HAPprime

More homological algebra with prime power groups

Version 0.3.2

13 February 2009

Paul Smith

Paul Smith
— Email: paul.smith@nuigalway.ie
— Homepage: http://www.maths.nuigalway.ie/~pas
— Address: Department of Mathematics,
National University of Ireland, Galway
Galway,
Ireland.

Copyright

Acknowledgements

HAPprime is supported by a Marie Curie Transfer of Knowledge grant based at the Department of Mathematics, NUI Galway (MTKD-CT-2006-042685)

Contents

Chapter 1

Introduction

1.1 Introduction to the HAPprime package

HAPprime is a package for the GAP computer algebra system (http://www.gap-system.org/), and which extends the HAP 'Homological Algebra Progamming' package written by Graham Ellis (http://hamilton.nuigalway.ie/Hap/www/). It provides algorithms and data structures for calculating cohomology ring presentations and resolutions of small prime-power groups. As well as new functions, HAPprime also provides some equivalents for some existing HAP functions that are much more memory-efficient and occasionally faster.

In particular, the main reasons you may want to use HAPprime are

- the calculation of resolutions of prime-power groups in HAPprime uses significantly less memory than the equivalent function in HAP, allowing resolutions (and cohomology ring presentations) of larger groups to be calculated (see Section 2.3);
- HAPprime can compute polynomial ring presentations for cohomology rings calculated using either HAP or HAPprime (see Section 2.2.1);
- we provide a method which ensures that complete and correct cohomology rings are computed. This is a an implementation of Len Evens' original proof of the finite presentation of the cohomology rings (see Section 2.2.2).

1.2 Required software

The HAPprime package requires GAP version 4.4 or greater and HAP version 1.8.9 or greater. For calculating provably-correct cohomology rings, the Singular commutative algebra system (http://www.singular.uni-kl.de/) and the singular GAP package are also required.

1.3 Installing HAPprime

To install the HAPprime Package, unpack the archive file into your GAP packages directory (either usually the pkg directory of your GAP 4 installation if you have access to it, or some local pkg directory that GAP can find). The HAPprime files will all be installed in a subdirectory called happrime-0.3.2.

1.4 Loading and testing HAPprime

The HAPprime package is not loaded by default when GAP is started. To load the package, type the following at the GAP prompt:

Example

```
gap> LoadPackage( "HAPprime");
```

If HAPprime isn't already in memory, it is loaded and the author information is displayed. If you are a frequent user of HAPprime, you might consider putting this line in your `.gaprc` file.

The correct installation of HAPprime can be tested by using the test routine `tst/testall.g`:

Example

```
gap> ReadPackage("HAPprime", "tst/testall.g");
+ HAPprime version 0.3.2 general tests
+ GAP4stones: 371057
+ HAPprime version 0.3.2 userguide examples
+ GAP4stones: 387662
+ HAPprime version 0.3.2 datatypes reference manual examples
+ GAP4stones: 382653
true
```

The number of `GAP4stones` will vary depending on your machine, but any additional lines of messages indicate problems with your installation.

The test routine calls a set of test files (**Reference: Test Files**) which can be found in the `tst` directory of the HAPprime installation. All of the routines listed in this user guide are tested, as are many of those in the datatype reference manual.

1.5 Documentation

The documentation for HAPprime is in two parts. This document is the user guide, which covers the main functions provided by HAPprime and examples of their use. There is also a more technical HAPprime datatypes reference manual which gives details of the new GAP datatypes defined and used internally by HAPprime, as well as outlining the algorithms used by the package.

1.5.1 MakeHAPprimeDoc

◊ `MakeHAPprimeDoc(`*`[manual-name]`*`)` (function)

Returns: nothing

The two manuals supplied with HAPprime - this user guide and the datatypes reference manual - are written using the GAPDoc package and are available in PDF, HTML and text format. It should not be necessary to rebuild these files, but should you wish to do so then this can be done using the function `MakeHAPprimeDoc`.

The optional argument *`manual-name`* is a string specifying which manuals to build. It may be one of the following

- `"all"` builds both manuals. This is the default
- `"userguide"` builds just the user guide
- `"datatypes"` builds just the datatypes reference manual

- `"internal"` builds both manuals, including the otherwise undocumented internal functions
- `"testexamples"` builds neither manual, but tests all of the examples using `TestManualExamples` (**GAPDoc: TestManualExamples**)

As well as building the manuals, this function at the same time builds GAP test files (**Reference: Test Files**) which means that all of the testable examples in the manuals are added to the HAPprime test routines described in Section 1.4.

