

Vector Enumeration Programs, version 3.04

Steve Linton

Vector Enumeration

This is the manual for version 3.04 of my vector enumeration programs `me`, `qme` and `zme`. These programs are distributed free of charge for non-commercial use, and there is NO WARRANTY WHATSOEVER.

This document does not describe the underlying mathematics, which are described in the papers referred to later, but describes the operation of the programs.

1 Introduction

The vector enumeration algorithm for fields is described in the papers *Constructing Matrix Representations of Finitely Presented Groups* (J. Symbolic Computation (12) 1991) and *On Vector Enumeration* (to appear in the Journal of Linear Algebra and Applications). The **me** and **qme** programs implement this algorithm, **me** for prime fields of order less than 256 and **qme** for the rational numbers. The **zme** program implements an extended algorithm which works over principle ideal rings, for the special case of the integers. The extended algorithm will be described in a future publication.

This document describes the installation and use of these three programs, the formats of input and output files and the various options which control the execution. Some notes on the structure of the programs is also included.

In *On Vector Enumeration*, an application of the algorithm to constructing quotients of algebra representations is described. These programs include a special option **-Q** to read an input format suited to that application and proceed accordingly.

1.1 What's New

1.1.1 What's New in Version 3.04

There are a number of changes in version 3.04. Several bugs were fixed, in particular one relating to meat-axe format output. Thanks to Thomas Breuer for spotting this one.

The **-L** command line option was added at this release, to allow logging information to be preceded by a comment designator.

The runtime in milliseconds was added to the output in several of the formats. This change should not break any reasonable program using the output.

Finally, a new '**Makefile**' is added, suitable for most **make** programs. The old '**Makefile**', which retains some advantages for development, is renamed '**GNUmakefile**'.

1.1.2 What's New in Versions 3.02 and 3.03

These versions (3.02 was never released) contain bug fixes relating to the integer vector enumeration program **zme** and changes to some undocumented features relating to the forthcoming GAP interface.

The **zme** bugs could cause a crash or improper non-termination.

1.1.3 What's New in Version 3.01

This version is a bug-fix version. The main bug fixed at this release is that under certain circumstances information could be lost during look-ahead, resulting in the output module being a proper pre-image of the correct module.

The fix for this problem involves allowing some definitions to be made, even during look-ahead. When the `-t` option see Section 4.6 [Limit Options], page 22 is used to limit the total number of dimensions, a proportion of the limit is allocated for these critical definitions. This proportion can be changed by a new form of the `-t` option.

1.1.4 What's New in Version 3

- The **zme** program is completely new in this version.
- The **quot** quotient-constructing program has been merged with **me** and corresponding functionality added to **qme** and **zme**. See Section 4.4 [Input File Options], page 20 and Section 3.2 [Input for Quotient Construction], page 14.
- A number of internal changes have improved performance for most examples, sometimes dramatically.
- The input format has been enriched in a number of ways see Chapter 3 [Input Formats], page 9 and the error handling and reporting in the parser has been greatly enhanced see Section 4.3 [Logging and Error Message Options], page 17.
- Three new output formats have been added: one for output to be read into AXIOM and two formats intended to be simply and quickly read by special-purpose programs. See Chapter 5 [Output Formats], page 24 and Section 4.2 [Output Options], page 16.
- A special option has been added to indicate that all the generators commute with one another, so that the module constructed will be a quotient of a polynomial ring. See Section 4.4 [Input File Options], page 20.

- All the programs have been combined into a single set of sources. See Chapter 7 [The Source Code], page 31.
- A number of new options have been added to control execution and limit the module dimension and the run time. See Section 4.6 [Limit Options], page 22
- A new option has been added to override the characteristic given in the presentation file. See Section 4.4 [Input File Options], page 20.
- A new options has been added to suppress the automatically generated relations for non-invertible generators.
- Early-closing is now disabled by default, in the interests of mathematical accuracy. It can be re-enabled using the `-e` option. See Section 4.1 [Strategic Options], page 15.

2 Installation

2.1 Installation Procedure

Since you're reading this manual you have probably already obtained the vector enumeration package and unpacked it from the compressed distribution. Just for completeness, or for getting a later version, the software is available from me, Steve Linton (sal@cs.stand.ac.uk), or by anonymous FTP from galois.maths.qmw.ac.uk. It consists of just one file `nme.tar.Z` (there may also be `nme.tar.z` and/or `nme.zoo`). The file `nme.README` is a copy of the `README` file contained in the distribution.

You should create a new directory *the nme home directory* into which to unpack the distribution which will then create subdirectories `./ve`, `./rat`, `./int`, `./docs` and `./examples`. The first three contain only `GNUmakefile`'s, `./docs` contains this TeXInfo file and `./examples` contains some input files for testing. The current directory contains the C and Bison source files, the master `Makefile` and `GNUmakefile` and the shell script `build.sh`.

There are three alternative ways of compiling the programs. Simple compilation should suffice for most users.

2.1.1 Simple Compilation

Users not intending to modify the programs or to compile them for multiple architectures in the same directory hierarchy should use simple compilation. Simply change to the nme home directory and:

1. Edit the shell script `build.sh` to:
 - Set the correct name for the C compiler (probably `cc` or `gcc`).
 - Set the compiler, pre-processor and linker flags so that the compiler will find the GMP libraries and include files see Section 2.2 [Pre-requisites], page 8 and will suitably optimize the programs.
 - Include the C version of `alloca` if it is not provided by your system (most BSD systems provide it, most System V systems do not).
 - Comment out appropriate sections of the script if you do not need all of the programs `me`, `qme` and `zme`. If you need only `me`, then you needn't worry about GMP.
2. Type `./build.sh`.

3. Cross your fingers.

Assuming everything works, you will end up with the three executable programs in the ‘bin’ subdirectory of the nme home directory. They may be moved to any convenient location.

2.1.2 General Installation

If you plan to develop enhancements to the programs, or wish to compile them for multiple architectures then you will need to install the package completely. You can do this either using the ‘Makefile’, mainly prepared for when the package is installed as a GAP share library, or using the ‘GNUmakefile’. For the latter you will need version 3.63 or later of GNU make. It will also be simpler if you have a C compiler with the `-M` option to prepare dependency lists automatically.

General installation and simple installation should not be done in the same directory, as they use subdirectories differently.

2.1.2.1 Using GNU Make

For general installation with GNU make you will need to arrange to have an environment variable `ARCH` set to some convenient name for the current hardware architecture. I do this with a line

```
setenv ARCH 'arch'
```

in my ‘.cshrc’ file, and a small shell script ‘arch’ that uses the hostname to decide what type of computer I am using. On Sun systems ‘arch’ is a system command.

You should then create a subdirectory ‘bin’ of the nme home directory. For each architecture xxx that is likely to be of interest you should also create subdirectories xxx of the ‘bin’, ‘ve’, ‘rat’ and ‘int’ subdirectories of the nme home directory.

You should then edit the file ‘Makefile.inc’ in the nme home directory and set the C compiler, and compiler, linker and pre-processor flags suitably.

Assuming that your command to call GNU make is `gmake`, running `gmake` in the nme home directory should make all three programs, while running it in ‘ve’ will make `me`, in ‘rat’ will make `qme` and running it in ‘int’ will make `zme`. The binaries will be stored in ‘bin/\$ARCH’, which can

conveniently be included in your path, and separate directories will be used for the object files for each architecture, so that they can be kept independently up to date.

2.1.2.2 Using the GAP Makefile

To use Martin Schoenert's ingenious Makefile to build the package, edit `'Makefile'` in the nme home directory and set the variables `INCDIRGMP` and `LIBDIRGMP` appropriately for your system. Then type `make cc` or `make gcc` according to which compiler you wish to use. By default this Makefile only makes `me` and `qme`, as `zme` is not used in the GAP share library. To make `zme` as well, edit the dependencies of the target `all`.

This will make the executables as `'bin/me.exe'`, `'bin/qme.exe'` and (if you edited the makefile appropriately) `'bin/zme.exe'`. These will be invoked by the scripts which are supplied as `'bin/me'`, `'bin/qme'` and `'bin/zme'`. If you wish to install the package for use on multiple hardware architectures, you should move the executables to (for example) `'bin/me.sun'` and replace the script with one that decides which executable to use.

