Share Package — Polycyclic

by

Bettina Eick
Fachbereich Mathematik
Universität Kassel

and

Werner Nickel
Fachbereich Mathematik
TU Darmstadt

Contents

1 Polycyclic Groups

A group G is called **polycyclic** if there exists a subnormal series in G with cyclic factors. Every polycyclic group is soluble and every supersoluble group is polycyclic. The class of polycyclic groups is closed with respect to forming subgroups, factor groups and extensions. Polycyclic groups can also be characterised as those soluble groups in which each subgroup is finitely generated.

K. A. Hirsch has initiated the investigation of polycyclic groups in 1938, see [Hir38a], [Hir38b], [Hir46], [Hir52], [Hir54], and their central position in infinite group theory has long been recognised since.

A well-known result of Hirsch asserts that each polycyclic group is finitely presented. In fact, a polycyclic group has a presentation which exhibits its polycyclic structure: a **pc-presentation** as defined in Section 2.1. Pc-presentations allow efficient computations with the groups they define. In particular, the word problem is efficiently solvable in a group given by a pc-presentation. Further, subgroups and factor groups of groups given by a pc-presentation can be handled effectively.

The share package polycyclic for GAP 4 is designed for computations with polycyclic groups which are given by a pc-presentation. The package contains methods to solve the word problem in such groups and to handle subgroups and factor groups of polycyclic groups. Based on these basic algorithms we present a collection of methods to construct polycyclic groups and to investigate their structure.

In [BCRS91] and [Seg90] the theory of problems which are decidable in polycyclic-by-finite groups has been started. As a result of these investigation we know that a large number of group theoretic problems are decidable by algorithms in polycyclic groups. However, practical algorithms which are suitable for computer implementations have not been obtained by this study. We have developed a new set of practical methods for groups given by pc-presentations, see for example [Eic00], and this package is a collection of implementations for these and other methods.

We refer to [Rob82], page 147ff, and [Seg83] for background on polycyclic groups. Further, in [Sim94] a variation of the basic methods for groups with pc-presentation is introduced. Finally, we note that the main GAP library contains many practical algorithms to compute with finite polycyclic groups, see Section 43 of the reference manual.

2 Pcp-groups - polycyclically presented groups

2.1 Introduction

Let G be a polycyclic group and let $G = C_1 \rhd C_2 \ldots C_n \rhd C_{n+1} = 1$ be a **polycyclic series**, that is, a subnormal series of G with non-trivial cyclic factors. For $1 \leq i \leq n$ we choose $g_i \in C_i$ such that $C_i = \langle g_i, C_{i+1} \rangle$. Then the sequence $(g_1, \ldots, g_n)$ is called a **polycyclic generating sequence of** G. Let I be the set of those $i \in \{1, \ldots, n\}$ with $r_i := [C_i : C_{i+1}]$ finite. Each element of G can be written *uniquely* as $g_1^{e_1} \cdots g_n^{e_n}$ with $e_i \in \mathbb{Z}$ for $1 \leq i \leq n$ and $0 \leq e_i < r_i$ for $i \in I$.

Each polycyclic generating sequence of G gives raise to a **power-conjugate (pc-) presentation** for G with the conjugate relations

$$g_i^{g_j} = g_{j+1}^{e(i,j,j+1)} \cdots g_n^{e(i,j,n)} \text{ for } 1 \leq j < i \leq n,$$

$$g_i^{g_j^{-1}} = g_{j+1}^{f(i,j,j+1)} \cdots g_n^{f(i,j,n)} \text{ for } 1 \leq j < i \leq n,$$

and the power relations

$$g_i^{r_i} = g_{i+1}^{l(i,i+1)} \cdots g_n^{l(i,n)} \text{ for } i \in I.$$

Vice versa, we say that a group G is defined by a pc-presentation if G is given by a presentation of the form above on generators $g_1, \ldots, g_n$. These generators are the **defining generators** of G. Here, I is the set of $1 \leq i \leq n$ such that g_i has a power relation. The positive integer r_i for $i \in I$ is called the relative order of g_i. If G is given by a pc-presentation, then G is polycyclic. The subgroups $C_i = \langle g_i, \ldots, g_n \rangle$ form a subnormal series $G = C_1 \geq \ldots \geq C_{n+1} = 1$ with cyclic factors and we have that $g_i^{r_i} \in C_{i+1}$. However, some of the factors of this series may be smaller than r_i for $i \in I$ or finite if $i \notin I\cdot$

If G is defined by a pc-presentation, then each element of G can be described by a word of the form $g_1^{e_1} \cdots g_n^{e_n}$ in the defining generators with $e_i \in \mathbb{Z}$ for $1 \leq i \leq n$ and $0 \leq e_i < r_i$ for $i \in I$. Such a word is said to be in **collected form**. In general, an element of the group can be represented by more than one collected word. If the pc-presentation has the property that each element of G has precisely one word in collected form, then the presentation is called **confluent** or **consistent**. If that is the case, the generators with a power relation correspond precisely to the finite factors in the polycyclic series and r_i is the order of C_i/C_{i+1}.

The polycyclic share package is designed for computations with polycyclic groups which are given by consistent pc-presentations. In particular, all the functions described below assume that we compute with a group defined by a consistent pc-presentation. See Section 2.2 for a routine that checks the consistency of a pc-presentation.

2.2 Collectors

Let G be a group defined by a pc-presentation as described in Section 2.1.

The process for computing the collected form for an arbitrary word in the generators of G is called **collection**. The basic idea in collection is the following. Given a word in the defining generators, one scans the word for occurrences of adjacent generators (or their inverses) in the wrong order or occurrences of subwords $g_i^{e_i}$ with $i \in I$ and e_i not in the range $0 \ldots r_{i-1}$. In the first case, the appropriate conjugacy relation is used to move the generator with the smaller index to the left. In the second case, one uses the appropriate power relation to move the exponent of g_i into the required range. These steps are repeated until a collected word is obtained.

There are a number of different strategies for collecting a given word to collected form. The strategy implemented in this package is **collection from the left** as described by [LGS90] and [Sim94]. We note that the collection method in this package is currently implemented in GAP code. As this is a very time-critical function, we plan to translate this to C-code in the near future.

The first step in defining a pc-presented group is setting up a data structure that knows the pc-presentation and has routines that perform the collection algorithm with words in the generators of the presentation. Such a data structure is called **a collector**. In this section we describe how to set up a collector by hand.