1.6 Displaying progress and calculation information

By default, the functions in HAPprime display no output (except for returning the result). The `InfoHAPprime` info class can be used to enable the printing of progress and calculation information during processing. Since some computations with HAPprime can take several hours, setting this to a higher level can be particularly useful for monitoring the progress of computations.

1.6.1 InfoHAPprime

◊ `InfoHAPprime` (info class)

The `InfoHAPprime` info class is used throughout the HAPprime package. Use `SetInfoLevel(InfoHAPprime, level)` to change the amount of information displayed about the progress of the computation (see `SetInfoLevel` (**Reference: SetInfoLevel**) in the GAP reference manual). The different distinct levels are:

- `0` print nothing (this is the default)
- `1` print some information, mainly progress information during computations that may take some time
- `2` print more detailed information, incluing details of internal calculations

Chapter 2

Examples

2.1 Computing the mod p group cohomology

Let G be a group and $\mathbb{F}$ be a field, and let $\mathbb{F}G$ be the group ring of G over $\mathbb{F}$. A free $\mathbb{F}G$-resolution of the ground ring is an exact sequence of module homomorphisms

$$\ldots \to M_{n+1} \to M_n \to M_{n-1} \to \ldots \to M_1 \to \mathbb{F}G \twoheadrightarrow \mathbb{F}$$

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 $n > 0$. The maps d_n are called boundary homomorphisms. In HAPprime we consider the case where G is a p-group and $\mathbb{F}$ is the prime field $GF(p)$, and this is assumed from now on.

If we now define the Abelian group D_n to be $Hom(M_n, \mathbb{F})$, the set of all homomorphisms $M_n \to \mathbb{F}$, we can obtain the dual of this sequence, which will be a cochain complex of Abelian group homomorphisms

$$\ldots \leftarrow D_{n+1} \leftarrow D_n \leftarrow D_{n-1} \leftarrow \ldots \leftarrow D_1 \leftarrow \mathbb{F} \leftarrow \mathbb{F}$$

Each group D_n will be isomorphic to $\mathbb{F}^{|M_n|}$ where $|M_n|$ is the rank of the module M_n. Unlike the resolution, this sequence will generally not be exact, but one can define the mod-p cohomology group of G at degree n to be

$$H^n(G, \mathbb{F}) = \frac{ker(D_n \to D_{n+1})}{im(D_{n-1} \to D_n)}$$

for all $n > 0$. As with the D_n, the mod-p cohomology groups will also be isomorphic to vector spaces over $\mathbb{F}$. In the case where the resolution R is minimal (where each module M_n has the minimal number of generators), the dimensions of the (co)homology groups $H^n(G, \mathbb{F})$ are identical to the dimensions of the resolution modules M_n. The group cohomology (and the similar group homology) is an invariant of G, and does not depend on a particular free $\mathbb{F}G$-resolution.

In the general case, there are thus two stages to computing the group cohomology of G up to the nth cohomology group:

1. Compute R, a free $\mathbb{F}G$-resolution for $\mathbb{F}G$, with at least $n+1$ terms.

2. Construct the cochain complex C from R and compute the n homology groups of C

For example, to calculate the 9th mod-p cohomology group of the 134th order 64 in the GAP small groups library (which is the Sylow 2-subgroup of the Mathieu group M_{12}), we can use the HAPprime

function ResolutionPrimePowerGroupRadical (3.1.1) to compute 10 terms of a free $\mathbb{F}G$-resolution for G and then use HAP functions to find the rank b_9 of the cohomology group, which will be isomorphic to $\mathbb{F}^{b_9}$. Alternatively, since ResolutionPrimePowerGroupRadical (3.1.1) always returns a minimal resolution, the cohomology group dimensions can be read directly from the resolution.