2.1.3 Compile-time options

There are three compile-time options which can be controlled by editing the file `'me.h'` in the distribution. They are

DEBUG	When the symbol <code>DEBUG</code> is defined many additional consistency checks and validations are included in the code. Also, the memory allocation routines in <code>'myalloc.c'</code> , which check for freeing pointers not allocated, overwrite freed memory to prevent access to it and so on, are used. Setting <code>DEBUG</code> slows the program down substantially and increases the size of the compiled code.
SCRUT	This controls compilation of the scrutinize feature. If <code>SCRUT</code> is selected and enabled with the <code>-s</code> option on the command line then the calculation performed will be checked frequently against a correct answer read in. See Section 4.5 [Debugging Options], page 21. If <code>SCRUT</code> is specified at compile-time but not enabled from the command line there is a small loss of performance as the checks must be bypassed at run-time.
LOGGING	This controls compilation of the code to produce logging and diagnostic messages. If <code>LOGGING</code> is defined at compile-time, then messages reporting progress will be output, under the control of the options described in Section 4.3 [Logging and Error Message Options], page 17. If logging messages are not needed a small gain in speed, and a rather larger reduction in code size can be achieved by not defining <code>LOGGING</code> at compile-time.

By default, `LOGGING` is defined, but `DEBUG` and `SCRUT` are not.

2.2 Pre-requisites

The `me` program is self-contained, but `qme` and `zme` require the GMP multiple-precision arithmetic library. Specifically, they require version 1.2.99 or later of the library. This is available by anonymous FTP from `sics.se`.

The makefiles supplied with the programs make use of non-standard features of GNU make version 3.63 and will not work with most other `make` programs or with earlier versions of GNU make. Scripts are supplied to build the programs from scratch see Section 2.1.1 [Simple Compilation], page 5, but anyone wishing to work on these programs will probably need GNU make see Section 2.1.2.2 [General Installation], page 7.

The input parser ‘`input.tab.c`’ is prepared using the GNU parser generator `bison` from the grammar file ‘`input.y`’. If you wish to change the parser in any way you will need to obtain `bison`. The supplied parser was prepared with `bison` version 1.18.

In order to format and read this manual you will need version 2.16 of the TeXInfo package and either TeX or GNU emacs. Alternatively contact me and I will send you a DVI or PostScript file.

3 Input Formats

3.1 A Presentation File

The basic format of the presentation file is

characteristic . generators . generators not known to be invertible . generators not known to be involutions . submodule generators . relations .

Anything after the final `.` is not read in by the program. Accordingly comments may appear here.

3.1.1 Characteristic

The characteristic is a non-negative integer and must always be present. For `me` it must be a prime number less than 256, and indicates the characteristic of the prime field over which the calculation will be performed. For `qme` or `zme` the characteristic must be 0.

The ‘`-C`’ option may be used to override the characteristic specified in the presentation file (which must still be given, but will not be checked). See Section 4.4 [Input File Options], page 20.

3.1.2 Generators

The *generators* part of the presentation defines the names which will be used for the generators of the algebra being constructed. These names must each be of one of two types:

- Single capital letters such as ‘`A`’.
- Strings of one or more lower-case letters such as ‘`a`’ or ‘`xyz`’.

Lower-case generator names must be separated by a comma, a semi-colon or whitespace.

The following specifications all define three generators ‘`A`’, ‘`B`’ and ‘`C`’:

```
ABC
A,B,C
```

```
A B;C,
AB C
```

On the other hand, the specification ‘`abc`’ defines one generator called ‘`abc`’. To define three generators ‘`a`’, ‘`b`’ and ‘`c`’ a specification such as ‘`a b c`’, ‘`a,b,c`’ or ‘`a b;c`’ is needed.

Once the generators are defined a longest-match scanner is constructed to read strings of generator names. This is then used to read the *generators not known to be invertible* and *generators not known to be involutions* sections of the presentation (and also used while reading the submodule generators and relations).

Providing no generator is an initial substring of another (which is very bad practice) no separators should be needed in *generators not known to be invertible* or *generators not known to be involutions*, although they might add readability.

By default every generator is assumed to be an involution, that is for a generator x , the relation $xx = 1$ is assumed. This assumption can be overridden by including the generator in one of the lists *generators not known to be invertible* or *generators not known to be involutions*. If a generator x is included in the first list then the program will make no assumptions about a multiplicative inverse of x . If a generator x is included in the second list then the program will add an extra generator x^{-1} and assume the relation $xx^{-1} = 1$.

It is an error for a generator to appear in both lists.

Either of these lists may have the special form ‘`*`’, which indicates that all generators are non-invertible or not involutions.

Thus, for example a module for the free algebra on generators ‘`a`’, ‘`b`’ and ‘`c`’ would be specified by:

```
characteristic. a,b,c.abc...submodule generators.relations.
```

or

```
characteristic. a,b,c.*...submodule generators.relations.
```

while a module for the free group on the same generators would be given by:

```
characteristic. a,b,c.*.*.submodule generators.relations.
```

and a module for the free product of three copies of the cyclic group of order two by:

characteristic. a,b,c...submodule generators.relations.

3.1.3 Relations

The *relations* section of the presentation contains the relations of the algebra to be constructed (apart from any relations implied by the generators being invertible or involutions). It may be empty; if not then it consists of two parts, separated by a colon.

3.1.3.1 Group Type Relations

The first part contains “group type” relators. These are products of invertible generators which are equal to 1 in the algebra to be constructed. These can be handled more efficiently than general (algebra) relations, so relations that are of this kind should be given in this section. It is an error for non-invertible generators to appear in this section.

The following constructions are available to write these products (called *group words*)

- $w1$ [*whitespace*] $w2$ denotes the product of group word $w1$ and group word $w2$. The *whitespace* is only needed if there is ambiguity in parsing the generator names. For example if ‘a’, ‘b’ and ‘ab’ were all generators then ‘ab’ would represent the generator ‘ab’, while ‘a b’ represented the product of ‘a’ and ‘b’.
- $w1*w2$ is equivalent to the product $w1w2$.
- $w\sim$ denotes the inverse of group word w .
- w^n denotes the n th power of group word w . n must be a positive integer.
- $w1\wedge w2$ denotes the conjugate of group word $w1$ by $w2$.
- $[w1,w2]$ denotes the commutator of group words $w1$ and $w2$.

Conjugation, inversion and powering bind more tightly than multiplication, but parentheses may be used. Thus ‘AB3’ denotes the word ‘ABBB’, while ‘(AB)3’ denotes ‘ABABAB’.

3.1.3.2 Algebra Relations

The second part of the relations section of the presentation contains more general relations. These may be either elements of the free algebra generated by the generators, which are equal to 1

in the algebra being constructed, or equations between elements of the free algebra which are true in the algebra being constructed.

Any group word (see above) may be given as an element of the free algebra, as can an element of the underlying field given as an integer (or quotient of integers for `qme`). Elements of the free algebra can be combined by addition with '+', subtraction ('-') or multiplication ('*'). The usual rules of precedence apply and parentheses may be used to override them.

3.1.3.3 Weight Specifications

Any relator or relation in either part of the relations section may be followed by a weight specification. This takes the form '`<w>`' or '`<w1,w2>`' and affects the order in which the relations are used by the program, higher weights meaning that the relation will be used less or later. See Chapter 6 [Strategy], page 29 for details of how the weights are applied.

In the first form '`w`' should be a positive integer and will be the weight used for that relation in both define and lookahead modes. In the second form '`w1`' and '`w2`' should both be positive integers. '`w1`' will be the weight used in define mode and '`w2`' in lookahead mode. For details of lookahead See Chapter 6 [Strategy], page 29.

3.1.4 Submodule Generators

The submodule generator section may be of one of two forms. The simpler form may only be used when the module to be constructed is cyclic (more accurately when its presentation has just one generator). In this case the submodule generator section may take the same form as the relations section, consisting of group-type relators and algebra relations. Weight specifications may be present, but will be ignored.

Note that in this case the submodule generators will actually be forced to fix a vector in the module constructed, rather than annihilate it. Mathematically, the "real" submodule generators are obtained from those input by subtracting 1.

If the module is not cyclic then the more general form must be used. This begins with the number s of module generators enclosed in curly brackets '{ }' followed by zero or more submodule generator entries. Each such entry may be of one of three forms:

3.1.4.1 Submodule Generators in Packed Form

A submodule generator in *packed form* consists of an s -tuple of elements of the free algebra, each given in the format specified in Section 3.1.3.2 [Algebra Relations], page 11, enclosed in parentheses and separated by commas.

Thus for example, the following presentation specifies the representation of the symmetric group S_3 obtained by identifying the fixed vectors of two copies of the natural permutation representation.

