functions, `zitof()` and `zftoi()`. Since the actual numeric representation of field elements depends on the kernel, you cannot convert a `FEL` to an `int` simply by type casting.

3.1.4 Low-level row operations

`zcleanrow()` .... Clean row
`zcleanrow2()` ...... Clean row

`znoc` .... Current row size
`zrowsize` .... Current row size
`zinsert()` .... Insert a mark into a row
`zextract()` .... Extract a mark from a row
`zfindpiv()` .... Find pivot column
`zsetlen()` .... Set the row length
`zaddrow()` .... Add two rows
`zmulrow()` .... Multiply a row by a coefficient
`zaddmulrow()` ...... Add a multiple of a row
`zmaprow()` .... Multiply a vector by a matrix
`zextractcol()` ..... Extract one column of a matrix
`zalloc()` .... Allocate memory and initialize
`zsize()` .... Size of a row or matrix
`zmoverow()` .... Copy a row
`zswaprow()` .... Exchange two rows
`zcmprow()` .... Compare two rows
`zadvance()` .... Advance a row pointer
`zpermrow()` .... Multiply a vector by a permutation

Internal data format A (row) vector is stored in memory as an array of bytes. The actual size of a row depends on the number of marks in the row and on the field order, because several field elements may be packed into one byte. However, the size is always a multiple of `sizeof(long)`. Thus, there may be unused bytes at the end of a row. The contents of these extra bytes (and even the contents of unused bits in partially used bytes) is not defined. For this reason, memory must be initialized before it is used, or else some functions as 'compare rows' may fail.

Packing of field elements occurs for field order $q \leq 16$. Let q be the field order and m the largest natural number with $q^m \leq 256$, i.e. $m := \lfloor 8/\log_2 q \rfloor$. Then, m elements are packed into one byte using the packing function

$$\mathrm{pack}(k_0, \ldots, k_{m-1}) = \sum_{i=0}^{m-1} k_i q^i$$

Packing of field elements is used exclusively for rows (vectors), not for polynomials or other data types.

Matrices A matrix is stored as a sequence of rows with no extra space between them. The rows and columns of a matrix are numbered $1, 2, \ldots n$. Column vectors may be stored as matrices with one column but this is very inefficient because each mark of the vector will occupy 4 Bytes.

Accessing rows and matrices Because of packing you cannot access the marks of a row with the usual bracket operator. Instead, the MeatAxe library provides functions which store and extract marks. A second difference is that the first entry in a vector has index 1, not 0 as in C arrays.

Packed rows must be initialized before they are used. `zinsert()` (and many other row operations) will produce strange results if used with uninitialized rows. Memory is initialized automatically in the following cases: allocation with `zalloc()`, copying a row with `zmoverow()`, reading a row from a file. A row can be initialized manually by multiplication with zero: `zmulrow(ptr,F_ZERO)`. The latter works even if no field has been selected.

3.1.5 Low-level matrix operations

`zmkpivot()` .... Build pivot table
`zmkechelon()` ...... Reduce to echelon form
`zmatinv()` .... Matrix inversion
`znullsp()` .... Null space

zquotinit() .... Prepare for projection on the quotient
zquot() .... Projection on the quotient
zquotop() .... Operation on the quotient

3.1.6 Seed vector generation

zpseed_vec() ...... Seed vector buffer
zpseed_free() ..... Clean up the seed vector generator
zpseed_init() ..... Initialize the seed vector generator
zpseed_make() ..... Make a seed vector
zpseed_next() ..... Make the next seed vector

The seed vector generator is used to walk through the one-dimensional subspaces of a given vector space V, the *seed space*. For each one-dimensional subspace $U < V$ the generator produces a representant $u \in U$. These vectors are called *seed vectors*.

Once a basis for the seed space is fixed, each vector of the seed space can be identified by a natural number as follows. Let $b_1,\ldots,b_n$ be the basis and $v = \lambda_1 b_1 + \ldots \lambda_n b_n \in V$ a vector. The coefficients $\lambda_i \in \mathrm{GF}(q)$ are identified with positive integers in the usual way (see `zftoi()` for details) and

$$N(v) := \lambda_1 + \lambda_2 q + \lambda_3 q^2 + \ldots \lambda_n q^{n-1}$$

is the 'number of v'. Seed vectors are those vectors where the leading coefficient is 1.

To use the seed vector generator you must first call `zpseed_init()` to define the seed space. Then seed vectors can be generated by either `zpseed_make()` or `zpseed_next()`.

3.1.7 Miscellaneous

gcd() .... Greatest common divisor
lcm() .... Least common multiple
MtxRandomInit() .. Initialize the random number generator
MtxRandom() .... Random number

3.2 Matrices over finite fields

matrix_t .... Matrix data structure

In the MeatAxe, matrices over finite fields are represented by `matrix_t` structures. Besides the marks, each matrix carries the field order, the number of rows and the number of columns. There is no global field order or row length as at the kernel layer.

Matrices can be created in many ways, including (i) calling `MatAlloc()`, (ii) making a copy of an existing matix with `MatDup()`, and (iii) by reading a matrix from a data file with `MatRead()` or `MatLoad()`. The application is responsible for releasing matrices which are no longer needed. Matrices may consume lage amounts of memory, so it always a good idea to release a matrix as early as possible. There is only one possibility of releasing a matrix: calling `MatFree()`.

3.2.1 Basic matrix functions

MatAlloc() .... Create a new matrix
MatFree() .... Delete a matrix
MatId() .... Identity matrix
MatDup() .... Duplicate a matrix

3.2.2 Matrix operations

`CharPolSeed` .... Seed for characteristic polynomial
`CharPolFactor()` .. Characteristic polynomial
`CharPol()` .... Characteristic polynomial
`MatEchelonize()` .. Reduce to echelon form
`MatEchelon()` ...... Echelon form of a matrix
`MatNullity()` ...... Nullity of a matrix
`MatNullity__()` .... Nullity of a matrix
`MatNullSpace_()` ... Null-space of a matrix
`MatNullspace()` ... Null-space of a matrix
`MatNullSpace__()` .. Null-space of a matrix
`MatAddMul()` .... Add a multiple of a matrix
`MatAdd()` .... Add a matrix
`MatCopyRegion()` .. Copy a rectanglular region of a matrix
`MatCut()` .... Cut a rectangle out of a matrix
`MatInsert_()` ...... Insert a matrix into a polynomial
`MatInsert()` .... Insert a matrix into a polynomial
`MatOrder()` .... Order of a matrix
`MatTrace()` .... Trace of a matrix
`MinPolFactor()` ... Minimal polynomial
`MinPol()` .... Minimal polynomial
`MatMulScalar()` ... Calculate the multiple of a matrix
`MatTensor()` .... Tensor product
`MatMul()` .... Multiply matrices
`MatTr()` .... Transpose a matrix
`MatInv()` .... Invert a matrix
`MatIsEchelon()` ... Check if a matrix is in echelon form
`MatPower()` .... Power of a matrix

3.2.3 Matrix i/o functions

`MatPrint()` .... Print a matrix on stdout
`MatRead()` .... Read a matrix from a file
`MatWrite()` .... Write a matrix to a file
`MatLoad()` .... Read a matrix from a file
`MatSave()` .... Write a matrix to a file

3.3 Representations and vector spaces

`ChangeBasis()` ..... Change basis
`IsIsomorphic()` ... Compare representations
`MakeEndomoprhisms()`Find endomorphism ring
`SpinUp()` .... Spin up
`Split()` .... Split a module
`QuotProj()` .... Projection on quotient
`IsSubspace()` ...... Vector space incidence relation

The functions descriped in this section operate on matrix representations. There is no special data type for matrix representations, but all functions expect that the representation is given as an array of n pointers to square matrices representing the generators of the algebra.

Spin-up and standard basis The spin-up algorith calculates, for a given 'seed' vector v the submodule generated by v, i.e., the smallest subspace which is invariant under the generators and contains v.

Normally, the basis vectors computed during the spin-up process are chosen randomly. However, the spin-up algorithm can run in 'canonical' or 'standard basis' mode. In this mode the result is invariant under a change of basis. More precisely, if v (seed vector) and $g_1, \ldots, g_n$ (generators) produce the basis $(b_1, \ldots, b_m)$, and A is any nonsingular matrix, then vA and $A^{-1}g_1A, \ldots, A^{-1}g_nA$ produce the basis $(b_1A, \ldots, b_mA)$.

Splitting a representation If a proper invariant subspace $U < V$ has been found for a matrix representation M, the restriction of M to U as well as the representation on V/U can be calculated. This is called splitting the representation. Note that the representation on V/U depends on a randomly chosen basis.

3.4 Bit strings

`bitstring_t` .... Bit string data type
`bs_setlen()` .... Set the bit string length
`bs_alloc()` .... Create a bit string
`bs_dup()` .... Duplicate a bit string
`bs_free()` .... Free a bit string
`bs_reset()` .... Clear a bit string
`bs_cmp()` .... Compare two bit strings
`bs_cpy()` .... Copy a bit string
`bs_set()` .... Set one bit in a bit string
`bs_clear()` .... Clear one bit in a bit string
`bs_test()` .... Test a bit in a bit string
`bs_read()` .... Read a bit string from a file
`bs_write()` .... Write a bit string to a file
`bs_and()` .... Intersection of two bit strings
`bs_or()` .... Union of two bit strings
`bs_minus()` .... Bit string difference
`bs_match()` .... Intersection count
`bs_issub()` .... Bit string incidence relation

The MeatAxe has two data types that represent subsets of the natural numbers, `set_t` and `bitstring_t`.

The `bitstring_t` type represents a string of 0's and 1's. You may also think of it as a subset of $\{1, 2, \ldots, N\}$, where each 1 in the bit string means that the subset contains the corresponding element. Unlike the `set_t` type bit strings are static objects which cannot change size. However, operations like the intersection of two sets are much faster with `bitstring_t` than with `set_t` objects.

The length of a bit strings is a global variable which can be set with `bs_setlen()`. You must call `bs_setlen()` at least once before creating a bit string. Since most bit string function assume that all bit strings have the same length, it is difficult to operate with bit strings of different sizes.

3.5 Sets of integers

`set_t` .... Integer set data structure
`SetPrint()` .... Write a set to stdout
`SetAlloc()` .... Create a new set
`SetFree()` .... Destroy a set
`SetDup()` .... Duplicate a set
`SetInsert()` .... Insert an element into a set

`SetContains()` ..... Check if a number is in a set
`SetRead()` .... Read a set from a file
`SetWrite()` .... Write a set to a file

The MeatAxe has two data types that represent subsets of the natural numbers, `set_t` and `bitstring_t`.

3.6 Poynomials

`poly_t` .... Polynomial data structure

3.6.1 Basic polynomial functions

`PolAlloc()` .... Allocate a polynomial
`PolFree()` .... Free a polynomial
`PolDup()` .... Duplicate a polynomial
`PolPrint()` .... Print a polynomial

3.6.2 Polynomial i/o functions

`PolRead()` .... Read a polynomial from a file
`PolWrite()` .... Write a polynomial to a file
`PolLoad()` .... Read a polynomial
`PolSave()` .... Write a polynomial

3.6.3 Polynomial operations

`PolFactorization()()`Factor a polynomial
`PolGcd()` .... G.c.d. of polynomials
`PolGcdEx()` .... G.c.d. of polynomials
`PolAdd()` .... Add polynomials
`PolMul()` .... Multiply polynomials
`PolDivMod()` .... Polynomial division
`PolMod()` .... Polynomial division
`PolCompare()` ...... Compare polynomials
`PolDerive()` .... Derive a polynomials

3.6.4 Polynomial factorization

`FpolMulP()` .... Multiply with an irreducible polynomial
`FpolMul()` .... Multiply factored polynomials
`FpolPrint()` .... Print a factored polynomial
`FpolAlloc()` .... Allocate a factored polynomial
`FpolFree()` .... Free a factored polynomial
`FpolDup()` .... Duplicate a factored polynomial

3.7 Permutations

`perm_t` .... Permutation data structure

Permutations operate on the set $\{1, 2, \ldots, n\}$, where n is the degree. A permutation is represented by the `perm_t` data structure. This structure also carries the degree of the permutation. This is important because you cannot, for example, multiply permutations of different degrees.

3.7.1 Basic permutation functions

`PermPrint()` .... Print a permutation
`PermAlloc()` .... Allocate a permutation
`PermFree()` .... Free a permutation
`PermDup()` .... Duplicate a permutation

3.7.2 Operations with permutations

`PermMul()` .... Multiply permutations
`PermOrder()` .... Order of a permutation
`PermPower()` .... Power of a permutation

3.7.3 File i/o for permutations

`PermRead()` .... Read a permutation
`PermWrite()` .... Write a permutation
`PermLoad()` .... Read a permutation
`PermSave()` .... Write a permutation

3.8 MeatAxe objects

3.9 General functions

3.9.1 Initialization

`MtxInit()` .... Initialize the library
`MtxCleanup()` ...... Terminate the library

3.9.2 Command line parser

`proginfo_t` .... Program information structure
`MtxSetCommandLine()` Initialize the command line parser
`MtxGetOpt()` .... Get next option
`MtxGetInt()` .... Get integer argument

The MeatAxe library provides a minimal framework for applications. All MeatAxe applications should use this framework to achieve a consistent behaviour. The functions described above enable an application to process options and arguments on the command line. They implement also a rudimentary on-line help and automatic processing of some common options like `-V`. Here is an example:

```
static char *helptext[] = {
  "SYNTAX",
  "    dummy [-ab] [-x <First>-<Last>] <File> ...",
  "",
  "DESCRIPTION",
  "    This program does nothing.",
  NULL};
static proginfo_t pinfo =
  { "dummy", "Do Nothing", "$Revision 1.16$", helptext };

int main(int argc, char *argv[])
{
    init_args(argc, argv, &pinfo);
```

```
    while (MtxGetOpt("abx:") != OPT_END)
    {   switch (opt_char)
        {
            case 'a': ignore_a(); break;
            case 'b': ignore_b(); break;
            case 'x':
                first = MtxGetInt();
                if (*opt_text_ptr++ != '-') error();
                last = MtxGetInt();
                break;
        }
    }
    ...
}
```

In this example, the `-x` option takes an argument of the form *first–last*. After the first `MtxGetInt()` `opt_text_ptr` points to the '-' and must be incremented for the next `MtxGetInt()` to work.

3.9.3 Program messages

`MtxFormatMessage()`Format a message
`MtxPrintMessage()` Print a message

3.9.4 Error handling and debugging

`MtxSetErrorHandler()`Define an application error handler
`MtxError()` .... Signal an error
`MtxErrorExit()` ... Exit program with error message
`MtxErrorRecord` ... Error data structure
`MtxErrorHandler` .. Error handler function type

Errors can occur inside the MeatAxe library for several reasons including

- insufficient system resources (memory, disk space),
- device or operating system failure,
- invalid parameters passed to a MeatAxe library function,
- invalid usage of MeatAxe library functions, for example missing initialization.

Note that invalid parameters are not always detected by the MeatAxe library. For example, most kernel functions such as `zadd()` do not check their arguments for the sake of performance. Thus, calling these function with invalid arguments may produce random results or even crash the program. On the other hand, most higher-level functions like `MatAdd()` do some plausibility checks on their arguments before processing them.

When an error is detected, the default action is to terminate the program immediately with an appropriate error message. While this minimizes the chance of not noticing an error, it may be undesirable in some situations. For example, the application may be able to recover from the error. In order to prevent the MeatAxe library from terminating the program, the application can define an application error handler. This is a function which is called on each error.