To describe the right hand sides of the relations in a pc-presentation we use either words from a free group or **generator exponent lists**. The latter are lists of integers which represent a word in a free group; while a word in a free group is a product of generators and inverses, a generator exponent list is a list in which generator numbers and exponents alternate. Each factor $g_i^{e_i}$ in a word is represented by the entries i, j.

1 ► `FromTheLeftCollector(` *n* `)`

returns an empty data structure for a collector with n generators. No generator has a relative order and no conjugate relations are defined. Two generators for which no conjugate relations are defined commute. Therefore, the collector returned by this function can be used to define a free abelian group of rank n.

```
gap> ftl := FromTheLeftCollector( 4 );
<<from the left collector with 4 generators>>
gap> UpdatePolycyclicCollector( ftl );
gap> PcpGroupByCollector( ftl );
Pcp-group with orders [ 0, 0, 0, 0 ]
gap> IsAbelian(last);
true
```

2 ► `SetRelativeOrder(` *coll*`,` *i*`,` *ro* `)`

set the relative order in collector *coll* for generator i to *ro*. The parameter *coll* is a collector as returned by the function 2.2.1, i is a generator number, i.e. an integer in the range $1, \ldots, n$ where n is the number of generators of the collector and *ro* is a non-negative integer. If *ro* is 0, then the generator with number i has infinite order and no power relation can be specified. A previously defined power relation is deleted.

3 ► `SetPower(` *coll*`,` *i*`,` *rhs* `)`

set the right hand side of the power relation for generator i in collector *coll* to *rhs*. An attempt to set the right hand side for a generator without a relative order results in an error. Right hand sides are by default assumed to be trivial. The parameter *coll* is a collector, i is a generators number and *rhs* is a generators exponent list or an element from a free group.

4 ► `SetConjugate(` *coll*`,` *j*`,` *i*`,` *rhs* `)`

set the right hand side of the conjugate relation for the generators j and i. Negative generators numbers refer to inverses of the generators. Conjugate relations are by default assumed to be trivial The parameter *coll* is a collector, *i* is a generators number and *rhs* is a generators exponent list or an element from a free group.

5 ▶ `UpdatePolycyclicCollector(` *coll* `)`

completes the data structures of a collector. This is usually the last step in setting up a collector. Among the steps performed is the completion of the conjugate relations. For each non-trivial conjugate relation of a generator, the corresponding conjugate relation of the inverse generator is calculated.

6 ▶ `IsConfluent(` *coll* `)`

tests if the collector *coll* is confluent. The function return true or false accordingly.

The following example specifies a collector for the infinite dihedral group.

```
gap> ftl := FromTheLeftCollector( 2 );
<<from the left collector with 2 generators>>
gap> SetRelativeOrder( ftl, 1, 2 );
gap> SetConjugate( ftl, 2, 1, [2,-1] );
gap> UpdatePolycyclicCollector( ftl );
gap> IsConfluent( ftl );
true
gap>
```

2.3 Pcp-elements – elements of a pc-presented group

A **pcp-element** is an element of a group defined by a consistent pc-presentation given by a collector. Suppose that $g_1, \ldots, g_n$ are the defining generators of the collector. Recall that each element g in this group can be written uniquely as collected word $g_1^{e_1} \cdots g_n^{e_n}$ with $e_i \in \mathbb{Z}$ and $0 \leq e_i < r_i$ for $i \in I$. The integer vector $[e_1, \ldots, e_n]$ is called the **exponent vector** of g. The following functions can be used to define pcp-elements via their exponent vector or via an arbitrary generator exponent word as introduced in Section 2.2.

1 ▶ `PcpElementByExponentsNC(` *coll*`,` *exp* `)`
▶ `PcpElementByExponents(` *coll*`,` *exp* `)`

returns the pcp-element with exponent vector *exp*. The exponent vector is considered relative to the defining generators of the pc-presentation.

2 ▶ `PcpElementByGenExpListNC(` *coll*`,` *word* `)`
▶ `PcpElementByGenExpList(` *coll*`,` *word* `)`

returns the pcp-element with generators exponent list *word*. The generators exponent list is considered relative to the defining generators of the pc-presentation.

These functions return pcp-elements in the category `IsPcpElement`. Presently, the only representation implemented for this category is `IsPcpElementRep`. (This allows us to be a little sloppy right now. The basic set of operations for `IsPcpElement` has not been defined yet. This is going to happen in one of the next version, certainly as soon as the need for different representations arises.)

3 ▶ `IsPcpElement(` *obj* `)`

returns true if the object *obj* is a pcp-element.

4 ▶ `IsPcpElementRep(` *obj* `)`

returns true if the object *obj* is represented as a pcp-element.

2.4 Methods for pcp-elements

Now we can describe attributes and functions for pcp-elements. The four basic attributes of a pcp-element, `Collector`, `Exponents`, `GenExpList` and `NameTag` are computed at the creation of the pcp-element. All other attributes are determined at runtime.

Let g be a pcp-element and $g_1, \ldots, g_n$ a polycyclic generating sequence of the underlying pc-presented group. Let $C_1, \ldots, C_n$ be the polycyclic series defined by $g_1, \ldots, g_n$.

The **depth** of a non-trivial element g of a pcp-group (with respect to the defining generators) is the integer i such that $g \in C_i \setminus C_{i+1}$. The depth of the trivial element is defined to be $n+1$. If $g \neq 1$ has depth i and $g_i^{e_i} \cdots g_n^{e_n}$ is the collected word for g, then e_i is the **leading exponent** of g.

If g has depth i, then we call $r_i = [C_i : C_{i+1}]$ the **factor order** of g. If $r < \infty$, then the smallest positive integer l with $g^l \in C_{i+1}$ is the called **relative order** of g. If $r = \infty$, then the relative order of g is defined to be 0. The index e of $\langle g, C_{i+1} \rangle$ in C_i / C_{i+1} is called **relative index** of g. We have that $r = el$.

We call a pcp-element **normed**, if its leading exponent is equal to its relative index. For each pcp-element g there exists an integer e such that g^e is normed.

1 ► `Collector(` *g* `)`

the collector to which the pcp-element g belongs.

2 ► `Exponents(` *g* `)`

returns the exponent vector of the pcp-element g wrt the defining generating set of the underlying collector.

3 ► `GenExpList(` *g* `)`

returns the generators exponent list of the pcp-element g wrt the defining generating set of the underlying collector.

4 ► `NameTag(` *g* `)`

the name used for printing the pcp-element g. Printing is done by using the name tag and appending the generator number of g.