Example

```
gap> G := SmallGroup(64, 134);;
gap> # First construct a FG-resolution for the group G
gap> R := ResolutionPrimePowerGroupRadical(G, 10);
Resolution of length 10 in characteristic 2 for <pc group of size 64 with
6 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> # Convert this into a cochain complex (over the prime field with p=2)
gap> C := HomToIntegersModP(R, 2);
Cochain complex of length 10 in characteristic 2 .

gap> # And get the rank of the 9th cohomology group
gap> b9 := Cohomology(C, 9);
55
gap>
gap> # Since R is a free resolution, the ranks of the cohomology groups
gap> # are the same as the module ranks in R
gap> ResolutionModuleRanks(R);
[ 3, 6, 10, 15, 21, 28, 36, 45, 55, 66 ]
```

2.2 Computing mod-p cohomology rings and their Poincaré series

The mod-p group cohomology of a p-group G, given a field $\mathbb{F} = GF(p)$, has a multiplicative structure, and so the group

$$H^*(G,\mathbb{F}) = \bigoplus_{i=0}^{\infty} H^i(G,\mathbb{F})$$

is a ring. Since $H^*(G,\mathbb{F})$ is isomorphic to a vector space over $\mathbb{F}$, it is also an algebra, in fact a graded algebra: for elements $e \in H^n(G,\mathbb{F})$ and $f \in H^m(G,\mathbb{F})$, the product ef is an element of $H^{m+n}(G,\mathbb{F})$.

Some functions for investigating the ring structure of $H^*(G,\mathbb{F})$ using GAP are already provided by HAP, and also by Marcus Bishop's Crime package http://www.math.uic.edu/~marcus/Crime/. There have also been implementations using other systems, in particular, Jon Carlson has computed the cohomology rings for all 2-groups of order 64 and fewer using MAGMA (see http://www.math.uga.edu/~lvalero/cohointro.html for results) and David Green has calculated the same, and some of order 128, using C (see http://www.math.uni-wuppertal.de/~green/Coho_v2/index.html for results).

2.2.1 A ring presentation for the mod p cohomology (up to degree n)

The cohomology ring $H^*(G,\mathbb{F})$ is an infinite vector space, but if all elements of degree higher than some degree n are ignored, a related finite algebra can be considered. Multiplication in this finite algebra can be represented by a multiplication table giving the product of each basis element in the

vector space (up to products of degree n). The HAP function `ModPCohomologyRing` (**HAP: ModPCohomologyRing**) computes such a finite algebra representation for the mod-p cohomology ring of a p-group G for a given value of n.

The full infinite-dimensional cohomology ring has a finite presentation as a quotient ring

$$H^*(G,\mathbb{F}) = \frac{\mathbb{F}[x_1, x_2, \ldots, x_n]}{\langle I_1, I_2, \ldots, I_m \rangle}$$

where the polynomial ring indeterminates x_i each have an associated degree d_i and the I_j are relations which together generate an ideal in the ring. Given a finite algebra A, the HAPprime function `ModPCohomologyRingPresentation` (3.3.1) calculates a presentation for the ring, modulo any generating elements of degree higher than n. If $H^*(G,\mathbb{F})$ has no generators or relations in degrees higher than n, then a ring presentation for an algebra computed via `ModPCohomologyRing` (**HAP: ModPCohomologyRing**) followed by `ModPCohomologyRingPresentation` (3.3.1) will be the same as a ring presentation for $H^*(G,\mathbb{F})$.

We shall calculate a ring presentation for the group `G := SmallGroup(16, 3)`:

Example

```
gap> G := SmallGroup(16, 3);;
gap> A := ModPCohomologyRing(G, 5);
<algebra of dimension 34 over GF(2)>
gap> ModPCohomologyRingPresentation(A);
Graded algebra GF(2)[ x_1, x_2, x_3, x_4, x_5 ] /
[ x_1*x_2, x_1^2, x_1*x_5, x_2^2*x_3+x_5^2 ] with indeterminate degrees
[ 1, 1, 2, 2, 2 ]
```

The object returned by `ModPCohomologyRingPresentation` (3.3.1) tells us that

$$H^*(G,\mathbb{F}) = \frac{\mathbb{F}[x_1, x_2, x_3, x_4, x_5]}{\langle x_1^2,\ x_1x_2,\ x_1x_5,\ x_2^2x_3 + x_5^2 \rangle}$$

where the indeterminates x_1 and x_2 are in degree one and the rest are in degree two. This assumes, however, that there are no generating elements in any degree higher than five. See Section 2.2.2 below for a method that also computes the maximum degree necessary to present the cohomology ring, and so avoids this limitation.

As well as taking the algebra as an argument, the function `ModPCohomologyRingPresentation` (3.3.1) can also take a group or a free $\mathbb{F}G$-resolution, and a value for n, in which case it performs the calculation of the algebra as well.

