Polycyclic

Computation with polycyclic groups

A GAP 4 package

by

Bettina Eick
Institut für Geometrie
TU Braunschweig

and

Werner Nickel
Fachbereich Mathematik
TU Darmstadt

Contents

1 Preface

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 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 the Chapter 2. 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 GAP 4 package polycyclic 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. This is described in the Section on polycyclic groups in the reference manual.

2 Introduction to polycyclic presentations

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 rise to a **power-conjugate (pc-) presentation** for G with the conjugate relations

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

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

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 GAP 4 package polycyclic 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 3 for a routine that checks the consistency of a pc-presentation.

A pc-presentation can be interpreted as a **rewriting system** in the following way. One needs need to add a new generator G_i for each generator g_i together with the relations $g_i G_i = 1$ and $G_i g_i = 1$. Any occurrence in a relation of an inverse generator g_i^{-1} is replaced by G_i. In this way one obtains a monoid presentation for the group G. With respect to a particular ordering on the set of monoid words in the generators $g_1, \ldots g_n, G_1, \ldots G_n$, the **wreath product ordering**, this monoid presentation is a rewriting system. If the pc-presentation is consistent, the rewriting system is confluent.

In this package we do not address this aspect of pc-presentations because it is of little relevance for the algorithms implemented here. For the definition of rewriting systems and confluence in this context as well as further details on the connections between pc-presentations and rewriting systems we recommend the book [Sim94].

3 Collectors

Let G be a group defined by a pc-presentation as described in the Chapter 2.

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 exist a number of different strategies for collecting a given word to collected form. The strategies implemented in this package are **collection from the left** as described by [LGS90] and [Sim94] and **combinatorial collection from the left** by [VL90]. In addition, the package provides access to Hall polynomials computed by Deep Thought for the multiplication in a nilpotent group, see [Mer97] and [LGS98].

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**.

To describe the right hand sides of the relations in a pc-presentation we use **generator exponent lists**; the word $g_{i_1}^{e_1} g_{i_2}^{e_2} \ldots g_{i_k}^{e_k}$ is represented by the generator exponent list $[i_1, e_1, i_2, e_2, \ldots, i_k, e_k]$.

3.1 Constructing a Collector

A collector for a group given by a pc-presentation starts by setting up an empty data structure for the collector. Then the relative orders, the power relations and the conjugate relations are added into the data structure. The construction is finalised by calling a routine that completes the data structure for the collector. The following functions provide the necessary tools for setting up a collector.

1 ► `FromTheLeftCollector(` n `)`

returns an empty data structure for a collector with n generators. No generator has a relative order, no right hand sides of power and conjugate relations are defined. Two generators for which no right hand side of a conjugate relation is 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> PcpGroupByCollector( ftl );
Pcp-group with orders [ 0, 0, 0, 0 ]
gap> IsAbelian(last);
true
```

If the relative order of a generators has been defined (see 3.1.2), but the right hand side of the corresponding power relation has not, then the order and the relative order of the generator are the same.

2 ▶ `SetRelativeOrder(` *coll*, *i*, *ro* `)`
▶ `SetRelativeOrderNC(` *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 3.1.1, i is a generator number and *ro* is a non-negative integer. The generator number i is an integer in the range $1, \ldots, n$ where n is the number of generators of the collector.

If *ro* is 0, then the generator with number i has infinite order and no power relation can be specified. As a side effect in this case, a previously defined power relation is deleted.

If *ro* is the relative order of a generator with number i and no power relation is set for that generator, then *ro* is the order of that generator.

The NC version of the function bypasses checks on the range of i.

```
gap> ftl := FromTheLeftCollector( 4 );
<<from the left collector with 4 generators>>
gap> for i in [1..4] do SetRelativeOrder( ftl, i, 3 ); od;
gap> G := PcpGroupByCollector( ftl );
Pcp-group with orders [ 3, 3, 3, 3 ]
gap> IsElementaryAbelian( G );
true
```

3 ▶ `SetPower(` *coll*, *i*, *rhs* `)`
▶ `SetPowerNC(` *coll*, *i*, *rhs* `)`

set the right hand side of the power relation for generator i in collector *coll* to (a copy of) *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 generator number and *rhs* is a generators exponent list or an element from a free group.

The no-check (NC) version of the function bypasses checks on the range of i and stores *rhs* (instead of a copy) in the collector.

4 ▶ `SetConjugate(` *coll*, *j*, *i*, *rhs* `)`
▶ `SetConjugateNC(` *coll*, *j*, *i*, *rhs* `)`

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

The generator number i can be negative in order to define conjugation by the inverse of a generator.

The no-check (NC) version of the function bypasses checks on the range of i and j and stores *rhs* (instead of a copy) in the collector.

5 ▶ `SetCommutator(` *coll*, *j*, *i*, *rhs* `)`

set the right hand side of the conjugate relation for the generators j and i with j larger than i by specifying the commutator of j and i. The parameter *coll* is a collector, j and i are generator numbers and *rhs* is a generator exponent list or an element from a free group.

The generator number i can be negative in order to define the right hand side of a commutator relation with the second generator being the inverse of a generator.

6 ▶ `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.

Note that `UpdatePolycyclicCollector` is automatically called by the function `PcpGroupByCollector` (see 4.3.1).

7 ▶ `IsConfluent(` *coll* `)`

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

Compare Chapter 2 for a definition of confluence.

Note that confluence is automatically checked by the function `PcpGroupByCollector` (see 4.3.1).

The following example defines a collector for a semidirect product of the cyclic group of order 3 with the free abelian group of rank 2. The action of the cyclic group on the free abelian group is given by the matrix

$$\begin{pmatrix} 0 & 1 \\ -1 & -1 \end{pmatrix}.$$

This leads to the following polycyclic presentation:

$$\langle g_1, g_2, g_3 | g_1^3, g_2^{g_1} = g_3, g_3^{g_1} = g_2^{-1} g_3^{-1}, g_3^{g_2} = g_3 \rangle.$$

```
gap> ftl := FromTheLeftCollector( 3 );
<<from the left collector with 3 generators>>
gap> SetRelativeOrder( ftl, 1, 3 );
gap> SetConjugate( ftl, 2, 1, [3,1] );
gap> SetConjugate( ftl, 3, 1, [2,-1,3,-1] );
gap> UpdatePolycyclicCollector( ftl );
gap> IsConfluent( ftl );
true
```

The action of the inverse of g_1 on $\langle g_2, g_2 \rangle$ is given by the matrix

$$\begin{pmatrix} -1 & -1 \\ 1 & 0 \end{pmatrix}.$$

The corresponding conjugate relations are automatically computed by `UpdatePolycyclicCollector`. It is also possible to specify the conjugation by inverse generators. Note that you need to run `UpdatePolycyclic-Collector` after one of the set functions has been used.