$0.AB..B.\{2\}(A-1,0),(0,A-1),(1+B+B2,-1-B-B2).B3,(AB)2:.$

3.1.4.2 Submodule Generators in Sparse Form

For brevity, when there are many submodule generators, a generator may be given in *sparse form*. This consists of one or more pairs:

module generator number , *free algebra element*

enclosed in square brackets. The *module generator number* must be between 1 and s , and the pairs within a sparse form submodule generator must be given in order of *module generator number*. Thus the first submodule generator in the above example could be rewritten ‘ $[1,A-1]$ ’.

3.1.4.3 Universal Submodule Generators

Finally, a series of submodule generators such as the first two in the example above may be abbreviated as a *universal submodule generator*. This takes the form ‘ $[first-last,x]$ ’, where x is a free algebra element in the usual form, and represents the $last - first + 1$ separate submodule generators ‘ $[first,x]$ ’ through ‘ $[last,x]$ ’ inclusive. Equivalently, the universal submodule generator ‘ $[first-last,x]$ ’ can be read as indicating that the free algebra element x annihilates module generators $first$ through $last$. The special case ‘ $[1-s,x]$ ’ can be further abbreviated to ‘ $[*,x]$ ’.

Thus the example can be abbreviated to

$0.AB..B.\{2\}[* ,A-1] , (1+B+B2,-1-B-B2).B3,(AB)2:.$

3.2 Input for Quotient Construction

When the `-Q` option is given to `me`, `qme` or `zme`, the extension for input files changes from `‘.pres’` to `‘.qin’` and a completely different format of input is expected.

The input essentially specifies two things:

- A vector space V and a set of matrices, which together specify a representation of a free algebra.
- A set of vectors in V which generate a sub-module W under the action of the algebra. That is a set of vectors whose (iterated) images under the matrices span a subspace W of V .

This is achieved by an input file consisting six parts separated by whitespace:

1. a specification of the characteristic as described in Section 3.1.1 [Characteristic], page 9;
2. the dimension of the vector space V as an unsigned decimal integer;
3. a string giving the generator names (used for printing and in some of the output formats). The names must be single characters, unseparated and terminated by whitespace;
4. an unsigned integer giving the number of generators of W ;
5. the vectors which generate W ;
6. the matrices giving the action of the free algebra on V . For each basis vector b of V all the images of b under the matrices (ie the appropriate rows of all the matrices) are given, followed by all the images of the next basis vector.

The last two sections involve giving vectors as part of the input. These are normally given in a sparse format, consisting of zero or more pairs `‘ i , $x(i)$ ’` separated by commas and enclosed in square brackets. The indices i must be strictly increasing within a vector and the pair `‘ i , $x(i)$ ’` denotes that the i th coefficient of the vector is $x(i)$. Unspecified coefficients are taken as zero. Thus in a 5 dimensional space V , `‘[1,1]’` denotes the vector `‘(1,0,0,0,0)’`, `‘[5,-1]’` denotes `‘(0,0,0,0,-1)’` and `‘[1,1,2,2,3,3,4,4,5,5]’` denotes `‘(1,2,3,4,5)’`.

For `me` only, a packed format is also available for these vectors (this feature should be available in all three programs, and hopefully will be in some future release). This is given as a series of field elements separated by commas and enclosed in parentheses. There must be exactly as many field elements as the specified dimension of V .

4 Command Line Options

The programs are controlled at run-time by options on the command line. There are no command line parameters except options. Options which do not take arguments may be combined, and, with a few exceptions, options may appear in any order. Thus ‘**me -g -i**’ is equivalent to ‘**me -gi**’ and to ‘**me -ig**’.

4.1 Strategic Options

The vector enumeration algorithm has some intrinsic flexibility. The exact algorithm used is controlled by these options. These options have no effect when **-Q** is also present. See Chapter 6 [Strategy], page 29.

-a +|-| *amount* Option

The **-a** option controls the amount of lookahead which takes place. The arguments **+** and **-** enable and disable lookahead (respectively), while a numeric argument *amount* specifies the how far ahead to look (in weights). See Chapter 6 [Strategy], page 29 for the exact details of when lookahead takes place. The default is to look ahead 2 weights.

-e + | - | *dim* | *min:max* Option

The option **-e** controls Early Closing. Early closing is said to occur when there are no blank entries in the program’s table, so that the table represents a representation of the free algebra. In this situation the table is usually, though not always, correct, and it can be beneficial to stop the program and check correctness by other means.

If the table is incorrect, one of the relations will fail to hold on it, and the correct table will be of smaller dimension.

The argument **+** enables early-closing at any dimension. The argument **-** disables early-closing completely. A single numeric argument *dim* allows early-closing only at that dimension, while a pair of numeric arguments *min* and *max* separated by a colon, but not by any whitespace, allows early-closing between dimensions *min* and *max* inclusive.

4.2 Output Options

On successful completion the programs may output their results in various formats, and with various details included or not. This group of options deals with selecting what information will be output, in what formats and to what files.

4.2.1 Options to control what information is output

-P	Option
-b	Option

The **-P** options controls the recording of pre-images. If this flag is present the program will keep track of the basis elements in terms of the module generators and words in the group (or algebra) generators. This information is printed out when the table is printed (ie when **-vs2** is present) and in appropriately formatted output. This option has no effect when **-Q** is also present.

The alternative name **-b** for this option is obsolete and is retained for compatibility.

-i	Option
-----------	--------

The option enables output of Images. If this flag is present then the images of the module generators in the output module are included in the output. This is only really relevant for non-cyclic modules as otherwise the generator is the first basis vector and can only be deleted if the output module is zero.

4.2.2 Options for Output Formats

These options control For the exact specification of the various output formats see Chapter 5 [Output Formats], page 24. The names of the files to which the variously formatted outputs are written are derived from a common *stem* set with the **-o** option. Section 4.2.3 [Output File Names], page 17.

-c	Option
-----------	--------

Enables Cayley format output. If this option is present a Cayley library containing the information will be written to *stem*. At present the pre-image information will not be written even if **-P** is present. This may be fixed in a future release.

-g Option
Enables GAP format output. If this option is present a file '*stem.g*' will be written, suitable for reading by GAP version 3.1 or later.

-m Option
Enables Meataxe format output. This output is written into a series of files whose names are obtained by appending '.' and the generator names to *stem*. The images, if **-i** was present, are written to '*stem.IM*'. This option is only available for **me** as there is no Meataxe format for integers or rationals.

-q Option
Enables plain ASCII output format. The output filename is '*stem.pa*'. This format is designed to be portable and simple to parse (for a computer).

-B Option
Enables plain binary output format. This is written to '*stem.pb*' and is essentially just the binary version of the output generated when **-q** is selected. It is not likely to be portable between different hardware architectures or different versions of the software, but should be very fast to read and write.

-x Option
Enables AXIOM output format. The output file-name is '*stem.input*'.

4.2.3 Output File Names

-o Option
Sets the output file stem *stem*. This is used to construct the names of the various differently formatted output files (see above). The default is **meout**, unless the **-Q** option is present, in which case the default is **qout**.

4.3 Logging and Error Message Options

As the program runs it prints various messages indicating progress and/or reporting errors. The level of detail reported, and the destination of these messages can be controlled by the options in this section. All of the progress messages can be suppressed by removing the compile-time option

LOGGING, reducing the program size and speeding it up slightly. If this is done then the `-v`, `-l` and `-L` options will be ignored with an error message.

-l *file* Option
Sets the Logging file. This *file* argument is a compulsory path name and logging output is directed to that file. The default is to send logging output to standard output.

-v *0 | + | (type level) ...* Option
This option controls the Verbosity of logging.

The argument may take one of the special forms `0` or `+`. The option `-v0` turns off all logging. No output should then be sent to the log file (unless further `-v` options are given). The options `-v+` turns on all possible logging messages, producing extremely copious details of all stages of the calculation.

Otherwise the argument is a series of *type level* pairs (no whitespace may appear within the argument). Each *type* is a single letter, selecting a category of event to be logged, the *level* is an unsigned integer indicating how detailed a log is required. At *level* `0` no events are logged. The following types are defined:

- a** Controls logging of group and algebra Actions as they are computed. The default level is `0`. All higher levels are equivalent and produce a lot of output.
- c** Controls logging of Coincidence processing. The default level is `0`. At levels greater than `0` all coincidences are logged as they are stacked and processed and all equations are logged as they are processed. At level `2` or higher the `vroot` subroutine which replaces a vector by its undeleted image is also logged.
- i** Controls logging of Initialisation of new basis vectors. The default level is `0`. At level `1` each new basis vector defined is logged. At levels `2` and higher the full details of the definition are given.
- k** Controls logging of packKs of the coset table. The default level is `1`. At levels greater than `0` each pack is logged. At levels greater than `1` each basis vector relocated during a pack is logged.
- l** Controls logging of Lattice operations (`zme` only). The default level is `1`. At this level and higher packing of the lattice and pushes of the lattice are logged. At levels greater than `1` each vector added to the lattice and

other substantial changes, and each vector pushed are logged. At level 3 or higher the Gauss-Jordan processing that happens as each vector is added and the effect of coincidences on the lattice are logged.