2.2.2 Calculating a provably-correct mod-p cohomology

Using the HAP function `ModPCohomologyRing` (**HAP: ModPCohomologyRing**), HAPprime can calculate a presentation for the cohomology ring $H^*(G,\mathbb{F})$ as described in Section 2.2.1) It is known (Evens 1961) that mod-p cohomology rings are finitely-generated, and hence given a sufficiently-large n the presentation calculated in the above manner will be the complete ring. HAPprime can compute a sufficient value for n (in fact, it will be near, and usually at, the minimum). HAPprime uses the original Evens proof constructively. For a group G, a Lyndon-Hochschild-Serre spectral sequence can be computed. This will converge, and the limiting sheet of the sequence will be a ring which is an associated graded ring of the mod-p cohomology ring $H^*(G,\mathbb{F})$. This means that, while it will not necessarily be isomorphic to $H^*(G,\mathbb{F})$, it will have the same additive structure, and the generators

and relations will lie in the same degrees. Thus the maximum degree in the presentation for the last sheet of the spectral sequence is a sufficient value for n. The spectral sequence computation requires minimal resolutions and correct cohomology rings for two smaller groups (a central subgroup N and the quotient group G/N), but these rings can be in turn computed by the same process (and the cohomology rings for some small groups can simply be stated). The cohomology ring for G will thus be proved correct by induction.

When called with just a group (i.e. no value for n), `ModPCohomologyRingPresentation` (3.3.1) performs this spectral sequence calculation to find n and then calculates the cohomology ring using this value for n. In the example below we use this function, and by setting the value of `InfoHAPprime` (1.6.1) to 1, we can see the details of the spectral sequence calculation.

Example

```
gap> SetInfoLevel(InfoHAPprime, 1);
gap> G := SmallGroup(16, 3);;
gap> A := ModPCohomologyRingPresentation(G);
#I  E_2 = GF(2)[ x_1, x_2 ] x GF(2)[ x_3, x_4 ]
#I    with generator degrees [ 1, 1 ] and [ 1, 1 ] respectively
#I    d_2(x_1) = zero
#I    d_2(x_2) = zero
#I    d_2(x_3) = x_1*x_2
#I    d_2(x_4) = x_1^2
#I  E_3 = GF(2)[ x_5, x_6, x_7, x_8, x_9 ]/[ x_6*x_9, x_6^2, x_5*x_6, x_5^2*x\
_7+x_9^2 ]
#I  E_3 = GF(2)[ x_2, x_1, x_4^2, x_3^2, x_1*x_3+x_2*x_4 ]/[ x_1^2*x_3+x_1*x_\
2*x_4, x_1^2, x_1*x_2, x_1^2*x_3^2 ]
#I    d_3(x_2) = zero
#I    d_3(x_1) = zero
#I    d_3(x_4^2) = zero
#I    d_3(x_3^2) = x_1^2*x_2+x_1*x_2^2 = 0*Z(2) mod I
#I    d_3(x_1*x_3+x_2*x_4) = zero
#I  E_4 = GF(2)[ x_10, x_11, x_12, x_13, x_14 ]/[ x_11*x_14, x_11^2, x_10*x_1\
1, x_10^2*x_12+x_14^2 ]
#I  E_4 = GF(2)[ x_2, x_1, x_4^2, x_3^2, x_1*x_3+x_2*x_4 ]/[ x_1^2*x_3+x_1*x_\
2*x_4, x_1^2, x_1*x_2, x_1^2*x_3^2 ]
#I    d_4(x_2) = zero
#I    d_4(x_1) = zero
#I    d_4(x_4^2) = zero
#I    d_4(x_3^2) = zero
#I    d_4(x_1*x_3+x_2*x_4) = zero
#I  E_inf = GF(2)[ x_10, x_11, x_12, x_13, x_14 ]/[ x_11*x_14, x_11^2, x_10*x\
_11, x_10^2*x_12+x_14^2 ]
#I  E_inf = GF(2)[ x_2, x_1, x_4^2, x_3^2, x_1*x_3+x_2*x_4 ]/[ x_1^2*x_3+x_1*\
x_2*x_4, x_1^2, x_1*x_2, x_1^2*x_3^2 ]
#I  Renaming indeterminates and sorting into increasing degree
Graded algebra GF(2)[ x_1, x_2, x_3, x_4, x_5 ] /
[ x_1*x_2, x_1^2, x_1*x_5, x_2^2*x_3+x_5^2 ] with indeterminate degrees
[ 1, 1, 2, 2, 2 ]
gap> # Now extract data about the presentation
gap> BaseRing(A);
GF(2)[x_1,x_2,x_3,x_4,x_5]
gap> GeneratorsOfPresentationIdeal(A);
[ x_1*x_2, x_1^2, x_1*x_5, x_2^2*x_3+x_5^2 ]
gap> IndeterminateDegrees(A);
```

```
[ 1, 1, 2, 2, 2 ]
gap> MaximumDegreeForPresentation(A);
4
```

This (the correct cohomology ring) is in fact the same as the presentation computed in Section 2.2.1 using $n = 5$ since there are no generators or relations of degree greater than four.

