# NQL
## Computing nilpotent quotients
## of
## L-presented groups

## A GAP 4 package

by

René Hartung
Institute of Computational Mathematics
University of Braunschweig

# Contents

# 1 Preface

In 1980, Grigorchuk [Gri80] gave an example of an infinite, finitely generated torsion group which provided a first explicit counter-example to the General Burnside Problem. This counter-example is nowadays called the *Grigorchuk group* and was originally defined as a group of transformations of the unit interval which preserve the Lebesgue measure. Beside being a counter-example to the General Burnside Problem, the Grigorchuk group was a first example of a group with an intermediate growth function (see [Gri83]) and was used in the construction of a finitely presented amenable group which is not elementary amenable (see [Gri98]).

The Grigorchuk group is not finitely presentable (see [Gri99]). However, in 1985, Igor Lysenok (see [Lys85]) determined the following recursive presentation for the Grigorchuk group:

$$\langle a, b, c, d \mid a^2, b^2, c^2, d^2, bcd, [d, d^a]^{\sigma^n}, [d, d^{acaca}]^{\sigma^n}, (n \in \mathbb{N}) \rangle,$$

where $\sigma$ is the homomorphism of the free group over $\{a, b, c, d\}$ which is induced by $a \mapsto c^a, b \mapsto d, c \mapsto b$, and $d \mapsto c$. Hence, the infinitely many relators of this recursive presentation can be described in finite terms using powers of the endomorphism $\sigma$.

In 2003, Bartholdi [Bar03] introduced the notion of an *L-presentation* for presentations of this type; that is, a group presentation of the form

$$G = \left\langle S \ \middle| \ Q \cup \bigcup_{\varphi \in \Phi^*} R^\varphi \right\rangle,$$

where $\Phi^*$ denotes the free monoid generated by a set of free group endomorphisms $\Phi$. He proved that various branch groups are finitely $L$-presented but not finitely presentable and that every free group in a variety of groups satisfying finitely many identities is finitely $L$-presented (e.g. the Free Burnside- and the Free $n$-Engel groups).

The NQL-package defines new GAP objects to work with finitely $L$-presented groups. The main part of the package is a nilpotent quotient algorithm for finitely $L$-presented groups; that is, an algorithm which takes as input a finitely $L$-presented group $G$ and a positive integer $c$. It computes a polycyclic presentation for the lower central series quotient $G/\gamma_{c+1}(G)$. Therefore, a nilpotent quotient algorithm can be used to determine the abelian invariants of the lower central series sections $\gamma_c(G)/\gamma_{c+1}(G)$ and the largest nilpotent quotient of $G$ if it exists.

Our nilpotent quotient algorithm generalizes Nickel's algorithm for finitely presented groups (see [Nic96]) which is implemented in the NQ-package; see [Nic03]. In difference to the NQ-package, the NQL-package is implemented in GAP only.

Since finite $L$-presentations generalize finite presentations, our algorithm also applies to finitely presented groups. It coincides with Nickel's algorithm in this special case.

Our algorithm can be readily modified to determine the $p$-quotients of a finitely $L$-presented group. An implementation is planned for future expansions of the package.

A detailed description of our algorithm can be found in [BEH08] or in the diploma thesis [Har08] which is publicly available from the website

    http://www-public.tu-bs.de:8080/~y0019492/pub/index.html

Further the NQL-package includes the algorithms of [Hara] and [EH] for approximating the Schur multiplier and the outer automorphism group, respectively, of finitely $L$-presented groups.

# 2 An Introduction to L-presented groups

Let $S$ be an alphabet, $Q$ and $R$ be subsets of the free group $F_S$ over this alphabet, and $\Phi$ be a set of free group endomorphisms $\varphi\colon F_S \to F_S$. An *L-presentation* is a quadruple $(S, Q, \Phi, R)$ and it is called *finite* if the sets $S$, $Q$, $\Phi$, and $R$ are finite. A (finite) $L$-presentation $(S, Q, \Phi, R)$ defines the (*finitely*) *L-presented group*

$$G = \left\langle S \mid Q \cup \bigcup_{\varphi \in \Phi^*} R^\varphi \right\rangle$$

where $\Phi^*$ denotes the free monoid generated by $\Phi$; that is, the closure of $\Phi \cup \{\mathrm{id}\}$ under composition.

The elements in $Q$ are the *fixed relators* and the elements in $R$ are the *iterated relators* of the $L$-presentation $(S, Q, \Phi, R)$. An $L$-presentation of the form $(S, \emptyset, \Phi, R)$ is an *ascending L-presentation* and it is an *invariant L-presentation* if the normal subgroup

$$K = \left\langle Q \cup \bigcup_{\varphi \in \Phi^*} R^\varphi \right\rangle^{F_S}$$

is $\varphi$-invariant for each $\varphi \in \Phi$; that is, if $K$ satisfies $K^\varphi \subset K$ for each $\varphi \in \Phi$. Note that every ascending $L$-presentation is invariant and for each $L$-presentation $(S, Q, \Phi, R)$ there is a unique *underlying ascending L-presentation* $(S, \emptyset, \Phi, R)$ which is invariant. In general it is not decidable whether or not a given $L$-presentation is invariant as this would require a solution to the word-problem.

In the remainder of this manual, an $L$-presented group is always finitely $L$-presented.

## 2.1 Creating an L-presented group

The construction of an $L$-presented group is similar to the construction of a finitely presented group (see Chapter "ref:creating finitely presented groups" of the GAP Reference manual for further details).

1 ▶ LPresentedGroup( *F*, *frels*, *endos*, *irels* )                                                           F

returns the GAP object of an $L$-presented group with the underlying free group $F$, the fixed relators *frels*, the set of endomorphisms *endos*, and the iterated relators *irels*. The input variables *frels* and *irels* need to be finite subsets of the underlying free group $F$ and *endos* needs to be a finite list of homomorphisms $F \to F$.

For example, the Grigorchuk group,

$$\left\langle a, b, c, d \mid a^2, b^2, c^2, d^2, bcd, [d, d^a]^{\sigma^n}, [d, d^{acaca}]^{\sigma^n}, (n \in \mathbb{N}_0) \right\rangle,$$

can be constructed as follows.

```
gap> F:=FreeGroup( "a", "b", "c", "d" );
<free group on the generators [ a, b, c, d ]>
gap> AssignGeneratorVariables( F );
#I  Assigned the global variables [ a, b, c, d ]
gap> frels:=[ a^2, b^2, c^2, d^2, b*c*d ];;
gap> endos:=[ GroupHomomorphismByImagesNC( F, F, [ a, b, c, d ], [ c^a, d, b, c ]) ];;
gap> irels:=[ Comm( d, d^a ), Comm( d, d^(a*c*a*c*a) ) ];;
gap> G:=LPresentedGroup( F, frels, endos, irels );
<L-presented group on the generators [ a, b, c, d ]>
```

There are various examples of finitely $L$-presented groups available in the library of the NQL-package.

2 ▸ ExamplesOfLPresentations( $n$ )        F

returns some well-known examples of finitely $L$-presented groups. The input of this function needs to be a positive integer at most 10.

$n = 1$ The Grigorchuk group on 4 generators; cf. [Gri80], [Lys85], and [Bar03], Theorem 4.6,

$n = 2$ the Grigorchuk group on 3 generators; cf. [Gri80], [Lys85], and [Bar03], Theorem 4.6,

$n = 3$ the lamplighter group $\mathbb{Z}_2 \wr \mathbb{Z}$; cf. [Bar03], Theorem 4.1,

$n = 4$ the Brunner-Sidki-Vieira group; cf. [BSV99] and [Bar03], Theorem 4.4,

$n = 5$ the Grigorchuk supergroup; cf. [BG02] and [Bar03], Theorem 4.6,

$n = 6$ the Fabrykowski-Gupta group; cf. [FG85] and [BEH08],

$n = 7$ the Gupta-Sidki group; cf. [Sid87] and [BEH08],

$n = 8$ an index-3 subgroup of the Gupta-Sidki group

$n = 9$ the Basilica group; cf. [GZ02] and [BV05],

$n = 10$ Baumslag's finitely generated, infinitely related group with a trivial multiplier; cf. [Bau71].