```
gap> SetConjugate( ftl, 2, -1, [2,-1,3,-1] );
gap> SetConjugate( ftl, 3, -1, [2,1] );
gap> IsConfluent( ftl );
Error, Collector is out of date called from
CollectWordOrFail( coll, ev1, [ j, 1, 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
brk>
gap> UpdatePolycyclicCollector( ftl );
gap> IsConfluent( ftl );
true
```

3.2 Accessing Parts of a Collector

1 ▶ `RelativeOrders(` *coll* `)`

returns (a copy of) the list of relative order stored in the collector *coll*.

2 ▶ `GetPower(` *coll*, *i* `)`
▶ `GetPowerNC(` *coll*, *i* `)`

returns a copy of the generator exponent list stored for the right hand side of the power relation of the generator i in the collector *coll*.

The no-check (NC) version of the function bypasses checks on the range of i and does not create a copy before returning the right hand side of the power relation.

3 ▶ `GetConjugate(` *coll*, *j*, *i* `)`
▶ `GetConjugateNC(` *coll*, *j*, *i* `)`

returns a copy of the right hand side of the conjugate relation stored for the generators j and i in the collector *coll* as generator exponent list. The generator j must be larger than i.

The no-check (NC) version of the function bypasses checks on the range of i and j and does not create a copy before returning the right hand side of the power relation.

4 ▶ `NumberOfGenerators(` *coll* `)`

returns the number of generators of the collector *coll*.

5 ▶ `ObjByExponents(` *coll*, *expvec* `)`

returns a generator exponent list for the exponent vector *expvec*. This is the inverse operation to `ExponentsByObj`. See `ExponentsByObj` (3.2.6) for an example.

6 ▶ `ExponentsByObj(` *coll*, *genexp* `)`

returns an exponent vector for the generator exponent list *genexp*. This is the inverse operation to `ObjByExponents`.

```
gap> G := UnitriangularPcpGroup( 4, 0 );
Pcp-group with orders [ 0, 0, 0, 0, 0, 0 ]
gap> coll := Collector ( G );
<<from the left collector with 6 generators>>
gap> ObjByExponents( coll, [6,-5,4,3,-2,1] );
[ 1, 6, 2, -5, 3, 4, 4, 3, 5, -2, 6, 1 ]
gap> ExponentsByObj( coll, last );
[ 6, -5, 4, 3, -2, 1 ]
```

3.3 Special Features

In this section we descibe collectors for nilpotent groups which make use of the special structure of the given pc-presentation.

1 ▶ `IsWeightedCollector(` *coll* `)`

checks if there is a function w from the generators of the collector *coll* into the positive integers such that $w(g) \geq w(x) + w(y)$ for all generators x, y and all generators g in (the normal of) $[x, y]$. If such a function does not exist, false is returned. If such a function exists, it is computed and stored in the collector. In addition, the default collection strategy for this collector is set to combinatorial collection.

2 ► `AddHallPolynomials(` *coll* `)`

is applicable to a collector which passes `IsWeightedCollector` and computes the Hall multiplication polynomials for the presentation stored in *coll*. The default strategy for this collector is set to evaluating those polynomial when multiplying two elements.

3 ► `String(` *coll* `)`

converts a collector *coll* into a string.

4 ► `FTLCollectorPrintTo(` *file*`,` *name*`,` *coll* `)`

stores a collector *coll* in the file *file* such that the file can be read back using the function 'Read' into GAP and would then be stored in the variable *name*.

5 ► `FTLCollectorAppendTo(` *file*`,` *name*`,` *coll* `)`

appends a collector *coll* in the file *file* such that the file can be read back into GAP and would then be stored in the variable *name*.

6 ► `UseLibraryCollector`

this property can be set to `true` for a collector to force a simple from-the-left collection strategy implemented in the GAP language to be used. Its main purpose is to help debug the collection routines.

7 ► `USE_LIBRARY_COLLECTOR`

this global variable can be set to `true` to force all collectors to use a simple from-the-left collection strategy implemented in the GAP language to be used. Its main purpose is to help debug the collection routines.

8 ► `DEBUG_COMBINATORIAL_COLLECTOR`

this global variable can be set to `true` to force the comparison of results from the combinatorial collector with the result of an identical collection performed by a simple from-the-left collector. Its main purpose is to help debug the collection routines.

9 ► `USE_COMBINATORIAL_COLLECTOR`

this global variable can be set to `false` in order to prevent the combinatorial collector to be used.

4 Pcp-groups - polycyclically presented groups

4.1 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 a 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 3.

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*. This list *word* consists of a sequence of generator numbers and their corresponding exponents and is of the form $[i_1, e_{i_1}, i_2, e_{i_2}, \ldots, i_r, e_{i_r}]$. 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.

4.2 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 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 with respect to the defining generating set of the underlying collector.

3 ► `GenExpList(` *g* `)`

returns the generators exponent list of the pcp-element g with respect to 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.

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

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

The function calls `UpdatePolycyclicCollector` (see 3.1.6) and checks the confluence (see 3.1.7) of the collector.

The non-check version bypasses these checks.

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 ]
```

5 Basic 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.

5.1 Elementary methods for pcp-groups

In this chapter we describe some important basic functions which are available for pcp-groups. A number of higherlevel 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 if U is finite and it returns the list [fail] otherwise.