3.10 The word generator

`WgInit()` .... Initialize the word generator
`WgFree()` .... Terminate the word generator
`WgMakeWord()` ...... Make a word
`WgMakeFingerPrint()` Calculate finger print
`WgSymbolicName()` . Symbolic name of a word

3.11 The lattice package

`CFInfo` .... Constituent data structure
`Lat_Info` .... Lattice data structure

3.12 Text file input/output

`StfClose()` .... Close a structured text file
`StfOpen()` .... Open a structured text file
`StfReadLine()` ..... Read next line
`StfGetName()` ...... Get file name
`StfGetInt()` .... Read an integer
`StfMatch()` .... Skip text
`StfGetVector()` ... Read a vector
`StfBeginEntry()` .. Begin entry
`StfEndEntry()` ..... End entry
`StfWriteValue()` .. Write a text
`StfWriteInt()` ..... Write an integer number
`StfWriteVector()` . Write a vector

The MeatAxe library provides functions for input and output of data in human-readable text format. Files that are created with this set of functions have a defined structure, and are refered to as 'structured text files' (STF). An examples for this type of files is the `.cfinfo` file which is maintained by the submodule lattice programs.

File format A structured text file is interpreted as a sequence of lines. While the STF input functions can read very long lines, the output functions try to limit the line length to 80 characters in order make the file more readable. Each line is one of the following:

- Lines starting with a '#' in column 1 are comment lines and are ignored completely. Empty lines are treated as comment lines, too.
- A non-comment line with a non-blank character in column 1 marks the beginning of a new entry. Such a line has the format

  Name `:=` *Value*

  Both *Name* and *Value* are arbitrary strings, except that they cannot contain leading or trailing blanks. In fact leading and trailing blanks as well as any blanks around the `:=` are removed on input.
- Lines starting with a whitspace character are interpreted as continuing lines. Obviously a continuing line may occur only after an entry has started. The contents of the continuing line, after leading lanks have been removed, are appended to *Value*.

Data formats Besides the removal of leading and trailing blanks there is no restriction on the format of *Value* in an STF entry. There are, however, predefined functions that read and write integers and sequences of integers. An application should use these functions where possible. The format used by the integer i/o functions is most easily demonstrated in an example:

```
Field := 7;
Multiplicity := [1,1,1,2];
Dimensions := [11,22,33,44,55];
```

The format has been chosen such that GAP can read the text file without modification.

Chapter 4

Library reference

`bitstring_t` **Bit string data type**

```
typedef struct { ... } bitstring_t;
```

Description The `bitstring_t` type represents a string of 0's and 1'. You may also think of it as a subset of $\{1, 2, \ldots, N\}$, where each 1 in the bit string means that the subset contains the corresponding element. Unlike the `set_t` type (see section 3.5) bit strings are static objects which cannot change size.

`bs_alloc()` **Create a bit string**

```
bitstring_t *bs_alloc();
```

Return........Pointer to the new bit string, or `NULL` on error.

Description This function creates a new bit string. Before yo can use `bs_alloc()` you must set the bit string length with `bs_setlen()`.

`bs_and()` **Intersection of two bit strings**

```
void bs_and(bitstring_t *dest, const bitstring_t *src);
```

`dest`..........Destination bit string.
`src`...........Source bit string, and-ed to `dest`.

Description This function computes the intersection of two bit strings, i.e., the logical 'and'. The result is stored in `dest` and overwrites the previous value.

`bs_clear()` **Clear one bit in a bit string**

```
void bs_clear(bitstring_t *b, int i);
```

`b`.............Pointer to the bit string.
`i`.............Index of bit to clear (starting from 0).

Description This function clears a single bit in a bit string.

See also `bs_set()` `bs_test()`

`bs_cmp()` **Compare two bit strings**

```
int bs_cmp(const bitstring_t *a, const bitstring_t *b);
```

`a` .... Pointer to the first bit string.
`b` .... Pointer to the second bit string.
Return........ 0 if the bit strings are equal. Otherwise the return value is different from zero.

Description This function compares two bit strings and returns zero if they are equal.

`bs_cpy()` **Copy a bit string**

```
void bs_cpy(bitstring_t *dest, const bitstring_t *src);
```

`dest` .... Pointer to the destination bit string.
`src` .... Pointer to the source bit string.

Description This function copy a bit string from `src` to `dest`. The bit strings must have the same size.

`bs_dup()` **Duplicate a bit string**

```
bitstring_t *bs_dup(const bitstring_t *s)
```

`s` .... Pointer to the bit string.
Return........ Pointer to the copy, or `NULL` on error.

Description This function creates an independent copy of an exiting bit tring.

`bs_free()` **Free a bit string**

```
void bs_free(bitstring_t *b);
```

`b` .... Pointer to the bit string.

Description This function frees a bit string, releasing all associated memory.

`bs_issub()` **Bit string incidence relation**

```
int bs_issub(const bitstring_t *a, const bitstring_t *b);
```

`a` .... First bit string.
`b` .... Second bit string.
Return........ 1 if $a \subseteq b$, 0 otherwise.

Description This function checks, if one bit string is contained in a second bit string. By definition, $a \subseteq b$ if and only if every bit which is set in a is also set in b.

`bs_match()` **Intersection count**

```
int bs_match(const bitstring_t *x, const bitstring_t *y);
```

`x` .... First bit string.
`y` .... Second bit string.
Return........0, 1 or 2 (see below).

Description This function compares two bit strings of equal size looking for bits which are set in both strings. The return value is 0 if there are no matching bits, 1 if there is exactly one matching bit, and 2 if there are 2 or more matching bits.

`bs_minus()` **Bit string difference**

```
void bs_minus(bitstring_t *dest, const bitstring_t *src);
```

`dest` .... Destination bit string.
`src` .... Source bit string, subtracted from `dest`.

Description `bs_minus()` computes the (set theoretical) difference of two bit strings, i.e., each bit in the result is set if and only if it is set in `dest` and cleared in `src`. `src` and `dest` must have have equal size.

The result is stored in `dest` and overwrites the previous value.

`bs_or()` **Union of two bit strings**

```
void bs_or(bitstring_t *dest, const bitstring_t *src);
```

`dest` .... Destination bit string.
`src` .... Source bit string, or-ed to `dest`.

Description This function computes the union of two bit strings. The result is stored in `dest` and overwrites the previous value.

`bs_read()` **Read a bit string from a file**

```
bitstring_t *bs_read(FILE *f);
```

`f` .... File to read from.
Return........Pointer to the bit string, or `NULL` on error.

Description This function reads a bit string from a file. The bit string size must have been set before with `bs_setlen()`. The return value is a pointer to the bit string or `NULL` on error.

See also `bs_read()`

bs_reset() — Clear a bit string

```
void bs_reset(bitstring_t *x);
```

x .... Pointer to the bit string.

Description This function clears all bits in a bit string.

bs_set() — Set one bit in a bit string

```
void bs_set(bitstring_t *b, int i);
```

b .... Pointer to the bit string.
i .... Index of bit to set (starting from 0).

Description This function sets a single bit in a bit string.

See also bs_clear() bs_test()

bs_setlen() — Set the bit string length

```
void bs_setlen(int l);
```

l .... Bit string length.

Description This function sets the bit string length which is used by most bit string functions.

bs_test() — Test a bit in a bit string

```
int bs_test(const bitstring_t *b, int i);
```

b .... Pointer to the bit string.
i .... Index of bit to test (starting from 0).
Return........1 if the bit is set, 0 otherwise.

Description This function tests a single bit in a bit string. The return value is 1 id the bit is set and 0 otherwise.

See also bs_clear() bs_set()

bs_write() — Write a bit string to a file

```
int bs_write(FILE *f, const bitstring_t *b);
```

f .... File to write to.
b .... Bit string to write.
Return........0 on success, −1 on error.

Description This function writes a bit string to a file. The bit string size must have been set previosly with `bs_setlen()`.

See also `bs_read()`

`CFInfo` **Constituent data structure**

```
typedef struct
{
    long dim, num, mult;
    long idword;
    poly_t *idpol;
    long peakword;
    poly_t *peakpol;
    long nmount;
    long ndotl;
    long spl;
}
CfInfo;
```

Description The `CFInfo` data structure contains all information about one irreducible constituent of a module. Most of this information is useful only in the context of a `Lat_Info` structure.

`dim` is the dimension of the constituent, and `num` enumerates constituents of same dimension. `mult` ist the multiplicity of this constituent in the module. `idword` and `idpol` specify the identifying word for this module. The identifying word is used by the CHOP program.

`ChangeBasis()` **Change basis**

```
int ChangeBasis(const matrix_t *M, int ngen, const matrix_t *gen[],
    matrix_t *newgen[])
```

`M` .... New basis.
`ngen` .... Number of matrices in `gen`.
`gen` .... Array of matrices to transform.
`newgen` .... Pointer to a buffer for the transformed generators.
Return .... The function returns `dest` or `NULL` on error.

Description This function performs a change of basis on a set of square matrices. Typically, the matrices are the generators for an algebra with respect to a basis B. `ChangeBasis()` calculates the matrix representation of the generators with respect to another basis, C. The rows of `M` must contain the new basis vectors, expressed in the old basis. Then, the new generators are given by

$$g_i' = M g_i M^{-1}$$

If `newgen` is different from `gen`, `newgen` must point to an array of at least `ngen` elements of type `matrix_t *`. The elements of `newgen` must not be in use, because they are overwritten by `ChangeBasis()` with newly allocated matrices. If `newgen` is euqal to `gen`, the new generators replace the original generators. In this case, `ChangeBasis()` performs the necessary clean-up before overwriting the generators.

`CharPol()` **Characteristic polynomial**

```
fpoly_t *CharPol(const matrix_t *mat);
```

`mat` .... Pointer to the matrix.
Return........ The characteristic polynomial of `mat` or `NULL` on error.

Description This function calculates the characteristic polynomial of a matrix in factored form. The return value is a pointer to a `fpoly_t` structure containing the irreducible factors of the characteristic polynomial.

See also `CharPolFactor()` `CharPolSeed`

`CharPolFactor()` **Characteristic polynomial**

```
poly_t *CharPolFactor(const matrix_t *mat);
```

`mat` .... Pointer to the matrix.
Return........ One factor of the characteristic polynomial of `mat`, or `NULL` if there are no more factors.

Description This function returns one factor of the characteristic polynomial of a given matrix. Further calls with a `NULL` argument return more factors or `NULL`, if there are no more factors. Note that the factors obtained in this way are in general not irreducible.

Here is how `CharPolFactor()` works: If `mat` is different from `NULL`, `CharPolFactor()` initializes its internal data and starts computing one cyclic subspace. The choice of starting vector for this first subspace depends on the global variable `CharPolSeed`. Usually, this variable has a value of 1, corresponding to the vector $(1, 0, \ldots, 0)$. Then, the polynomial of the matrix restricted to that cyclic subspace is constructed and returned to the caller.

If `mat` is `NULL` on the next call, `CharPolFactor()` resumes at the point where it returned the last time, calculates the next cyclic subspace and so on, until the complete space is exhausted.

Note: Since the function uses static variables to store information across multiple calls, your program must not use `CharPolFactor()` on more than one matrix at the same time.

See also `CharPol()` `CharPolSeed`

`CharPolSeed` **Seed for characteristic polynomial**

```
long CharPolSeed = 1;
```

Description This variable is used by `CharPolFactor()` to select the first seed vector. By default, `CharPolSeed` has the value 1, i.e., the first seed vector is $(1, 0, \ldots, 0)$. Assigning the value 2 selects the start vector $(0, 1, \ldots, 0)$ in all subsequent calls to `CharPolFactor()`.

If `CharPolSeed` is out of bounds, `CharPolFactor()` will reset it to 1.

See also `CharPol()` `CharPolFactor()`

`FEL` **Finite field element**

```
typedef unsigned char FEL;
```

Description The `FEL` data type represents an element of the finite field. The actual definition of `FEL` depends on the kernel that was used in compiling the MeatAxe. The 'standard' kernel for fields up to GF(256) defines a `FEL` as an unsigned character (which is assumed to be 8 bits wide).

`FpolAlloc()` **Allocate a factored polynomial**

```
fpoly_t *FpolAlloc();
```

Return........Pointer to a newly allocated `fpoly_t` structure, or `NULL` on error.

Description This function allocates and initializes a factored polynomial (`fpoly_t`) data structure.

See also `FpolDup()` `FpolFree()`

`FpolDup()` **Duplicate a factored polynomial**

```
fpoly_t *FpolDup(const fpoly_t *s);
```

`s` .... Pointer to a factored polynomial.
Return........The function returns a pointer to a copy of `s`, or `NULL` on error.

Description This function creates a copy of a factored polynomial.

See also `FpolAlloc()` `FpolFree()`

`FpolFree()` **Free a factored polynomial**

```
void FpolFree(fpoly_t *x);
```

`x` .... Pointer to the factored polynomial.

Description This function frees a factored polynomial (`fpoly_t`) data structure, including any internal data structures.

See also `FpolAlloc()` `FpolDup()`

`FpolMul()` **Multiply factored polynomials**

```
fpoly_t *FpolMul(fpoly_t *dest, const fpoly_t *src);
```

`dest`..........Factored polynomial to modify.
`src`...........Factored polynomial.
Return........The function returns `dest` or `NULL` on error.

Description Multiplies the factored polynomial `dest` by the factored polynomial `src`.

See also `FpolMulP()`

`FpolMulP()` — Multiply with an irreducible polynomial

```
fpoly_t *FpolMulP(fpoly_t *dest, const poly_t *src, long pwr);
```

`dest` .... Factored polynomial to modify.
`src` .... Irreducible polynomial.
`pwr` .... Power of the irreducible polynomial.
Return........ The function returns `dest` or `NULL` on error.

Description This function multiplies a factored polynomial with the power of an an irreducible factor.

See also `FpolMul()`

`FpolPrint()` — Print a factored polynomial

```
void FpolPrint(const char *name, const fpoly_t *p);
```

`name` .... Name of the polynomial, or `NULL`.
`p` .... Pointer to the factored polynomial.

Description This function prints a factored polynomial in human-readable format on the standard output. If `name` is not `NULL`, "name =" is printed before the polynomial.

`gcd()` — Greatest common divisor

```
long gcd(long a,long b);
```

`a` .... First number.
`b` .... Second number.
Return........ Greatest common divisor of `a` and `b`.

Description This function returns the greatest common divisor of two integers. If both numbers are positive, the result is positive. If one or both numbers are negative, the result may be positive or negative. If both numbers are zero, the result is zero.

See also `lcm()`

`IsIsomorphic()` — Compare representations

```
int IsIsomorphic(int ngen, const matrix_t *gen1[], const CfInfo *info1,
    const matrix_t *gen2[], matrix_t **trans, int use_pw)
```

`ngen` .... Number of generators.
`gen1` .... Generators for the first representation.
`info1` .... Pointer to an `CfInfo` structure for the first representation.
`gen2` .... Generators for the second representation.
`trans` .... Buffer for basis transformation matrix, or `NULL`.
`use_pw` .... If different from zero, use peak word instead of the identifying word (see below).
Return........ 1 if the representations are isomorphic, 0 if they are not isomorphic, and -1 on error.