- m** Controls logging of memory allocated by the routines in `'allocs.c'` This is only permitted when `DEBUG` was defined at compile time. At all levels greater than 0 the allocation is logged after submodule generators are pushed. At level 2 or higher it is also logged after every weight increase. Whenever the allocation is logged certain consistency checks are also performed and any anomalies are reported.
- p** Controls logging of relators Pushed. The default level is 0. At level 1 or higher it logs each relator pushed from a basis element. At level 2 or higher it also logs the image of each algebra relator (which will then be forced to zero).
- s** Controls logging of Stages in the progress of the program. The default level is 1. At level 1 or higher the program prints brief messages at major stages in the progress of the program: after the input file is read; after the submodule generators are processed and at the end of the run. At level 2 or higher the program also prints the relators as they are read in, and the whole table at completion. At level 3 or higher the table is also printed after the submodule generators have been processed.

When the `-Q` option is also present see Section 4.4 [Input File Options], page 20 this suboption has a different effect. At level 1 a message is printed after the submodule generators have been read in and another on completion. A level 2 or higher messages are also printed after every hundred sets of images have been read in.
- w** Controls logging of Weight changes. The default level is 1. At any level greater than zero it logs every change of weight (See Chapter 6 [Strategy], page 29).

-L *Prefix*

Option

Use of this option causes the argument string to be printed at the start of every line of logging output. It can be used if the log information is later to be read into another program, to cause the messages to be treated as comments.

Note that no space is inserted after the prefix string. If one is needed it should be supplied as part of the string. For example `-L'#I '`.

-W *Warnlevel* Option

This option controls the level of detail in the warning messages produced by the input parser in response to incorrect input file syntax.

Whatever the value, the parser will attempt to recover and read the rest of the file in order to report any further errors. Some errors (such as a repeated generator name) are ignorable, but most will prevent the actual calculation taking place. After reading the whole input file the program will continue with the calculation only if no non-ignorable errors occurred.

At a *Warnlevel* of 0 no messages are produced. The program will either run or stop with a non-zero return code.

At a *Warnlevel* of 1 a message describing each error is printed.

At a *Warnlevel* of 2 the approximate line number and position of the error is also reported.

At higher levels the offending line is printed, together with an indication of the location of the error.

The default level is 2.

4.4 Input File Options

These options specify, or alter the meaning of, the main input file read by the program.

-p *Presentation* Option

This option specifies the name of the main input file (the presentation). The *Presentation* is extended to a pathname by adjoining **.pres**, unless the **-Q** option appears before the **-p**, in which case **.qin** will be adjoining instead (and the input file will be expected to be in a different format). By default input is read from standard input. See Chapter 3 [Input Formats], page 9 for details of the format of the input files.

-A Option

This option indicates that the generators may be assumed to commute with one another (so that computations are taking place in a polynomial ring). Relations of the form $XY = YX$ will be added for every pair of generators.

-C *Characteristic* Option

This option overrides the characteristic specified in the input file. The argument *Characteristic* is the new characteristic, which must be 0 for *zme* or *qme* and a prime between 2 and 251 inclusive for *me*.

-Q Option

This option switches the program from the default behaviour of reading a *.pres* presentation file and constructing the module presented to reading a *.qin* quotient specification and constructing the quotient action. It should appear *before* **-p** if both are present.

-n Option

Normally the programs generate the extra relation $x=x$, for each non-invertible generator x . This is to ensure that all definitions will be made eventually, so that the table will close. Otherwise, inputting the natural presentation for the free algebra would cause no action.

This option suppresses these relations. It is only safe to use when you are sure that every generator appears as the first letter of (a term of) a relation.

4.5 Debugging Options

Most users should not need these options, and indeed will have compiled the programs so that they are not available. They have no effect if **-Q** was specified.

-s + | - | *Filename Stem* Option

This option controls the *Scrutinize* feature. This is only available if the compile-time option **SCRUT** was set and is never available for *zme*. The scrutinize system maintains a homomorphism from the module under construction to one read in at the beginning (which should be the correct module). This is used to check all deductions and coincidences. Any discrepancies found are reported.

The `+` argument turns scrutinize on, the `-` argument turns it off (the default). A *Filename Stem* argument turns it on and selects the module to be read in. The *Stem* defaults to `meout`. The actual filename is obtained by adjoining `.pb` to the stem, and the module is read in in the plain binary format described in Section 4.2.2 [Options for Output Formats], page 16.

-y Option
 This option turns on Bison's own debugging for the input parser. It is only available when `DEBUG` was specified at compile-time. Rather copious information is output on the behaviour of the push-down automaton that parses the presentation.

4.6 Limit Options

This group of options imposes limits of time, space or weights on the program. If a limit is exceeded the program will exit with a non-zero return code and write no output.

-w *MaxWeight* Option
 This option imposes a limit on the maximum weight of basis element that can be defined. The default limit is 100. See Chapter 6 [Strategy], page 29 for details of the use of weights. This option has no effect when the `-Q` option is also given as weights are not used in that case.

-T *Time Limit* Option
 This option imposes a limit on the CPU time used by the program. The program will stop soon after the *Time Limit* CPU seconds have been used (*Time Limit* is given in floating point). The CPU time is not checked as often as it might be. The default is to have no limit.

-t *Max Dimension* | *Max Dimension:Reserved Percentage* Option
 The option limits the dimension of the module being constructed. The program will attempt to complete the calculation in a smaller space, but may not succeed. The default limit is very large indeed. This option has no effect when `-Q` is also given, as the table size never increases.

By default 5% of the dimension limit is reserved for certain critical definitions (occurring during coincidence processing). If there is no space for a critical definition when one

is needed the program exits with an error. The proportion of reserved space can be altered using the second form of this option.

5 Output Formats

On successful completion the programs may write out the results in various ways. Which format(s) are used, and what filenames the output is written to are controlled by various command line options. Section 4.2 [Output Options], page 16.

This chapter describes the format of the files which are written. There are four formats intended to be read into other systems and two formats intended to be easy for special purpose programs to read.

5.1 Cayley Format

This format is intended for reading by version 3 of the Cayley computational algebra system produced by John Cannon and associates at the Department of Pure Mathematics, Sydney University, Sydney, NSW, AUSTRALIA. Contact john@maths.su.oz.au for details.

The output file is a Cayley library and sets the following variables:

fld	This is set to be the field (or, in the case of zme , ring) over which calculations have been performed.
vs	This is set to be a vector space (or module) over fld , of the same dimension as the module constructed by the program.
grp	This is matrix group over vs . Its generators have the same names as the generators specified for the vector enumerator, and their actions on vs are those computed by the program.
Images	This is a matrix with as many rows as there were module generators input to the enumerator and as many columns as the dimension of the module constructed. It gives the images in the module constructed of the original module generators. It is only present if the -i option was selected at run-time.
Lattice	This is only present for zme and describes the torsion of the module constructed. It is a matrix with as many columns as the dimension of the module and a row for each lattice generator.
VERuntime	This is the CPU time used in vector enumeration, in milliseconds.

This output format has not been tested much except with group algebra presentations over finite fields. It may not work in other situations. The `-P` option to write out preimages has no effect on this output format.

5.2 GAP format

This format produces a file intended to be read by the GAP system, version 3.1 or later. This system is produced by Lehrstuhl D für Mathematik, RWTH-Aachen, Aachen, GERMANY and is available for anonymous FTP from `samson.math.rwth-aachen.de`.

The output file is a GAP source file which defines the following global variables:

<code>field</code>	The field (or ring) over which the enumeration was performed.
<code>VERunTime</code>	The CPU time in milliseconds used by the enumeration.
<code>images_mat</code>	This item is only present if the <code>-i</code> option was given. It consists of a matrix of field entries with as many columns as the dimension of the module constructed and a row for each module generator specified in the input file, giving the image of that generator in the final module.
<code>ggen</code>	This item is only present if the <code>-P</code> (or <code>-b</code>) option was given at run-time. For each generator <i>gen</i> specified in the presentation a global GAP variable <i>ggen</i> is given the value <code>AbstractGenerator("gen")</code> .
<code>preImages</code>	This item is only present if the <code>-P</code> (or <code>-b</code>) option was given at run-time. It consists of a list of records, one for each basis vector in (dimension of) the module constructed. Each such basis vector is the image of one of the free module generators by a product of generators, and each record has two fields <code>modGen</code> , which contains the number of the free module generator (starting at 1) and <code>word</code> which contains a word (given in terms of <code>IdWord</code> , the <i>ggen</i> and their inverses) giving the appropriate product of generators.
<code>gen_gen</code>	For each generator <i>gen</i> of the presentation, a global variable <i>gen_gen</i> is set to the matrix giving the action of <i>gen</i> on the module constructed.
<code>gens</code>	The list <i>gens</i> is a list of all the <i>gen_gen</i> .
<code>lattice_mat</code>	This is only produced by <code>zme</code> and contains the lattice that describes the torsion of the constructed module.