7 ► `ClosureGroup(` U`,` V `)`

returns the group generated by U and V.

8 ► `NormalClosure(` U`,` V `)`

returns the normal closure of V under action of U.

9 ► `HirschLength(` U `)`

returns the Hirsch length of U.

10 ► `CommutatorSubgroup(` U`,` V `)`

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

11 ► `PRump(` U`,` p `)`

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

12 ► `SmallGeneratingSet(` U `)`

returns a small generating set for U.

5.2 Elementary properties of pcp-groups

1 ► `IsSubgroup(` *U*`,` *V* `)`

tests if V is a subgroup of U.

2 ► `IsNormal(` *U*`,` *V* `)`

tests if V is normal in U.

3 ► `IsNilpotentGroup(` *U* `)`

checks whether U is nilpotent.

4 ► `IsAbelian(` *U* `)`

checks whether U is abelian.

5 ► `IsElementaryAbelian(` *U* `)`

checks whether U is elementary abelian.

6 ► `IsFreeAbelian(` *U* `)`

checks whether U is free abelian.

5.3 Subgroups of pcp-groups

A subgroup of a pcp-group G can be defined by a set of generators as described in Section 4.3. However, many computations with a subgroup U need an **induced generating sequence** or **igs** of U. An igs is a sequence of generators 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 form 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. This means that each operation that is applied to the first list is also applied to the second list.

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. This means that each operation that is applied to the first list is also applied to the second list.

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. This means that each operation that is applied to the first list and the element *gens* is also applied to the second list and the lement *gens2*. The third version is available for efficiency reasons and assumes that the second list *igs2* is not only a generating set, but an igs.

5.4 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 with respect to 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* `)`

let *pcp* be a Pcp of a subgroup or a factor group of a pcp-group. This function computes a new pcp-group whose defining generators correspond to the generators in *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 := ExamplesOfSomePcpGroups(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 with respect to pcp1
gap> ExponentsByPcp( pcp1, g );
[ 1, 0, 1, 0 ]

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

5.5 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 $G \rightarrow G/N$. 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 ) )`.

5.6 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 *hom*. 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.

5.7 Changing the defining pc-presentation

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 $G \to H$.

2 ► `PcpGroupBySeries( ser )`
► `PcpGroupBySeries( ser, "snf" )`

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 $G \to H$. In the second form, the series must have abelian factors. The pcp of the group returned corresponds to a decomposition of each abelian factor into a direct product of cyclic groups.

```
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 ]
```

5.8 Printing a pc-presentation

By default, a pcp-group is printed using its relative orders only. The following methods can be used to view the pcp presentation of the group.

1 ► `PrintPcpPresentation(` *G* `)`
► `PrintPcpPresentation(` *pcp* `)`

prints the pcp presentation defined by the igs of G or the pcp *pcp*. As usual, the trivial conjugate relations are omitted from this presentation to shorten notation. Also, the relations obtained from conjugating with inverse generators are included only if the conjugating generator has infinite order. If this generator has finite order, then the conjugation relation is a consequence of the remaining relations.

2 ► `PrintPcpPresentation(` *G*`, "all" )`
► `PrintPcpPresentation(` *pcp*`, "all" )`

prints the full defining pcp presentation defined by the igs of G or the pcp *pcp*. Here the trivial conjugate relations are always included and also the conjugation with inverses is always included.

5.9 Converting to and from a presentation

1 ► `IsomorphismPcpGroup(` *G* `)`

returns an isomorphism from G onto a pcp-group H. There are various methods installed for this operation and some of these methods are part of the Polycyclic package, while others may be part of other packages.

For example, Polycyclic contains methods for this function in the case that G is a finite pc-group or a finite solvable permutation group. Further, Polycyclic can convert a finitely presented group with a polycyclic presentation into a pcp group via IsomorphismPcpGroup.

Other examples for methods for IsomorphismPcpGroup are the methods for the case that G is a crystallographic group (see Cryst) or the case that G is an almost crystallographic group (see AClib). A general method for the case that G is a rational polycyclic matrix group will be included in the Polenta package. A general method for the case that G is a polycyclic finitely presented group will be included in the IPCQ package.

2 ► `IsomorphismPcGroup(` *G* `)`

pc-groups are a representation for finite polycyclic groups. This function can convert finite pcp-groups to pc-groups.

6 Libraries and examples of pcp-groups

6.1 Libraries of various types of polycyclic 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 an odd integer, then 'fail' is returned. If n is zero or not an integer, then the infinite dihedral group is returned.

3 ▶ `UnitriangularPcpGroup(` *n*, *c* `)`

returns a pcp-group isomorphic to the group of upper triangular in $GL(n, R)$ where $R = \mathbb{Z}$ if $c = 0$ and $R = \mathbb{F}_p$ if $c = p$. The natural unitriangular matrix representation of the returned pcp-group G can be obtained as $G! \cdot isomorphism$.

4 ▶ `SubgroupUnitriangularPcpGroup(` *mats* `)`

mats should be a list of upper unitriangular $n \times n$ matrices over $\mathbb{Z}$ or over $\mathbb{F}_p$. This function returns the subgroup of the corresponding 'UnitriangularPcpGroup' generated by the matrices in *mats*.

5 ▶ `InfiniteMetacyclicPcpGroup(` *n*, *m*, *r* `)`

Infinite metacyclic groups are classified in [BK00]. Every infinite metacyclic group G is isomorphic to a finitely presented group $G(m, n, r)$ with two generators a and b and relations of the form $a^n = b^m = 1$ and $[a, b] = a^{1-r}$, where m, n, r are three non-negative integers with $mn = 0$ and r relatively prime to m. If $r \equiv -1 \bmod m$ then n is even, and if $r \equiv 1 \bmod m$ then $m = 0$. Also m and n must not be 1.

Moreover, $G(m, n, r) \cong G(m', n', s)$ if and only if $m = m'$, $n = n'$, and either $r \equiv s$ or $r \equiv s^{-1} \bmod m$.