5 ► `Depth(` *g* `)`

returns the depth of the pcp-element g relative to the defining generators.

6 ► `LeadingExponent(` *g* `)`

returns the leading exponent of pcp-element g relative to the defining generators. If g is the identity element, the functions returns 'fail'

7 ► `RelativeOrder(` *g* `)`

returns the relative order of the pcp-element g with respect to the defining generators.

8 ► `RelativeIndex(` *g* `)`

returns the relative index of the pcp-element g with respect to the defining generators.

9 ► `FactorOrder(` *g* `)`

returns the factor order of the pcp-element g with respect to the defining generators.

10 ► `NormingExponent(` *g* `)`

returns a positive integer e such that the pcp-element g raised to the power of e is normed.

11 ► `NormedPcpElement(` *g* `)`

returns the normed element corresponding to the pcp-element g.

2.5 Pcp-groups - groups of pcp-elements

A **pcp-group** is a group consisting of pcp-elements such that all pcp-elements in the group share the same collector. Thus the group G defined by a polycyclic presentation and all its subgroups are pcp-groups.

1 ▶ `PcpGroupByCollectorNC(` *coll* `)`
▶ `PcpGroupByCollector(` *coll* `)`

returns a pcp-group build from the collector *coll*.

2 ▶ `Group(` *gens*`,` *id* `)`

returns the group generated by the pcp-elements *gens* with identity *id*.

3 ▶ `Subgroup(` G`,` *gens* `)`

returns a subgroup of the pcp-group G generated by the list *gens* of pcp-elements from G.

```
gap>  ftl := FromTheLeftCollector( 2 );;
gap>  SetRelativeOrder( ftl, 1, 2 );
gap>  SetConjugate( ftl, 2, 1, [2,-1] );
gap>  UpdatePolycyclicCollector( ftl );
gap>  G:= PcpGroupByCollectorNC( ftl );
Pcp-group with orders [ 2, 0 ]
gap> Subgroup( G, GeneratorsOfGroup(G){[2]} );
Pcp-group with orders [ 0 ]
```

2.6 Basis methods and functions for pcp-groups

Pcp-groups are groups in the GAP sense and hence all generic GAP methods for groups can be applied for pcp-groups. However, for a number of group theoretic questions GAP does not provide generic methods that can be applied to pcp-groups. For some of these questions there are functions provided in polycyclic.

In this section we describe some important basic functions which are available for pcp-groups. A number of higher-level functions are outlined in later sections and chapters.

Let U, V and N are subgroups of a pcp-group.

1 ▶ `U = V`

decides if U and V are equal as sets.

2 ▶ `Size(` U `)`

returns the size of U.

3 ▶ `Random(` U `)`

returns a random element of U.

4 ▶ `Index(` U`,` V `)`

returns the index of V in U if V is a subgroup of U. The function does not check if V is a subgroup of U and if it is not, the result is not meaningful.

5 ▶ $g \in U$

checks if g is an element of U.

6 ▶ `Elements(` U `)`

returns a list containing all elements of U. This functions will not terminate if U is infinite.

7 ▶ `IsSubgroup(` *U*`,` *V* `)`

tests if V is a subgroup of U.

8 ▶ `IsNormal(` *U*`,` *V* `)`

tests if V is normal in U.

9 ▶ `ClosureGroup(` *U*`,` *V* `)`

returns the group generated by U and V.

10 ▶ `NormalClosure(` *U*`,` *V* `)`

returns the normal closure of V under action of U.

11 ▶ `U / N`

return the factor group of U modulo N. Clearly, N must be normal in U.

12 ▶ `HirschLength(` *U* `)`

returns the Hirsch length of U.

13 ▶ `CommutatorSubgroup(` *U*`,` *V* `)`

returns the group generated by all commutators $[u, v]$ with u in U and v in V.

14 ▶ `PRump(` *U*`,` *p* `)`

returns the subgroup $U'U^p$ of U where p is a prime number.

15 ▶ `IsNilpotentGroup(` *U* `)`

checks whether U is nilpotent.

16 ▶ `IsAbelian(` *U* `)`

checks whether U is abelian.

17 ▶ `IsElementaryAbelian(` *U* `)`

checks whether U is elementary abelian.

18 ▶ `IsFreeAbelian(` *U* `)`

checks whether U is free abelian.

2.7 Igs - induced generating sequences for subgroups

A subgroup of a pcp-group G can be defined by a set of generators as described in Section 2.5. However, many computations with a subgroup U need an **induced generating sequence** or **igs** of U. An igs is a sequence of generator of U whose list of exponent vectors form a matrix in upper triangular form. Note that there may exist many igs of U. The first one calculated for U is stored as an attribute.

An induced generating sequence of a subgroup of a pcp-group G is a list of elements of G. An igs is called **normed**, if each element in the list is normed. Moreover, it is **canonical**, if the exponent vector matrix is in Hermite Normal Form. The following functions can be used to compute induced generating sequence for a given subgroup U of G.

1 ▶ `Igs(` *U* `)`
▶ `Igs(` *gens* `)`
▶ `IgsParallel(` *gens*, *gens2* `)`

returns an induced generating sequence of the subgroup U of a pcp-group. In the second for the subgroup is given via a generating set *gens*. The third form computes an igs for the subgroup generated by *gens* carrying *gens2* through as shadows.

2 ▶ `Ngs(` *U* `)`
▶ `Ngs(` *igs* `)`

returns a normed induced generating sequence of the subgroup U of a pcp-group. The second form takes an igs as input and norms it.

3 ▶ `Cgs(` *U* `)`
▶ `Cgs(` *igs* `)`
▶ `CgsParallel(` *gens*, *gens2* `)`

returns a canonical generating sequence of the subgroup U of a pcp-group. In the second form the function takes an igs as input and returns a canonical generating sequence. The third version takes a generating set and computes a canonical generating sequence carrying *gens2* through as shadows.

For a large number of methods for pcp-groups U we will first of all determine an *igs* for U. Hence it might speed up computations, if a known *igs* for a group U is set *a priori*. The following functions can be used for this purpose.

4 ▶ `SubgroupByIgs(` *G*, *igs* `)`
▶ `SubgroupByIgs(` *G*, *igs*, *gens* `)`

returns the subgroup of the pcp-group G generated by the elements of the induced generating sequence *igs*. Note that *igs* must be an induced generating sequence of the subgroup generated by the elements of the *igs*. In the second form *igs* is a igs for a subgroup and *gens* are some generators. The function returns the subgroup generated by *igs* and *gens*.

