

**An algorithm for computing quotients
of prime-power order for finitely presented groups
and its implementation in GAP**

Frank Celler*, M.F. Newman⁺, Werner Nickel*, Alice C. Niemeyer⁻

December 1993

* Lehrstuhl D für Mathematik
RWTH Aachen
52056 Aachen
Germany

- Department of Mathematics
University of Western Australia
Nedlands, WA 6009
Australia

+ School of Mathematical Sciences
The Australian National University
ACT 0200
Australia

Abstract: We describe an algorithm for computing quotients of prime-power order for finitely presented groups and its implementation in GAP. We use the opportunity (given by the design of the GAP language) to give rather more detail about such implementations than is available outside programs. We also describe some of the impact of this implementation process on the design of the GAP kernel.

Contents

| | |
|--|-----------|
| 1 Introduction | 1 |
| 2 An outline of the Prime Quotient Algorithm | 4 |
| 2.1 Preliminaries | 4 |
| 2.2 A prime covering algorithm | 10 |
| 2.3 The Prime Quotient Algorithm | 17 |
| 3 The implementation of the prime quotient algorithm in GAP | 19 |
| 3.1 About GAP | 20 |
| 3.2 Problems with the implementation in GAP 2.4 | 21 |
| 3.3 Kernel changes | 24 |
| 3.3.1 <i>ag</i> -groups and presentations | 24 |
| 3.3.2 <i>pc</i> -presentations | 25 |
| 4 Documentation | 27 |
| 4.1 GAP Functions for the Prime Quotient Algorithm | 27 |
| 4.2 Internal Functions | 31 |
| 4.2.1 Composition Functions for Elements of Pcps | 38 |
| 5 Appendix 1 Some sample runs | 42 |
| 6 Appendix 2 The listing | 44 |
| Bibliography | 63 |

1 Introduction

Implementations of prime quotient algorithms have proved to be very useful in the analysis of p -groups. Examples of this are the study of Burnside groups (Havas and Newman 1980) and the listing of groups of order 256 (O'Brien 1991). The task of a prime quotient algorithm is to compute consistent power commutator presentations for certain finite quotients of prime-power order of a finitely presented group. It is desirable to have an efficient implementation of such an algorithm in all general purpose programming systems designed for the study of groups. The aim of this report is to describe such an algorithm in more detail than is available in the literature and to describe how it was implemented in the programming system GAP (Groups, Algorithms and Programming, (see Schönert *et al.*, 1993)).

The development of GAP began in Aachen in 1986. It is designed as a system with a small kernel of basic algorithms written in the programming language **C** and provides a language especially designed for the study of groups. Algorithms in GAP are generally written in the GAP language with time consuming parts supported by the kernel. In many group theoretic algorithms a significant amount of time is spent in the relevant “arithmetics”, for example in multiplying permutations in algorithms for permutation groups or multiplying elements (done by collecting words into normal words) in algorithms for polycyclic groups. Therefore the standard arithmetics are part of the kernel. The kernel also provides a garbage collector, thus algorithms written in the GAP language do not have to deal with the allocation and deallocation of storage. These features, together with debugging facilities, are intended to allow users to implement efficient algorithms quickly in the GAP language.

The prime quotient algorithm we discuss in this document is the one described by Havas and Newman (1980). It was formerly called the Canberra Nilpotent Quotient Algorithm but the term nilpotent quotient algorithm is now used for algorithms which compute quotients to terms of the lower central series. The algorithm was

originally implemented by Alford and Havas in FORTRAN (see Alford, Havas and Newman 1975). We refer to this program as the FORTRAN standalone. Over the years other implementations have arisen. The FORTRAN standalone was embedded into the system Cayley (Cannon, 1984). Cayley was translated into C in 1988 and thus a C-version of the prime quotient algorithm was obtained. This C-version was significantly improved by O'Brien in 1991. The Vaughan-Lee (1990) collector from the left has been added and polished by Vaughan-Lee and O'Brien. We refer to this program as the C-standalone.

We describe the algorithm in Section 2, its implementation in GAP in Section 3 and give the documentation in Section 4. We call the GAP implementation the Prime Quotient Algorithm (PQA).

In 1990 Nickel and Niemeyer implemented the algorithm in GAP 2.4. This version of the algorithm was more than 60 times slower than the FORTRAN standalone. The reason for the time difference lay to a great extent in the design of the polycyclic group module in GAP 2.4 (see Bischops 1989). After (electronic) discussions between them and Celler and Neubüser in Aachen, Celler adapted Bischops' soluble group module for inclusion in GAP 3.0 and modified it significantly to provide a test version on which to base a new implementation of the prime quotient algorithm in GAP 3.0. Niemeyer adapted the GAP 2.4 version of the prime quotient algorithm to GAP 3.0. Since the performance of the algorithm was still very slow, Nickel and Niemeyer, in 1990, worked out a proposal for a kernel data structure supporting the needs of the prime quotient algorithm. It was based on a detailed analysis of the time behavior of the algorithm and many discussions with Newman. The proposal motivated Celler to redesign the soluble group module, reimplement parts of it and add new features. The new kernel module made it possible to reach runtimes for the prime quotient algorithm in GAP which were within a factor of 2 of the FORTRAN and C-standalones. Some recent performance figures are given in Appendix 1.

This version is currently included in the GAP 3.2 release (1993) as its basic prime quotient algorithm. It is listed in Appendix 2. With the development of share libraries for GAP the C-standalone program is now available from within GAP. This provides a more flexible and sophisticated prime quotient algorithm in GAP when needed.

Apology This document has already been too long in the writing (as various clues in the text reveal). We believe it has reached a form where it can be useful for the interested reader. We are painfully aware of some of its shortcomings. However we believe that it is better to make the report public now rather than after a further inevitable delay given the dispersment of the authors and their other commitments. We apologise for the shortcomings and thank the people responsible for this series for allowing it appear in its current form. We would welcome suggestions for improvement. They should be sent to: Mike.Newman@maths.anu.edu.au

An electronic version of this report is available by anonymous ftp from: [maths.anu.edu.au](ftp://maths.anu.edu.au/pub/Papers/report9327.dvi) in the directory `pub/Papers` as `report9327.dvi`

Information about GAP can be obtained from:
gap@samson.math.rwth-aachen.de

2 An outline of the Prime Quotient Algorithm

This section describes the Prime Quotient Algorithm (PQA). We begin with some background material.

2.1 Preliminaries

In the following p is a prime. A p -group is a group of p -power order.

Definition 2.1 For a finite group G the *Frattini subgroup* $\Phi(G)$ is the intersection of all maximal subgroups of G .

The basic properties of the Frattini subgroup of a finite group G are well known and can be found for instance in Huppert (1967, III.3). The following theorem, known as Burnside's Basis Theorem, is important in the study of p -groups (see Huppert 1967, III.3.15).

Theorem 2.2 Let G be a finite p -group with $|G/\Phi(G)| = p^d$. Then every minimal generating set of G has exactly d elements.

Let G be a group of order p^n and let $G = G_0 \geq G_1 \geq \dots \geq G_n = \{1\}$ be a composition series for G . Choose elements $a_i \in G$ for $1 \leq i \leq n$ such that $G_{i-1} = \langle G_i, a_i \rangle$. Then $\mathcal{A} = \{a_1, \dots, a_n\}$ is a generating set for G and the set

$$\mathcal{R} = \{ a_i^p = v_{ii}, [a_k, a_j] = v_{jk} \mid 1 \leq i \leq n, 1 \leq j < k \leq n \},$$

where v_{jk} , for $k \geq j$, is a word in the generators a_{k+1}, \dots, a_n , is a defining set of relations for G . The presentation $\{\mathcal{A} \mid \mathcal{R}\}$ is a *power commutator presentation* for G . On the other hand, given a power commutator presentation $\{\mathcal{A} \mid \mathcal{R}\}$ it *exhibits*

the “composition” series $G = G_0 \geq G_1 \geq \cdots \geq G_n = \langle 1 \rangle$, where $G_{i-1} = \langle a_i, \dots, a_n \rangle$ for $1 \leq i \leq n$. Some of the factors G_{i-1}/G_i may be trivial, so the order of G is at most p^n . A word $w(a_1, \dots, a_n)$ in the generators is *normal* if it is of the form $a_1^{e_1} \cdots a_n^{e_n}$ with $0 \leq e_i < p$.

