HAPprime

Datatypes reference manual

Version 0.6

9 June 2011

Paul Smith

Paul Smith
Email: paul.smith@st-andrews.ac.uk
Homepage: http://www.cs.st-andrews.ac.uk/~pas
Address: School of Computer Science,
University of St Andrews
St Andrews,
UK.

Copyright

Acknowledgements

HAPprime was developed with the support of a Marie Curie Transfer of Knowledge grant based at the Department of Mathematics, NUI Galway (MTKD-CT-2006-042685) and is maintained with the support of the HPC-GAP grant at the School of Computer Science, University of St Andrews.

Contents

1 Introduction **5**
1.1 Internal function reference 6

2 Resolutions **7**
2.1 The `HAPResolution` datatype in HAPprime 7
2.2 Implementation: Constructing resolutions 7
2.3 Resolution construction functions 8
2.4 Resolution data access functions 9
2.5 Example: Computing and working with resolutions 10
2.6 Miscellaneous resolution functions 11

3 $\mathbb{F}G$-modules **12**
3.1 The `FpGModuleGF` datatype 12
3.2 Implementation details: Block echelon form 13
3.3 Construction functions 16
3.4 Data access functions 19
3.5 Generator and vector space functions 22
3.6 Block echelon functions 25
3.7 Sum and intersection functions 28
3.8 Miscellaneous functions 31

4 $\mathbb{F}G$-module homomorphisms **33**
4.1 The `FpGModuleHomomorphismGF` datatype 33
4.2 Calculating the kernel of a $\mathbb{F}G$-module homorphism by splitting into two homomorphisms 33
4.3 Calculating the kernel of a $\mathbb{F}G$-module homorphism by column reduction and partitioning 34
4.4 Construction functions 35
4.5 Data access functions 36
4.6 Image and kernel functions 38

5 General Functions **41**
5.1 Matrices 41
5.2 Groups 43

6 Internal functions **44**
6.1 Matrices as G-generators of a $\mathbb{F}G$-module vector space 44
6.2 $\mathbb{F}G$-modules 49
6.3 Resolutions 51
6.4 Test functions 52

Index **54**

Chapter 1

Introduction

The HAPprime package is a GAP package which supplements the HAP package (http://hamilton.nuigalway.ie/Hap/www/), providing new and improved functions for doing homological algebra over small prime-power groups. A detailed overview of the HAPprime package, with examples and documentation of the high-level functions, is provided in the accompanying HAPprime user guide.

This document, the datatypes reference manual, supplements the HAPprime user guide. It describes the new GAP datatypes defined by the HAPprime package, and all of the associated functions for working with each of these datatypes. The datatypes are

`HAPResolution`
: (Chapter 2) this datatype, defined in the HAP package, represents a free $\mathbb{F}G$-resolution of a $\mathbb{F}G$-module. HAPprime extends the definition of this datatype to save memory, and provides additional functions to operate on resolutions.

`FpGModuleGF`
: (Chapter 3) a free $\mathbb{F}G$-module compactly represented in terms of generating elements, with operations that do as much manipulation as possible within this form, thus minimizing memory use.

`FpGModuleHomomorphismGF`
: (Chapter 4) a free linear homomorphism between two $\mathbb{F}G$-modules, each represented as a `FpGModuleGF`. this also uses the compact generator form to save memory in its operations.

In addition, Chapter 5 provides documentation for some general functions defined in HAPprime which extend some of the basic GAP functionality in areas such as matrices and polynomials.

Each chapter of this reference manual begins with an overview of the datatype, and then implementation details of any interesting functions. The function reference of related functions then follows, subdivided into sections of related functions. Examples demonstrating the use of each function are given at the end of each section.

Earlier versions of this datatypes reference manual also documented the datatypes `GradedAlgebraPresentation`, `HAPRingHomomorphism` and `HAPDerivation`. The definitions of these datatypes and their related functions are now part of HAP and will be documented as part of that package.

1.1 Internal function reference

This version of the datatypes reference manual has been specially built to also provide documentation for all of the internal functions of HAPprime. (This can be done using the optional argument to `MakeHAPprimeDoc` (**HAPprime: MakeHAPprimeDoc**).) The documentation for these functions is found in Chapter 6.

Chapter 2

Resolutions

A free $\mathbb{F}G$-resolution of an $\mathbb{F}G$-module M is a sequence of module homomorphisms

$$\ldots \to M_{n+1} \to M_n \to M_{n-1} \to \ldots \to M_1 \to M_0 \twoheadrightarrow M$$

Where each M_n is a free $\mathbb{F}G$-module and the image of $d_{n+1} : M_{n+1} \to M_n$ equals the kernel of $d_n : M_n \to M_{n-1}$ for all $n > 0$.

2.1 The `HAPResolution` datatype in HAPprime

Both HAP and HAPprime use the `HAPResolution` datatype to store resolutions, and you should refer to the HAP documentation for full details of this datatype. With resolutions computed by HAP, the boundary maps which define the module homomorphisms are stored as lists of $\mathbb{Z}G$-module words, each of which is an integer pair `[i,g]`. By contrast, when HAPprime computes resolutions it stores the boundary maps as lists of G-generating vectors (as used in `FpGModuleHomomorphismGF`, see Chapter 4). Over small finite fields (and in particular in GF(2)), these compressed vectors take far less memory, saving at least a factor of two for long resolutions. The different data storage method is entirely an internal change - as far as the used is concerned, both versions behave exactly the same.

2.2 Implementation: Constructing resolutions

Given the definition of a free $\mathbb{F}G$-resolution given above, a resolution of a module M can be calculated by construction. If there are k generators for M, we can set M_0 equal to the free $\mathbb{F}G$-module $(\mathbb{F}G)^k$, and the module homomorphism $d_0 : M_0 \to M$ to be the one that sends the ith standard generator of $(\mathbb{F}G)^k$ to the ith element of M. We can now recursively construct the other modules and module homomorphisms in a similar manner. Given a boundary homomorphism $d_n = M_n \to M_{n-1}$, the kernel of this can be calculated. Then given a set of generators (ideally a small set) for $ker(d_n)$, we can set $M_{n+1} = (\mathbb{F}G)^{|ker(d_n)|}$, and the new module homomorphism d_{n+1} to be the one mapping the standard generators of M_{n+1} onto the generators of $ker(d_n)$.

HAPprime implements the construction of resolutions using this method. The construction is divided into two stages. The creation of the first homomorphism in the resolution for M is performed by the function `LengthZeroResolutionPrimePowerGroup` (2.3.2), or for a resolution of the trivial $\mathbb{F}G$-module $\mathbb{F}$, the first two homomorphisms can be stated

without calculation using LengthOneResolutionPrimePowerGroup (2.3.1). Once this initial sequence is created, a longer resolution can be created by repeated application of one of ExtendResolutionPrimePowerGroupGF (**HAPprime: ExtendResolutionPrimePowerGroupGF**), ExtendResolutionPrimePowerGroupRadical (**HAPprime: ExtendResolutionPrimePowerGroupRadical**) or ExtendResolutionPrimePowerGroupGF2 (**HAPprime: ExtendResolutionPrimePowerGroupGF2**), each of which extends the resolution by one stage by constructing a new module and homomorphism mapping onto the minimal generators of the kernel of the last homomorphism of the input resolution. These extension functions differ in speed and the amount of memory that they use. The lowest-memory version, ExtendResolutionPrimePowerGroupGF (**HAPprime: ExtendResolutionPrimePowerGroupGF**), uses the block structure of module generating vectors (see Section 3.2.1) and calculates kernels of the boundary homomorphisms using KernelOfModuleHomomorphismSplit (4.6.3) and finds a minimal set of generators for this kernel using MinimalGeneratorsModuleGF (3.5.9). The much faster but memory-hungry ExtendResolutionPrimePowerGroupRadical (**HAPprime: ExtendResolutionPrimePowerGroupRadical**) uses KernelOfModuleHomomorphism (4.6.3) and MinimalGeneratorsModuleRadical (3.5.9) respectively. ExtendResolutionPrimePowerGroupGF2 (**HAPprime: ExtendResolutionPrimePowerGroupGF2**) uses KernelOfModuleHomomorphismGF (4.6.3) whic partitions the boundary homomorphism matrix using $\mathbb{F}G$-reduction. This gives a small memory saving over the Radical method, but can take as long as the GF scheme.

The construction of resolutions of length n is wrapped up in the functions ResolutionPrimePowerGroupGF, ResolutionPrimePowerGroupRadical and ResolutionPrimePowerGroupAutoMem, which (as well as the extension functions) are fully documented in Section (**HAPprime: ResolutionPrimePowerGroup**) of the HAPprime user manual.

2.3 Resolution construction functions

2.3.1 LengthOneResolutionPrimePowerGroup

▷ LengthOneResolutionPrimePowerGroup(*G*) (function)

Returns: HAPResolution

Returns a free $\mathbb{F}G$-resolution of length 1 for group *G* (which must be of a prime power), i.e. the resolution

$$\mathbb{F}G^{k_1} \to \mathbb{F}G \twoheadrightarrow \mathbb{F}$$

This function requires very little calculation: the first stage of the resolution can simply be stated given a set of minimal generators for the group.

2.3.2 LengthZeroResolutionPrimePowerGroup

▷ LengthZeroResolutionPrimePowerGroup(*M*) (function)

Returns: HAPResolution

Returns a minimal free $\mathbb{F}G$-resolution of length 0 for the FpGModuleGF module *M*, i.e. the resolution

$$\mathbb{F}G^{k_0} \twoheadrightarrow M$$

This function requires little calculation since the the first stage of the resolution can simply be stated if the module has minimal generators: each standard generator of the zeroth-degree module M_0 maps onto a generator of `M`. If `M` does not have minimal generators, they are calculated using `MinimalGeneratorsModuleRadical` (3.5.9).

2.4 Resolution data access functions

2.4.1 ResolutionLength

▷ `ResolutionLength(R)` (method)

Returns: Integer

Returns the length (i.e. the maximum index k) in the resolution `R`.

2.4.2 ResolutionGroup

▷ `ResolutionGroup(R)` (method)

Returns: Group

Returns the group of the resolution `R`.

2.4.3 ResolutionFpGModuleGF

▷ `ResolutionFpGModuleGF(R, k)` (method)

Returns: `FpGModuleGF`

Returns the module M_k in the resolution `R`, as a `FpGModuleGF` (see Chapter 3), assuming the canonical action.

2.4.4 ResolutionModuleRank

▷ `ResolutionModuleRank(R, k)` (method)

Returns: Integer

Returns the $\mathbb{F}G$ rank of the kth module M_k in the resolution.

2.4.5 ResolutionModuleRanks

▷ `ResolutionModuleRanks(R)` (method)

Returns: List of integers

Returns a list containg the $\mathbb{F}G$ rank of the each of the modules M_k in the resolution `R`.

2.4.6 BoundaryFpGModuleHomomorphismGF

▷ `BoundaryFpGModuleHomomorphismGF(R, k)` (method)

Returns: `FpGModuleHomomorphismGF`

Returns the kth boundary map in the resolution `R`, as a `FpGModuleHomomorphismGF`. This represents the linear homomorphism $d_k : M_k \to M_{k-1}$.

2.4.7 ResolutionsAreEqual

▷ `ResolutionsAreEqual(R, S)` (operation)

Returns: Boolean

Returns `true` if the resolutions appear to be equal, `false` otherwise. This compares the torsion coefficients of the homology from the two resolutions.

2.5 Example: Computing and working with resolutions

In this example we construct a minimal free $\mathbb{F}G$-resolution of length four for the group $G = D_8 \times Q_8$ of order 64, which will be the sequence

$$(\mathbb{F}G)^{22} \to (\mathbb{F}G)^{15} \to (\mathbb{F}G)^{9} \to (\mathbb{F}G) \twoheadrightarrow \mathbb{F}$$

We first build each stage explicitly, starting with `LengthOneResolutionPrimePowerGroup` (2.3.1) followed by repeated applications of `ExtendResolutionPrimePowerGroupRadical` (**HAPprime: ExtendResolutionPrimePowerGroupRadical**). We extract various properties of this resolution. Finally, we construct equivalent resolutions for G using `ResolutionPrimePowerGroupGF` (**HAPprime: ResolutionPrimePowerGroupGF for group**) and `ResolutionPrimePowerGroupGF2` (**HAPprime: ResolutionPrimePowerGroupGF2 for group**) and check that the three are equivalent.

Example

```
gap> G := DirectProduct(DihedralGroup(8), SmallGroup(8, 4));
<pc group of size 64 with 6 generators>
gap> R := LengthOneResolutionPrimePowerGroup(G);
Resolution of length 1 in characteristic 2 for <pc group of size 64 with
6 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> R := ExtendResolutionPrimePowerGroupRadical(R);;
gap> R := ExtendResolutionPrimePowerGroupRadical(R);;
gap> R := ExtendResolutionPrimePowerGroupRadical(R);
Resolution of length 4 in characteristic 2 for <pc group of size 64 with
6 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> #
gap> ResolutionLength(R);
4
gap> ResolutionGroup(R);
<pc group of size 64 with 6 generators>
gap> ResolutionModuleRanks(R);
[ 4, 9, 15, 22 ]
gap> ResolutionModuleRank(R, 3);
15
gap> M2 := ResolutionFpGModuleGF(R, 2);
Full canonical module FG^9 over the group ring of <pc group of size 64 with
6 generators> in characteristic 2

gap> d3 := BoundaryFpGModuleHomomorphismGF(R, 3);
```

```
<Module homomorphism>

gap> ImageOfModuleHomomorphism(d3);
Module over the group ring of <pc group of size 64 with
6 generators> in characteristic 2 with 15 generators in FG^9.

gap> #
gap> S := ResolutionPrimePowerGroupGF(G, 4);
Resolution of length 4 in characteristic 2 for <pc group of size 64 with
6 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> ResolutionsAreEqual(R, S);
true
gap> T := ResolutionPrimePowerGroupGF2(G, 4);
Resolution of length 4 in characteristic 2 for <pc group of size 64 with
6 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> ResolutionsAreEqual(R, T);
true
```

Further example of constructing resolutions and extracting data from them are given in Sections 3.4.11, 3.5.11, 3.6.3, 4.5.7 and 4.6.4 in this reference manual, and also the chapter of (**HAPprime: Examples**) in the HAPprime user guide.

