

Basic Package Documentation

Version 1.0

May 2008

Tom Hoffman
Klaus Lux

Tom Hoffman — Email: thoffman@coastal.edu

Klaus Lux — Email: klux@math.arizona.edu

Abstract

This package calculates the basic algebra which is Morita equivalent to a block of a group algebra of a finite group in characteristic dividing the order of the group. GAP must be able to access the table of marks and the modular character table for the group. It is beneficial, but not required, for GAP to have access to the Atlas of Finite Group Representations via the AtlasRep package.

Copyright

© 2006 by Tom Hoffman and Klaus Lux

Acknowledgements

This documentation was prepared with the GAPDoc package of Frank Lübeck and Max Neunhöffer.

Contents

1	Introduction	5
1.1	Installation Instructions	5
1.2	The Algorithm	6
1.3	GAP data structure for basic algebras	6
2	User Functions	10
2.1	User Variables	10
2.1.1	BasicWorkingSpace	10
2.1.2	BasicMemAvailable	10
2.2	Fully automated calculation	11
2.2.1	AutoCalcBasic	11
2.2.2	InfoKond	11
3	Constructing Basic Algebras	13
3.1	Condensation	13
3.1.1	FindFaithBurn	13
3.1.2	FindCondSubgroup	15
3.1.3	GetPerms	16
3.1.4	FindAlgGens	17
3.1.5	Condense	17
3.2	Constructing Projective Indecomposable Modules	18
3.2.1	CondenseModuleTom	18
3.2.2	SpinUpPIMs	20
3.3	Constructing the Basic Algebra	20
3.3.1	FindBasicHoms	20
4	Other Functions in the Basic Package	22
4.1	Functions for Groups	22
4.1.1	FindOverGroupsTom	22
4.1.2	FindMaximalOverGroupsTom	22
4.2	Permutation Calculations	23
4.2.1	FindPerms	23
4.2.2	ImprimPerm	23
4.2.3	MakePermOnCond	24
4.2.4	Kond	25
4.2.5	MakeGensKondSubgroup	25
4.3	Algebra Calculations	25
4.3.1	GetConstList	25
4.3.2	NewSpinUp	26
4.3.3	BigSpin	26
4.3.4	QuiverToBasic	27
4.3.5	Spinning	27

4.3.6	MultiplyVecMat	27
4.3.7	MakeAction	27
4.3.8	ReadWords	28
4.3.9	ParseWord	28
4.3.10	WordPol	28
4.4	Miscellaneous Functions	29
4.4.1	CompressMat	29
4.4.2	UncompressMat	29
A	Variables	30
A.1	Miscellaneous Variables	30
A.1.1	CFInfo	30
A.1.2	MeatAxe	30
A.1.3	MesserHalter	30
B	A note on algebra generators	31

Chapter 1

Introduction

Let G be a finite group and p a prime dividing the order of G . We use FG to denote the group algebra of G with F a field of characteristic p . For the purposes of this package, F is a splitting field for FG . Since FG is a finite dimensional algebra, it can be decomposed into a direct sum of indecomposable two-sided ideals. These summands are called the blocks of FG . An algebra is called basic if modulo its radical it is a direct sum skew fields.

Given a group G , a prime p and a number b between 1 and the number of blocks of FG , this package constructs the basic algebra for the b^h block of FG . A description of the algorithm used by this package can be found in Section 1.2. The data structure used to store basic algebras in GAP is described in Section 1.3.

Chapter 2 documents the command `AutoCalcBasic` (2.2.1). This command automates the process of computing the basic algebra and should be sufficient for most users. Chapter 3 steps through the computation of basic algebras highlighting the important subroutines used by this package. In particular, following the outline of Chapter 3, a user could use a condensation subgroup different from the one chosen by `AutoCalcBasic` (2.2.1).

Chapter 4 is a reference chapter meant for expert users and documents other functions developed for this package. These functions are internal and are documented here for development purposes. Appendix A lists the global variables used by this package and Appendix B describes the labeling used for generators of the condensation subalgebra.

1.1 Installation Instructions

The Basic package uses external binaries and has only been tested on Linux systems. The `atlasrep` package of version 1.3 or higher is required by the Basic package and is available at <http://www.gap-system.org/Packages/packages.html>

The “Basic” package is bundled in two ways. First, the `.zoo` archive: ‘`basicpkg.zoo`’. This archive is unpacked by either the command

```
unzoo -x basicpkg.zoo
or
zoo -extract basicpkg.zoo
```

Second, the `.tar.gz` archive: ‘`basicpkg.tar.gz`’. This archive is unpacked by the command

```
tar -xvzf basicpkg.tar.gz
```

When either archive is unpacked, a directory `basic` is created. To complete the installation of the Basic package, go to the directory `basic` and call

```
/bin/sh ./configure path
```

where `path` is a path to the main GAP root directory. If you installed package in a local directory, `path` needs to be the full path to the main GAP root directory. If you are installing this package in the standard `pkg` directory, you should use

```
/bin/sh ./configure ../..
```

After successfully configuring the package, call

```
make install
```

to compile and install the binaries.

If you installed this package in a local directory, you will need to start GAP using the `-l` option for it to find the package. The path you give for the local directory needs to be the full path. For example, if you installed the package in the directory `/home/foo/somethingelse/mygap/pkg`, then starting GAP with the command

```
gap -l "/home/foo/somethingelse/mygap/"
```

will ensure that GAP will be able to find this package.

If you installed GAP on several architectures, you must execute the `configure/make` steps for the Basic package on each of the architectures after configuring GAP itself on this architecture.

The Basic package is loaded into GAP using the command `LoadPackage("basic");`

1.2 The Algorithm

This section gives a brief overview of the computations being performed by the Basic package. For details see [Hof04]. Let G be a group, F a field of characteristic p such that p divides the order of G , and b the number of a block of the group algebra FG , as described above. Again, the field F is a splitting field for FG .

For our purposes here, G can be either a group as recognized in GAP by `IsGroup` (**Reference: IsGroup**) or is the name of a group as recognized by the `atlasrep` package, see `DisplayAtlasInfo` (**AtlasRep: DisplayAtlasInfo**). The blocks of FG are numbered by their position in the list returned by the function `BlocksInfo` (**Reference: BlocksInfo**).

The algorithm used by the Basic package is

1. Find a condensation subgroup H for b^{th} block of FG (see `InfoKond` (2.2.2)).
2. Find a “small” generating set for the condensation subalgebra $eFGe$, where $e \in FG$ is the sum of elements in H divided by the order of H (see `FindAlgGens` (3.1.4)).
3. Construct the projective indecomposable $eFGe$ -modules, up to isomorphism, for the b^{th} block of $eFGe$ (see `SpinUpPIMs` (3.2.2)). Let n be the number of projective indecomposable $eFGe$ -modules, up to isomorphism and let P_i denote these modules for $1 \leq i \leq n$.
4. For $1 \leq i, j \leq n$, construct a basis for $Hom(P_i, P_j / Rad^2(P_j))$, i.e., the space of homomorphisms from P_i to P_j whose images are not zero modulo the radical of P_j squared. This package computes these homomorphisms as matrices (see `FindBasicHoms` (3.3.1)). These homomorphisms are a generating set for the basic algebra of the b^{th} block of FG .
5. For each $1 \leq i \leq n$, use a spinning algorithm to construct an ordered basis for P_i and construct matrices describing the action of the generators this ordered basis (see `QuiverToBasic` (4.3.4)).

For detailed descriptions of the algorithms used in the Basic package, see [Hof04] and [Lux97].

1.3 GAP data structure for basic algebras

This section describes the data structure being returned as the basic algebra. Let e_i label the identity map on P_i . Let Q be the directed graph with vertex set corresponding to the set of e_i and edges from e_j to e_i corresponding to the basis homomorphisms from P_i to P_j computed in 4 above. This directed graph is the quiver, or Ext-quiver, of the b^{th} block of the basic algebra of FG .

The data structure used in the Basic package for basic algebras is constructed in two pieces. First is the quiver which is stored as a record containing information about the group and group algebra necessary for the construction of the basic algebra. These components are:

group The name of the group as a string.

blocknumber The number of the block.

generators A list of the names of the generators of the basic algebra. This includes the idempotents corresponding to projective indecomposable modules and the required maps between them.

npims The number of projective indecomposable modules in the block of the group algebra.

pimnames A list of the names of the projective indecomposable modules in the block of the condensation algebra. These names follow the Atlas naming convention.

cartan The cartan matrix of the block of the group algebra.

field The splitting field of the group.

dim A list of the dimensions of the projective indecomposable modules.

adjmat The adjacency matrix of the projective indecomposable modules. The $(i, j)^{th}$ entry is defined to be the number homomorphisms $P_i \rightarrow Rad(P_j)/Rad^2(P_j)$ where P_i and P_j are projective indecomposable modules in *pimnames*.

There are also components for each of the generators. These components are themselves records describing the action of these elements on the Ext-quiver. The components of these records are:

start The position in the list *pimnames* of the PIM for which this generator is not the zero map.

ende The position in the list *pimnames* of the PIM which contains the image of this generator.

name The name of this generator.

mat A matrix describing the action of this generator.

The basic algebra is constructed from the quiver and recorded by appending the field *matrices* to the record for the quiver. The field *matrices* is itself a record with components for each of the projective indecomposable modules. These components are themselves records with components:

n where n is an integer from one up to the number of generators of the basic algebra, is a compressed matrix describing the action of the corresponding generator of the basic algebra on this projective indecomposable module.

spinningtree is a list of records describing the details of the spinning process used to generate the basis of the projective indecomposable module.

perm is a list which as a permutation, see `PermList` (**Reference: PermList**), describes how *spinningtree* has been reordered from the original construction.