This function returns the metacyclic group with parameters n, m and r as a pcp-group with the pc-presentation $\langle x, y | x^n, y^m, y^x = y^r \rangle$. This presentation is easily transformed into the one above via the mapping $x \mapsto b^{-1}, y \mapsto a$.

6 ▶ `HeisenbergPcpGroup(` *n* `)`

returns the Heisenberg group on $2n$ **generators as pcp-group. This gives a group of Hirsch length 3**n.

7 ▶ `MaximalOrderByUnitsPcpGroup(` *f* `)`

takes as input a normed, irreducible polynomial over the integers. Thus f defines a field extension F over the rationals. This function returns the split extension of the maximal order O of F by the unit group U of O, where U acts by right multiplication on O.

8 ▶ `BurdeGrunewaldPcpGroup(` *s*, *t* `)`

returns a nilpotent group of Hirsch length 11 which has been constructed by Burde und Grunewald. If s is not 0, then this group has no faithful 12-dimensional linear representation.

6.2 Some asorted example groups

The functions in this section provide some more example groups to play with. They come with no further description and their investigation is left to the interested user.

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

returns an example of a metabelian group. The input parameters must be two positive integers greater than 1.

2 ► `ExamplesOfSomePcpGroups(` *n* `)`

this function takes values n in 1 up to 16 and returns for each input an example of a pcp-group. The groups in this example list have been used as test groups for the functions in this package.

7 Higher level methods for pcp-groups

This is a description of some higher level functions of the polycyclic 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. For background and a description of the underlying algorithms we refer to [Eic01a].

7.1 Subgroup series in pcp-groups

Many algorithm for pcp-groups work by induction using some series through the group. In this section we provide a number of useful series for pcp-groups. An **efa series** is a normal series with elementary or free abelian factors. See [Eic00] for outlines 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 ▶ `SemiSimpleEfaSeries(` U `)`

returns an efa series of U such that every factor in the series is semisimple as a module for U over a finite field or over the rationals.

4 ▶ `DerivedSeries(` U `)`

the derived series of U.

5 ▶ `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.

6 ▶ `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.

7 ▶ `LowerCentralSeries(` U `)`

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

8 ▶ `UpperCentralSeries(` U `)`

the upper central series of U. This function always terminates, but it may terminate at a proper subgroup of U.

9 ▶ `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 := ExamplesOfSomePcpGroups(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.

10 ► `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.

11 ► `PcpsOfEfaSeries(` U `)`

returns a list of pcps corresponding to an efa series of U.

```
gap> G := ExamplesOfSomePcpGroups(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( DerivedSeries(G), "snf" );
[ Pcp [ g2, g3, g1 ] with orders [ 2, 2, 4 ],
  Pcp [ g4^2, g3^-2, g2^2*g4^2 ] with orders [ 4, 0, 0 ],
  Pcp [ g4^8 ] with orders [ 0 ] ]
gap> G.1^4 in DerivedSubgroup( G );
true
gap> G.1^2 = G.4;
true
```

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

7.2 Orbit stabilizer methods for pcp-groups

Let U be a pcp-group which acts on a set Ω. One of the fundamental problems in algorithmic group theory is the determination of orbits and stabilizers of points in Ω under the action of U. We distinguish two cases: the case that all considered orbits are finite and the case that there are infinite orbits. In the latter case, an orbit cannot be listed and a description of the orbit and its corresponding stabilizer is much harder to obtain.

If the considered orbits are finite, then the following two functions can be applied to compute the considered orbits and their corresponding stabilizers.

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

The input *gens* can be an igs or a pcp of a pcp-group U. The elements in the list *gens* act as the elements in the list *acts* via the function *oper* on the given points; that is, *oper(point, acts[i])* applies the ith generator to a given point. Thus the group defined by *acts* must be a homomorphic image of the group defined by *gens*. The first function returns a record containing the orbit as component 'orbit' and and igs for the stabilizer as component 'stab'. 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 ] ] ] )
```

If the considered orbits are infinite, then it may not always be possible to determine a description of the orbits and their stabilizers. However, as shown in [EO02] and [Eic02], it is possible to determine stabilizers and check if two elements are contained in the same orbit if the given action of the polycyclic group is a unimodular linear action on a vector space. The following functions are available for this case.

2 ► `StabilizerIntegralAction(` *U*, *mats*, *v* `)`
► `OrbitIntegralAction(` *U*, *mats* , *v*, *w* `)`

The first function computes the stabilizer in U of the vector v where the pcp group U acts via *mats* on an integral space and v and w are elements in this integral space. The second function checks whether v and w are in the same orbit and the function returns either *false* or a record containing an element in U mapping v to w and the stabilizer of v.

3 ► `NormalizerIntegralAction(` *U*, *mats*, *B* `)`
► `ConjugacyIntegralAction(` *U*, *mats*, *B*, *C* `)`

The first function computes the normalizer in U of the lattice with the basis B, where the pcp group U acts via *mats* on an integral space and B is a subspace of this integral space. The second functions checks whether the two lattices with the bases B and C are contained in the same orbit under U. The function returns either *false* or a record with an element in U mapping B to C and the stabilizer of B.

```
# get a pcp group and a free abelian normal subgroup
gap> G := ExamplesOfSomePcpGroups(8);
Pcp-group with orders [ 0, 0, 0, 0, 0 ]
gap> efa := EfaSeries(G);
[ Pcp-group with orders [ 0, 0, 0, 0, 0 ],
  Pcp-group with orders [ 0, 0, 0, 0 ],
  Pcp-group with orders [ 0, 0, 0 ],
  Pcp-group with orders [  ] ]
gap> N := efa[3];
Pcp-group with orders [ 0, 0, 0 ]
gap> IsFreeAbelian(N);
true

# create conjugation action on N
gap> mats := LinearActionOnPcp(Igs(G), Pcp(N));
[ [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  [ [ 0, 0, 1 ], [ 1, -1, 1 ], [ 0, 1, 0 ] ],
  [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ] ]

