NQ

A GAP 4 Package
computing nilpotent factor groups of finitely presented groups

Based on the ANU Nilpotent Quotient Program

Version 2.2

February 2007

Werner Nickel

Werner Nickel
— Email: nickel at mathematik.tu-darmstadt.de
— Homepage: http://www.mathematik.tu-darmstadt.de/ nickel
— Address: Fachbereich Mathematik, AG 2
TU Darmstadt
Schlossgartenstr. 7
64289 Darmstadt
Germany

Copyright

Acknowledgements

The development of this program was started while the author was supported by an Australian National University PhD scholarship and an Overseas Postgraduate Research Scholarship.

Further development of this program was done with support from the DFG-Schwerpunkt-Projekt "'Algorithmische Zahlentheorie und Algebra"'.

Over the years a number of people have made useful suggestions that found their way into the code: Mike Newman, Michael Vaughan-Lee, Joachim Neubüser, Charles Sims.

I thank Volkmar Felsch and Joachim Neubüser for their careful examination of the package prior to its release for GAP 4.

This documentation was prepared with the GAPDoc package by Frank Lübeck and Max Neunhöffer.

Contents

Chapter 1

Introduction

This package provides an interface between GAP 4 and the Australian National University Nilpotent Quotient Program (ANU NQ). The ANU NQ was implemented as part of the author's work towards his PhD at the Australian National University, hence the name of the program. The program takes as input a finite presentation of a group and successively computes factor groups modulo the terms of the lower central series of the group. These factor groups are computed in terms of polycyclic presentations.

The ANU NQ is implemented in the programming language C. The implementation has been developed in a Unix environment and Unix is currently the only operating system supported. It runs on a number of different Unix versions, e.g. Solaris and Linux.

For integer matrix computations it relies on the GNU MP [GNU02] package and requires this package to be installed on your system.

This package relies on the functionality for polycyclic groups provided by the GAP package polycyclic [EN02] and requires the package polycyclic to be installed as a GAP package on your computer system.

Comments, bug reports and suggestions are very welcome.

This manual contains references to parts of the GAP Reference Manual which are typeset in a slightly idiosyncratic way. The following example shows how such references are printed: 'For further information on creating a free group see `FreeGroup` (**Reference: FreeGroup**).' The text in bold face refers to the GAP Reference Manual.

Each item in the list of references at the end of this manual is followed by a list of numbers that specify the pages of the manual where the reference occurs.

Chapter 2

General remarks

In this chapter we define notation used throughout this manual and recollect basic facts about nilpotent groups. We also provide some background information about the functionality implemented in this package.

2.1 Commutators and the Lower Central Series

The *commutator* of two elements h_1 and h_2 of a group G is the element $h_1^{-1}h_2^{-1}h_1h_2$ and is denoted by $[h_1,h_2]$. It satisfies the equation $h_1h_2 = h_2h_1[h_1,h_2]$ and can be interpreted as the correction term that has to be introduced into a word if two elements of a group are interchanged. Iterated commutators are written in *left-normed fashion*: $[h_1,h_2,\ldots,h_{n-1},h_n] = [[h_1,h_2,\ldots,h_{n-1}],h_n]$.

The *lower central series* of G is defined inductively as $\gamma_1(G) = G, \gamma_i(G) = [\gamma_{i-1}(G),G]$ for $i \geq 2$. Each term in the lower central series is a normal (even fully invariant) subgroup of G. The factors of the lower central series are abelian groups. On each factor the induced action of G via conjugation is the trivial action.

The factor $\gamma_k(G)/\gamma_{k+1}(G)$ is generated by the elements $[g,h]\gamma_{k+1}(G)$, where g runs through a set of (representatives of) generators for $G/\gamma_2(G)$ and h runs through a set of (representatives of) generators for $\gamma_{k-1}(G)/\gamma_k(G)$. Therefore, each factor of the lower central series is finitely generated if G is finitely generated.

If one factor of the lower central series is finite, then all subsequent factors are finite. Then the exponent of the $k+1$-th factor is a divisor of the exponent of the k-th factor of the lower central series. In particular, the exponents of all factors of the lower central series are bounded by the exponent of the first finite factor of the lower central series.

2.2 Nilpotent groups

A group G is called *nilpotent* if there is a positive integer c such that all $(c+1)$-fold commutators are trivial in G. The smallest integer with this property is called the *nilpotency class* of G. In terms of the lower central series a group $G \neq 1$ has nilpotency class c if and only if $\gamma_c(G) \neq 1$ and $\gamma_{c+1}(G) = 1$.

Examples of nilpotent groups are finite p-groups, the group of unitriangular matrices over a ring with one and the factor groups of a free group modulo the terms of its lower central series.

Finiteness of a nilpotent group can be decided by the group's commutator factor group. A nilpotent group is finite if and only if its commutator factor group is finite. A group whose commutator factor group is finite can only have finite nilpotent quotient groups.

By refining the lower central series of a finitely generated nilpotent group one can obtain a (sub)normal series $G_1 > G_2 > ... > G_{k+1} = 1$ with cyclic (central) factors. Therefore, every finitely generated nilpotent group is *polycyclic*. Such a *polycyclic series* gives rise to a polycyclic generating sequence by choosing a generator a_i for each cyclic factor G_i/G_{i+1}. Let I be the set of indices such that G_i/G_{i+1} is finite. A simple induction argument shows that every element of the group can be written uniquely as a *normal word* $a_1^{e_1}\dots a_n^{e_n}$ with integers e_i and $0 \leq e_i < m_i$ for $i \in I$.

2.3 Nilpotent presentations

From a polycyclic generating sequence one can obtain a *polycyclic presentation* for the group. The following set of power and commutator relations is a defining set of relations. The *power relations* express $a_i^{m_i}$ in terms of the generators $a_{i+1},\dots,a_n$ whenever G_i/G_{i+1} is finite with order m_i. The *commutator relations* are obtained by expressing $[a_j,a_i]$ for $j > i$ as a word in the generators $a_{i+1},\dots,a_n$. If the polycyclic series is obtained from refining the lower central series, then $[a_j,a_i]$ is even a word in $a_{j+1},\dots,a_n$. In this case we obtain a nilpotent presentation.