5.3 Meataxe Format

This format is only available with `me`, as the Meataxe has no facilities for handling integers or rationals. It is intended to produce input suitable for reading into the Meataxe programs of R.A. Parker.

In this output format the action of each generator, and the images if `-i` was given, is written to separate file. The generator name is appended to the file name stem. See Section 4.2 [Output Options], page 16 for more details.

Each matrix is ready to be read in by the meataxe program `cv`.

The `-P` option to record pre-images has no effect on this output format.

5.4 AXIOM format

This format produces output suitable for reading into the AXIOM computer algebra system distributed by NAG, Ltd..

The file produced assigns to the following variables:

<code>fld</code>	This is set to be the field or ring over which the enumeration was performed.
<code>VERunTime</code>	This is the CPU time in milliseconds used by the enumeration.
<code>images_mat</code>	This is set to be a matrix of field entries giving the images in the module constructed of the original free module generators.
<code>rec</code>	This is only set if the <code>-P</code> option is given, when it is set to be the type <code>Record(modGen : PositiveInteger, word : Polynomial Integer).</code>
<code>gen_names</code>	This is only set when the <code>-P</code> option is given. It is of type <code>LIST Symbol</code> and contains the names of the generators from the presentation file.
<code>preImagesL</code>	This is only set when the <code>-P</code> option is given. It is of type <code>LIST LIST Any</code> and is used only to set:

preImages

This is only set when the `-P` option is given and the `preImagesL` coerced to type `LIST rec`. It has a member for each basis vector (dimension) of the module constructed. Each such basis vector is the image of one of the original free module generators under a product of algebra generators. The `modGen` field of the record specifies the free module generator (starting at 1), which the `word` field is the appropriate monomial in the symbols corresponding to the generators.

gen_gen For each generator *gen* given in the presentation, an AXIOM variable `gen_gen` is set equal to the matrix (with entries in `fld`) giving the action of the generator on the module constructed.

gens This list contains all the `gen_gen`.

lattice_mat

This is only produced by `zme`. It is a matrix giving the torsion of the module constructed.

5.5 Plain ASCII format

This format is an unadorned ASCII format intended to be machine read. It consists of ASCII numbers separated by whitespace.

The first line is a header, consisting of four or five numbers

- The dimension of the constructed module
- The number of algebra generators (not included inverses). That is the number of generators whose actions appear in this file.
- The number of free module generator images in the file. This will be 0 if the `-i` option was not given at run-time, and will otherwise be the number of free module generators in the presentation.
- A boolean (C style, so 1 for true and 0 for false) entry indicating whether pre-image information appears in the file (ie whether the `-P` option was given at run-time).
- For `zme` only, the number of generators of the torsion lattice.

The next part of the file gives the images of the free module generators, and is only present if the `-i` option was given at run-time. If it is present then there is a line for each module generator giving its image in the module constructed.

These vectors, and others later in the file are given as field elements separated by spaces. There is no terminator, but the number of field elements expected can be deduced from the header information. Finite field elements are given as integers between 0 and *characteristic*-1. Rational numbers are given as '*numerator/denominator*'. Integers and numerators and denominators of rationals may be too large to read into any fixed-size type.

The next section, which will only be present if the `-P` option was given at run-time (which can be deduced from the header information), gives the pre-images of the basis of the module constructed in the free module. For each basis vector there is a line giving the free module generator and a word in the algebra generators. The free module generator is given as an integer (they start at 1), the algebra generators as integers, in the order in which they are defined in the presentation, starting at 1, with a negative number denoting the multiplicative inverse of its absolute value. The words are terminated by a zero.

The next section gives the matrices for the action of the generators. The rows are given as vectors in the format described above, and the matrices are not separated.

For `zme` only, the lattice generators are given as a series of vectors.

Finally, the CPU time used, in milliseconds appears on a line by itself.

5.6 Plain Binary Format

This format is essentially identical to the Plain ASCII format described in Section 5.5 [Plain ASCII format], page 27, except that the individual entries, instead of being whitespace separated decimal numbers are fixed-length binary numbers. All numbers are written as `unsigned int` or `int` (for generator numbers in pre-images), except that for `qme` and `zme`, the rational numbers and integers are written using the `mpz_out_raw` function of GMP.

6 Strategy

The vector enumeration algorithm, as described in *On Vector Enumeration*, for instance, leaves unanswered a number of practically important questions about the order in which relations and submodule generators are used. A set of answers to these questions is known as *an enumeration strategy*.

The strategy used by these programs is based on the coset enumeration strategy defined in *Double Coset Enumeration, Journal of Symbolic Computation (12) 1991*. This is an HLT-based strategy (something about which we have no choice for vector enumeration) based on two key ideas, *lookahead*, originally due to M.J.T. Guy, and *weights* due to R.A. Parker.

6.1 Weights

In this approach, every relation of the presentation, every basis vector in the module constructed and (for **zme**) every vector in the lattice and every generator, has a weight, which is a positive integer.

The weights of the relations may be given in the presentation file See Section 3.1.3.3 [Weight Specifications], page 12. Otherwise they are given default values. For a group-type relator, see Section 3.1.3.1 [Group Type Relations], page 11 the default weight is half the length of the relator. For an algebra relation, see Section 3.1.3.2 [Algebra Relations], page 11 the default weight is 3.

For **zme** the weights of the generators are currently fixed at 2. A mechanism for setting them will be added to a future release.

The weight of a basis vector or lattice vector created during submodule processing is 1. After that, there is a current weight, which begins at 2 and increases steadily. At current weight w , all *(basis vector, relation)* pairs whose weights total w (or less, if somehow they have not been processed already) will be processed. New basis vectors and lattice vectors created during this processing will be assigned weight w .

In **zme**, there is additional processing, because, as well as the *(basis vector, relation)* pairs, the image of every lattice vector under every generator must be computed and forced to lie in the lattice. At current weight w , every *(lattice vector, generator)* pair whose weights sum to w (or less) is processed.

There is a maximum weight, and if the current weight reaches the maximum weight without the calculation being complete then the program stops. The maximum weight is 99 by default, and can be set from the command line Section 4.1 [Strategic Options], page 15.

The art of setting weights to obtain optimum performance is, unfortunately, just that: *an art*. All I can suggest is experimentation, possibly with related, but smaller presentations. Very complex algebra relations should probably have their weight increased from the default.

Certain relations are adjoined automatically to the presentation. These are given standard weights. The relations ' $xx=1$ ', $xx^{-1}=1$ or $x=x$ adjoined for involutory, invertible or arbitrary generators x have weight 3. The relations $xy=yx$ adjoined for each pair of generators x and y when the $-A$ flag is present see Section 4.4 [Input File Options], page 20 have weight 2.

6.2 Lookahead

The idea of lookahead is to detect in advance coincidences which can be proved without further definitions. From time to time, the program shifts into *lookahead mode*, and attempts to process some further relations without defining new basis vectors. It then shifts back to the normal state, called *define mode* and resumes processing from where it left off.

The program will only change modes when the current weight is about the change. It goes into lookahead mode if the number of basis vectors has doubled since it last entered define mode (or since the end of submodule processing if this is the first lookahead). It looks ahead a number of weights controlled by the command line option $-a$ see Section 4.1 [Strategic Options], page 15, or 2 weights by default.

A relation may have a different weight in lookahead mode from its weight in define mode. Indeed the relations $xx=1$, $xx^{-1}=1$ or $x=x$, adjoined automatically to the presentation, have weight 6 in lookahead mode, (rather than 3) as their main function is to make sure that all entries in the table are filled eventually.

7 The Source Code

This chapter provides a little information about the arrangement and operation of my source code. It is intended only for those trying to fix local compilation problems or to make small changes or additions. If you are interested in getting more deeply involved with the innards of the program please feel free to get in touch with me.