2.6 Miscellaneous resolution functions

2.6.1 BestCentralSubgroupForResolutionFiniteExtension

▷ `BestCentralSubgroupForResolutionFiniteExtension(G[, n])` (operation)

Returns: Group

Returns the central subgroup of `G` that is likely to give the smallest module ranks when using the HAP function `ResolutionFiniteExtension` (**HAP: ResolutionFiniteExtension**). That function computes a non-minimal resolution for `G` from the twisted tensor product of resolutions for a normal subgroup $N \lhd G$ and the quotient group G/N. The ranks of the modules in the resolution for G are the products of the module ranks of the resolutions for these smaller groups. This function tests `n` terms of the minimal resolutions for all the central subgroups of G and the corresponding quotients to find the subgroup/quotient pair with the smallest module ranks. If `n` is not provided, then $n = 5$ is used.

Chapter 3

$\mathbb{F}G$-modules

Let $\mathbb{F}G$ be the group ring of the group G over the field $\mathbb{F}$. In this package we only consider the case where G is a finite p-groups and $\mathbb{F} = \mathbb{F}_p$ is the field of p elements. In addition, we only consider free $\mathbb{F}G$-modules.

3.1 The `FpGModuleGF` datatype

Modules and submodules of free $\mathbb{F}G$-modules are represented in HAPprime using the `FpGModuleGF` datatype, where the '`GF`' stands for 'Generator Form'. This defines a module using a group G and a set of G-generating vectors for the module's vector space, together with a group action which operates on those vectors. A free $\mathbb{F}G$-module $\mathbb{F}G$ can be considered as a vector space $\mathbb{F}^{|G|}$ whose basis is the elements of G. An element of $(\mathbb{F}G)^n$ is the direct sum of n copies of $\mathbb{F}G$ and, as an element of `FpGModuleGF`, is represented as a vector of length $n|G|$ with coefficients in $\mathbb{F}$. Representing our $\mathbb{F}G$-module elements as vectors is ideal for our purposes since GAP's representation and manipulation of vectors and matrices over small prime fields is very efficient in both memory and computation time.

The HAP package defines a `FpGModule` object, which is similar but stores a vector space basis rather than a G-generating set for the module's vector space. Storing a G-generating set saves memory, both in passive storage and in allowing more efficient versions of some computation algorithms.

There are a number of construction functions for `FpGModuleGF`s: see 3.3.1 for details. A $\mathbb{F}G$-module is defined by the following:

- `gens`, a list of G-generating vectors for the underlying vector space. These do not need to be minimal - they could even be a vector space basis. The `MinimalGeneratorsModule` functions (3.5.9) can be used to convert a module to one with a minimal set of generators.
- `group`, the group G for the module
- `action`, a function `action(g, u)` that represents the module's group action on vectors. It takes a group element $g \in G$ and a vector `u` of length `actionBlockSize` and returns another vector `v` of the same length that is the product $v = gu$. If `action` is not provided, the canonical group permutation action is used. If the vector `u` is an integer multiple of `actionBlockSize` in length, the function `action` acts block-wise on the vector.
- `actionBlockSize`, the length of vectors upon which `action` operates. This is usually the order of the group, $|G|$ (for example for the canonical action), but it is possible to specify this to

support other possible group actions that might act on larger vectors. `actionBlockSize` will always be equal to the ambient dimension of the module $\mathbb{F}G^1$.

The group, action and block size are internally wrapped up into a record `groupAndAction`, with entries `group`, `action` and `actionBlockSize`. This is used to simplify the passing of parameters to some functions.

Some additional information is sometimes needed to construct particular classes of `FpGModuleGF`:

- `ambientDimension`, the length of vectors in the generating set: for a module $(\mathbb{F}G)^n$, this is equal to $n\times$`actionBlockSize`. This is needed in the case when the list of generating vectors is empty.
- `form`, a string detailing whether the generators are known to be minimal or not, and if so in which format. It can be one of `"unknown"`, `"fullcanonical"`, `"minimal"`, `"echelon"` or `"semiechelon"`. Some algorithms require a particular form, and algorithms such as `EchelonModuleGenerators` (3.6.1) that manipulate a module's generators to create these forms set this entry.

3.2 Implementation details: Block echelon form

3.2.1 Generating vectors and their block structure

Consider the vector representation of an element in the $\mathbb{F}G$-module $(\mathbb{F}G)^2$, where G is a group of order four:

$$v \in (\mathbb{F}G)^2 = (g_1+g_3, g_1+g_2+g_4) = [1010|1101]$$

The first block of four entries in the vector correspond to the first $\mathbb{F}G$ summand and the second block to the second summand (and the group elements are numbered in the order provided by the GAP function `Elements` (**Reference: Elements**)).

Given a G-generating set for a $\mathbb{F}G$-module, the usual group action permutes the group elements, and thus the effect on the vector is to permute the equivalent vector elements. Each summand is independent, and so elements are permuted within the blocks (normally of size $|G|$) seen in the example above. A consequence of this is that if any block (i.e. summand) in a generator is entirely zero, then it remains zero under group (or $\mathbb{F}$) multiplication and so that generator contributes nothing to that part of the vector space. This fact enables some of the structure of the module's vector space to be inferred from the G-generators, without needing a full vector space basis . A desirable set of G-generators is one that has many zero blocks, and what we call the 'block echelon' form is one that has this property.

3.2.2 Matrix echelon reduction and head elements

The block echelon form of a $\mathbb{F}G$-module generating set is analagous to the echelon form of matrices, used as a first stage in many matrix algorithms, and we first briefly review matrix echelon form. In a (row) echelon-form matrix, the head element in each row (the first non-zero entry) is the identity, and is to the right of the head in the previous row. A consequence of this is that the values below each head are all zero. All zero rows are at the bottom of the matrix (or are removed). GAP also defines a semi-echelon form, in which it is guaranteed that all values below each head is zero, but not that each head is to the right of those above it.

Matrices can be converted into (semi-)echelon form by using Gaussian elimination to perform row reduction (for example the GAP function `SemiEchelonMat` (**Reference: SemiEchelonMat**)). A typical algorithm gradually builds up a list of matrix rows with unique heads, which will eventually be an echelon-form set of basis elements for the row space of the matrix. This set is initialised with the first row of the matrix, and the algorithm is then applied to each subsequent row in turn. The basis rows in the current set are used to reduce the next row of the matrix: if, after reduction, it is non-zero then it will have a unique head and is added to the list of basis rows; if it is zero then it may be removed. The final set of vectors will be a semi-echelon basis for the row space of the original matrix, which can then be permuted to give an echelon basis if required.

3.2.3 Echelon block structure and minimal generators

In the same way that the echelon form is useful for vector space generators, we can convert a set of $\mathbb{F}G$-module generators into echelon form. However, unlike multiplication by a field element, the group action on generating vectors also permutes the vector elements, so a strict echelon form is less useful. Instead, we define a 'block echelon' form, treating the blocks in the vector (see example above) as the $\mathbb{F}G$-elements to be converted into echelon form. In block-echelon form, the first non-zero block in each row is as far to the right as possible. Often, the first non-zero block in a row will be to the right of the first non-zero block in the row above, but when several generating vectors are needed in a block, this may not be the case. The following example creates a random submodule of $(\mathbb{F}G)^n$ by picking five generating vectors at random. This module is first displayed with the original generators, and then they are converted to block echelon form using the the HAPprime function `EchelonModuleGenerators` (3.6.1). The two generating sets both span the same vector space (i.e. the same $\mathbb{F}G$ module), but the latter representation is much more useful.

Example

```
gap> M := FpGModuleGF(RandomMat(5, 32, GF(2)), DihedralGroup(8));;
gap> Display(M);
Module over the group ring of Group( [ f1, f2, f3 ] )
 in characteristic 2 with 5 generators in FG^4.
[.1..1.1.|.1....1.|1111.11.|11.1111.]
[11.1..1.|1....11.|1...111.|1...11..]
[11..1.1.|1.1.1...|11...1..|.11.11..]
[11111111|111...1.|.11...1.|.1..1111]
[.1111111|1.1.111.|..1..1..|1.111...]
gap> echM := EchelonModuleGenerators(M);
rec( module := Module over the group ring of <pc group of size 8 with
    3 generators> in characteristic 2 with 4 generators in FG^
    4. Generators are in minimal echelon form., headblocks := [ 1, 2, 3, 4 ] )
gap> Display(echM.module);
Module over the group ring of Group( [ f1, f2, f3 ] )
 in characteristic 2 with 4 generators in FG^4.
[.1..1.1.|.1....1.|1111.11.|11.1111.]
[........|.1111..1|111.1...|.11.11.1]
[........|........|.1..1.1.|.1.1.111]
[........|........|........|..1111.1]
Generators are in minimal echelon form.gap>
gap> M = echM.module;
true
```

The main algorithm for converting a set of generators into echelon form is

SemiEchelonModuleGeneratorsDestructive (3.6.1). The generators for the $\mathbb{F}G$ module are represented as rows of a matrix, and (with the canonical action) the first $|G|$ columns of this matrix correspond to the first block, the next $|G|$ columns to the second block, and so on. The first block of the matrix is taken and the vector space V_b spanned by the rows of that block is found (which will be a a subspace of $\mathbb{F}^{|G|}$). Taking the rows in the first block, find (by gradually leaving out rows) a minimal subset that generates V_b. The rows of the full matrix that correspond to this minimal subset form the first rows of the block-echelon form generators. Taking those rows, and all G-multiples of them, now calculate a semi-echelon basis for the vector space that they generate (using SemiEchelonMatDestructive (**Reference: SemiEchelonMatDestructive**)). This is used to reduce all of the other generators. Since the rows we have chosen span the space of the first block, the first block in all the other rows will be reduced to zero. We can now move on to the second block.

We now look at the rows of the matrix that start (have their first non-zero entry) in the second block. In addition, some of the generators used for the first block might additionally give rise to vector space basis vectors with head elements in the second blocks. The rows need to be stored during the first stage and reused here. We find a minimal set of the matrix rows with second-block heads that, when taken with any second-block heads from the first stage, generate the entire space spanned by the second block. The vector-space basis from this new minimal set is then used to reduce the rest of the generating rows as before, reducing all of the other rows' second blocks to zero. The process is then repeated for each other block. Any generators that are completely zero are then removed. The algorithm is summarised in the following pseudocode:

```
Let X be the list of generators still to convert (initially all the generators)
Let Xe = [] be the list of generators already in block-echelon form
Define X{b} to represent the $b$th block from generators X
Define V(X) to represent the vector space generated by generators X
------------------------------------------------------------------------------
for b in [1..blocks]
  1. Find a minimal set of generators Xm from X such that
     V(Xm{b} + Xe{b}) = V(X{b} + Xe{b})
  2. Remove Xm from X and add them to Xe
  3. Find a semi-echelon basis for V(Xe) and use this to reduce the elements
     of block b in remaining vectors of X to zero
end for
```

The result of this algorithm is a new generating set for the module that is minimal in the sense that no vector can be removed from the set and leave it still spanning the same vector space. In the case of a $\mathbb{F}G$-module with $\mathbb{F}$=GF(2), this is a globally minimal set: there is no possible alternative set with fewer generators.

3.2.4 Intersection of two modules

Knowing the block structure of the modules enables the intersection of two modules to be calculated more efficiently. Consider two modules U and V with the block structure as given in this example:

Example

```
gap> DisplayBlocks(U);
[*..]
[**.]
```

```
[.*.]
gap> DisplayBlocks(V);
[.**]
[.**]
[..*]
```

To calculate the intersection of the two modules, it is normal to expand out the two modules to find the vector space $U_{\mathbb{F}}$ and $V_{\mathbb{F}}$ of the two modules and calculate the intersection as a standard vector-space intersection. However, in this case, since U has no elements in the last block, and V has no elements in the first block, the intersection must only have elements in the middle block. This means that the first generator of U and the last generator of V can not be in the intersection and can be ignored for the purposes of the intersection calculation. In general, rather than expanding the entirety of U and V into an $\mathbb{F}$-basis to calculate their intersection, one can expand U and V more intelligently into $\mathbb{F}$-bases $U'_{\mathbb{F}}$ and $V'_{\mathbb{F}}$ which are smaller than $U_{\mathbb{F}}$ and $V_{\mathbb{F}}$ but have the same intersection.

Having determined (by comparing the block structure of U and V) that only the middle block in our example contributes to the intersection, we only need to expand out the rows of U and V that have heads in that block. The first generator of U generates no elements in the middle block, and trivially be ignored. The second row of U may or may not contribute to the intersection: this will need expanding out and echelon reduced. The expanded rows that don't have heads in the central block can then be discarded, with the other rows forming part of the basis of $U'_{\mathbb{F}}$. Likewise, the third generator of U is expanded and echelon reduced to give the rest of the basis for $U'_{\mathbb{F}}$. To find $V'_{\mathbb{F}}$, the first two generators are expanded, semi-echelon reduced and the rows with heads in the middle block kept. The third generator can be ignored. Finally, the intersection of $U'_{\mathbb{F}}$ and $V'_{\mathbb{F}}$ can found using, for example, `SumIntersectionMatDestructive` (5.1.1).

This algorithm, implemented in the function `IntersectionModulesGF` (3.7.3), will (for modules whose generators have zero columns) use less memory than a full vector-space expansion, and in the case where U and V have no intersection, may need to perform no expansion at all. In the worst case, both U and V will need a full expansion, using no more memory than the naive implementation. Since any full expansion is done row-by-row, with echelon reduction each time, it will in general still require less memory (but will be slower).

3.3 Construction functions

3.3.1 FpGModuleGF construction functions

▷ `FpGModuleGF(`*gens, G[, action, actionBlockSize]*`)` (operation)
▷ `FpGModuleGF(`*gens, groupAndAction*`)` (operation)
▷ `FpGModuleGF(`*ambientDimension, G[, action, actionBlockSize]*`)` (operation)
▷ `FpGModuleGF(`*ambientDimension, groupAndAction*`)` (operation)
▷ `FpGModuleGF(`*G, n*`)` (operation)
▷ `FpGModuleGF(`*groupAndAction, n*`)` (operation)
▷ `FpGModuleGF(`*HAPmod*`)` (operation)
▷ `FpGModuleGFNC(`*gens, G, form, action, actionBlockSize*`)` (operation)
▷ `FpGModuleGFNC(`*ambientDimension, G, action, actionBlockSize*`)` (operation)
▷ `FpGModuleGFNC(`*gens, groupAndAction[, form]*`)` (operation)

Returns: FpGModuleGF

Creates and returns a FpGModuleGF module object. The most commonly-used constructor requires a list of generators *gens* and a group *G*. The group action and block size can be specified using the *action* and *actionBlockSize* parameters, or if these are omitted then the canonical action is assumed. These parameters can also be wrapped up in a groupAndAction record (see 3.1).