Furthermore, every free group in a variety of groups satisfying finitely many identities is finitely $L$-presented. Some of these groups are available from the NQL-package using the following operations; for further details we refer to the diploma thesis [Har08].

3 ▸ FreeEngelGroup( $n$, $num$ )        O

returns an $L$-presentation for the free $n$-Engel group on $num$ generators; that is, the free group in the variety of $num$-generated groups satisfying the $n$-Engel identity.

4 ▸ FreeBurnsideGroup( $exp$, $num$)        O

returns an $L$-presentation for the free Burnside group on $num$ generators with exponent $exp$; that is, the free group in the variety of $num$-generated groups with exponent $exp$.

5 ▸ FreeNilpotentGroup( $c$, $num$ )        O

returns an $L$-presentation for the free nilpotent group of class $c$ on $n$ generators; that is, the free group in the variety of $num$-generated, nilpotent groups with nilpotency class $c$.

6 ▸ GeneralizedFabrykowskiGuptaLpGroup( $n$ )        O

returns an $L$-presentation for the $n$-th generalized Fabrykowski-Gupta group as constructed in [BEH08].

7 ▸ LamplighterGroup( $fil$, $int$)        C
  ▸ LamplighterGroup( $fil$, $PcGroup$)        C

returns a finite $L$-presentation for the lamplighter group on $int$ lamp states in the first case, if $fil$ is the filter IsLpGroup. In the second case, the group $PcGroup$ must be a finite cyclic group. Then the method

returns a finite *L*-presentation for the lamplighter group on `Size(`*PcGroup*`)` lamp states; for details on the
*L*-presentation see [Bar03].

```
gap> LamplighterGroup( IsLpGroup, 2 );
<L-presented group on the generators [ a, t, u ]>
gap> LamplighterGroup( IsLpGroup, CyclicGroup(3) );
<L-presented group on the generators [ a, t, u ]>
```

## 2.2 The underlying free group

An *L*-presented group is defined as an image of its underlying free group. Note that these are two different
GAP objects. The elements of the *L*-presented group are represented by words in the underlying free group.

1 ▶ `FreeGroupOfLpGroup(` *LpGroup* `)`                                                                A

returns the underlying free group of the *L*-presented group *LpGroup*.

2 ▶ `FreeGeneratorsOfLpGroup(` *LpGroup* `)`                                                            A

returns the generators of the free group which underlies the *L*-presented group *LpGroup*.

3 ▶ `GeneratorsOfGroup(` *LpGroup* `)`                                                                  O

returns the generators of the *L*-presented group *LpGroup*. These are the images of the generators of the
underlying free group under the natural homomorphism.

4 ▶ `UnderlyingElement(` *elm* `)`                                                                      O

returns the preimage of an *L*-presented group element *elm* in the underlying free group. More precisely, each
element of an *L*-presented group is represented by an element in the free group. This method returns the
corresponding element in the free group.

5 ▶ `ElementOfLpGroup(` *fam*, *elm* `)`                                                                O

returns the element in the *L*-presented group represented by the word *elm* on the generators of the underlying
free group, if *fam* is the family of *L*-presented group elements.

```
gap> F:=FreeGroup( 2 );;
gap> G:=LPresentedGroup( F, [ F.1^2 ], [ IdentityMapping( F ) ], [ F.2 ] );;
gap> FreeGroupOfLpGroup( G ) = F;
true
gap> GeneratorsOfGroup( G );
[ f1, f2 ]
gap> FreeGeneratorsOfLpGroup( G );
[ f1, f2 ]
gap> last = last2;
false
gap> UnderlyingElement( G.1 );
f1
gap> last in F;
true
gap> ElementOfLpGroup( ElementsFamily( FamilyObj( G ) ), last2 ) in G;
true
```

## 2.3 Accessing an L-presentation

The fixed relators, the iterated relators, and the endomorphisms of an *L*-presented group are accessible with the following methods.

1 ▶ FixedRelatorsOfLpGroup( *LpGroup* )                                                                          A

returns the fixed relators of the *L*-presented group *LpGroup* as elements of the underlying free group.

2 ▶ IteratedRelatorsOfLpGroup( *LpGroup* )                                                                       A

returns the iterated relators of the *L*-presented group *LpGroup* as elements of the underlying free group.

3 ▶ EndomorphismsOfLpGroup( *LpGroup* )                                                                          A

returns the endomorphisms of the *L*-presented group *LpGroup* as endomorphisms of the underlying free group.

```
gap> F:=FreeGroup( 2 );;
gap> G:=LPresentedGroup( F, [ F.1^2 ], [ IdentityMapping( F ) ], [ F.2 ] );
<L-presented group on the generators [ f1, f2 ]>
gap> FixedRelatorsOfLpGroup( G );
[ f1^2 ]
gap> IteratedRelatorsOfLpGroup( G );
[ f2 ]
gap> EndomorphismsOfLpGroup( G );
[ IdentityMapping( <free group on the generators [ f1, f2 ]> ) ]
```

## 2.4 Attributes and properties of L-presented groups

For the method-selection of the nilpotent quotient algorithm, an *L*-presented group may have the following attributes and properties.

1 ▶ UnderlyingAscendingLPresentation( *LpGroup* )                                                                A

returns the underlying ascending *L*-presentation of *LpGroup*; that is, if *LpGroup* is finitely *L*-presented by $(S, Q, \Phi, R)$, the underlying ascending *L*-presentation is $(S, \emptyset, \Phi, R)$.

2 ▶ UnderlyingInvariantLPresentation( *LpGroup* )                                                                A

attempts to compute a 'good' underlying invariant *L*-presentation for *LpGroup*; that is, if *LpGroup* is finitely *L*-presented by $(S, Q, \Phi, R)$, then this method seeks to find a subset $Q' \subseteq Q$ such that $(S, Q', \Phi, R)$ is an invariant *L*-presentation. Note that there is always the underlying ascending *L*-presentation $(S, \emptyset, \Phi, R)$. However, for the efficiency of the nilpotent quotient algorithm it is important that the subset $Q'$ is as big as possible.

Since it is undecidable, in general, whether or not a given *L*-presentation is invariant, there is no algorithm which can determine the best possible underlying invariant *L*-presentation. The method implemented for this attribute tries to compute a 'good' invariant *L*-presentation and will return the underlying ascending *L*-presentation in the worst case.

This attribute can be set manually using SetUnderlyingInvariantLPresentation. For instance, the Grigorchuk group

$$\left\langle a, b, c, d \;\middle|\; a^2, b^2, c^2, d^2, bcd, [d, d^a]^{\sigma^n}, [d, d^{acaca}]^{\sigma^n}, (n \in \mathbb{N}_0) \right\rangle,$$

is invariantly *L*-presented and therefore, it should be constructed as follows

```
gap> F:=FreeGroup( "a", "b", "c", "d" );;
gap> AssignGeneratorVariables( F );
#I  Assigned the global variables [ a, b, c, d ]
gap> frels:=[ a^2, b^2, c^2, d^2, b*c*d ];;
gap> endos:=[ GroupHomomorphismByImagesNC( F, F, [ a, b, c, d ], [ c^a, d, b, c ]) ];;
gap> irels:=[ Comm( d, d^a ), Comm( d, d^(a*c*a*c*a) ) ];;
gap> G:=LPresentedGroup( F, frels, endos, irels );
<L-presented group on the generators [ a, b, c, d ]>
gap> SetUnderlyingInvariantLPresentation( G, G );;
```

3 ▶ **IsAscendingLPresentation(** *LpGroup* **)**                                                                                    P

checks whether the *L*-presentation of *LpGroup* is ascending; that is, if the set of fixed relators is empty. This property is set automatically when creating an *L*-presented group with no fixed relators using the function `LPresentedGroup`.