To help with the understanding of this structure, the following is the basic algebra data structure for the principal block of the group A_5 in characteristic 3.

```
basicalg:=rec(
  group := "A5",
  generators := [ "1a", "1b", "1a1b1", "1b1a1" ],
  npims := 2,
  pimnames := [ "1a", "1b" ],
  cartan := [ [ 2, 1 ], [ 1, 2 ] ],
  field := GF(3),
  dim := [ 3, 3 ],
  adjmat := [ [ 0, 1 ], [ 1, 0 ] ],
  1a := rec(
    start := 1,
    ende := 1,
```

```

name := "id1a",
mat := [ [ Z(3)^0, 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3) ],
         [ 0*Z(3), 0*Z(3), Z(3)^0 ] ] ),
1b := rec(
  start := 2,
  ende := 2,
  name := "id1b",
  mat := [ [ Z(3)^0, 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3) ],
           [ 0*Z(3), 0*Z(3), Z(3)^0 ] ] ),
1a1b1 := rec(
  start := 2,
  ende := 1,
  name := "1a1b1",
  mat := [ [ Z(3)^0, Z(3)^0, Z(3)^0 ], [ Z(3), Z(3)^0, 0*Z(3) ],
           [ Z(3)^0, Z(3), 0*Z(3) ] ] ),
1b1a1 := rec(
  start := 1,
  ende := 2,
  name := "1b1a1",
  mat := [ [ Z(3), Z(3)^0, 0*Z(3) ], [ Z(3), 0*Z(3), Z(3) ],
           [ Z(3), Z(3)^0, 0*Z(3) ] ] ),
matrices := rec(
  pim1a := rec(
    1 := [ [ 1, 1, Z(3)^0 ], [ 2, 2, Z(3)^0 ] ],
    2 := [ [ 3, 3, Z(3)^0 ] ],
    3 := [ [ 3, 2, Z(3)^0 ] ],
    4 := [ [ 1, 3, Z(3)^0 ] ],
    perm := [ 1, 3, 2 ],
    spinningtree := [ rec(
      ende := 1,
      name := [ ],
      tree := [ ] ), rec(
      ende := 1,
      name := [ "1b1a1", "1a1b1" ],
      tree := [ 3, 3 ] ), rec(
      ende := 2,
      name := [ "1b1a1" ],
      tree := [ 1, 4 ] ) ] ),
  pim1b := rec(
    1 := [ [ 1, 1, Z(3)^0 ] ],
    2 := [ [ 2, 2, Z(3)^0 ], [ 3, 3, Z(3)^0 ] ],
    3 := [ [ 2, 1, Z(3)^0 ] ],
    4 := [ [ 1, 3, Z(3)^0 ] ],
    perm := [ 2, 1, 3 ],
    spinningtree := [ rec(
      ende := 1,
      name := [ "1a1b1" ],
      tree := [ 2, 3 ] ), rec(
      ende := 2,
      name := [ ],
      tree := [ ] ), rec(
      ende := 2,
      name := [ "1a1b1", "1b1a1" ],
      tree := [ 1, 4 ] ) ] ) ), blocknumber := 1 );

```


This record gives $\{1a, 1b, 1a1b1, 1b1a1\}$ as a generating set for the basic algebra of the principal block of A_5 in characteristic 3. While these generators all have the same dimension, pay attention to the *start* and *ende* components. The data in `matrices.pim1a` can be used to reconstruct the projective indecomposable module $1a$ as follows. Start with the vector $v_1 = (1, 0, 0)$ at $1a$. Now, `matrices.pim1a.perm` is the permutation $(2, 3)$, which tells us that the second and third entries in `matrices.pim1a.spinningtree` are reversed from the original construction order. Let $v_3 = v_1 * 1b1a1.mat = (2, 1, 0)$ and $v_2 = v_3 * 1a1b1.mat = (0, 0, 2)$. The vectors v_1, v_2, v_3 form an ordered basis for $pim1a$, the projective cover of $1a$. The following example gives this computation in GAP using the above record and the function `MultiplyVecMat` (4.3.6).

Example

```
gap> v1:=rec(ende:=1,name:=[],vec:=[1,0,0]);
rec( ende := 1, name := [ ], vec := [ 1, 0, 0 ] )
gap> v3:=MultiplyVecMat(v1,basicalg.1b1a1);
rec( vec := [ Z(3), Z(3)^0, 0*Z(3) ], ende := 2, name := [ "1b1a1" ] )
gap> v2:=MultiplyVecMat(v3,basicalg.1a1b1);
rec( vec := [ Z(3)^0, 0*Z(3), Z(3) ], ende := 1, name := [ "1b1a1", "1a1b1" ] )
```

Notice that the vectors v_1 and v_2 are at $1a$ and v_3 is at $1b$. This is important for writing the action of the generators on the basis. For instance, the generator $1a$ fixes the first two vectors and sends v_3 to 0, whereas $1b$ sends the first two vectors to 0 and fixes the third.

Chapter 2

User Functions

For most users, this chapter contains everything you need to calculate the basic algebra which is Morita equivalent to a block of a group algebra.

2.1 User Variables

The variables in this section are given default values that should work for most users. Users should feel free to adjust these variables to suit their needs. The variables `BasicWorkingSpace` (2.1.1) and `BasicMemAvailable` (2.1.2) describe where external files will be saved and how much memory the Basic package assumes is available on your computer. Make sure that `BasicWorkingSpace` (2.1.1) points to a directory that has plenty of free space. This package creates a lot of temporary files, so it is a bad idea to redirect this variable to a directory containing files that will get lost among the garbage.

`AutoCalcBasic` (2.2.1) will change its methods depending on the amount of memory available and the amount of memory it thinks it will need. The differences between these methods are trading time for space. So increasing `BasicMemAvailable` (2.1.2) can help with larger groups. One should keep in mind that `BasicMemAvailable` (2.1.2) should not be the total memory on the machine (please leave some room for at least the operating system).

2.1.1 BasicWorkingSpace

◇ `BasicWorkingSpace` (global variable)

BasicWorkingSpace is the directory where the data files for the MeatAxe will be stored. After loading the Basic package, you can change this directory by making it a readwrite variable using the function `MakeReadWriteGlobal` (**Reference: MakeReadWriteGlobal**) and then using the function `Directory` (**Reference: Directory**) as in the following example.

Example

```
gap> BasicWorkingSpace;  
dir("/tmp/tmp.JmXqQu/")  
gap> MakeReadWriteGlobal("BasicWorkingSpace");  
gap> BasicWorkingSpace:=Directory("/tmp/GapData/");  
dir("/tmp/GapData/")
```

2.1.2 BasicMemAvailable

◇ `BasicMemAvailable` (global variable)

BasicMemAvailable is a list where the second entry is the maximum amount of memory available to GAP in kilobytes. The basic package assumes this is also the maximum amount of memory available to

the MeatAxe. If this is not the case, this number should be changed. The amount of memory available to the MeatAxe may change some of the methods used by the basic package. If you want GAP to access more memory, see `Command line options` (**Reference: Command Line Options**).

2.2 Fully automated calculation

To use the automated form of the Basic package, you need three pieces of data. First, you need a group. If possible, you should use the Atlas name for your group. For a list of all the named groups see `DisplayAtlasInfo` (**AtlasRep: DisplayAtlasInfo**). If your group does not appear in this list, or you have a different description for some reason, you can use a group as long as `IsGroup` (**Reference: IsGroup**) returns true.

The argument *prime* needs to be an integer such that `IsPrime` (**Reference: IsPrime**) returns true and which divides the order of your group.

The integer *blocknumber* needs to be between 1 and the length of the list returned by `BlocksInfo` (**Reference: BlocksInfo**). While this program will return the basic algebra for any block, the blocks of defect zero are trivial and can be computed faster by hand.

2.2.1 AutoCalcBasic

◇ `AutoCalcBasic(group, prime, blocknumber)` (function)

◇ `AutoCalcBasic(groupname, prime, blocknumber)` (function)

Returns: a list containing the basic algebra record as the third element. The first entry in this list describes the generators of the condensation algebra, see `FindAlgGens` (3.1.4), and the second entry gives the words for the generators for the condensation subgroup inside its normalizer and also the words for any *prime* Sylow elements that might have been used, see `FindCondSubgroup` (3.1.2)

This function can be called either with a group *group* or with the Atlas name of a group, *groupname*. For a list of named groups, see `DisplayAtlasInfo` (**AtlasRep: DisplayAtlasInfo**). `AutoCalcBasic` is the automation of the Basic package. The idea is to construct the *Projective Indecomposable Modules* (PIMs) contained in the *blocknumber* block of a condensation algebra and use the relevant homomorphisms between these PIMs to construct the basic algebra. Blocks are labeled as returned by `BlocksInfo` (**Reference: BlocksInfo**). *prime* is the characteristic of the basic algebra and should divide the order of *groupname* or *group*.

Example

```
gap> basalg:=AutoCalcBasic("M11",2,1);;
gap> basalg[1];
[ rec( genlist := [ 2, 6 ] ) ]
gap> basalg[2];
[ [ "ab^{2}a^{-1}b^{2}", "a^{-2}" ],
  [ "a^{2}ba^{-2}", "aba^{-2}b^{-1}", "a^{-1}bab", "b^{2}a^{2}" ] ]
gap> RecNames(basalg[3]);
[ "group", "generators", "npims", "pimnames", "cartan", "field", "dim",
  "adjmat", "1a", "2a", "4a", "1a2a1", "1a4a1", "2a1a1", "2a2a1", "4a1a1",
  "4a4a1", "matrices" ]
```

See also `BlocksInfo` (**Reference: BlocksInfo**), `DisplayAtlasInfo` (**AtlasRep: DisplayAtlasInfo**), `FindAlgGens` (3.1.4), `FindCondSubgroup` (3.1.2), and `RecNames` (**Reference: RecNames**).

Before using the function `AutoCalcBasic` (2.2.1) with a particular group, it might be useful to run `InfoKond` (2.2.2). The condensation subgroup used by `AutoCalcBasic` (2.2.1) is the one returned by `InfoKond` (2.2.2). For information on using a different condensation subgroup, see `FindFaithBurn` (3.1.1).