An empty FpGModuleGF can be constructed by specifying a group and an *ambientDimension* instead of a set of generators. A module spanning $(\mathbb{F}G)^n$ with canonical generators and action can be constructed by giving a group *G* and a rank *n*. A FpGModuleGF can also be constructed from a HAP FpGModule *HAPmod*.

The generators in a FpGModuleGF do not need to be a minimal set. If you wish to create a module with minimal generators, construct the module from a non-minimal set *gens* and then use one of the MinimalGeneratorsModule functions (3.5.9). When constructing a FpGModuleGF from a FpGModule, the HAP function GeneratorsOfFpGModule (**HAP: GeneratorsOfFpGModule**) is used to provide a set of generators, so in this case the generators will be minimal.

Most of the forms of this function perform a few (cheap) tests to make sure that the arguments are self-consistent. The NC versions of the constructors are provided for internal use, or when you know what you are doing and wish to skip the tests. See Section 3.3.5 below for an example of usage.

3.3.2 FpGModuleFromFpGModuleGF

▷ FpGModuleFromFpGModuleGF(*M*) (operation)

Returns: FpGModule

Returns a HAP FpGModule that represents the same module as the FpGModuleGF *M*. This uses ModuleVectorSpaceBasis (3.5.7) to find the vector space basis for *M* and constructs a FpGModule with this vector space and the same group and action as *M* See Section 3.3.5 below for an example of usage.

TODO: This currently constructs an FpGModule object explicitly. It should use a constructor once one is provided

3.3.3 MutableCopyModule

▷ MutableCopyModule(*M*) (operation)

Returns: FpGModuleGF

Returns a copy of the module *M* where the generating vectors are fully mutable. The group and action in the new module is identical to that in *M* - only the list of generators is copied and made mutable. (The assumption is that this function used in situations where you just want a new generating set.)

3.3.4 CanonicalAction

▷ CanonicalAction(*G*) (attribute)

▷ CanonicalActionOnRight(*G*) (attribute)

▷ CanonicalGroupAndAction(*G*) (attribute)

Returns: Function action(g,v) or a record with elements (group, action, actionOnRight, actionBlockSize)

Returns a function of the form action(g,v) that performs the canonical group action of an element g of the group *G* on a vector v (acting on the left by default, or on the right with the OnRight version). The GroupAndAction version of this function returns the actions in a record together with

the group and the action block size. Under the canonical action, a free module $\mathbb{F}G$ is represented as a vector of length $|G|$ over the field $\mathbb{F}$, and the action is a permutation of the vector elements.

Note that these functions are attributes of a group, so that the canonical action for a particular group object will always be an identical function (which is desirable for comparing and combining modules and submodules).

3.3.5 Example: Constructing a `FpGModuleGF`

The example below constructs four different $\mathbb{F}G$-modules, where G is the quaternion group of order eight, and the action is the canonical action in each case:

1. `M` is the module $(\mathbb{F}G)^3$
2. `S` is the submodule of $(\mathbb{F}G)^3$ with elements only in the first summand
3. `T` is a random submodule `M` generated by five elements
4. `U` is the trivial (zero) submodule of $(\mathbb{F}G)^4$

We check whether `S`, `T` and `U` are submodules of `M`, examine a random element from `M`, and convert `S` into a HAP `FpGModule`. For the other functions used in this example, see Section 3.8.

Example

```
gap> G := SmallGroup(8, 4);;
gap> M := FpGModuleGF(G, 3);
Full canonical module FG^3 over the group ring of <pc group of size 8 with
3 generators> in characteristic 2
gap> gen := ListWithIdenticalEntries(24, Zero(GF(2)));;
gap> gen[1] := One(GF(2));;
gap> S := FpGModuleGF([gen], G);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 1 generator in FG^
3. Generators are in minimal echelon form.
gap> T := RandomSubmodule(M, 5);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 5 generators in FG^3.
gap> U := FpGModuleGF(32, CanonicalGroupAndAction(G));
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 0 generators in FG^
4. Generators are in minimal echelon form.
gap>
gap> IsSubModule(M, S);
true
gap> IsSubModule(M, T);
true
gap> IsSubModule(M, U);
false
gap>
gap> e := RandomElement(M);
<a GF2 vector of length 24>
gap> Display([e]);
 . 1 1 . . 1 . . . . . 1 . . 1 1 . . 1 . 1 . . 1
gap> IsModuleElement(S, e);
false
```

```
gap> IsModuleElement(T, e);
true
gap>
gap> FpGModuleFromFpGModuleGF(S);
Module of dimension 8 over the group ring of <pc group of size 8 with
3 generators> in characteristic 2
```

3.4 Data access functions

3.4.1 ModuleGroup

▷ `ModuleGroup(M)` (operation)

Returns: Group

Returns the group of the module `M`. See Section 3.4.11 below for an example of usage.

3.4.2 ModuleGroupOrder

▷ `ModuleGroupOrder(M)` (operation)

Returns: Integer

Returns the order of the group of the module `M`. This function is identical to `Order(ModuleGroup(M))`, and is provided for convenience. See Section 3.4.11 below for an example of usage.

3.4.3 ModuleAction

▷ `ModuleAction(M)` (operation)

Returns: Function

Returns the group action function of the module `M`. This is a function `action(g, v)` that takes a group element `g` and a vector `v` and returns a vector `w` that is the result of $w = gv$. See Section 3.4.11 below for an example of usage.

3.4.4 ModuleActionBlockSize

▷ `ModuleActionBlockSize(M)` (operation)

Returns: Integer

Returns the block size of the group action of the module `M`. This is the length of vectors (or the factor of the length) upon which the group action function acts. See Section 3.4.11 below for an example of usage.

3.4.5 ModuleGroupAndAction

▷ `ModuleGroupAndAction(M)` (operation)

Returns: Record with elements `(group, action, actionOnRight, actionBlockSize)`

Returns details of the module's group and action in a record with the following elements:

- `group` The module's group

- `action` The module's group action, as a function of the form `action(g, v)` that takes a vector `v` and returns the vector $w = gv$
- `actionOnRight` The module's group action when acting on the right, as a function of the form `action(g, v)` that takes a vector `v` and returns the vector $w = vg$
- `actionBlockSize` The module's group action block size. This is the ambient dimension of vectors in the module $\mathbb{F}G$

This function is provided for convenience, and is used by a number of internal functions. The canonical groups and action can be constructed using the function `CanonicalGroupAndAction` (3.3.4). See Section 3.4.11 below for an example of usage.

3.4.6 ModuleCharacteristic

▷ `ModuleCharacteristic(M)` (operation)

Returns: Integer

Returns the characteristic of the field $\mathbb{F}$ of the $\mathbb{F}G$-module `M`. See Section 3.4.11 below for an example of usage.

3.4.7 ModuleField

▷ `ModuleField(M)` (operation)

Returns: Field

Returns the field $\mathbb{F}$ of the $\mathbb{F}G$-module `M`. See Section 3.4.11 below for an example of usage.

3.4.8 ModuleAmbientDimension

▷ `ModuleAmbientDimension(M)` (operation)

Returns: Integer

Returns the ambient dimension of the module `M`. The module `M` is represented as a submodule of $\mathbb{F}G^n$ using generating vectors for a vector space. This function returns the dimension of this underlying vector space. This is equal to the length of each generating vector, and also $n \times$`actionBlockSize`. See Section 3.4.11 below for an example of usage.

3.4.9 AmbientModuleDimension

▷ `AmbientModuleDimension(M)` (operation)

Returns: Integer

The module `M` is represented a submodule embedded within the free module $\mathbb{F}G^n$. This function returns n, the dimension of the ambient module. This is the same as the number of blocks. See Section 3.4.11 below for an example of usage.

3.4.10 DisplayBlocks (for FpGModuleGF)

▷ `DisplayBlocks(M)` (operation)

Returns: nothing

Displays the structure of the module generators *gens* in a compact human-readable form. Since the group action permutes generating vectors in blocks of length `actionBlockSize`, any block that

contains non-zero elements will still contain non-zero elements after the group action, but a block that is all zero will remain all zero. This operation displays the module generators in a per-block form, with a * where the block is non-zero and . where the block is all zero.

The standard GAP methods `View` (**Reference: View**), `Print` (**Reference: Print**) and `Display` (**Reference: Display**) are also available.) See Section 3.6.3 below for an example of usage.

3.4.11 Example: Accessing data about a `FpGModuleGF`

In the following example, we construct three terms of a (minimal) resolution of the dihedral group of order eight, which is a chain complex of $\mathbb{F}G$-modules.

$$(\mathbb{F}G)^3 \to (\mathbb{F}G)^3 \to \mathbb{F}G \to \mathbb{F} \to 0$$

We obtain the last homomorphism in this chain complex and calculate its kernel, returning this as a `FpGModuleGF`. We can use the data access functions described above to extract information about this module.

See Chapters 4 and 2 respectively for more information about $\mathbb{F}G$-module homomorphisms and resolutions in HAPprime

Example

```
gap> R := ResolutionPrimePowerGroupRadical(DihedralGroup(8), 2);
Resolution of length 2 in characteristic 2 for <pc group of size 8 with
3 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> phi := BoundaryFpGModuleHomomorphismGF(R, 2);
<Module homomorphism>

gap> M := KernelOfModuleHomomorphism(phi);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 15 generators in FG^3.

gap> # Now find out things about this module M
gap> ModuleGroup(M);
<pc group of size 8 with 3 generators>
gap> ModuleGroupOrder(M);
8
gap> ModuleAction(M);
function( g, v ) ... end
gap> ModuleActionBlockSize(M);
8
gap> ModuleGroupAndAction(M);
rec( group := <pc group of size 8 with 3 generators>,
  action := function( g, v ) ... end,
  actionOnRight := function( g, v ) ... end, actionBlockSize := 8 )
gap> ModuleCharacteristic(M);
2
gap> ModuleField(M);
GF(2)
gap> ModuleAmbientDimension(M);
24
```

```
gap> AmbientModuleDimension(M);
3
```

3.5 Generator and vector space functions

3.5.1 ModuleGenerators

▷ `ModuleGenerators(M)` (operation)

Returns: List of vectors

Returns, as the rows of a matrix, a list of the set of currently-stored generating vectors for the vector space of the module M. Note that this set is not necessarily minimal. The function `ModuleGeneratorsAreMinimal` (3.5.2) will return `true` if the set is known to be minimal, and the `MinimalGeneratorsModule` functions (3.5.9) can be used to ensure a minimal set, if necessary. See Section 3.5.11 below for an example of usage.

3.5.2 ModuleGeneratorsAreMinimal

▷ `ModuleGeneratorsAreMinimal(M)` (operation)

Returns: Boolean

Returns `true` if the module generators are known to be minimal, or `false` otherwise. Generators are known to be minimal if the one of the `MinimalGeneratorsModule` functions (3.5.9) have been previously used on this module, or if the module was created from a HAP `FpGModule`. See Section 3.5.11 below for an example of usage.

3.5.3 ModuleGeneratorsAreEchelonForm

▷ `ModuleGeneratorsAreEchelonForm(M)` (operation)

Returns: Boolean

Return `true` if the module generators are known to be in echelon form, or (i.e. `EchelonModuleGenerators` (3.6.1) has been called for this module), or `false` otherwise. Some algorithms work more efficiently if (or require that) the generators of the module are in block-echelon form, i.e. each generator in the module's list of generators has its first non-zero block in the same location or later than the generator before it in the list. See Section 3.5.11 below for an example of usage.

3.5.4 ModuleIsFullCanonical

▷ `ModuleIsFullCanonical(M)` (operation)

Returns: Boolean

Returns `true` if the module is known to represent the full module $\mathbb{F}G^n$, with canonical generators and group action, or `false` otherwise. A module is only known to be canonical if it was constructed using the canonical module `FpGModuleGF` constructor (`FpGModuleGF` (3.3.1)). If this is true, the module is displayed in a concise form, and some functions have a trivial implementation. See Section 3.5.11 below for an example of usage.

3.5.5 ModuleGeneratorsForm

▷ `ModuleGeneratorsForm(M)` (operation)

Returns: String

Returns a string giving the form of the module generators. This may be one of the following:

- `"unknown"` The form is not known
- `"minimal"` The generators are known to be minimal, but not in any particular form
- `"fullcanonical"` The generators are the canonical (and minimal) generators for $\mathbb{F}G^n$
- `"semiechelon"` The generators are minimal and in semi-echelon form.
- `"echelon"` The generators are minimal and in echelon form

See Section 3.5.11 below for an example of usage.

3.5.6 ModuleRank

▷ `ModuleRank(M)` (operation)

▷ `ModuleRankDestructive(M)` (operation)

Returns: Integer

Returns the rank of the module `M`, i.e. the number of minimal generators. If the module is already in minimal form (according to `ModuleGeneratorsAreMinimal` (3.5.2)) then the number of generators is returned with no calculation. Otherwise, `MinimalGeneratorsModuleGF` (3.5.9) or `MinimalGeneratorsModuleGFDestructive` (3.5.9) respectively are used to find a set of minimal generators. See Section 3.5.11 below for an example of usage.

3.5.7 ModuleVectorSpaceBasis

▷ `ModuleVectorSpaceBasis(M)` (operation)

Returns: List of vectors

Returns a matrix whose rows are a basis for the vector space of the `FpGModuleGF` module `M`. Since `FpGModuleGF` stores modules as a minimal G-generating set, this function has to calculate all G-multiples of this generating set and row-reduce this to find a basis. See Section 3.5.11 below for an example of usage.

TODO: A GF version of this one

3.5.8 ModuleVectorSpaceDimension

▷ `ModuleVectorSpaceDimension(M)` (operation)

Returns: Integer

Returns the dimension of the vector space of the module `M`. Since `FpGModuleGF` stores modules as a minimal G-generating set, this function has to calculate all G-multiples of this generating set and row-reduce this to find the size of the basis. See Section 3.5.11 below for an example of usage.

TODO: A GF version of this function

3.5.9 MinimalGeneratorsModule

▷ `MinimalGeneratorsModuleGF(M)` (operation)
▷ `MinimalGeneratorsModuleGFDestructive(M)` (operation)
▷ `MinimalGeneratorsModuleRadical(M)` (operation)

Returns: FpGModuleGF

Returns a module equal to the `FpGModuleGF` *M*, but which has a minimal set of generators. Two algorithms are provided:

- The two `GF` versions use `EchelonModuleGenerators` (3.6.1) and `EchelonModuleGeneratorsDestructive` (3.6.1) respectively. In characteristic two, these return a set of minimal generators, and use less memory than the `Radical` version, but take longer. If the characteristic is not two, these functions revert to `MinimalGeneratorsModuleRadical`.
- The `Radical` version uses the radical of the module in a manner similar to the function **HAP: GeneratorsOfFpGModule**. This is much faster, but requires a considerable amount of temporary storage space.