2.2.3 Computing Poincaré series

The Poincaré series for the mod-p cohomology ring $H^*(G,\mathbb{F})$ is the infinite series

$$a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots$$

where a_k is the dimension of the vector space $H^k(G,\mathbb{F})$. The Poincaré function is a rational function $P(x)/Q(x)$ which is equal to the Poincaré series.

The ranks of the modules in a minimal resolution for a group G are identical to the dimensions a_k, so a minimal resolution can be used to calculate the Poincaré series without first calculating the cohomology ring. This is the method used by the HAP function `PoincareSeries` (**HAP: PoincareSeries**), but will only give the correct answer for a sufficiently-long resolution. HAP has a method for calculating a resolution that is likely to be long enough, but cannot prove that this is sufficient.

HAPprime can instead calculate a provably-correct Poincaré series for a mod-p cohomology ring, and without first calculating the ring itself. The final sheet of a Lyndon-Hochschild-Serre spectral sequence for a group G will be a ring with the same additive structure as the cohomology ring for G. This will thus have the same Poincaré series, and can be used to provide the Poincaré series for the cohomology ring without having to also compute the cohomology ring. This is implemented in the HAPprime function `PoincareSeriesLHS` (3.2.1).

As well as being provably correct, `PoincareSeriesLHS` (3.2.1) is also often faster than the related HAP function, as demonstrated in this example:

Example

```
gap> G := SmallGroup(64, 210);;
gap> # Compute the Poincare series using HAP
gap> P1 := PoincareSeries(G);time;
(x_1^4+x_1^2+x_1+1)/(-x_1^7+3*x_1^6-5*x_1^5+7*x_1^4-7*x_1^3+5*x_1^2-3*x_1+1)
46434
gap> # Compute the Poincare series using HAPprime
gap> P2 := PoincareSeriesLHS(G);time;
(x_1^4+x_1^2+x_1+1)/(-x_1^7+3*x_1^6-5*x_1^5+7*x_1^4-7*x_1^3+5*x_1^2-3*x_1+1)
1889
gap> P1 = P2;
true
```

In this case, HAP needs to compute 14 terms of a resolution for this group of order 64 before it is confident that it has a stable Poincaré series. By contrast HAPprime, in calculating the spectral sequence, needs to compute 5 terms of two resolutions of groups of order 8 and then construct a (non-minimal) resolution for G (also of length 5) from these two. Both methods give the same answer, but only HAPprime's is guaranteed to be correct.

2.3 Comparing the memory usage and speed of HAPprime and HAP's ResolutionPrimePowerGroup functions

For small p-groups, the group ring $\mathbb{F}G$ can be considered as a vector space of rank $|G|$ with the elements of G as its basis elements. Each module M_n in a $\mathbb{F}G$-resolution is also a vector space (of dimension $|M_n||G|$) and the boundary maps d_n can be represented as vector space homomorphisms. As a result, standard linear algebra techniques can be used to compute a minimal resolution by constructing a sequence of module homomorphisms where the kernel of one map is the image of the next, and where the modules have minimal generating sets. See Chapter (**HAPprime Datatypes: Resolutions**) in the datatypes manual for further details.

As the groups get larger, this approach becomes less feasible due to the amount of time and memory needed to store and compute the null space of large matrices. The HAP function `ResolutionPrimePowerGroup` (**HAP: ResolutionPrimePowerGroup**) and the HAPprime functions `ResolutionPrimePowerGroupRadical` (3.1.1) and `ResolutionPrimePowerGroupGF` (3.1.1) all use this linear algebra approach, but the HAPprime functions are optimised to save memory, allowing the computation of resolutions which are longer, or are of larger groups, than are possible using HAP alone.

2.3.1 HAPprime takes less memory to store resolutions

Consider computing a resolution of a group of an arbitrary group of order 128, `G = SmallGroup(128, 844)` using HAP. Computation is performed on a dual-core Intel Core2Duo running at 2.66MHz, and the memory available to GAP is the standard initial allocation of 256Mb.