4 ▶ **IsInvariantLPresentation(** *LpGroup* **)**                                                                                    P

attempts to check whether the *L*-presentation of *LpGroup* is invariant. In general, one cannot decide whether or not a given *L*-presentation is invariant. There are mainly two methods implemented for this property. The first method seeks to find a 'good' underlying invariant *L*-presentation using the operation `Under-lyingInvariantLPresentation`. If this latter *L*-presentation coincides with the *L*-presentation of *LpGroup*, then *LpGroup* is invariantly *L*-presented. If this method fails, then the second method uses the nilpotent quotient algorithm for *L*-presented groups which yields a necessary condition for an *L*-presented group to be invariantly *L*-presented. Note that the latter method may not terminate. For instance, both methods fail on Baumslag's finitely generated, infinitely related group with trivial multiplier returned by `ExamplesOfL-Presentations`.

5 ▶ **EmbeddingOfAscendingSubgroup(** *LpGroup* **)**                                                                                A

stores an embedding of an ascendingly *L*-presented subgroup of the *L*-presented group *LpGroup*. This attribute is set for ascending *L*-presentations only. In this case, the identity map of *LpGroup* is returned. This attribute is used in the FR-package which can construct various finitely *L*-presented groups. The embedding is useful for a nilpotent quotient algorithm of a non-invariantly *L*-presented group.

## 2.5 Methods for L-presented groups

1 ▶ **MappedWord(** *x*, *gens*, *imgs* **)**                                                                                         O

returns the group element obtained from *x* by replacing each occurrence of a generator in *gens* by the corresponding element in the list *imgs*. The lists *gens* and *imgs* need to have the same length.

2 ▶ **EpimorphismFromFpGroup(** *LpGroup*, *n* **)**                                                                                  O

returns an epimorphism from a finitely presented group onto *LpGroup*. The finitely presented group is obtained from *LpGroup* by applying only words of length at most *n* in the endomorphisms of *LpGroup* to the iterated relators of *LpGroup*.

3 ▶ **SplitExtensionByAutomorphismsLpGroup(** *LpGroup*, *H*, *auts***)**                                                            O

returns an *L*-presentation for the split extension of *LpGroup* by an *L*-presented or by a finitely presented group *H*. The action of a generator of *H* on *LpGroup* is given by an automorphism in the list *auts*. Thus for each generator of *H* there must be an automorphism in the list *auts*.

```
gap> F := FreeGroup( "a" );
<free group on the generators [ a ]>
gap> H := F / [ F.1^3 ];
<fp group on the generators [ a ]>
gap> U := ExamplesOfLPresentations( 8 );
<L-presented group on the generators [ t, u, v ]>
gap> aut:=GroupHomomorphismByImagesNC( U, U, [ U.1, U.2, U.3 ], [ U.2, U.3, U.1 ] );
[ t, u, v ] -> [ u, v, t ]
gap> SplitExtensionByAutomorphismsLpGroup( U, H, [ aut ] );
<L-presented group on the generators [ t, u, v, a ]>
```

4 ▶ **= (** *elm1*, *elm2* **)**                                                                                    O

uses the operation `NqEpimorphismNilpotentQuotient` to compare the images of *elm1* and *elm2* in a nilpotent quotient of the group. The implemented method successively increases the class of the considered quotient until the images differ. Hence, this method may not terminate and it will only determine whether the elements *elm1* and *elm2* are different.

5 ▶ **AsLpGroup(** *G* **)**                                                                                         O

returns an ascending *L*-presentation for a finitely presented group *G* or for a free group *G*.

6 ▶ **IsomorphismLpGroup(** *G* **)**                                                                               O

returns an isomorphism from a finitely presented group *G* or from a free group *G* to the *L*-presented group obtained from the method `AsLpGroup`.

7 ▶ **Display(** *G* **)**                                                                                            O

prints the *L*-presentation of *G*.

```
gap> F:=FreeGroup( 2 );
<free group on the generators [ f1, f2 ]>
gap> G:=F/[ F.1^2, F.2^2, Comm( F.1, F.2 ) ];
<fp group on the generators [ f1, f2 ]>
gap> IsomorphismLpGroup( G );
[ f1, f2 ] -> [ f1, f2 ]
gap> Range(last);
<L-presented group on the generators [ f1, f2 ]>
gap> Display(last);
generators = [ f1, f2 ]
fixed relators = [ ]
endomorphism = [
IdentityMapping( <free group on the generators [ f1, f2 ]> ) ]
iterated relators = [
f1^2,
f2^2,
f1^-1*f2^-1*f1*f2 ]
```

# 3 Nilpotent Quotients of L-presented groups

Our nilpotent quotient algorithm for finitely $L$-presented groups generalizes Nickel's algorithm for finitely presented groups; see [Nic96]. It determines a nilpotent presentation for the lower central series quotient of an invariantly $L$-presented group. A nilpotent presentation is a polycyclic presentation whose polycyclic series refines the lower central series of the group (see the description in the NQ-package for further details). In general, our algorithm determines a polycyclic presentation for the nilpotent quotient of an arbitrary finitely $L$-presented group. For further details on our algorithm we refer to [BEH08] or to the diploma thesis [Har08].

## 3.1 New methods for L-presented groups

1 ▶ NilpotentQuotient( $LpGroup$[, $c$] )                                           O

returns a polycyclic presentation for the class-$c$ quotient $LpGroup/\gamma_{c+1}(LpGroup)$ if $c$ is specified. If $c$ is not given, this method attempts to compute the largest nilpotent quotient of $LpGroup$ and will terminate only if $LpGroup$ has a largest nilpotent quotient.

The following example computes the class-5 quotient of the Grigorchuk group.

```
gap> G := ExamplesOfLPresentations( 1 );;
gap> H := NilpotentQuotient( G, 5 );
Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> lcs := LowerCentralSeries( H );
[ Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ],
  Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2 ],
  Pcp-group with orders [ 2, 2, 2, 2, 2 ], Pcp-group with orders [ 2, 2, 2 ],
  Pcp-group with orders [ 2, 2 ], Pcp-group with orders [  ] ]
gap> List( [ 1..5 ], x -> lcs[ x ] / lcs[ x+1 ] );
[ Pcp-group with orders [ 2, 2, 2 ], Pcp-group with orders [ 2, 2 ],
  Pcp-group with orders [ 2, 2 ], Pcp-group with orders [ 2 ],
  Pcp-group with orders [ 2, 2 ] ]
```

2 ▶ LargestNilpotentQuotient( $LpGroup$ )                                            A

returns the largest nilpotent quotient of the $L$-presented group $LpGroup$ if it exists. It uses the method NilpotentQuotient described above. If $LpGroup$ has no largest nilpotent quotient, this method will not terminate.

3 ▶ NqEpimorphismNilpotentQuotient( $LpGroup$[, $c$] )                                O
  ▶ NqEpimorphismNilpotentQuotient( $LpGroup$[, $PcpGroup$] )                         O

In the first case, this method returns an epimorphism from the $L$-presented group $LpGroup$ onto its class-$c$ quotient $LpGroup/\gamma_{c+1}(LpGroup)$ if $c$ is specified. If $c$ is not given, this method attempts to compute an epimorphism onto the largest nilpotent quotient of $LpGroup$. If $LpGroup$ does not have a largest nilpotent quotient, this method will not terminate.