See Section 3.5.11 below for an example of usage.

3.5.10 RadicalOfModule

▷ `RadicalOfModule(M)` (operation)

Returns: FpGModuleGF

Returns radical of the `FpGModuleGF` module *M* as another `FpGModuleGF`. The radical is the module generated by the vectors $v - gv$ for all v in the set of generating vectors for *M* and all g in a set of generators for the module's group.

The generators for the returned module will not be in minimal form: the `MinimalGeneratorsModule` functions (3.5.9) can be used to convert the module to a minimal form if necessary. See Section 3.5.11 below for an example of usage.

3.5.11 Example: Generators and basis vectors of a `FpGModuleGF`

Starting with the same module as in the earlier example (Section 3.4.11), we now investigate the generators of the module `M`. The generating vectors (of which there are 15) returned by the function `KernelOfModuleHomomorphism` (4.6.3) are not a minimal set, but the function `MinimalGeneratorsModuleGF` (3.5.9) creates a new object `N` representing the same module, but now with only four generators. The vector space spanned by these generators has 15 basis vectors, so representing the module by a G-generating set instead is much more efficient. (The original generating set in `M` was in fact an $\mathbb{F}$-basis, so the dimension of the vector space should come as no surprise.)

We can also find the radical of the module, and this is used internally for the faster, but more memory-hungry, `MinimalGeneratorsModuleRadical` (3.5.9).

Example

```
gap> R := ResolutionPrimePowerGroupRadical(DihedralGroup(8), 2);;
gap> phi := BoundaryFpGModuleHomomorphismGF(R, 2);;
gap> M := KernelOfModuleHomomorphism(phi);;
gap> #
gap> ModuleGenerators(M);
[ <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
```

```
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24> ]
gap> ModuleGeneratorsAreMinimal(M);
false
gap> ModuleGeneratorsForm(M);
"unknown"
gap> #
gap> N := MinimalGeneratorsModuleGF(M);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 4 generators in FG^
3. Generators are in minimal echelon form.

gap> M = N;    # Check that the new module spans the same space
true
gap> ModuleGeneratorsAreEchelonForm(N);
true
gap> ModuleIsFullCanonical(N);
false
gap> M = N;
true
gap> ModuleVectorSpaceBasis(N);
[ <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24>, <a GF2 vector of length 24>,
  <a GF2 vector of length 24> ]
gap> ModuleVectorSpaceDimension(N);
15
gap> #
gap> N2 := MinimalGeneratorsModuleRadical(M);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 4 generators in FG^
3. Generators are minimal.

gap> #
gap> R := RadicalOfModule(M);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 120 generators in FG^3.

gap> N2 = N;
true
```

3.6 Block echelon functions

3.6.1 EchelonModuleGenerators

▷ EchelonModuleGenerators(*M*) (operation)
▷ EchelonModuleGeneratorsDestructive(*M*) (operation)
▷ SemiEchelonModuleGenerators(*M*) (operation)
▷ SemiEchelonModuleGeneratorsDestructive(*M*) (operation)
▷ EchelonModuleGeneratorsMinMem(*M*) (operation)
▷ EchelonModuleGeneratorsMinMemDestructive(*M*) (operation)
▷ SemiEchelonModuleGeneratorsMinMem(*M*) (operation)
▷ SemiEchelonModuleGeneratorsMinMemDestructive(*M*) (operation)

Returns: Record (module, headblocks)

Returns a record with two components:

- module A module whose generators span the same vector space as that of the input module *M*, but whose generators are in a block echelon (or semi-echelon) form
- headblocks A list giving, for each generating vector in *M*, the block in which the head for that generating row lies

In block-echelon form. each generator is row-reduced using *G*g-multiples of the other other generators to produce a new, equivalent generating set where the first non-zero block in each generator is as far to the right as possible. The resulting form, with many zero blocks, can allow more memory-efficient operations on the module. See Section 3.2 for details. In addition, the standard (non-MinMem) form guarantees that the set of generators are minimal in the sense that no generator can be removed from the set while leaving the spanned vector space the same. In the GF(2) case, this is the global minimum.

Several versions of this algorithm are provided. The SemiEchelon versions of these functions do not guarantee a particular generator ordering, while the Echelon versions sort the generators into order of increasing initial zero blocks. The non-Destructive versions of this function return a new module and do not modify the input module; the Destructive versions change the generators of the input module in-place, and return this module. All versions are memory-efficient, and do not need a full vector-space basis. The MinMem versions are guaranteed to expand at most two generators at any one time, while the standard version may, in the (unlikely) worst-case, need to expand half of the generating set. As a result of this difference in the algorithm, the MinMem version is likely to return a greater number of generators (which will not be minimal), but those generators typically have a greater number of zero blocks after the first non-zero block in the generator. The MinMem operations are currently only implemented for GF(2) modules. See Section 3.6.3 below for an example of usage.

3.6.2 ReverseEchelonModuleGenerators

▷ ReverseEchelonModuleGenerators(*M*) (operation)
▷ ReverseEchelonModuleGeneratorsDestructive(*M*) (operation)

Returns: FpGModuleGF

Returns an FpGModuleGF module whose vector space spans the same space as the input module *M*, but whose generating vectors are modified to try to get as many zero blocks as possible at the end of each vector. This function performs echelon reduction of generating rows in a manner similar to EchelonModuleGenerators (3.6.1), but working from the bottom upwards. It is guaranteed that this function will not change the head block (the location of the first non-zero block) in each generating row, and hence if the generators are already in an upper-triangular form (e.g. following a call to

EchelonModuleGenerators (3.6.1)) then it will not disturb that form and the resulting generators will be closer to a diagonal form.

The Destructive version of this function modifies the input module's generators in-place and then returns that module; the non-Destructive version works on a copy of the input module and so will not modify the original module.

This operation is currently only implemented for GF(2) modules. See Section 3.5.11 below for an example of usage.

3.6.3 Example: Converting a FpGModuleGF to block echelon form

We can construct a larger module than in the earlier examples (Sections 3.4.11 and 3.5.11) by taking the kernel of the third boundary homomorphism of a minimal resolution of a group of order 32, which as returned by the function KernelOfModuleHomomorphism (4.6.3) has a generating set with many redundant generators. We display the block structure of the generators of this module after applying various block echelon reduction functions.

Example

```
gap> R := ResolutionPrimePowerGroupRadical(SmallGroup(32, 10), 3);;
gap> phi := BoundaryFpGModuleHomomorphismGF(R, 3);;
gap> #
gap> M := KernelOfModuleHomomorphism(phi);
Module over the group ring of <pc group of size 32 with
5 generators> in characteristic 2 with 65 generators in FG^4.

gap> #
gap> N := SemiEchelonModuleGenerators(M);
rec( module := Module over the group ring of <pc group of size 32 with
    5 generators> in characteristic 2 with 5 generators in FG^
    4. Generators are in minimal semi-echelon form.
    , headblocks := [ 2, 3, 1, 1, 3 ] )
gap> DisplayBlocks(N.module);
Module over the group ring of Group( [ f1, f2, f3, f4, f5 ] )
 in characteristic 2 with 5 generators in FG^4.
[.*.*]
[..**]
[***.]
[****]
[..**]
Generators are in minimal semi-echelon form.
gap> N2 := SemiEchelonModuleGeneratorsMinMem(M);
rec( module := Module over the group ring of <pc group of size 32 with
    5 generators> in characteristic 2 with 9 generators in FG^4.
    , headblocks := [ 2, 1, 3, 1, 1, 4, 1, 3, 4 ] )
gap> DisplayBlocks(N2.module);
Module over the group ring of Group( [ f1, f2, f3, f4, f5 ] )
 in characteristic 2 with 9 generators in FG^4.
[.*..]
[**..]
[..**]
[****]
[****]
[...*]
```

```
[****]
[..**]
[...*]

gap> #
gap> EchelonModuleGeneratorsDestructive(M);;
gap> DisplayBlocks(M);
Module over the group ring of Group( [ f1, f2, f3, f4, f5 ] )
 in characteristic 2 with 5 generators in FG^4.
[***.]
[****]
[.*.*]
[..**]
[..**]
Generators are in minimal echelon form.
gap> ReverseEchelonModuleGeneratorsDestructive(M);
Module over the group ring of <pc group of size 32 with
5 generators> in characteristic 2 with 5 generators in FG^
4. Generators are in minimal echelon form.

gap> DisplayBlocks(M);
Module over the group ring of Group( [ f1, f2, f3, f4, f5 ] )
 in characteristic 2 with 5 generators in FG^4.
[***.]
[****]
[.*..]
[..*.]
[..**]
Generators are in minimal echelon form.
```

3.7 Sum and intersection functions

3.7.1 DirectSumOfModules

▷ `DirectSumOfModules(M, N)` (operation)
▷ `DirectSumOfModules(coll)` (operation)
▷ `DirectSumOfModules(M, n)` (operation)

Returns: FpGModule

Returns the `FpGModuleGF` module that is the direct sum of the specified modules (which must have a common group and action). The input can be either two modules `M` and `N`, a list of modules `coll`, or one module `M` and an exponent `n` specifying the number of copies of `M` to sum. See Section 3.7.5 below for an example of usage.

If the input modules all have minimal generators and/or echelon form, the construction of the direct sum guarantees that the output module will share the same form.

3.7.2 DirectDecompositionOfModule

▷ `DirectDecompositionOfModule(M)` (operation)
▷ `DirectDecompositionOfModuleDestructive(M)` (operation)

Returns: List of FpGModuleGFs

Returns a list of FpGModuleGFs whose direct sum is equal to the input FpGModuleGF module *M*. The list may consist of one element: the original module.

This function relies on the block structure of a set of generators that have been converted to both echelon and reverse-echelon form (see EchelonModuleGenerators (3.6.1) and ReverseEchelonModuleGenerators (3.6.2)), and calls these functions if the module is not already in echelon form. In this form, it can be possible to trivially identify direct summands. There is no guarantee either that this function will return a decomposition if one is available, or that the modules returned in a decomposition are themselves indecomposable. See Section 3.7.5 below for an example of usage.

The Destructive version of this function uses the Destructive versions of the echelon functions, and so modifies the input module and returns modules who share generating rows with the modified *M*. The non-Destructive version operates on a copy of *M*, and returns modules with unique rows.

3.7.3 IntersectionModules

▷ IntersectionModules(*M*, *N*) (operation)
▷ IntersectionModulesGF(*M*, *N*) (operation)
▷ IntersectionModulesGFDestructive(*M*, *N*) (operation)
▷ IntersectionModulesGF2(*M*, *N*) (operation)

Returns: FpGModuleGF

Returns the FpGModuleGF module that is the intersection of the modules *M* and *N*. This function calculates the intersection using vector space methods (i.e. SumIntersectionMatDestructive (**HAPprime Datatypes: SumIntersectionMatDestructive**)). The standard version works on the complete vector space bases of the input modules. The GF version considers the block structure of the generators of *M* and *N* and only expands the necessary rows and blocks. This can lead to a large saving and memory if *M* and *N* are in echelon form and have a small intersection. See Section 3.2.4 for details. See Section 3.7.5 below for an example of usage. The GF2 version computes the intersection by a *G*-version of the standard vector space algorithm, using EchelonModuleGenerators (3.6.1) to perform echelon reduction on an augmented set of generators. This is much slower than the GF version, but may use less memory.

The vector spaces in FpGModuleGFs are assumed to all be with respect to the same canonical basis, so it is assumed that modules are compatible if they have the same group and the same ambient dimension.

The Destructive version of the GF function corrupts or permutes the generating vectors of *M* and *N*, leaving it invalid; the non-destructive version operates on copies of them, leaving the original modules unmodified. The generating vectors in the module returned by this function are in fact also a *vector space* basis for the module, so will not be minimal. The returned module can be converted to a set of minimal generators using one of the MinimalGeneratorsModule functions (3.5.9).

This operation is currently only defined for GF(2).

3.7.4 SumModules

▷ SumModules(*M*, *N*) (operation)

Returns: FpGModuleGF

Returns the FpGModuleGF module that is the sum of the input modules *M* and *N*. This function simply concatenates the generating vectors of the two modules and returns the result. If a set of

minimal generators are needed then use one of the MinimalGeneratorsModule functions (3.5.9) on the result. See Section 3.7.5 below for an example of usage.

The vector spaces in FpGModuleGF are assumed to all be with respect to the same canonical basis, so it is assumed that modules are compatible if they have the same group and the same ambient dimension.

3.7.5 Example: Sum and intersection of FpGModuleGFs

We can construct the direct sum of $\mathbb{F}G$-modules, and (attempt to) calculate a direct decomposition of a module. For example, we can show that

$$(\mathbb{F}G)^2 \oplus \mathbb{F}G = \mathbb{F}G \oplus \mathbb{F}G \oplus \mathbb{F}G$$

Example

```
gap> G := CyclicGroup(64);;
gap> FG := FpGModuleGF(G, 1);
Full canonical module FG^1 over the group ring of <pc group of size 64 with
6 generators> in characteristic 2

gap> FG2 := FpGModuleGF(G, 2);
Full canonical module FG^2 over the group ring of <pc group of size 64 with
6 generators> in characteristic 2

gap> M := DirectSumOfModules(FG2, FG);
Full canonical module FG^3 over the group ring of <pc group of size 64 with
6 generators> in characteristic 2

gap> DirectDecompositionOfModule(M);
[ Module over the group ring of <pc group of size 64 with
    6 generators> in characteristic 2 with 1 generator in FG^
    1. Generators are in minimal echelon form.
    , Module over the group ring of <pc group of size 64 with
    6 generators> in characteristic 2 with 1 generator in FG^
    1. Generators are in minimal echelon form.
    , Module over the group ring of <pc group of size 64 with
    6 generators> in characteristic 2 with 1 generator in FG^
    1. Generators are in minimal echelon form.
     ]
```

We can also compute the sum and intersection of $\mathbb{F}G$-modules. In the example below we construct two submodules of $\mathbb{F}G$, where G is the dihedral group of order four: M is the submodule generated by g_1+g_2, and N is the submodule generated by $g_1+g_2+g_3+g_4$. We calculate their sum and intersection. Since N is in this case a submodule of M it is easy to check that the correct results are obtained.