# take an arbitrary vector and compute its stabilizer
gap> StabilizerIntegralAction(G,mats, [2,3,4]);
Pcp-group with orders [ 0, 0, 0, 0 ]
gap> Igs(last);
[ g1, g3, g4, g5 ]

# check orbits with some other vectors
gap> OrbitIntegralAction(G,mats, [2,3,4],[3,1,5]);
rec( stab := Pcp-group with orders [ 0, 0, 0, 0 ], prei := g2 )

gap> OrbitIntegralAction(G,mats, [2,3,4], [4,6,8]);
false

# compute the orbit of a subgroup of Z^3 under the action of G
gap> NormalizerIntegralAction(G, mats, [[1,0,0],[0,1,0]]);
Pcp-group with orders [ 0, 0, 0, 0, 0 ]
gap> Igs(last);
[ g1, g2^2, g3, g4, g5 ]
```

7.3 Centralizers, Normalizers and Intersections

In this section we list a number of operations for which there are methods installed to compute the corresponding features in polycyclic groups.

1 ▶ `Centralizer(` U, g `)`
▶ `IsConjugate(` U, g, h `)`

These functions solve the conjugacy problem for elements in pcp-groups and they can be used to compute centralizers. The first method returns a subgroup of the given group U, the second method either returns a conjugating element or false if no such element exists.

The methods are based on the orbit stabilizer algorithms described in [EO02]. For nilpotent groups, an algorithm to solve the conjugacy problem for elements is described in [Sim94].

2 ► `Centralizer(` *U* `,` *V* `)`
► `Normalizer(` *U* `,` *V* `)`
► `IsConjugate(` *U* `,` *V* `,` *W* `)`

These three functions solve the conjugacy problem for subgroups and compute centralizers and normalizers of subgroups. The first two functions return subgroups of the input group U, the third function returns a conjugating element or false if no such element exists.

The methods are based on the orbit stabilizer algorithms described in [Eic02]. For nilpotent groups, an algorithm to solve the conjugacy problems for subgroups is described in [Lo98a].

3 ► `Intersection(` *U* `,` *N* `)`

A general method to compute intersections of subgroups of a pcp-group is described in [Eic01a], but it is not yet implemented here. However, intersections of subgroups $U, N \leq G$ can be computed if N is normalising U. See [Sim94] for an outline of the algorithm.

7.4 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 := ExamplesOfSomePcpGroups(15);
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 ]
```

7.5 Subgroups of finite index and maximal subgroups

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. Also, we refer to [Lo98b] for an alternative appraoch.

1 ► `MaximalSubgroupClassesByIndex(` 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 the conjugacy classes 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. Such a subgroup exists in every polycyclic group and this function computes such a subgroup using LowIndexNormal. However, we note that this function is not very efficient and the function NilpotentByAbelianByFiniteSeries may well be more efficient on this task.

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

gap> MaximalSubgroupClassesByIndex( 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 ]
```

7.6 Further attributes for pcp-groups based on the Fitting subgroup

In this section we provide a variety of other attributes for pcp-groups. Most of the methods below are based or related to the Fitting subgroup of the given group. We refer to [Eic01b] for a description of the underlying methods.

1 ▶ `FittingSubgroup(` U `)`

returns the Fitting subgroup of U; that is, the largest nilpotent normal subgroup of U.

2 ▶ `IsNilpotentByFinite(` U `)`

checks whether the Fitting subgroup of U has finite index.

3 ▶ `Centre(` U `)`

returns the centre of U.

4 ▶ `FCCentre(` U `)`

returns the FC-centre of U; that is, the subgroup containing all elements having a finite conjugacy class in U.

5 ▶ `PolyZNormalSubgroup(` U `)`

returns a normal subgroup N of U such that N has a polycyclic series with infinite factors only.

6 ▶ `NilpotentByAbelianByFiniteSeries(` U `)`

returns a normal series $1 \leq F \leq A \leq U$ such that F is nilpotent, A/F is abelian and U/A is finite. This series is computed using the Fitting subgroup and the centre of the Fitting factor.

7.7 Functions for nilpotent groups

There are (very few) functions which are available for nilpotent groups only. First, there are the different central series. These are available for all groups, but for nilpotent groups they terminate and provide series though the full group. Secondly, the determination of a minimal generating set is available for nilpotent groups only.

1 ▶ `MinimalGeneratingSet(` U `)`

```
gap> G := ExamplesOfSomePcpGroups(14);
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 ]
```

7.8 Random methods for pcp-groups

Below we introduce a function which computes orbit and stabilizer using a random method. This function tries to approximate the orbit and the stabilizer, but the returned orbit or stabilizer may be incomplete. This function is used in the random methods to compute normalizers and centralizers. Note that determinstic methods for these purposes are also available.

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

2 ▶ `RandomCentralizerPcpGroup(` *U*, *g* `)`
▶ `RandomCentralizerPcpGroup(` *U*, *V* `)`

3 ▶ `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 ]
```

7.9 Non-abelian tensor product and Schur extensions

1 ► `SchurExtension( G )`

Let G be a polycyclic group with a polycyclic generating sequence consisting of n elements. This function computes the largest central extension H of G such that H is generated by n elements. If F/R is the underlying polycyclic presentation for G, then H is isomorphic to $F/[R, F]$.

```
gap> G := DihedralPcpGroup( 0 );
Pcp-group with orders [ 2, 0 ]
gap> Centre( G );
Pcp-group with orders [  ]
gap> H := SchurExtension( G );
Pcp-group with orders [ 2, 0, 0, 0 ]
gap> Centre( H );
Pcp-group with orders [ 0, 0 ]
gap> H/Centre(H);
Pcp-group with orders [ 2, 0 ]
gap> Subgroup( H, [H.1,H.2] ) = H;
true
```