5 ▶ `AddToIgs(` *igs*, *gens* `)`
▶ `AddToIgsParallel(` *igs*, *gens*, *igs2*, *gens2* `)`
▶ `AddIgsToIgs(` *igs*, *igs2* `)`

sifts the elements in the list *gens* into *igs*. The second version has the same functionality and carries shadows. The third version is available for efficiency reasons and assumes that the second list *igs2* is not only a generating set, but an igs.

2.8 Pcps – polycyclic presentation sequences for subfactors

A subfactor of a pcp-group G is again a polycyclic group for which a polycyclic presentation can computed. However, to compute a polycyclic presentation for a given subfactor can be time-consuming. Hence we introduce **polycyclic presentation sequences** or **Pcp** to compute more efficiently with subfactors. (Note that a subgroup is also a subfactor and thus can be handled by a pcp)

A pcp for a pcp-group U or a subfactor U/N can be created with one of the following functions.

1 ▶ `Pcp(` *U* `)`
▶ `Pcp(` *U*, *N* `)`
▶ `Pcp(` *U*, *"snf"* `)`
▶ `Pcp(` *U*, *N*, *"snf"* `)`

returns a polycyclic presentation sequence for the subgroup U or the quotient group U modulo N. If the parameter `"snf"` is present, the function can only be applied to an abelian subgroup U or abelian subfactor

U/N. The pcp returned will correspond to a decomposition of the abelian group into a direct product of cyclic groups.

A pcp is a component object which behaves similar to a list representing an igs of the subfactor in question. The basic functions to obtain the stored values of this component object are as follows. Let *pcp* be a pcp for a subfactor U/N of the defining pcp-group G.

2 ► `GeneratorsOfPcp(` *pcp* `)`

this returns a list of elements of U corresponding to an igs of U/N.

3 ► `pcp[i]`

returns the i-th element of *pcp*.

4 ► `Length(` *pcp* `)`

returns the number of generators in *pcp*.

5 ► `RelativeOrdersOfPcp(` *pcp* `)`

the relative orders of the igs in *U/N*.

6 ► `DenominatorOfPcp(` *pcp* `)`

returns an igs of N.

7 ► `NumeratorOfPcp(` *pcp* `)`

returns an igs of U.

8 ► `GroupOfPcp(` *pcp* `)`

returns U.

9 ► `OneOfPcp(` *pcp* `)`

returns the identity element of G.

The two main features of pcp are the possibility to compute exponent vectors wrt to a pcp and to compute the group defined by the corresponding igs of U/N.

10 ► `ExponentsByPcp(` *pcp*, *g* `)`

returns the exponent vector of g with respect to the generators of *pcp*. This is the exponent vector of gN with respect to the igs of *U/N*.

11 ► `PcpGroupByPcp(` *pcp* `)`

returns the group whose defining generators correspond to the generators of *pcp*.

```
gap>  G := DihedralPcpGroup(0);
Pcp-group with orders [ 2, 0 ]
gap>  pcp := Pcp(G);
Pcp [ g1, g2 ] with orders [ 2, 0 ]
gap>  pcp[1];
g1
gap>  Length(pcp);
2
gap>  RelativeOrdersOfPcp(pcp);
[ 2, 0 ]
```

```
gap>  DenominatorOfPcp(pcp);
[  ]
gap>  NumeratorOfPcp(pcp);
[ g1, g2 ]
gap>  GroupOfPcp(pcp);
Pcp-group with orders [ 2, 0 ]
gap> OneOfPcp(pcp);
identity

gap> G := PcpExamples[5];
Pcp-group with orders [ 2, 0, 0, 0 ]
gap> D := DerivedSubgroup( G );
Pcp-group with orders [ 0, 0, 0 ]
gap>  GeneratorsOfGroup( G );
[ g1, g2, g3, g4 ]
gap>  GeneratorsOfGroup( D );
[ g2^-2, g3^-2, g4^2 ]

# an ordinary pcp for G / D
gap> pcp1 := Pcp( G, D );
Pcp [ g1, g2, g3, g4 ] with orders [ 2, 2, 2, 2 ]

# a pcp for G/D in independent generators
gap>  pcp2 := Pcp( G, D, "snf" );
Pcp [ g2, g3, g1 ] with orders [ 2, 2, 4 ]

gap>  g := Random( G );
g1*g2^-4*g3*g4^2

# compute the exponent vector of g in G/D wrt pcp1
gap> ExponentsByPcp( pcp1, g );
[ 1, 0, 1, 0 ]

# compute the exponent vector of g in G/D wrt pcp2
gap>  ExponentsByPcp( pcp2, g );
[ 0, 1, 1 ]
```

2.9 Factor groups of pcp-groups

Pcp's for subfactors of pcp-groups have already been described above. These are usually used within algorithms to compute with pcp-groups. However, it is also possible to explicitly construct factor groups and their corresponding natural homomorphisms.

1 ▶ `NaturalHomomorphism(` *G*`,` *N* `)`

returns the natural homomorphism $G \to G/N$. Its image is the factor group G/N.

2 ▶ `G/N`
▶ `FactorGroup(` *G*`,` *N* `)`

returns the desired factor as pcp-group without giving the explicit homomorphism. This function is just a wrapper for `PcpGroupByPcp( Pcp( G, N ) )`.

2.10 Homomorphisms for pcp-groups

IsPcpGHBI is a representation used to define group homomorphisms by generators and images from a pcp-group into another pcp-group. Such homomorphisms can be compared and multiplied. Moreover, we provide the following functions.

IsToPcpGHBI is a representation used to define group homomorphisms by generators and images from an arbitrary group into a pcp-group. Here, only very restricted functionality is provided. This is mostly used for converting other groups to pcp-groups.

1 ► `GroupHomomorphismByImages(` *G*`,` *H*`,` *gens*`,` *imgs* `)`

returns the homomorphism from the (pcp-) group G to the pcp-group H mapping the generators of G in the list *gens* to the corresponding images in the list *imgs* of elements of H.

2 ► `Kernel(` *hom* `)`

returns the kernel of the homomorphism *hom* from a pcp-group to a pcp-group.

3 ► `Image(` *hom* `)`
► `Image(` *hom*`,` *U* `)`
► `Image(` *hom*`,` *g* `)`

returns the image of the whole group, of U and of g, respectively, under the homomorphism *hom*.

4 ► `PreImage(` *hom*`,` *U* `)`

returns the complete preimage of the subgroup U under the homomorphism *home*. If the domain of *hom* is not a pcp-group, then this function only works properly if *hom* is injective.