Example

```
gap> G := DihedralGroup(4);;
gap> M := FpGModuleGF([[1,1,0,0]]*One(GF(2)), G);
Module over the group ring of <pc group of size 4 with
2 generators> in characteristic 2 with 1 generator in FG^
1. Generators are in minimal echelon form.

gap> N := FpGModuleGF([[1,1,1,1]]*One(GF(2)), G);
```

```
Module over the group ring of <pc group of size 4 with
2 generators> in characteristic 2 with 1 generator in FG^
1. Generators are in minimal echelon form.

gap> #
gap> S := SumModules(M,N);
Module over the group ring of <pc group of size 4 with
2 generators> in characteristic 2 with 2 generators in FG^1.

gap> I := IntersectionModules(M,N);
Module over the group ring of <pc group of size 4 with
2 generators> in characteristic 2 with 1 generator in FG^1.

gap> #
gap> S = M and I = N;
true
```

3.8 Miscellaneous functions

3.8.1 = (for FpGModuleGF)

▷ =(*M*, *N*) (operation)

Returns: Boolean

Returns `true` if the modules are equal, `false` otherwise. This checks that the groups and actions in each module are equal (i.e. identical), and that the vector space spanned by the two modules are the same. (All vector spaces in `FpGModuleGF`s of the same ambient dimension are assumed to be embedded in the same canonical basis.) See Section 3.5.11 above for an example of usage.

3.8.2 IsModuleElement

▷ IsModuleElement(*M*, *elm*) (operation)

▷ IsModuleElement(*M*, *coll*) (operation)

Returns: Boolean

Returns `true` if the vector *elm* can be interpreted as an element of the module *M*, or `false` otherwise. If a collection of elements is given as the second argument then a list of responses is returned, one for each element in the collection. See Section 3.3.5 above for an example of usage.

3.8.3 IsSubModule

▷ IsSubModule(*M*, *N*) (operation)

Returns: Boolean

Returns `true` if the module *N* is a submodule of *M*. This checks that the modules have the same group and action, and that the generators for module *N* are elements of the module *M*. (All vector spaces in `FpGModuleGF`s of the same ambient dimension are assumed to be embedded in the same canonical basis.) See Section 3.3.5 above for an example of usage.

3.8.4 RandomElement

▷ `RandomElement(M[, n])` (operation)

Returns: Vector

Returns a vector which is a random element from the module `M`. If a second argument, `n` is give, then a list of `n` random elements is returned. See Section 3.3.5 above for an example of usage.

3.8.5 Random Submodule

▷ `RandomSubmodule(M, ngens)` (operation)

Returns: `FpGModuleGF`

Returns a `FpGModuleGF` module that is a submodule of `M`, with `ngens` generators selected at random from elements of `M`. These generators are not guaranteed to be minimal, so the rank of the submodule will not necessarily be equal to `ngens`. If a module with minimal generators is required, the `MinimalGeneratorsModule` functions (3.5.9) can be used to convert the module to a minimal form See Section 3.3.5 above for an example of usage.

Chapter 4

$\mathbb{F}G$-module homomorphisms

4.1 The `FpGModuleHomomorphismGF` datatype

Linear homomorphisms between free $\mathbb{F}G$-modules (as `FpGModuleGF` objects - see Chapter 3) are represented in HAPprime using the `FpGModuleHomomorphismGF` datatype. This represents module homomorphisms in a similar manner to $\mathbb{F}G$-modules, using a set of generating vectors, in this case vectors that generate the images in the target module of the generators of the source module.

Three items need to be passed to the constructor function `FpGModuleHomomorphismGF` (4.4.1):

- `source` the source `FpGModuleGF` module for the homomorphism
- `target` the target `FpGModuleGF` module for the homomorphism
- `gens` a list of vectors that are the images (in `target`) under the homomorphisms of each of the generators stored in `source`

4.2 Calculating the kernel of a $\mathbb{F}G$-module homorphism by splitting into two homomorphisms

HAPprime represents a homomorphism between two $\mathbb{F}G$-modules as a list of generators for the image of the homomorphism. Each generator is given as an element in the target module, represented as a vector in the same manner as used in the `FpGModuleGF` datatype (see Chapter 3). Given a set of such generating vectors, an $\mathbb{F}$-generating set for the image of the homomorphism (as elements of the target module's vector space) is given by taking all G-multiples of the generators. Writing the vectors in this expanded set as a matrix, the kernel of the boundary homomorphism is the (left) null-space of this matrix. As with `FpGModuleGF`s, the block structure of the generating vectors (see Section 3.2.1) allows this null-space to be calculated without necessarily expanding the whole matrix.

This basic algorithm is implemented in the HAPprime function `KernelOfModuleHomomorphismSplit` (4.6.3). The generating vectors for a module homomorphism H are divided in half, with the homomorphism generated by the first half of the generating vectors being called U and that by the second half being called V. Given this partition the kernel of H can be defined as

$$ker(H) = preim_U(I) \cap [-preim_V(I)]$$

where

- $I = im(U) \cap im(V)$ is the intersection of the images of the two homomorphisms U and V
- $preim_U(I)$ the set of all preimages of I under U
- $preim_V(I)$ the set of all preimages of I under V

Rather than computing the complete set of preimages, instead the implementation takes a preimage representative of each generator for I and adds the kernel of the homomorphisms U and V. The means that instead of calculating the null-space of the full expanded matrix, we can compute the answer by calculating the kernels of two homomorphisms with fewer generators, as well as the intersection of two modules, and some preimage representatives. Each of these operations takes less memory than the naive null-space calculation. The intersection of two $\mathbb{F}G$-modules can be compactly calculated using the generators' block structure (see Section 3.2.4), while the kernels of U and V can be computed recursively using these same algorithm. The block structure can also help in calculating the preimage, but at a considerable cost in time, so this is not done. However, since U and V have fewer generators than the original homomorphism H, a space saving is still made.

In the case where the problem is seperable, i.e. a U and V can be found for which there is no intersection, this approach can give a large saving. The separable components of the homomorphism can be readily identified from the block structure of the generators (they are the rows which share no blocks or heads with other rows), and the kernels of these can be calculated independently, with no intersection to worry about. This is implemented in the alternative algorithm `KernelOfModuleHomomorphismIndependentSplit` (4.6.3).

4.3 Calculating the kernel of a $\mathbb{F}G$-module homorphism by column reduction and partitioning

The list of generators of the image of a $\mathbb{F}G$-module homomorphism can be interpreted as the rows of a matrix A with elements in $\mathbb{F}G$, and it is the kernel of this matrix which must be found (i.e. the solutions to $xA = 0$. If column reduction is performed on this matrix (by adding $\mathbb{F}G$-multiples of other columns to a column), the kernel is left unchanged, and this process can be performed to enable the kernel to be found by a recursive algorithm similar to standard back substitution methods.

Given the matrix $A = (a_{ij})$, take the $\mathbb{F}G$-module generated by the first row (a_{1j}) and find a minimal (or small) subset of elements $\{a_{1j}\}_{j\in J}$ that generate this module. Without altering the kernel, we can permute the columns of A such that $J = \{1 \dots t\}$. Taking $\mathbb{F}$ and G-multiples of these columns from the remaining columns, the first row of these columns can be reduced to zero, giving a new matrix A'. This matrix can be partitioned as follows:

$$\begin{pmatrix} B & 0 \\ C & D \end{pmatrix}$$

where B is $1 \times t$, C is $(m-1) \times t$ and D is $(m-1) \times (n-t)$. It is assumed that B and C are 'small' and operations on these can can be easily handled in memory using standard linear algebra, while D may still be large.

Taking the $\mathbb{F}G$-module generated by the t columns which form the BC partition of the matrix, we compute E, a set of minimal generators for the submodule of this which is zero in the first row. These are added as columns at the end of A', giving a matrix

$$\begin{pmatrix} B & 0 & 0 \\ C & D & E \end{pmatrix}$$

The kernel of this matrix can be shown to be

$$\begin{pmatrix} \ker B & 0 \\ L & \ker(DE) \end{pmatrix}$$

where

$$L = preim_B((\ker(DE))C)$$

The augmentation of D with E guarantees that this preimage always exists. Since B and C are small, both $\ker B$ and L are easy to compute using linear algebra, while $\ker(DE)$ can be computed by recursion.

Unfortunately, E can be large, and the accumulated increase of size of the matrix over many recursions negates the time and memory saving that this algorithm might be expected to give. Testing indicates that it is currently no faster than the `KernelOfModuleHomomorphismSplit` (4.6.3) method, and does not save much memory over the full expansion using linear algebra. An improved version of this algorithm would reduce E by D before augmentation, thus adding a smaller set of generators and restricting the explosion in size. If D were already in echelon form, this would also be time-efficient.

4.4 Construction functions

4.4.1 FpGModuleHomomorphismGF construction functions

▷ `FpGModuleHomomorphismGF(S, T, gens)` (operation)
▷ `FpGModuleHomomorphismGFNC(S, T, gens)` (operation)

Returns: `FpGModuleHomomorphismGF`

Creates and returns an `FpGModuleHomomorphismGF` module homomorphism object. This represents the homomorphism from the module `S` to the module `T` with a list of vectors `gens` whose rows are the images in `T` of the generators of `S`. The modules must (currently) be over the same group.

The standard constructor checks that the homomorphism is compatible with the modules, i.e. that the vectors in `gens` have the correct dimension and that they lie within the target module `T`. It also checks whether the generators of `S` are minimal. If they are not, then the homomorphism is created with a copy of `S` that has minimal generators (using `MinimalGeneratorsModuleRadical` (3.5.9)), and `gens` is also copied and converted to agree with the new form of `S`. If you wish to skip these checks then use the `NC` version of this function.

IMPORTANT: The generators of the module `S` and the generator matrix `gens` must be remain consistent for the lifetime of this homomorphism. If the homomorphism is constructed with a mutable source module or generator matrix, then you must be careful not to modify them while the homomorphism is needed.

4.4.2 Example: Constructing a `FpGModuleHomomorphismGF`

In this example we construct the module homomorphism $\phi : (\mathbb{F}G)^2 \to \mathbb{F}G$ which maps both generators of $(\mathbb{F}G)^2$ to the generator of $\mathbb{F}G$

Example

```
gap> G := SmallGroup(8, 4);;
gap> im := [1,0,0,0,0,0,0,0]*One(GF(2));
[ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ]
gap> phi := FpGModuleHomomorphismGF(
>              FpGModuleGF(G, 2),
>              FpGModuleGF(G, 1),
>              [im, im]);
<Module homomorphism>
```

4.5 Data access functions

4.5.1 SourceModule

▷ `SourceModule(phi)` (operation)

Returns: `FpGModuleGF`

Returns the source module for the homomorphism *phi*, as an `FpGModuleGF`.

4.5.2 TargetModule

▷ `TargetModule(phi)` (operation)

Returns: FpGModuleGF

Returns the targetmodule for the homomorphism *phi*, as an `FpGModuleGF`.

4.5.3 ModuleHomomorphismGeneratorMatrix

▷ `ModuleHomomorphismGeneratorMatrix(phi)` (operation)

Returns: List of vectors

Returns the generating vectors `gens` of the representation of the homomorphism *phi*. These vectors are the images in the target module of the generators of the source module.

4.5.4 DisplayBlocks (for FpGModuleHomomorphismGF)

▷ `DisplayBlocks(phi)` (method)

Returns: nothing

Prints a detailed description of the module in human-readable form, with the module generators and generator matrix shown in block form. The standard GAP methods `View` (**Reference: View**), `Print` (**Reference: Print**) and `Display` (**Reference: Display**) are also available.)

4.5.5 DisplayModuleHomomorphismGeneratorMatrix

▷ `DisplayModuleHomomorphismGeneratorMatrix(phi)` (method)

Returns: nothing

Prints a detailed description of the module homomorphism generating vectors `gens` in human-readable form. This is the display method used in the `Display` (**Reference: Display**) method for this datatype.

4.5.6 DisplayModuleHomomorphismGeneratorMatrixBlocks

▷ `DisplayModuleHomomorphismGeneratorMatrixBlocks(`*phi*`)` (method)

Returns: nothing

Prints a detailed description of the module homomorphism generating vectors `gens` in human-readable form. This is the function used in the `DisplayBlocks` (4.5.4) method.

4.5.7 Example: Accessing data about a `FpGModuleHomomorphismGF`

A free $\mathbb{F}G$ resolution is a chain complex of $\mathbb{F}G$-modules and homomorphisms, and the homomorphisms in a `HAPResolution` (see Chapter 2) can be extracted as a `FpGModuleHomomorphismGF` using the function `BoundaryFpGModuleHomomorphismGF` (2.4.6). We construct a resolution `R` and then examine the third resolution in the chain complex, which is a $\mathbb{F}G$-module homomorphism $d_3 : (\mathbb{F}G)^7 \to (\mathbb{F}G)^5$.

Example

```
gap> R := ResolutionPrimePowerGroupRadical(SmallGroup(64, 141), 3);;
#I  Dimension 2: rank 5
#I  Dimension 3: rank 7
gap> d3 := BoundaryFpGModuleHomomorphismGF(R, 3);;
gap> SourceModule(d3);
Full canonical module FG^7 over the group ring of <pc group of size 64 with
6 generators> in characteristic 2

gap> TargetModule(d3);
Full canonical module FG^5 over the group ring of <pc group of size 64 with
6 generators> in characteristic 2

gap> ModuleHomomorphismGeneratorMatrix(d3);
<an immutable 7x320 matrix over GF2>
gap> DisplayBlocks(d3);
Module homomorphism with source:
Full canonical module FG^7 over the group ring of Group(
[ f1, f2, f3, f4, f5, f6 ] )
 in characteristic 2

and target:
Full canonical module FG^5 over the group ring of Group(
[ f1, f2, f3, f4, f5, f6 ] )
 in characteristic 2

and generator matrix:
[*.*.*]
[*****]
[.**..]
[.**..]
[..**.]
[...**]
[...*.]
```

Note that the module homomorphism generating vectors in a resolution calculated using HAPprime are in block-echelon form (see Section 3.2). This makes it efficient to compute the kernel

of this homomorphism using KernelOfModuleHomomorphismSplit (4.6.3), as described in Section 4.2, since there is only a small intersection between the images generated by the top and bottom halves of the generating vectors.

4.6 Image and kernel functions

4.6.1 ImageOfModuleHomomorphism

▷ ImageOfModuleHomomorphism(*phi*) (operation)
▷ ImageOfModuleHomomorphism(*phi, M*) (operation)
▷ ImageOfModuleHomomorphism(*phi, elm*) (operation)
▷ ImageOfModuleHomomorphism(*phi, coll*) (operation)
▷ ImageOfModuleHomomorphismDestructive(*phi, elm*) (operation)
▷ ImageOfModuleHomomorphismDestructive(*phi, coll*) (operation)

Returns: FpGModuleGF, vector or list of vectors depending on argument

For a module homomorphism *phi*, the one-argument function returns the module that is the image of the homomorphism, while the two-argument versions return the result of mapping of an FpGModuleGF *M*, a module element *elm* (given as a vector), or a collection of module elements *coll* through the homomorphism. This uses standard linear algebra to find the image of elements from the source module.