To be more precise, a *nilpotent presentation* is given on a finite number of generators $a_1,\dots,a_n$. Let I be the set of indices such that G_i/G_{i+1} is finite. Let m_i be the order of G_i/G_{i+1} for $i \in I$. Then a nilpotent presentation has the form

$$\langle a,\dots,a_n | a_i^{m_i} = w_{ii}(a_{i+1},\dots,a_n) \text{ for } i \in I;\ [a_j,a_i] = w_{ij}(a_{j+1},\dots,a_n) \text{ for } 1 \leq i < j \leq n\rangle$$

Here, $w_{ij}(a_k,\dots,a_n)$ denotes a group word in the generators $a_k,\dots,a_n$.

In a group given by a polycyclic presentation each element in the group can be written as a *normal word* $a_1^{e_1}\dots a_n^{e_n}$ with $e_i \in \mathbb{Z}$ and $0 \leq e_i < m_i$ for $i \in I$. A procedure called *collection* can be used to convert an arbitrary word in the generators into an equivalent normal word. In general, the resulting normal word need not be unique. The result of collecting a word may depend on the steps chosen during the collection procedure. A polycyclic presentation with the property that two different normal words are never equivalent is called *consistent*. A polycyclic presentation derived from a polycyclic series as above is consistent. The following example shows an inconsistent polycyclic presentation

$$\langle a,b \mid a^2, b^a = b^2\rangle$$

as $b = baa = ab^2a = a^2b^4 = b^4$ which implies $b^3 = 1$. Here we have the equivalent normal words b^3 and the empty word. It can be proved that consistency can be checked by collecting a finite number of words in the given generating set in two essentially different ways and checking if the resulting normal forms are the same in both cases. See Chapter 9 of the book [Sim94] for an introduction to polycyclic groups and polycyclic presentations.

For computations in a polycyclic group one chooses a consistent polycyclic presentation as it offers a simple solution to the word problem: Equality between two words is decided by collecting both words to their respective normal forms and comparing the normal forms. Nilpotent groups and nilpotent presentations are special cases of polycyclic groups and polycyclic presentations. Nilpotent presentations allow specially efficient collection methods. The package Polycyclic provides algorithms to compute with polycyclic groups given by a polycyclic presentation.

However, inconsistent nilpotent presentations arise naturally in the nilpotent quotient algorithm. There is an algorithm based on the test words for consistency mentioned above to modify the arising inconsistent presentations suitably to obtain a consistent one for the same group.

2.4 A sketch of the algorithm

The input for the ANU NQ in its simplest form is a finite presentation $\langle X|R\rangle$ for a group G. The first step of the algorithm determines a nilpotent presentation for the commutator quotient of G. This is a presentation of the class-1 quotient of G. Call its generators $a_1,...,a_d$. It also determines a homomorphism of G onto the commutator quotient and describes it by specifying the image of each generator in X as a word in the a_i.

For the general step assume that the algorithm has computed a nilpotent presentation for the class-c quotient of G and that $a_1,...,a_d$ are the generators introduced in the first step of the algorithm. Furthermore, there is a map from X into the class-c quotient describing the epimorphism from G onto $G/\gamma_{c+1}(G)$.

Let $b_1,...b_k$ be the generators from the last step of the algorithm, the computation of $\gamma_c(G)/\gamma_{c+1}(G)$. This means that $b_1,...b_k$ generate $\gamma_c(G)/\gamma_{c+1}(G)$. Then the commutators $[b_j,a_i]$ generate $\gamma_{c+1}(G)/\gamma_{c+2}(G)$. The algorithm introduces new, central generators c_{ij} into the presentation, adds the relations $[b_j,a_i] = c_{ij}$ and modifies the existing relations by appending suitable words in the c_{ij}, called *tails*, to the right hand sides of the power and commutator relations. The resulting presentation is a nilpotent presentation for the *nilpotent cover* of $G/\gamma_{c+1}(G)$. The nilpotent cover is the largest central extension of $G/\gamma_{c+1}(G)$ generated by d elements. It is is uniquely determined up to isomorphism.

The resulting presentation of the nilpotent cover is in general inconsistent. Consistency is achieved by running the consistency test. This results in relations among the generators c_{ij} which can be used to eliminate some of those generators or introduce power relations. After this has been done we have a consistent nilpotent presentation for the nilpotent cover of $G/\gamma_{c+1}(G)$.

Furthermore, the nilpotent cover need not satisfy the relations of G. In other words, the epimorphism from G onto $G/\gamma_{c+1}(G)$ cannot be lifted to an epimorphism onto the nilpotent cover. Applying the epimorphism to each relator of G and collecting the resulting words of the nilpotent cover yields a set of words in the c_{ij}. This gives further relations between the c_{ij} which leads to further eliminations or modifications of the power relations for the c_{ij}.

After this, the inductive step of the ANU NQ is complete and a consistent nilpotent presentation for $G/\gamma_{c+2}(G)$ is obtained together with an epimorphism from G onto the class-$(c+1)$ quotient.

Chapter 11 of the book [Sim94] discusses a nilpotent quotient algorithm. A description of the implementation in the ANU NQ is contained in [Nic96]

2.5 Identical Relations

Let w be a word in free generators $x_1,\dots,x_n$. A group G satisfies the relation $w = 1$ *identically* if each map from $x_1,\dots,x_n$ into G maps w to the identity element of G. We also say that G satisfies the *identical relation* $w = 1$ or satisfies the *law* $w = 1$. In slight abuse of notation, we call the elements $x_1,\dots,x_n$ *identical* generators.

Common examples of identical relations are: A group of nilpotency class at most c satisfies the law $[x_1,\dots,x_{c+1}] = 1$. A group that satisfies the law $[x,y,\dots,y] = 1$ where y occurs n-times, is called an n-Engel group. A group that satisfies the law $x^d = 1$ is a group of exponent d.

To describe finitely presented groups that satisfy one or more laws, we extend a common notation for finitely presented groups by specifying the identical generators as part of the generator list, separated from the group generators by a semicolon: For example

$$\langle a,b,c;x,y|x^5,[x,y,y,y]\rangle$$

is a group on 3 generators a,b,c of exponent 5 satisfying the 3rd Engel law. The presentation above is equivalent to a presentation on 3 generators with an infinite set of relators, where the set of relators consists of all fifth powers of words in the generators and all commutators $[x,y,y,y]$ where x and y run through all words in the generators a,b,c. The standalone programme accepts the notation introduced above as a description of its input. In GAP 4 finitely presented groups are specified in a different way, see `NilpotentQuotient` (3.1.1) for a description.