If a pcp-group $PcpGroup$ is given as additional parameter, then $PcpGroup$ has to be a nilpotent quotient of $LpGroup$. The method computes an epimorphism from the $L$-presented group $LpGroup$ onto $PcpGroup$.

The following example computes an epimorphism from the Grigorchuk group onto its class-5, class-7, and class-10 quotient.

```
gap> G := ExamplesOfLPresentations( 1 );
<L-presented group on the generators [ a, b, c, d ]>
gap> epi := NqEpimorphismNilpotentQuotient( G, 5 );
[ a, b, c, d ] -> [ g1, g2*g3, g2, g3 ]
gap> H := Image( epi );
Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> NilpotencyClassOfGroup( H );
5
gap> H := NilpotentQuotient( G, 7 );
Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> NilpotentQuotient( G, 10 );
Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> NqEpimorphismNilpotentQuotient( G, H );
[ a, b, c, d ] -> [ g1, g2*g3, g2, g3 ]
gap> Image( last );
Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
```

4 ▶ **AbelianInvariants(** *LpGroup* **)**                                                                      O

computes the abelian invariants of the *L*-presented group *LpGroup*. It uses the operation `NilpotentQuotient` described above (see 3.1.1).

```
gap> G := ExamplesOfLPresentations( 1 );;
gap> AbelianInvariants( G );
[ 2, 2, 2 ]
```

## 3.2 A brief description of the algorithm

In the following we give a brief description of the nilpotent quotient algorithm for an arbitrary finitely *L*-presented group. For further details, we refer to [BEH08] and the diploma thesis [Har08] .

Let $(S, Q, \Phi, R)$ be a finite *L*-presentation defining the *L*-presented group $G$ and let $(S, Q', \Phi, R)$ be an underlying invariant *L*-presentation. Write $\bar{G}$ for the invariantly *L*-presented group defined by $(S, Q', \Phi, R)$.

The first step in computing a polycyclic presentation for $G/\gamma_c(G)$ is to determine a nilpotent presentation for $\bar{G}/\gamma_c(\bar{G})$. This will be done by induction on $c$. The induction step of our algorithm generalizes the induction step of Nickel's algorithm which mainly relies on Hermite normal form computations. In order to use this rather fast linear algebra, we must require the group to be invariantly *L*-presented. Therefore, the fixed relators must be handled separately by reducing to an underlying invariant *L*-presentation first.

The induction step of our algorithm then returns a nilpotent presentation $H$ for the quotient $\bar{G}/\gamma_c(\bar{G})$ and an epimorphism $\delta\colon \bar{G} \to H$. Both are used to determine a polycyclic presentation for the nilpotent quotient $G/\gamma_c(G)$ using an extension $\delta'\colon F_S \to H$ of the epimorphism $\delta$. The quotient $G/\gamma_c(G)$ is isomorphic to the factor group $H/\langle Q^{\delta'}\rangle^H$. We use the Polycyclic-package to compute a polycyclic presentation for $H/\langle Q^{\delta'}\rangle^H$.

The efficiency of this general approach depends on the underlying invariant *L*-presentation $(S, Q', \Phi, R)$. The set of fixed relators $Q'$ should be as large as possible. Otherwise, the nilpotent quotient $H$ can be large even if the nilpotent quotient $G/\gamma_c(G)$ is rather small.

The following example demonstrates the different behavior of our nilpotent quotient algorithm for the Grigorchuk group with its finite *L*-presentation

$$\Big(\{a, c, b, d\}, \{a^2, b^2, c^2, d^2, bcd\}, \{\sigma\}, \{[d, d^a], [d, d^{acaca}]\}\Big).$$

This latter *L*-presentation is obviously an invariant *L*-presentation. Hence, we can either use the property `IsInvariantLPresentation` or the attribute `UnderlyingInvariantLPresentation`. First, one has to construct the group as described in Section 2.1:

```
gap> F := FreeGroup( "a", "b", "c", "d" );
<free group on the generators [ a, b, c, d ]>
gap> AssignGeneratorVariables( F );
#I  Assigned the global variables [ a, b, c, d ]
gap> rels := [ a^2, b^2, c^2, d^2, b*d*c ];;
gap> endos := [ GroupHomomorphismByImagesNC( F, F, [ a, b, c, d ], [ c^a, d, b, c ] ) ];;
gap> itrels := [ Comm( d, d^a ), Comm( d, d^(a*c*a*c*a) ) ];;
gap> G := LPresentedGroup( F, rels, endos, itrels );
<L-presented group on the generators [ a, b, c, d ]>
gap> List( rels, x -> x^endos[1] );
[ a^-1*c^2*a, d^2, b^2, c^2, d*c*b ]
```

The property `IsInvariantLPresentation` can be set manually using `SetInvariantLPresentation`.

```
gap> SetIsInvariantLPresentation( G, true );
gap> NilpotentQuotient( G, 4 );
Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> StringTime( time );
" 0:00:00.032"
```

On the other hand, one can use the attribute `UnderlyingInvariantLPresentation` as follows.

```
gap> U := LPresentedGroup( F, rels, endos, itrels );
<L-presented group on the generators [ a, b, c, d ]>
gap> SetUnderlyingInvariantLPresentation( G, U );
gap> NilpotentQuotient( G, 4 );
Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> StringTime( time );
" 0:00:00.028"
```

For saving memory the first method should be preferred in this case. In general, the *L*-presentation is not invariant (or not known to be invariant) and thus the underlying invariant *L*-presentation has fewer fixed relators than the group *G* itself. In this case, the second method is the method of choice.

There is a brute-force method implemented for the operation `UnderlyingInvariantLPresentation` which works quite well on the `ExamplesOfLPresentations`. However, in the worst case, this method will return the underlying ascending *L*-presentation. The following example shows the influence of this choice to the runtime of the nilpotent quotient algorithm. After defining the group *G* as above, we set the attribute `UnderlyingInvariantLPresentation` as follows.

```
gap> SetUnderlyingInvariantLPresentation( G, UnderlyingAscendingLPresentation( G ) );
gap> NilpotentQuotient( G, 4 );
Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> StringTime( time );
" 0:00:02.700"
```

# 4 The underlying functions

## 4.1 Nilpotent Quotient Systems for invariant L-presentations

For an invariantly $L$-presented group $G$, our algorithm computes a nilpotent presentation for $G/\gamma_{c+1}(G)$ by computing a *weighted nilpotent quotient system* for $G/G'$ and extending it inductively to a weighted nilpotent quotient system for $G/\gamma_{c+1}(G)$.

In the NQL package, a weighted nilpotent quotient system is a record containing the following entries:

*Lpres*
> the invariantly $L$-presented group $G$.

*Pccol*
> FromTheLeftCollector of the nilpotent quotient represented by this quotient system.

*Imgs*
> the images of the generators of the $L$-presented group $G$ under the epimorphism onto the nilpotent quotient *Pccol*. For each generator of $G$ there is an integer or a generator exponent list. If the image is an integer *int*, the image is a definition of the *int*-th generator of the nilpotent presentation *Pccol*.

*Epimorphism*
> an epimorphism from the $L$-presented group $G$ onto its nilpotent quotient *Pccol* with the images of the generators given by *Imgs*.

*Weights*
> a list of the weight of each generator of the nilpotent presentation *Pccol*.

*Definitions*
> the definition of each generator of *Pccol*. Each generator in the quotient system has a definition as an image or as a commutator of the form $[a_j, a_i]$ where $a_j$ and $a_i$ are generators of a certain weight. If the $i$-th entry is an integer, the $i$-th generator of *Pccol* has a definition as an image. Otherwise, the $i$-th entry is a 2-tuple $[k, l]$ and the $i$-th generator has a definition as commutator $[a_k, a_l]$.