Two words u, v in \mathcal{A} are *equivalent* if they represent the same element of the group defined by $\{\mathcal{A} \mid \mathcal{R}\}$. The fundamental importance of power commutator presentations arises from the observation that given a word w in the generators \mathcal{A} , the power commutator presentation $\{\mathcal{A} \mid \mathcal{R}\}$ can be used to compute an equivalent normal word. This computation of an equivalent normal word is referred to as *collection* (for an account see for example Leedham-Green and Soicher, 1990, or Vaughan-Lee, 1990). It is assumed that the words v_{jk} in a power commutator presentation are normal.

In what follows “word” means semigroup word. If a word is not normal it has a minimal non-normal subword of the form a_i^p or of the form $a_j a_i$ with $j > i$. These minimal non-normal subwords are the left-hand sides of the relations in a power commutator presentation. Collection of a word consists of a sequence of steps each of which replaces a minimal non-normal subword by the right-hand side of the corresponding relation. For normal words the sequence is empty. Otherwise a first step is applied to the given non-normal word. Each further step is applied to the result of the previous step. The word resulting from each step is equivalent to the given word.

A fundamental result is that every collection (whatever steps it consists of) of a non-normal word results in a normal word after a finite number of steps (see for example Sims, to appear). In other words collection is an algorithm. The normal word resulting from collecting the word w will be denoted (w) , it may depend on the sequence of steps.

For example, let G be the group defined by the presentation

$$\{a, b \mid a^2, b^a = b^5, b^{16}\}.$$

Then $baa = b$ or $baa = ab^5a = a^2b^{25} = b^9$. This example shows that b^9 and b are equivalent words. The problems this raises will be discussed below.

There are various “uniform” methods for collecting a word. For instance, *collection from the left* always replaces the left-most minimal non-normal subword at each step. This is the basic collection method which is now used in implementations.

Multiplication of two elements of G amounts to computing a normal word for the product by collecting the concatenation of words representing the elements. Given a word w in \mathcal{A} one can find another word u in \mathcal{A} such that u represents the inverse of w in G in the following way. Initially consider the formal inverse u of w . Replace an occurrence in u of the inverse a_k^{-1} with least subscript k by $a_k^{p-1}v$, where v is the formal inverse of v_{kk} and repeat this step. This process terminates with u being a word in \mathcal{A} for the inverse of w . Thus inversion of a group element amounts to computing a word in \mathcal{A} which represents its inverse and then collecting it to a normal word.

As has been shown above, there may be more than one normal word representing a given group element. If each element is represented by a unique normal word, then the power commutator presentation is *consistent*. In this case two group elements are equal only if they are represented by the same normal word. If $\{\mathcal{A} \mid \mathcal{R}\}$ is consistent, then the order of the group it defines is p^n .

For mathematical and computational reasons, power commutator presentations with an added feature are considered. Let $\{\mathcal{A} \mid \mathcal{R}\}$ be a power commutator presentation for a finite p -group H with $\mathcal{A} = \{a_1, \dots, a_n\}$. Let d be the minimal number of generators required to generate H . The additional property is that there exists a d -element subset \mathcal{X} of \mathcal{A} such that \mathcal{X} generates H and for each generator $a_k \in \mathcal{A} \setminus \mathcal{X}$ there is at least one relation in \mathcal{R} of the form $[a_j, a_i] = a_k$ or of the form $a_i^p = a_k$. One of these relations is labelled the *definition* of a_k . The presentation $\{\mathcal{A} \mid \mathcal{R}\}$ with this property is called *labelled*. If G is a group with generating set $\{g_1, \dots, g_b\}$ and τ is an epimorphism of G onto H , we call τ a *labelled* epimorphism if each generator $a_k \in \mathcal{X}$ occurs as the *last* generator in the image under τ of at least one of the generators g_i of G . One of these images is labelled the *definition* of the generator a_k . If $\{\mathcal{A} \mid \mathcal{R}\}$ is a labelled power commutator presentation for H and τ a labelled epimorphism from G to H , then every generator in \mathcal{A} has a definition.

In a labelled power commutator presentation the fact that a relation is a definition is emphasised by using '=' instead of '=' in the relation. For a labelled epimorphism the symbol '=' is also used to specify a definition.

Definition 2.3 Let G be a group and p a prime. The series

$$G = \mathcal{P}_0(G) \geq \mathcal{P}_1(G) \geq \cdots \quad \text{with } \mathcal{P}_i(G) = [\mathcal{P}_{i-1}(G), G] (\mathcal{P}_{i-1}(G))^p$$

for $i \geq 1$ is the *lower exponent- p central series* of G . If there exists an integer $c \geq 0$ such that $\mathcal{P}_c(G) = \langle 1 \rangle$, then G is a p -group and the smallest such integer is called the *exponent- p class* of G .

If G is a p -group, then $\mathcal{P}_1(G)$ is the Frattini subgroup $\Phi(G)$. Let p^d be the order of the Frattini quotient, then d is the *generator number* of G .

Let $\{\mathcal{A} \mid \mathcal{R}\}$ be a power commutator presentation obtained from a composition series by refining the lower p -central series of a p -group G . Define a function ω from \mathcal{A} into the set $\{1, \dots, c\}$ mapping a to b if $a \in \mathcal{P}_{b-1}(G) \setminus \mathcal{P}_b(G)$. Then ω is a *weight function* and $\omega(a)$ is the *weight* of a . The elements of \mathcal{A} satisfy the following relations \mathcal{R} :

$$a_i^p = \prod_{a_k \in \mathcal{A}, \omega(a_k) > \omega(a_i)} a_k^{\alpha(i,i,k)}, \text{ for } 1 \leq i \leq n \text{ and}$$

$$[a_j, a_i] = \prod_{a_k \in \mathcal{A}, \omega(a_k) \geq \omega(a_j) + \omega(a_i)} a_k^{\alpha(i,j,k)}, \text{ for } 1 \leq i < j \leq n,$$

with $0 \leq \alpha(i, j, k) < p$. The power commutator presentation together with a weight function is a *weighted power commutator presentation* and is denoted by $\{\mathcal{A} \mid \mathcal{R}, \omega\}$.

The following result is due to Wamsley (1974) and was improved by Vaughan-Lee (1984). Recall that (w) is a normal word resulting from collecting w .

Theorem 2.4 *A power commutator presentation $\{\mathcal{A} \mid \mathcal{R}\}$ is consistent if and*

only if the following words collect to the empty word:

$$\begin{aligned}
& \left((a_k a_j) a_i \right) \left(a_k (a_j a_i) \right)^{-1} \quad \text{for } 1 \leq i < j < k \leq n, i \leq d, \\
& \left((a_k^p) a_j \right) \left(a_k^{p-1} (a_k a_j) \right)^{-1} \quad \text{for } 1 \leq j < k \leq n, j \leq d, \\
& \left((a_j a_i) a_i^{p-1} \right) \left(a_j (a_i^p) \right)^{-1} \quad \text{for } 1 \leq i < j \leq n, \\
& \left((a_i^p) a_i \right) \left(a_i (a_i^p) \right)^{-1} \quad \text{for } 1 \leq i \leq n.
\end{aligned}$$

The PQA takes as input a prime p , a finite presentation

$$\{g_1, \dots, g_b \mid r_1(g_1, \dots, g_b), \dots, r_m(g_1, \dots, g_b)\}$$

for a group G , and an integer c . The output is a labelled weighted consistent power commutator presentation for $G/\mathcal{P}_c(G)$ and a labelled epimorphism of G onto this quotient. In our implementation giving c is optional.