This notation can also be used in words that mix group and identical generators as in the following example:

$$\langle a,b,c;x|[x,c],[a,x,x,x]\rangle$$

The first relator specifies a law which says that c commutes with all elements of the group. The second turns a into a third right Engel element.

An element a is called *a right n-th Engel element* or *a right n-Engel element* if it satisfies the commutator law $[a,x,...,x] = 1$ where the identical generator x occurs n-times. Likewise, an element b is called an *left n-th Engel element* or *left n-Engel element* if it satisfies the commutator law $[x,b,b,...b] = 1$.

Let G be a nilpotent group. Then G satisfies a given law if the law is satisfied by a certain finite set of instances given by Higman's Lemma, see [Hig59]. The ANU NQ uses Higman's Lemma to obtain a finite presentation for groups that satisfy one or several identical relations.

2.6 Expression Trees

Expressions involving commutators play an important role in the context of nilpotent groups. Expanding an iterated commutator produces a complicated and long expression. For example,

$$[x,y,z] = y^{-1}x^{-1}yxz^{-1}x^{-1}y^{-1}xyz.$$

Evaluating a commutator $[a,b]$ is done efficiently by computing the equation $(ba)^{-1}ab$. Therefore, for each commutator we need to perform two multiplications and one inversion. Evaluating $[x,y,z]$ needs four multiplications and two inversions. Evaluation of an iterated commutator with n components takes $2n-1$ multiplications and $n-1$ inversions. The expression on the right hand side above needs 9 multiplications and 5 inversions which is clearly much more expensive than evaluating the commutator directly.

Assuming that no cancellations occur, expanding an iterated commutator with n components produces a word with $2^{n+1}-2^{n-1}-2$ factors half of which are inverses. A similar effect occurs whenever a compact expression is expanded into a word in generators and inverses, for example $(ab)^{49}$.

Therefore, it is important not to expand expressions into a word in generators and inverses. For this purpose we provide a mechanism which we call here *expression trees*. An expression tree preserves the structure of a given expression. It is a (binary) tree in which each node is assigned an operation and whose leaves are generators of a free group or integers. For example, the expression $[(xy)^2,z]$ is stored as a tree whose top node is a commutator node. The right subtree is just a generator node (corresponding to z). The left subtree is a power node whose subtrees are a product node on the left and an integer node on the right. An expression tree can involve products, powers, conjugates and commutators. However, the list of available operations can be extended.

Evaluation of an expression tree is done recursively and requires as many operations as there are nodes in the tree. An expression tree can be evaluated in a specific group by the function `EvaluateExpTree` (3.2.2).

A presentation specified by expression trees is a record with the components `.generators` and `.relations`. See section 3.2 for a description of the functions that produce and manipulate expression trees.

Example
```
gap> RequirePackage( "nq" );
true
gap> gens := ExpressionTrees( 2 );
[ x1, x2 ]
gap> r1 := LeftNormedComm( [gens[1],gens[2],gens[2]] );
Comm( x1, x2, x2 )
gap> r2 := LeftNormedComm( [gens[1],gens[2],gens[2],gens[1]] );
Comm( x1, x2, x2, x1 )
gap> pres := rec( generators := gens, relations := [r1,r2] );
rec( generators := [ x1, x2 ],
  relations := [ Comm( x1, x2, x2 ), Comm( x1, x2, x2, x1 ) ] )
```

2.7 A word about the implementation

The ANU NQ is written in C, but not in ANSI C. I hope to make one of the next versions ANSI compliable. However, it uses a fairly restricted subset of the language so that it should be easy to compile it in new environments. The code is 64-bit clean. If you have difficulties with porting it to a new environment, let me know and I'll be happy to assist if time permits.

The program has two collectors: a simple collector from the left as described in [LGS90] and a combinatorial from the left collector as described in [VL90]. The combinatorial collector is always faster than the simple collector, therefore, it is the collector used by this package by default. This can be changed by modifying the global variable `NqDefaultOptions` (3.4.2).

In a polycyclic group with generators that do not have power relations, exponents may become arbitrarily large. Experience shows that this happens rarely in the computations done by the ANU NQ. Exponents are represented by 32-bit integers. The collectors perform an overflow check and abort the computation if an overflow occurred. In a GNU environment the program can be compiled using the 'long long' 64-bit integer type. For this uncomment the relevant line in src/Makefile and recompile the program.

As part of the step that enforces consistency and the relations of the group, the ANU NQ performs computations with integer matrices and converts them to Hermite Normal Form. The algorithm used here is a variation of the Kanan-Bachem algorithm based on the GNU multiple precision package GNU MP [GNU02]. Experience shows that the integer matrices are usually fairly sparse and Kanan-Bachem seems to be sufficient in this context. However, the implementation might benefit from a more efficient strategy for computing Hermite Normal Forms. This is a topic for further investigations.

As the program does not compute the Smith Normal Form for each factor of the lower central series but the Hermite Normal Form, it does not necessarily obtain a minimal generating set for each factor of the lower central series. The following is a simple example of this behaviour. We take the presentation

$$\langle x, y | x^2 = y \rangle$$

The group is clearly isomorphic to the additive group of the integers. Applying the ANU NQ to this presentation gives the following nilpotent presentation:

$$\langle A, B | A^2 = B, [B, A] \rangle$$

A nilpotent presentation on a minimal generating set would be the presentation of the free group on one generator:

$$\langle A| \rangle$$

2.8 The input format of the standalone

The input format for finite presentations resembles the way many people write down a presentation on paper. Here are some examples of presentations that the ANU NQ accepts:

```
< a, b | >                         # free group of rank 2
< a, b, c; x, y |
            [a,b,c],               # a left normed commutator
            [b,c,c,c]^6,           # another one raised to a power
            a^2 = c^-3*a^2*c^3,    # a relation
            a^(b*c) = a,           # a conjugate relation
            (a*[b,(a*c)])^6,       # something that looks complicated
            [x,y,y,y,y],           # an identical relation
            [c,x,x,x,x,x]          # c is a fifth right Engel element
>
```

A presentation starts with '<' followed by a list of generators separated by commas. Generator names are strings that contain only upper and lower case letters, digits, dots and underscores and that do not start with a digit. The list of generator names is separated from the list of relators/relations by the symbol '|'. The list of generators can be followed by a list of identical generators separated by a semicolon. Relators and relations are separated by commas and can be mixed arbitrarily. Parentheses can be used in order to group subexpressions together. Square brackets can be used in order to form left normed commutators. The symbols '*' and '^' can be used to form products and powers, respectively. The presentation finishes with the symbol '>'. A comment starts with the symbol '#' and finishes at the end of the line. The file src/presentation.c contains a complete grammar for the presentations accepted by the ANU NQ.