A weighted nilpotent quotient system of an invariantly $L$-presented group can be computed with the following functions.

1 ▶ InitQuotientSystem( *LpGroup* )                                                            O

computes a weighted nilpotent quotient system for the abelian quotient of the $L$-presented group *LpGroup*.

2 ▶ ExtendQuotientSystem( *QS* )                                                              O

extends the weighted nilpotent quotient system *QS* for a class-$c$ quotient of an invariantly $L$-presented group to a weighted nilpotent quotient system of its class-$c + 1$ quotient.

```
gap> G := ExamplesOfLPresentations( 1 );
<L-presented group on the generators [ a, b, c, d ]>
gap> Q := InitQuotientSystem( G );
rec( Lpres := <L-presented group on the generators [ a, b, c, d ]>,
  Pccol := <<from the left collector with 3 generators>>,
  Imgs := [ 1, [ 2, 1, 3, 1 ], 2, 3 ], Epimorphism := [ a, b, c, d ] ->
    [ g1, g2*g3, g2, g3 ], Weights := [ 1, 1, 1 ], Definitions := [ 1, 3, 4 ]
 )
gap> ExtendQuotientSystem( Q );
rec( Lpres := <L-presented group on the generators [ a, b, c, d ]>,
  Pccol := <<from the left collector with 5 generators>>,
  Imgs := [ 1, [ 2, 1, 3, 1 ], 2, 3 ],
  Definitions := [ 1, 3, 4, [ 2, 1 ], [ 3, 1 ] ],
  Weights := [ 1, 1, 1, 2, 2 ], Epimorphism := [ a, b, c, d ] ->
    [ g1, g2*g3, g2, g3 ] )
```

## 4.2 Attributes of L-presented groups related with the nilpotent quotient algorithm

To avoid repeated extensions of a weighted nilpotent quotient system the largest known quotient system
is stored as an attribute of the invariantly *L*-presented group. For non-invariantly *L*-presented groups (or
groups which are not known to be invariantly *L*-presented) the known epimorphisms onto the nilpotent
quotients are stored as an attribute.

1 ▶ `NilpotentQuotientSystem(` *LpGroup* `)`                                                                    A

stores the largest known weighted nilpotent quotient system of an invariantly *L*-presented group.

```
gap> G := ExamplesOfLPresentations( 1 );;
gap> NilpotentQuotient( G, 5 );
Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> NilpotentQuotientSystem( G );
rec( Lpres := <L-presented group on the generators [ a, b, c, d ]>,
  Pccol := <<from the left collector with 10 generators>>,
  Imgs := [ 1, [ 2, 1, 3, 1 ], 2, 3 ],
  Definitions := [ 1, 3, 4, [ 2, 1 ], [ 3, 1 ], [ 4, 2 ], [ 4, 3 ], [ 7, 1 ],
      [ 8, 2 ], [ 8, 3 ] ], Weights := [ 1, 1, 1, 2, 2, 3, 3, 4, 5, 5 ],
  Epimorphism := [ a, b, c, d ] -> [ g1, g2*g3, g2, g3 ] )
gap> NilpotencyClassOfGroup( PcpGroupByCollectorNC( last.Pccol ) );
5
```

2 ▶ `NilpotentQuotients(` *LpGroup* `)`                                                                          A

stores all known epimorphisms onto the nilpotent quotients of *LpGroup*. The nilpotent quotients are acces-
sible by the operation `Range`.

```
gap> G:=ExamplesOfLPresentations( 3 );;
gap> HasIsInvariantLPresentation( G );
false
gap> NilpotentQuotient( G, 3 );
Pcp-group with orders [ 0, 2, 2, 2 ]
gap> NilpotentQuotients( G );
[ [ a, t, u ] -> [ g2, g1, g2 ], [ a, t, u ] -> [ g2, g1, g2 ],
  [ a, t, u ] -> [ g2, g1, g2 ] ]
gap> Range( last[2] );
```

```
    Pcp-group with orders [ 0, 2, 2 ]
```

The underlying invariant *L*-presentation has stored its largest weighted nilpotent quotient system as an attribute.

```
    gap> NilpotentQuotientSystem( UnderlyingInvariantLPresentation( G ) );
    rec( Lpres := <L-presented group on the generators [ a, t, u ]>,
      Pccol := <<from the left collector with 9 generators>>, Imgs := [ 1, 2, 3 ],
      Definitions := [ 1, 2, 3, [ 2, 1 ], [ 3, 2 ], [ 4, 1 ], [ 4, 2 ], [ 5, 2 ],
          [ 5, 3 ] ], Weights := [ 1, 1, 1, 2, 2, 3, 3, 3, 3 ],
      Epimorphism := [ a, t, u ] -> [ g1, g2, g3 ] )
```

## 4.3 The Info-Class InfoNQL

To get some information about the progress of the algorithm, one can use the info class `InfoNQL`.

1 ▶ `InfoNQL`

is the info class of the **NQL**-package. If the info-level is 1, the info-class gives further information on the progress of the nilpotent quotient algorithm for *L*-presented groups. The info-level 2 also includes some information on the runtime of our algorithm while the info-level 3 is mainly used for debugging-purposes. An example of such a session for the Grigorchuk group is shown below:

```
    gap> SetInfoLevel( InfoNQL, 1 );;
    gap> G:=ExamplesOfLPresentations( 1 );
    #I  The Grigorchuk group on 4 generators
    <L-presented group on the generators [ a, b, c, d ]>
    gap> NilpotentQuotient( G, 3 );
    #I  Class 1: 3 generators with relative orders: [ 2, 2, 2 ]
    #I  Class 2: 2 generators with relative orders: [ 2, 2 ]
    #I  Class 3: 2 generators with relative orders: [ 2, 2 ]
    Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2 ]
    gap> SetInfoLevel( InfoNQL, 2 );
    gap> NilpotentQuotient( G, 5 );
    #I  Time spent for spinning algo:  0:00:00.004
    #I  Class 4: 1 generators with relative orders: [ 2 ]
    #I  Runtime for this step  0:00:00.028
    #I  Time spent for spinning algo:  0:00:00.008
    #I  Class 5: 2 generators with relative orders: [ 2, 2 ]
    #I  Runtime for this step  0:00:00.036
    Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
```

2 ▶ `InfoNQL_MAX_GENS`

this global variable sets the limit of generators whose relative order will be shown on each step of the nilpotent quotient algorithm, if the info-level of `InfoNQL` is positive.

# 5 Subgroups of L-presented groups

As shown in [Harb] it is possible to deal with finite index subgroups of *L*-presented groups algorithmically. The NQL-package provides straightforward methods to deal with these subgroups.

## 5.1 Creating a subgroup of an L-presented group

There are two ways of defining subgroups of finite index of an *LpGroup*. The first is to define the subgroup by its generators while the second defines the subgroup by a coset-table. Generators of subgroup of the latter type can be computed with the usual Schreier-algorithm.

1 ▶ Subgroup( *G*, *gens* )                                                    F

creates the subgroup *U* of *G* generated by *gens*. The Parent value of *U* will be *G*.

For example, the branching subgroup of the Grigorchuk group can be defined as follows

```
gap> G := ExamplesOfLPresentations(1);;
gap> a := G.1;; b := G.2;; c := G.3;; d := G.4;;
gap> K := Subgroup( G, [ Comm( a, b ), Comm( b^a, d ), Comm( b, d^a ) ] );
Group([ a^-1*b^-1*a*b, b^-1*a^-1*d^-1*a*b*a^-1*d*a, a^-1*b^-1*a*d^-1*a^-1*b*a*d ])
```

2 ▶ SubgroupLpGroupByCosetTable( *G*, *Tab* )                                    O

creates the subgroup *U* of *G* which is represented by the coset-table *Tab*.