The PQA proceeds by computing power commutator presentations for the quotients $G/\mathcal{P}_i(G)$ in turn. In the first step it computes a labelled weighted consistent power commutator presentation for $G/\mathcal{P}_1(G) = G/\Phi(G)$ and an epimorphism from G onto $G/\mathcal{P}_1(G)$. This is a presentation on d generators which commute pairwise and each has order p . For positive i , assume a labelled weighted consistent power commutator presentation for the quotient $G/\mathcal{P}_i(G)$ and a labelled epimorphism of G onto this quotient have been computed. Now a labelled consistent power commutator presentation for $G/\mathcal{P}_{i+1}(G)$ and a labelled epimorphism of G onto this quotient are to be computed. The basic step takes as input:

- 1) the finite presentation for G ;
- 2) a labelled weighted consistent power commutator presentation for the factor group $K \cong G/\mathcal{P}_i(G)$;
- 3) a labelled epimorphism $\theta : G \rightarrow K$.

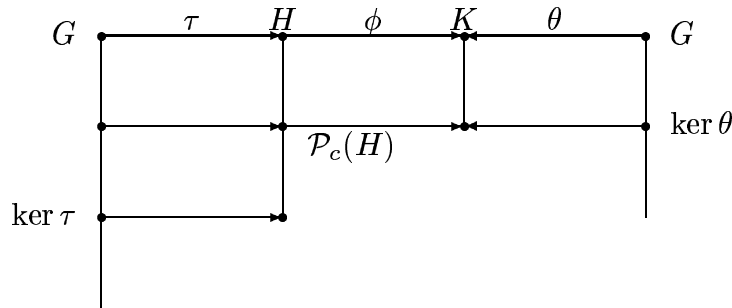
The output is:

- 1) a labelled weighted consistent power commutator presentation for the factor group $H \cong G/\mathcal{P}_{i+1}(G)$;

- 2) an epimorphism $\phi : H \rightarrow K$;
- 3) a labelled epimorphism $\tau : G \rightarrow H$ with $\tau\phi = \theta$.

If during the basic step it is discovered that $\mathcal{P}_i(G) = \mathcal{P}_{i+1}(G)$ then $G/\mathcal{P}_i(G)$ is the largest p -quotient of G and the algorithm terminates returning the consistent power commutator presentation for K and the epimorphism θ .

The basic step is illustrated by the following diagram, where the input is described on the right and the output is described on the left.



Definition 2.5 Let G be a finite p -group with generator number d and exponent- p class c . The group H is a *descendant* of G , if H has generator number d and $H/\mathcal{P}_c(H)$ is isomorphic to G . It is an *immediate descendant* of G if it has exponent- p class $c + 1$.

Theorem 2.6 Let G be a finite p -group with generator number d and exponent- p class c . There exists a finite group G^* with generator number d and exponent- p class at most $c + 1$ such that every immediate descendant of G is isomorphic to a quotient of G^* .

Let F be the free group of rank d and R a normal subgroup of F such that $F/R \cong G$. Define $R^* := [R, F]R^p$. Ascione (1979) and O'Brien (1990) prove the theorem by showing that the group $G^* := F/R^*$ has the required properties. Further, they prove that the isomorphism type of F/R^* depends only on G , not on R .

Definition 2.7 G^* is called the *p -covering group* of G . A power commutator presentation $\{\mathcal{A}^* \mid \mathcal{R}^*\}$ for G^* is called a *p -covering presentation* for the presentation $\{\mathcal{A} \mid \mathcal{R}\}$ of G , if \mathcal{A} is a subset of \mathcal{A}^* and every relation in \mathcal{R}^* differs from

some relation in \mathcal{R} or from the trivial relation $1 = 1$ only by a word in the elements of $\mathcal{A}^* \setminus \mathcal{A}$.

2.2 A prime covering algorithm

The task of a prime covering algorithm is to determine a labelled consistent power commutator presentation for the p -covering group of a p -group. Here we describe such an algorithm – the Prime Covering Algorithm.

- The input to the Prime Covering Algorithm is a labelled weighted consistent power commutator presentation for a finite p -group K of class c .
- The output is a labelled weighted consistent power commutator presentation for the p -covering group K^* of K .

Let $\{\mathcal{A} \mid \mathcal{R}, \omega\}$ be a labelled weighted consistent power commutator presentation for K . Consider those relations in \mathcal{R} which are not definitions and whose left-hand side is either a_i^p with $\omega(a_i) \leq c$ or $[a_j, a_i]$ with $\omega(a_j) + \omega(a_i) \leq c + 1$. Let s be the number of these relations. We obtain a power commutator presentation $\{\hat{\mathcal{A}} \mid \hat{\mathcal{R}}\}$ for K^* as follows. Let $\hat{\mathcal{A}}$ be the set $\{a_1, \dots, a_n, a_{n+1}, \dots, a_{n+s}\}$ and $\hat{\mathcal{R}}$ be the set of relations defined by:

- 1) initialise $\hat{\mathcal{R}}$ to contain all relations of \mathcal{R} which are definitions;
- 2) modify each non-definition $a_i^p = v_{ii}$ or $[a_k, a_j] = v_{jk}$ of \mathcal{R} to read $a_i^p = v_{ii} a_r$ or $[a_k, a_j] = v_{jk} a_r$ for some $r \in \{n+1, \dots, n+s\}$ where different non-definitions are modified by different a_r and add the modified relations to $\hat{\mathcal{R}}$.
- 3) add to $\hat{\mathcal{R}}$ the relations $[a_r, a_i] = 1$ for $n+1 \leq r \leq n+s$ and $1 \leq i < r$;
- 4) add to $\hat{\mathcal{R}}$ the relations $a_r^p = 1$ for $n+1 \leq r \leq n+s$.

Each generator in $\hat{\mathcal{A}}$ has a definition in terms of earlier generators, but this presentation is not necessarily labelled because some of the definitions may not have the required form. Note that the elements of $\hat{\mathcal{A}} \setminus \mathcal{A}$ are central and of order p ; we use this throughout the following proofs.

Define the function $\hat{\omega}$ from $\hat{\mathcal{A}}$ into the set $\{1, \dots, c+1\}$ by setting

$$\hat{\omega}(a_k) = \begin{cases} \omega(a_k) & \text{for } 1 \leq k \leq n, \\ \omega(a_i) + 1 & \text{if } k > n \\ & \text{and the definition of } a_k \text{ has left-hand side } a_i^p, \\ \omega(a_j) + \omega(a_i) & \text{if } k > n \\ & \text{and the definition of } a_k \text{ has left-hand side } [a_j, a_i]. \end{cases}$$

The function $\hat{\omega}$ need not be a weight function, because $\hat{\omega}(a)$ may not lie in the specified term of the \mathcal{P} -series; nevertheless we refer to $\hat{\omega}(a)$ for $a \in \hat{\mathcal{A}}$ as the *weight* of a .

Lemma 2.8 $\{\hat{\mathcal{A}} \mid \hat{\mathcal{R}}\}$ is a power commutator presentation for the p -covering group G^* of G .

Definition 2.9 An element of $\hat{\mathcal{A}} \setminus A$ is a *tail*. Let \mathcal{E} be the set of tails defined by either a power relation or by a commutator relation with left-hand side $[a_j, a_i]$ and $\hat{\omega}(a_i) = 1$. A word in $\hat{\mathcal{A}}$ is r -heavy if it has weight at least r .

We will show that all the tails can be expressed in terms of the elements of \mathcal{E} . Moreover, this can be done independently of how collection is implemented.