The Destructive versions of the function will corrupt the second parameter, which must be mutable as a result. The version of this operation that returns a module does not guarantee that the module will be in minimal form, and one of the MinimalGeneratorsModule functions (3.5.9) should be used on the result if a minimal set of generators is needed.

4.6.2 PreImageRepresentativeOfModuleHomomorphism

▷ PreImageRepresentativeOfModuleHomomorphism(*phi, elm*) (operation)
▷ PreImageRepresentativeOfModuleHomomorphism(*phi, coll*) (operation)
▷ PreImageRepresentativeOfModuleHomomorphism(*phi, M*) (operation)
▷ PreImageRepresentativeOfModuleHomomorphismGF(*phi, elm*) (operation)
▷ PreImageRepresentativeOfModuleHomomorphismGF(*phi, coll*) (operation)

For an element *elm* in the image of *phi*, this returns a representative of the set of preimages of *elm* under *phi*, otherwise it returns fail. If a list of vectors *coll* is provided then the function returns a list of preimage representatives, one for each element in the list (the returned list can contain fail entries if there are vectors with no solution). For an FpGModuleGF module *M*, this returns a module whose image under *phi* is *M* (or fail). The module returned will not necessarily have minimal generators, and one of the MinimalGeneratorsModule functions (3.5.9) should be used on the result if a minimal set of generators is needed.

The standard functions use linear algebra, expanding the generator matrix into a full matrix and using SolutionMat (**Reference: SolutionMat**) to calculate a preimage of *elm*. In the case where a list of vectors is provided, the matrix decomposition is only performed once, which can save significant time.

The GF versions of the functions can give a large memory saving when the generators of the homomorphism *phi* are in echelon form, and operate by doing back-substitution using the generator form of the matrices.

4.6.3 KernelOfModuleHomomorphism

▷ `KernelOfModuleHomomorphism(phi)` (operation)
▷ `KernelOfModuleHomomorphismSplit(phi)` (operation)
▷ `KernelOfModuleHomomorphismIndependentSplit(phi)` (operation)
▷ `KernelOfModuleHomomorphismGF(phi)` (operation)

Returns: `FpGModuleGF`

Returns the kernel of the module homomorphism *phi*, as an `FpGModuleGF` module. There are three independent algorithms for calculating the kernel, represented by different versions of this function:

- The standard version calculates the kernel by the obvious vector-space method. The homomorphism's generators are expanded into a full vector-space basis and the kernel of that vector space homomorphism is found. The generators of the returned module are in fact a vector space basis for the kernel module.
- The `Split` version divides the homomorphism into two (using the first half and the second half of the generating vectors), and uses the preimage of the intersection of the images of the two halves to calculate the kernel (see Section 4.2). If the generating vectors for *phi* are in block echelon form (see Section 3.2), then this approach provides a considerable memory saving over the standard approach.
- The `IndependentSplit` version splits the generating vectors into sets that generate vector spaces which have no intersection, and calculates the kernel as the sum of the kernels of those independent rows. If the generating vectors can be decomposed in this manner (i.e. the the generator matrix is in a diagonal form), this will provide a very large memory saving over the standard approach.
- The `GF` version performs column reduction and partitioning of the generator matrix to enable a recursive approach to computing the kernel (see Section 4.3). The level of partitioning is governed by the option `MaxFGExpansionSize`, which defaults to 10^9, allowing about 128Mb of memory to be used for standard linear algebra before partitioning starts. See (**Reference: Options Stack**) for details of using options in GAP

None of these basis versions of the functions guarantee to return a minimal set of generators, and one of the `MinimalGeneratorsModule` functions (3.5.9) should be used on the result if a minimal set of generators is needed. All of the functions leave the input homomorphism *phi* unchanged.

4.6.4 Example: Kernel and Image of a `FpGModuleHomomorphismGF`

A free $\mathbb{F}G$-resolution of a module is an exact sequence of module homomorphisms. In this example we use the functions `ImageOfModuleHomomorphism` (4.6.1) and `KernelOfModuleHomomorphism` (4.6.3) to check that one of the sequences in a resolution is exact, i.e. that in the sequence

$$M_3 \to M_2 \to M_1$$

the image of the first homomorphism $d_3 : M_3 \to M_2$ is the kernel of the second homomorphism $d_2 : M_2 \to M_1$

We also demonstrate that we can find the image and preimage of module elements under our module homomorphisms. We take an element `e` of M_2, in this case by taking the first generating element of the kernel of d_2, and map it up to M_3 and back.

Finally, we compute the kernel using the other available methods, and check that the results are the same.

Example

```
gap> R := ResolutionPrimePowerGroupRadical(SmallGroup(8, 3), 3);;
gap> d2 := BoundaryFpGModuleHomomorphismGF(R, 2);;
gap> d3 := BoundaryFpGModuleHomomorphismGF(R, 3);;
gap> #
gap> I := ImageOfModuleHomomorphism(d3);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 4 generators in FG^3.

gap> K := KernelOfModuleHomomorphism(d2);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 15 generators in FG^3.

gap> I = K;
true
gap> #
gap> e := ModuleGenerators(K)[1];;
gap> PreImageRepresentativeOfModuleHomomorphism(d3, e);
<a GF2 vector of length 32>
gap> f := PreImageRepresentativeOfModuleHomomorphism(d3, e);
<a GF2 vector of length 32>
gap> ImageOfModuleHomomorphism(d3, f);
<a GF2 vector of length 24>
gap> last = e;
true
gap> #
gap> L := KernelOfModuleHomomorphismSplit(d2);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 5 generators in FG^3.

gap> K = L;
true
gap> M := KernelOfModuleHomomorphismGF(d2);
Module over the group ring of <pc group of size 8 with
3 generators> in characteristic 2 with 4 generators in FG^
3. Generators are minimal.

gap> K = M;
true
```

Chapter 5

General Functions

Some of the functions provided by HAPprime are not specifically aimed at homological algebra or extending the HAP package. The functions in this chapter, which are used internally by HAPprime extend some of the standard GAP functions and datatypes.

5.1 Matrices

For details of the standard GAP vector and matrix functions, see (**Tutorial: matrices**) and (**Reference: Matrices**) in the GAP tutorial and reference manuals. HAPprime provides improved versions of a couple of standard matrix operations, and two small helper functions.

5.1.1 SumIntersectionMatDestructive

▷ `SumIntersectionMatDestructive(`*`U, V`*`)` (operation)
▷ `SumIntersectionMatDestructiveSE(`*`Ubasis, Uheads, Vbasis, Vheads`*`)` (operation)

Returns a list of length 2 with, at the first position, the sum of the vector spaces generated by the rows of *U* and *V*, and, at the second position, the intersection of the spaces.

Like the GAP core function `SumIntersectionMat` (**Reference: SumIntersectionMat**), this performs Zassenhaus' algorithm to compute bases for the sum and the intersection. However, this version operates directly on the input matrices (thus corrupting them), and is rewritten to require only approximately 1.5 times the space of the original input matrices. By contrast, the original GAP version uses three times the memory of the original matrices to perform the calculation, and since it doesn't modify the input matrices will require a total of four times the space of the original matrices.

The function `SumIntersectionMatDestructiveSE` takes as arguments not a pair of generating matrices, but a pair of semi-echelon basis matrices and the corresponding head locations, such as is returned by a call to `SemiEchelonMatDestructive` (**Reference: SemiEchelonMatDestructive**) (these arguments must all be mutable, so `SemiEchelonMat` (**Reference: SemiEchelonMat**) cannot be used). This function is used internally by `SumIntersectionMatDestructive`, and is provided for the occasions when the user might already have the semi-echelon versions available, in which case a small amount of time will be saved.

5.1.2 SolutionMat (for multiple vectors)

▷ SolutionMat(*M*, *V*) (operation)
▷ SolutionMatDestructive(*M*, *V*) (operation)

Calculates, for each row vector v_i in the matrix *V*, a solution to $x_i \times M = v_i$, and returns these solutions in a matrix X, whose rows are the vectors x_i. If there is not a solution for a v_i, then fail is returned for that row.

These functions are identical to the kernel functions SolutionMat (**Reference: SolutionMat**) and SolutionMatDestructive (**Reference: SolutionMatDestructive**), but are provided for cases where multiple solutions using the same matrix *M* are required. In these cases, using this function is far faster, since the matrix is only decomposed once.

The Destructive version corrupts both the input matrices, while the non-Destructive version operates on copies of these.

5.1.3 IsSameSubspace

▷ IsSameSubspace(*U*, *V*) (operation)

Returns true if the subspaces spanned by the rows of *U* and *V* are the same, false otherwise.

This function treats the rows of the two matrices as vectors from the same vector space (with the same basis), and tests whether the subspace spanned by the two sets of vectors is the same.

5.1.4 PrintDimensionsMat

▷ PrintDimensionsMat(*M*) (operation)

Returns a string containing the dimensions of the matrix *M* in the form "mxn", where m is the number of rows and n the number of columns. If the matrix is empty, the returned string is "empty".

5.1.5 Example: matrices and vector spaces

GAP uses rows of a matrix to represent basis vectors for a vector space. In this example we have two matrics U and V that we suspect represent the same subspace. Using SolutionMat (5.1.2) we can see that V lies in U, but IsSameSubspace (5.1.3) shows that they are the same subspace, as is confirmed by having identical sums and intersections.

Example

```
gap> U := [[1,2,3],[4,5,6]];;
gap> V := [[3,3,3],[5,7,9]];;
gap> SolutionMat(U, V);
[ [ -1, 1 ], [ 1, 1 ] ]
gap> IsSameSubspace(U, V);
true
gap> SumIntersectionMatDestructive(U, V);
[ [ [ 1, 2, 3 ], [ 0, 1, 2 ] ], [ [ 0, 1, 2 ], [ 1, 0, -1 ] ] ]
gap> IsSameSubspace(last[1], last[2]);
true
gap> PrintDimensionsMat(V);
"2x3"
```

5.2 Groups

Small groups in GAP can be indexed by their small groups library number (**smallgrp: The Small Groups Library**). An alternative indexing scheme, the Hall-Senior number, is used by Jon Carlson to publish his cohomology ring calculations at http://www.math.uga.edu/~lvalero/cohointro.html. To allow comparison with these results, we provide a function that converts from the GAP small groups library numbers to Hall-Senior number for the groups of order 8, 16, 32 and 64.

5.2.1 HallSeniorNumber

▷ `HallSeniorNumber(`*`order, i`*`)` (attribute)

▷ `HallSeniorNumber(`*`G`*`)` (attribute)

Returns: Integer

Returns the Hall-Senior number for a small group (of order 8, 16, 32 or 64). The group can be specified an *order*, *i* pair from the GAP (**smallgrp: The Small Groups Library**) library, or as a group *G*, in which case `IdSmallGroup` (**smallgrp: IdSmallGroup**) is used to identify the group.

Example

```
gap> HallSeniorNumber(32, 5);
20
gap> HallSeniorNumber(SmallGroup(64, 1));
11
```

Chapter 6

Internal functions

6.1 Matrices as G-generators of a $\mathbb{F}G$-module vector space

Both `FpGModuleGF` (Chapter 3) and `FpGModuleHomomorphismGF` (Chapter 4) store a matrix whose rows are G-generators for a module vector space (the module and the homomorphism's image respectively). The internal functions listed here provide common operations for dealing with these matrices.

6.1.1 HAPPRIME_ValueOptionMaxFGExpansionSize

▷ `HAPPRIME_ValueOptionMaxFGExpansionSize(`*`field, group`*`)` (operation)

Returns: Integer

Returns the maximum matrix expansion size. This is read from the `MaxFGExpansionSize` option from the GAP options stack (**Reference: Options Stack**), computed using the `MaxFGExpansionMemoryLimit` option.

6.1.2 HAPPRIME_KernelOfGeneratingRowsDestructive

▷ `HAPPRIME_KernelOfGeneratingRowsDestructive(`*`gens, rowlengths, GA`*`)` (operation)

Returns: List

Returns a list of generating vectors for the kernel of the $\mathbb{F}G$-module homomorphism defined by the generating rows *`gens`* using the group and action *`GA`*.

This function computes the kernel recursively by partitioning the generating rows into

$$\begin{pmatrix} B & 0 \\ C & D \end{pmatrix}$$

doing column reduction if necessary to get the zero block at the top right. The matrices $B >$ and C are small enough to be expanded, while the kernel of D is calculated by recursion. The argument *`rowlengths`* lists the number of non-zero blocks in each row; the rest of each row is taken to be zero. This allows the partitioning to be more efficiently performed (i.e. column reduction is not always required).

The GAP options stack (**Reference: Options Stack**) variable `MaxFGExpansionSize` can be used to specify the maximum allowable expanded matrix size. This governs the size of the B and C matrices, and thus the number of recursions before the kernel of D is also computed by recursion. A high value for will allow larger expansions and so faster computation at the cost of more memory. The `MaxFGExpansionMemoryLimit` option can also be used, which sets the maximum amount of memory

that GAP is allowed to use (as a string containing an integer with the suffix `k`, `M` or `G` to indicate kilobyes, megabytes or gigabytes respectively). In this case, the function looks at the free memory available to GAP and computes an appropriate value for `MaxFGExpansionSize`.

6.1.3 HAPPRIME_GActMatrixColumns

▷ `HAPPRIME_GActMatrixColumns(g, Vt, GA)` (operation)
▷ `HAPPRIME_GActMatrixColumnsOnRight(g, Vt, GA)` (operation)

Returns: Matrix

Returns the matrix that results from the applying the group action $u = gv$ (or $u = vg$ in the case of the `OnRight` version of this function) to each *column* vector in the matrix `Vt`. By acting on *columns* of a matrix (i.e. the transpose of the normal GAP representation), the group action is just a permutation of the rows of the matrix, which is a fast operation. The group and action are passed in `GA` using the `ModuleGroupAndAction` (3.4.5) record.

If the input matrix `Vt` is in a compressed matrix representation, then the returned matrix will also be in compressed matrix representation.

6.1.4 HAPPRIME_ExpandGeneratingRow

▷ `HAPPRIME_ExpandGeneratingRow(gen, GA)` (operation)
▷ `HAPPRIME_ExpandGeneratingRows(gens, GA)` (operation)
▷ `HAPPRIME_ExpandGeneratingRowOnRight(gen, GA)` (operation)
▷ `HAPPRIME_ExpandGeneratingRowsOnRight(gens, GA)` (operation)

Returns: List

Returns a list of G-generators for the vector space that corresponds to the of G-generator `gen` (or generators `gens`). This space is formed by multiplying each generator by each element of `G` in turn, using the group and action specified in `GA` (see `ModuleGroupAndAction` (3.4.5)). The returned list is thus $|G|$ times larger than the input.