2.2.2 InfoKond

◇ `InfoKond(group, prime, blocknumber)` (function)

◇ `InfoKond(groupname, prime, blocknumber)` (function)

Returns: a record containing information about a particular condensation subalgebra of the group algebra. The components of this record are:

permchar the permutation character of the conjugacy class of the condensation subgroup.

norm the scalar product of *permchar* with itself.

numberinburn the number in the table of marks of the conjugacy class of the condensation subgroup.

cartan the cartan matrix for the block of the group algebra given by *blocknumber*.

dimmodchars the dimensions of the modular characters of the group which are in the block.

splitf the splitting field of the group.

dimfixmod the dimensions of the simple modules in the condensed algebra.

dimfixproj the dimensions of the projective indecomposable modules in the condensed algebra.

This function uses the record returned by `FindFaithBurn` (3.1.1) to select a condensation subgroup which will give the smallest condensation algebra for the block *blocknumber* of the group algebra. `InfoKond` can be called with either *group* being a group, or *groupname* a string name of a group. Using either method, the group being referred to must have both a character table, see `CharacterTable` (**Reference: CharacterTable**), and a table of marks, see `TableOfMarks` (**Reference: TableOfMarks**), in GAP. *prime* is a prime number which divides the order of the group.

Example

```
gap> grpinfo:=InfoKond("M11",2,1);
rec( permchar := [ 880, 0, 16, 0, 0, 0, 0, 0, 0, 0 ], norm := [ [ 112 ] ],
    numberinburn := 13, cartan := [ [ 4, 2, 2 ], [ 2, 5, 1 ], [ 2, 1, 3 ] ],
    dimmodchars := [ 1, 10, 44 ], splitf := 2, dimfixmod := [ 1, 2, 4 ],
    dimfixproj := [ 16, 16, 16 ] )
```

Chapter 3

Constructing Basic Algebras

The previous chapter covered constructing basic algebras in a completely automated manner. In this chapter, our process is covered in smaller pieces. This section is intended for more advanced users who may want to apply these programs in more general situations, or with different condensation subgroups.

Our method for constructing basic algebras from groups breaks up into three components. These components are condensation, constructing the projective indecomposable modules, and building the quiver and algebra. We divide this chapter into three sections, one for each component. These sections will be as independent as possible from each other, so the reader should feel free to jump to whichever section is of interest.

This chapter contains many references to GAP's table of marks and associated programs for retrieving data from tables of marks. A reference for the theory of tables of marks and how to construct them is [Pfe97].

3.1 Condensation

In the Basic package, we use fixed point condensation which was first introduced in [Tha81] and has been more recently discussed in [Lux97] and [Hof04].

For this section, we are assuming that G is a group or an Atlas name for a group, p is a prime integer which divides the order of G , and b is a number corresponding to a block of G , see `BlocksInfo` (Reference: `BlocksInfo`).

The first step in condensation is to find a condensation subgroup. To do this, we use the program `FindFaithBurn` (3.1.1), which relies on theory appearing in [Lux97]. `FindFaithBurn` (3.1.1) returns a record containing information on possible condensation subgroups, listed in order of increasing size. `InfoKond` (2.2.2) calls `FindFaithBurn` (3.1.1) and picks the last (largest size) subgroup in the record.

3.1.1 FindFaithBurn

◇ `FindFaithBurn(group, prime, blocknumber)` (function)
◇ `FindFaithBurn(groupname, prime, blocknumber)` (function)

Returns: a record similar to the one returned by `InfoKond` (2.2.2) except that this one contains the information for all possible condensation subgroups given $prime$ and $blocknumber$. The components of this record are:

permchars the permutation characters of the conjugacy classes of the possible condensation subgroups.

numbersinburn the numbers in the table of marks of the conjugacy classes of the possible condensation subgroups.

cartan the cartan matrix for the block of the group algebra given by $blocknumber$.

splitf the splitting field of the group.

dimmodchars the dimensions of the modular characters for *group* which are in block *blocknumber*.

dimfixmod the dimensions of the simple modules in the condensed algebra. Each row corresponds to a possible condensation subgroup.

dimfixproj the dimensions of the projective indecomposable modules in the condensed algebra. Each row corresponds to a possible condensation subgroup.

FindFaithBurn uses the characters and modular characters for the block *blocknumber* and the table of marks to find all the possible condensation subgroups and the corresponding information as returned above. The code for this function is given in [Lux97]. In the first form, *group* is taken as a permutation group and in the second form *groupname* is the string name of a group which must have both a character table, see `CharacterTable` (**Reference: CharacterTable**), and a table of marks, see `TableOfMarks` (**Reference: TableOfMarks**), in GAP. *prime* is a prime number which divides the order of the group with name *groupname*.

Example

```
gap> ginfo:=FindFaithBurn("M11",2,1);
rec(
  permchars := [ [ 7920, 0, 0, 0, 0, 0, 0, 0, 0, 0 ], [ 2640, 0, 12, 0, 0, 0,
    0, 0, 0, 0 ], [ 1584, 0, 0, 0, 4, 0, 0, 0, 0, 0 ],
    [ 880, 0, 16, 0, 0, 0, 0, 0, 0, 0 ] ], numbersinburn := [ 1, 3, 6, 13 ],
  cartan := [ [ 4, 2, 2 ], [ 2, 5, 1 ], [ 2, 1, 3 ] ], splitf := 2,
  dimmodchars := [ 1, 10, 44 ],
  dimfixmod := [ [ 1, 10, 44 ], [ 1, 4, 14 ], [ 1, 2, 8 ], [ 1, 2, 4 ] ],
  dimfixproj := [ [ 112, 96, 144 ], [ 40, 36, 48 ], [ 24, 20, 28 ],
    [ 16, 16, 16 ] ] )
gap> InfoKond("M11",2,1);
rec( permchar := [ 880, 0, 16, 0, 0, 0, 0, 0, 0, 0 ], norm := [ [ 112 ] ],
  numberinburn := 13, cartan := [ [ 4, 2, 2 ], [ 2, 5, 1 ], [ 2, 1, 3 ] ],
  dimmodchars := [ 1, 10, 44 ], splitf := 2, dimfixmod := [ 1, 2, 4 ],
  dimfixproj := [ 16, 16, 16 ] )
gap> info:=rec(permchar:=ginfo.permchars[3],
  numberinburn:=ginfo.numberinburn[3],
  cartan:=ginfo.cartan,
  dimmodchars:=ginfo.dimmodchars,
  dimfixmod:=ginfo.dimfixmod[3],
  dimfixproj:=ginfo.dimfixproj[3]);
gap> ct:=CharacterTable("M11");
CharacterTable( "M11" )
gap> info.norm:=MatScalarProducts(ct,[info.permchar],[info.permchar]);
[ [ 320 ] ]
gap> info;
rec( permchar := [ 1584, 0, 0, 0, 4, 0, 0, 0, 0, 0 ], numberinburn := 6,
  cartan := [ [ 4, 2, 2 ], [ 2, 5, 1 ], [ 2, 1, 3 ] ],
  dimmodchars := [ 1, 10, 44 ], dimfixmod := [ 1, 2, 8 ],
  dimfixproj := [ 24, 20, 28 ], norm := [ [ 320 ] ] )
```

Note that condensation subgroups are not returned by `FindFaithBurn` (3.1.1). We get a list of numbers describing a conjugacy class of subgroups in GAP's Table of Marks, see `TableOfMarks` (**Reference: TableOfMarks**). When we select a number, we could use the representative of this subgroup which is stored in GAP's Table of Marks. However, whenever possible, we take advantage of *Theorem 3.1* in [LW01] which states that a condensation subgroup can be extended by certain normalizing elements while verifying generation of the condensation subalgebra. The function `FindCondSubgroup` (3.1.2) returns the information we need to apply this theorem.

3.1.2 FindCondSubgroup

◇ FindCondSubgroup(*group*, *prime*, *blocknumber*, *groupinfo*) (function)

◇ FindCondSubgroup(*groupname*, *prime*, *blocknumber*, *groupinfo*) (function)

Returns: a record containing the information about the condensation subgroup necessary for constructing generators for the condensation subalgebra of the group algebra. The components of this record are:

SimpleRep the conjugacy class of subgroups in the table of marks which correspond to a module for the group algebra which contains all the simple modules which also appear in the condensation algebra of the group algebra.

Normalizer the conjugacy class of subgroups in the table of marks which contains the normalizer of the condensation subgroup.

kdim the dimension of the module corresponding to the subgroup generated by the condensation subgroup and the normalizing elements.

gens a list of 2 lists of straight line programs, see StraightLineProgram (**Reference: StraightLine-Program**), in terms of the generators of the normalizer of the condensation subgroup given in the table of marks. The first list gives the generators of the condensation subgroup and the second gives elements which normalize the condensation subgroup and have order divisible by *prime*.

strline human readable words, formatted for L^AT_EX, in *a,b,...* corresponding to the straight line programs in *gens*.

horder the order of the condensation subgroup.

korder the order of the subgroup generated by the condensation subgroup and the normalizing elements described in *gens*.

script true if the generators for *groupname* stored in the table of marks, see UnderlyingGroup (**Reference: UnderlyingGroup!for tables of marks**), agree with the standard generators in the atlasrep package, see IsStandardGeneratorsOfGroup (**Reference: IsStandardGeneratorsOf-Group**). Otherwise false.

The main purpose of this function is to deal with the fact that the table of marks contains information for conjugacy classes of subgroups, not subgroups. This is an issue since we use the table of marks to find the normalizer of the condensation subgroup, so we find the conjugate which is a subgroup of the normalizer given by table of marks. The rest of the information comes from straightforward character and modular character calculations. FindCondSubgroup can be called either with *group* a group, or *groupname*, the Atlas name of a group, see DisplayAtlasInfo (**AtlasRep: DisplayAtlasInfo**), as a string. *prime* is a prime number which divides the order of the group corresponding to *groupname*. *blocknumber* is an integer corresponding to a block of the group, as listed by BlocksInfo (**Reference: BlocksInfo**). *groupinfo* is a record containing condensation information about the group. This can be the record returned by InfoKond (2.2.2) or a similarly constructed record using data from FindFaithBurn (3.1.1).