Lemma 2.10 For each $a \in \hat{\mathcal{A}} \setminus A$ there exists a relation

$$a = W(\mathcal{E}),$$

where $W(\mathcal{E})$ is a word in the elements of \mathcal{E} which is a consequence of the relations in $\hat{\mathcal{R}}$.

Proof: The statement follows from the next lemma by reverse induction on $\hat{\omega}(a)$. ■

For a commutator relation with left-hand side $[x, y]$ let $v_{[x,y]}$ denote the right-hand side of the commutator relation $[x, y]$ in \mathcal{R} and let $t_{[x,y]}$ be such that $v_{[x,y]}t_{[x,y]}$ is the right-hand side of the commutator relation $[x, y]$ in $\hat{\mathcal{R}}$. Thus $t_{[x,y]}$ is either the empty word or an element of $\hat{\mathcal{A}} \setminus A$.

Lemma 2.11 *Let $b \in \{2, \dots, c\}$. Assume that for each $\tilde{a} \in \hat{\mathcal{A}} \setminus \mathcal{A}$ with $\hat{\omega}(\tilde{a}) > b$ there exists a relation*

$$\tilde{a} = \tilde{W}(\mathcal{E})$$

which is a consequence of the relations in $\hat{\mathcal{R}}$. Then for each $a \in \hat{\mathcal{A}} \setminus \mathcal{A}$ with $\hat{\omega}(a) = b$ there exists a relation

$$a = W(\mathcal{E})$$

which is a consequence of the relations in $\hat{\mathcal{R}}$.

Proof: We refer to the tails for which we know of the existence of such a relation as being *of the right kind*. They are the tails in \mathcal{E} and those of weight greater than b .

If a is defined by $[x, u] =: v_{[x,u]}a$, we prove the statement by induction on $m = \omega(u)$. Note that $2m \leq b$.

If $m = 1$, then $a \in \mathcal{E}$.

For $m > 1$ the definition of u in $\hat{\mathcal{R}}$ is either of the form $[y, z] =: u$ or of the form $y^p =: u$, because $u \in \mathcal{A}$ and the presentation $\{\mathcal{A} \mid \mathcal{R}\}$ is labelled.

Case 1: $[y, z] =: u$. Then $\omega(z) = 1$ and thus $\omega(y) = m - 1$.

The result is proved by collecting the word xyz in two ways which begin with different steps and comparing the resulting normal words $((xy)z)$ and $(x(yz))$. We introduce words $w_1, \dots, w_6, t_3, t_6$ in the course of describing the two collections and then retrace the steps to show they give the desired conclusion.

If the first step is interchanging x and y then $(xy)z \rightarrow yxv_{[x,y]}t_{[x,y]}z$ where $v_{[x,y]}$ and $t_{[x,y]}$ are $(b - 1)$ -heavy. In the course of the collection there will be a (unique) step which replaces the subword xz by $zxv_{[x,z]}t_{[x,z]}$. That step will be applied to a word of the form $yxzw_1$ where w_1 is a not necessarily normal word which is $(b - 1)$ -heavy since it was obtained by applying collection steps to the word $v_{[x,y]}t_{[x,y]}z$. Note that the form of w_1 does not depend on the intermediate steps. The result of the step is $yzxv_{[x,z]}t_{[x,z]}w_1$. The collection continues until the next step replaces the subword yz . That step will replace a word of the form $yzxw_2$ and give $zyxw_2$. The collection finally yields $zyxw_3t_3$ where w_3 involves only generators in \mathcal{A} and t_3 only tails.

If the first step is interchanging y and z , the next step interchanges x and z :

$$\begin{aligned} x(yz) &\rightarrow xzyu \\ &\rightarrow zxv_{[x,z]}t_{[x,z]}yu. \end{aligned}$$

In the course of the collection there will be a (unique) step which replaces the subword xy by $yxv_{[x,y]}t_{[x,y]}$. That step will be applied to a word of the form $zxyw_4$, where w_4 contains u . The result of the step is $zyxv_{[x,y]}t_{[x,y]}w_4$. Collection continues until it replaces the subword xu . That step will be applied to a word of the form in $zyxuw_5$ and gives $zyuxv_{[x,u]}aw_5$. The collection finally yields $zyuxw_6t_6$ where w_6 involves only generators in \mathcal{A} and t_6 only tails.

Since $((xy)z) = (x(yz))$ and $\{\mathcal{A} \mid \mathcal{R}\}$ is consistent, $t_3 = t_6$ is a consequence of the relations in $\hat{\mathcal{R}}$. Now t_6 contains the generator a (once) while t_3 does not contain a . So it suffices to prove t_3 and t_6a^{-1} differ only in tails of the right kind.

Consider the steps of the first collection as far as tails are concerned. The word w_1 contains $t_{[x,y]}$, which is not of the right kind, once. All the other tails come from steps which collect z past a generator in $v_{[x,y]}$. Since $v_{[x,y]}$ is $(b-1)$ -heavy and z has weight 1, such a collection step introduces a tail of the right kind. The words w_2 and t_3 contain $t_{[x,y]}$ and $t_{[x,z]}$ once each. Since w_1 is $(b-1)$ -heavy and $v_{[x,z]}$ is $(b-m+1)$ -heavy (and $b-m+1 \geq 2$) all the other tails are of the right kind by hypothesis.

Consider the steps of the second collection. The word w_4 contains $t_{[x,z]}$ and u once each and otherwise is $(b-m+1)$ -heavy. All the tails of weight greater than b in w_4 are of the right kind by hypothesis. Since y has weight $m-1$, the induction hypothesis gives that all tails of weight b in w_4 are of the right kind. The word w_5 contains $t_{[x,y]}$ and $t_{[x,z]}$ once each and otherwise is $(b-m+1)$ -heavy. All the tails introduced in the steps from $zyxv_{[x,y]}t_{[x,y]}w_4$ to $zyxuw_5$ have weight greater than b and so are of the right kind. Finally since $v_{[x,u]}$ is b -heavy, the word t_6 contains a , $t_{[x,y]}$ and $t_{[x,z]}$ once each and all the other tails are of the right kind. The result follows.

Case 2: $y^p =: u$. Then $\omega(y) = m-1$. The result is proved by collecting the word xy^p in two different ways which begin with different steps and comparing the resulting

normal words $(x(y^p))$ and $((xy)y^{p-1})$. As in Case 1 we introduce words w_1, \dots, w_p and t_p in the course of describing the two collections and then retrace the steps to show they give the desired conclusion.

If the first step is the application of the power relation, then $x(y^p) \rightarrow xu$. The next step yields $xu \rightarrow uxv_{[x,u]}a$. If the first step is interchanging x and y then $(xy)y^{p-1} \rightarrow yxv_{[x,y]}t_{[x,y]}y^{p-1}$. In the course of the collection there will be a (unique) step which replaces the subword xy by $yxv_{[x,y]}t_{[x,y]}$. That step will be applied to a word of the form yxw_1 where w_1 contains exactly $p - 2$ occurrences of the generator y and one occurrence of $t_{[x,y]}$. The result of the step is $y^2xv_{[x,y]}t_{[x,y]}w_1$. The collection continues until all generators y have been interchanged with x . In each case the subword xy is replaced in a word of the form $y^i xyw_i$, where w_i contains exactly $p - i - 1$ occurrences of y and i occurrences of $t_{[x,y]}$. After the last replacement of the subword xy the result is $y^p xv_{[x,y]}t_{[x,y]}w_{p-1}$. The collection finally yields $uxw_p t_p$, where w_p only involves generators in \mathcal{A} and t_p only tails.

Since $(x(y^p)) = ((xy)y^{p-1})$ and $\{\mathcal{A} \mid \mathcal{R}\}$ is consistent, $a = (t_p)$ is a consequence of the relations in $\hat{\mathcal{R}}$. It suffices to prove that t_p contains only tails of the right kind.