Example

```
gap> G := SmallGroup(128, 844);;
gap> R := ResolutionPrimePowerGroup(G, 9);
Resolution of length 9 in characteristic 2 for <pc group of size 128 with
7 generators> .

gap> time;
27685
gap> # Can we construct a resolution of length ten?
gap> R := ResolutionPrimePowerGroup(G, 10);
exceeded the permitted memory ('-o' command line option) at
res := SemiEchelonMatDestructive( List( mat, ShallowCopy ) );
 called from
SemiEchelonMat( NullspaceMat( BndMat ) ) called from
ZGbasisOfKernel( i - 1 ) called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
```

The HAPprime function `ResolutionPrimePowerGroupRadical` (3.1.1) uses an almost identical algorithm, but stores its boundary maps more efficiently. As a result, with the same memory allowance:

Example

```
gap> G := SmallGroup(128, 844);;
gap> R := ResolutionPrimePowerGroupRadical(G, 9);
Resolution of length 9 in characteristic 2 for <pc group of size 128 with
7 generators> .
```

```
No contracting homotopy available.
A partial contracting homotopy is available.

gap> time;
25321
gap> # Can we construct a resolution of length ten?
gap> R := ExtendResolutionPrimePowerGroupRadical(R);;
gap> # Yes! How about eleven?
gap> R := ExtendResolutionPrimePowerGroupRadical(R);
Resolution of length 11 in characteristic 2 for <pc group of size 128 with
7 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> ResolutionModuleRanks(R);
[ 3, 6, 11, 19, 30, 44, 62, 85, 113, 146, 185 ]
gap>
gap> # But it will run out of memory if we try to go to twelve terms
gap> R := ExtendResolutionPrimePowerGroupRadical(R);
exceeded the permitted memory (‘-o’ command line option) at
...
```

The HAPprime version can compute two further terms of the resolution, which given the sizes of the additional modules represents a considerable improvement. Just representing the homomorphism $d_{10} : (\mathbb{F}G)^{146} \to (\mathbb{F}G)^{113}$ as vectors requires nearly as much memory again as representing the first nine homomorphisms. To compute and store the same resolution of length 11 using `ResolutionPrimePowerGroup` (**HAP: ResolutionPrimePowerGroup**) would need a little over three times the memory used here by HAPprime. The time taken by both versions is very similar.

In the example above, note also the use of the HAPprime function `ExtendResolutionPrimePowerGroupRadical` (3.1.2), which makes it much easier to add terms to an existing resolution. In standard HAP, if one decides that a resolution is too short and that more terms are required, then the entire resolution must be computed again from scratch.

2.3.2 HAPprime takes less memory to compute resolutions

The function `ResolutionPrimePowerGroupGF` (3.1.1) uses a new algorithm to compute the kernel of $\mathbb{F}G$-module homomorphisms when $\mathbb{F}G$-modules are represented using a set of G-generating vectors (see (**HAPprime Datatypes: FG-module homomorphisms**) in the datatypes reference manual). This provides a further memory saving over `ResolutionPrimePowerGroupRadical` (3.1.1), although at the cost of a much slower computation time:

Example

```
gap> G := SmallGroup(128, 844);;
gap> R := ResolutionPrimePowerGroupGF(G, 9);
Resolution of length 9 in characteristic 2 for <pc group of size 128 with
7 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> time;
422742
gap> R := ExtendResolutionPrimePowerGroupGF(R);;
```

```
gap> R := ExtendResolutionPrimePowerGroupGF(R);;
gap> R := ExtendResolutionPrimePowerGroupGF(R);;
gap> R := ExtendResolutionPrimePowerGroupGF(R);;
gap> R := ExtendResolutionPrimePowerGroupGF(R);;
gap> R := ExtendResolutionPrimePowerGroupGF(R);;
Resolution of length 15 in characteristic 2 for <pc group of size 128 with
7 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> ResolutionModuleRanks(R);
[ 3, 6, 11, 19, 30, 44, 62, 85, 113, 146, 185, 231, 284, 344, 412 ]
gap> # But it will run out of (the inital 256Mb) of memory at sixteen terms
```

Using `ResolutionPrimePowerGroupGF` (3.1.1) we can get a further four terms of the resolution. For this resolution, this represents a memory saving of a factor of five over `ResolutionPrimePowerGroupRadical` (3.1.1) and fifteen over `ResolutionPrimePowerGroup` (**HAP: ResolutionPrimePowerGroup**), although it does take fifteen times as long as either of those just to compute the first nine terms, and scales less well with size.