2 ► `SchurExtensionEpimorphism( G )`

returns the projection from the Schur extension G^* of G onto G. See the function `SchurExtension`. The kernel of this epimorphism is the direct product of the Schur multiplicator of G and a direct product of n copies of $\mathbb{Z}$ where n is the number of generators in the polycyclic presentation for G. The Schur multiplicator is the intersection of the kernel and the derived group of the source. See also the function `SchurCovering`.

```
gap> gl23 := Range( IsomorphismPcpGroup( GL(2,3) ) );
Pcp-group with orders [ 2, 3, 2, 2, 2 ]
gap> SchurExtensionEpimorphism( gl23 );
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10 ] -> [ g1, g2, g3, g4, g5,
id, id, id, id, id ]
gap> Kernel( last );
Pcp-group with orders [ 0, 0, 0, 0, 0 ]
gap> SchurMultiplicator( gl23 );
[  ]
gap> Intersection( Kernel(epi), DerivedSubgroup( Source(epi) ) );
[  ]
```

There is a crossed pairing from G into $(G^*)'$ which can be defined via this epimorphism:

```
gap> G := DihedralPcpGroup(0);
Pcp-group with orders [ 2, 0 ]
gap> epi := SchurExtensionEpimorphism( G );
[ g1, g2, g3, g4 ] -> [ g1, g2, id, id ]
gap> PreImagesRepresentative( epi, G.1 );
g1
gap> PreImagesRepresentative( epi, G.2 );
g2
gap> Comm( last, last2 );
g2^-2*g4
```

3 ► `SchurCovering( G )`

computes a Schur covering group of the polycyclic group G. A Schur covering is a largest central extension H of G such that the kernel M of the projection of H onto G is contained in the commutator subgroup of H.

If G is given by a presentation F/R, then M is isomorphic to the subgroup $R \cap [F,F]/[R,F]$. Let C be a complement to $R \cap [F,F]/[R,F]$ in $R/[R,F]$. Then F/C is isomorphic to H and R/C is isomorphic to M.

```
gap> G := AbelianPcpGroup( 3,[] );
Pcp-group with orders [ 0, 0, 0 ]
gap> ext := SchurCovering( G );
Pcp-group with orders [ 0, 0, 0, 0, 0, 0 ]
gap> Centre( ext );
Pcp-group with orders [ 0, 0, 0 ]
gap> IsSubgroup( DerivedSubgroup( ext ), last );
true
```

4 ▶ `SchurMultiplicator(` G `)`

computes the isomorphism type of the Schur multiplicator of G and returns a list of pairs describing the isomorphism type as a direct product of cyclic groups. The first component of a pair is the order of the cyclic group. The second component specifies how often that cyclic group occurs as a direct factor. The isomorphism type of the Schur multiplicator is the direct product of the groups specified by the list of pairs.

If a cyclic factor is infinite, then the first component of the corresponding pair is 0.

Note that the Schur multiplicator of a polycyclic group is a fintiely generated abelian group.

```
gap> G := DihedralPcpGroup( 0 );
Pcp-group with orders [ 2, 0 ]
gap> DirectProduct( G, AbelianPcpGroup( 2, [] ) );
Pcp-group with orders [ 0, 0, 2, 0 ]
gap> SchurMultiplicator( last );
[ [ 2, 4 ], [ 0, 1 ] ]
```

5 ▶ `NonAbelianExteriorSquareEpimorphism(` G `)`

returns the epimorphism of the non-abelian exterior square of a polycylic group G onto the derived group of G. The non-abelian exterior square can be defined as the derived subgroup of a Schur cover of G. The isomorphism type of the non-abelian exterior square is unique despite the fact that the isomorphism type of a Schur cover of a polycyclic groups need not be unique. The derived group of a Schur cover has a natural projection onto the derived group of G which is what the function returns.

The kernel of the epimorphism is isomorphic to the Schur multiplicator of G.

```
gap> G := ExamplesOfSomePcpGroups( 3 );
Pcp-group with orders [ 0, 0 ]
gap> G := DirectProduct( G,G );
Pcp-group with orders [ 0, 0, 0, 0 ]
gap> SchurMultiplicator( G );
[ [ 0, 1 ], [ 2, 3 ] ]
gap> epi := NonAbelianExteriorSquareEpimorphism( G );
[ g2^-2*g5, g4^-2*g10, g6, g7, g8, g9 ] -> [ g2^-2, g4^-2, id, id, id, id ]
gap> Kernel( epi );
Pcp-group with orders [ 0, 2, 2, 2 ]
gap> Collected( AbelianInvariants( last ) );
[ [ 0, 1 ], [ 2, 3 ] ]
```

6 ▶ `NonAbelianExteriorSquare(` G `)`

computes the non-abelian exterior square of a polycylic group G. See the explanation for `NonAbelianExteriorSquareEpimorphism`. The natural projection of the non-abelian exterior square onto the derived group of G is stored in the component `!.epimorphism`.

There is a crossed pairing from G into $G \wedge G$. See the function `SchurExtensionEpimorphism` for details. The crossed pairing is stored in the component `!.crossedPairing`. This is the crossed pairing λ in [EN07].

```
gap> G := DihedralPcpGroup(0);
Pcp-group with orders [ 2, 0 ]
gap> GwG := NonAbelianExteriorSquare( G );
Pcp-group with orders [ 0 ]
gap> lambda := GwG!.crossedPairing;
function( g, h ) ... end
gap> lambda( G.1, G.2 );
g2^2*g4^-1
```

7 ► `NonAbelianTensorSquareEpimorphism(` G `)`

returns for a polycyclic group G the projection of the non-abelian tensor square $G \otimes G$ onto the non-abelian exterior square $G \wedge G$. The range of that epimorphism has the component `!.epimorphism` set to the projection of the non-abelian exterior square onto the derived group of G. See also the function `NonAbelianExteriorSquare`.