For instance, the branching subgroup of the Grigorchuk group can be defined by the following coset-table

```
gap> Tab := [ [ 2, 1, 6, 9, 10, 3, 11, 12, 4, 5, 7, 8, 15, 16, 13, 14 ],
  [ 2, 1, 6, 9, 10, 3, 11, 12, 4, 5, 7, 8, 15, 16, 13, 14 ],
  [ 3, 6, 1, 5, 4, 2, 8, 7, 10, 9, 12, 11, 14, 13, 16, 15 ],
  [ 3, 6, 1, 5, 4, 2, 8, 7, 10, 9, 12, 11, 14, 13, 16, 15 ],
  [ 4, 7, 5, 1, 3, 8, 2, 6, 13, 14, 15, 16, 9, 10, 11, 12 ],
  [ 4, 7, 5, 1, 3, 8, 2, 6, 13, 14, 15, 16, 9, 10, 11, 12 ],
  [ 5, 8, 4, 3, 1, 7, 6, 2, 14, 13, 16, 15, 10, 9, 12, 11 ],
  [ 5, 8, 4, 3, 1, 7, 6, 2, 14, 13, 16, 15, 10, 9, 12, 11 ] ]
gap> U := SubgroupLpGroupByCosetTable( G, Tab );
Group(<subgroup of L-presented group, no generators known>)
gap> U = K;
true
```

The generators of *U* can be computed with the Schreier-algorithm which is implemented in the method 'GeneratorsOfGroup'.

## 5.2 Computing the index of finite index subgroups

In principle, it is possible to compute the index of a finite index subgroup of an *LpGroup* [Harb]. The method reduces the case to certain finitely presented groups by applying only finitely many endomorphisms to the iterated relations. It then uses coset enumeration for finitely presented groups to compute an upper bound on the index of the subgroup. If the coset enumeration for finitely presented groups terminated, the method attempts to prove that the upper bound is sharp. For further details we refer to [Harb].

1 ▶ IndexInWholeGroup( *H* )                                                                                      M

attempts to compute the index of *H* in its parent group.

```
gap> G:=ExamplesOfLPresentations(1);;
gap> a := G.1;; b := G.2;; c := G.3;; d := G.4;;
gap> K := Subgroup( G, [ Comm(a,b), Comm( b, d^a ), Comm( b^a, d )] );;
gap> IndexInWholeGroup( K );
16
```

2 ▶ Index( *H*, *I* )                                                                                              M

attempts to compute the index of *I* in the subgroup *H*. The subgroup *I* must be contained in *H*.

```
gap> G:=ExamplesOfLPresentations(1);;
gap> a := G.1;; b := G.2;; c := G.3;; d := G.4;;
gap> K := Subgroup( G, [ Comm(a,b), Comm( b, d^a ), Comm( b^a, d )] );;
gap> KxK := Subgroup( G, [ Comm(b,d^a), Comm(b^a,d), Comm(d^a,c^(a*c)),
> Comm( d^(a*c), c^a ), Comm( d, c^(a*c*a) ), Comm( d^(a*c*a), c) ] );;
gap> Index( K, KxK );
4
```

3 ▶ CosetTableInWholeGroup( *H* )                                                                                 M

computes a coset-table for the subgroup *H* in its parent group.

```
gap> CosetTableInWholeGroup( K );
[ [ 2, 1, 6, 9, 10, 3, 11, 12, 4, 5, 7, 8, 15, 16, 13, 14 ],
  [ 2, 1, 6, 9, 10, 3, 11, 12, 4, 5, 7, 8, 15, 16, 13, 14 ],
  [ 3, 6, 1, 5, 4, 2, 8, 7, 10, 9, 12, 11, 14, 13, 16, 15 ],
  [ 3, 6, 1, 5, 4, 2, 8, 7, 10, 9, 12, 11, 14, 13, 16, 15 ],
  [ 4, 7, 5, 1, 3, 8, 2, 6, 13, 14, 15, 16, 9, 10, 11, 12 ],
  [ 4, 7, 5, 1, 3, 8, 2, 6, 13, 14, 15, 16, 9, 10, 11, 12 ],
  [ 5, 8, 4, 3, 1, 7, 6, 2, 14, 13, 16, 15, 10, 9, 12, 11 ],
  [ 5, 8, 4, 3, 1, 7, 6, 2, 14, 13, 16, 15, 10, 9, 12, 11 ] ]
```

## 5.3 Technical details

For performance issues the following global variables can be used to modify the behaviour of the coset enumeration:

1 ▶ NQL_TCSTART

defines the maximal word-length of endomorphisms in the free monoid which are applied to the iterated relations.

2 ▶ NQL_CosetEnumerator

defines the coset enumeration process used for finitely presented groups. It should be a function which take as input a subgroup *h* of a finitely presented group and it computes a coset table in the whole group. The default uses the following method of the ACE-package

```
function ( h )
    local  f, rels, gens;
    f := FreeGeneratorsOfFpGroup( Parent( h ) );
    rels := RelatorsOfFpGroup( Parent( h ) );
    gens := List( GeneratorsOfGroup( h ), UnderlyingElement );
    return ACECosetTable( f, rels, gens : silent := true,
        hard := true,
        max := 10 ^ 8,
        Wo := 10 ^ 8 );
```

If the ACE-package is not available, the library coset enumeration process is used.

# 6 Approximating the Schur multiplier

The algorithm in [Hara] approximates the Schur multiplier of an invariantly finitely $L$-presented group by the quotients in its Dwyer-filtration. This is implemented in the NQL-package and the following methods are available:

1 ▶ GeneratingSetOfMultiplier( *LpGroup* )                                                                      A

uses Tietze transformations for computing an equivalent set of relators for *LpGroup* so that a generating set for its Schur multiplier can be read off easily.

2 ▶ FiniteRankSchurMultiplier( *LpGroup*, *c* )                                                                 O

computes a finitely generated quotient of the Schur multiplier of *LpGroup*. The method computes the image of the Schur multiplier of *LpGroup* in the Schur multiplier of its class-$c$ quotient.

3 ▶ EndomorphismsOfFRSchurMultiplier ( *LpGroup*, *c* )                                                         O

computes a list of endomorphisms of the FiniteRankSchurMultiplier of *LpGroup*. These are the endomorphisms of the invariant $L$-presentation induced to FiniteRankSchurMultiplier.

4 ▶ EpimorphismCoveringGroups( *LpGroup*, *d*, *c* )                                                            O

computes an epimorphism of the covering group of the class-$d$ quotient onto the covering group of the class-$c$ quotient.

5 ▶ EpimorphismFiniteRankSchurMultiplier( *LpGroup*, *d*, *c* )                                                 O

computes an epimorphism of the $d$-th FiniteRankSchurMultiplier of the invariant *LpGroup* onto the $c$-th FiniteRankSchurMultiplier. Its restricts the epimorphism EpimorphismCoveringGroups to the corresponding finite rank multipliers.

6 ▶ ImageInFiniteRankSchurMultiplier( *LpGroup*, *c*, *elm* )                                                   F

computes the image of the free group element *elm* in the $c$-th FiniteRankSchurMultiplier. Note that *elm* must be a relator contained in the Schur multiplier of *LpGroup*; otherwise, the function fails in computing the image.