Typically, the input for the standalone is put into a file by using a standard text editor. The file can be passed as an argument to the function `NilpotentQuotient` (3.1.1). It is also possible to put a presentation in the standalone's input format into a string and use the string as argument for `NilpotentQuotient` (3.1.1).

Chapter 3

The Functions of the Package

3.1 Nilpotent Quotients of Finitely Presented Groups

3.1.1 NilpotentQuotient

◊ `NilpotentQuotient( [output-file,] fp-group[, id-gens][,][c] )` (function)
◊ `NilpotentQuotient( [output-file,] input-file[, c] )` (function)

The parameter `fp-group` is either a finitely presented group or a record specifying a presentation by expression trees (see section 2.6). The parameter `input-file` is a string specifying the name of a file containing a finite presentation in the input format (cf. section 2.8) of the ANU NQ. Such a file can be prepared by a text editor or with the help of the function `NqStringFpGroup` (3.3.2).

Let G be the group defined by `fp-group` or the group defined in `input-file`. The function computes a nilpotent presentation for $G/\gamma_{c+1}(G)$ if the optional parameter `c` is specified. If `c` is not given, then the function attempts to compute the largest nilpotent quotient of G and it will terminate only if G has a largest nilpotent quotient. See section 3.5 for a possibility to follow the progress of the computation.

The optional argument `id-gens` is a list of generators of the free group underlying the finitely presented group `fp-group`. The generators in this list are treated as identical generators. Consequently, all relations of the `fp-group` involving these generators are treated as identical relations for these generators.

In addition to the arguments explained above, the function accepts the following options as shown in the first example below:

- `group` This option can be used instead of the parameter `fp-group`.
- `input_string` This option can be used to specify a finitely presented group by a string in the input format of the standalone program.
- `input_file` This option specifies a file with input for the standalone program.
- `output_file` This option specifies a file for the output of the standalone.
- `idgens` This options specifies a list of identical generators.
- `class` This option specifies the nilpotency class up to which the nilpotent quotient will be computed.

The following example computes the class-5 quotient of the free group on two generators.

Example
```
gap> F := FreeGroup( 2 );
<free group on the generators [ f1, f2 ]>
gap> ## Equivalent to:  NilpotentQuotient( : group := F, class := 5 );
gap> ##                 NilpotentQuotient( F : class := 5 );
gap> H := NilpotentQuotient( F, 5 );
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
gap> lcs := LowerCentralSeries( H );;
gap> for i in [1..5] do Print( lcs[i] / lcs[i+1], "\n" ); od;
Pcp-group with orders [ 0, 0 ]
Pcp-group with orders [ 0 ]
Pcp-group with orders [ 0, 0 ]
Pcp-group with orders [ 0, 0, 0 ]
Pcp-group with orders [ 0, 0, 0, 0, 0, 0 ]
```

Note that the lower central series in the example is part of the data returned by the standalone program. Therefore, the execution of the function LowerCentralSeries takes no time.

The next example computes the class-4 quotient of the infinite dihedral group. The group is soluble but not nilpotent. The first factor of its lower central series is a Klein four group and all the other factors are cyclic or order 2.

Example
```
gap> F := FreeGroup( 2 );
<free group on the generators [ f1, f2 ]>
gap> G := F / [F.1^2, F.2^2];
<fp group on the generators [ f1, f2 ]>
gap> H := NilpotentQuotient( G, 4 );
Pcp-group with orders [ 2, 2, 2, 2, 2 ]
gap> lcs := LowerCentralSeries( H );;
gap> for i in [1..Length(lcs)-1] do
>       Print( AbelianInvariants(lcs[i] / lcs[i+1]), "\n" );
> od;
[ 2, 2 ]
[ 2 ]
[ 2 ]
[ 2 ]
gap>
```

In the following example identical generators are used in order to express the fact that the group is nilpotent of class 3. A group is nilpotent of class 3 if it satisfies the identical relation $[x_1,x_2,x_3,x_4] = 1$ (cf. Section 2.5). The result is the free nilpotent group of class 3 on two generators.

Example
```
gap> F := FreeGroup( "a", "b", "w", "x", "y", "z" );
<free group on the generators [ a, b, w, x, y, z ]>
gap> G := F / [ LeftNormedComm( [F.3,F.4,F.5,F.6] ) ];
<fp group of size infinity on the generators [ a, b, w, x, y, z ]>
gap> ## The following is equivalent to:
```

```
gap> ##   NilpotentQuotient( G : idgens := [F.3,F.4,F.5,F.6] );
gap> H := NilpotentQuotient( G, [F.3,F.4,F.5,F.6] );
Pcp-group with orders [ 0, 0, 0, 0, 0 ]
gap> NilpotencyClassOfGroup(H);
3
gap> LowerCentralSeries(H);
[ Pcp-group with orders [ 0, 0, 0, 0, 0 ], Pcp-group with orders [ 0, 0, 0 ],
  Pcp-group with orders [ 0, 0 ], Pcp-group with orders [  ] ]
```

The following example uses expression trees in order to specify the third Engel law for the free group on 3 generators.