With the result of this function one can compute the groups in the commutative diagram at the beginning of the paper [EN07]. The kernel of the returned epimorphism is the group $\nabla(G)$. The kernel of the composition of this epimorphism and the above mention projection onto G' is the group $J(G)$.

```
gap> G := DihedralPcpGroup(0);
Pcp-group with orders [ 2, 0 ]
gap> G := DirectProduct(G,G);
Pcp-group with orders [ 2, 0, 2, 0 ]
gap> alpha := NonAbelianTensorSquareEpimorphism( G );
[ g9*g25^-1, g10*g26^-1, g11*g27, g12*g28, g13*g29, g14*g30, g15, g16,
g17,
  g18, g19, g20, g21, g22, g23, g24 ] -> [ g2^-2*g6, g4^-2*g12, g8,
  g9, g10,
  g11, id, id, id, id, id, id, id, id, id, id ]
gap> gamma := Range( alpha )!.epimorphism;
[ g2^-2*g6, g4^-2*g12, g8, g9, g10, g11 ] -> [ g2^-2, g4^-2, id, id,
id, id ]
gap> JG := Kernel( alpha * gamma );
Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> Image( alpha, JG );
Pcp-group with orders [ 2, 2, 2, 2 ]
gap> SchurMultiplicator( G );
[ [ 2, 4 ] ]

\>NonAbelianTensorSquare( <G> )

computes  for a  polycyclic group  <G> the  non-abelian  tensor square
$G\otimes G$.

\beginexample
gap> G := AlternatingGroup( IsPcGroup, 4 );
<pc group of size 12 with 3 generators>
gap> PcGroupToPcpGroup( G );
Pcp-group with orders [ 3, 2, 2 ]
gap> NonAbelianTensorSquare( last );
```

```
Pcp-group with orders [ 2, 2, 2, 3 ]
gap> PcpGroupToPcGroup( last );
<pc group of size 24 with 4 generators>
gap> DirectFactorsOfGroup( last );
[ Group([ f1, f2, f3 ]), Group([ f4 ]) ]
gap> List( last, Size );
[ 8, 3 ]
gap> IdGroup( last2[1] );
[ 8, 4 ]       # the quaternion group of Order 8

gap> G := DihedralPcpGroup( 0 );
Pcp-group with orders [ 2, 0 ]
gap> ten := NonAbelianTensorSquare( G );
Pcp-group with orders [ 0, 2, 2, 2 ]
gap> IsAbelian( ten );
true
```

8 ▶ `NonAbelianExteriorSquarePlusEmbedding(` G `)`

returns an embedding from the non-abelian exterior square $G \wedge G$ into an extensions of $G \wedge G$ by $G \times G$. For the significance of the group see the paper [EN07]. The range of the epimorphism is the group $\tau(G)$ in that paper.

9 ▶ `NonAbelianExteriorSquarePlusEmbedding(` G `)`

returns the group $\tau(G)$ in [EN07].

10 ▶ `NonAbelianTensorSquarePlusEpimorphism(` G `)`

returns an epimorphisms of $\nu(G)$ onto $\tau(G)$. The group $\nu(G)$ is an extension of the non-abelian tensor square $G \otimes G$ of G by $G \times G$. The group $\tau(G)$ is an extension of the non-abelian exterior square $G \wedge G$ by $G \times G$. For details see [EN07].

11 ▶ `NonAbelianTensorSquarePlus(` G `)`

returns the group $\nu(G)$ in [EN07].

12 ▶ `WhiteheadQuadraticFunctor(` G `)`

returns Whitehead's universal quadratic functor of G, see [EN07] for a description.

7.10 Schur covers and Schur towers

A finite p-group G is a Schur tower, if $G/\gamma_{i+1}(G)$ is a Schur cover of $G/\gamma_i(G)$ for every i, where $\gamma_i(G)$ is the i-th term of the lower central series of G. This section contains a function to determine the Schur covers of a finite p-group up to isomorphism and it gives access to two libraries of Schur tower p-groups.

1 ▶ `SchurCovers(` G `)`

Let G be a finite p-group defined as a pcp group. This function returns a complete and irredundant set of isomorphism types of Schur covers of G. The algorithm implements a method of Nickel's Phd Thesis.

8 Cohomology for pcp-groups

The GAP 4 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 is outlined in [Eic00].

As a preparation for the cohomology computation, we introduce the cohomology records. These records provide the technical setup for our cohomology computations.

8.1 Cohomology records

Cohomology records provide the necessary technical setup for the cohomology computations for polycyclic groups.

1 ► `CRRecordByMats(` *U* `,` *mats* `)`

creates an external module. Let U be a pcp group which acts via the list of matrices *mats* on a vector space of the form $\mathbb{Z}^n$ or $\mathbb{F}_p^n$. Then this function creates a record which can be used as input for the cohomology computations.

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

creates an internal module. Let U be a pcp group and let A be a normal elementary or free abelian normal subgroup of A or let *pcp* be a pcp of a normal elementary of free abelian subfactor of U. Then this function creates a record which can be used as input for the cohomology computations.

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 right hand sides of the polycyclic relators of *factor* as generator exponents lists and a description for the corresponding left hand sides.

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*.

8.2 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 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*.

There is an additional function which can be used to compute the second cohomology group over an arbitrary finitely generated abelian group. The finitely generated abelian group should be realized as a factor of a free abelian group modulo a lattice. The function is called as

7 ► `TwoCohomologyModCR(` C `,` *lat* `)`

where C is a cohomology record and *lat* is a basis for a sublattice of a free abelian module. The output format is the same as for `TwoCohomologyCR`.

8.3 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 fixpoints 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.

8.4 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*.

8.5 Constructing pcp groups as extensions

This section contains an example application of the second cohomology group to the construction of pcp groups as extensions. The following constructs extensions of the group of upper unitriangular matrices with its natural lattice.

```
# get the group and its matrix action
gap> G := UnitriangularPcpGroup(3,0);
Pcp-group with orders [ 0, 0, 0 ]
gap> mats := G!.mats;
[ [ [ 1, 1, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  [ [ 1, 0, 0 ], [ 0, 1, 1 ], [ 0, 0, 1 ] ],
  [ [ 1, 0, 1 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ] ]

# set up the cohomology record
gap> C := CRRecordByMats(G,mats);;

# compute the second cohomology group
gap> cc := TwoCohomologyCR(C);;

# the abelian invariants of H^2(G,M)
gap> cc.factor.rels;
[ 2, 0, 0 ]

# construct an extension which corresponds to a cocycle that has
# infinite image in H^2(G,M)
gap> c := cc.factor.prei[2];
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, -1, 1 ]
gap> H := ExtensionCR( CR, c);
Pcp-group with orders [ 0, 0, 0, 0, 0, 0 ]

# check that the extension does not split - get the normal subgroup
gap> N := H!.module;
Pcp-group with orders [ 0, 0, 0 ]

# create the interal module
gap> C := CRRecordBySubgroup(H,N);;

# use the complements routine
gap> ComplementClassesCR(C);
[  ]
```

9 Matrix Representations

This chapter describes functions which compute with matrix representations for pcp-groups. So far the routines in this package are only able to compute matrix representations for torsion-free nilpotent groups.