The following example tackels the Schur multiplier of the Grigorchuk group.

```
gap> G := ExamplesOfLPresentations( 1 );;
gap> gens := GeneratingSetOfMultiplier( G );
rec( FixedGens := [ b^-2*c^-2*d^-2*b*c*d*b*c*d ],
  IteratedGens := [ d^-1*a^-1*d^-1*a*d*a^-1*d*a,
      d^-1*a^-1*c^-1*a^-1*c^-1*a^-1*d^-1*a*c*a*c*a*d*a^-1*c^-1*a^-1*c^-1*a^
        -1*d*a*c*a*c*a ],
  BasisGens := [ a^2, b*c*d, b^-2*d^-2*b*c*d*b*c*d, b^-2*c^-2*b*c*d*b*c*d ],
  Endomorphisms := [ [ a, b, c, d ] -> [ a^-1*c*a, d, b, c ] ] )
gap> H := FiniteRankSchurMultiplier( G, 5 );
```

```
Pcp-group with orders [ 2, 2, 2 ]
gap> GeneratorsOfGroup( H );
[ g15, g17, g16 ]
gap> EndomorphismsOfFRSchurMultiplier( G, 5 );
[ [ g15, g16, g17 ] -> [ g15, id, g16 ] ]
gap> Kernel( last[1] );
Pcp-group with orders [ 2 ]
gap> GeneratorsOfGroup( last );
[ g16 ]
gap> EpimorphismFiniteRankSchurMultipliers( G, 5, 2 );
[ g15, g16, g17 ] -> [ g10, id, g13 ]
gap> Range( last ) = FiniteRankSchurMultiplier( G, 2 );
true
gap> Kernel( EpimorphismFiniteRankSchurMultipliers( G, 5, 2 ) );
Pcp-group with orders [ 2 ]
gap> GeneratorsOfGroup( last );
[ g16 ]
gap> Kernel( EpimorphismFiniteRankSchurMultipliers( G, 5, 2 ) ) =
> Kernel( EndomorphismsOfFRSchurMultiplier( G, 5 )[1] );
true
gap> ImageInFiniteRankSchurMultiplier( G, 5, gens.FixedGens[1] );
g15
gap> ImageInFiniteRankSchurMultiplier(G,5,Image(gens.Endomorphisms[1],
> gens.IteratedGens[1] ) );
g16
gap> ImageInFiniteRankSchurMultiplier(G,5,gens.IteratedGens[1] );
g17
```

# 7 Investigating the automorphism group

Let $G$ be a finitely generated group. Then the term $\gamma_c G$ of the lower central series is fully invariant subgroup of $G$. Thus every automorphism $\alpha \in \mathrm{Aut}(G)$ induces an automorphism $\varphi_c \in \mathrm{Aut}(G/\gamma_c G)$. We obtain a homomorphism $\nu_c \colon \mathrm{Aut}(G) \to \mathrm{Aut}(G/\gamma_c G)$, $\alpha \mapsto \alpha_c$. This homomorphism map the inner automorphism $\mathrm{Inn}(G)$ onto $\mathrm{Inn}(G/\gamma_c G)$ and thus we obtain a homomorphism

$$\nu_c \colon \mathrm{Out}(G) \to \mathrm{Out}(G/\gamma_c G)\cdot$$

Similar, for every $d \leq c$, we obtain a homomorphism $\mu_{c,d} \colon \mathrm{Out}(G/\gamma_c G) \to \mathrm{Out}(G/\gamma_d G)$. Since $\nu_d = \nu_c \circ \mu_{c,d}$, this yields that

$$\mathrm{im}(\nu_d) \leq \ldots \leq \mathrm{im}(\mu_{c,d}) \leq \mathrm{im}(\mu_{c-1,d}) \leq \ldots \leq \mathrm{im}(\mu_{d,d}) = \mathrm{Out}(G/\gamma_d G)\cdot$$

This sequence can be used to guess the shape of $\mathrm{im}(\nu_d)$ and therefore to guess the shape of $\mathrm{Out}(G)/\ker \nu_d$. The AutPGrp-Package can be used to compute the images $\mathrm{im}(\mu_{c,d})$ if the abelian quotient of $G$ is elementary abelian. For further details we refer to [EH].

1 ▶ `AutomorphismGroupSequence(` *PcpGroup* `)`                 F

if the abelianization of *PcpGroup* is elementary abelian, this method computes a list of the images of the outer automorphism group of $G/\gamma_c G$ in $\mathrm{Out}(G/\gamma_d G)$ for any $d \leq c$ with $\mathrm{Out}(G/\gamma_d G)$ being still solvable. More precisely, the entry **a[i][j]** denotes the image of $\mathrm{Out}(G/\gamma_{j+1} G)$ in $\mathrm{Out}(G/\gamma_{i+1} G)$.

In the following example we consider the nilpotent quotients of the Grigorchuk group and compute its outer automorphism group sequence.

```
gap> G := ExamplesOfLPresentations( 1 );;
gap> A := AutomorphismGroupSequence( G, 5 );;
[1,2]: ab [ [ 2, 1 ] ]
[1,3]: ab [ [ 2, 1 ] ]
[1,4]: ab [ [ 2, 1 ] ]
[1,5]: ab [ [ 2, 1 ] ]

[2,3]: ab [ [ 2, 1 ] ]
[2,4]: ab [ [ 2, 1 ] ]
[2,5]: ab [ [ 2, 1 ] ]

[3,4]: id [ 16, 11 ]
[3,5]: ab [ [ 2, 2 ] ]

[4,5]: ab [ [ 2, 2 ] ]
```

# 8 On a parallel nilpotent quotient algorithm

We included a parallel version of NQL's nilpotent quotient algorithm using the ParGap-package of GAP; see [Coo04]. In this chapter, we outline the basic usage of this parallel part of the NQL-package. For further details on the parallel GAP-sessions we refer to the ParGap-manual [Coo04]. We note that the ParGap-package has some bottlenecks in practice. Nevertheless the significant speed-up of our computations on a multiple-core system shows that this is a reasonable extension of the NQL-package.

For using the parallel version of the nilpotent quotient algorithm, you will need to install the ParGap-package as described in its manual [Coo04]. When using Version 1.1.2 of the ParGap-package, you will need to apply the following patch to `pargap/lib/masslave.g` as otherwise the ParGap-session may crash. On a linux machine you can simply use `patch < ../../nql/gap/pargap/patch` from within the directory `pargap/lib/`.

```
--- masslave.g 2001-11-16 13:17:04.000000000 +0100
+++ masslave.g 2009-05-06 12:20:19.000000000 +0200
@@ -467,8 +467,9 @@
   if Length(deltas)>1 then max2 := Maximum(max2, deltas[Length(deltas)-1]); fi;
   max1 := deltas[Length(deltas)];
   pos1 := Position( List(slaveArray, x->realtime-x.time), max1 );
-  if max1 > slaveTaskTimeFactor and max1 > 30
-     and slaveTaskTime[pos2].total > 60 then
+  if max1 > slaveTaskTimeFactor and
+     max1 > 30 and pos2 <> fail and
+     slaveTaskTime[pos2].total > 60 then
    Print("SLAVE ",pos1," SEEMS DEAD!!\n");
   fi;
 end);
```