7.1 Generalities

7.1.1 Important macros

The file ‘`meint.h`’ contains both the external declarations for most of the functions in the programs and a large number of critical macros. The macros are crucial to the way in which the three programs are generated from one set of sources, and to the operation of the `DEBUG`, `SCRUT` and `LOGGING` compile-time flags. Among the more significant ones are:

DIE() In `DEBUG` mode (when the pre-processor symbol `DEBUG` is defined) `DIE()` prints an error message indicating the file and line number and calls `abort()` generating a core dump. In non-`DEBUG` mode it simply exits with a non-zero return code.

ASSERT(*condition*)

In `DEBUG` mode this macro checks whether *condition* is true and calls `DIE()` if it is not. In non-`DEBUG` mode it generates no code (in particular *condition* is not evaluated). This is the principle method of including consistency checks in the debugging version of the program.

IFDEBUG(*code fragment*)

Simply executes *code fragment* only in `DEBUG` mode. For example after de-allocating the contents of a pointer it is good practice to put a `NULL` in it to avoid referencing the freed memory.

LOG(*condition*, *code fragment*)

This macro generates no code unless the `LOGGING` pre-processor symbol is defined, in which case it executes *code fragment* if *condition* is true. Typically *condition* checks one of the global variables such as `logcoin` that control the verbosity of logging see Section 4.3 [Logging and Error Message Options], page 17.

SC(*code fragment*)

Executes *code fragment* only if the `SCRUT` pre-processor symbol is defined and the global variable `scrut` (set by the `-s` command-line option Section 4.5 [Debugging Options], page 21) is `true`.

`PS(vector, code for packed, code for sparse)`

In `me`, but not in the other programs, a vector in the module under construction can be represented in one of two forms (packed and sparse, see *Constructing Matrix Representations of Finitely Presented Groups*). It is often necessary to use separate pieces of code to deal with the two cases.

When `me` is being compiled, this macro expands to an `if ... else ...` statement that executes the appropriate code in each case. When the other programs are being compiled it expands to *code for sparse*.

7.1.2 Arithmetic

Arithmetic in the underlying field (ring) is provided by macros such as `Fadd`, `Fmul`, `FOne` which enable much of the coding to be independent of the underlying arithmetic. In some places (mainly in `vector.c`) explicit `#if ...#endif` structures are used where speed requires special code for the different programs.

Pre-processor symbols `ME`, `QME` and `ZME`, indicate which programs are being compiled, but in the interest of flexibility, a separate set `GFP`, `RATIONAL` and `INTEGRAL` are used to indicate which arithmetic is being used. For the `GFP` case many of the macros (those dealing with allocating space, for example) expand to no code, and they all expand to inline code fragments. For the other arithmetics almost all the macros expand to calls to GMP routines.

7.1.3 Prototyping and Other Language Issues

The file `'global.h'` attempts to make the code compatible with both ANSI and traditional C compilers, based on whether the pre-processor symbol `__STDC__` is defined. In particular, the macro `PT` is defined to either return or ignore its argument, so as to provide proto-typing when it is allowed. The type `pointer` is also defined to be either `char *` or `void *` as appropriate.

7.2 The Source Files

`'alloca.c'`

This is taken from a GNU source, and emulates the BSD library function `alloca` on systems where it is not provided. `alloca` is used only by the parser, so speed is not critical. On BSD systems use the library function instead.

'allocs.c'

This file contains subroutines to allocate certain types of objects quickly (by allocating space for them in large groups and then maintaining a free queue). It is used for vector headers and (except in **me**) for small blocks requested by GMP.

'coin.c' This file contains the code that maintains the coincidence stacks and processes coincidences. Coincidence processing has changed a little from earlier releases and the published descriptions in that a coincidences is simply stored as a vector to set equal to 0, and the choice of which (if any) basis vector to delete is deferred until the coincidence is destacked. This is principally for the benefit of **zme**, but seems beneficial in general. The important **vroot** subroutine that replaces a vector by its undeleted image is also here.

'comline.c'

This file processes the options on the command line see Chapter 4 [Command Line Options], page 15 and sets global variables appropriately

'ilatt.h' This file contains macros and declarations used only by **'lattice.c'**. Declarations for routines exported from **'lattice.c'** are in **'latt.h'**. This file is only part of **zme**.

'input.c' This file contains functions called by the input parser (and a driving routine to call the parser and tidy the presentation afterwards).

'input.h' A number of functions and macros are used only at the input end of the program, by **'input.c'**, **'input.tab.c'** and **'scanner.c'**. They are declared here to avoid further complicating **'meint.h'**.

'input.tab.c'

This file contains the parser produced by bison from **'input.y'**. It is called **'inputtab.c'** under MS-DOS.

'input.y' This file is the bison grammar file which produces **'input.tab.c'**. It is called **'inputtab.y'** under MS-DOS.

'latt.h' This file contains the declarations of functions and globals exported by **'lattice.c'** the routines that manage the torsion lattice in **zme**.

'lattice.c'

This file contains the code that manages the torsion lattice used in **zme**. The exact role of this lattice will be described in a future publication on the integer vector enumeration algorithm. In a future release this file may be replaced by one using R.A.Parker's p-adic methods to manipulate the lattice.

'me.c' This file contains a lot of global variables and some miscellaneous subroutines. It is mostly a graveyard for functions with no other natural home. The routine **SetDefaults** which sets default values for many run-time options is here.

- 'me.h'** This header file contains features which users might wish to change, such as the compile-time parameters see Section 2.1.3 [Compile-time options], page 7 and some type declarations and limits.
- 'meint.h'** This file is the main header file of the entire system. Everything starts from here.
- 'memain.c'** This file contains `main()` and other outer-level routines. The basic sequence of operations is controlled from here.
- 'myalloc.c'** This file provides a debugging version of `malloc`, `free` and `realloc`, and a routine `DumpHeap` to report the current state of heap allocation. It records the file and line from which each block was allocated, pre-loads each allocated block with a nonsense value (hex E6), and tests freed blocks for consistency. It greatly slows down the program and is used only in `DEBUG` mode.
- 'myalloc.h'** This is the header file associated with `'myalloc.c'`.
- 'out.c'** This file contains the output routines, both those used to output the results of a successful run and those used to output logging information during a run. The output routines have been extensively revised since the last release and each output format is now described by a large data structure and some associated static subroutines. Hopefully this will make it easier to add new output formats.
- 'pack.c'** This file simply contains the subroutine to pack the table and recover the space occupied by deleted rows.
- 'push.c'** This file contains the routines that compute the action of the algebra on the table, and so the routines used to process the relations. Definition of new basis vectors is also handled here.
- 'scanner.c'** This file contains the lexical analyser which tokenises the input. This cannot be done with (f)lex, as the analyser has to be changed after reading the initial list of generators. This file also contains the code that keeps track of the lines of input so as to be able to report syntax errors properly.
- 'scrut.c'** This file contains the code implementing the `SCRUT` feature. No code from this file is compiled unless the `SCRUT` pre-processor symbol is defined (in `'me.h'`).
- 'vector.c'** This file contains the basic vector allocation, deallocation and arithmetic subroutines. It is speed critical.

8 Examples

Finally we consider a number of simple examples. The presentation files used for these examples are in the ‘./examples’ subdirectory of the distribution, as are a number of others.

8.1 The natural permutation representation of S_3

The symmetric group S_3 is also the dihedral group D_6 , and so is presented by two involutions with product of order 3. Taking the permutation action on the cosets of the cyclic group generated by one of the involutions we obtain the following presentation file (‘s3.pres’).

```
0.AB...A:.(AB)3:.
```

Here 0 is the characteristic; AB specifies two generators A and B (note that **ab** would specify one generator **ab**). The two empty sections ... indicate that both generators are involutions. There is one “submodule generator” A (actually A^{-1} is the submodule generator) and one relation $(AB)^3 = 1$.

It is of course, much more sensible to use a normal Todd-Coxeter program to perform this calculation, as the representation being constructed is a permutation representation.