Example

```
gap> et := ExpressionTrees( 5 );
[ x1, x2, x3, x4, x5 ]
gap> comm := LeftNormedComm( [et[1], et[2], et[2], et[2]] );
Comm( x1, x2, x2, x2 )
gap> G := rec( generators := et, relations := [comm] );
rec( generators := [ x1, x2, x3, x4, x5 ],
  relations := [ Comm( x1, x2, x2, x2 ) ] )
gap> H := NilpotentQuotient( G : idgens := [et[1],et[2]] );
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 4, 2, 2,
  0, 6, 6, 0, 0, 2, 10, 10, 10 ]
gap> TorsionSubgroup( H );
Pcp-group with orders [ 2, 2, 2, 2, 2, 2, 2, 10, 10, 10 ]
gap> lcs := LowerCentralSeries( H );;
gap> NilpotencyClassOfGroup( H );
5
gap> for i in [1..5] do Print( lcs[i] / lcs[i+1], "\n" ); od;
Pcp-group with orders [ 0, 0, 0 ]
Pcp-group with orders [ 0, 0, 0 ]
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0 ]
Pcp-group with orders [ 2, 4, 2, 2, 0, 6, 6, 0, 0, 2 ]
Pcp-group with orders [ 10, 10, 10 ]
gap> for i in [1..5] do Print( AbelianInvariants(lcs[i]/lcs[i+1]), "\n" ); od;
[ 0, 0, 0 ]
[ 0, 0, 0 ]
[ 0, 0, 0, 0, 0, 0, 0, 0 ]
[ 2, 2, 2, 2, 2, 2, 2, 0, 0, 0 ]
[ 10, 10, 10 ]
```

The example above also shows that the relative orders of an abelian polycyclic group need not be the abelian invariants (elementary divisors) of the group. Each zero corresponds to a generator of infinite order. The number of zeroes is always correct.

3.1.2 NilpotentEngelQuotient

◊ `NilpotentEngelQuotient( [output-file,] fp-group, n[, id-gens][,][c] )` (function)
◊ `NilpotentEngelQuotient( [output-file,] input-file, n[, c] )` (function)

This function is a special version of `NilpotentQuotient` (3.1.1) which enforces the n-th Engel identity on the nilpotent quotients of the group specified by `fp-group` or by `input-file`. It accepts the same options as `NilpotentQuotient`.

The Engel condition can also be enforced by using identical generators and the Engel law and `NilpotentQuotient` (3.1.1). See the examples there.

The following example computes the relatively free fifth Engel group on two generators, determines its (normal) torsion subgroup and computes the corresponding quotient group. The quotient modulo the torsion subgroup is torsion-free. Therefore, there is a nilpotent presentation without power relations. The example computes a nilpotent presentation for the torsion free factor group through the upper central series. The factors of the upper central series in a torsion free group are torsion free. In this way one obtains a set of generators of infinite order and the resulting nilpotent presentation has no power relations.

Example
```
gap> G := NilpotentEngelQuotient( FreeGroup(2), 5 );
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 10,
  0, 0, 30, 0, 3, 3, 10, 2, 0, 6, 0, 0, 30, 2, 0, 9, 3, 5, 2, 6, 2, 10, 5, 5,
  2, 0, 3, 3, 3, 3, 3, 5, 5, 3, 3 ]
gap> NilpotencyClassOfGroup(G);
9
gap> T := TorsionSubgroup( G );
Pcp-group with orders [ 3, 3, 2, 2, 3, 3, 2, 9, 3, 5, 2, 3, 2, 10, 5, 2, 3,
  3, 3, 3, 3, 5, 5, 3, 3 ]
gap> IsAbelian( T );
true
gap> AbelianInvariants( T );
[ 3, 3, 3, 3, 3, 3, 3, 3, 30, 30, 30, 180, 180 ]
gap> H := G / T;
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 10,
  0, 0, 30, 0, 5, 0, 2, 0, 0, 10, 0, 2, 5, 0 ]
gap> H := PcpGroupBySeries( UpperCentralSeries(H), "snf" );
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0 ]
gap> ucs := UpperCentralSeries( H );;
gap> for i in [1..NilpotencyClassOfGroup(H)] do
>        Print( ucs[i]/ucs[i+1], "\n" );
> od;
Pcp-group with orders [ 0, 0 ]
Pcp-group with orders [ 0 ]
Pcp-group with orders [ 0, 0 ]
Pcp-group with orders [ 0, 0, 0 ]
Pcp-group with orders [ 0, 0, 0, 0, 0, 0 ]
Pcp-group with orders [ 0, 0, 0, 0 ]
Pcp-group with orders [ 0, 0 ]
Pcp-group with orders [ 0, 0, 0 ]
```

3.1.3 NqEpimorphismNilpotentQuotient

◊ `NqEpimorphismNilpotentQuotient( [output-file,] fp-group[, id-gens][,][c] )`
(function)

This function computes an epimorphism from the group G given by the finite presentation `fp-group` onto $G/\gamma_{c+1}(G)$. If `c` is not given, then the largest nilpotent quotient of G is computed and an epimorphism from G onto the largest nilpotent quotient of G. If G does not have a largest nilpotent quotient, the function will not terminate if c is not given.

The optional argument `id-gens` is a list of generators of the free group underlying the finitely presented group `fp-group`. The generators in this list are treated as identical generators. Consequently, all relations of the `fp-group` involving these generators are treated as identical relations for these generators.

If identical generators are specified, then the epimorphism returned maps the group generated by the 'non-identical' generators onto the nilpotent factor group. See the last example below.

The function understands the same options as the function `NilpotentQuotient` (3.1.1).

Example

```
gap> F := FreeGroup(3);
<free group on the generators [ f1, f2, f3 ]>
gap> phi := NqEpimorphismNilpotentQuotient( F, 5 );
[ f1, f2, f3 ] -> [ g1, g2, g3 ]
gap> Image( phi, LeftNormedComm( [F.3, F.2, F.1] ) );
g12
gap> F := FreeGroup( "a", "b" );
<free group on the generators [ a, b ]>
gap> G := F / [ F.1^2, F.2^2 ];
<fp group on the generators [ a, b ]>
gap> phi := NqEpimorphismNilpotentQuotient( G, 4 );
[ a, b ] -> [ g1, g2 ]
gap> Image( phi, Comm(G.1,G.2) );
g3*g4
gap> F := FreeGroup( "a", "b", "u", "v", "x" );
<free group on the generators [ a, b, u, v, x ]>
gap> a := F.1;; b := F.2;; u := F.3;; v := F.4;; x := F.5;;
gap> G := F / [ x^5, LeftNormedComm( [u,v,v,v] ) ];
<fp group of size infinity on the generators [ a, b, u, v, x ]>
gap> phi := NqEpimorphismNilpotentQuotient( G : idgens:=[u,v,x], class:=5 );
[ a, b ] -> [ g1, g2 ]
gap> U := Source(phi);
Group([ a, b ])
gap> ImageElm( phi, LeftNormedComm( [U.1*U.2, U.2^-1,U.2^-1,U.2^-1,] ) );
id
```

Note that the last epimorphism is a map from the group generated by a and b onto the nilpotent quotient. The identical generators are used only to formulate the identical relator. They are not generators of the group G. Also note that the left-normed commutator above is mapped to the identity as G satisfies the specified identical law.