Consider the steps of the second collection as far as tails are concerned. The word w_1 contains $t_{[x,y]}$, which is not of the right kind, once. All other tails come from steps which collect y past a generator in $v_{[x,y]}$. Since $v_{[x,y]}$ is $(b - 1)$ -heavy and y has weight $m - 1$, such a collection step introduces tails of the right kind. The word w_i for $i < p$ contains the generator $t_{[x,y]}$, which is not of the right kind, exactly i times and all other tails are of the right kind. Finally the word w_p does not contain any tails. The word t_p has p occurrences of the tail $t_{[x,y]}$ and all other tails are of the right kind. But since the tails have exponent p the normal word (t_p) has only tails of the right kind. ■

We obtain a presentation $\{\tilde{\mathcal{A}} \mid \tilde{\mathcal{R}}\}$ for the same group by replacing the occurrence of every element in $\hat{\mathcal{A}} \setminus \mathcal{E}$ in every relation in $\hat{\mathcal{R}}$ by the corresponding word in elements of \mathcal{E} (deleting identity relations) and setting $\tilde{\mathcal{A}}$ to be \mathcal{E} . In practice, the elements of $\hat{\mathcal{A}} \setminus \mathcal{E}$ are never introduced into the presentation, and every time a word in the generators \mathcal{E} is computed, it is inserted into the presentation immediately.

The next lemma shows that \mathcal{E} can be reduced even further. Let \mathcal{F} be the set of those elements a in \mathcal{E} for which a is defined as a commutator or as the p -th power of a generator y , where y is defined as a p -th power.

Lemma 2.12 *For each $a \in \hat{\mathcal{A}} \setminus \mathcal{A}$ there exists a relation*

$$a = W(\mathcal{F}),$$

where $W(\mathcal{F})$ is a word in the elements of \mathcal{F} which is a consequence of the relations in $\hat{\mathcal{R}}$.

Proof:

The statement follows by reverse induction on $\hat{\omega}(a)$ from the next lemma. The induction hypothesis, *i.e.* the case $\hat{\omega}(a) = c + 1$, is proved by the same argument as used in the proof of the next lemma specialised to this case. ■

Lemma 2.13 *Let $b \in \{2, \dots, c\}$. If for each $\tilde{a} \in \hat{\mathcal{A}} \setminus \mathcal{A}$ with $\hat{\omega}(\tilde{a}) > b$ there exists a relation*

$$\tilde{a} = \tilde{W}(\mathcal{F})$$

which is a consequence of the relations in $\hat{\mathcal{R}}$, then there exists a relation

$$a = W(\mathcal{F})$$

which is a consequence of the relations in $\hat{\mathcal{R}}$ for each $a \in \hat{\mathcal{A}} \setminus \mathcal{A}$ with $\hat{\omega}(a) = b$.

Proof:

All we need to prove is that each element in $\mathcal{E} \setminus \mathcal{F}$ has an expression as a word in \mathcal{F} . Then the assertion follows from Lemma 2.10, by replacing the occurrences of the element in \mathcal{E} in a word by its expression in the elements of \mathcal{F} . We call an element of \mathcal{E} a *tail of the right kind*, if we know that it can be expressed as a word in the elements in \mathcal{F} .

Let $a \in \mathcal{E} \setminus \mathcal{F}$ with $\hat{\omega}(a) = b$. Then $u^p =: a$ and $[x, y] =: u$. Since $u \in \mathcal{A}$ we know that $\omega(y) = 1$, thus $\omega(x) = b - 2$, since $\omega(u) = b - 1$. Note that $b - 2 \geq 1$.

The result is proved by collecting the word $x^p y$ in two different ways which begin with different steps and comparing the resulting normal words $((x^p)y)$ and $(x^{p-1}(xy))$. By analogy to the proof of Lemma 2.11 we introduce words w_1, \dots, w_{p+2} , t_1 and t_2 in the course of describing the two collections and then retrace the steps to show they give the desired conclusion.

If the first step is applying the power relation then $(x^p)y \rightarrow v_{x^p} t_{x^p} y$. The collection continues and finally yields $yw_1 t_1$. Here w_1 is a normal word involving only generators in \mathcal{A} and t_1 involves only tails.

If the first step is interchanging x and y , the next step again interchanges x and y :

$$\begin{aligned} x^{p-1}(xy) &\rightarrow x^{p-1}yxu \\ &\rightarrow x^{p-2}yxuxu. \end{aligned}$$

In the course of the collection there will be a (unique) step which replaces the subword xy by $yxv_{[x,y]}t_{[x,y]}$. That step will be applied to a word of the form $x^{p-2}yw_2$ where w_2 is a not necessarily normal word that contains exactly two occurrences of x and two occurrences of u . The result of the step is $x^{p-3}yxuw_2$. The collection continues until all generators x have been interchanged with y . In each case the subword xy is replaced in a word of the form $x^{p-i}yw_i$, where w_i contains exactly i occurrences of x and i occurrences of u and $i < p$. The collection continues until the next step replaces the subword u^p by a . It is applied to a word of the form $yw_p u^p w_{p+1}$. The result is $yw_p a w_{p+1}$. Finally the collection yields $yw_{p+2} t_2$, where w_{p+2} involves only generators in \mathcal{A} and t_2 only tails. Note that a occurs exactly once in t_2 .

Since $((x^p)y) = (x^{p-1}(xy))$ and $\{\mathcal{A} \mid \mathcal{R}\}$ is consistent, $t_1 = t_2$ is a consequence of the relations in $\hat{\mathcal{R}}$. So it suffices to prove that t_1 and $a^{-1}t_2$ contain only tails of the right kind.

Consider the steps of the first collection as far as tails are concerned. The tails in w_1 arise either by interchanging two generators of weight at least $(b-1)$ and are thus of the right kind or by interchanging y with a generator of weight at least $(b-1)$ and are also of the right kind. Consider the steps of the second collection. All tails occurring in the word w_2 come from applying steps to the $(b-2)$ -heavy word $xuxu$. Thus all the tails that arise are of the right kind. Similarly all tails in w_i for $i \leq p+2$ are of the right kind, since they occur by applying collection steps to a

$(b - 2)$ -heavy word in which the only element of weight $(b - 2)$ is x . All generators occurring in t_1 and t_2 are, therefore, of the right kind with the exception of the one occurrence of a in t_2 . ■

We obtain a power commutator presentation $\{\tilde{\mathcal{A}} \mid \tilde{\mathcal{R}}\}$ for the p -covering group G^* of G on $\mathcal{A} \cup \mathcal{F}$ by replacing each $a \in \hat{\mathcal{A}} \setminus \mathcal{F}$ by the corresponding word $W(\mathcal{F})$ as given by Lemma 2.12. This presentation is not necessarily consistent.

2.3 The Prime Quotient Algorithm

Let us now return to the PQA. Given a labelled consistent power commutator presentation for the quotient $K \cong G/\mathcal{P}_i(G)$ and a labelled epimorphism θ of G onto K the task of the basic step is to obtain a labelled consistent power commutator presentation for the quotient $H \cong G/\mathcal{P}_{i+1}(G)$ and a labelled epimorphism τ of G onto H . This is done using a refined version of Theorem 2.4.