2.3.3 Automatic selection of the best method

The two functions `ResolutionPrimePowerGroupRadical` (3.1.1) and `ResolutionPrimePowerGroupGF` (3.1.1) offer a trade-off between time and memory. The function `ResolutionPrimePowerGroupAutoMem` (3.1.1) automates the decision of which version to use, switching from the `Radical` to the `GF` version when it estimates that it is about to run out of available memory for the faster version. In this example, we have also increase the `InfoHAPprime` (1.6.1) info level to display progress information. At level two, the rank of each module in the resolution is displayed as it is calculated, giving an indication of progress. With this setting, the user is also notified when the `AutoMem` function switches, and the `GF` function displays a rolling estimate of its completion time (which is not shown since that output is overwritten when completed)

Example

```
gap> G := SmallGroup(128, 844);;
gap> SetInfoLevel(InfoHAPprime, 2);
gap> R := ResolutionPrimePowerGroupAutoMem(G, 15);
#I  Dimension 2: rank 6
#I  Dimension 3: rank 11
#I  Dimension 4: rank 19
#I  Dimension 5: rank 30
#I  Dimension 6: rank 44
#I  Dimension 7: rank 62
#I  Dimension 8: rank 85
#I  Dimension 9: rank 113
#I  Finding kernel of homomorphism by splitting:
#I   - Finding kernel of U
#I   - Finding kernel of V
#I   - Finding intersection of U and V
#I   - Finding intersection preimages
#I  Dimension 10: rank 146
#I  Finding kernel of homomorphism by splitting:
#I   - Finding kernel of U
```

```
#I   - Finding kernel of V
#I   - Finding intersection of U and V
#I   - Finding intersection preimages
#I  Dimension 11: rank 185
#I  Finding kernel of homomorphism by splitting:
#I   - Finding kernel of U
#I   - Finding kernel of V
#I   - Finding intersection of U and V
#I   - Finding intersection preimages
#I  Dimension 12: rank 231
#I  Finding kernel of homomorphism by splitting:
#I   - Finding kernel of U
#I   - Finding kernel of V
#I   - Finding intersection of U and V
#I   - Finding intersection preimages
#I  Dimension 13: rank 284
#I  Finding kernel of homomorphism by splitting:
#I   - Finding kernel of U
#I   - Finding kernel of V
#I   - Finding intersection of U and V
#I   - Finding intersection preimages
#I  Dimension 14: rank 344
#I  Finding kernel of homomorphism by splitting:
#I   - Finding kernel of U
#I   - Finding kernel of V
#I   - Finding intersection of U and V
#I   - Finding intersection preimages
#I  Dimension 15: rank 412
Resolution of length 15 in characteristic 2 for <pc group of size 128 with
7 generators> .
No contracting homotopy available.
A partial contracting homotopy is available.

gap> StringTime(time);
" 5:45:53.613"
```

Chapter 3

Functions for Homological Algebra

3.1 Resolutions

3.1.1 ResolutionPrimePowerGroup

◊ `ResolutionPrimePowerGroupRadical(G, n)` (operation)
◊ `ResolutionPrimePowerGroupGF(G, n)` (operation)
◊ `ResolutionPrimePowerGroupAutoMem(G, n)` (operation)
◊ `ResolutionPrimePowerGroupGF2(G, n)` (operation)
◊ `ResolutionPrimePowerGroupRadical(M, n)` (operation)
◊ `ResolutionPrimePowerGroupGF(M, n)` (operation)
◊ `ResolutionPrimePowerGroupAutoMem(M, n)` (operation)
◊ `ResolutionPrimePowerGroupGF2(M, n)` (operation)

Returns: `HAPResolution`

Returns n terms of a minimal free $\mathbb{F}G$-resolution for either the ground ring of a prime power group G or of a module M. For the module version, M must be passed as an `FpGModuleGF` object - see (**HAPprime Datatypes: FG-modules**) in the HAPprime datatypes reference manual.

Three versions of this function are provided:

`ResolutionPrimePowerGroupRadical` uses the same resolution-building method as the HAP function `ResolutionPrimePowerGroup` (**HAP: ResolutionPrimePowerGroup**), but stores the resolution in a different format that takes only about half the memory of the HAP version.