Running `qme -vs2 -ps3` to process this file (the `-vs2` simply causes the presentation to be printed once it has been read) we get:

```
%qme -vs2 -ps3
Submodule gens
Weight 1 group type    A
Relator
Weight 3 group type    BB
Weight 3 group type    AA
Weight 3 group type    ABABAB
Done submodule generators
Starting weight 2 in define mode, 1 alive out of 1
Starting weight 3 in define mode, 1 alive out of 1
Starting weight 4 in define mode, 1 alive out of 1
Looking ahead ...
Starting weight 5 in lookahead mode, 3 alive out of 4
Starting weight 6 in lookahead mode, 3 alive out of 4
...done
Packing 4 to 3
Starting weight 5 in define mode, 3 alive out of 3
```

```

Starting weight 6 in define mode, 3 alive out of 3
Starting weight 7 in define mode, 3 alive out of 3
Closed, 3 rows defined
3 live dimensions

```

If we also add the `-g` option to get GAP format output then the program writes a file ‘`meout.g`’ containing:

```

field := Rationals;
VERunTime := 16;
gen_A := field.one*[[1,0,0],
[0,0,1],
[0,1,0]];
gen_B := field.one*[[0,1,0],
[1,0,0],
[0,0,1]];
gens := [gen_A,gen_B];

```

8.2 A Quotient of a Permutation Representation

The permutation representation constructed in Section 8.1 [A permutation representation], page 35 fixes the all-ones vector (as do all permutation representations). We see easily enough that this is $1+B+BA$ times the module generator (the vector $(0,0,0)$). Accordingly we can write down the following presentation for the quotient module (‘`s3a.pres`’):

```
0.AB...A:0=1+B+BA.(AB)3:.
```

Knowing that we want a dimension 2 module we can turn on early-closing see Section 4.1 [Strategic Options], page 15 and run `qme -ps3a -e2 -c` (we will take Cayley output this time) with the following result:

```

%qme -ps3a -e2 -c
Read Input
Done submodule generators
Starting weight 2 in define mode, 2 alive out of 3
Packing 3 to 2
Early Closing
Closed, 2 rows defined
2 live dimensions

```

Here we see the advantage of early-closing when the final dimension is known. The output file in this case is ‘`meout`’:

```

LIBRARY meout;
fld : rationals;
vs : vector space (2,fld);
grp : matrix group (vs);
grp.generators :
    A = MAT(
        1,0:
        -1,-1),
    B = MAT(
        0,1:
        1,0);
VERuntime = 16;

```

8.3 A Non-cyclic Module

If we take the direct product of two copies of the permutation representation constructed in Section 8.1 [A permutation representation], page 35, we can identify the fixed vectors in the two copies in the following presentation:

$$0.AB\dots\{2\}[* , A-1] , [1, 1+B+BA] = [2, 1+B+BA] . (AB)^3 : .$$

which we can run with `qme -ps3b` to get

```

qme -ps3b
Read Input
Done submodule generators
Starting weight 2 in define mode, 5 alive out of 7
Starting weight 3 in define mode, 5 alive out of 7
Starting weight 4 in define mode, 5 alive out of 7
Starting weight 5 in define mode, 5 alive out of 7
Closed, 7 rows defined
Packing 7 to 5
5 live dimensions

```

In this case it is interesting to look at the images of the module generators and pre-images of the basis vectors. We choose AXIOM format and rerun the program with `qme -v0 -Pix -e5 -ps3b`. This produces nothing on the standard output, but writes the file `'meout.input'` as follows (some indentation and spacing has been added for ease of reading):

```

fld := FRAC INT
VERunTime := 16
images_mat := matrix([[1,0,0,0,0],_
    [0,1,0,0,0]]) :: MATRIX fld

```

```

rec := Record(modGen : PositiveInteger, word : Polynomial Integer)
gen_names : LIST Symbol := [A,B]

preImagesL : LIST LIST Any := [_
    [ 1, 1],_
    [ 2, 1],_
    [ 1, B],_
    [ 1, B*A],_
    [ 2, B]]
preImages := preImagesL :: LIST rec

gen_A := matrix([[1,0,0,0,0],_
    [0,1,0,0,0],_
    [0,0,0,1,0],_
    [0,0,1,0,0],_
    [1,-1,1,1,-1]]) :: MATRIX fld
gen_B := matrix([[0,0,1,0,0],_
    [0,0,0,0,1],_
    [1,0,0,0,0],_
    [0,0,0,1,0],_
    [0,1,0,0,0]]) :: MATRIX fld

gens := [gen_A,gen_B]

```

From the `images_mat` section of this file we see that the two module generators are just the first and second basis vectors.

From the `preImages` section we see that the five basis vectors are images of the following elements of the free two-generator module for the free two-generator algebra: $(1,0)$, $(0,1)$, $(B,0)$, $(BA,0)$ and $(0,B)$.

8.4 A Monoid Representation

The Coxeter monoid of type B_2 has a transformation representation on four points. This can be constructed as a matrix representation over $GF(3)$, from the following presentation (`'mond8.pres'`):

```

3.gena genb.*.:gena = 1 .
: genagena = gena , genbgenb = genb,
(genagenb)2 = (genbgena)2.

```

This presentation uses lower-case generator names, so there are two generators `gena` and `genb`. The space separating them in the initial list of generators is needed to prevent them being read as one generator `genagenb`. The `*` on the first line indicates that both generators are not (known to be) invertible. Indeed, we go on to specify that they are idempotents.

In the relations no space between generator names is needed as the longest-match scanner can always sort out what we mean. Some space might have improved readability however.

We can process this with ‘me’ (since we are in positive characteristic):

```
%me -vs2 -pmond8 -g -e4
Submodule gens
Weight 3 algebra type gena=1
Relator
Weight 3 algebra type genb=genb
Weight 3 algebra type gena=gena
Weight 3 algebra type gena*genb*gena*genb=genb*gena*genb*gena
Weight 3 algebra type genb*genb=genb
Weight 3 algebra type gena*gena=gena
Done submodule generators
Starting weight 2 in define mode, 1 alive out of 2
Starting weight 3 in define mode, 1 alive out of 2
Starting weight 4 in define mode, 1 alive out of 2
Looking ahead ...
Starting weight 5 in lookahead mode, 4 alive out of 7
Starting weight 6 in lookahead mode, 4 alive out of 7
...done
Packing 7 to 4
Starting weight 5 in define mode, 4 alive out of 4
Starting weight 6 in define mode, 4 alive out of 4
Starting weight 7 in define mode, 4 alive out of 4
Starting weight 8 in define mode, 4 alive out of 9
Starting weight 9 in define mode, 4 alive out of 9
Starting weight 10 in define mode, 4 alive out of 9
Closed, 9 rows defined
Packing 9 to 4
4 live dimensions
```

It is clear that the relations `genb=genb` and `gena=gena` have no effect, since other relations will force the table to be completed. We could therefore have used the `-n` option see Section 4.4 [Input File Options], page 20, to save a small amount of effort.

The output file from this calculation is:

```
field := GF(3);
VERunTime := 16;
gen_gena := field.one*[[1,0,0,0],
[0,0,1,0],
[0,0,1,0],
[0,0,0,1]];
gen_genb := field.one*[[0,1,0,0],
```

```

[0,1,0,0],
[0,0,0,1],
[0,0,0,1]];
gens := [gen_gena,gen_genb];

```

and the matrices are clearly non-invertible.

8.5 An Integer Module with Torsion

The algorithm for integer module enumeration will be described fully in a forthcoming paper, however one problem which arises is dealing with the information that a generator takes a basis vector to a non-integer multiple of itself. This implies (if the result is to be finite dimensional) that the basis vector is a torsion vector, but does not exactly define the torsion. A further relation is needed to resolve the problem. As an example, consider the presentation (in ‘tors.pres’):

```
0.X*...:2*X = 1.:X7=1.
```

Here the submodule generator implies that X acts as $1/2$ on the module generator (the first basis vector). This leads to an extending lattice which is finally terminated by the relation. The module then collapses and is seen to be one-dimensional with 127-torsion.

The output from the run is:

```

zme -ptors -g
Read Input
Done submodule generators
Starting weight 2 in define mode, 2 alive out of 2
Pushing lattice at weight 2
Starting weight 3 in define mode, 2 alive out of 2
Pushing lattice at weight 3
Starting weight 4 in define mode, 2 alive out of 2
Pushing lattice at weight 4
Starting weight 5 in define mode, 1 alive out of 8
Pushing lattice at weight 5
Starting weight 6 in define mode, 1 alive out of 8
Pushing lattice at weight 6
Starting weight 7 in define mode, 1 alive out of 8
Pushing lattice at weight 7
Closed, 8 rows defined
Packing 8 to 1
Packed lattice from 8 to 1
1 live dimensions

```

and the output file ‘meout.g’ contains:

```
field := Integers;
VERunTime := 16;
gen_X := field.one*[[64]];
gens := [gen_X];
lattice_mat := field.one*[[127]];
```

So that we our module has 127-torsion and X acts on it as 64 (which is 1/2 mod 127).