Description This function decides if two irreducible representations are isomorphic. To compare the representations, the function needs an identifying word for the first representation, i.e., the fields `info1->idword`, `info1->idpol` and `info1->spl` must be set, and the generators in `gen1` must be in standard basis with respect to the identifying word. If `use_pw` is nonzero, the peak word is used instead of the idword. In this case, `gen1` must of course be in standard basis with respect to the peak word.

If the representations are isomorphic, and `trans` is not `NULL`, the basis transformation which makes the second representation identical to the first is stored into `*trans`. To be more precise, if g_i is the matrix in `gen1[i]`, h_i the matrix in `gen2[i]`, and T the matrix returned in `trans`, then $Th_iT^{-1} = g_i$.

`IsSubspace()` **Vector space incidence relation**

```
int IsSubspace(const matrix_t *u, const matrix_t *v, long ngen);
```

`u` .... Generating set for the first space, U.
`v` .... Pointer to a basis for the second space, V.
`ngen` .... Number of vectors in `sub` to consider.
Return........ The function returns 1 if $U \leq V$, 0 if $U \not\leq V$, and -1 on error.

Description This function compares two vector spaces $U, V \in \mathrm{GF}(q)^n$ and returns 1, if $U \leq V$. The spaces to compare are given as matrices where the rows of `u` (linearly) generate U and the rows of `v` generate V. Thus, `u` and `v` must be matrices over the same field and with the same number of columns. Also, `v` must be in full echelon form while there is no further restriction on `u`.

In normal mode, `ngen` is 0. Then, the rows of `u` are checked one-by-one to see if they are in the span of the rows of `v`. If this test passes for each row of `u`, the return value is 1, otherwise it is 0. There is no special return value for the cases $U = V$ or $U > V$.

If `ngen` is different from zero, the function assumes that U is generated by the first `ngen` rows, and only this number of rows are checked. For example, U may be a cyclic submodule, which is generated from a single vector under the action of an algebra. Then, one only needs to check if the generating vector, which is supposed to be the first row of `u`, is contained in V.

`Lat_Info` **Lattice data structure**

```
typedef struct
{
    char BaseName[LAT_MAXBASENAME];
    int Field;
    int NGen;
    int NCf;
    CfInfo Cf[LAT_MAXCF];
} Lat_Info;
```

Description The `Lat_Info` data structure contains all information about a module, which is gathered during the submodule lattice calculation.

`lcm()` **Least common multiple**

```
long lcm(long a,long b);
```

`a` .... First number.
`b` .... Second number.
Return........ Least common nultiple of `a` and `b`.

Description This function returns the least common multiple of two integers. If both numbers are positive, the result is positive. If one or both numbers are negative, the result may be positive or negative. If both numbers are zero, a division by zero occurs.

See also `gcd()`

`MakeEndomoprhisms()` **Find endomorphism ring**

```
int MakeEndomorphisms(int ngen, const matrix_t *gen[], const matrix_t *nsp,
    matrix_t *endo[])
```

`ngen` .... Number of generators in `gen`.
`gen` .... List of generators.
`nsp` .... Idword kernel (see below).
`endo` .... Pointer to result buffer.
Return .... 0 on success, −1 on error.

Description This function calculates a basis for the endomorphism ring E of an irreducible module. The dimension of E is equal to the degree $[F' : F]$ of the splitting field extension for the module.

The function takes three input arguments: `ngen` is the number of generators of the algebra, and `gen` points to an array of `ngen` matrices containing the representations of the generators. `nsp` is the kernel of an idword (identifying word) for the module, i.e., an algebra element a with $\dim(\ker(a_V)) = [F` : F]$.

On success, basis of E (a set of matrices) is stored into `endo`. If the function fails, no matrices are stored in `endo`.

`MatAdd()` **Add a matrix**

```
matrix_t *MatAdd(matrix_t *dest, const matrix_t *src);
```

`dest` .... Matrix to add to.
`src` .... Matrix to add.
Return .... The function returns `dest`, or `NULL` on error.

Description This function adds `src` to `dest`. The matrices must be compatible, i.e., they must be over the same field and they muast have the same size.

See also `MatAddMul()` `MatMul()` `MatMulScalar()`

`MatAddMul()` **Add a multiple of a matrix**

```
matrix_t *MatAddMul(matrix_t *dest, const matrix_t *src, FEL coeff);
```

`dest` .... Matrix to add to.
`src` .... Matrix to add.
`coeff` .... Coefficient.
Return .... The function returns `dest`, or `NULL` on error.

Description This function adds a multiple of a matrix to another matrix. The matrices must be compatible for addition. `MatAddMul()` handles special cases (`coeff` equals 0 or 1) in an intelligent way, so there is no need for the caller to do this.

See also `MatAdd()` `MatMul()` `MatMulScalar()`

`MatAlloc()` **Create a new matrix**

```
matrix_t *MatAlloc(long fl, long nor, long noc);
```

`fl` .... Field order.
`nor` .... Number of rows.
`noc` .... Number of columns.
Return........ Pointer to the new matrix or `NULL` on error.

Description This function creates a new matrix with `nor` rows and `noc` columns over GF(`fl`). The return value is a pointer to the allocated `matrix_t` structure, or `NULL` if there is not enough memory.
To destroy a matrix, use `MatFree()`, *not* `free()`.

See also `MatDup()` `MatFree()` `MatId()`

`MatCopyRegion()` **Copy a rectanglular region of a matrix**

```
int MatCopyRegion(matrix_t *dest, int destrow, int destcol,
    const matrix_t *src, int row1, int col1, int nrows, int ncols)
```

`dest` .... Pointer to the destination matrix.
`destrow` ...... Destination row.
`destcol` ...... Destination column.
`src` .... Pointer to the source matrix.
`row1` .... First row in region.
`col1` .... First column in region.
`nrows` .... Number of rows to cut. -1 means as many rows as possible.
`ncols` .... Number of columns to cut. -1 means as many columns as possible.
Return........ Returns 0 on success, or -1 on error.

Description This function copies a rectangular region of a matrix into another matrix. `row1` and `col1` define the upper left corner of the region. `nrows` and `ncols` are the height (number of rows) and width (number of columns) of the region. Both `nrows` and `ncols` can be -1. In this case the region extends up to the last row or last column, respectivly. `destrow` and `destcol` specify where the rectangle is placed inside `dest`. As uusual, all row and column indexes start with 1, not 0.
`src` and `dest` muts be matrices over the same field. Both source and destination rectangle must not exceed the size of the matrix. In particular, it is not possible to extend `dest`'s size with this function.

See also `MatCut()`

`MatCut()` **Cut a rectangle out of a matrix**

```
matrix_t *MatCut(const matrix_t *src, int row1, int col1, int nrows,
    int ncols);
```

`src` .... Pointer to the matrix.
`row1` .... First row in region.
`col1` .... First column in region.
`nrows` .... Number of rows to cut. −1 means as many rows as possible.
`ncols` .... Number of columns to cut. −1 means as many columns as possible.
Return .... Pointer to a new matrix containing the specified region of `src`, or `NULL` on error.

Description This function cuts a rectangular region out of a matrix. `row1` and `col1` define the upper left corner of the region. As usual, indexes start with 1, not 0. `nrows` and `ncols` are the height (number of rows) and width (number of columns) of the region. Both `nrows` and `ncols` can be −1. In this case the region extends up to the last row or last column, respectivly. For example, to extract the first 10 rows from a matrix independently of the number of columns, you could say `MatCut(mat,1,1,10,-1)`. The rectangle defined by `row1`, `col1`, `nrows` and `ncols` must not exceed the size of the matrix.

The function returns a new matrix containing a copy of the specified rectangle. In case of error, the return value is `NULL`.

See also `MatCopyRegion()`

`MatDup()` **Duplicate a matrix**

```
matrix_t *MatDup(const matrix_t *src);
```

`src` .... Pointer to the matrix.
Return .... Pointer to a copy of `mat`, or `NULL` on error.

Description This function creates a copy of an existing matrix.

See also `MatAlloc()` `MatFree()` `MatId()`

`MatEchelon()` **Echelon form of a matrix**

```
matrix_t *MatEchelon(const matrix_t *mat);
```

`mat` .... Pointer to the matrix.
Return .... Echelon form of `mat`, or `NULL` on error.

Description This function calculates the echelon form of the matrix `mat`. Unlike `MatEchelonize()`, this function does not modify the original matrix. The result is a newly allocated matrix which must be freed by the caller.

See also `MatEchelonize()` `MatIsEchelon()`

`MatEchelonize()` — Reduce to echelon form

```
matrix_t *MatEchelonize(matrix_t *mat);
```

`mat` .... Pointer to the matrix.
Return........ The function returns `mat`, or `NULL` on error.

Description This function performs a Gaussian elimination on the matrix `mat`. On return, `mat` is in echelon form. If the rank of `mat` was smaller than the number of rows, some rows are removed during the process.

Note that this function modifies the original matrix. If you need a non-destructive Gaussian elimination use `MatEchelon()`.

See also `MatEchelon()` `MatIsEchelon()`

`MatFree()` — Delete a matrix

```
void MatFree(matrix_t *m);
```

`m` .... Pointer to the matrix.
Return........ 0 on success, -1 on error.

Description This function frees a matrix which has beed created by `MatAlloc()`. This implies freeing the internal data buffer as well as the `matrix_t` structure itself.

See also `MatAlloc()` `MatDup()` `MatId()`

`MatId()` — Identity matrix

```
matrix_t *MatId(long fl, long nor);
```

`fl` .... Field order.
`nor` .... Number of rows.
Return........ Pointer to the matrix, or `NULL` on error.

Description This function creates an identity matrix with `nor` nows over GF(`fl`).

See also `MatAlloc()` `MatDup()` `MatFree()`

`MatInsert()` — Insert a matrix into a polynomial

```
matrix_t *MatInsert(const matrix_t *mat, const poly_t *pol);
```

`mat` .... Pointer to the matrix.
`pol` .... Pointer to the polynomial.
Return........ The function returns `mat`, or `NULL` on error.

Description This function inserts a matrix into a polynomial. Unlike `MatInsert_()`, this function is not destructive. It allocates a new matrix for the result and does not modify the original matrix.

`mat` must be a square matrix, and `pol` must be over the same field as `mat`.

See also `MatInsert_()`

`MatInsert_()` **Insert a matrix into a polynomial**

```
matrix_t *MatInsert_(matrix_t *mat, const poly_t *pol);
```

`mat` .... Pointer to the matrix.
`pol` .... Pointer to the polynomial.
Return........ The function returns `mat`, or `NULL` on error.

Description This function inserts a matrix into a polynomial. This function is destructive. It uses `mat` for the result of the insertion, and does not allocate a new matrix for the result.

`mat` must be a square matrix, and `pol` must be over the same field as `mat`.

See also `MatInsert()`

`MatInv()` **Invert a matrix**

```
matrix_t *MatInv(const matrix_t *mat);
```

`mat` .... Pointer to the matrix.
Return........ Inverse matrix or `NULL` on error.

Description This function calculates the inverse of a matrix. `mat` must be a non-singular square matrix. The inverse matrix is returned in a newly allocated `matrix_t` structure, and the original matrix remains unchanged.

See also `zmatinv()`

`MatIsEchelon()` **Check if a matrix is in echelon form**

```
int MatIsEchelon(const matrix_t *mat);
```

`mat` .... Pointer to the matrix.
Return........ 1 if the matrix is in echelon form, 0 if not, -1 on error.

Description This function checks if a given matrix is in (full) echelon form. A matrix is in full echelon form, if each row contains a 1 as first non-zero element, and all the other rows have a 0 at this position.

See also `MatEchelon()` `MatEchelonize()`

`MatLoad()` **Read a matrix from a file**

```
matrix_t *MatLoad(const char *fn);
```

`fn` .... File name.
Return........ Pointer to the matrix, or `NULL` on error.

Description This function opens a file, reads a single matrix, and closes the file.
To read more than one matrix from a file, use `MatRead()`.

See also `MatRead()`

`MatMul()` **Multiply matrices**

```
matrix_t *MatMul(matrix_t *dest, const matrix_t *src);
```

`dest` .... Left factor.
`src` .... Right factor.
Return........ The function returns `dest`, or `NULL` on error.

Description This function multiplies the matrix `dest` from the right by the matrix `src`. The matrices must be compatible for multiplication, i.e. they must be over the same field, and the number of columns of `dest` must be equal to the number f rows of `src`. The result of the multiplication is stored in `dest`, overwriting the original contents.

See also `MatPower()`

`MatMulScalar()` **Calculate the multiple of a matrix**

```
matrix_t *MatMulScalar(matrix_t *dest, FEL coeff);
```

`dest` .... Pointer to the matrix.
`coeff` .... Value to multiply with.
Return........ The function returns `dest`.

Description This function multiplies a matrix by a constant value.

See also `MatAdd()` `MatAddMul()` `MatMul()`

`MatNullity()` **Nullity of a matrix**

```
long MatNullity(const matrix_t *mat);
```

`mat` .... Pointer to the matrix.
Return........ Nullity of `mat`, or -1 on error.

Description This function calculates the dimension of the null-space of a matrix. Unline `MatNullity__()`, this function does not modify the matrix.

See also `MatNullity__()` `MatNullspace()` `MatNullSpace_()` `MatNullSpace__()`

MatNullity__() **Nullity of a matrix**

`long MatNullity__(matrix_t *mat);`

`mat` .... Pointer to the matrix.
Return........ Nullity of `mat`, or −1 on error.

Description This function calculates the dimension of the null-space of a matrix and deletes the matrix.

See also `MatNullity() MatNullspace() MatNullSpace_() MatNullSpace__()`

MatNullspace() **Null-space of a matrix**

`matrix_t *MatNullspace(const matrix_t *mat);`

`mat` .... Pointer to the matrix.
Return........ Pointer to the null-space of `mat`, or `NULL` on error.

Description This function calculates the null-space of a matrix. Unlike `MatNullSpace_()`, `MatNullspace()` does not change the original matrix, but it temporarily allocates a copy of the matrix.

See also `MatNullity() MatNullity__() MatNullSpace_() MatNullSpace__()`

MatNullSpace_() **Null-space of a matrix**

`matrix_t *MatNullSpace_(matrix_t *mat);`

`mat` .... Pointer to the matrix.
Return........ Pointer to the null-space of `mat`, or `NULL` on error.

Description This function reduces a matrix to echelon form and returns its null-space. Unlike `MatNullspace()`, `MatNullSpace_()` modifies the orginal matrix, but uses less memory since no temporary workspace is allocated.

See also `MatNullity() MatNullity__() MatNullspace() MatNullSpace__()`

MatNullSpace__() **Null-space of a matrix**

`matrix_t *MatNullSpace__(matrix_t *mat);`

`mat` .... Pointer to the matrix.
Return........ Pointer to the null-space of `mat`, or `NULL` on error.

Description This function calculates the null-space of a matrix and deletes the matrix.

See also `MatNullity() MatNullity__() MatNullspace() MatNullSpace_()`

`MatOrder()` **Order of a matrix**

```
long MatOrder(const matrix_t *mat);
```

`mat` .... Pointer to the matrix.
Return........ The order of `mat`, or -1 on error.

Description This function calculates the order of a matrix. `mat` must be a non-singular, square matrix.

Even if `mat` is non-singular, the function may fail. This happens if the order is greater than 1000000, or if the order on any cyclic subspace is grater than 1000.

`MatPower()` **Power of a matrix**

```
matrix_t *MatPower(const matrix_t *mat, long n);
```

`mat` .... Pointer to the matrix.
`n` .... Exponent.
Return........ `n`-th power of `mat`, or `NULL` on error.