For a list of generators `gens` $[v_1, v_2, \ldots, v_n]$, `HAPPRIME_ExpandGeneratingRows` (6.1.4) returns the list $[g_1v_1, g_2v_1, \ldots, g_1v_2, g_2v_2, \ldots, g_{|G|}v_n]$ In other words, the form of the returned matrix is block-wise, with the expansions of each row given in turn. This function is more efficient than repeated use of `HAPPRIME_ExpandGeneratingRow` (6.1.4) since it uses the efficient `HAPPRIME_GActMatrixColumns` (6.1.3) to perform the group action on the whole set of generating rows at a time.

The function `HAPPRIME_ExpandGeneratingRowsOnRight` (6.1.4) is the same as above, but the group action operates on the right instead.

6.1.5 HAPPRIME_AddGeneratingRowToSemiEchelonBasisDestructive

▷ `HAPPRIME_AddGeneratingRowToSemiEchelonBasisDestructive(basis, gen, GA)` (operation)

Returns: Record with elements `vectors` and `basis`

This function augments a vector space basis with another generator. It returns a record consisting of two elements: `vectors`, a set of semi-echelon basis vectors for the vector space spanned by the sum of the input `basis` and all G-multiples of the generating vector `gen`; and `heads`, a list of the head elements, in the same format as returned by `SemiEchelonMat` (**Reference: SemiEchelonMat**).

The generator *gen* is expanded according to the group and action specified in the *GA* record (see ModuleGroupAndAction (3.4.5)).

If the input *basis* is not zero, it is also modified by this function, to be the new basis (i.e. the same as the vectors element of the returned record).

6.1.6 HAPPRIME_ReduceVectorDestructive

▷ HAPPRIME_ReduceVectorDestructive(*v, basis, heads*) (operation)

Returns: Boolean

Reduces the vector *v* (in-place) using the semi-echelon set of vectors *basis* with heads *heads* (as returned by SemiEchelonMat (**Reference: SemiEchelonMat**)). Returns true if the vector is completely reduced to zero, or false otherwise.

6.1.7 HAPPRIME_ReduceGeneratorsOfModuleByXX

▷ HAPPRIME_ReduceGeneratorsOfModuleBySemiEchelon(*gens, GA*) (operation)
▷ HAPPRIME_ReduceGeneratorsOfModuleBySemiEchelonDestructive(*gens, GA*) (operation)
▷ HAPPRIME_ReduceGeneratorsOfModuleByLeavingOneOut(*gens, GA*) (operation)
▷ HAPPRIME_ReduceGeneratorsOnRightByLeavingOneOut(*gens, GA*) (operation)

Returns: List of vectors

Returns a subset of the module generators *gens* over the group with action specified in the *GA* record (see ModuleGroupAndAction (3.4.5)) that will still generate the module.

The BySemiEchelon functions gradually expand out the module generators into an $\mathbb{F}$-basis, using that $\mathbb{F}$-basis to reduce the other generators, until the full vector space of the module is spanned. The generators needed to span the space are returned, and should be a small set, although not minimal. The Destructive version of this function will modify the input *gens* parameter. The non-destructive version makes a copy first, so leaves the input arguments unchanged, at the expense of more memory.

The ByLeavingOneOut function is tries repeatedly leaving out generators from the list *gens* to find a small subset that still generates the module. If the generators are from the field GF(2), this is guaranteed to be a minimal set of generators. The OnRight version computes a minimal subset which generates the module under group multiplication on the right.

6.1.8 HAPPRIME_DisplayGeneratingRows

▷ HAPPRIME_DisplayGeneratingRows(*gens, GA*) (operation)

Returns: nothing

Displays a set of *G*-generating rows a human-readable form. The elements of each generating vector are displayed, with each block marked by a separator (since the group action on a module vector will only permute elements within a block).

This function is used by Display for both FpGModuleGF and FpGModuleHomomorphismGF.

NOTE: This is currently only implemented for GF(2)

Example

```
gap> HAPPRIME_DisplayGeneratingRows(
>  ModuleGenerators(M), HAPPRIME_ModuleGroupAndAction(M));
[...1..11|........|.......1|........|........]
[........|........|........|.1....11|........]
[........|........|........|........|..1.1.1.]
[........|.1.1..1.|........|........|........]
```

```
[........|........|......11|........|........]
[........|........|1......1|........|........]
```

6.1.9 HAPPRIME_GeneratingRowsBlockStructure

▷ HAPPRIME_GeneratingRowsBlockStructure(*gens, GA*) (operation)

Returns: Matrix

Returns a matrix detailing the block structure of a set of module generating rows. The group action on a generator permutes the vector in blocks of length GA.actionBlockSize: any block that contains non-zero elements will still contain non-zero elements after the group action; any block that is all zero will remain all zero. This operation returns a matrix giving this block structure: it has a one where the block is non-zero and zero where the block is all zero.

Example

```
gap> b := HAPPRIME_GeneratingRowsBlockStructure(
>  ModuleGenerators(M), ModuleActionBlockSize(M));
[ [ 1, 0, 1, 1, 1 ], [ 1, 0, 1, 1, 1 ], [ 0, 1, 1, 1, 1 ], [ 0, 0, 1, 1, 1 ] ]
```

6.1.10 HAPPRIME_DisplayGeneratingRowsBlocks

▷ HAPPRIME_DisplayGeneratingRowsBlocks(*gens, actionBlockSize*) (operation)

Returns: nothing

Displays a set of G-generating rows a compact human-readable form. Each generating rows can be divided into blocks of length *actionBlockSize*. The generating rows are displayed in a per-block form: a * where the block is non-zero and . where the block is all zero.

This function is used by DisplayBlocks (3.4.10) (for FpGModuleGF) and DisplayBlocks (4.5.4) (for FpGModuleHomomorphismGF).

Example

```
gap> HAPPRIME_DisplayGeneratingRowsBlocks(
>  ModuleGenerators(M), HAPPRIME_ModuleGroupAndAction(M));
[*.*..]
[...*.]
[....*]
[.*...]
[..*..]
[..*..]
```

6.1.11 HAPPRIME_IndependentGeneratingRows

▷ HAPPRIME_IndependentGeneratingRows(*blocks*) (operation)

Returns: List of lists

Given a block structure as returned by HAPPRIME_GeneratingRowsBlockStructure (6.1.9), this decomposes a set of generating rows into sets of independent rows. These are returned as a list of row indices, where each set of rows share no blocks with any other set.

Example

```
gap> DisplayBlocks(M);
Module over the group ring of Group( [ f1, f2, f3 ] )
 in characteristic 2 with 6 generators in FG^5.
[**...]
```

```
[.*...]
[.**..]
[.**..]
[...*.]
[....*]
Generators are in minimal echelon form.
gap> gens := ModuleGenerators(M);;
gap> G := ModuleGroup(M);;
gap> blocks := HAPPRIME_GeneratingRowsBlockStructure(gens, G);
[ [ 1, 1, 0, 0, 0 ], [ 0, 1, 0, 0, 0 ], [ 0, 1, 1, 0, 0 ], [ 0, 1, 1, 0, 0 ],
  [ 0, 0, 0, 1, 0 ], [ 0, 0, 0, 0, 1 ] ]
gap> HAPPRIME_IndependentGeneratingRows(blocks);
[ [ 1, 2, 3, 4 ], [ 5 ], [ 6 ] ]
```

6.1.12 HAPPRIME_GactFGvector

▷ `HAPPRIME_GactFGvector(g, v, MT)` (operation)

Returns: Vector

Returns the vector that is the result of the action $u = gv$ of the group element g on a module vector v (according to the group multiplication table *MT*. This operation is the quickest current method for a single vector. To perform the same action on a set of vectors, it is faster write the vectors as columns of a matrix and use `HAPPRIME_GActMatrixColumns` (6.1.3) instead.

6.1.13 HAPPRIME_CoefficientsOfGeneratingRowsXX

▷ `HAPPRIME_CoefficientsOfGeneratingRows(gens, GA, v)` (operation)
▷ `HAPPRIME_CoefficientsOfGeneratingRows(gens, GA, coll)` (operation)
▷ `HAPPRIME_CoefficientsOfGeneratingRowsDestructive(gens, GA, v)` (operation)
▷ `HAPPRIME_CoefficientsOfGeneratingRowsDestructive(gens, GA, coll)` (operation)
▷ `HAPPRIME_CoefficientsOfGeneratingRowsGF(gens, GA, v)` (operation)
▷ `HAPPRIME_CoefficientsOfGeneratingRowsGF(gens, GA, coll)` (operation)
▷ `HAPPRIME_CoefficientsOfGeneratingRowsGFDestructive(gens, GA, v)` (operation)
▷ `HAPPRIME_CoefficientsOfGeneratingRowsGFDestructive(gens, GA, coll)` (operation)
▷ `HAPPRIME_CoefficientsOfGeneratingRowsGFDestructive2(gens, GA, v)` (operation)
▷ `HAPPRIME_CoefficientsOfGeneratingRowsGFDestructive2(gens, GA, coll)` (operation)

Returns: Vector, or list of vectors

For a single vector *v*, this function returns a vector x giving the G-coefficients from *gens* needed to generate *v*, i.e. the solution to the equation $x * A = v$, where A is the expansion of *gens*. If there is no solution, `fail` is returned. If a list of vectors, *coll*, then a vector is returned that lists the solution for each vector (any of which may be `fail`). The standard forms of this function use standard linear algebra to solve for the coefficients. The `Destructive` version will corrupt both *gens* and *v*. The `GF` versions use the block structure of the generating rows to expand only the blocks that are needed to find the solution before using linear algebra. If the generators are in echelon form, this can save memory, but is slower.

The `GFDestructive2` functions also assume an echelon form for the generators, but use back-substitution to find a set of coefficients. This can save a lot of memory but is again slower.

6.1.14 HAPPRIME_GenerateFromGeneratingRowsCoefficientsXX

▷ HAPPRIME_GenerateFromGeneratingRowsCoefficients(*gens, GA, c*) (operation)
▷ HAPPRIME_GenerateFromGeneratingRowsCoefficients(*gens, GA, coll*) (operation)
▷ HAPPRIME_GenerateFromGeneratingRowsCoefficientsGF(*gens, GA, c*) (operation)
▷ HAPPRIME_GenerateFromGeneratingRowsCoefficientsGF(*gens, GA, coll*) (operation)

Returns: Vector, or list of vectors

For a vector *c*, returns (as a vector), the module element generated by multiplying *c* by the expansion of the generators *gens*. For a list of coefficient vectors *coll*, this returns a list of generating vectors.

The standard versions of this function use standard linear algebra. The GF versions only performs the expansion of necessary generating rows, and only expands by one group element at a time, so will only need at most twice the amount of memory as that to store *gens*, which is a large saving over expanding the generators by every group element at the same time, as in a naive implementation. It may also be faster.

6.1.15 HAPPRIME_RemoveZeroBlocks

▷ HAPPRIME_RemoveZeroBlocks(*gens, GA*) (operation)

Returns: Vector

Removes from a set of generating vectors *gens* (with ModuleGroupAndAction (3.4.5) *GA*) any blocks that are zero in every generating vector. Removal is done in-place, i.e. the input argument *gens* will be modified to remove the zero blocks. Zero blocks are unaffected by any row or expansion operation, and can be removed to save time or memory in those operations. The function returns the original block structure as a vector, and this can be used in the function HAPPRIME_AddZeroBlocks (6.1.16) to reinstate the zero blocks later, if required. See the documentation for that function for more detail of the block structure vector.

6.1.16 HAPPRIME_AddZeroBlocks

▷ HAPPRIME_AddZeroBlocks(*gens, blockStructure, GA*) (operation)

Returns: List of vectors

Adds zero blocks to a set of generating vectors *gens* to make it have the block structure given in *blockStructure* (for a given ModuleGroupAndAction (3.4.5) *GA*). The generators *gens* are modified in place, and also returned.

The *blockStructure* parameter is a vector of which is the length of the required output vector and has zeros where zero blocks should be, and is non-zero elsewhere. Typically, an earlier call to HAPPRIME_RemoveZeroBlocks (6.1.15) will have been used to remove the zero blocks, and this function and such a *blockStructure* vector is returned by this function. HAPPRIME_AddZeroBlocks (6.1.16) can be used to reinstate these zero blocks.

6.2 $\mathbb{F}G$-modules

$\mathbb{F}G$-modules in HAPprime use the datatype FpGModuleGF (Chapter 3). Internally, this uses many of the functions listed in Section 6.1, and further internal functions are listed below.

6.2.1 HAPPRIME_DirectSumForm

▷ HAPPRIME_DirectSumForm(*current, new*) (operation)

Returns: String

Returns a string containing the form of the generator matrix if the direct sum is formed between a FpGModuleGF with the form *current* and a FpGModuleGF with the form *new*. The direct sum is formed by placing the two module generating matrices in diagonal form. Given the form of the two generating matrices, this allows the form of the direct sum to be stated. See ModuleGeneratorsForm (3.5.5) for information about form strings.

6.2.2 HAPPRIME_PrintModuleDescription

▷ HAPPRIME_PrintModuleDescription(*M, func*) (operation)

Returns: nothing

Used by PrintObj (**Reference: PrintObj**), ViewObj (**Reference: ViewObj**), Display (**Reference: Display**) and DisplayBlocks (3.4.10), this helper function prints a description of the module *M*. The parameter *func* can be one of the strings "print", "view", "display" or "displayblocks", corresponding to the print different functions that might be called.

6.2.3 HAPPRIME_ModuleGeneratorCoefficients

▷ HAPPRIME_ModuleGeneratorCoefficients(*M, elm*) (operation)
▷ HAPPRIME_ModuleGeneratorCoefficientsDestructive(*M, elm*) (operation)
▷ HAPPRIME_ModuleGeneratorCoefficients(*M, coll*) (operation)
▷ HAPPRIME_ModuleGeneratorCoefficientsDestructive(*M, coll*) (operation)

Returns: Vector

Returns the coefficients needed to make the module element *elm* as a linear and *G*-combination of the module generators of the FpGModuleGF *M*. The coefficients are returned in standard vector form, or if there is no solution then fail is returned. If a list of elements is given, then a list of coefficients (or fails) is returned. The Destructive form of this function might change the elements of of *M* or *elm*. The non-Destructive version makes copies to ensure that they are not changed.

See also HAPPRIME_ModuleElementFromGeneratorCoefficients (6.2.4).