Example

```
gap> info:=InfoKond("M11",2,1);
rec( permchar := [ 880, 0, 16, 0, 0, 0, 0, 0, 0, 0 ], norm := [ [ 112 ] ],
    numberinburn := 13, cartan := [ [ 4, 2, 2 ], [ 2, 5, 1 ], [ 2, 1, 3 ] ],
    dimmodchars := [ 1, 10, 44 ], splitf := 2, dimfixmod := [ 1, 2, 4 ],
    dimfixproj := [ 16, 16, 16 ] )
gap> FindCondSubgroup("M11",2,1,info);
rec( SimpleRep := 28, SimpleRepMult := [ 4, 2, 2 ], Normalizer := 35,
    kdim := 7, knumber := 35,
    gens := [ [ <straight line program>, <straight line program> ],
    [ <straight line program>, <straight line program>,
    <straight line program>, <straight line program> ] ],
```

```

    strline := [ [ "ba^{-2}b^{-1}", "a^{2}" ],
      [ "a^{2}ba^{-2}", "aba^{-2}b^{-1}", "a^{-1}bab", "b^{2}a^{2}" ] ],
    horder := 9, korder := 144, script := true )
gap> ginfo:=FindFaithBurn("M11",2,1);
gap> info:=rec(permchar:=ginfo.permchars[3],
numberinburn:=ginfo.numbersinburn[3],
cartan:=ginfo.cartan,
dimmodchars:=ginfo.dimmodchars,
dimfixmod:=ginfo.dimfixmod[3],
dimfixproj:=ginfo.dimfixproj[3]);
gap> ct:=CharacterTable("M11");
CharacterTable( "M11" )
gap> info.norm:=MatScalarProducts(ct,[info.permchar],[info.permchar]);
[ [ 320 ] ]
gap> FindCondSubgroup("M11",2,1,info);
rec( SimpleRep := 28, SimpleRepMult := [ 4, 2, 2 ], Normalizer := 21,
    kdim := 80, knumber := 21,
    gens := [ [ <straight line program> ], [ <straight line program>,
      <straight line program> ] ],
    strline := [ [ "b^{-1}ab^{-1}" ], [ "b", "b^{2}" ] ], horder := 5,
    korder := 20, script := true )

```

`FindCondSubgroup` (3.1.2) returns straight line programs, see `StraightLineProgram` (Reference: **StraightLineProgram**), for the generators of a condensation subgroup and any suitable normalizing elements in terms of the generators of its normalizer. Now we construct permutations for the action of the group on this condensation subgroup and for the group on the subgroup containing the condensation subgroup and the normalizing elements returned by `FindCondSubgroup` (3.1.2). The function `GetPerms` (3.1.3) returns these permutations.

3.1.3 GetPerms

◇ `GetPerms(group, groupinfo, subgroupinfo)` (function)

◇ `GetPerms(groupname, groupinfo, subgroupinfo)` (function)

Returns: a list of the form $[[2, hperms], [2, kperms]]$ where $hperms$ are the permutations of $group$, or $groupname$, acting on the cosets of the condensation subgroup and $kperms$ are the permutations of $group$, or $groupname$, acting on the cosets of the subgroup generated by the condensation subgroup and the normalizing elements returned by `FindCondSubgroup` (3.1.2), see Theorem 3.1 in [LW01].

$groupinfo$ and $subgroupinfo$ are records as returned by `InfoKond` (2.2.2) and `FindCondSubgroup` (3.1.2), respectively.

Example

```

gap> grpinfo:=InfoKond("M11",5,1);
[ 45, 55, 60, 60 ]
gap> subinfo:=FindCondSubgroup("M11",5,1,grpinfo);
gap> GetPerms("M11",grpinfo,subinfo);

```

See also `InfoKond` (2.2.2) and `FindCondSubgroup` (3.1.2).

The permutations returned by `GetPerms` (3.1.3) are important since they allow us to construct a condensation subalgebra which is Morita equivalent to the group algebra without first constructing the group algebra. For details on how this is done see [Lux97] or [Hof04].

Our next step is finding a small generating set for this condensation subalgebra. The idea is to construct “random” elements in the subalgebra and check if they generate the full space. If not, we add more elements and check again. Once a generating set is found, the elements are checked one at a time to remove redundant generators. This work is done by the function `FindAlgGens` (3.1.4).

3.1.4 FindAlgGens

◇ FindAlgGens(*grp*, *prime*, *condperms*, *normperms*, *dim*, *numsylgens*, *grpinfo*)
(function)

◇ FindAlgGens(*grpname*, *prime*, *condperms*, *normperms*, *dim*, *numsylgens*, *grpinfo*)
(function)

Returns: a list of records describing the generators for the condensation subalgebra. See Appendix B to see how these algebra elements are named. The components of these records are:

genlist a list of integers from the set $\{1, \dots, 11\}$ denoting which generators are being used.

resstrline the permutations which give the generators for the condensation algebra with normalizing elements.

bigresstrline the permutations which give the generators for the condensation algebra.

The union of all of the elements described in these records is our generating set for the condensation algebra.

While the actual generators are not returned by FindAlgGens, they have been constructed. The MeatAxe matrices, in sparse format, for these generators, and the normalizing elements, are in the directory BasicWorkingSpace (2.1.1) with the names *z.i* where *i* runs through the numbers 1 up to the number of normalizing elements plus the number of generators.

condperms and *normperms* are lists of permutations as returned by MakePermOnCondNorm (4.2.3), entries 3 and 2 respectively. *dim* is the dimension condensation algebra over the splitting field and *numsylgens* is the number of normalizing elements being used. This function finds a generating set by trial and error. A few algebra elements are constructed and then checked to see if they generate the algebra. If not, more generators are added and the process is repeated. Once a generating set is found, the elements are systematically checked to see if they are necessary. The generating set returned is minimal in that no proper subset will generate the space. If *numsylgens* > 0, then the calculations are not done in the regular representation of the condensed algebra, but in a smaller dimensional module as described by [LW01].

Example

```
gap> grpinfo:=InfoKond("M11",2,1);;
gap> subinfo:=FindCondSubgroup("M11",2,1,grpinfo);;
gap> perms:=GetPerms("M11",grpinfo,subinfo);;
gap> alggens:=FindAlgGens("M11",2,perms[1],perms[2],subinfo.kdim,
> Length(subinfo.gens[2]),grpinfo);;
gap> alggens[1].genlist;
[ 2, 6 ]
```

See also FindCondSubgroup (3.1.2), InfoKond (2.2.2), GetPerms (3.1.3), and MakePermOnCondNorm (4.2.3).

Now that we have a small generating set for the condensation subalgebra, the function Condense (3.1.5) constructs elements in the condensation subalgebra corresponding to elements in the group.

3.1.5 Condense

◇ Condense(*permutations*, *elements*, *fieldsize*) (function)

Returns: "sparse" or "matrix" depending on which MeatAxe format is used to store the constructed elements.

Condense condenses the permutations *elements* using the subgroup generated by the permutations *permutations*. The constructed MeatAxe matrices are over a field of size *fieldsize* and are named *Kdg.i*, where *i* goes from 1 up to the size of *elements*, and are stored in BasicWorkingSpace (2.1.1).

Example

```
gap> tom:=TableOfMarks("M11");;
gap> InfoKond("M11",5,1);;
gap> last.numberinburn;
```

```

18
gap> gp:=UnderlyingGroup(tom);
Group([ (1,2) (3,5) (4,9) (6,10), (1,3,4,7) (2,12,10,11) ])
gap> groupperms:=GeneratorsOfGroup(gp);
[ (1,2) (3,5) (4,9) (6,10), (1,3,4,7) (2,12,10,11) ]
gap> subgroup:=RepresentativeTomByGeneratorsNC(tom,18,groupperms);
Group([ (2,10) (4,5) (6,7) (9,11), (3,6,12,11) (4,9,5,7) ])
gap> subgroupperms:=GeneratorsOfGroup(subgroup);
[ (2,10) (4,5) (6,7) (9,11), (3,6,12,11) (4,9,5,7) ]
gap> Condense(subgroupperms,groupperms,5);
"matrix"

```

See also `TableOfMarks` (**Reference:** `TableOfMarks`), `InfoKond` (2.2.2), `UnderlyingGroup` (**Reference:** `UnderlyingGroup`!for tables of marks), `GeneratorsOfGroup` (**Reference:** `GeneratorsOfGroup`), `RepresentativeTomByGeneratorsNC` (**Reference:** `RepresentativeTomByGeneratorsNC`), and `Flat` (**Reference:** `Flat`).

Given a *group* as either an object in GAP for which `IsGroup` (**Reference:** `IsGroup`) returns true or a string naming a group in the list returned by `DisplayAtlasInfo` (**AtlasRep:** `DisplayAtlasInfo`), we can construct a condensation subalgebra of the group algebra of *group* in characteristic dividing the order of *group*. The next section describes functions for working in the condensation subalgebra.

3.2 Constructing Projective Indecomposable Modules

We construct the Projective Indecomposable Modules, or PIMs, of the condensation subalgebra using peakwords, as described in [Lux97] and [Hof04]. The first step in this process is to find the peakwords for each of the simple modules corresponding to the PIMs. In the simplest case, we would use the `MeatAxe` to chop the condensation subalgebra. As the dimension of the condensation subalgebra increases, chopping it becomes less reasonable. Fortunately, in many cases, there are smaller algebras which still contain all of the simple modules (possibly not the PIMs).

Much of the information we use to construct modules is contained in GAP's table of marks. We find the largest subgroup (actually, conjugacy class of subgroups) such that the action of the group on the cosets of this subgroup contains all the simple modules of the condensation subalgebra. The record returned by `FindCondSubgroup` (3.1.2) has a component called *SimpleRep* which is the number of this conjugacy class in GAP's table of marks. We use the function `CondenseModuleTom` (3.2.1) to construct `MeatAxe` matrices for this module in the directory `BasicWorkingSpace` (2.1.1).

3.2.1 CondenseModuleTom

◇ `CondenseModuleTom(grp, prime, makenumber, condnumber, algebragens, subgrpinfo)` (function)

Returns: nothing in GAP. *CondenseModuleTom* constructs the module corresponding to the subgroup given by *makenumber* in the `BasicWorkingSpace` (2.1.1) as `MeatAxe` matrices named *mod.i* where *i* ranges over the number of generators for the condensation subalgebra.