5 ► `PreImagesRepresentative(` *hom*`,` *g* `)`

returns a preimage of the element g under the homomorphism *hom*.

6 ► `IsInjective(` *hom* `)`

checks if the homomorphism *hom* is injective.

2.11 Changing the defining pc-presentation

The following functions should actually return isomorphisms.

1 ► `RefinedPcpGroup(` *G* `)`

returns a new pcp-group isomorphic to G whose defining polycyclic presentation is refined; that is, the corresponding polycyclic series has prime or infinite factors only. If H is the new group, then $H! \cdot bijection$ is the isomorphism $G \rightarrow H$.

2 ► `PcpGroupBySeries(` *ser* `)`

returns a new pcp-group isomorphic to the first subgroup G of the given series *ser* such that its defining pcp refines the given series. The series must be subnormal and $H! \cdot bijection$ is the isomorphism $G \rightarrow H$.

```
gap> G := DihedralPcpGroup(0);
Pcp-group with orders [ 2, 0 ]
gap>  U := Subgroup( G, [Pcp(G)[2]^1440]);
Pcp-group with orders [ 0 ]
gap>  F := G/U;
Pcp-group with orders [ 2, 1440 ]
gap> RefinedPcpGroup(F);
Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 3, 3, 5 ]
```

```
gap> ser := [G, U, TrivialSubgroup(G)];
[ Pcp-group with orders [ 2, 0 ],
  Pcp-group with orders [ 0 ],
  Pcp-group with orders [  ] ]
gap>  PcpGroupBySeries(ser);
Pcp-group with orders [ 2, 1440, 0 ]
```

2.12 Converting to pc-presentations

1 ► `IsomorphismPcpGroup(` *G* `)`

returns a pcp-group isomorphic to G if G is a pc-group or a soluble permutation group.

2 ► `IsomorphismPcGroup(` *G* `)`

returns a pc-group if G is a finite pcp-group.

2.13 Some generic pcp-groups

There are the following generic pcp-groups available.

1 ► `AbelianPcpGroup(` *n*`,` *rels* `)`

constructs the abelian group on n generators such that generator i has order $rels[i]$. If this order is infinite, then $rels[i]$ should be either unbound or 0.

2 ► `DihedralPcpGroup(` *n* `)`

constructs the dihedral group of order n. If n is not an even integer, then 'fail' is returned. If n is not an integer, then the infinite dihedral group is returned.

3 ► `UnitriangularPcpGroup(` *n* `)`

returns a pcp-group isomorphic to the group of upper triangular matrices in $GL(n, Z)$.

4 ► `SubgroupUnitriangularPcpGroup(` *mats* `)`

returns the subgroup generated by the upper triangular matrices in *mats* as a pcp-group.

2.14 Some example pcp-groups

1 ► `ExampleOfMetabelianGroup(` *a*`,` *k* `)`

returns an example of a metabelian group.

2 ► `PcpExamples`

is a list of pcp-groups which can serve as first examples to try some of the functions in this package.

3 ► `EddiesExamples`

a list with more examples provided by Eddie Lo.

4 ► `NqExamples`

a list of nilpotent pcp-groups generated by the Nilpotent Quotient Algorithm.

3 Methods for pcp-groups

This is a description of some higher level functions of the polycyclic share package of GAP 4. Throughout this chapter we let G be a pc-presented group and we consider algorithms for subgroups U and V of G.

3.1 Orbit stabilizer methods for pcp-groups

The following two functions are implementations of the polycyclic group algorithm to compute orbits and stabilizers, if the orbits are finite.

1 ► `PcpOrbitStabilizer(` *point*, *gens*, *acts*, *oper* `)`
► `PcpOrbitsStabilizers(` *points*, *gens*, *acts*, *oper* `)`

The input *gens* can an igs or a pcp of a pcp-group U. The elements in the list *gens* act as the elements in the list *acts* and the function *oper* on the given points. The first function returns a record containing 'orbit' and 'stab'. The latter is an induced igs of the stabilizer. The second function returns a list of records, each record contains 'repr' and 'stab'.

Both of these functions run forever on infinite orbits.

```
gap> G := DihedralPcpGroup( 0 );
Pcp-group with orders [ 2, 0 ]
gap> mats := [ [[-1,0],[0,1]], [[1,1],[0,1]] ];;
gap> pcp := Pcp(G);
Pcp [ g1, g2 ] with orders [ 2, 0 ]
gap> PcpOrbitStabilizer( [0,1], pcp, mats, OnRight );
rec( orbit := [ [ 0, 1 ] ],
     stab := [ g1, g2 ],
     word := [ [ [ 1, 1 ] ], [ [ 2, 1 ] ] ] )
```

3.2 Subgroup series in pcp-groups

Many algorithm for pcp-groups work by induction using some series through the group. In this section we prove a number of useful series for pcp-groups. Note that an **efa series** is a normal series with elementary or free abelian factors. See [Eic00] for an outline on the algorithms of a number of the available series.

1 ► `PcpSeries(` *U* `)`

returns the polycyclic series of U defined by an igs of U.

2 ► `EfaSeries(` *U* `)`

returns a normal series of U with elementary or free abelian factors.

3 ► `DerivedSeries(` *U* `)`

the derived series of U.

4 ► `RefinedDerivedSeries(` *U* `)`

the derived series of U refined to an efa series such that in each abelian factor of the derived series the free abelian factor is at the top.

5 ► `RefinedDerivedSeriesDown(` *U* `)`

the derived series of U refined to an efa series such that in each abelian factor of the derived series the free abelian factor is at the bottom.

6 ► `LowerCentralSeries(` *U* `)`

the lower central series of U. If U does not have a largest nilpotent quotient group, then this function may not terminate.

7 ► `TorsionByPolyEFSeries(` *U* `)`

returns an efa series of U such that all torsion-free factors are at the top and all finite factors are at the bottom. Such a series might not exist for U and in this case the function returns fail.

```
gap> G := PcpExamples[5];
Pcp-group with orders [ 2, 0, 0, 0 ]
gap> Igs(G);
[ g1, g2, g3, g4 ]

gap> PcpSeries(G);
[ Pcp-group with orders [ 2, 0, 0, 0 ],
  Pcp-group with orders [ 0, 0, 0 ],
  Pcp-group with orders [ 0, 0 ],
  Pcp-group with orders [ 0 ],
  Pcp-group with orders [  ] ]

gap> List( PcpSeries(G), Igs );
[ [ g1, g2, g3, g4 ], [ g2, g3, g4 ], [ g3, g4 ], [ g4 ], [  ] ]
```

Algorithms for pcp-groups often use an efa series of G and work down over the factors of this series. Usually, pcp's of the factors are more useful than the actual factors. Hence we provide the following.