6.2.4 HAPPRIME_ModuleElementFromGeneratorCoefficients

▷ HAPPRIME_ModuleElementFromGeneratorCoefficients(*M, c*) (operation)
▷ HAPPRIME_ModuleElementFromGeneratorCoefficients(*M, coll*) (operation)

Returns: Vector

Returns an element from the module *M*, constructed as a linear and *G*-sum of the module generators as specified in *c*. If a list of coefficient vectors is given, a list of corresponding module elements is returned.

See also HAPPRIME_ModuleGeneratorCoefficients (6.2.3)

6.2.5 HAPPRIME_MinimalGeneratorsVectorSpaceGeneratingRowsDestructive

▷ HAPPRIME_MinimalGeneratorsVectorSpaceGeneratingRowsDestructive(*vgens, GA*) (operation)
▷ HAPPRIME_MinimalGeneratorsVectorSpaceGeneratingRowsOnRightDestructive(*vgens,*

GA) (operation)

Returns: FpGModuleGF

Returns a module with minimal generators that is equal to the $\mathbb{F}G$-module with *vector space* basis *vgens* and ModuleGroupAndAction (3.4.5) as specified in *GA*. The solution is computed by the module radical method, which is fast at the expense of memory. This function will corrupt the matrix *gens*.

This is a helper function for MinimalGeneratorsModuleRadical (3.5.9) that is also used by ExtendResolutionPrimePowerGroupRadical (**HAPprime: ExtendResolutionPrimePowerGroupRadical**) (which knows that its module is already in vector-space form).

6.2.6 HAPPRIME_IsGroupAndAction

▷ HAPPRIME_IsGroupAndAction(*obj*) (operation)

Returns: Boolean

Returns true if *obj* appears to be a groupAndAction record (see ModuleGroupAndAction (3.4.5)), or false otherwise.

6.3 Resolutions

For details of the main resolution functions in HAPprime, see Chapter 2 of this datatypes reference manual, and (**HAPprime: Resolutions**) in the HAPprime user guide. This section describes the internal helper functions used by the higher-level functions.

6.3.1 HAPPRIME_WordToVector

▷ HAPPRIME_WordToVector(*w*, *dim*, *orderG*) (method)

Returns: HAP word (list of lists)

Returns the boundary map vector that corresponds to the HAP word vector *w* with module ambient dimension *dim* and group order *orderG* (assumed to be the actionBlockSize). A HAP word vector has the following format: [[block, elm], [block, elm], ...] where block is a block number and elm is a group element index (see example below).

See also HAPPRIME_VectorToWord (6.3.2)

Example

```
gap> G := CyclicGroup(4);;
gap> v := HAPPRIME_WordToVector([ [1,2],[2,3] ], 2, Order(G));
<a GF2 vector of length 8>
gap> HAPPRIME_DisplayGeneratingRows([v], CanonicalGroupAndAction(G));
[.1..|..1.]
gap> HAPPRIME_VectorToWord(v, Order(G));
[ [ 1, 2 ], [ 2, 3 ] ]
```

6.3.2 HAPPRIME_VectorToWord

▷ HAPPRIME_VectorToWord(*vec*, *orderG*) (function)

Returns: Vector

The HAP word format vector that corresponds to the boundary vector *vec* with actionBlockSize assumed to be *orderG*.

See HAPPRIME_WordToVector (6.3.1) for a few more details and an example.

6.3.3 HAPPRIME_BoundaryMatrices

▷ HAPPRIME_BoundaryMatrices(*R*) (attribute)

Returns: List of matrices

If *R* is a resolution which stores its boundaries as a list of matrices (e.g. one created by HAPprime, this list is returned. Otherwise, fail is returned. Note that the first matrix in this list corresponds to the zeroth degree: for resolutions of modules, this is the generators of the module; for resolutions of groups, this is the empty matrix. The second matrix corresponds to the first degree, and so on.

6.3.4 HAPPRIME_AddNextResolutionBoundaryMapMatNC

▷ HAPPRIME_AddNextResolutionBoundaryMapMatNC(*R, BndMat*) (operation)

Returns: HapResolution

Returns the resolution *R* extended by one term, where that term is given by the boundary map matrix *BndMat*. If *BndMat* is not already in compressed matrix form, it will be converted into this form, and if the boundaries in *R* are not already in matrix form, they are all converted into this form.

6.3.5 HAPPRIME_CreateResolutionWithBoundaryMapMatsNC

▷ HAPPRIME_CreateResolutionWithBoundaryMapMatsNC(*G, BndMats*) (operation)

Returns: HapResolution

Returns a HAP resolution object for group *G* where the module homomorphisms are given by the boundary matrices in the list *BndMats*. This list is indexed with the boundary matrix for degree *zero* as the first element. If the resolution is the resolution of a module, the module's minimal generators are this first boundary matrix, otherwise (for the resolution of a group), this should be set to be the empty matrix [].

6.4 Test functions

Internal helper functions for testing HAPprime.

6.4.1 HAPPRIME_SingularIsAvailable

▷ HAPPRIME_SingularIsAvailable() (function)

Returns: Boolean

The Singular package can be succesfully loaded whether the singular executable is present or not, so this function attempts to check for the presence of this executable by searching on the system path and checking for global variables set by the Singular.

Whether this function returns true or false will not affect the rest of this package: it only affects which tests are run by the happrime.txt and testall.g test routines.

6.4.2 HAPPRIME_Random2Group

▷ HAPPRIME_Random2Group(*[orderG]*) (operation)

▷ HAPPRIME_Random2GroupAndAction(*[orderG]*) (operation)

Returns: Group or groupAndAction record

Returns a random 2-group, or a `groupAndAction` record (see `ModuleGroupAndAction` (3.4.5)) with the canonical action. The order may be specified as an argument, or if not then a group is chosen randomly (from a uniform distribution) over all of the possible groups with order from 2 to 128.

Example

```
gap> HAPPRIME_Random2Group();
<pc group of size 8 with 3 generators>
gap> HAPPRIME_Random2Group();
<pc group of size 32 with 5 generators>
```

6.4.3 HAPPRIME_TestResolutionPrimePowerGroup

▷ `HAPPRIME_TestResolutionPrimePowerGroup(`*`[ntests]`*`)` (operation)

Returns: Boolean

Returns `true` if `ResolutionPrimePowerGroupGF` (**HAPprime: ResolutionPrimePowerGroupGF for group**) and `ResolutionPrimePowerGroupRadical` (**HAPprime: ResolutionPrimePowerGroupRadical for group**) appear to be working correctly, or `false` otherwise. This repeatedly creates resolutions of length 6 for random 2-groups (up to order 128) using both of the HAPprime resolution algorithms, and compares them both with the original HAP `ResolutionPrimePowerGroup` (**HAP: ResolutionPrimePowerGroup**) and checks that they are equal. The optional argument *`ntests`* specifies how many resolutions to try: the default is 25.

Index

=
  for FpGModuleGF, 31

AmbientModuleDimension, 20

BestCentralSubgroupForResolution-
    FiniteExtension, 11
BoundaryFpGModuleHomomorphismGF, 9

CanonicalAction, 17
CanonicalActionOnRight, 17
CanonicalGroupAndAction, 17

DirectDecompositionOfModule, 28
DirectDecompositionOfModule-
    Destructive, 28
DirectSumOfModules
  for collection of modules, 28
  for n copies of the same module, 28
  for two modules, 28
DisplayBlocks
  for FpGModuleGF, 20
  for FpGModuleHomomorphismGF, 36
DisplayModuleHomomorphismGenerator-
    Matrix, 36
DisplayModuleHomomorphismGenerator-
    MatrixBlocks, 37

EchelonModuleGenerators, 26
EchelonModuleGeneratorsDestructive, 26
EchelonModuleGeneratorsMinMem, 26
EchelonModuleGeneratorsMinMem-
    Destructive, 26

FpGModuleFromFpGModuleGF, 17
FpGModuleGF
  construction from FpGModule, 16
  construction from generators and groupAn-
    dAction, 16
  construction from generators, group and ac-
    tion, 16
  construction of empty module from group
    and action, 16
  construction of empty module from
    groupAndAction, 16
  construction of full canonical module, 16
  construction of full canonical module from
    groupAndAction, 16
FpGModuleGFNC
  construction for empty module with group
    and action, 16
  construction from generators and groupAn-
    dAction, 16
  construction from generators, group and ac-
    tion, 16
FpGModuleHomomorphismGF, 35
FpGModuleHomomorphismGFNC, 35

HallSeniorNumber
  for group, 43
  for SmallGroup library ref, 43
HAPPRIME_AddGeneratingRowToSemi-
    EchelonBasisDestructive, 45
HAPPRIME_AddNextResolutionBoundaryMap-
    MatNC, 52
HAPPRIME_AddZeroBlocks, 49
HAPPRIME_BoundaryMatrices, 52
HAPPRIME_CoefficientsOfGeneratingRows
  for collection of vectors, 48
  for vector, 48
HAPPRIME_CoefficientsOfGeneratingRows-
    Destructive
  for collection of vectors, 48
  for vector, 48
HAPPRIME_CoefficientsOfGenerating-
    RowsGF
  for collection of vectors, 48
  for vector, 48
HAPPRIME_CoefficientsOfGeneratingRows-
    GFDestructive

for collection of vectors, 48
for vector, 48
HAPPRIME_CoefficientsOfGeneratingRows-GFDestructive2
for collection of vectors, 48
for vector, 48
HAPPRIME_CreateResolutionWithBoundaryMapMatsNC, 52
HAPPRIME_DirectSumForm, 50
HAPPRIME_DisplayGeneratingRows, 46
HAPPRIME_DisplayGeneratingRowsBlocks, 47
HAPPRIME_ExpandGeneratingRow, 45
HAPPRIME_ExpandGeneratingRowOnRight, 45
HAPPRIME_ExpandGeneratingRows, 45
HAPPRIME_ExpandGeneratingRowsOnRight, 45
HAPPRIME_GactFGvector, 48
HAPPRIME_GActMatrixColumns, 45
HAPPRIME_GActMatrixColumnsOnRight, 45
HAPPRIME_GenerateFromGeneratingRows-Coefficients
for collection of vectors, 49
for vector, 49
HAPPRIME_GenerateFromGeneratingRows-CoefficientsGF
for collection of vectors, 49
for vector, 49
HAPPRIME_GeneratingRowsBlockStructure, 47
HAPPRIME_IndependentGeneratingRows, 47
HAPPRIME_IsGroupAndAction, 51
HAPPRIME_KernelOfGeneratingRows-Destructive, 44
HAPPRIME_MinimalGeneratorsVectorSpace-GeneratingRowsDestructive, 50
HAPPRIME_MinimalGeneratorsVector-SpaceGeneratingRowsOnRight-Destructive, 51
HAPPRIME_ModuleElementFromGenerator-Coefficients
for collection of elements, 50
for element, 50
HAPPRIME_ModuleGeneratorCoefficients
for collection of elements, 50
for element, 50
HAPPRIME_ModuleGeneratorCoefficients-Destructive
for collection of elements, 50
for element, 50
HAPPRIME_PrintModuleDescription, 50
HAPPRIME_Random2Group, 52
HAPPRIME_Random2GroupAndAction, 52
HAPPRIME_ReduceGeneratorsOfModuleBy-LeavingOneOut, 46
HAPPRIME_ReduceGeneratorsOfModuleBy-SemiEchelon, 46
HAPPRIME_ReduceGeneratorsOfModuleBy-SemiEchelonDestructive, 46
HAPPRIME_ReduceGeneratorsOnRightBy-LeavingOneOut, 46
HAPPRIME_ReduceVectorDestructive, 46
HAPPRIME_RemoveZeroBlocks, 49
HAPPRIME_SingularIsAvailable, 52
HAPPRIME_TestResolutionPrimePower-Group, 53
HAPPRIME_ValueOptionMaxFGExpansion-Size, 44
HAPPRIME_VectorToWord, 51
HAPPRIME_WordToVector, 51

ImageOfModuleHomomorphism
image of homomorphism, 38
of collections of elements, 38
of element, 38
of module, 38
ImageOfModuleHomomorphismDestructive
of collection of elements, 38
of element, 38
IntersectionModules, 29
IntersectionModulesGF, 29
IntersectionModulesGF2, 29
IntersectionModulesGFDestructive, 29
IsModuleElement
for collection of elements, 31
for element, 31
IsSameSubspace, 42
IsSubModule, 31

KernelOfModuleHomomorphism, 39
KernelOfModuleHomomorphismGF, 39
KernelOfModuleHomomorphismIndependent-Split, 39

KernelOfModuleHomomorphismSplit, 39

LengthOneResolutionPrimePowerGroup, 8
LengthZeroResolutionPrimePowerGroup, 8

MinimalGeneratorsModuleGF, 24
MinimalGeneratorsModuleGFDestructive, 24
MinimalGeneratorsModuleRadical, 24
ModuleAction, 19
ModuleActionBlockSize, 19
ModuleAmbientDimension, 20
ModuleCharacteristic, 20
ModuleField, 20
ModuleGenerators, 22
ModuleGeneratorsAreEchelonForm, 22
ModuleGeneratorsAreMinimal, 22
ModuleGeneratorsForm, 23
ModuleGroup, 19
ModuleGroupAndAction, 19
ModuleGroupOrder, 19
ModuleHomomorphismGeneratorMatrix, 36
ModuleIsFullCanonical, 22
ModuleRank, 23
ModuleRankDestructive, 23
ModuleVectorSpaceBasis, 23
ModuleVectorSpaceDimension, 23
MutableCopyModule, 17

PreImageRepresentativeOfModuleHomomorphism
  for collection of elements, 38
  for element, 38
  for module, 38
PreImageRepresentativeOfModuleHomomorphismGF
  for collection of elements, 38
  for element, 38
PrintDimensionsMat, 42

RadicalOfModule, 24
RandomElement, 32
RandomSubmodule, 32
ResolutionFpGModuleGF, 9
ResolutionGroup, 9
ResolutionLength, 9
ResolutionModuleRank, 9
ResolutionModuleRanks, 9
ResolutionsAreEqual, 10
ReverseEchelonModuleGenerators, 26
ReverseEchelonModuleGeneratorsDestructive, 26

SemiEchelonModuleGenerators, 26
SemiEchelonModuleGeneratorsDestructive, 26
SemiEchelonModuleGeneratorsMinMem, 26
SemiEchelonModuleGeneratorsMinMemDestructive, 26
SolutionMat
  for multiple vectors, 42
SolutionMatDestructive
  for multiple vectors, 42
SourceModule, 36
SumIntersectionMatDestructive, 41
SumIntersectionMatDestructiveSE, 41
SumModules, 29

TargetModule, 36