8.6 Quotient Construction Example

Finally, we look at an example of the use of the program with the `-Q` option to construct quotients of representations. As our input we take the deleted permutation representation of the alternating group A6, which has degree 5. The matrices giving the action are:

A = (0,0,1,0,0)	B = (1,0,0,0,0)
(1,0,0,0,0)	(0,0,0,0,1)
(0,1,0,0,0)	(0,1,0,0,0)
(0,0,0,1,0)	(0,0,1,0,0)
(0,0,0,0,1)	(1,1,1,1,1)

The submodule is generated by such vectors as (1,1,0,0,0). We can write this information into an input file ‘a6.qin’ (it is envisaged that these files would normally be machine generated) as follows:

```
2 5 AB
3
[1,1,2,1]
[1,1,3,1]
[2,1,4,1]
[2,1]
[0,1]
[0,1]
[4,1]
[1,1]
[1,1]
[3,1]
[2,1]
[4,1]
(1,1,1,1,1)
```

Here the first two lines are header information, the next three are submodule generators, and the remainder are the rows of the matrices (suitable formatted). This can be read by `me -Q -vs2` and produces the following output:

```
%me -Q -pa6 -vs2
Read submodule generators 2 live dimensions
Read 0 sets of images
           2 alive      2 blanks
All read in 1 alive
```

With `-ig` as well the output file ‘`qout.g`’ is:

```
field := GF(2);
VERunTime := 16;
images_mat := field.one*[[1],
[1],
[1],
[1],
[1]];
gen_A := field.one*[[1]];
gen_B := field.one*[[1]];
gens := [gen_A,gen_B];
```

Note that the `images_mat` section contains non-trivial information (that all the basis vectors in the original presentation have image 1 in the quotient).

8.7 A Quotient of a Polynomial Ring

The quotient of a polynomial ring by the ideal generated by some polynomials will be finite-dimensional just when the polynomials have finitely many common roots in the algebraic closure of the ground ring. For example, three polynomials in three variables give us the following presentation for the quotient of their ideal (`poly.pres`):

```
0.ABC.*..:
A+B+C=0,
AB+BC+CA = 0,
ABC=1..
```

We run this with the `-A` option to indicate that all the generators commute. We obtain:

```
% qme -A -ppoly -vs2
Submodule gens
```

```

Weight 3 algebra type ABC=1
Weight 3 algebra type AB+BC+CA=0
Weight 3 algebra type A+B+C=0
Relator
Weight 2 algebra type BC=CB
Weight 2 algebra type AC=CA
Weight 2 algebra type AB=BA
Done submodule generators
Starting weight 2 in define mode, 6 alive out of 9
Starting weight 3 in define mode, 6 alive out of 9
Starting weight 4 in define mode, 5 alive out of 11
Starting weight 5 in define mode, 6 alive out of 20
Starting weight 6 in define mode, 6 alive out of 20
Closed, 20 rows defined
Packing 20 to 6
6 live dimensions

```

Thus we see that these polynomials have six common roots. We could determine them by simultaneously diagonalising the matrices giving the action of the generators on the quotient.

Concept Index

A

Abelian algebras	21
Algebra relations	11
Architectures, hardware	6
Arithmetic	8, 32
ASSERT	31
AXIOM	17, 26

B

Basis vectors defined, limit	22
Bison	8
Build script	5

C

Cayley	16, 24
Characteristic	9
Characteristic, over-riding	21
Closing, early	15
Code, structure and conventions	31
Command-line options	15
Comments	9
Commutator	11
Compilation	5
Compilation, general	6
Compilation, options	7
Compilation, simple	5
Conjugate	11
CPU time limit	22
Critical definitions, space reserved for	22
Cyclic modules	12

D

DEBUG	7, 31
DEBUG-mode	7
Debugging options	21
Debugging, bison	22
Diagnostic messages	17
DIE	31
Dimension limit	22
Directories	5

Distribution	5
Documentation	8

E

Early-closing	15
Error messages	17

F

Field specification	9
File, log	18
Flags, command-line	15
Free algebra elements	11

G

GAP	17, 25
Generator names	9
Generators not invertible	10
Generators not involutions	10
Getting hold of the programs	5
GMP	8, 32
Group relators	11
Group words	11

H

Hardware architectures	6
------------------------------	---

I

IFDEBUG	31
Images of module generators	16
Input files	9, 14, 20
Installation	5
Internals	31
Invertible generators	10
Involuntary generators	10

L

Level, warning	20
LOG	31
Log prefix	19
Logfile	18

logging 7
 Logging 17
 LOGGING 7, 31
 Lookahead 15, 30

M

Macros 31
 Make program 6
 Make programs 8
 Makefile 2
 Meataxe 17, 26
 meout 17
 Modules, cyclic 12
 Modules, non-cyclic 12
 Multi-precision 8
 Multiple architectures 6

N

New features 2
 Non-cyclic modules 12

O

Options, command-line 15
 Options, order 15
 Output 16
 Output filenames 17
 Output files 24
 Output formats 24

P

Packed and Sparse Vectors 32
 Packed form of submodule generators 13
 Papers 2
 Parser 8
 Plain ASCII 17, 27
 Plain binary 17, 28
 Polynomial rings 21
 Pre-images 16
 Precedence 11
 Prefix for log messages 19
 Presentation 9, 20
 Proto-typing 32
 PS 32

PT 32

Q

Q - the rational field 9
 qout 17
 Quotient constructing 21
 Quotient construction 14

R

References 2
 Relations 11
 Relators 11
 Reserved percentage 22
 runtime 2

S

SC 31
 Scrutinize 21, 31
 Scrutinize 7
 Separators 9
 Source code 31
 Sparse form of submodule generators 13
 Stem for output file names 17
 Strategy 29
 Submodule generators 12

T

Table size limit 22
 Texinfo 8
 Time, CPU, limit 22

U

Universal submodule generators 13

V

Verbosity of messages 18
 Version 3, new features 3
 Version 3.01, new features 3
 Version 3.02 and 3.02, new features 3
 Version 3.04, new features 2

W

Warnings, control of 20

Warnlevel..... 20
 Weights..... 12, 22, 29

Z

Z - the ring of integers..... 9

Options Index

-		-n.....	21
-a.....	15	-o.....	17
-A.....	20	-p.....	20
-b.....	16	-P.....	16
-B.....	17	-q.....	17
-c.....	16	-Q.....	14, 21
-C.....	21	-s.....	21
-e.....	15	-t.....	22
-g.....	16	-T.....	22
-i.....	16	-v.....	18
-l.....	18	-w.....	22
-L.....	19	-W.....	19
-m.....	17	-x.....	17
		-y.....	22

Table of Contents

Vector Enumeration	1
1 Introduction	2
1.1 What's New.....	2
1.1.1 What's New in Version 3.04	2
1.1.2 What's New in Versions 3.02 and 3.03	3
1.1.3 What's New in Version 3.01	3
1.1.4 What's New in Version 3	3
2 Installation	5
2.1 Installation Procedure.....	5
2.1.1 Simple Compilation	5
2.1.2 General Installation	6
2.1.2.1 Using GNU Make	6
2.1.2.2 Using the GAP Makefile	7
2.1.3 Compile-time options	7
2.2 Pre-requisites	8
3 Input Formats	9
3.1 A Presentation File.....	9
3.1.1 Characteristic	9
3.1.2 Generators	9
3.1.3 Relations	11
3.1.3.1 Group Type Relations	11
3.1.3.2 Algebra Relations.....	11
3.1.3.3 Weight Specifications	12
3.1.4 Submodule Generators	12
3.1.4.1 Submodule Generators in Packed Form.....	13
3.1.4.2 Submodule Generators in Sparse Form	13
3.1.4.3 Universal Submodule Generators	13
3.2 Input for Quotient Construction.....	14
4 Command Line Options	15
4.1 Strategic Options	15
4.2 Output Options.....	16
4.2.1 Options to control what information is output	16
4.2.2 Options for Output Formats	16

4.2.3	Output File Names	17
4.3	Logging and Error Message Options	17
4.4	Input File Options	20
4.5	Debugging Options	21
4.6	Limit Options	22
5	Output Formats	24
5.1	Cayley Format	24
5.2	GAP format	25
5.3	Meataxe Format	26
5.4	AXIOM format	26
5.5	Plain ASCII format	27
5.6	Plain Binary Format	28
6	Strategy	29
6.1	Weights	29
6.2	Lookahead	30
7	The Source Code	31
7.1	Generalities	31
7.1.1	Important macros	31
7.1.2	Arithmetic	32
7.1.3	Prototyping and Other Language Issues	32
7.2	The Source Files	32
8	Examples	35
8.1	The natural permutation representation of S_3	35
8.2	A Quotient of a Permutation Representation	36
8.3	A Non-cyclic Module	37
8.4	A Monoid Representation	38
8.5	An Integer Module with Torsion	40
8.6	Quotient Construction Example	41
8.7	A Quotient of a Polynomial Ring	42
	Concept Index	44
	Options Index	47