3.1.4 LowerCentralFactors

◊ `LowerCentralFactors( ... )` (function)

This function accepts the same arguments and options as `NilpotentQuotient` (3.1.1) and returns a list containing the abelian invariants of the central factors in the lower central series of the specified group.

Example

```
gap> LowerCentralFactors( FreeGroup(2), 6 );
[ [ 0, 0 ], [ 0 ], [ 0, 0 ], [ 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ]
```

3.2 Expression Trees

3.2.1 ExpressionTrees

◊ `ExpressionTrees( m[, prefix] )` (function)
◊ `ExpressionTrees( str1, str2, str3, ... )` (function)

The argument `m` must be a positive integer. The function returns a list with `m` expression tree symbols named x1, x2,... The optional parameter `prefix` must be a string and is used instead of `x` if present.

Alternatively, the function can be executed with a list of strings `str1`, `str2`, ... It returns a list of symbols with these strings as names.

The following operations are defined for expression trees: multiplication, inversion, exponentiation, forming commutators, forming conjugates.

Example

```
gap> t := ExpressionTrees( 3 );
[ x1, x2, x3 ]
gap> tree := Comm( t[1], t[2] )^3/LeftNormedComm( [t[1],t[2],t[3],t[1]] );
Comm( x1, x2 )^3/Comm( x1, x2, x3, x1 )
gap> t := ExpressionTrees( "a", "b", "x" );
[ a, b, x ]
gap> tree := Comm( t[1], t[2] )^3/LeftNormedComm( [t[1],t[2],t[3],t[1]] );
Comm( a, b )^3/Comm( a, b, x, a )
```

3.2.2 EvaluateExpTree

◊ `EvaluateExpTree( tree, symbols, values )` (function)

The argument `tree` is an expression tree followed by the list of those symbols `symbols` from which the expression tree is built up. The argument `values` is a list containing a constant for each symbol. The function substitutes each value for the corresponding symbol and computes the resulting value for `tree`.

Example

```
gap> F := FreeGroup( 3 );
<free group on the generators [ f1, f2, f3 ]>
gap> t := ExpressionTrees( "a", "b", "x" );
[ a, b, x ]
gap> tree := Comm( t[1], t[2] )^3/LeftNormedComm( [t[1],t[2],t[3],t[1]] );
Comm( a, b )^3/Comm( a, b, x, a )
```

```
gap> EvaluateExpTree( tree, t, GeneratorsOfGroup(F) );
f1^-1*f2^-1*f1*f2*f1^-1*f2^-1*f1*f2*f1^-1*f2^-1*f1*f2*f1^-1*f3^-1*f2^-1*f1^
-1*f2*f1*f3*f1^-1*f2^-1*f1*f2*f1*f2^-1*f1^-1*f2*f1*f3^-1*f1^-1*f2^-1*f1*f2*f3
```

3.3 Auxiliary Functions

3.3.1 NqReadOutput

◊ `NqReadOutput( stream )` (function)

The only argument `stream` is an output stream of the ANU NQ. The function reads the stream and returns a record that has a component for each global variable used in the output of the ANU NQ, see `NqGlobalVariables` (3.4.3).

3.3.2 NqStringFpGroup

◊ `NqStringFpGroup( fp-group[, idgens] )` (function)

The function takes a finitely presented group `fp-group` and returns a string in the input format of the ANU NQ. If the list `idgens` is present, then it must contain generators of the free group underlying the finitely presented group `FreeGroupOfFpGroup` (**Reference: FreeGroupOfFpGroup**). The generators in `idgens` are treated as identical generators.

Example

```
gap> F := FreeGroup(2);
<free group on the generators [ f1, f2 ]>
gap> G := F / [F.1^2, F.2^2, (F.1*F.2)^4];
<fp group on the generators [ f1, f2 ]>
gap> NqStringFpGroup( G );
"< x1, x2 |\n    x1^2,\n    x2^2,\n    x1*x2*x1*x2*x1*x2*x1*x2\n>\n"
gap> Print( last );
< x1, x2 |
    x1^2,
    x2^2,
    x1*x2*x1*x2*x1*x2*x1*x2
>
gap> PrintTo( "dihedral", last );
gap> ## The following is equivalent to:
gap> ##     NilpotentQuotient( : input_file := "dihedral" );
gap> NilpotentQuotient( "dihedral" );
Pcp-group with orders [ 2, 2, 2 ]
gap> Exec( "rm dihedral" );
gap> F := FreeGroup(3);
<free group on the generators [ f1, f2, f3 ]>
gap> H := F / [ LeftNormedComm( [F.2,F.1,F.1] ),
>               LeftNormedComm( [F.2,F.1,F.2] ), F.3^7 ];
<fp group on the generators [ f1, f2, f3 ]>
gap> str := NqStringFpGroup( H, [F.3] );
```

```
"< x1, x2; x3 |\n    x1^-1*x2^-1*x1*x2*x1^-1*x2^-1*x1^-1*x2*x1^2,\n    x1^-1*x\
2^-1*x1*x2^-1*x1^-1*x2*x1*x2,\n    x3^7\n>\n"
gap> NilpotentQuotient( : input_string := str );
Pcp-group with orders [ 7, 7, 7 ]
```

3.3.3 NqStringExpTrees

◊ NqStringExpTrees(fp-group[, idgens]) (function)

The function takes a finitely presented group fp-group given in terms of expression trees and returns a string in the input format of the ANU NQ. If the list idgens is present, then it must contain a sublist of the generators of the presentation. The generators in idgens are treated as identical generators.