The input of the basic step of the PQA supplies an epimorphism $\theta : G \rightarrow K$, given by the images of the generators of G (as given by the finite presentation). For ψ an epimorphism of G onto H we have $P_i(G)\psi = P_i(G\psi) = P_i(H) = \mathcal{P}_i(G)/\mathcal{P}_{i+1}(G)$ and the epimorphism $\tilde{\psi} : G \rightarrow K$ induced by ψ satisfies $\ker \tilde{\psi} = P_i(G) = \ker \theta$. Thus θ and $\tilde{\psi}$ differ only by an automorphism of K . Therefore we can assume that ψ is a lifting of θ onto H , *i.e.* $\tilde{\psi} = \theta$. Note that the elements in $\mathcal{P}_i(G)/\mathcal{P}_{i+1}(G)$ are central and of order p in H . Therefore the image of an element g in G under ψ is of the form $g\theta z_g$ with $z_g \in \mathcal{P}_i(G)/\mathcal{P}_{i+1}(G)$. Define a map $\tau : G \rightarrow H : g \mapsto g\theta u_g$ where u_g is an (unknown) element of $\mathcal{P}_i(G)/\mathcal{P}_{i+1}(G)$, hence an element which is central and of order p in H . Then τ is a homomorphism if and only if the images of the generators of G under τ satisfy the relators of G . Let r be a relator of G . The image $r\tau$ has the form $r\theta u_r$, where u_r is a word in the u_g , since the u_g are central. But θ is a homomorphism, thus $r\theta = 1$ and we obtain the relation $u_r = 1$. Let U be the normal subgroup generated by all u_r . The map induced by τ from G to H/U is a homomorphism. Thus $K = H/U$ and ψ is the map induced by τ .

Let Y be the \mathbb{F}_p -space with basis $\mathcal{F} \cup \{u_g\}$. Let M denote the kernel of the epimorphism of H onto K ; then M is a homomorphic image of Y . The following

theorem describes the kernel of the homomorphism from Y onto M in a way suitable for computation. It considers certain non-normal words in K^* .

Theorem 2.14 *Let Y be the free abelian group on $\mathcal{F} \cup \{u_g\}$ and $\{\tilde{\mathcal{A}} \mid \tilde{\mathcal{R}}\}$ the p -covering presentation of the d -generator group $G/\mathcal{P}_i(G)$. Let W consist of the words in $\{a_1, \dots, a_n\}$ listed in Theorem 2.4.*

Let S be the set of elements of Y obtained by collecting the words in W with respect to $\{\tilde{\mathcal{A}} \mid \tilde{\mathcal{R}}\}$ and let $T = \{r_i(g_1^T, \dots, g_b^T) \mid 1 \leq i \leq m\}$ be the set of elements of Y obtained by evaluating the relators of G in the images of the generators of G under the map τ . Then M is isomorphic to $Y/\langle S \cup T \rangle$.

A basis $\{m_1, \dots, m_t\}$ for M can be computed using the Gaussian elimination algorithm. Let \mathcal{A}^* be the set $\mathcal{A} \cup \{m_1, \dots, m_t\}$. The image of each element a_{n+i} can be expressed in the basis of M . This expression replaces a_{n+i} in $\tilde{\mathcal{R}}$ and the u_g in the map τ . Thus a labelled consistent power commutator presentation $\{\mathcal{A}^* \mid \mathcal{R}^*\}$ for $G/\mathcal{P}_{i+1}(G)$ and a labelled epimorphism τ from G to H are obtained. In particular it can be shown that the elements in the basis of M are words in the elements of \mathcal{F} . Therefore, the function ω induces a weight function ω^* on $\{\mathcal{A}^* \mid \mathcal{R}^*\}$ by setting $\omega^*(a_k) = \omega(a_k)$ for $a_k \in \mathcal{A}$ and $\omega^*(m_k) = c + 1$.

3 The implementation of the prime quotient algorithm in GAP

In this chapter we describe some of the problems encountered when implementing the prime quotient algorithm in GAP and some of the solutions. A first version was written by Nickel and Niemeyer in GAP 2.4. It was possible to write this quickly, demonstrating the ability of GAP to allow rapid prototyping. It turned out that the implementation in GAP 2.4 was relatively slow when compared to the FORTRAN standalone. It was about a factor of 60 slower on the examples tested. There are five major reasons for this. The following three were eliminated in the implementation of GAP 3. The others are of a more general nature.

- A major problem was the large amount of garbage created and the consequential cost of garbage collection. The cause for this was that each change of the presentation - of which there are many as Section 2 shows - resulted in the creation and deletion of many bags.
- A second reason was that the FORTRAN standalone uses the observation that in doing computations in class $c + 1$ there was already significant information about the class c quotient. The lack of flexibility in the group multiplication and inversion functions meant that this could not be done in GAP 2.4.
- The evaluation of relations was slower because there was no kernel support for this.

In this chapter we detail these problems and describe how they have been eliminated in GAP 3. In summary this was done by the introduction of a new internal data structure to store pc -presentations and by new internal functions. Other improvements were achieved by carefully implementing critical parts of the relevant collectors. The details of the changes are described in Section 3.3. We begin this chapter by discussing some relevant features of GAP.

3.1 About GAP

The central part of GAP is a C-program which during execution reads input, evaluates it and prints the result. This C-program is referred to as the GAP *kernel*. The input has to be formulated in a special format, the GAP *language*. This interpreted language is especially designed for group theory.

The part of the GAP kernel responsible for the interpretation of the GAP language is the *evaluator*. Another part, GASMAN (see Schönert, 1987), is the GAP storage manager and is responsible for creating, moving and deleting objects of the GAP language.

The data structures of the GAP language can be divided into three classes: constants, internal data structures and GAP data structures. *Constants* are objects like integers, elements of finite fields, and permutations. Constants cannot be manipulated in the GAP language; all that can be done with them is to combine them by an arithmetic operation and to create a new constant. *GAP data structures* are built up from two basic data structures provided by the GAP language, the record and the list. The components of a record or a list can be chosen from any of the above classes of data structures. The GAP language provides mechanisms to manipulate records and lists. *Internal data structures* are objects which cannot be manipulated by any of the mechanisms built into the GAP language, most of them are not accessible from the GAP language, for example finite fields. Those which are accessible are similar to constants in that the GAP language does not provide mechanisms to manipulate them. These accessible internal data structures are equipped with a set of functions which create, manipulate, and access internal data structures. These functions are accessible in the GAP language and are called *internal functions*.

The main reason for the existence of internal data structures and internal functions is efficiency. Each internal data structure could be represented by a GAP data structure, however the internal data structure stores the data much more efficiently than an equivalent GAP data structure. Also, internal data structures are accessed and manipulated by C-subroutines of the GAP kernel and it is possible to design the internal data structure and the routines using it such that good results can be achieved.

Functions in the kernel are called *kernel functions*. For instance, internal functions and arithmetic functions for constants are kernel functions.

The objects in GAP are stored in *bags*. A bag is a part of memory pointed to by a *handle*. Each bag has a data type associated with it, such as T_LIST, which indicates that it is a list, and this data type specifies the structure of the bag.

GASMAN is responsible for allocating and deleting bags. If a bag is referenced, it is *alive*. If a bag is not alive, it is part of the *garbage*. If the amount of available memory becomes too small GASMAN invokes a *garbage collection*, which frees the space occupied by garbage and moves the living bags to one part of the memory. Garbage collections are quite expensive in GAP 3.

3.2 Problems with the implementation in GAP 2.4

One of the major bottle necks for the speed of the algorithm in GAP 2.4 was the internal data structure used to store *pc*-presentations. In GAP 2.4 *pc*-presentations could not be modified. Therefore, every time the prime quotient algorithm required a change in the *pc*-presentation, a new internal data structure for a *pc*-presentation was created. The old *pc*-presentation became obsolete and was deleted. This created the following problems. Enough memory to hold two copies of a *pc*-presentation was necessary. Further, the creation and deletion of *pc*-presentations created a lot of garbage. Consequently, this increased the number of garbage collections. These garbage collections were time consuming. In each exponent- p class it was necessary to change the *pc*-presentation a number of times. This follows from the proof of Lemma 2.11 and Lemma 2.13. The percentage of the runtime spent in manipulating with *pc*-presentations in GAP 2.4 was very high compared to the percentage of the runtime spent in the FORTRAN standalone implementation for the same task. In Section 5.1 we give a short outline of the details of the implementation of *pc*-presentation in the kernel of GAP 2.4 and a more detailed outline of the implementation in GAP 3, which considerably reduces the percentage of the runtime of the algorithm spent in manipulating with presentations.