Description This function calculates the n-th power of a matrix, using the binary method. This method is generally faster than multiplying the matrix n times by itself. On the other hand, a third matrix is temporarily created in addition to the original matrix and the result matrix. The cases $n = 0$ and $n = 1$ are handled separately, avoiding unnecessary memory allocation and calculation.

Negative exponents are not allowed. To calculate a negative power, you must first invert the matrix with `MatInv()` and then call `MatPower()` with the inverted matrix and a positive exponent.

See also `MatMul()` `MatInv()`

`MatPrint()` **Print a matrix on stdout**

```
void MatPrint(char *name, const matrix_t *m);
```

`name` .... Name to print before the matrix, or `NULL`.
`m` .... Pointer to the matrix.

Description This function prints a matrix on the standard output in a human-readable format. If `name` is not `NULL` the name followed by an equal sign is printed before the matrix.

`MatRead()` **Read a matrix from a file**

```
matrix_t *MatRead(FILE *f);
```

`f` .... File to read from.
Return........ Pointer to the matrix, or `NULL` on error.

Description This function opens reads a matrix from a file.

See also `MatLoad()`

matrix_t — **Matrix data structure**

```
typedef struct
{
    long fl, nor, noc;
    PTR d;
} matrix_t;
```

Description The `matrix_t` structure represents a matrix ofer a finite field. `fl` is the field order, `nor` contains the number of rows, and `noc` the number of columns. `d` is a pointer to the matrix elements, organized as an array of rows of length `noc`.

Both `nor` and `noc` may be zero. In this case, `d` ist still a valid pointer, but the memory block it points to may have size zero.

MatSave() — **Write a matrix to a file**

```
int MatSave(const matrix_t *mat, const char *fn);
```

`mat` .... Pointer to the matrix.
`fn` .... File name.
Return........ The function returns 0 on success and -1 on error.

Description This function opens a file, writes a matrix to the file, and closes the file. If a file with the specified name already exists, the old contents of the file are destroyd.

To write more than one matrix to a file, use `MatWrite()`.

See also `MatWrite()`

MatTensor() — **Tensor product**

```
matrix_t *MatTensor(const matrix_t *m1, const matrix_t *m2);
```

`m1` .... Pointer to the first matrix.
`m2` .... Pointer to the second matrix.
Return........ The tensor product of `m1` and `m2`, or `NULL` on error.

Description This function calculates the (Kronecker) tensor product $|m1| \otimes |m2|$. Both matrices must be over the same field.

MatTr() — **Transpose a matrix**

```
matrix_t *MatTr(const matrix_t *mat);
```

`mst` .... Pointer to the matrix.
Return........ Pointer to the transposed matrix or `NULL` on error.

Description This function transposes a matrix. A new matrix is allocated for the result. The original matrix remains unchanged.

`MatTrace()` **Trace of a matrix**

`FEL MatTrace(const matrix_t *mat);`

`mat` .... Pointer to the matrix.
Return........ The function returns the trace of `mat`. In case of error, the return value is `F_ZERO`.

Description This function calculates the sum of all diagonal elements of a matrix. Note that the matrix need not be square.

`MatWrite()` **Write a matrix to a file**

`int MatWrite(FILE *f, const matrix_t *mat);`

`f` .... Pointer to the file.
`mat` .... Pointer to the matrix.
Return........ The function returns 0 on success and −1 on error.

Description This function writes a matrix to a file.

See also `MatSave()`

`MinPol()` **Minimal polynomial**

`fpoly_t *MinPol(matrix_t *mat);`

`mat` .... Pointer to the matrix.
Return........ Minimal polynomial of `mat`, or `NULL` on error.

Description This function calculates the minimal polynomial of a matrix in factored form. The return value is a pointer to a `fpoly_t` structure containing the irreducible factors of the minimal polynomial.

See also `MinPolFactor()`

`MinPolFactor()` **Minimal polynomial**

`poly_t *MinPolFactor(matrix_t *mat);`

`mat` .... Pointer to the matrix, or `NULL`.
Return........ One factor of the minmal polynomial of `mat`, or `NULL` if there are no more factors.

Description This function returns one factor of the minimal polynomial of a given matrix. Further calls with a `NULL` argument return more factors or `NULL`, if there are no more factors. Note that the factors obtained in this way are in general not irreducible.

If `mat` is different from `NULL`, `MinPolFactor()` initializes its internal data and starts computing one cyclic subspace. Then, the polynomial of the matrix restricted to that cyclic subspace is constructed and returned to the caller.

If `mat` is `NULL` on the next call, `MinPolFactor()` resumes at the point where it returned the last time, calculates the next cyclic subspace and so on, until the complete space is exhausted.

Note: Since the function uses static variables to store information across multiple calls, your program must not use `MinPolFactor()` on more than one matrix at the same time.

See also `MinPol()`

MtxCleanup() — Terminate the library

```
void MtxCleanup();
```

Description This function terminates the MeatAxe library. Depending on the message level, some information like CPU time is written to stdout.

See also `MtxInit()`

MtxError() — Signal an error

```
void MtxError(const char *file, int line, const char *func,
    const char *msg, ...)
```

`file` .... Name of the source file where the error occured.
`line` .... Line number where the error occured.
`func` .... Name of the Function where the error occured.
`msg` .... Error description.
.... Optional arguments to be inserted into the error description.
Return........ The return value is always 0.

Description This function is mainly used internally by the MeatAxe library to report errors. If an application error handler has been installed, it is executed. Otherwise an error message is written to the error log file and the program is terminmated.

The error message, `msg`, is formatted with `MtxFormatMessage()`.

See also `MtxErrorExit()` `MtxSetErrorHandler()`

MtxErrorExit() — Exit program with error message

```
int MtxErrorExit(const char *msg, ...);
```

`msg` .... Message to print or `NULL`.
.... Optional arguments to be inserted into the message.

Description This function prints an error message to the error log file (`stderr` by default) and terminates the calling process. `msg` is the error message and may contain special format characters. See `MtxFormatMessage()` for a full description. In particular, `%E` takes an integer argument and produces the corresponding MeatAxe error message.

See also `MtxError()` `MtxSetErrorHandler()`

MtxErrorHandler — Error handler function type

```
typedef void MtxErrorHandler(const MtxErrorRecord *err);
```

Description This is the type of an application error handler.

See also `MtxSetErrorHandler()`

`MtxErrorRecord` **Error data structure**

```
typedef struct
{
    const char *FileName;
    int LineNo;
    const char *FunctionName;
    char Text[MTX_MAXERRORTEXT];
} MtxErrorRecord;
```

Description This data structure contains detailed information on an error that occured inside the MeatAxe library.

See also `MtxSetErrorHandler()`

`MtxFormatMessage()` **Format a message**

```
int MtxFormatMessage(char *buf, int bufsize, const char *msg, va_list al);
```

`buf` .... Buffer for the message.
`bufsize` ...... Size of buffer.
`msg` .... The message text.
`al` .... Optional arguments for the message (see below).

Description This function formats a message using a `printf`-like syntax. The following table contains all format codes that may be used within `msg`:

Code	Argument type	Description
`%d`	`int`	Decimal integer
`%s`	`char *`	Text
`%E`	`int`	A MeatAxe error message. The argument must be one of the `MTX_ERR_xxx` constants defined in `meataxe.h`.
`%S`	none	System error message corresponding to the current value of `errno`.

See also `MtxPrintMessage()`

`MtxGetInt()` **Get integer argument**

```
long MtxGetInt();
```

Description This function is to read an integer argument which follows a command line option. `MtxGetInt()` should be used immediately after `MtxGetOpt()`. `opt_text_ptr` must point to a string of decimal digits. `MtxGetInt()` scans this string, returns the integer value, and increments `opt_text_ptr` to the character after the last digit. If the argument is not a number in decimal notation `MtxGetInt()` returns the value `GETINT_ERR`.

See also `MtxGetOpt()` `MtxSetCommandLine()`

`MtxGetOpt()` — Get next option

```
extern char opt_char;
extern char opt_text[];
extern char *opt_text_ptr;
extern int opt_ind;
int MtxGetOpt(const char *pattern);
```

`pattern` ...... List of valid options (see below).
Return........Next option or `OPT_END`.

Description This function gets the next option from the command line. Before using `MtxGetOpt()`, the command line parser must be initialized with `MtxSetCommandLine()`. Then, the application may call `MtxGetOpt()` successively until the return value is `OPT_END`, indicating that there are no more command line options.

`pattern` is a list of valid options. Each option consists of one character, optionally followed by a colon indicating that the option takes an argument. The options `-Q`, `-V`, `-B`, and `-L` are handled automatically. These options need not be specified in `pattern`. In order to process these common options, the application should always call `MtxGetOpt()` even if there are no options specific to the program.

`MtxGetOpt()` returns the option character or the special value `OPT_END` if there are no more options. In addition, some global variables are set on return:

- `opt_char` contains the option character.
- `opt_ind` contains the index of the argument to be read next. For example, if `MtxGetOpt()` return `OPT_END`, `opt_ind` is the index of the first non-optional argument.
- `opt_text`, an array of characters, contains the argument (if present).
- `opt_text_ptr` points to `opt_text[0]`.

This setting enables `MtxGetInt()` to access the argument immediately after `MtxGetOpt()` returns.

See also `MtxGetInt()` `MtxSetCommandLine()`

`MtxInit()` — Initialize the library

```
int MtxInit();
```

Return........MeatAxe version number, or -1 on error.

Description This function initializes the MeatAxe library including finite field arithmetic and file i/o functions. It must be called before any other MeatAxe library function. `MtxInit()` returns a version number which is different for each implementation of the arithmetic. Currently there are two versions, a small one for fields up to GF(256) and a big version for field orders up to 2^{16}.

See also `MtxCleanup()`

`MtxPrintMessage()` — **Print a message**

```
int MtxPrintMessage(FILE *f, const char *fmt, ...);
```

`f` .... File to write to.
`fmt` .... Message to write
`...` .... Optional arguments for the message (see below).

Description This function writes a message to a file using a `printf`-like syntax. See `MtxFormatMessage()` for details.

See also `MtxFormatMessage()`

`MtxRandom()` — **Random number**

```
long MtxRandom(void);
#define MtxRandomInt(max) ((unsigned int) (MtxRandom() % (max)))
```

Return........ Pseudo random number.

Description The function `MtxRandom()` returns a (pseudo) random number. `MtxRandomInt()` returns a (pseudo) random number in the range 0. . . `max`.

See also `MtxRandomInit()`

`MtxRandomInit()` — **Initialize the random number generator**

```
void MtxRandomInit(unsigned x);
```

`x` .... Seed value.

Description This function initializes the random number generator.

See also `MtxRandom()`

`MtxSetCommandLine()` — **Initialize the command line parser**

```
void MtxSetCommandLine(int argc, char *argv[], const proginfo_t *pi);
```

`argc` .... Argument count.
`argv` .... Command line arguments.
`pi` .... Pointer to `proginfo_t` structure with program information.

Description This function initializes the command line parser. This is usually done done at the beginning of the program's `main()` function. `argc` is the number of arguments, and `argv` is an array of pointers to the arguments. Note that `argv[0]` is always the name of the program itself. So, the first argument is in `argv[1]`. `pi` is a pointer to a `proginfo_t` structure containing general information about the program. This data must be set up by the caller.

`MtxSetCommandLine()` performs the following tasks:

1. If the first argument is '-help', the help text contained in `pi` is printed and the program terminates.
2. The `MTXLIB` and `MTXBIN` environment variables are evaluated. If set, their values overwrite the default directory names that were chosen at compile time.
3. The command line parser is initialized. This is required for later calls of `MtxGetOpt()`.

See also `proginfo_t` `MtxGetInt()` `MtxGetOpt()`

`MtxSetErrorHandler()` — Define an application error handler

```
typedef struct { ... } MtxErrorRecord;
typedef void MtxErrorHandler(const MtxErrorRecord *err);
MtxErrorHandler *MtxSetErrorHandler(MtxErrorHandler *h);
```

`h` .... Address of the new error handler or `NULL` to restore the default handler.
Return........Returns the previous error handler.

Description This function defines an application error handler which is called evera time an error occurs inside the MeatAxe library.

See also `MtxError()` `MtxErrorExit()` `MtxErrorRecord` `MtxErrorHandler`

`perm_t` — Permutation data structure

```
typedef struct
{
    long deg;
    long *d;
} perm_t;
```

Description The `perm_t` structure represents a permutation. `deg` is the degree and `d` is a pointer to the permutation proper, which is stored internally as an array of long integers. To be more precise, `d[0]` contains the image of 1 under the permutation, and `d[n-1]` contains the image of n. This is the same sructure which is used by the kernel-level permutation functions.

`PermAlloc()` — Allocate a permutation

```
perm_t *PermAlloc(long deg);
```

`deg` .... Degree.
Return........Pointer to the new permutation, or `NULL` on error.

Description This function creates a permutation of the degree `deg`. It returns a pointer to the new permutation `NULL` on error. The new permutation is initialized to the identity.

See also `PermFree()`

`PermDup()` **Duplicate a permutation**

```
perm_t *PermDup(const perm_t *src);
```

`src` .... Pointer to the permutation.
Return........ Pointer to a copy of `src` or `NULL` on error.

Description This function creates a copy of an existing permutation.

`PermFree()` **Free a permutation**

```
int PermFree(perm_t *p);
```

`p` .... Pointer to the permutation.
Return........ The function returns 0 on success and -1 on error.

Description This function deletes a permutation and returns the memory to the system. Do not use `free()` on permutations because this would only free the `perm_t` structure but not the data buffer.

See also `PermAlloc()`

`PermLoad()` **Read a permutation**

```
perm_t *PermLoad(char *fn);
```

`fn` .... File name.
Return........ Pointer to the permutation read from the file, or `NULL` on error.

Description This function opens a file, reads a single permutation, and closes the file. The return value is a pointer to the permutation or `NULL` on error. If the file contains more than one permutation, only the first one is read.

If a permutation was successfully read, the function returns a pointer to a newly created `perm_t` object. The caller is responsible for deleting this object as soon as the permutation is no longer needed.

See also `PermRead()`

`PermMul()` **Multiply permutations**

```
perm_t *PermMul(perm_t *dest, const perm_t *src);
```

`dest` .... Pointer to the first permutation.
`src` .... Pointer to the second permutation.
Return........ The function returns `dest`, or `NULL` on error.

Description This function multiplies `dest` from the right by `src`. Both permutations must have the same degree.

`PermOrder()` **Order of a permutation**

```
long PermOrder(const perm_t *perm);
```

`perm` .... Pointer to the permutation.
Return .... The function returns the order of `dest`, or −1 on error.

Description This function calculates the order of a permutation.

`PermPower()` **Power of a permutation**

```
perm_t *PermPower(const perm_t *p, long n);
```

`perm` .... Pointer to the permutation.
`n` .... Exponent.
Return .... `n`-th power of `perm` or `NULL` on error.

Description This function calculates the `n`-th power of a permutation. It allocates a new permutation on each call, leaving the original permutation intact. The caller is responsible for deleting the result when it is no longer needed.

n must be positive or 0.

`PermPrint()` **Print a permutation**

```
int PermPrint(const char *name, const perm_t *perm);
```

`name` .... Name to print before the permutation or `NULL` (see below).
`perm` .... Pointer to the permutation.
Return .... The function returns 0 on success and −1 on error.