CondenseModuleTom produces the condensed module of *grp* restricted to the subgroup with the number *makenumber* in `TableOfMarks` (**Reference:** `TableOfMarks`). *makenumber* must be greater than one. *grp* is a string of the group name. *prime* is the characteristic of the condensation subalgebra. *condnumber* is the number in `TableOfMarks` (**Reference:** `TableOfMarks`) of the condensation subgroup. *algebragens* is a list of the words, see Appendix B, which together with normalizing elements returned by `FindCondSubgroup` (3.1.2) describe a generating set of the condensation subalgebra. *subgrpinfo* is record of information about the condensation subgroup as returned by `FindCondSubgroup` (3.1.2).

Example

```

gap> grpinfo:=InfoKond("M11",2,1);;
gap> subinfo:=FindCondSubgroup("M11",2,1,grpinfo);;

```

```
gap> alggens:=[];;
gap> alggens[1]:=rec();;
gap> alggens[1].genlist:=[2,6];;
gap> CondenseModuleTom("M11",2,subinfo.SimpleRep,grpinfo.numberinburn,
gap> alggens,subinfo);
```

After executing the example above, there are six MeatAxe matrices labelled *mod.i* in the directory BasicWorkingSpace (2.1.1) even though only two algebra generators were given as input. The first four matrices correspond to the four normalizing elements listed in the record *subinfo*.

See also InfoKond (2.2.2), FindCondSubgroup (3.1.2), and Records (Reference: Records).

Now that we can build a module containing all the simple modules for the condensation subalgebra, we call on the MeatAxe to chop this module. Continuing the example above we get:

```
Example
gap> Process(BasicWorkingSpace, chop, NoneIn, NoneOut, ["-g", "6", "mod"]);
0
```

Now the file *mod.cfinfo* contains the composition data for the module *mod*.

The function SpinUpPIMs (3.2.2) constructs the PIMs of our condensation algebra. Before this is done, we need a representation containing all of the PIMs. The regular representation satisfies this condition. The function Condense (3.1.5) is used to build the regular representation as in the following example.

```
Example
gap> grpinfo:=InfoKond("M11",2,1);;
gap> subinfo:=FindCondSubgroup("M11",2,1,grpinfo);;
gap> perms:=GetPerms("M11",grpinfo,subinfo);;
gap> Length(subinfo.gens[2]);
4
gap> Length(perms[1][2]);
10
gap> gens:=perms[1][2]{{(10-4+1)..10}};;
gap> alggens:=FindAlgGens("M11",2,perms[1],perms[2],subinfo.kdim,4,grpinfo);;
gap> alggens[1].genlist;
[ 2, 6 ]
gap> Append(gens,[alggens[1].bigresstrline[2]]);
gap> Append(gens,[alggens[1].bigresstrline[6]]);
gap> permlist:=[perms[1][1],perms[1][2]{{1..(10-4)}}];;
gap> Condense(permlist,gens,grpinfo.splitf);
"sparse"
```

After working through the previous example, we have six matrices labelled *Kdg.1* through *Kdg.6* in the directory BasicWorkingSpace (2.1.1). These matrices are text files ready to be converted into MeatAxe matrices. When calling SpinUpPIMs (3.2.2), these matrices need to be in MeatAxe format, but the computations will be faster if they are in sparse format (as indicated by the return value of Condense (3.1.5)). The following example calls the MeatAxe functions *zcv* and *sparse2mat* from GAP to convert the files created in the previous example.

```
Example
gap> for i in [1..6] do
> options=["-Q", Concatenation("Kdg.",String(i)),
> Concatenation("sx.",String(i))];
> Process(BasicWorkingSpace,zcv,NoneIn,NoneOut,options);
> options=["-Q", Concatenation("sx.",String(i)),
> Concatenation("x.",String(i))];
> Process(BasicWorkingSpace,sparse2mat,NoneIn,NoneOut,options);
> od;
```

In the last few examples we constructed MeatAxe matrices for three representations named *mod*, *x*, *sx*. These representations, respectively, are a representation containing all simple $eM_{11}e$ modules in the principal block of $eM_{11}e$, and the regular representation of $eM_{11}e$ in both standard and sparse MeatAxe formats. Now we call SpinUpPIMs (3.2.2) to construct the PIMs of $eM_{11}e$ in the principal block.

3.2.2 SpinUpPIMs

◇ `SpinUpPIMs(repr, srepr, wordrepr, block, grpinfo, wordmult)` (function)

Returns: a list of composition factors of *repr* whose corresponding PIMs have been constructed in the directory `BasicWorkingSpace` (2.1.1).

srepr is the MeatAxe sparse matrix representation of the MeatAxe matrix representation *repr* which contains all the projective indecomposable modules corresponding to its simple modules. *wordrepr* is a MeatAxe matrix representation which contains all the simple *repr* modules. *block* is the number of the block containing the PIMs being constructed. *grpinfo* is the corresponding record returned by `InfoKond` (2.2.2). *wordmult* is a list of the multiplicities of the simple modules in the representation *wordrepr* and can be gotten . This list is returned by `InfoKond` (2.2.2) as the record component *dimfixproj* for modules of the condensation algebra. *SpinUpPIMs* constructs the projective indecomposable modules of *repr* which are in block *block* using the peakwords obtained from *wordrepr*. Continuing the example from above, we have:

Example

```
gap> SpinUpPIMs("x", "sx", "mod", 1, grpinfo, [4, 2, 2]);
[ "1a", "2a", "4a" ]
```

The files in `BasicWorkingSpace` (2.1.1) named *pimn.i*, where *n* is the name of simple module and *i* is a number between one and the number of generators of the algebra, are the MeatAxe matrices for the PIMs.

3.3 Constructing the Basic Algebra

The main idea for constructing the basic algebra is to realize it as the endomorphism ring of a progenerator. As in [Hof04], the endomorphism ring of progenerator is generated by homomorphisms from the heads of PIMs into the second Loewy layers of PIMs. This work is completed by the function `FindBasicHoms` (3.3.1).

3.3.1 FindBasicHoms

◇ `FindBasicHoms(groupname, gnamelist, repr, wordrepr, grpinfo)` (function)

Returns: a record describing the basic algebra as in Section 1.3. The components of this record are:

group the string *groupname*.

generators the list of generators of the basic algebra. This includes certain maps between the projective indecomposable modules and idempotents.

npims the number of projective indecomposable modules in the block of the group algebra used in the function `InfoKond` (2.2.2) to create *grpinfo*. This is the same as the length of *gnamelist*.

pimnames is the list *gnamelist*.

cartan the cartan matrix of the block of the group algebra used in the function `InfoKond` (2.2.2) to create *grpinfo*.

field the splitting field of the group *groupname*.

dim a list of the dimensions of the projective indecomposable modules.

adjmat the adjacency matrix of the projective indecomposable modules. The $(i, j)^{th}$ entry is defined to be the number homomorphisms $P_i \rightarrow Rad(P_j)/Rad^2(P_j)$ where P_i and P_j are projective indecomposable modules in *gnamelist*.

There are also components for each of the generators. These components are themselves records describing the action of these elements on the Ext-quiver. The components of these records are: *start*, *ende*, *name*, and *mat*.

matrices record containing construction information for the PIMs in terms of the generators of the basic algebra. See `QuiverToBasic` (4.3.4) for a complete description.

FindBasicHoms constructs the record described above by calculating the homomorphisms $P_i \rightarrow \text{Rad}(P_j)/\text{Rad}^2(P_j)$ for all projective indecomposable modules in *gname*list of the group *groupname* using the representation *repr*. *wordrepr* is a representation that contains all of the simple modules corresponding to *gname*list. Continuing the example from `SpinUpPIMs` (3.2.2), we get

Example

```
gap> bas:=FindBasicHoms("M11",[ "1a", "2a", "4a" ],"x","mod",grpinfo);;
gap> RecNames(bas);
[ "group", "generators", "npims", "pimnames", "cartan", "field", "dim",
  "adjmat", "1a", "2a", "4a", "1a2a1", "1a4a1", "2a1a1", "2a2a1", "4a1a1",
  "4a4a1", "matrices" ]
```

The function `AutoCalcBasic` (2.2.1), described in Chapter 2, is an automation of the process described in this chapter.

Chapter 4

Other Functions in the Basic Package

This chapter covers functions contained in the Basic package which have not been described previously. These programs are called either by the function `AutoCalcBasic` (2.2.1) or functions described in the previous chapter. These functions are included here for development purposes.

4.1 Functions for Groups

4.1.1 FindOverGroupsTom

◇ `FindOverGroupsTom(tom, number)` (function)

Returns: a list of numbers corresponding to conjugacy classes of subgroups which contain elements of the conjugacy class given by *number* in the table of marks, see `TableOfMarks` (**Reference:** `TableOfMarks`), *tom*.

Example

```
gap> tom:=TableOfMarks("M11");
TableOfMarks( "M11" )
gap> FindOverGroupsTom(tom,13);
[ 13, 19, 20, 24, 25, 26, 31, 32, 33, 35, 36, 38, 39 ]
```

See also `TableOfMarks` (**Reference:** `TableOfMarks`).

4.1.2 FindMaximalOverGroupsTom

◇ `FindMaximalOverGroupsTom(tom, number)` (function)

Returns: a list of length 3, the first entry being numbers corresponding to conjugacy classes of maximal subgroups which contain a subgroup in the conjugacy class given by *number* in the table of marks, see `TableOfMarks` (**Reference:** `TableOfMarks`), *tom*. The second entry is the list of class lengths of these groups, and the third entry is the position of these groups in the list of maximal subgroups returned by `MaximalSubgroupsTom` (**Reference:** `MaximalSubgroupsTom`).

Example

```
gap> tom:=TableOfMarks("M11");
TableOfMarks( "M11" )
gap> FindMaximalOverGroupsTom(tom,13);
[ [ 38, 35 ], [ 11, 55 ], [ 1, 3 ] ]
```

See also `TableOfMarks` (**Reference:** `TableOfMarks`).