Another reason for the slowness of the algorithm was the lack of flexibility with which group multiplication and inversion could be performed on GAP level. In the proofs of Lemma 2.11 and Lemma 2.13 it is necessary to collect words of the form

$((xy)z)$ or $(x(yz))$ with respect to a given pc -presentation $\{\mathcal{A}^* \mid \mathcal{R}^*\}$. The percentage of time spent in evaluating these expressions in the overall runtime of the prime quotient algorithm in GAP 2.4 was much larger than the percentage of time spent in the FORTRAN standalone implementation. It is possible to use some mathematical knowledge about the algorithm to reduce the time spent. From theoretical observations we know that the pc -presentation $\{\mathcal{A}^* \mid \mathcal{R}^*\}$ is a presentation for a covering group G^* of a group G . It differs from a consistent pc -presentation $\{\mathcal{A} \mid \mathcal{R}\}$ of G in that it has some additional generators \mathcal{F} , which are central and of order p , and for each relation r^* in \mathcal{R}^* there is a relation r in \mathcal{R} with the same left hand side such that the right hand side of r^* differs from the right hand side of r only by a (possibly empty) word in the additional generators \mathcal{F} . Let us assume that x, y and z are in \mathcal{A} , then we know that the expression $((xy)z)^{-1}(x(yz))$ collects to the identity element when collected using $\{\mathcal{A} \mid \mathcal{R}\}$, since this is a consistent pc -presentation for G . Collecting the same expression using the pc -presentation $\{\mathcal{A}^* \mid \mathcal{R}^*\}$ yields a word v in the generators \mathcal{F} . In the algorithm we need to compute this word. Collect $((xy)z)$ and $(x(yz))$ using $\{\mathcal{A}^* \mid \mathcal{R}^*\}$ and denote the collected words by w_1t_1 and w_2t_2 , where w_1 and w_2 are words in \mathcal{A} and t_1 and t_2 are words in \mathcal{F} . Since $\{\mathcal{A} \mid \mathcal{R}\}$ is consistent we know that $w_1 = w_2$, hence $v = t_1^{-1}t_2$. Therefore, to compute v it is only necessary to compute the inverse of t_1 and multiply it with t_2 . This is a much easier task, since the elements of \mathcal{F} are central and of order p . If we computed v by collecting $((xy)z)^{-1}(x(yz))$ using $\{\mathcal{A}^* \mid \mathcal{R}^*\}$, the computation would be much more expensive, since it involves the computation of the inverse of a word in \mathcal{A}^* and the multiplication of two words in \mathcal{A}^* .

The faster way of multiplying $((xy)z)$ and $(x(yz))$ is however specific to the prime quotient algorithm and was not automatically performed by the internal functions dealing with pc -presentations in GAP 2.4. In GAP 3, this has been changed and had a positive impact on the speed of the prime quotient algorithm.

Another part of the algorithm which used a larger percentage of the runtime of the algorithm in GAP 2.4 than the FORTRAN standalone is the following. In computing the normal subgroup which has to be factored out of the p -covering group in order to find the largest epimorphic image of the group G of a given exponent- p class, the relators of G have to be evaluated on the images of the generators of G

under a certain map. In GAP 2.4 there was no support from the kernel to perform this evaluation and the evaluation had to be implemented on the GAP level. In GAP 3, there is a kernel function which evaluates a relator on the images of the generators involved under a given map. Again, this had a significant impact on the percentage of time used in GAP 2.4 to perform this task compared with the FORTRAN standalone implementation. However, the percentage of time used in the current version of GAP is still larger than in the C-implementation. This is due to the fact, that the C-implementation can deal with powers and commutators which occur in relations. A commutator $[a, b]$ is stored in the C-implementation in a way that keeps the information that this word is the commutator of the words a and b , and a p -th power a^p is stored retaining the information that this is the p -th power of the word a . In GAP however this information is lost. The commutator $[a, b]$ is stored as the word $a^{-1}b^{-1}ab$ and the p -th power a^p is stored as $a \dots a$. Assume that a given relator r contains the subword $[a, b]$, for a, b words in the generators of G . In the C-implementation the image of this subword under a map φ is evaluated by mapping the subword to $[a\varphi, b\varphi]$. In GAP however, this is done by evaluating the subword $(a\varphi)^{-1}(b\varphi)^{-1}a\varphi b\varphi$. But when collecting commutators, it is possible to use a more efficient method than expanding it and collecting the expanded word. Similarly it is possible to collect $(a\varphi)^p$ much more quickly than $a\varphi \dots a\varphi$. Therefore it would be desirable if the relations in GAP were stored in a way which keeps track of the p -th powers and commutators occurring in them. (This is planned for the near future. In the meantime the problem can, to a significant amount, be worked round as is shown by the two presentations G_2 and G_3 in Appendix 1.) Furthermore, GAP cannot deal with relations, but only with relators. If $a = b$ is a relation which is to be passed to the prime quotient algorithm it has to be passed as the relator ab^{-1} . But in many situations arising in practice it is more efficient to compute the image of this relation under a map φ by computing the images of the right hand side and left hand side separately rather than dealing with inverses. Thus it would also be desirable that GAP can handle relations as well as relators. These changes to the way GAP handles relations would also give users the opportunity to specify relations in a way that includes their knowledge about them and forces GAP to collect them according to the way they are given.

3.3 Kernel changes

In this section we give an account of the changes in the kernel for GAP 3 which were needed to solve three of the problems detailed above.

3.3.1 *ag*-groups and presentations

A finite polycyclic group G is represented by an internal data structure which stores the generators and relations of G . Such groups are called *ag*-groups. The elements of such a group in their canonical representation are called *ag*-words.

In order to be able to use generic, representation independent, algorithms, it must be possible to compare elements of *ag*-groups with the infix operators `*`, `.`, `/` and `^` without mentioning the group. Therefore *ag*-words have to carry the information needed for computing with them. Thus an *ag*-word w is equipped with a fixed meaning as an element of a group G . This has the consequence that the group G and its presentation may not be changed.

A prime quotient algorithm constantly changes the presentation in order to add new generators, remove redundant generators or modify an inconsistent presentation. Hence the data structures of *ag*-groups are not suitable.

Internal functions for the prime quotient algorithm had to be developed in a context not relying on a fixed *ag*-group. In addition to the data type of an *ag*-group there is the data type *pc*-presentation in GAP 3. A *pc*-presentation does not describe a fixed group, but a changeable presentation. A *pc*-presentation P consists of generators and relations. The generators behave like elements of a free group, *i.e.* they can be multiplied completely independently of P . The relations are a collection of *pc*-relations in these generators.

In order to compose words w_1 and w_2 in the generators of a *pc*-presentation P with respect to P , one of the functions `ProductPcp`, `QuotientPcp`, `LeftQuotient`, `CommPcp` or `ConjugatePcp` has to be used. All of these functions take as first argument the *pc*-presentation P followed by the words w_1 and w_2 . They return their result as a normal word with respect to P . The function `NormalWordPcp` returns the normal form of a word w with respect to P .

Composition by functions has the advantage that words do not carry a reference to a fixed presentation. Thus changing a presentation does not influence the meaning

of a word stored at another place, because a word w obtains its meaning as an element of a group G only by specifying the presentation P in one of the functions.

3.3.2 *pc*-presentations

In order to avoid time consuming type checking it was necessary to create a new word type instead of using abstract generators. This only generates a very small overhead.

In the following, “word” denotes a sequence of abstract generators without any further information, “*pc*-word” describes an element of a *pc*-presentation and “*S*-word” a free word as described below. “Abstract words” are either words, *S*-words or *pc*-words. The different types of words cannot be distinguished at GAP level.