Description This function prints a permutation on the standard output using cycle notation. If `name` is not `NULL` the name followed by an equal sign is printed before the permutation. For example, the statement `PermPrint("P",P);` may produce the following output:

```
P=(1 9)(2 3 6)(4 5 7)
```

Fixed points are always suppressed in the output.

`PermRead()` **Read a permutation**

```
perm_t *PermRead(FILE *f);
```

`f` .... File to read from.
Return .... Pointer to the permutation read from `f`, or `NULL` on error.

Description This function reads a permutation from a file. `f` must be a pointer to an open file with read permission.

If a permutation was successfully read, the function returns a pointer to a newly created `perm_t` object. The caller is responsible for deleting this object as soon as the permutation is no longer needed.

See also `PermLoad()`

`PermSave()` **Write a permutation**

```
int PermSave(const perm_t *perm, char *fn);
```

`perm` .... Permutation to write.
`fn` .... File name.
Return........ The function returns 0 on success and −1 on error.

Description This function creates a file, writes a single permutation to the file and closes the file. If a f ile with the specified name already exists, it's contents are destroyed.

See also `PermWrite()`

`PermWrite()` **Write a permutation**

```
int PermWrite(FILE *f, const perm_t *perm);
```

`f` .... File to write to.
`perm` .... Permutation to write.
Return........ The function returns 0 on success and −1 on error.

Description This function writes a permutation to a file. The file `f` must be open for writing.

See also `PermSave()`

`PolAdd()` **Add polynomials**

```
poly_t *PolAdd(poly_t *dest, const poly_t *src);
```

`dest` .... Pointer to the first polynomial.
`src` .... Pointer to the second polynomial.
Return........ The function returns `dest`, or `NULL` on error.

Description Thus function adds `src` to `dest` and returns `dest`. The polynomials must be over the same field. Otherwise, the return value is `NULL`.

`PolAlloc()` **Allocate a polynomial**

```
poly_t *PolAlloc(long fl, long degree);
```

`fl` .... Field order.
`degree` .... Degree of the polynomial.
Return........ Pointer to a new `poly_t` structure or `NULL` on error.

Description This function creates a new polynomial over GF(`fl`) and initializes its value to x^n, with $n =$ `degree`.

See also `PolDup()` `PolFree()`

`PolCompare()` **Compare polynomials**

`int PolCompare(const poly_t *a, const poly_t *b);`

`a` .... First polynomial.
`b` .... Second polynomial.
Return........ The return value is 1 if $a > b$, -1 if $a < b$, and 0 if $a = b$. In case of error, the function returns -2.

Description This function compares two polynomials and returns 0 if the polynomials are equal, -1 if $a < b$, and 1 if $a > b$. The ordering of polynomials is defined as follows.

1. If a and b are over different fields, the polynomials over the larger field is greater.
2. Otherwise, if they have different degrees, the polynomial with the higher degree is greater.
3. If both field and degree are equal, the result of the comparison depends on the internal representation of the field elements and does not have any significance.

`PolDerive()` **Derive a polynomials**

`poly_t *PolDerive(poly_t *pol);`

`pos` .... Pointer to the polynomial.
Return........ Returns `pol`.

Description This function derives a polynomial. Note that the derived polynomial is stored in `pol`, replacing the original polynomial. The following piece of code shows how to keep the original polynomial intact while calculating the derivative:

```
poly_t *pol, *der;
...
der = PolDerive(PolDup(pol));
```

`PolDivMod()` **Polynomial division**

`poly_t *PolDivMod(poly_t *a, const poly_t *b);`

`a` .... Pointer to the first polynomial (numerator).
`src` .... Pointer to the second polynomial (denominator).
Return........ The function returns the quotient, or `NULL` on error.

Description This function performs a polynomial division. Given two polynomials a and $b \neq 0$, which must be over the same field, `PolDivMod()` finds two polynomials q and r such that $a = qb + r$, and $\deg(r) < \deg(b)$.

The quotient, q, is returned as the function result. This is a newly allocated polynomial. The caller is responsible for deleting the quotient when it no longer needed.

The remainder, r, is stored in `a` and replaces the original value. If you need to keep the value of `a` you must make a copy using `PulDup()` before calling `PolDivMod()`. `b` is not changed.

See also `PolMod()`

`PolDup()` **Duplicate a polynomial**

`poly_t *PolDup(const poly_t *x);`

`x` .... Pointer to the polynomial.
Return........ Pointer to a copy of `x`, or `NULL` on error.

Description This function creates a copy of an existing polynomial.

See also `PolAlloc()` `PolFree()`

`PolFactorization()()` **Factor a polynomial**

`fpoly_t *PolFactorization(const poly_t *pol);`

`pol` .... Polynomial to factor.
Return........ The factorization of `pol` or `NULL` on error.

Description This function decomposes a polynomial into irreducible factors using the Berlekamp algorithm.

`PolFree()` **Free a polynomial**

`void PolFree(poly_t *x);`

`x` .... Pointer to the polynomial.

Description This function frees a polynomial data structure and cleans up all internal data.

See also `PolAlloc()` `PolDup()`

`PolGcd()` **G.c.d. of polynomials**

`poly_t *PolGcd(const poly_t *a, const poly_t *b);`

`a` .... First polynomial.
`b` .... Second polynomial.
Return........ Greates common divisor of `a` and `b`, or `NULL` on error.

Description This function calculates the gratest common divisor of two poynomials. The polynomials must be over the same field, and at least one of them must be different from zero.

Unlike most polynomial functions, `PolGcd()` normalizes the result, i.e. the leading coefficient of the g.c.d. is always one.

See also `PolGcdEx()`

`PolGcdEx()` **G.c.d. of polynomials**

```
int PolGcdEx(const poly_t *a, const poly_t *b, poly_t **result);
```

`a` .... First polynomial.
`b` .... Second polynomial.
`result` .... Result buffer for the g.c.d. and coefficients (see below).
Return........ 0 on sucess, -1 on error.

Description Given two polynomials $a, b \in \mathrm{GF}(q)[x]$, this function calculates the greatest common divisor $g := \gcd(a, b)$ and two polynomial $p, q \in \mathrm{GF}(q)[x]$ such that $g = pa + qb$. As with `PolGcd()`, a and b must not both be zero, and the g.c.d. is always normalized.

`result` must be a pointer to an array of three `poly_t *` items. If the function is successful, pointers to g, p, and q are stored in `result[0]`, `result[1]`, and `result[2]`, respectively. The caller is responsible for destroying these polynomials when they are no longer needed. In particular, you must not use the same `result` buffer again with `PolGcdEx()` before the contents have been freed.

See also `PolGcd()`

`PolLoad()` **Read a polynomial**

```
poly_t *PolLoad(char *fn)
```

`fn` .... File name.
Return........ Pointer to the polynomial read from the file, or `NULL` on error.

Description This function opens a file, reads a single polynomial, and closes the file. The return value is a pointer to the polynomial or `NULL` on error. If the file contains more than one polynomial, only the first one is read.

If a polynomial was successfully read, the function returns a pointer to a newly created `poly_t` object. The caller is responsible for deleting this object as soon as it no longer needed.

See also `PolRead()`

`PolMod()` **Polynomial division**

```
poly_t *PolMod(poly_t *a, const poly_t *b)
```

`a` .... Pointer to the first polynomial (numerator).
`src` .... Pointer to the second polynomial (denominator).
Return........ The function returns `a`, which has been reduced modulo `b`. On error the function returns `NULL`.

Description This function reduces a modulo b.

See also PolDivMod()

PolMul() **Multiply polynomials**

```
poly_t *PolMul(poly_t *dest, const poly_t *src);
```

dest .... Pointer to the first polynomial.
src .... Pointer to the second polynomial.
Return........ The function returns dest, or NULL on error.

Description Thus function multiplies src by dest and returns dest. The polynomials must be over the same field. Otherwise, the return value is NULL.

PolPrint() **Print a polynomial**

```
void PolPrint(char *name, const poly_t *p);
```

name .... Name to print before the polynomial or NULL (see below).
p .... Pointer to the polynomial.
Return........ 0 on success, −1 on error.

Description This function prints a polynomial on the standard output in a human-readable form. If name is not NULL the name followed by an equal sign is printed before the polynomial. For example, the statement PolPrint("P",P); may produce the following output:

```
P=3x^2+x+1
```

PolRead() **Read a polynomial from a file**

```
poly_t *PolRead(FILE *f);
```

f .... File to read from.
Return........ Pointer to the plynomial, or NULL on error.

Description This function reads a polynomial from a file. If successful, the return value is a pointer to a new poly_t object. The caller is responsible for deleting this polynomial with PolFree() when it is no longer needed.

PolSave() **Write a polynomial**

```
int PolSave(const poly_t *pol, char *fn);
```

pol .... Polynomial to write.
fn .... File name.
Return........ The function returns 0 on success and −1 on error.

Description This function creates a file, writes a single polynomial to the file and closes the file. If a f ile with the specified name already exists, it's contents are destroyed.

See also `PolWrite()`

`PolWrite()` **Write a polynomial to a file**

```
int PolWrite(FILE *f, const poly_t *p);
```

`f` .... File to write to.
`p` .... Pointer to the polynomial.
Return........0 on success, −1 on error.

Description This function writes a polynomial to a file.

`poly_t` **Polynomial data structure**

```
typedef struct
{
    tMtxObjType id;
    long fl;
    long deg;
    size_t size;
    FEL *buf;
}
poly_t;
```

Description The `poly_t` structure represents a polynomial over a finite field. `fl` is the field order, and `deg` is the degree of the polynomial. The coefficients are stored in `buf` as an array of `deg` + 1 field elements. `buf[i]` contains the coefficient of x^i. The `size` member is used internally to keep track of the buffer's size.

Polynomials need not be normalized. However, many polynomial functions depend on the leading coefficient being non-zero.

`proginfo_t` **Program information structure**

```
typedef struct
{
    char *name;
    char *shortdesc;
    char *rcsrev;
    char **helptext;
} proginfo_t;
```

Description This data structure is used to store information about the program. It is used by the command line parser, e.g., to display the help text.

The meanings of the fields are as follows. `name` is the program name, usually the same name as the executable file. `shortdesc` is a one-line description of the program which will be included in the help text. `rcsrev` should contain the RCS version number. If you don't use RCS, you can insert the version number by hand, e.g.

```
    rcsversion = "$Revision: 2.11";
```

`helptext` is the help text, which will be displayed if the user uses the `-help` option. See `zgetopt()` for an example.

`QuotProj()` **Projection on quotient**

`matrix_t *QuotProj(matrix_t *subspace, matrix_t *vectors):`

Return........Projection of `vectors` on the quotient by `subspace`, or `NULL` on error.

Description This function calculates the projection of a matrix onto the quotient by a subspace. As with `Split()`, the subspace need not be in echelon form, but it will be converted to echelon form during the projection. The return value is a pointer to the projection, which is always in echelon form.

See also `SpinUp()` `Split()`

`set_t` **Integer set data structure**

```
typedef struct
{
    size_t len, max;
    long *buf;
} set_t;
```

Description The type `set_t` represents a set of (long) integers. Internally, the set is stored as a sorted list. `set_t` should be used if the number of elements in the set is typically much less than the elements themselves. In other cases, when the number of elements is comparable to the elements themselves, the `bitstring_t` type may be more suitable.

`SetAlloc()` **Create a new set**

`set_t *SetAlloc();`

Return........Pointer to the new set or `NULL` on error.

Description This function creates a set. The set is initially empty. To destroy a set, use `SetFree()`, *not* `free()`.

See also `SetDup()` `SetFree()`

`SetContains()` **Check if a number is in a set**

`int SetContains(const set_t *set, long elem);`

`set`..........Pointer to the set.
`elem`.........Number to check.
Return........1 if `set` contains `elem`, 0 else.

Description This function checks if `set` contains the number `elem`.

See also `SetInsert()`

`SetDup()` **Duplicate a set**

```
set_t *SetDup(const set_t *s);
```

`s` .... Pointer to the set.
Return........Pointer to a copy of the set, or `NULL` on error.

Description This function creates a copy of an existing set.

See also `SetAlloc()` `SetFree()`

`SetFree()` **Destroy a set**

```
void SetFree(set_t *x);
```

`x` .... Pointer to the set.

Description This function frees an integer set. The argument must be a `set_t` structure which has previously been allocated with `SetAlloc()`.

See also `SetAlloc()` `SetDup()`

`SetInsert()` **Insert an element into a set**

```
int SetInsert(set_t *set, long elem);
```

`set` .... Pointer to the set.
`elem`.........Number to insert.
Return........0 on success, −1 on error.

Description This function inserts the number `elem` into `set`.

See also `SetContains()`

`SetPrint()` **Write a set to stdout**

```
void SetPrint(char *name, const set_t *s);
```

`name`.........Name to print before the set, or `NULL`.
`set` .... Pointer to the set.

Description This function prints a set in human-readable format on the standard output. If `name` is different from `NULL`, `name` and an '=' is printed before the set.

`SetRead()` **Read a set from a file**

```
set_t *SetRead(FILE *f);
```

`f` .... Pointer to the file.
Return........ Pointer to the set read from `f`, or `NULL` on error.

Description This function reads a set from a file.

See also `SetWrite()`

`SetWrite()` **Write a set to a file**

```
int SetWrite(FILE *f,const set_t *set);
```

`f` .... Pointer to the file.
`set` .... Pointer to the set.
Return........ 0 on success, −1 on error.

Description This function write a set to a file.

See also `SetRead()`

`SFOpen()` **Open file**

```
FILE *SFOpen(const char *name, int mode);
```

`name` .... File name.
`mode` .... Open mode and additional flags (see below).
Return........ Pointer to an open file or `NULL` on error.

Description This function opens a file. `mode` must be any of `FM_READ` (open for reading), `FM_CREAT` (create a new file) or `FM_APPEND` (append to existing file or create a new file). Additional flags may be or'ed to the mode:

Mode	Meaning
`FM_LIB`	If the file does not exist in the current directory, look in the library directory. The library directory is defined either by the environment variable `MTXLIB` or at compile-time by the symbol `MTXLIB`.
`FM_TEXT`	Open in text mode. This flag must be used on some systems (e.g., MS-DOS) to open text files. By default, files are assumed to contain binary data.
`FM_NOERROR`	Do not generate an error if the file does not exist.

The return value is a pointer to the open file or NULL on error. Possible error codes are (in `MtxErrNo`) are `ERR_BADARG` indicating an invalid mode, or `ERR_FILEOPEN` if the file could not be opened for whatever reason.

See also `SFSeek()` `zreadhdr()` `zwritehdr()`

SFSeek() — Set file pointer

```
int SFSeek(FILE *file, long pos);
```

`file` .... Pointer to `FILE` structure.
`pos` .... New position of file pointer.
Return .... The function returns 0 on success. Any other value indicates an error.

Description This function sets the file pointer to a given position. if `pos` is greater or equal to zero, it is interpreted as an absolute position (relative to start of file). If `pos` is negative, the file pointer is moved to the end of file.

See also `SFOpen()` `zseek()`

SInit() — Initialize OS dependent stuff

```
void SInit();
```

Description This function is called during library initialization. It performs any OS-dependent actions. Applications never call this function directly.

Smalloc() — Allocate memory

```
void *Smalloc(size_t nbytes);
```

`nbytes` .... Size of memory block to allocate.
Return .... Pointer to memory block or `NULL` on error.

Description This function works like `malloc()`. Unlike some implementations of `malloc()`, it returns a valid pointer if `nbytes` is zero.