Now, you are ready for creating a ParGap-session and you can load the NQL-package from within ParGap using `RequirePackage` as usual. The same methods as described previously are available. The following example shows the application of the `NilpotentQuotient`-method to the Grigorchuk group on a quad-core machine. Note that the significant speed-up of the nilpotent quotient algorithm is especially noticeable for large nilpotent quotients. This parallel version of NQL successfully computes some nilpotent quotients which normally took more than a month to complete.

```
GAP4, Version: 4.4.12 of 17-Dec-2008, i686-pc-linux-gnu-gcc
GAP4, Version: 4.4.12 of 17-Dec-2008, i686-pc-linux-gnu-gcc
GAP4, Version: 4.4.12 of 17-Dec-2008, i686-pc-linux-gnu-gcc
GAP4, Version: 4.4.12 of 17-Dec-2008, i686-pc-linux-gnu-gcc
GAP4, Version: 4.4.12 of 17-Dec-2008, i686-pc-linux-gnu-gcc
gap> TOPCnumSlaves;
4
gap> RequirePackage("NQL");
true
```

```
gap> G:=ExamplesOfLPresentations(1);
<L-presented group on the generators [ a, b, c, d ]>
gap> SetInfoLevel(InfoNQL,1);
gap> NilpotentQuotient(G,2);
#I  Class 1: 3 generators with relative orders: [ 2, 2, 2 ]
#I  Computing a polycyclic presentation for the covering group...
#I  Checking the consistency relations...
master -> 1: (AGGLOM_TASK): [ [ -3, 1 ], [ -3, 2 ], [ -2, 1 ], [ 2, -1 ],
  [ 3, -1 ] ]
master -> 2: (AGGLOM_TASK): [ [ 3, -2 ], [ 1 ], [ 2 ], [ 3 ] ]
1 -> master: [ [ 0, 0, 0, 0, 0, -2, 0 ], [ 0, 0, 0, 0, 0, 0, -2 ],
  [ 0, 0, 0, 0, -2, 0, 0 ], [ 0, 0, 0, 0, -2, 0, 0 ],
  [ 0, 0, 0, 0, 0, -2, 0 ] ]
2 -> master: [ [ 0, 0, 0, 0, 0, 0, -2 ], [ 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0, 0 ] ]
#I  Broadcasting the slaves...
#I  Inducing the endomorphisms...
master -> 1:  1
master -> 2:  2
master -> 3:  3
master -> 4:  4
3 -> master: [ 2, 1 ]
UPDATE: [ 3, [ 2, 1 ] ]
1 -> master: [ 2, 1, 8, 1 ]
UPDATE: [ 1, [ 2, 1, 8, 1 ] ]
2 -> master: [ 2, 1, 3, 1, 4, 1 ]
UPDATE: [ 2, [ 2, 1, 3, 1, 4, 1 ] ]
master -> 1:  5
master -> 2:  6
master -> 3:  7
4 -> master: [ 4, -1, 6, -1, 10, -1 ]
UPDATE: [ 4, [ 4, -1, 6, -1, 10, -1 ] ]
1 -> master: [ 6, 1, 8, 2 ]
UPDATE: [ 5, [ 6, 1, 8, 2 ] ]
2 -> master: [ 4, 2, 6, 1, 7, 1, 10, 1 ]
UPDATE: [ 6, [ 4, 2, 6, 1, 7, 1, 10, 1 ] ]
3 -> master: [ 6, 1 ]
UPDATE: [ 7, [ 6, 1 ] ]
master -> 1:  8
master -> 2:  9
master -> 3:  10
1 -> master: [ 10, 1 ]
UPDATE: [ 8, [ 10, 1 ] ]
2 -> master: [  ]
UPDATE: [ 9, [  ] ]
3 -> master: [ 10, -1 ]
UPDATE: [ 10, [ 10, -1 ] ]
#I  Broadcasting the slaves...
#I  Mapping the relations...
master -> 1:  1
master -> 2:  2
master -> 3:  3
```

```
master -> 4:  4
1 -> master: [ 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ]
2 -> master: [ 0, 0, 0, 2, 0, 1, 1, 0, 0, 1 ]
3 -> master: [ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 ]
4 -> master: [ 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 ]
master -> 1:  5
master -> 2:  6
master -> 3:  7
1 -> master: [ 0, 0, 0, 1, 0, 1, 1, 0, 0, 1 ]
2 -> master: [ 0, 0, 0, 0, 0, 0, 0, 0, 2, 0 ]
3 -> master: [ 0, 0, 0, 0, 0, 0, 0, 4, 6, 4 ]
#I  Start spinning...
#I  Extend the quotient system...
#I  Class 2: 2 generators with relative orders: [ 2, 2 ]
Pcp-group with orders [ 2, 2, 2, 2, 2 ]
```

Note that the only difference in the parallel version of the NQL-package is a parallel version of the operation ExtendQuotientSystem. This latter operation covers the induction step of the nilpotent quotient algorithm.

# Bibliography

[Bar03]  Laurent Bartholdi. Endomorphic presentations of branch groups. *J. Algebra*, 268:419–443, 2003.

[Bau71]  Gilbert Baumslag. A finitely generated, infinitely related group with trivial multiplicator. 5:131–136, 1971.

[BEH08]  Laurent Bartholdi, Bettina Eick, and René Hartung. A nilpotent quotient algorithm for certain infinitely presented groups and its applications. *Internat. J. Algebra Comput.*, 18(8):1321–1344, 2008.

[BG02]  Laurent Bartholdi and Rostislav I. Grigorchuk. On parabolic subgroups and Hecke algebras of some fractal groups. *Serdica Math. J.*, 28(1):47–90, 2002.

[BSV99]  A. M. Brunner, Said Sidki, and Ana Cristina Vieira. A just-nonsolvable torsion-free group defined on the binary tree. 211(1):99–114, 1999.

[BV05]  Laurent Bartholdi and Bálint Virág. Amenability via random walks. *Duke Math. J.*, 130(1):39–56, 2005.

[Coo04]  Gene Cooperman. *ParGAP*, 2004. A GAP4 package, see [GAP4].

[EH]  Bettina Eick and René Hartung. Investigating some self-similar groups via nilpotent quotients. Preprint.

[FG85]  Jacek Fabrykowski and Narain Gupta. On groups with sub-exponential growth functions. *J. Indian Math. Soc. (N.S.)*, 49(3-4):249–256 (1987), 1985.

[Gri80]  R.I. Grigorchuk. Burnside's problem on periodic groups. *Functional Analysis and its Applications*, 14:41–43, 1980.

[Gri83]  R. I. Grigorchuk. On the Milnor problem of group growth. *Dokl. Akad. Nauk SSSR*, 271(1):30–33, 1983.

[Gri98]  R. I. Grigorchuk. An example of a finitely presented amenable group that does not belong to the class EG. *Mat. Sb.*, 189(1):79–100, 1998.

[Gri99]  R. I. Grigorchuk. On the system of defining relations and the Schur multiplier of periodic groups generated by finite automata. In *Groups St. Andrews 1997 in Bath, I*, volume 260 of *London Math. Soc. Lecture Note Ser.*, pages 290–317. Cambridge Univ. Press, Cambridge, 1999.

[GZ02]  Rostislav Grigorchuk and Andrzej Zuk. On a torsion-free weakly branch group defined by a three state automaton. *Internat. J. Algebra Comput.*, 12(1–2):223–246, 2002.

[Hara]  René Hartung. Approximating the Schur multiplier of certain infinitely presented groups via nilpotent quotients. Preprint.

[Harb]  René Hartung. Coset enumeration for certain infinitely presented groups. Preprint.

[Har08]  René Hartung. *A nilpotent quotient algorithm for finitely L-presented groups*. Diploma thesis, University of Braunschweig, 2008.
http://www-public.tu-bs.de:8080/~y0019492/pub/index.html.

[Lys85]  I.G. Lysenok. A system of defining relations for a Grigorchuk group. *Mathematical Notes*, 38:784–792, 1985.

[Nic96]  Werner Nickel. Computing nilpotent quotients of finitely presented groups. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 25:175–191, 1996.

[Nic03]  Werner Nickel. *NQ*, 2003. A GAP4 package, see [GAP4].

[Sid87]  Said Sidki. On a 2-generated infinite 3-group: The presentation problem. *Journal of Algebra*, 110:13–23, 1987.

# 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".