4.2 Permutation Calculations

4.2.1 FindPerms

◇ `FindPerms(groupname, number, fieldsize, helms, kelms)` (function)

Returns: 1 on success and 0 otherwise.

FindPerms constructs MeatAxe permutations in `BasicWorkingSpace` (2.1.1) for the action of the generators of the group described by *groupname* and the elements described by the straight line programs, see `StraightLineProgram` (**Reference: StraightLineProgram**), *helms* and *kelms* in terms of the generators of the subgroup described by the normalizer in the table of marks, see `NormalizerTom` (**Reference: NormalizerTom**), of the subgroup described by *number* in `TableOfMarks` (**Reference: TableOfMarks**) acting on the cosets of the subgroup described by *helms*. *groupname* must be an admissible name for the atlasrep package. To get a list of admissible names, use the program `DisplayAtlasInfo` (**AtlasRep: DisplayAtlasInfo**).

Example

```
gap> gpinfo:=InfoKond("M11",2,1);;
gap> subinfo:=FindCondSubgroup("M11",2,1);;
gap> FindPerms("M11",gpinfo.numberinburn,2,subinfo.gens[1],subinfo.gens[2]);
1
```

See also `InfoKond` (2.2.2) and `FindCondSubgroup` (3.1.2).

4.2.2 ImprPerm

◇ `ImprPerm(group, intgroup, subgroup, elements)` (function)

◇ `ImprPerm3(group, intgroup1, intgroup2, subgroup, elements)` (function)

Returns: *ImprPerm* returns a list of lists describing the action of *elements* on the cosets of *subgroup* in *group*.

ImprPerm3 returns a list of lists describing the action of *elements* on the cosets of *intgroup1*, *intgroup2*, and *subgroup* in *group*, respectively.

ImprPerm uses the intermediate subgroup *intgroup* to construct the action of *elements* on the cosets of *subgroup* in *group*. The intermediate subgroup *intgroup* must be a subgroup, see `IsSubgroup` (**Reference: IsSubgroup**), of *group* and *subgroup* must be a subgroup of *intgroup*. Permutations can be made from the list returned by *ImprPerm* with the function `PermList` (**Reference: PermList**).

ImprPerm3 is a variation of *ImprPerm* which uses two intermediate subgroups *intgroup1* and *intgroup2* to construct the action of *elements* on the cosets of *subgroup* in *group*. The intermediate subgroup *intgroup1* must be a subgroup, see `IsSubgroup` (**Reference: IsSubgroup**), of *group*, *intgroup2* must be a subgroup of *intgroup1*, and *subgroup* must be a subgroup of *intgroup2*.

Example

```
gap> gp:=SymmetricGroup(5);;
gap> intgp:=AlternatingGroup(5);;
gap> subgp:=SylowSubgroup(intgp,2);;
gap> lst:=ImprPerm(gp,intgp,subgp,GeneratorsOfGroup(gp));
[ [ 13, 14, 15, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 19, 20, 21, 29, 30,
    28, 18, 16, 17, 24, 22, 23, 26, 27, 25 ],
  [ 21, 19, 20, 17, 18, 16, 22, 23, 24, 29, 30, 28, 27, 25, 26, 6, 4, 5, 2,
    3, 1, 7, 8, 9, 14, 15, 13, 12, 10, 11 ] ]
gap> List(lst, x->PermList(x));
[ (1,13,10,7,4) (2,14,11,8,5) (3,15,12,9,6) (16,19,29,27,23) (17,20,30,25,24) (18,
  21,28,26,22), (1,21) (2,19) (3,20) (4,17) (5,18) (6,16) (7,22) (8,23) (9,24) (10,
  29) (11,30) (12,28) (13,27) (14,25) (15,26) ]
gap> intgp2:=Group([(1,2) (3,4), (1,3) (2,4), (1,2,3)]);
Group([ (1,2) (3,4), (1,3) (2,4), (1,2,3) ])
gap> lst:=ImprPerm3(gp,intgp,intgp2,subgp,GeneratorsOfGroup(gp));
[ [ [ 1, 2 ], [ 2, 1 ] ],
```

```

[ [ 5, 1, 2, 3, 4, 7, 10, 6, 8, 9 ], [ 7, 6, 8, 10, 9, 2, 1, 3, 5, 4 ] ],
[ [ 13, 14, 15, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 19, 20, 21, 29, 30,
    28, 18, 16, 17, 24, 22, 23, 26, 27, 25 ],
  [ 21, 19, 20, 17, 18, 16, 22, 23, 24, 29, 30, 28, 27, 25, 26, 6, 4, 5,
    2, 3, 1, 7, 8, 9, 14, 15, 13, 12, 10, 11 ] ] ]
gap> List(lst, y->List(y, x->PermList(x)));
[ [ (), (1,2) ], [ (1,5,4,3,2) (6,7,10,9,8), (1,7) (2,6) (3,8) (4,10) (5,9) ],
  [ (1,13,10,7,4) (2,14,11,8,5) (3,15,12,9,6) (16,19,29,27,23) (17,20,30,25,
    24) (18,21,28,26,22), (1,21) (2,19) (3,20) (4,17) (5,18) (6,16) (7,22) (8,
    23) (9,24) (10,29) (11,30) (12,28) (13,27) (14,25) (15,26) ] ] ]

```

See also `SymmetricGroup` (**Reference: `SymmetricGroup`**), `AlternatingGroup` (**Reference: `AlternatingGroup`**), `SylowSubgroup` (**Reference: `SylowSubgroup`**), `Group` (**Reference: `Group`**), and `List` (**Reference: `List`**).

4.2.3 MakePermOnCond

◇ `MakePermOnCond(groupname, number, cond)` (function)

◇ `MakePermOnCondNorm(groupname, number, cond, normcond)` (function)

Returns: a list whose entries are lists with the first entry being the number of generators of the group and the second a list of permutations where the first permutations are group generators, followed by generators of the condensation subgroup. It should be noted that for `MakePermOnCondNorm`, the normalizing elements come last in the list of generators for the condensation subgroup.

For `MakePermOnCond`, the permutations correspond to the action of the group on the condensation subgroup. `groupname` is the name of an admissible group, see `DisplayAtlasInfo` (**AtlasRep: `DisplayAtlasInfo`**), as a string. `number` is the number in `TableOfMarks` (**Reference: `TableOfMarks`**) of the normalizer of the condensation subgroup. `cond` is a list of straight line programs, see `StraightLineProgram` (**Reference: `StraightLineProgram`**), in the generators of the normalizer which describe generators of the condensation subgroup.

For `MakePermOnCondNorm`, the three sets of permutations correspond to the action of the group on a maximal subgroup of the condensation subgroup, the condensation subgroup union normalizing elements, and the condensation subgroup, respectively. `groupname` is the name of an admissible group, see `DisplayAtlasInfo` (**AtlasRep: `DisplayAtlasInfo`**), as a string. `number` is the number in `TableOfMarks` (**Reference: `TableOfMarks`**) of the normalizer of the condensation subgroup. `cond` and `normcond` are straight line programs, see `StraightLineProgram` (**Reference: `StraightLineProgram`**), in the generators of the normalizer which describe generators of the condensation subgroup and normalizing p-elements, respectively.

Example

```

gap> subinfo:=FindCondSubgroup("m11",2,1);;
gap> perms:=MakePermOnCond("m11",subinfo.Normalizer,subinfo.gens[1]);;
gap> Length(perms);
1
gap> Length(perms[1]);
2
gap> Length(perms[1][2]);
4
gap> LargestMovedPoint(perms[1][2]);
880
gap> perms:=MakePermOnCondNorm("m11",subinfo.Normalizer,subinfo.gens[1],
> subinfo.gens[2]);;
gap> Length(perms);
3
gap> LargestMovedPoint(perms[1][2]);
55
gap> LargestMovedPoint(perms[2][2]);
55

```



```
gap> LargestMovedPoint(perms[3][2]);
880
```

See also `FindCondSubgroup` (3.1.2), `Length` (**Reference:** `Length`), and `LargestMovedPoint` (**Reference:** `LargestMovedPoint`).

4.2.4 Kond

◇ `Kond(prime, permname, matname, permnum, gennum)` (function)

Returns: nothing in GAP

prime is the characteristic. *permname* is the base name of the MeatAxe permutation files. *permnum* is the number of permutations to convert. *matname* is the base name for the output matrices in MeatAxe format. *gennum* is the number of generators of the condensations subgroup. *Kond* constructs the condensed MeatAxe matrices corresponding to the permutations. All of the MeatAxe files should be in `BasicWorkingSpace` (2.1.1). *Kond* is equivalent to the MeatAxe shell script `kd.sh`.

4.2.5 MakeGensKondSubgroup

◇ `MakeGensKondSubgroup(condperms, elmtperms, field)` (function)

Returns: nothing in GAP. Constructs MeatAxe format matrices in the `BasicWorkingSpace` (2.1.1) directory. These matrices are the elements *elmtperms* in the representation described by *condperms*.

MakeGensKondSubgroup writes the necessary permutations to the `BasicWorkingSpace` (2.1.1) directory in MeatAxe format and uses the MeatAxe functions *zmo* and *zkd* to construct the condensed matrices for the elements in the list *elmtperms*. These matrices will be labelled *kdg.i*, where *i* is the position of the element in the list *elmtperms*.

Example

```
gap> subinfo:=FindCondSubgroup("m11",2,1);
gap> perms:=MakePermOnCond("m11",subinfo.Normalizer,subinfo.gens[1]);
gap> MakeGensKondSubgroup(perms[1],perms[1][2],2);
gap> Process(BasicWorkingSpace, Ls, NoneIn, StdOut, []);
g.  g.2  g.4  kd.1  kd.orb  kdg.2  kdg.4
g.1  g.3  kd.  kd.2  kdg.1   kdg.3  p002.zzz
0
```

See also `FindCondSubgroup` (3.1.2), `MakePermOnCond` (4.2.3), and `Process` (**Reference:** `Process`).