See also `Srealloc()`

SpinUp() — Spin up

```
matrix_t *SpinUp(const matrix_t *seed, int ngen, const matrix_t **gen,
    int flags);
```

`seed` .... Matrix with seed vectors.
`ngen` .... Number of generators in `gen`.
`gen` .... List of generators.
`flags` .... Flags, a combination of `SF_xxx` constants (see below).
Return .... Span of the seed vector(s) under the action of the generators, or `NULL` on error.

Description This function calculates the submodule generated by one or more 'seed' vectors under the action of a set of matrices. `gen` must be an array of `ngen` (pointers to) square matrices, and seed must be a matrix with the same number of columns and any number of rows. Of course, all matrices must be over the same field.

`flags` must be a combination of the following values:

- `SF_FIRST` ..... Only the first row of `seed` is taken as seed vector.
- `SF_EACH` ...... Each row of `seed` is taken as seed vector.
- `SF_MAKE` ...... One vector from each 1-dimensional subspace of the row space of `seed` is taken as seed vector.
- `SF_SUB` .... Find a submodule: spin up seed vectors one-by-one until a seed vector generates a proper submodule.
- `SF_CYCLIC` .... Find a cyclic vector: spin up vectors one-by-one until a seed vector generates the whole space.
- `SF_COMBINE` ... Calculate the submodule generated by the set of all seed vectors. This ist typically used with `SF_EACH` to calculate the submodule generate by the row space of `seed`.
- `SF_STD` .... Spin up 'canonically'. Note: This increases both computation time and memory usage.

Note that the seed modes, `SF_FIRST`, `SF_EACH` and `SF_MAKE`, and the search modes, `SF_SUB`, `SF_CYCLIC`, `SF_COMBINE`, are mutually exclusive. If, in mode `SF_SUB` or `SF_CYCLIC`, no seed vector generates a proper submodule or the whole space, respectively, this is *not* considered an error, and the return value is not `NULL`. The rows of the matrix returned by `SpinUp()` always form a basis of an invariant subspace, but you must examine the number of rows of that matrix to find out if it is a proper subspace, or null, or the whole space.

`Split()` **Split a module**

```
int Split(matrix_t *subspace, int ngen, const matrix_t *gen[],
    matrix_t **sub, matrix_t **quot)
```

- `subspace` ..... Pointer to an invariant subspace.
- `ngen` .... Number of generators.
- `gen` .... Generators.
- `sub` .... Buffer for action on the subspace.
- `quot` .... Buffer for action on the quotient.
- Return .... The function returns 0 on success. Any other return value indicates an error.

Description Given a matrix representation of an Algebra A and an A-invariant subspace, this function calculates matrix representations corrsponding to both subspace and quotient.

`subspace` is a basis for the invariant subspace. This matrix will be converted to echelon if necessary. `ngen` is the number of generators, and `gen` is an array of (pointers) to matrices containing the representation of the generators. The result consists of `ngen` matrices for both subspace and quotient. Pointers to these matrices are stored in `sub` and `quot`, respectively. Both `sub` and `quot` may be `NULL`, indicating that the corresponding representation should not be calculated.

Here is an example for `SpinUp()` and `Split()`:

```
matrix_t *gens[5];
matrix_t *sub[5], quot[5];
matrix_t *seed;
matrix_t *subspace;
...
```

```
subspace = SpinUp(seed,5,gens,SF_FIRST|SF_SUB);
if (subspace->nor > 0 && subspace->nor < subspace->noc)
{
    printf("Split!\n");
    Split(subspace,5,gens,sub,quot);
}
else
    printf("Not split!\n");
```

See also `SpinUp()` `QuotProj()`

`Srealloc()` **Resize memory block**

```
void *Srealloc(size_t nbytes);
```

`buf` .... Pointer to memory block.
`nbytes` .... Desired size.
Return........ Pointer to resized memory block or NULL on error.

Description This function works like `relloc()` but handles zero-length blocks correctly.

See also `Smalloc()`

`StfBeginEntry()` **Begin entry**

```
int StfBeginEntry(StfData *f, const char *name);
```

`f` .... Pointer to a structured text file (STF) object.
`name` .... Name of the entry.
Return........ The function returns 0 on success and -1 on error.

Description This function begins a new entry. Be sure to terminate any incomplete entries with `StfEndEntry()` before you start a new entry. If you don't, the incomplete entry may be lost, and the data file may become corrupted.

Before yo use `StfBeginEntry()`, check if you can do the job with one of the `StfWriteXXX()` functions. In particular, there are functions to write integers and sequences of integers. If you have more complicated data to write, you may need to construct the output manually. Here is an example:

```
StfBeginEntry(f,"Param");
StfPut(f,"(");
StfPutInt(f,11);
StfPut(f,":");
StfPutInt(f,22);
StfPut(f,")");
StfEndEntry(f);
```

This code produces the following line in the output file:

```
Param := (11:22);
```

See also `StfClose()` `StfEndEntry()` `StfGetInt()` `StfGetName()` `StfGetVector()` `StfOpen()` `StfReadLine()` `StfWriteInt()` `StfWriteValue()` `StfWriteVector()`

`StfClose()` **Close a structured text file**

`int StfClose(StfData *f);`

`f` .... Pointer to an open structured text file (STF) object.
Return........ The function returns 0 on success. Any other return value indicates an error.

Description This function closes a structured text file. Closing the file implies that the memory occupied by the `StfData` structure is freed. Thus, after return, `f` is invalid and must not be dereferenced.

See also `StfBeginEntry()` `StfEndEntry()` `StfGetInt()` `StfGetName()` `StfGetVector()` `StfOpen()` `StfReadLine()` `StfWriteInt()` `StfWriteValue()` `StfWriteVector()`

`StfEndEntry()` **End entry**

`int StfEndEntry(StfData *f);`

`f` .... Pointer to a structured text file (STF) object.
Return........ The function returns 0 on success and −1 on error.

Description This function terminates the current entry and flushes the STF's line buffer. See `StfBeginEntry()` for an example.

See also `StfBeginEntry()` `StfClose()` `StfGetInt()` `StfGetName()` `StfGetVector()` `StfOpen()` `StfReadLine()` `StfWriteInt()` `StfWriteValue()` `StfWriteVector()`

`StfGetInt()` **Read an integer**

`int StfGetInt(StfData *f, int *buf);`

`f` .... Pointer to a structured text file (STF) object.
`buf` .... Pointer to a buffer receiving the value.
Return........ The function returns 0 on success and −1 on error.

Description This function gets one integer from the current line and increments the read pointer accordingly. Before this function is called, a line must have been read with `StfReadLine()` and prepared with `StfGetname()`.

Reading starts at the current position. Any leading blanks are skipped. On return, the new current position is the character following the last digit. If there is no integer to read, the current position is not changed, and the function retutns −1.

Here is an example:

```
StfFile *f = StfOpen("test","r");
int dim, degree, result = 0;
while (result == 0 && StfReadLine(f) == 0)
{

    char *name = StfGetName(f);
    if (!strcmp(name,"Dimension"))
        result = StfGetInt(f,&dim);
```

```
        else if (!strcmp(name,"Degree"))
            result = StfGetInt(f,&degree);
    }
```

This code fragment opens a STF and reads two parameters, “Degree” and “Dimension”, and stores their values into the variables `degree` and `dim`, respectively.

See also `StfBeginEntry()` `StfClose()` `StfEndEntry()` `StfGetName()` `StfGetVector()` `StfOpen()` `StfReadLine()` `StfWriteInt()` `StfWriteValue()` `StfWriteVector()`

`StfGetName()` — Get file name

```
const char *StfGetName(StfData *f);
```

`f` .... Pointer to a structured text file (STF) object.
Return........Name found in text line or `NULL` on error.

Description This function extracts the name part of internal line buffer and prepares the buffer for further parsing with `StfGetXXX()` functions.

`StfGetName()` can be called only after a line was successfully read with `StfReadLine()`. It must be called before any of the `StfGetXXX()` functions. See `StfGetInt()` for an example.

The text line must have the format *name*`:=`*value*. Then, the return value is a pointer to *name* and `f->GetPtr` points to the first non-space character after the `:=`.

See also `StfBeginEntry()` `StfClose()` `StfEndEntry()` `StfGetInt()` `StfGetVector()` `StfOpen()` `StfReadLine()` `StfWriteInt()` `StfWriteValue()` `StfWriteVector()`

`StfGetVector()` — Read a vector

```
int StfGetVector(StfData *f, int *bufsize, int *buf);
```

`f` .... Pointer to a structured text file (STF) object.
`bufsize` ...... Pointer to a variable containing the buffer size. On return, the variable contains the number of elements actually read.
`buf` .... Pointer to a buffer receiving the data.
Return........The function returns 0 on success and -1 on error.

Description This function reads a sequence of integers. The sequence must have been written with `StfWriteVector()` or at least in the same compatible format.

Before using this function, a line must have been read with `StfReadLine()` and prepared with `StfGetname()`. Reading starts at the current position, i.e., at the first non-space character after the `:=`, or at the first character that was not read by the previous `StfGetXXX()` or `StfMatch()`.

The caller must supply two buffers, the data buffer (`buf`) and an integer buffer (`buflen`). When `StfGetVector()` is called, the variable pointed to by `bufsize` must contain the maximal number of integers that can be stored in `buf`. On successful return, the variable contains the number of integers that were actually stored, which may be smaller than the original value.

If the vector is to long to fit into the user-supplied buffer, the function reads as many entries as possible and returns -1.

Here is an example:

```
char *name = StfGetName(f);
if (!strcmp(name,"Vector"))
{
    int vec[10];
    int vecsize = 10;
    StfGetVector(f,&vecsize,vec);
    printf("%d elements read\n",*vecsize);
}
```

See also `StfBeginEntry()` `StfClose()` `StfEndEntry()` `StfGetInt()` `StfGetName()` `StfOpen()` `StfReadLine()` `StfWriteInt()` `StfWriteValue()` `StfWriteVector()`

`StfMatch()` — Skip text

`int StfMatch(StfData *f, const char *pattern);`

`f` .... Pointer to a structured text file (STF) object.
`pattern` ...... The text to be skipped.
Return........ The function returns 0, if the complete text in `pattern` has beed skipped. Otherwise the return value is −1.

Description This function reads (and skips) the text given by `pattern`.

Before using this function, a line must have been read with `StfReadLine()` and prepared with `StfGetname()`. Reading starts at the current position, i.e., at the first non-space character after the `:=`, or at the first character that was not read by the previous `StfGetXXX()` or `StfMatch()`.

A space in `pattern` matches any number (including 0) of spaces or tabs. Any other characters in `pattern` are matched one-to-one against the input line.

If `pattern` is matched completely, the current position is updated to the character after the last matched character. Otherwise, the current position is not changed and the function returns −1. !seealso @stf

---

`StfOpen()` — Open a structured text file

`StfData *StfOpen(const char *name, const char *mode);`

`name` .... File name.
`mode` .... Open mode (see below for details).
Return........ On success, the return value is a pointer to a `StfData` structure which may be used with `StfXXX()` functions. If the file cannot be opened the return value is `NULL`.

Description This function opens a structured text file. `name` and `mode` have the same semantics as with `fopen()`.

Mode "w" opens the file for writing, for example with `StfWriteInt()`. If a file with the specified name already exists, it is truncated to length zero. Otherwise a new file is created.

Mode "r" opens the file for reading. If the file does not exist the function fails. You cannot use any of the `StfWriteXXX()` or `StfPutXX()` with a file opened for reading.

See also `StfBeginEntry()` `StfClose()` `StfEndEntry()` `StfGetInt()` `StfGetName()` `StfGetVector()` `StfReadLine()` `StfWriteInt()` `StfWriteValue()` `StfWriteVector()`

`StfReadLine()` — Read next line

```
int StfReadLine(StfData *f);
```

`f` .... Pointer to a structured text file (STF) object.
Return........ The function returns 0 on success and −1 on end-of-file or error.

Description This function reads a single text line into the STF object's internal line buffer and prepares the text for parsing with `StfGetXXX()` functions.

`StfReadLine()` strips comments and assembles multi-line texts into a single line. Thus, the application need not handle comments and multi-line texts.

See also `StfBeginEntry()` `StfClose()` `StfEndEntry()` `StfGetInt()` `StfGetName()` `StfGetVector()` `StfOpen()` `StfWriteInt()` `StfWriteValue()` `StfWriteVector()`

`StfWriteInt()` — Write an integer number

```
int StfWriteInt(StfData *f, const char *name, int value);
```

`f` .... Pointer to a structured text file (STF) object.
`name` .... Name of the entry.
`value` .... Value of the entry.
Return........ The function returns 0 on success and −1 on error.

Description This function writes an integer to a structured text file. For example, the statement

```
StfWriteInt(f,"Dimension",42);
```

produces the following output line:

```
Dimension := 42;
```

See also `StfBeginEntry()` `StfClose()` `StfEndEntry()` `StfGetInt()` `StfGetName()` `StfGetVector()` `StfOpen()` `StfReadLine()` `StfWriteValue()` `StfWriteVector()`

`StfWriteValue()` — Write a text

```
int StfWriteValue(StfData *f, const char *name, const char *value);
```

`f` .... Pointer to a structured text file (STF) object.
`name` .... Name of the entry.
`value` .... Value of the entry.
Return........ The function returns 0 on success and −1 on error.

Description This function writes an arbitrary text to a structured text file. For example, the statement

```
StfWriteValue(f,"Note","This is a note");
```

produces the following output line:

```
Note := This is a note;
```

Note that any leading spaces in the value will be stripped off when reading the file.

See also `StfBeginEntry()` `StfClose()` `StfEndEntry()` `StfGetInt()` `StfGetName()` `StfGetVector()` `StfOpen()` `StfReadLine()` `StfWriteInt()` `StfWriteVector()`

`StfWriteVector()` — Write a vector

```
int StfWriteVector(StfData *f, const char *name, int size, int *value);
```

`f` .... Pointer to a structured text file (STF) object.
`name` .... Name of the entry.
`size` .... Size of the vector (number of entries, not bytes).
`value` .... Pointer to the vector.
Return .... The function returns 0 on success and −1 on error.

Description This function writes a sequence of integers to a structured text file. For example, the statement

```
int dims[5] = {11,22,33,44,55};
StfWriteVector(f,"Dimensions",dims,5);
```

produces the following output line:

```
Dimensions := [11,22,33,44,55];
```

See also `StfBeginEntry()` `StfClose()` `StfEndEntry()` `StfGetInt()` `StfGetName()` `StfGetVector()` `StfOpen()` `StfReadLine()` `StfWriteInt()` `StfWriteValue()`

`STimeLimit()` — Set CPU time limit

```
void STimeLimit(long nsecs);
```

`nsecs` .... CPU time limit in seconds.

Description This function sets a CPU time limit for the calling process. If the process does not exit within the specified time, it it killed.

Note: This function is not available on all platforms.

See also `STimeUsed()`

`STimeUsed()` — CPU time

```
long STimeUsed();
```

Return .... CPU time.

Description This function returns the CPU time used by the calling process in units of 1/10 seconds.

See also `STimeLimit()`

`WgFree()` — Terminate the word generator

```
void WgFree(wgdata_t *b);
```

`n` .... Pointer to word generator data.

Description This function terminates the word generator specified by `b` and cleans up internal data structures. Note that the generators that were passed to `WgInit()` are *not* freed.