8 ► `PcpsBySeries(` *ser* `)`
► `PcpsBySeries(` *ser*, *"snf"* `)`

returns a list of pcp's corresponding to the factors of the series. If the second argument is present, then each pcp corresponds to a decomposition of the abelian groups into direct factors.

9 ► `PcpsOfEfaSeries(` *U* `)`

return a list of pcp's corresponding to an efa series of U.

```
gap> G := PcpExamples[5];
Pcp-group with orders [ 2, 0, 0, 0 ]
```

```
gap> PcpsBySeries( DerivedSeries(G));
[ Pcp [ g1, g2, g3, g4 ] with orders [ 2, 2, 2, 2 ],
  Pcp [ g2^-2, g3^-2, g4^2 ] with orders [ 0, 0, 4 ],
  Pcp [ g4^8 ] with orders [ 0 ] ]
gap> PcpsBySeries( RefinedDerivedSeries(G));
[ Pcp [ g1, g2, g3 ] with orders [ 2, 2, 2 ],
  Pcp [ g4 ] with orders [ 2 ],
  Pcp [ g2^2, g3^2 ] with orders [ 0, 0 ],
  Pcp [ g4^2 ] with orders [ 2 ],
  Pcp [ g4^4 ] with orders [ 2 ],
  Pcp [ g4^8 ] with orders [ 0 ] ]

gap> PcpsBySeries( RefinedDerivedSeries(G), "snf");
[ Pcp [ g1, g2, g3 ] with orders [ 2, 2, 2 ],
  Pcp [ g4 ] with orders [ 2 ],
  Pcp [ g2^2, g3^2 ] with orders [ 0, 0 ],
  Pcp [ g4^2 ] with orders [ 2 ],
  Pcp [ g4^4 ] with orders [ 2 ],
  Pcp [ g4^8 ] with orders [ 0 ] ]

gap>  PcpsOfEfaSeries( G );
[ Pcp [ g1 ] with orders [ 2 ],
  Pcp [ g2 ] with orders [ 0 ],
  Pcp [ g3 ] with orders [ 0 ],
  Pcp [ g4 ] with orders [ 0 ] ]
```

3.3 Random methods and functions available for pcp-groups

Below we introduce a function which computes orbit and stabilizer using a random method. This function always terminates, also if the orbit is infinite. But the returned orbit or stabilizer might be incomplete. This function is used in the random methods to compute normalizers and centralizers.

1 ► `RandomOrbitStabilizerPcpGroup(` *U*, *point*, *oper* `)`

2 ► `RandomCentralizerPcpGroup(` *U*, *g* `)`

3 ► `RandomCentralizerPcpGroup(` *U*, *V* `)`

4 ► `RandomNormalizerPcpGroup(` *U*, *V* `)`

```
gap> G := DihedralPcpGroup(0);
Pcp-group with orders [ 2, 0 ]
gap> mats := [[[-1, 0],[0,1]], [[1,1],[0,1]]];
[ [ [ -1, 0 ], [ 0, 1 ] ], [ [ 1, 1 ], [ 0, 1 ] ] ]
gap> pcp := Pcp(G);
Pcp [ g1, g2 ] with orders [ 2, 0 ]

gap> RandomPcpOrbitStabilizer( [1,0], pcp, mats, OnRight ).stab;
#I  Orbit longer than limit: exiting.
[  ]
```

```
gap> g := Igs(G)[1];
g1
gap> RandomCentralizerPcpGroup( G, g );
#I  Stabilizer not increasing: exiting.
Pcp-group with orders [ 2 ]
gap> Igs(last);
[ g1 ]
```

3.4 Intersection of subgroups

Currently, only intersections of subgroups $U, N \leq G$ can be computed if N is normalising U. See [Sim94] for an outline of the algorithm.

1 ▶ `NormalIntersection(` *U* `,` *N* `)`

3.5 Finite subgroups

There are various finite subgroups of interest in polycyclic groups. See [Eic00] for a description of the algorithms underlying the functions in this section.

1 ▶ `TorsionSubgroup(` *U* `)`

If the set of elements of finite order forms a subgroup, then we call it the **torsion subgroup**. This function determines the torsion subgroup of U, if it exists, and returns fail otherwise. Note that a torsion subgroup does always exist if U is nilpotent.

2 ▶ `NormalTorsionSubgroup(` *U* `)`

Each polycyclic groups has a unique largest finite normal subgroup. This function computes it for U.

3 ▶ `IsTorsionFree(` *U* `)`

This function checks if U is torsion free. It returns true or false.

4 ▶ `FiniteSubgroupClasses(` *U* `)`

There exist only finitely many conjugacy classes of finite subgroups in a polycyclic group U and this function can be used to compute them. The algorithm underlying this function proceeds by working down a normal series of U with elementary or free abelian factors. The following function can be used to give the algorithm a specific series.

5 ▶ `FiniteSubgroupClassesBySeries(` *U* `,` *pcps* `)`

```
gap> G := NqExamples[2];
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 4, 0 ]
gap> TorsionSubgroup(G);
Pcp-group with orders [ 5, 2 ]
gap> NormalTorsionSubgroup(G);
Pcp-group with orders [ 5, 2 ]
gap> IsTorsionFree(G);
false
gap> FiniteSubgroupClasses(G);
[ Pcp-group with orders [ 5, 2 ]^G,
  Pcp-group with orders [ 2 ]^G,
  Pcp-group with orders [ 5 ]^G,
  Pcp-group with orders [  ]^G ]
```

```
gap> G := DihedralPcpGroup( 0 );
Pcp-group with orders [ 2, 0 ]
gap> TorsionSubgroup(G);
fail
gap> NormalTorsionSubgroup(G);
Pcp-group with orders [  ]
gap> IsTorsionFree(G);
false
gap> FiniteSubgroupClasses(G);
[ Pcp-group with orders [ 2 ]^G,
  Pcp-group with orders [ 2 ]^G,
  Pcp-group with orders [  ]^G ]
```

3.6 Subgroups of finite index

Here we outline functions to determine various types of subgroups of finite index in polycyclic groups. Again, see [Eic00] for a description of the algorithms underlying the functions in this section.