4.3 Algebra Calculations

4.3.1 GetConstList

◇ `GetConstList(modulename, namelist, repr)` (function)

Returns: a list of the composition factors of the modules with names consisting of *modulename* concatenated with each of the entries in the list *namelist* which are also composition factors of the representation *repr*. The names returned are the labels in *repr*.

GetConstList compares the composition factors of the module *modulename* corresponding to the simple modules of *repr* listed in *namelist* against the composition factors of *repr*. This checking needs to be done since the naming conventions in the MeatAxe are independent for separate modules.

Example

```
gap> gpinfo:=InfoKond("M11",2,1);
gap> subinfo:=FindCondSubgroup("M11",2,1);
gap> alggens:=[];
gap> alggens[1]:=rec();
gap> alggens[1].genlist:=[2,6];
gap> CondenseModuleTom("M11",2,subinfo.SimpleRep,gpinfo.numberinburn,
gap> alggens,subinfo);
```

```
gap> Process(BasicWorkingSpace, chop, NoneIn, NoneOut, ["-g", "6", "mod"]);;
gap> GetConstList("mod", [""], "mod");
[ "1a", "2a", "4a" ]
```

See also [InfoKond \(2.2.2\)](#), [FindCondSubgroup \(3.1.2\)](#), [Record \(Reference: Record Assignment\)](#), [CondenseModuleTom \(3.2.1\)](#), [RandomMat \(Reference: RandomMat\)](#), and [Process \(Reference: Process\)](#).

4.3.2 NewSpinUp

◇ [NewSpinUp\(name, repr, sparserepr, ngen, pimdims, words, wordrepr\)](#) (function)

Returns: nothing.

NewSpinUp uses the peakwords *words*, as returned by [ReadWords \(4.3.8\)](#), to construct the projective indecomposable module for the simple module *name* in the MeatAxe matrix representation *repr* with *ngen* generators. The MeatAxe sparse matrix representation *sparserepr* is used for constructing the words more efficiently. *pimdims* is a list of the dimensions of the projective indecomposable modules to be spun up. This list is returned by [InfoKond \(2.2.2\)](#) as the record component *dimfixproj* for modules of the condensation algebra. *wordrepr* is the representation used to find the words, which is needed to get the names correct.

4.3.3 BigSpin

◇ [BigSpin\(sparserepr, ngens, seed\)](#) (function)

Returns: the dimension of the spun up space.

BigSpin spins up *seed* using the MeatAxe sparse matrix representation *repr* with *ngens* generators. This function should be used when all the generators of *repr* will not fit into memory and the spinning needs to be done one generator at a time. This is one of the functions where we need to be careful with the value of [BasicMemAvailable \(2.1.2\)](#). If this value is set too low, *BigSpin* might be used when the faster MeatAxe routines can be used.

Example

```
gap> gpinfo:=InfoKond("M11",2,1);;
gap> subinfo:=FindCondSubgroup("M11",2,1);;
gap> alggens:=[];;
gap> alggens[1]:=rec();;
gap> alggens[1].genlist:=[2,6];;
gap> CondenseModuleTom("M11",2,subinfo.SimpleRep,gpinfo.numberinburn,
gap> alggens,subinfo);
gap> vec:=RandomMat(1, 16, GF(2));
< mutable compressed matrix 1x1584 over GF(2) >
gap> m:=MeatAxeString(vec, 2);;
gap> outfile:=Filename(BasicWorkingSpace, "tvec");;
gap> name:=OutputTextFile( outfile, false);;
gap> SetPrintFormattingStatus( name, false);
gap> PrintTo(name, m);
gap> Process(BasicWorkingSpace, zcv, NoneIn, NoneOut, ["tvec", "vec"]);;
gap> for i in [1..6] do
> options:=["7", Concatenation("mod.",String(i)),
> Concatenation("spmod.",String(i))];
> Process(BasicWorkingSpace, mat2sparse, NoneIn, NoneOut, options);
> od;
gap> BigSpin("spmod", 6, "vec");
15
```

See also [InfoKond \(2.2.2\)](#), [FindCondSubgroup \(3.1.2\)](#), [Record \(Reference: Record Assignment\)](#), [CondenseModuleTom \(3.2.1\)](#), [RandomMat \(Reference: RandomMat\)](#), [MeatAxeString \(AtlasRep: MeatAxeString\)](#), [Filename \(Reference: Filename\)](#), [OutputTextFile \(Reference: OutputTextFile\)](#),

SetPrintFormattingStatus (**Reference: SetPrintFormattingStatus**), PrintTo (**Reference: PrintTo**), Concatenation (**Reference: Concatenation**), and Process (**Reference: Process**).

4.3.4 QuiverToBasic

◇ `QuiverToBasic(quiver)` (function)

Returns: a record containing the basic algebra. *quiver* is the record described in FindBasicHoms (3.3.1) without the *matrices* component. The basic algebra record is *quiver* with the additional component *matrices*, which is also a record which for each of the *generators* has the components:

perm is the permutation of the actions as returned by MakeAction (4.3.7).

spinningtree is a list describing the spinning used to generate the basis of the projective indecomposable module. This record also contains the action of the *generators* as compressed matrices, see CompressMat (4.4.1), on the projective indecomposable modules.

QuiverToBasic uses the matrices for the homomorphisms returned by FindBasicHoms (3.3.1) to construct the basic algebra.

4.3.5 Spinning

◇ `Spinning(quiver, startpim)` (function)

Returns: a basis for *startpim* as a list of vectors. These vectors contain more information than a list of coefficients, see MultiplyVecMat (4.3.6).

Spinning is a spinning algorithm specifically designed for projective indecomposable modules in a *quiver* or basic algebra. *startpim* is the number corresponding to the position of the desired projective indecomposable module in the list *quiver.pimnames*.

4.3.6 MultiplyVecMat

◇ `MultiplyVecMat(vector, matrix)` (function)

Returns: the image of *vector* under *matrix* in a *quiver* or basic algebra.

MultiplyVecMat is necessary since matrices in a *quiver* are maps between projective indecomposable modules in the *quiver*. Because of this, every matrix carries its start and end. In order to multiply a *vector* by a *matrix*, it must first be checked that *vector* lives in the module which is the domain of *matrix*.

4.3.7 MakeAction

◇ `MakeAction(list, basicalgebra)` (function)

Returns: a record which describes the action of *basicalgebra* on *list*. The components of this record are:

matrices the matrices describing the action of the generators of *basicalgebra* on *list*.

spaces is a list of bases for vector spaces corresponding to each projective indecomposable module and some other information which will be used to construct the spinning tree. A spinning tree is a record which describes how to generate a basis element as images of the generators.

perm is a list which as a permutation describes how the spinning tree has been reordered from the order in which it was originally generated through spinning.

The function *QuiverToBasic* (4.3.4) uses the record returned by *MakeAction* to complete the record for a *quiver* to a record for a basic algebra.

4.3.8 ReadWords

◇ **ReadWords**(*repr*) (function)

Returns: the peakwords for the representation in BasicWorkingSpace (2.1.1) named *repr*.

ReadWords calls the MeatAxe function *pwkond* to calculate the peakwords of *repr*. This will fail if *repr* has not been chopped with the MeatAxe function *chop*. *ReadWords* reads in the output of *pwkond*. This data will then be processed and returned as a list of lists, where each sublist contains the name of a composition factor, its peakword, and the polynomial used. Each composition factor of *repr* will appear once, up to isomorphism, in this list.

Example

```
gap> gpinfo:=InfoKond("M11",2,1);;
gap> subinfo:=FindCondSubgroup("M11",2,1);;
gap> alggens:=[];;
gap> alggens[1]:=rec();;
gap> alggens[1].genlist:=[2,6];;
gap> CondenseModuleTom("M11",2,subinfo.SimpleRep,gpinfo.numberinburn,
gap> alggens,subinfo);
gap> Process(BasicWorkingSpace, chop, NoneIn, NoneOut, ["-g","6","mod"]);;
gap> ReadWords("mod");
[ [ "mod4a", "(a+b+cd+fe+bd+fc+bedf)", "pol=x\n" ],
  [ "mod2a", "(a+b+fe)", "pol=x\n" ],
  [ "mod1a", "(cd+fe+afb)", "pol=x+1\n" ] ]
```

See also InfoKond (2.2.2), FindCondSubgroup (3.1.2), CondenseModuleTom (3.2.1), Process (Reference: Process), and BasicWorkingSpace (2.1.1).

4.3.9 ParseWord

◇ **ParseWord**(*word*) (function)

Returns: a parsed out word.

ParseWord parses the words found by the MeatAxe function *pwkond* as returned by ReadWords (4.3.8).

Example

```
gap> ParseWord("(a+b+cd+fe+bd+fc+bedf)");
[ [ [ [ [ "b" ], "e" ], "d" ], "f" ], [ [ [ "f" ], "c" ], "b" ],
  [ [ "c" ], [ "b" ], "d" ], [ [ "f" ], "e" ], [ "a" ], [ "b" ] ] ]
```

See also ReadWords (4.3.8).

4.3.10 WordPol

◇ **WordPol**(*representation*, *sparserep*, *word*, *polynomial*) (function)

Returns: nothing.

WordPol constructs the word *word* in the matrix representation *representation* in the directory BasicWorkingSpace (2.1.1) and then applies the polynomial *polynomial* to it. The result is stored in the file *word* in the directory BasicWorkingSpace (2.1.1). *sparserep* is the name of the sparse format matrices for *representation*. Using *sparserep* can speed up the production of words for large sparse representations. If there is no sparse representation, then *sparserep* should be given as an empty string.