See also `WgInit()` `WgMakeFingerPrint()` `WgMakeWord()` `WgSymbolicName()`

`WgInit()` **Initialize the word generator**

```
wgdata_t *WgInit(int ngen, const matrix_t *gen[]);
```

`ngen` .... Number of generators.
`gen` .... Pointer to array of generators.
Return .... Pointer to a new word generator data structure, or `NULL` on error.

Description This function initializes the word generator for a given matrix algebra. `ngen` is the number of generators, and `gen` is an array of pointers to the generators. The generators must be square matrices of the same dimension over the same field.

On success, `WgInit()` returns a pointer to an internal data structure which can be used in subsequent calls to `WgMakeWord()` and `WgFree()`. If an error occurs, the return value is NULL.

Note that the word generator does not make internal copies of the generators. The caller must assure that the generators are not deleted or modified in any way as long as the word generator is used.

See also `WgFree()` `WgMakeFingerPrint()` `WgMakeWord()` `WgSymbolicName()`

`WgMakeFingerPrint()` **Calculate finger print**

```
void WgMakeFingerPrint(wgdata_t *b, long fp[]);
```

`b` .... Pointer to word generator data.
`fp` .... Buffer for the finger print (6 numbers).

Description This function calculates the 'finger print' of a module, i.e. the nullities of the first 6 words. `fp` must be a pointer to an array of at least 6 long integers.

See also `WgFree()` `WgInit()` `WgMakeWord()` `WgSymbolicName()`

`WgMakeWord()` **Make a word**

```
matrix_t *WgMakeWord(wgdata_t *b, long n);
```

`b` .... Pointer to word generator data.
`n` .... Word number.
Return .... Matrix representation of word `n`, or `NULL` on error.

Description This function calculates a 'random' element in a matrix algebra. The algebra, i.e., a set of generating matrices, must be specified with `WgInit()`, and `b` is the pointer returned by `WgInit()`. `n` is the number of the element to calculate. Refer to section 3.10 for more details.

Here is an example demonstrating the usage of the word generator:

```
matrix_t *mat[3];
wgdata_t *wg;
matrix_t *word;
long nul;
...
wg = WgInit(3,mat);
word = WgMakeWord(wg,1833);
nul = MatNullity__(word);
printf("Word 1833 has nullity %ld\n",nul);
WgFree(wg);
```

This example initializes the word generator with three matrices, calculates the word number 1833 and prints out its nullity.

See also `WgFree()` `WgInit()` `WgMakeFingerPrint()` `WgSymbolicName()`

`WgSymbolicName()` **Symbolic name of a word**

```
const char *WgSymbolicName(const wgdata_t *b, long n);
```

`b` .... Pointer to word generator data.
`n` .... Word number.
Return........Symbolic name of the word `n`.

Description This function returns a symbolic representation of the word number `n` as a polynomial in the generators. The generators are named a, b, c...

The return value is a pointer to a static buffer which is overwritten on each call.

See also `WgFree()` `WgInit()` `WgMakeFingerPrint()` `WgMakeWord()`

`zadd()` **Finite field addition**

```
FEL zadd(FEL a, FEL b);
```

`a` .... First field element.
`b` .... Second field element.
Return........`a + b`

Description This function returns the sum of two field elements. Before you can use this function, the field must have been selected with `zsetfield()`. The arguments are not checked. If either argument is not in the allowed range the result is undefined and the program may crash.

`zadd()` may be implemented as a macro. In this case, it is guaranteed that both arguments are evaluated exactly once.

See also `zdiv()` `zinv()` `zmul()` `zneg()` `zsub()`

`zaddmulrow()` — **Add a multiple of a row**

```
PTR zaddmulrow(PTR dest, PTR src, FEL f);
```

`dest` .... The vector to add to.
`dest` .... The vector to add.
`f` .... Coefficient.
Return........ The function always returns `dest`.

Description This function adds a multiple of `src` to `dest`.

See also `zaddrow()` `zmulrow()`

`zaddrow()` — **Add two rows**

```
PTR zaddrow(PTR dest, PTR src);
```

`dest` .... The row to add to.
`src` .... The row to add.
Return........ The function always returns `dest`.

Description This function adds `src` to `dest`. The row size must have been set with `zsetlen()`.

See also `zaddmulrow()` `zmulrow()`

`zadvance()` — **Advance a row pointer**

```
void zadvance(PTR *ptr, long nrows);
```

`ptr` .... Pointer to the `PTR` variable to increment.

Description This function increments a pointer by `nrows` rows using the current row size. This functions is typically used to step through the rows of a matrix. Before it can be used, the row size must have been set with `zsetlen()`. Here is an example:

```
PTR mat;
zsetlen(100);
mat = zalloc(100);
zadvance(&mat,49);
```

After this sequence, `mat` points to the 50th row of the 100×100 matrix `mat`.

Note that the first argument is not a `PTR` but a `*PTR`, i.e., a pointer to a `PTR`! Thus you will always need the `&` operator as in the example.

`zalloc()` — **Allocate memory and initialize**

```
PTR zalloc(long nrows);
```

`nrows` .... Number of rows.
Return........ Returns a pointer to the memory block or NULL on error.

Description This function allocates a block of memory for a vector or a matrix. Memory is initialized to zero. `nrows` ist the number of rows to allocate.

Field order and row size must have been set with ¡zsetfield()¿ and `zsetlen()`, respectively. Unlike `malloc()` on some systems, this function does not fail if `nrows` is zero. In this case, a 'zero-length' block is allocated.

See also `zsize()`

`zchar` **Characteristic**

`extern long zchar;`

Description The `zchar` variable contains the characteristic of the current field. Like `zfl`, this variable may be used anywhere, but it must not be modified directly.

See also `zfl` `zgen`

`zcleanrow()` **Clean row**

`int zcleanrow(PTR row, PTR matrix, long nor, const long *piv);`

`row` .... Pointer to row to be cleaned.
`matrix` .... Matrix to clean with.
`nor` .... Number of rows.
`piv` .... Pivot table for `matrix`.
Return........ The return value is always 0.

Description This function 'cleans' a row with a matrix, i.e., it adds suitable multiples of the rows of `matrix` to `row` such that all pivot positions in `row` are zero. `nor` is the number of rows in `matrix`. The number of columns (for both `matrix` and `row`) must have been set before with `zsetlen()`.

`piv` is a pointer to the pivot table of `matrix`, which can be generated `zmkpivot()`.

See also `zcleanrow2()`

`zcleanrow2()` **Clean row**

`int zcleanrow(PTR row, PTR matrix, long nor, const long *piv);`

`row` .... Pointer to row to be cleaned.
`matrix` .... Matrix to clean with.
`nor` .... Number of rows.
`piv` .... Pivot table for `matrix`.
Return........ The return value is always 0.

Description This function acts exactly as `zcleanrow()`. In addition, it stores the operations performed in `row2`. `row2` must be a row of at least `nor` entries. It must be initialized to zero by the caller, or the results are not defined.

On return, `row2` contains the coefficients by which the rows of `matrix` were multiplied and then subtracted from `row`.

See also `zcleanrow()`

`zcmprow()` **Compare two rows**

```
int zcmprow(PTR dest, PTR src);
```

`dest` .... Pointer to the first row
`src` .... Pointer to the second row
Return........ The function returns 0 if the two rows are identical, and 1 otherwise.

Description This function compares two rows. As with all row operations, the row length must have been set before with `zsetlen()`. `zcmprow()` checks only for equality and does not define an ordering of rows.

`zdiv()` **Finite field division**

```
FEL zdiv(FEL a, FEL b);
```

`a` .... First field element.
`b` .... Second field element.
Return........ `a/b`

Description This function returns the quotient of two field elements. Before you can use this function, the field must have been selected with `zsetfield()`. The arguments are not checked. If either argument is not in the allowed range, or if `b` is zero, the result is undefined and the program may crash.

`zdiv()` may be implemented as a macro. In this case, it is guaranteed that both arguments are evaluated exactly once.

See also `zadd()` `zinv()` `zmul()` `zneg()` `zsub()`

`zembed()` **Embed a subfield**

```
FEL zembed(FEL a, long subfield);
```

`a` .... Field element.
`subfield` ..... Subfield order.
Return........ `a`, embedded into the current field.

Description This function embeds an element of a subfield into the currently selected field. `subfield` must be a divisor of the current field order.

See also `zrestrict()`

`zextract()` **Extract a mark from a row**

```
FEL zextract(PTR row, long col);
```

`row` .... Pointer to the row.
`col` .... Index of mark to extract.
Return........ `col`-th entry of `row`.

Description This function returns the entry at position `col` of a row. Note that column indexes start with 1, not 0. Like `zinsert()`, this function does not depend on the current row size. Reading beyond the end of a row will probably not produce an error, but the result is undefined.

See also `zinsert()`

`zextractcol()` **Extract one column of a matrix**

`PTR zextractcol(PTR mat,long nor,long col,PTR result);`

`mat` .... Pointer to the matrix.
`nor` .... Number of rows in matrix.
`col` .... Column to extract (starting with 1).
`result` .... Pointer to buffer for the extracted column.
Return........ The function returns `result`.

Description This function extracts one column out of a matrix and stores it as a row vector in `result`. The number of columns of the matrix must have been set with `zsetlen()`. `nor` is the number of rows in the matrix. The result is a row with `nor` entries, i.e., the length of `result` must be at least `nor`. `mat` and `result` must not overlap, or ther esult is undefined.

`zfindpiv()` **Find pivot column**

`long zfindpiv(PTR row, FEL *mark);`

`row` .... Pointer to the row.
Return........ Index of the first non-zero entry in `row`, or 0 it alle entries are zero.

Description This function scans the vector `row` and finds the first non-zero mark. The mark is stored into `mark` and its position (counting from 1) is returned. If the whole vector is zero, `zfindpiv()` returns 0 and leaves `mark` unchanged.

`zfl` **Field order**

`extern long zfl;`

Description This variable contains the order of the current field. `zfl` max be used in expressiond but must never modified directly. To change the current field, use `zsetfield()`.

See also `zfl` `zgen`

`zftogap()` **Convert to GAP format**

`char *zftogap(FEL f);`

`f` .... Field element.
Return........ Pointer to the GAP representtion of `f`.

Description This function takes a field element and returns the GAP representation of this element. The return value is a pointer to a static buffer which is overwritten on each call.

`zftoi()` **Convert field element to integer**

```
long zftoi(FEL f);
```

`f` .... The field element.
Return........ The number corresponding to `f`.

Description This function converts a field element to an integer, using a 'canonical' representation of field elements as integers which may be different from the internal representation. In particular, the prime field is mapped on $\{0, \ldots, p-1\}$. See section 3.1.3 for details. `zftoi()` should be used whenever field elements are converted with `printf()`.

See also `zitof()`

`zgen` **Field generator**

```
extern FEL zgen;
```

Description This variable contains a genrator for the multiplicative group of the current field.

See also `zfl` `zgen`

`zinsert()` **Insert a mark into a row**

```
void zinsert(PTR row, long col, FEL mark);
```

`row` .... Pointer to the row.
`col` .... Insert positions.
`mark` .... Value to insert.

Description `zinsert()` inserts the field element `mark` at position `col` into `row`. Before this function can be used, a field must be selected with `zsetlen()`. Once a field is selected, `zinsert()` works with any row, regardless of the current row size. So, if you are working with two rows of different size, you do not have to call `zsetlen()` prior to each `zinsert()`. On the other hand, this means that there is no protection against writing beyond the end of a row. The caller must assure that `row` points to a row of at least `col` columns. Otherwise the program may crash or produce undefined results.

See also `zextract()`

`zinv()` **Finite field inversion**

```
FEL zinv(FEL a);
```

`a` .... Field element.
Return........ 1/`a`

Description This function returns the multiplicative inverse of a field element. Before you can use this function, the field must have been selected with `zsetfield()`. The argument is not checked. If `a` is not a valid field element or zero the result is undefined and the program may crash.

`zinv()` may be implemented as a macro. In this case, it is guaranteed that the argument is evaluated exactly once.

See also `zadd()` `zdiv()` `zmul()` `zneg()` `zsub()`

`zitof()` — Convert integer to field element

```
FEL zitof(long l);
```

`l` .... The integer.
Return........ The field element corresponding to `l`.

Description This function converts an integer to a field element using the same mapping as explained with `zftoi()`. Both functions are inverse in the sense that the expression `f == zitof(zftoi(f))` is always true. `zitof()` should be used whenever field elements are converted with `scanf()`.

See also `zftoi()`

`zmaprow()` — Multiply a vector by a matrix

```
PTR zmaprow(PTR row, PTR matrix, long nor, PTR result);
```

`row` .... The vector.
`matrix` .... The matrix.
`nor` .... Number of rows in `matrix`.
`result` .... Pointer to a buffer where the result will be stored.
Return........ The function always returns `result`.

Description This function multiplies the vector `row` from the right by the matrix `mat` and stores the result into `result`. `nor` is the number of rows of the matrix which must coincide with the size (number of columns) of the vector. The number of columns in both `mat` and `result` is determined by the current row size.

Note: `result` and `row` must not overlap. Otherwise the result is undefined.

`zmatinv()` — Matrix inversion

```
int zmatinv(PTR mat, PTR result);
```

`mat` .... Pointer to the matrix.
`result` .... Pointer to result buffer.
Return........ The function returns 0 on success and -1 on error.

Description This function calculates the inverse of a square matrix. This is a is a low-level, destructive matrix inversion which, unlike `matinv()`, does not allocate a temporary matrix.

Both `mat` and `result` must be square matrices, and the row size must be set by the caller. `result` need not be initialized. A return value of 0 indicates that the function was successfull and the inverse matrix is now in `result`. If the matrix is singular, the function returns -1. In any case the original matrix is destroyed.

See also `MatInv()`

`zmkechelon()` — Reduce to echelon form

```
long zmkechelon(PTR matrix, long nor, long *piv);
```

`matrix` .... Matrix to reduce.
`nor` .... Number of rows.
`piv` .... Pointer to pivot table buffer.
Return........ This function returns the rank of `matrix`.

Description This function converts a matrix to semi-echelon form and builds a pivot table suitable for use with `zcleanrow()`. A matrix is said to be in

- *semi-echelon form* if each row contains a first nonzero element (pivot element), and all entries below this element are zero.
- *normalized semi-echelon form* if it is in semi-echelon form and each pivot element is equal to 1.

In the MeatAxe, matrices are often stored in semi-echelon form. Many functions and programs expect their arguments to be in semi-echelon form.

`matrix` is a matrix of `nor` rows. The number of columns must have been set with `zsetlen()`. `piv` must be a pointer to an array of at least `nor+1` long integers. On return, `matrix` is in semi-echelon form. Zero rows are moved to the end, and the number of non-zero rows (i.e., the rank) is stored in `piv[0]`. The rest of the `piv` array contains the pivot table, beginning with `piv[1]` for the first row. The return value is the rank.

See also `zmkpivot()`

`zmkpivot()` — Build pivot table

```
int zmkpivot(PTR matrix, long nor, long *piv);
```

`matrix` .... Pointer to the matrix.
`nor` .... Number of rows.
`piv` .... Pointer to pivot table buffer.
Return........ The return value is 0 on success, or -1 on error.

Description This function is used to find the pivot elements of a matrix which is already in semi-echelon form (see `zmkechelon()`). `nor` is the number of rows in `matrix`. The number of columns must have been set before with `zsetlen()`. `piv` must be a pointer to an array of at least $|nor| + 1$ long integers. The pivot positions are stored here, beginning with `piv[1]` for the first row.