1 ▶ `MaximalSubgroupClassesPIndex(` *U* `,` *p* `)`

Each maximal subgroup of a polycyclic group U has p-power index for some prime p. This function can be used to determine conjugacy class representatives of all maximal subgroups of p-power index for a given prime p.

2 ▶ `LowIndexSubgroupClasses(` *U* `,` *n* `)`

There are only finitely many subgroups of a given index in a polycyclic group U. This function computes conjugacy classes of all subgroups of index n in U.

3 ▶ `LowIndexNormals(` *U* `,` *n* `)`

This function computes the normal subgroups of index n in U.

4 ▶ `NilpotentByAbelianNormalSubgroup(` *U* `)`

This function returns a normal subgroup N of finite index in U such that N is nilpotent-by-abelian. Note that such a subgroup exists in every polycyclic group. Note that this function is not very efficient.

```
gap> G := PcpExamples[2];
Pcp-group with orders [ 0, 0, 0, 0, 0, 0 ]

gap> MaximalSubgroupClassesPIndex( G, 61 );;
gap> max := List( last, Representative );;
gap> List( max, x -> Index( G, x ) );
[ 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61,
  61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61,
  61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61,
  61, 61, 61, 61, 61, 61, 226981 ]

gap> LowIndexSubgroupClasses( G, 61 );;
gap> low := List( last, Representative );;
gap> List( low, x -> Index( G, x ) );
[ 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61,
  61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61,
  61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61, 61,
  61, 61, 61, 61, 61, 61 ]
```

3.7 Functions for nilpotent groups

1 ▶ `MinimalGeneratingSet(` *U* `)`

2 ▶ `Centre(` *U* `)`

3 ▶ `UpperCentralSeries(` *U* `)`

```
gap> G := NqExamples[1];
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 4, 0, 5, 5, 4, 0, 6,
  5, 5, 4, 0, 10, 6 ]
gap> IsNilpotent(G);
true

gap> PcpsBySeries( LowerCentralSeries(G));
[ Pcp [ g1, g2 ] with orders [ 0, 0 ],
  Pcp [ g3 ] with orders [ 0 ],
  Pcp [ g4 ] with orders [ 0 ],
  Pcp [ g5 ] with orders [ 0 ],
  Pcp [ g6, g7 ] with orders [ 0, 0 ],
  Pcp [ g8 ] with orders [ 0 ],
  Pcp [ g9, g10 ] with orders [ 0, 0 ],
  Pcp [ g11, g12, g13 ] with orders [ 5, 4, 0 ],
  Pcp [ g14, g15, g16, g17, g18 ] with orders [ 5, 5, 4, 0, 6 ],
  Pcp [ g19, g20, g21, g22, g23, g24 ] with orders [ 5, 5, 4, 0, 10, 6 ] ]

gap> PcpsBySeries( UpperCentralSeries(G));
[ Pcp [ g1, g2 ] with orders [ 0, 0 ],
  Pcp [ g3 ] with orders [ 0 ],
  Pcp [ g4 ] with orders [ 0 ],
  Pcp [ g5 ] with orders [ 0 ],
  Pcp [ g6, g7 ] with orders [ 0, 0 ],
  Pcp [ g8 ] with orders [ 0 ],
  Pcp [ g9, g10 ] with orders [ 0, 0 ],
  Pcp [ g11, g12, g13 ] with orders [ 5, 4, 0 ],
  Pcp [ g14, g15, g16, g17, g18 ] with orders [ 5, 5, 4, 0, 6 ],
  Pcp [ g19, g20, g21, g22, g23, g24 ] with orders [ 5, 5, 4, 0, 10, 6 ] ]

gap> MinimalGeneratingSet(G);
[ g1, g2 ]
```

4 Cohomology for pcp-groups

The share package polycyclic provides methods to compute the first and second cohomology group for a pcp-group U and a finite dimensional $\mathbb{Z}U$ or FU module A where F is a finite field. The algorithm for determining the first cohomology group are outlined in [Eic00].

4.1 Cohomology records

First we need some methods to create a module for a pcp-group U. This module can either be defined `externally` via a matrix operation of U or `internally` using an elementary or free abelian normal subfactor.

1 ► `CRRecordByMats(` U`,` *mats* `)`

creates an external module. The input *mats* is a list of integer or finite field matrices. This list corresponds to *Pcp(U)* and defines the matrix action of the elements of *Pcp(U)*.

2 ► `CRRecordBySubgroup(` U`,` A `)`
► `CRRecordByPcp(` U`,` *pcp* `)`

creates an internal module. The input A or *pcp* defines an elementary or free abelian normal subgroup or subfactor of U.

The returned cohomology record C contains the following entries:

factor
: a pcp of the acting group. If the module is external, then this is *Pcp(U)*. If the module is internal, then this is *Pcp(U, A)* or *Pcp(U, GroupOfPcp(pcp))*.

mats, *invs* and *one*
: the matrix action of *factor* with acting matrices, their inverses and the identity matrix.

dim and *char*
: the dimension and characteristic of the matrices.

relators and *enumrels*
: the relators of *factor* as words and an enumeration list for them.

central
: is true, if the matrices *mats* are all trivial. This is used locally for efficiency reasons.

And additionally, if C defines an internal module, then it contains:

group
: the original group U.

normal
: this is either *Pcp(A)* or the input *pcp*.

extension
: information on the extension of A by U/A.

4.2 Cohomology groups

Let U be a pcp-group and A a free or elementary abelian pcp-group and a U-module. By $Z^i(U, A)$ be denote the group of i-th cocycles and by $B^i(U, A)$ the i-th coboundaries. The factor $Z^i(U, A)/B^i(U, A)$ is the i-th cohomology group. Since A is elementary or free abelian, the groups $Z^i(U, A)$ and $B^i(U, A)$ are elementary or free abelian groups as well.

The polycyclic share package provides methods to compute first and second cohomology group for a polycyclic group U. We write all involved groups additively and we use an explicit description by bases for them. Let C be the cohomology record corresponding to U and A.

Let $f_1, \ldots, f_n$ be the elements in the entry *factor* of the cohomology record C. Then we use the following embedding of the first cocycle group to describe 1-cocycles and 1-coboundaries: $Z^1(U, A) \to A^n : \delta \mapsto (\delta(f_1), \ldots, \delta(f_n))$

For the second cohomology group we recall that each element of $Z^2(U, A)$ defines an extension H of A by U. Thus there is a pc-presentation of H extending the pc-presentation of U given by the record C. The extended presentation is defined by tails in A; that is, each relator in the record entry *relators* is extended by an element of A. The concatenation of these tails yields a vector in A^l where l is the length of the record entry *relators* of C. We use these tail vectors to describe $Z^2(U, A)$ and $B^2(U, A)$. Note that this description is dependent on the chosen presentation in C. However, the factor $Z^2(U, A)/B^2(U, A)$ is independent of the chosen presentation.