Example

```
gap> gpinfo:=InfoKond("M11",2,1);;
[ 112, 96, 144 ]
gap> subinfo:=FindCondSubgroup("M11",2,1);
[ 112, 96, 144 ]
gap> alggens:=[];;
gap> alggens[1]:=rec();;
gap> alggens[1].genlist:=[2,6];;
```

```

gap> CondenseModuleTom("M11",2,subinfo.SimpleRep,gpinfo.numberinburn,
gap> alggens,subinfo);
gap> Process(BasicWorkingSpace, chop, NoneIn, NoneOut, ["-g","6","mod"]);;
gap> word:=ParseWord("(a+b+cd+fe+bd+fc+bedf),");
[ [ [ [ [ "b" ], "e" ], "d" ], "f" ], [ [ [ "f" ], "c" ], "b" ],
  [ [ "c" ], [ "b" ], "d" ], [ [ "f" ], "e" ], [ "a" ], [ "b" ] ]
gap> WordPol("M11","",word,words[3][3]);
Word: b+a+fe+bd+cd+fc+bedf

```

See also `Process` ([Reference: `Process`](#)), `BasicWorkingSpace` ([2.1.1](#)), `ParseWord` ([4.3.9](#)), `FindCondSubgroup` ([3.1.2](#)), `InfoKond` ([2.2.2](#)), and `CondenseModuleTom` ([3.2.1](#)).

4.4 Miscellaneous Functions

4.4.1 CompressMat

◇ `CompressMat(matrix)` (function)

Returns: a compressed matrix as a list of entries of the form [row, column, entry].

`CompressMat` records the position of nonzero entries in *matrix* and stores them, with the entry, in a list. For highly sparse matrices, this saves space when storing matrices.

Example

```

gap> M:=[[1,0,0,0,0],[0,0,0,0,0],[2,5,0,0,1],[0,0,0,9,0],[0,0,0,0,4]];
[ [ 1, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0 ], [ 2, 5, 0, 0, 1 ], [ 0, 0, 0, 9, 0 ],
  [ 0, 0, 0, 0, 4 ] ]
gap> CompressMat(M);
[ [ 1, 1, 1 ], [ 3, 1, 2 ], [ 3, 2, 5 ], [ 3, 5, 1 ], [ 4, 4, 9 ],
  [ 5, 5, 4 ] ]

```

4.4.2 UncompressMat

◇ `UncompressMat(matrix, dimension, field)` (function)

Returns: a *dimension* by *dimension* matrix with entries in *field*.

`UncompressMat` undoes the action of `CompressMat` ([4.4.1](#)). *dimension* and *field* must be given to `UncompressMat` since this information is not stored by `CompressMat` ([4.4.1](#)).

Example

```

gap> M:=[[1,0,0,0,0],[0,0,0,0,0],[2,5,0,0,1],[0,0,0,9,0],[0,0,0,0,4]];
[ [ 1, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0 ], [ 2, 5, 0, 0, 1 ], [ 0, 0, 0, 9, 0 ],
  [ 0, 0, 0, 0, 4 ] ]
gap> Cmat:=CompressMat(M);
[ [ 1, 1, 1 ], [ 3, 1, 2 ], [ 3, 2, 5 ], [ 3, 5, 1 ], [ 4, 4, 9 ],
  [ 5, 5, 4 ] ]
gap> UncompressMat(Cmat,5,Rationals);
[ [ 1, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0 ], [ 2, 5, 0, 0, 1 ], [ 0, 0, 0, 9, 0 ],
  [ 0, 0, 0, 0, 4 ] ]

```

Appendix A

Variables

A.1 Miscellaneous Variables

These variables are defined to allow the function `Process` (**Reference:** `Process`) to be used throughout the basic package in a convenient manner and should not be changed.

A.1.1 `CFInfo`

◇ `CFInfo` (global variable)

CFInfo is a necessary variable when reading MeatAxe cinfo files.

A.1.2 `MeatAxe`

◇ `MeatAxe` (global variable)

MeatAxe is a necessary variable when reading in MeatAxe files.

A.1.3 `MesserHalter`

◇ `MesserHalter` (global variable)

MesserHalter is the directory where the MeatAxe binary files are kept. This variable should not be changed since some of the MeatAxe programs included with this package may not be included in other distributions.

Appendix B

A note on algebra generators

As discussed in Section 3.1, we use a subalgebra eFG_e of the group algebra FG during the construction of the basic algebra. While any generating set of the group G gives a generating set for FG , we do not have a similar result for eFG_e . The function `FindAlgGens` (3.1.4) finds a small generating set for eFG_e . The purpose of this section is to explain the notation used for labeling the generators of eFG_e .

A small generating set for eFG_e is found by picking "random" elements in eFG_e and checking the dimension of the space they generate. Since these calculations need to be reproducible, we use straight line programs, see `StraightLineProgram` (**Reference: StraightLineProgram**), to construct elements in eFG_e . For the purposes of the `Basic` package, we use the following standard words (this is the output of a straight line program which is not given here) to generate elements of eFG_e :

- $z_1(a, b) = a,$
- $z_2(a, b) = b,$
- $z_3(a, b) = z_1(a, b) * z_2(a, b),$
- $z_4(a, b) = z_3(a, b)^2 * z_2(a, b),$
- $z_5(a, b) = z_3(a, b) * z_4(a, b),$
- $z_6(a, b) = z_3(a, b) * z_5(a, b),$
- $z_7(a, b) = z_4(a, b) * z_5(a, b),$
- $z_8(a, b) = z_3(a, b) * z_6(a, b),$
- $z_9(a, b) = z_7(a, b) * z_3(a, b) * z_2(a, b),$
- $z_{10}(a, b) = z_4(a, b) * z_3(a, b) * z_2(a, b) * z_5(a, b),$
- $z_{11}(a, b) = z_4(a, b) * z_3(a, b) * z_2(a, b).$

Since these eleven words are a small subset of eFG_e , there is no reason to believe that they will contain a generating set. For notation, let $z(a, b)$ be the list $[z_1(a, b), z_2(a, b), z_3(a, b), z_4(a, b), z_5(a, b), z_6(a, b), z_7(a, b), z_8(a, b), z_9(a, b), z_{10}(a, b), z_{11}(a, b)]$. When necessary, more elements are created by applying these straight line programs to the elements we have already created. This is done methodically, as described in the following algorithm:

1. Let a, b be permutations.
2. Compute $gens := z(a, b)$.
3. Initialize $perm1 := 1$ and $perm2 := 2$.
4. While $gens$ is not a generating set:

- (a) Increment $perm2$.
 - (b) If $perm1 = perm2$ then increment $perm2$.
 - (c) If $perm2 > 11$ then
 - i. increment $perm1$,
 - ii. set $perm2 := 1$.
 - (d) Append $z(gens[perm1], gens[perm2])$ onto $gens$.
5. Return the positions of the generators in the list $gens$.

The reason for covering this labelling of generators here is to explain the input and output of some functions in the Basic package.

First, we consider `FindAlgGens` (3.1.4). This function returns a list of records. Each record in this list corresponds to a pass through the *while* loop in the above algorithm. These records all contain a field named *genlist*, which is a list of numbers from 1 to 11 corresponding to the elements of $z(gens[perm1], gens[perm2])$ which are part of the generating set.

To use a command like `CondenseModuleTom` (3.2.1), we need to construct a record describing a generating set of eFG_e . To do this in GAP, we use the following sequence of commands.

Example

```
gap> alggens:=[];;
gap> alggens[1]:=rec();;
gap> alggens[1].genlist:=[2,6];;
gap> alggens[2]:=rec();;
gap> alggens[2].genlist:=[7];;
gap> alggens;
[ rec( genlist := [ 2, 6 ] ), rec( genlist := [ 7 ] ) ]
```

The list *alggens* corresponds to the set consisting of $z_2(a, b)$, $z_6(a, b)$ and $z_7(z_1(a, b), z_3(a, b))$. One should be careful here, `CondenseModuleTom` (3.2.1) will not verify that the record it is given corresponds to a generating set for eFG_e .

References

- [Hof04] Thomas R. Hoffman. *Constructing Basic Algebras for the Principal Block of Sporadic Simple Groups*. Dissertation, University of Arizona, 2004. [6](#), [13](#), [16](#), [18](#), [20](#)
- [Lux97] Klaus Lux. *Algorithmic Methods in Modular Representation Theory*. Habilitation thesis, Rheinisch-Westfälischen Technischen Hochschule Aachen, 1997. [6](#), [13](#), [14](#), [16](#), [18](#)
- [LW01] Klaus Lux and Markus Wiegmann. Determination of socle series using the condensation method. *J. Symbolic Comput.*, 31(1-2):163–178, 2001. Computational algebra and number theory (Milwaukee, WI, 1996). [14](#), [16](#), [17](#)
- [Pfe97] Götz Pfeiffer. The subgroups of M_{24} , or how to compute the table of marks of a finite group. *Experiment. Math.*, 6(3):247–270, 1997. [13](#)
- [Tha81] J. G. Thackray. *Modular Representations of Some Finite Groups*. PhD thesis, University of Cambridge, 1981. [13](#)

Index

AutoCalcBasic, [11](#)
 for named groups, [11](#)

BasicMemAvailable, [10](#)
BasicWorkingSpace, [10](#)
BigSpin, [26](#)

CFInfo, [30](#)
CompressMat, [29](#)
Condense, [17](#)
CondenseModuleTom, [18](#)

FindAlgGens, [17](#)
 for named groups, [17](#)
FindBasicHoms, [20](#)
FindCondSubgroup, [15](#)
 for named groups, [15](#)
FindFaithBurn, [13](#)
 for named groups, [13](#)
FindMaximalOverGroupsTom, [22](#)
FindOverGroupsTom, [22](#)
FindPerms, [23](#)

GetConstList, [25](#)
GetPerms, [16](#)
 for named groups, [16](#)

ImprimPerm, [23](#)
ImprimPerm3, [23](#)
InfoKond, [11](#)
 for named groups, [11](#)

Kond, [25](#)

MakeAction, [27](#)
MakeGensKondSubgroup, [25](#)
MakePermOnCond, [24](#)
MakePermOnCondNorm, [24](#)
MeatAxe, [30](#)
MesserHalter, [30](#)
MultiplyVecMat, [27](#)

NewSpinUp, [26](#)

ParseWord, [28](#)

QuiverToBasic, [27](#)

ReadWords, [28](#)

Spinning, [27](#)
SpinUpPIMs, [20](#)

UncompressMat, [29](#)

WordPol, [28](#)