`ResolutionPrimePowerGroupGF` calculates the resolution using HAPprime's G-generator form of modules, which reduces memory use by around a factor of two over `ResolutionPrimePowerGroupRadical`, but is slower by an order of magnitude.

`ResolutionPrimePowerGroupAutoMem` automatically switches between the two previous versions based on the available memory. It uses the `Radical` version until it gets close to the limit of the available memory, and then switches to the `GF` version.

`ResolutionPrimePowerGroupGF2` calculates the resolution by $\mathbb{F}G$-matrix partitioning. The amount of partitioning is governed by the (**Reference: Options Stack**) option `MaxFGExpansionSize`. The default value means that until the boundary map takes about 128Mb, the method is equivalent to `ResolutionPrimePowerGroupRadical`, and then it tends towards `ResolutionPrimePowerGroupGF` in terms of time, but saves less memory.

See the HAPprime datatypes reference manual for details of the different algorithms, in particular the chapters on the G-generator form of $\mathbb{F}G$-modules (**HAPprime Datatypes: FG-modules**) and $\mathbb{F}G$-module homomorphisms (**HAPprime Datatypes: FG-module homomorphisms**) and on resolutions (**HAPprime Datatypes: Resolutions**).

3.1.2 ExtendResolutionPrimePowerGroup

◊ `ExtendResolutionPrimePowerGroupRadical(R)` (operation)
◊ `ExtendResolutionPrimePowerGroupGF(R)` (operation)
◊ `ExtendResolutionPrimePowerGroupAutoMem(R)` (operation)
◊ `ExtendResolutionPrimePowerGroupGF2(R)` (operation)

Returns: `HAPResolution`

Returns the resolution `R` extended by one term. The three variants offer a choice between memory and speed, and correspond to the different versions of `ResolutionPrimePowerGroup` in HAPprime. See the documentation (3.1.1) for those functions for a description of the different variants.

3.2 Poincaré Series

3.2.1 PoincareSeriesLHS

◊ `PoincareSeriesLHS(G)` (attribute)

Returns: Rational function

For a finite p-group `G`, this function calculates and returns a quotient of polynomials $f(x) = P(x)/Q(x)$ (i.e. the Poincaré series) whose coefficient of x^k equals the rank of the vector space $H_k(G,\mathbb{F}_p)$ for all k in the range $k = 1$ to $k = n$.

This function computes a Lyndon-Hoschild-Serre spectral sequence for the p-group G. The last sheet of this sequence will have the same additive structure as the mod-p group cohomology ring of G, and thus the same Poincaré series, which is returned by this function.

See Section 2.2.3 for an example and more description.

3.3 Cohomology Ring structure

3.3.1 ModPCohomologyRingPresentation (for group)

◊ `ModPCohomologyRingPresentation(G)` (attribute)
◊ `ModPCohomologyRingPresentation(G, n)` (operation)
◊ `ModPCohomologyRingPresentation(R)` (operation)
◊ `ModPCohomologyRingPresentation(A)` (operation)

Returns: `GradedAlgebraPresentation`

Calculates and returns a cohomology ring presentation for the group G. See (**HAPprime Datatypes: Presentations of graded algebras**) in the datatypes reference manual for details of the `GradedAlgebraPresentation` type.

If the only argument is a p-group `G` then this function computes and returns the provably-correct cohomology ring presentation. This version first computes the Lyndon-Hoschild-Serre Spectral Sequence until convergence to find the additive structure of the cohomology ring, and then computes the cohomology ring up to and including the maximum necessary generator or relation, using the `(G, n)` method described below. For certain groups, the cohomology ring is returned without computation:

the known mod-p cohomology ring presentation for cyclic groups is returned without calculation, and for groups which can be expressed as a direct product, the cohomology ring is computed as a tensor product of its direct factors (thus the cohomology ring of all Abelian groups are also returned with minimal computation.)

When given a p-group `G` and integer `n`, this function computes the presentation modulo all elements of degree greater `n`. Alternatively, a minimal resolution `R` (with `n` terms) can be input, or a structure constant algebra `A` with embedded degrees (from `ModPCohomologyRing` (**HAP: ModPCohomologyRing**)).

See Section 2.2.1 and 2.2.2 for examples and more description. See also `LHSSpectralSequence` (**HAPprime Datatypes: LHSSpectralSequence**) for details of options that can be used to guide the spectral sequence computation.

Index