Example

```
gap> x := ExpressionTrees( 2 );
[ x1, x2 ]
gap> rels := [x[1]^2, x[2]^2, (x[1]*x[2])^5];
[ x1^2, x2^2, (x1*x2)^5 ]
gap> NqStringExpTrees( rec( generators := x, relations := rels ) );
"< x1, x2 |\n    x1^2,\n    x2^2,\n    (x1*x2)^5\n>\n"
gap> Print( last );
< x1, x2 |
    x1^2,
    x2^2,
    (x1*x2)^5
>
gap> x := ExpressionTrees( 3 );
[ x1, x2, x3 ]
gap> rels := [LeftNormedComm( [x[2],x[1],x[1]] ),
>              LeftNormedComm( [x[2],x[1],x[2]] ), x[3]^7 ];
[ Comm( x2, x1, x1 ), Comm( x2, x1, x2 ), x3^7 ]
gap> NqStringExpTrees( rec( generators := x, relations := rels ) );
"< x1, x2, x3 |\n    [ x2, x1, x1 ],\n    [ x2, x1, x2 ],\n    x3^7\n>\n"
gap> Print( last );
< x1, x2, x3 |
    [ x2, x1, x1 ],
    [ x2, x1, x2 ],
    x3^7
>
```

3.3.4 NqElementaryDivisors

◊ NqElementaryDivisors(int-mat) (function)

The function ElementaryDivisorsMat (**Reference: ElementaryDivisorsMat**) only returns the non-zero elementary divisors of an integer matrix. This function computes the elementary divisors of `int-mat` and adds the appropriate number of zeroes in order to make it easier to recognize the isomorphism type of the abelian group presented by the integer matrix. At the same time ones are stripped from the list of elementary divisors.

3.4 Global Variables

3.4.1 NqRuntime

◊ `NqRuntime` (global variable)

This variable contains the number of milliseconds of runtime of the last call of ANU NQ.

Example

```
gap> NilpotentEngelQuotient( FreeGroup(2), 5 );
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 10,
  0, 0, 30, 0, 3, 3, 10, 2, 0, 6, 0, 0, 30, 2, 0, 9, 3, 5, 2, 6, 2, 10, 5, 5,
  2, 0, 3, 3, 3, 3, 3, 5, 5, 3, 3 ]
gap> NqRuntime;
18200
```

3.4.2 NqDefaultOptions

◊ `NqDefaultOptions` (global variable)

This variable contains a list of strings which are the standard command line options passed to the ANU NQ in each call. Modifying this variable can be used to pass additional options to the ANU NQ.

Example

```
gap> NqDefaultOptions;
[ "-g", "-p", "-C", "-s" ]
```

The option `-g` causes the ANU NQ to produce output in GAP-format. The option `-p` prevents the ANU NQ from listing the pc-presentation of the nilpotent quotient at the end of the calculation. The option `-C` invokes the combinatorial collector. The option `-s` is effective only in conjunction with options for computing with Engel identities and instructs the ANU NQ to use only semigroup words in the generators as instances of an Engel law.

3.4.3 NqGlobalVariables

◊ `NqGlobalVariables` (global variable)

This variable contains a list of strings with the names of the global variables that are used in the output stream of the ANU NQ. While the output stream is read, these global variables are assigned new values. To avoid overwriting these variables in case they contain values, their contents is saved before reading the output stream and restored afterwards.

3.5 Diagnostic Output

While the standalone program is running it can be asked to display progress information. This is done by setting the info class `InfoNQ` to 1 via the function `SetInfoLevel` (**Reference: SetInfoLevel**).

Example

```
gap> NilpotentQuotient(FreeGroup(2),5);
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
gap> SetInfoLevel( InfoNQ, 1 );
gap> NilpotentQuotient(FreeGroup(2),5);
#I  Class 1: 2 generators with relative orders  0 0
#I  Class 2: 1 generators with relative orders: 0
#I  Class 3: 2 generators with relative orders: 0 0
#I  Class 4: 3 generators with relative orders: 0 0 0
#I  Class 5: 6 generators with relative orders: 0 0 0 0 0 0
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
gap> SetInfoLevel( InfoNQ, 0 );
```

Chapter 4

Examples

4.1 Right Engel elements

An old problem in the context of Engel elements is the question: Is a right n-Engel element left n-Engel? It is known that the answer is no. For details about the history of the problem, see [NN94]. In this paper the authors show that for $n > 4$ there are nilpotent groups with right n-Engel elements no power of which is a left n-Engel element. The insight was based on computations with the ANU NQ which we reproduce here. We also show the cases $5 > n$.

Example

```
gap> RequirePackage( "nq" );
true
gap> ##  SetInfoLevel( InfoNQ, 1 );
gap> ##
gap> ##  setup calculation
gap> ##
gap> et := ExpressionTrees( "a", "b", "x" );
[ a, b, x ]
gap> a := et[1];; b := et[2];; x := et[3];;
gap>
gap> ##
gap> ##  define the group for n = 2,3,4,5
gap> ##
gap>
gap> rengel := LeftNormedComm( [a,x,x] );
Comm( a, x, x )
gap> G := rec( generators := et, relations := [rengel] );
rec( generators := [ a, b, x ], relations := [ Comm( a, x, x ) ] )
gap> ## The following is equivalent to:
gap> ##   NilpotentQuotient( : input_string := NqStringExpTrees( G, [x] ) )
gap> H := NilpotentQuotient( G, [x] );
Pcp-group with orders [ 0, 0, 0 ]
gap> LeftNormedComm( [ H.2,H.1,H.1 ] );
id
gap> LeftNormedComm( [ H.1,H.2,H.2 ] );
id
```

This shows that each right 2-Engel element in a finitely generated nilpotent group is a left 2-Engel element. Note that the group above is the largest nilpotent group generated by two elements, one of which is right 2-Engel. Every nilpotent group generated by an arbitrary element and a right 2-Engel element is a homomorphic image of the group H.