The pivot table is used by some functions like `zcleanrow()` which perform Gaussian elimination.

See also `zmkechelon()`

zmoverow() — **Copy a row**

```
void zmoverow(PTR dest, PTR src);
```

`dest` .... Pointer to the destination.
`src` .... Pointer to the source.

Description This function copies the contents of one row to another row. As with all row operations, the row length must have been set before with `zsetlen()`.

See also `zswaprow()`

zmul() — **Finite field multiplication**

```
FEL zmul(FEL a, FEL b);
```

`a` .... First field element.
`b` .... Second field element.
Return........ `a · b`

Description This function returns the product of two field elements. Before you can use this function, the field must have been selected with `zsetfield()`. The arguments are not checked. If either argument is not in the allowed range the result is undefined and the program may crash.

`zmul()` may be implemented as a macro. In this case, it is guaranteed that both arguments are evaluated exactly once.

See also `zadd()` `zdiv()` `zinv()` `zneg()` `zsub()`

zmulrow() — **Multiply a row by a coefficient**

```
PTR zmulrow(PTR row, FEL mark);
```

`row` .... Pointer to the row.
`mark` .... Coefficient.
Return........ The function always returns `row`.

Description This function multiplies `row` by the scalar `mark` using the current row size and returns `row`.

Multiplying a row with zero `F_ZERO` initializes the memory.

See also `zaddmulrow()` `zaddrow()`

zneg() — **Additive inverse**

```
FEL zneg(FEL a);
```

`a` .... Field element.
Return........ `−a`

Description This function returns the additive inverse of a field element. Before you can use this function, the field must have been selected with `zsetfield()`. The argument is not checked. If `a` is not a valid field element the result is undefined and the program may crash.

`zneg()` may be implemented as a macro. In this case, it is guaranteed that the argument is evaluated exactly once.

See also `zadd()` `zdiv()` `zinv()` `zmul()` `zsub()`

`znoc` **Current row size**

```
extern long znoc;
```

Description This variable contains the current number of columns used for all row operations. `znoc` is updated automatically when the row size is changed with `zsetlen()`.

See also `zsetlen()` `zrowsize`

`znullsp()` **Null space**

```
long znullsp(PTR matrix, long nor, long *piv, PTR nsp);
```

`matrix` .... Pointer to the matrix.
`nor` .... Number of rows.
`piv` .... Pointer to pivot table.
`nsp` .... Pointer to result.
Return........ The function returns the dimension of the null-space, or -1 on error.

Description This function calculates the null-space of a matrix. The matrix is passed as first argument, `nor` is the number of rows, `piv` must be a pointer to an array of at least $|nor| + 1$ integers, and `nsp` must be a pointer to a square matrix of size `nor`.

If the function is successfull (non-negative return value),

- `matrix` is reduced to echelon form
- `nsp` contains the null-space in echelon form
- `piv` contains a pivot table for the null space, and the null-space dimension is duplicated in `piv[0]`.

`zpermrow()` **Multiply a vector by a permutation**

```
PTR zpermrow(PTR row, long *perm, PTR result);
```

Return........ The function returns `result`.

Description This function multiplies the vector `row` from the right with the permutation `perm` and stores the result into `result`. Multiplication of vectors by permutations is defined as follows: If the permutation maps i to k, then the i-th mark of the vector is stored in the k-th position of the result.

Note: `result` and `row` must not overlap. Otherwise the result is undefined.

zpseed_free() — Clean up the seed vector generator

```
void zpseed_free();
```

Description This function frees all resources used internally by the generator.

See also zpseed_init() zpseed_make() zpseed_next() zpseed_vec()

zpseed_init() — Initialize the seed vector generator

```
int zpseed_init(long sdim, PTR seedbasis);
```

sdim..........Dimension of seed space.
seedbasis....Basis for the seed space.
Return........The function returns 0 on success, and −1 on error.

Description This function initializes the seed vector generator. It must be called at least once before using zpseed_make() or zpseed_next(). seedbasis is a basis for the seed space, i.e., a matrix, and sdim is the number of basis vectors. Remember that the row size must have been set before with zsetlen().

seedbasis need not be in semi-echelon form. It is not checked that the basis vectors are linearly independent. If they are not, you will get seed vectors which are multiples of each other or different numbers will produce the same vector.

There is no internal buffer for the basis. zpseed_init() just saves a pointer to the basis for internal use. Thus, you must not modify or free the basis as long as you use the seed vector generator.

See also zpseed_free() zpseed_make() zpseed_next() zpseed_vec()

zpseed_make() — Make a seed vector

```
long zpseed_make(long num);
```

num...........Seed vector number.
Return........Seed vector number, or −1 on error.

Description This function makes a seed vector given its number. The seed vector generator must have been initialized with zpseed_init(). The seed vector is stored in zpseed_vec and its number is returned. A return value of -1 indicates an error.

zpseed_make() will not reject a number which corresponds to a non-normalized seed vector. See zpseed_init() for details.

See also zpseed_free() zpseed_init() zpseed_next() zpseed_vec()

zpseed_next() — Make the next seed vector

```
long zpseed_next();
```

Return........Seed vector number, −1 on error.

Description This function calculates the next normalized seed vector and stores it in `zpseed_vec`. Successive calls of `zpseed_next()` will step through all normalized seed vectors. If the seed space is exhausted, `zpseed_next()` returns -1.

If you call `zpseed_make(N)`, `zpseed_next()` will calculate with the next normalized seed vector after N. Seed `zpseed_init()` for details.

See also `zpseed_free()` `zpseed_init()` `zpseed_make()` `zpseed_vec()`

`zpseed_vec()` **Seed vector buffer**

```
extern PTR zpseed_vec;
```

Description `zpseed_vec` is maintained by the seed vector generator. After a successful call of `zpseed_make()` or `zpseed_next()`, the seed vector is stored in this variable.

See also `zpseed_free()` `zpseed_init()` `zpseed_make()` `zpseed_next()`

`zquot()` **Projection on the quotient**

```
int zquot(PTR space, long dim, PTR quot)
```

`space`......... Pointer to the list of vectors to project.
`dim`........... Number of vetors in `space`.
Return........ 0 on success, -1 on error.

Description Given two subspaces $U, W \leq V$, this function calculates the projection of U on the quotient V/W. Before you can use `zquot()`, the subspace W must be specified with `zquotinit()`.

The projection depends on the basis for the subspace W. It is calculated in the following way: Let $V = F^{n\times n}$ and $w_1, \ldots, w_s$ be a basis for the subspace $W \leq V$. The basis, written as a matrix of row vectors, is assumed to be in semi-echelon form. By looking at the pivot columns we can construct the vectors $w_{s+1}, \ldots, w_n$ by taking all vectors which have a exactly one 1 at any non-pivot position and are zero otherwise. Then, $(w_1, \ldots, w_s, w_{s+1}, \ldots, w_n)$ is a basis for V in semi-echelon form and defines the decomposition of any vector into subspace and quotient part.

The result is stored in `quot`, which must be a matrix with `dim` rows and $n - s$ columns. On return, the i-th row of `quot` contains the projection of the i-th row of `space`. The result is not in echelon form and some of the rows may be zero.

See also `zquotinit()` `zquotop()`

`zquotinit()` **Prepare for projection on the quotient**

```
int zquotinit(PTR subspace, long dim, long *piv);
```

`subspace`..... Pointer to the list of vectors to project.
`dim`........... Number of vetors in `space`.
`piv`........... Pointer to pivot table.
Return........ 0 on success, -1 on error.

Description This function is used to specify the subspace used by the quotient projection functions, `zquot()` and `zquotop()`. You must always call `zquot_init()` before `zquot()` or `zquotop()`. `subspace` is a pointer to a basis for the subspace, which must be in semi-echelon form, and `dim` is the dimension of the subspace. `piv` must be a pointer to a pivot table for the basis. The pivot table can be calculated with `zmkpiv()`.

See also `zquot()` `zquotop()`

`zquotop()` **Operation on the quotient**

```
int zquotop(PTR matrix, PTR quot);
```

`matrix` .... Pointer to the matrix
`quot` .... Pointer to a buffer for the rsult.
Return........ 0 on success, −1 on error.

Description After a subspace has been specified with `zquotinit()`, this function calculates the operation of any matrix on the quotient. Of course, this makes sense only if the subspace is invariant under the matrix, but this is not checked. If the subspace is not invariant, the results are undefined.

`matrix` must be a pointer to a square matrix with the same number of columns as the subspace. The result is a square matrix with the dimension of the quotient. This matrix is stored in `quot`.

See also `zquot()` `zquotinit()`

`zreadhdr()` **Open file and read header**

```
FILE *zreadhdr(const char *name, long *field, long *nrows, long *ncols);
```

`name` .... File name.
`field`......... Pointer to buffer for first header entry (usually the field order).
`nor` .... Pointer to buffer for second header entry (usually the number of rows).
`noc` .... Pointer to buffer for third header entry (usually the number of columns).
Return........ Pointer to open file, or `NULL` on error.

Description This function opens a data file for input and reads the file header (3 long integers). The exact meaning of the header values three numbers depends on the file type. For a matrix they are 'field order', 'number of rows' and 'number of columns'.

See also `zwritehdr()`

`zreadlong()` **Read long integers**

```
size_t zreadlong(FILE *f, long* buf, size_t n);
```

`f` .... File to read from.
`buf` .... Pointer to buffer.
`n` .... Number of integers to read.
Return........ Number of integers that were actually read. A return value different from `n` indicates an error.

Description This function reads `n` long integers from the file `f` into the array `buf`. `buf` must point to a memory area of at least `n*sizeof(long)` bytes and `f` must be open for reading.

This function expects that the numbers in the file are 4-byte integers in 'little-endian' format, i.e. the least significant byte first. Using a machine-independent data format makes MeatAxe data files more portable, but there are also some disadvantages:

1. The conversion to and from machine-independent format involves several arithmetic operations for each number read/written.
2. The highest number which can be read/written is $2^{32} - 1$, even on 64 bit machines.

See also `zwritelong()`

`zreadvec()` **Read rows**

```
size_t zreadvec(FILE *f, PTR buf, size_t n);
```

`f` .... Pointer to File.
`buf` .... Pointer to data buffer.
`n` .... Number of rows to read.
Return........ The function returns the number of rows that were successully read. A return value different from `n` indicates an error.

Description This function reads 1 or more rows from a file. The row size must have been set before.

See also `zseek()` `zwritevec()`

`zrestrict()` **Restrict to a subfield**

```
FEL zrestrict(FEL a, long subfield);
```

`a` .... Field element.
`subfield` ..... Subfield order.
Return........ The restricted field element.

Description This function restricts a field element to a subfield. The return value represents the same element as `a` but with respect to the subfield. Note that you cannot use the return value of `zrestrict()` (for field arithmetic, say) until you change to the subfield with `zsetlen(subfield,...)`.

`a` must be of the form `zembed(b,subfield)`. Otherwise the result is undefined. For example, restricting from GF(8) to GF(2) works only with the prime field elements, i.e., 0 and 1.

See also `zembed()`

`zrowsize` **Current row size**

```
extern size_t zrowsize;
extern size_t zrowsize_io;
```

Description `zrowsize` is the current row size, i.e., the number of bytes occupied by one row. This is always a multiple of `sizeof(long)`. `zrowsize_io` contains the 'true' size of a row, i.e., the number of bytes that are actually used. This is also the row size in data files.

See also `znoc` `zsetlen()`

`zseek()` **Open file and read header**

`int zseek(FILE *f, long pos);`

`f` .... Pointer to File.
`pos` .... Row number (1-based).
Return........ The function returns 0 on success and −1 on error.

Description This function sets the read/write pointer of file `f` to position `pos`. I.e., the next `zreadvec()` or `zwritevec()` will start at row `pos`. Note that row numbers start with 1, not 0. If `pos` is different from 1, the row size must have been set before with `zsetlen()`.

You should always use `zseek()` rather than `fseek()` because `zseek()` knows about MeatAxe file headers and adjusts the file pointer appropriately.

See also `zreadvec()` `zwritevec()`

`zsetfield()` **Set the field**

`int zsetfield(long field);`

`field`........ Field order.
Return........ The function returns 0 on success, and −1 otherwise.

Description This function sets the current field to GF(`field`) and initializes the field arithmetic. Most kernel functions require that a field has been selected before they are used.

`zsetfield()'` reads an arithmetic table file into memory. If the file does not exist, `zsetlen()` tries to generate one by executing the `maketab` program.

See also `zsetlen()`

`zsetlen()` **Set the row length**

`int zsetlen(long ncols);`

`ncols`........ Number of columns per row.
Return........ The function returns 0 on success, and −1 otherwise.

Description This function sets the current row size to `ncols`. The row size is used for low-level row operations, e.g. `zaddrow()`.

See also `zsetfield()`

`zsize()` — Size of a row or matrix

```
size_t zsize(long nrows);
```

`nrows` .... Number of rows.

Description This function returns the number of bytes occupied by `nrows` rows of the current row size. This is the number of bytes that `zalloc(nrows)` would allocate. The row size must have been specified previously with `zsetlen()`.

See also `zalloc()`

`zsub()` — Finite field subtraction

```
FEL zsub(FEL a, FEL b);
```

`a` .... First field element.
`b` .... Second field element.
Return .... `a` − `b`

Description This function returns the difference of two field elements. Before you can use this function, the field must have been selected with `zsetfield()`. The arguments are not checked. If either argument is not in the allowed range the result is undefined and the program may crash.

`zsub()` may be implemented as a macro. In this case, it is guaranteed that both arguments are evaluated exactly once.

See also `zadd()` `zdiv()` `zinv()` `zmul()` `zneg()`

`zswaprow()` — Exchange two rows

```
void zswaprow(PTR dest, PTR src);
```

`dest` .... Pointer to the first row
`src` .... Pointer to the second row

Description This function exchanges the contents of two rows. As with all row operations, the row length must have been set before with `zsetlen()`.

See also `zmoverow()`

`zwritehdr()` — Open file and write header

```
FILE *zwritehdr(const char *name, long field, long nrows, long ncols):
```

`name` .... File name.
`field` .... First header entry (usually the field order).
`nor` .... Second header entry (usually the number of rows).
`noc` .... Third header entry (usually the number of columns).
Return .... Pointer to open file, or `NULL` on error.

Description This function opens a data file for input and writes the the file header. If the file does not exist, a new file is created. If the file exists it is overwritten.

See also `zreadhdr()`

`zwritelong()` **Write long integers**

`size_t zwritelong(FILE *f, const long *buf, size_t n);`

`f` .... File to write to.
`buf` .... Pointer to buffer.
`n` .... Number of integers to write.
Return........ Number of integers that were actually written. A return value different from `n` indicates an error.

Description This function writes `n` long integers from the the array `buf` to the file `f`. `buf` must point to a memory area of at least `n*sizeof(long)` bytes and `f` must be open for writing. The numbers are written in a machine-independent format which can be read by `zreadlong()`.

See also `zreadlong()`

`zwritevec()` **Write rows**

`size_t zwritevec(FILE *f, PTR  buf, size_t n);`

`f` .... Pointer to File.
`buf` .... Pointer to data buffer.
`n` .... Number of rows to write.
Return........ The function returns the number of rows that were successully written. A return value different from `n` indicates an error.

Description This function writes 1 or more rows to a file. The row size must have been set before.

See also `zseek()` `zreadvec()`

Index