The following functions are available to compute explicitly the first and second cohomology group as described above.

1 ► `OneCoboundariesCR(` C `)`

2 ► `OneCocyclesCR(` C `)`

3 ► `TwoCoboundariesCR(` C `)`

4 ► `TwoCocyclesCR(` C `)`

5 ► `OneCohomologyCR(` C `)`

6 ► `TwoCohomologyCR(` C `)`

The first 4 functions return bases of the corresponding group. The last 2 functions need to describe a factor of additive abelian groups. They return the following descriptions for these factors.

gcc
: the basis of the cocycles of C.

gcb
: the basis of the coboundaries of C.

factor
: a description of the factor of cocycles by coboundaries. Usually, it would be most convenient to use additive mappings here. However, these are not available in case that A is free abelian and thus we use a description of this additive map as record. This record contains

gens
: a base for the image.

rels
: relative orders for the image.

imgs
: the images for the elements in *gcc*.

prei
: preimages for the elements in *gens*.

denom
: the kernel of the map; that is, another basis for *gcb*.

4.3 Extended 1-cohomology

In some cases more information on the first cohomology group is of interest. In particular, if we have an internal module given and we want to compute the complements using the first cohomology group, then we need additional information. This extended version of first cohomology is obtained by the following functions.

1 ▶ `OneCoboundariesEX(` *C* `)`

returns a record consisting of the entries

`basis`
: a basis for $B^1(U,A) \leq A^n$.

`transf`
: There is a derivation mapping from A to *B^1(U,A)*. This mapping is described here as transformation from A to *basis*.

`fixpts`
: the fixed points of A. This is also the kernel of the derivation mapping.

2 ▶ `OneCocyclesEX(` *C* `)`

returns a record consisting of the entries

`basis`
: a basis for $Z^1(U,A) \leq A^n$.

`transl`
: a special solution. This is only of interest in case that C is an internal module and in this case it gives the translation vector in A^n used to obtain complements corresponding to the elements in *basis*. If C is not an internal module, then this vector is always the zero vector.

3 ▶ `OneCohomologyEX(` *C* `)`

returns the combined information on the first cohomology group.

4.4 Extensions and Complements

The natural applications of first and second cohomology group is the determination of extensions and complements. Let C be a cohomology record.

1 ▶ `ComplementCR(` C, c `)`

returns the complement corresponding to the 1-cocycle c. In the case that C is an external module, we construct the split extension of U with A first and then determine the complement. In the case that C is an internal module, the vector c must be an element of the affine space corresponding to the complements as described by `OneCocyclesEX`.

2 ▶ `ComplementsCR(` C `)`

returns all complements using the correspondence to $Z^1(U, A)$. Further, this function returns fail, if $Z^1(U, A)$ is infinite.

3 ▶ `ComplementClassesCR(` C `)`

returns complement classes using the correspondence to $H^1(U, A)$. Further, this function returns fail, if $H^1(U, A)$ is infinite.

4 ▶ `ComplementClassesEfaPcps(` U, N, *pcps* `)`

Let N be a normal subgroup of U. This function returns the complement classes to N in U. The classes are computed by iteration over the U-invariant efa series of N described by *pcps*. If at some stage in this iteration infinitely many complements are discovered, then the function returns fail. (Even though there might be only finitely many conjugacy classes of complements to N in U.)

5 ▶ `ComplementClasses(` [V,] U, N `)`

Let N and U be normal subgroups of V with $N \leq U \leq V$. This function attempts to compute the V-conjugacy classes of complements to N in U. The algorithm proceeds by iteration over a V-invariant efa series of N. If at some stage in this iteration infinitely many complements are discovered, then the algorithm returns fail.

6 ▶ `ExtensionCR(` C, c `)`

returns the extension corresponding to the 2-cocycle c.

7 ▶ `ExtensionsCR(` C `)`

returns all extensions using the correspondence to $Z^2(U, A)$. Further, this function returns fail, if $Z^2(U, A)$ is infinite.

8 ▶ `ExtensionClassesCR(` C `)`

returns extension classes using the correspondence to $H^2(U, A)$. Further, this function returns fail, if $H^2(U, A)$ is infinite.

9 ▶ `SplitExtensionPcpGroup(` U, *mats* `)`

returns the split extension of U by the U-module described by *mats*.

Bibliography

[BCRS91] G. Baumslag, F.B. Cannonito, D.J.S. Robinson, and D. Segal. The algorithmic theory of polycyclic-by-finite groups. *J. Alg.*, 142:118 − 149, 1991.

[Eic00] B. Eick. Computing with infinite polycyclic groups. In *Groups and Computation III*, Amer. Math. Soc. DIMACS Series. (DIMACS, 1999), 2000.

[Hir38a] K.A. Hirsch. On infinite soluble groups (I). *Proc. London Math. Soc.*, 44(2):53–60, 1938.

[Hir38b] K.A. Hirsch. On infinite soluble groups (II). *Proc. London Math. Soc.*, 44(2):336–414, 1938.

[Hir46] K.A. Hirsch. On infinite soluble groups (III). *J. London Math. Soc.*, 49(2):184–94, 1946.

[Hir52] K.A. Hirsch. On infinite soluble groups (IV). *J. London Math. Soc.*, 27:81–85, 1952.

[Hir54] K.A. Hirsch. On infinite soluble groups (V). *J. London Math. Soc.*, 29:250–251, 1954.

[LGS90] C[harles] R. Leedham-Green and L[eonard] H. Soicher. Collection from the left and other strategies. *J. Symbolic Comput.*, 9(5 & 6):665–675, 1990.

[Rob82] D.J. Robinson. *A Course in the Theory of Groups*, volume 80 of *Graduate Texts in Math.* Springer-Verlag, New York, Heidelberg, Berlin, 1982.

[Seg83] D. Segal. *Polycyclic Groups*. Cambridge University Press, Cambridge, 1983.

[Seg90] D. Segal. Decidable properties of polycyclic groups. *Proc. London Math. Soc.* (3), 61:497–528, 1990.

[Sim94] C. C. Sims. *Computation with Finitely Presented Groups*. Cambridge University Press, 1994.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "PermutationCharacter" comes before "permutation group".