9.1 Unitriangular matrix groups

1 ▶ `UnitriangularMatrixRepresentation( G )`

computes a faithful representation of a torsion-free nilpotent group G as unipotent lowertriangular matrices over the integers. The pc-presentation for G must not contain any power relations. The algorithm is described in [dGN02].

2 ▶ `IsMatrixRepresentation(` G, *matrices* `)`

checks if the map defined by maping the i-th generator of the pcp-group G to the i-th matrix of *matrices* defines a homomorphism.

9.2 Upper unitriangular matrix groups

We call a matrix upper unitriangular if it is an upper triangular matrix with ones on the main diagonal. The weight of an upper unitriangular matrix is the number of diagonals above the main diagonal that contain zeroes only.

The subgroup of all upper unitriangular matrices of $GL(n, \mathbb{Z})$ is torsion-free nilpotent. The k-th term of its lower central series is the set of all matrices of weight $k - 1$. The $\mathbb{Z}$-rank of the k-th term of the lower central series modulo the $(k + 1)$-th term is $n - k$.

1 ▶ `IsomorphismUpperUnitriMatGroupPcpGroup(` G `)`

takes a group G generated by upper unitriangular matrices over the integers and computes a polycylic presentation for the group. The function returns an isomorphism from the matrix group to the pcp group. Note that a group generated by upper unitriangular matrices is necessarily torsion-free nilpotent.

2 ▶ `SiftUpperUnitriMatGroup(` G `)`

takes a group G generated by upper unitriangular matrices over the integers and returns a recursive data structure L with the following properties: L contains a polycyclic generating sequence for G, using L one can decide if a given upper unitriangular matrix is contained in G, a given element of G can be written as a word in the polycyclic generating sequence. L is a representation of a chain of subgroups of G refining the lower centrals series of G.. It contains for each subgroup in the chain a minimal generating set.

3 ▶ `RanksLevels(` L `)`

takes the data structure returned by `SiftUpperUnitriMat` and prints the Z-rank of each the subgroup in L.

4 ▶ `MakeNewLevel(` m `)`

creates one level of the data structure returned by `SiftUpperUnitriMat` and initialises it with weight m.

5 ▶ `SiftUpperUnitriMat( gens, level, M )`

takes the generators *gens* of an upper unitriangular group, the data structure returned *level* by `SiftUpperUnitriMat` and another upper unitriangular matrix M. It sift M through *level* and adds M at the appropriate place if M is not contained in the subgroup represented by *level*.

The function `SiftUpperUnitriMatGroup` illustrates the use of `SiftUpperUnitriMat`.

```
InstallGlobalFunction( "SiftUpperUnitriMatGroup", function( G )
    local   firstlevel,  g;

    firstlevel := MakeNewLevel( 0 );
    for g in GeneratorsOfGroup(G) do
        SiftUpperUnitriMat( GeneratorsOfGroup(G), firstlevel, g );
    od;
    return firstlevel;
end );
```

6 ▶ `DecomposeUpperUnitriMat(` *level*, *M* `)`

takes the data structure *level* returned by `SiftUpperUnitriMatGroup` and a upper unitriangular matrix M and decomposes M into a word in the polycyclic generating sequence of *level*.

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.

[BK00] James R. Beuerle and Luise-Charlotte Kappe. Infinite metacyclic groups and their non-abelian tensor squares. *Proc. Edinburgh Math. Soc. (2)*, 43(3):651–662, 2000.

[dGN02] Willem A. de Graaf and Werner Nickel. Constructing faithful representations of finitely-generated torsion-free nilpotent groups. *J. Symbolic Comput.*, 33(1):31–41, 2002.

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

[Eic01a] Bettina Eick. Computations with polycyclic groups. Habilitationsschrift, Kassel, 2001.

[Eic01b] Bettina Eick. On the Fitting subgroup of a polycyclic-by-finite group and its applications. *J. Alg.*, 242:176 – 187, 2001.

[Eic02] Bettina Eick. Orbit-stabilizer problems and computing normalizers for polycyclic groups. *J. Symb. Comp.*, 34:1 – 19, 2002.

[EN07] Bettina Eick and Werner Nickel. Computing the schur multiplicator and the non-abelian tensor square of a polycyclic group. *submitted*, 2007.

[EO02] Bettina Eick and Gretchen Ostheimer. On the orbit stabilizer problem for integral matrix actions of polycyclic groups. *Accepted by Math. Comp*, 2002.

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

[LGS98] C. R. Leedham-Green and Leonard H. Soicher. Symbolic collection using Deep Thought. *LMS J. Comput. Math.*, 1:9–24 (electronic), 1998.

[Lo98a] E. H. Lo. Finding intersection and normalizer in finitely generated nilpotent groups. *J. Symbol. Comput.*, 25:45 – 59, 1998.

[Lo98b] Eddie H. Lo. Enumerating finite index subgroups of polycyclic groups. Unpublished report, 1998.

[Mer97] Wolfgang W. Merkwitz. Symbolische Multiplikation in nilpotenten Gruppen mit Deep Thought. Diplomarbeit, RWTH Aachen, 1997.

[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] Charles C. Sims. *Computation with finitely presented groups*, volume 48 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 1994.

[VL90] M. R. Vaughan-Lee. Collection from the left. *J. Symbolic Comput.*, 9(5-6):725–733, 1990.

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".

G

H

I

K

L

M

N

O

P