Example
```
gap> rengel := LeftNormedComm( [a,x,x,x] );
Comm( a, x, x, x )
gap> G := rec( generators := et, relations := [rengel] );
rec( generators := [ a, b, x ], relations := [ Comm( a, x, x, x ) ] )
gap> H := NilpotentQuotient( G, [x] );
Pcp-group with orders [ 0, 0, 0, 0, 0, 4, 2, 2 ]
gap> LeftNormedComm( [ H.1,H.2,H.2,H.2 ] );
id
gap> h := LeftNormedComm( [ H.2,H.1,H.1,H.1 ] );
g6^2*g7*g8
gap> Order( h );
4
```

The element h has order 4. In a nilpotent group without 2-torsion a right 3-Engel element is left 3-Engel.

Example
```
gap> rengel := LeftNormedComm( [a,x,x,x,x] );
Comm( a, x, x, x, x )
gap> G := rec( generators := et, relations := [rengel] );
rec( generators := [ a, b, x ], relations := [ Comm( a, x, x, x, x ) ] )
gap> H := NilpotentQuotient( G, [x] );
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 12, 0, 5, 10, 2, 0, 30,
  5, 2, 5, 5, 5, 5 ]
gap> LeftNormedComm( [ H.1,H.2,H.2,H.2,H.2 ] );
id
gap> h := LeftNormedComm( [ H.2,H.1,H.1,H.1,H.1 ] );
g9*g10^2*g11^10*g12^5*g13^2*g14^8*g15*g16^6*g17^10*g18*g20^4*g21^4*g22^2*g23^2
gap> Order( h );
60
```

The previous calculation shows that in a nilpotent group without $2,3,5$-torsion a right 4-Engel element is left 4-Engel.

Example
```
gap> rengel := LeftNormedComm( [a,x,x,x,x,x] );
Comm( a, x, x, x, x, x )
gap> G := rec( generators := et, relations := [rengel] );
rec( generators := [ a, b, x ], relations := [ Comm( a, x, x, x, x, x ) ] )
gap> H := NilpotentQuotient( G, [x], 9 );
Pcp-group with orders [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 30,
  0, 0, 30, 0, 3, 6, 0, 0, 10, 30, 0, 0, 0, 0, 30, 30, 0, 0, 3, 6, 5, 2, 0,
  2, 408, 2, 0, 0, 0, 10, 10, 30, 10, 0, 0, 0, 3, 3, 3, 2, 204, 6, 6, 0, 10,
  10, 10, 2, 2, 2, 0, 300, 0, 0, 18 ]
gap> LeftNormedComm( [ H.1,H.2,H.2,H.2,H.2,H.2 ] );
id
gap> h := LeftNormedComm( [ H.2,H.1,H.1,H.1,H.1,H.1 ] );;
gap> Order( h );
infinity
```

Finally, we see that in a torsion-free group a right 5-Engel element need not be a left 5-Engel element.

Chapter 5

Installation of the Package

Installation of the package requires the GNU multiple precision library (GNU MP [GNU02]). If this library is not available on your system, it has to be installed first. A copy of GNU MP can be obtained via anonymous ftp from many file servers around the world, for example: www.swox.com/gmp/. Installation of the ANU NQ is done in two steps. First the configure script is run:

Installation
```
./configure
```

Among other things it will check whether it succeeds in locating the GNU MP library and the corresponding include files. If configure reports no problems, the next step is to start the compilation:

Installation
```
make
```

If the configure script told you that it could not find the GNU MP include files or the GNU MP libraries, you have to run make with additional parameters. For example, if you have installed GNU MP yourself in your home directory, you would type something like the following

Installation
```
 make GNU_MP_INC=/home/me/gmp-4.1.1 GNU_MP_LIB=/home/me/gmp-4.1.1/.libs
```

A compiled version of the program named ‘nq’ is then placed into the directory ‘bin/<complicated name>’. The <complicated name> component encodes the operating system and the compiler used. This allows you to compile NQ on several architectures sharing the same files system.

If there are any warnings or even fatal error messages during the compilation process, please send a copy to the address at the end of this document together with information about your operating system, the compiler you used and any changes you might have made to the source code. I will have a look at your problems and try to fix them.

After the compilation is finished you can check if the ANU NQ is running properly on your system. Simply type

Installation
```
make test
```

The file runs some computations and compares their output with the output files in the directory ‘examples’. If testNq reports any errors, please follow the instruction that testNq prints out.

The installation is completed by compiling the manual of the package. This is done from within GAP.

Installation
```
gap> NqBuildManual();
Composing XML document . . .
Parsing XML document . . .
Checking XML structure . . .
LaTeX version and calling latex and pdflatex:
    writing LaTeX file, 3 x latex, bibtex and makeindex, 2 x pdflatex, dvips
Text version . . .
#I  first run, collecting cross references, index, toc, bib and so on . . .
#I  table of contents complete.
#I  producing the index . . .
#I  reading bibliography data files . . .
#I  writing bibliography . . .
#I  second run through document . . .
Writing manual.six file . . .
And finally the HTML version . . .
#I  first run, collecting cross references, index, toc, bib and so on . . .
#I  table of contents complete.
#I  producing the index . . .
#I  reading bibliography data files . . .
#I  writing bibliography . . .
#I  second run through document . . .
gap> ?NilpotentQuotient
Help: several entries match this topic - type ?2 to get match [2]

[1] ANU NQ: NilpotentQuotient
[2] ANU NQ: NilpotentQuotient
[3] Reference: NilpotentQuotientOfFpLieAlgebra
gap>
```

References

[EN02] Bettina Eick and Werner Nickel. Polycyclic. *http://www.mathematik.tu-darmstadt.de/ nickel/polycyclic/*, 2002. 4

[GNU02] *GNU MP*. http://www.swox.com/gmp/, 2002. 4, 9, 23

[Hig59] G. Higman. Some remarks on varieties of groups. *Quart. J. Math. Oxford*, 2(10):165–178, 1959. 8

[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. 9

[Nic96] Werner Nickel. Computing nilpotent quotients of finitely presented groups. In *Geometric and Computational Perspectives on Infinite Groups*, volume 25 of *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, pages 175–191, 1996. 7

[NN94] M. F. Newman and Werner Nickel. Engel elements in groups. *"J. Pure Appl. Algebra"*, 96:39–45, 1994. 21

[Sim94] C. C. Sims. *Computation with Finitely Presented Groups*. Cambridge University Press, 1994. 6, 7

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

Index