Words of type T_WORD are realised as a sequence of handles of type T_AGEN. These abstract generators of type T_AGEN are created by the function `AbstractGenerator` which returns a word of length 1. The *S*-words (type T_SWORD) are realised as a sequence of pairs whose first entry points to a list of abstract generators and whose second entry represents an exponent. Whenever an *S*-word w_1 is to be composed with a word w_2 , a copy w'_1 of w_1 is transformed into a word and the composition is performed using the words w'_1 and w_2 . When two *S*-words with different lists of abstract generators are to be composed, the composition is done using copies of both words.

Like *S*-words, *pc*-words are of type T_SWORD, but their first entry is a reference to information that allows the generators in the *pc*-word to be matched with the generators of a *pc*-presentation. We say that the *pc*-word is compatible with that *pc*-presentation.

If the function `ProductPcp` is called with arguments a *pc*-presentation P and two abstract words w_1 and w_2 , it tries to convert copies of w_1 and w_2 into *pc*-words compatible with P . This can be done successfully if the generators occurring in w_1 and w_2 are among the generators of P . If w_1 or w_2 is already a *pc*-word compatible with P , nothing has to be done for that word. In all other cases an error occurs.

In a prime quotient algorithm the words w_1 and w_2 will, in almost all cases, be *pc*-words compatible with a *pc*-presentation P since they were created in computations involving P . Thus in these cases it will not be necessary to create copies of the words when used as arguments to `ProductPcp`.

If generators have to be added to a pc -presentation P with generators g_1, \dots, g_t , the list of generators of P can be extended. This does not change the compatibility with P of existing pc -words. If generators of a presentation P have to be deleted then words involving deleted generators become incompatible with the pc -presentation after the deletion has taken place.

4 Documentation

4.1 GAP Functions for the Prime Quotient Algorithm

This section describes the prime quotient functions.

`PQuotient(G, p, cl)`

`PrimeQuotient(G, p, cl)`

`PQuotient` computes quotients of prime power order of finitely presented groups. G must be a group given by generators and relations. `PQuotient` expects G to be a record with the record fields `generators` and `relators`. The record field `generators` must be a list of abstract generators created by the function `AbstractGenerator`. The record field `relators` must be a list of words in the generators which are the relators of the group. The argument p must be a prime and cl has to be an integer, which specifies that the quotient of prime power order computed by `PQuotient` is the largest p -quotient of G of class at most cl . `PQuotient` returns a record `Q`, the PQP record, which has, among others, the following record fields describing the prime quotient Q .

`generators`: A list of abstract generators which generate Q .

`pcp` : The internal power-commutator presentation for Q .

`dimensions`: A list, where `dimensions[i]` is the dimension of the i -th factor in the lower exponent- p central series calculated by the PQA.

`prime`: The integer p , which is a prime.

`definedby`: A list which contains the definition of the k -th generator as its k -th entry. There are three different types of entries, namely lists, positive and negative integers.

`[j,i]`: the generator is defined to be the commutator of the j -th and the i -th element in `generators`.

i: the generator is defined as the p -th power of the i -th element in generators.

-i: the generator is defined as an image of the i -th generator in the finite presentation for G , consequently it must be a generator of weight 1.

epimorphism: A list containing an image in Q of each generator of G . If the image is the i -th element of generators of Q it is stored as the integer i . If the image is the abstract word w in the generators of Q then the word w is stored.

The following is an example of the computation of the largest 5-quotient of class 4 of the group given by the finite presentation $\{a, b \mid a^{25}/(ab)^5, [a, b]^5, (a^b)^{25}\}$.

```
# Define the group
gap> a := AbstractGenerator("a");
a
gap> b := AbstractGenerator("b");
b
gap> G := rec( generators := [a,b],
               relators := [ a^25/(a*b)^5, Comm(a,b)^5, (a^b)^25 ] );
rec( generators := [ a, b ],
     relators := [
       a^25*b^-1*a^-1*b^-1*a^-1*b^-1*a^-1*b^-1*a^-1*b^-1*a^-1,
       a^-1*b^-1*a*b*a^-1*b^-1*a*b*a^-1*b^-1*a*b*a^-1*b^-1*a*b*a^-1*b^-1*a*b,
       b^-1*a^25*b ] )
# Call PQuotient
gap> P := PQuotient( G, 5, 4 );
PQP( rec(
  generators := [ g1, g2, a3, a4, a6, a7, a11, a12, a14 ],
  definedby := [ -1, -2, [ 2, 1 ], 1, [ 3, 1 ], [ 3, 2 ], [ 5, 1 ],
                [ 5, 2 ], [ 6, 2 ] ],
  prime := 5,
  dimensions := [ 2, 2, 2, 3 ],
  epimorphism := [ 1, 2 ],
  powerRelators := [ g1^5/(a4), g2^5/(a4^4), a3^5, a4^5, a6^5, a7^5, a11^5,
                    a12^5, a14^5 ],
  commutatorRelators := [ Comm(g2,g1)/(a3),
                          Comm(a3,g1)/(a6), Comm(a3,g2)/(a7),
                          Comm(a6,g1)/(a11), Comm(a6,g2)/(a12),
                          Comm(a7,g1)/(a12), Comm(a7,g2)/(a14) ],
  definingCommutators := [ [ 2, 1 ], [ 3, 1 ], [ 3, 2 ], [ 5, 1 ],
                           [ 5, 2 ], [ 6, 1 ], [ 6, 2 ] ] ) )
```

The PQA returns a PQP record for the exponent-5 class 4 quotient. Note that instead of printing the PQP record P an equivalent representation is printed which can be read into GAP. See PQP for details.

The quotient defined by P has nine generators,

$g1, g2, a3, a4, a6, a7, a11, a12, a14,$

stored in the list `P.generators`. From `powerRelators` we can read off that

$$g1^5 =: a4 \text{ and } g2^5 = a4^4$$

and all other generators have trivial 5-th power. From the list `commutatorRelators` we can read off the non-trivial commutator relations

$$\begin{aligned} \text{Comm}(g_2, g_1) &=: a_3, \text{Comm}(a_3, g_1) =: a_6, \text{Comm}(a_3, g_2) =: a_7, \\ \text{Comm}(a_6, g_1) &=: a_{11}, \text{Comm}(a_6, g_2) =: a_{12}, \text{Comm}(a_7, g_1) = a_{12}, \\ &\text{and } \text{Comm}(a_7, g_2) =: a_{14}. \end{aligned}$$

In this list `:=` denotes that the generator on the right hand side is defined as the left hand side. This information is given by the list `definedby`. The list `dimensions` shows that P is a class-4 quotient of order $5^2 \cdot 5^2 \cdot 5^2 \cdot 5^3 = 5^9$. The epimorphism of G onto the quotient P is given by the map $a \mapsto g_1$ and $b \mapsto g_2$.

`SavePQP(file, Q, N)`

`SavePQP` saves the PQP record Q to the file $file$ in such a way that the file can be read by GAP. The name of the record in the file will be N . This differs from printing Q to a file in that the required abstract generators are also created in $file$.

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
gap> G := rec( generators := [a,b],
> relators := [ a^25/(a*b)^5, Comm(a,b)^5, (a^b)^25 ] );;
gap> P := PQuotient( G, 5, 4 );;
gap> SavePQP( "Quo54", P, "Q" );
gap> Exec("more Quo54");
g1 := AbstractGenerator("g1");
g2 := AbstractGenerator("g2");
a3 := AbstractGenerator("a3");
a4 := AbstractGenerator("a4");
a6 := AbstractGenerator("a6");
a7 := AbstractGenerator("a7");
a11 := AbstractGenerator("a11");
a12 := AbstractGenerator("a12");
a14 := AbstractGenerator("a14");
Q := PQP( rec(
  generators := [ g1, g2, a3, a4, a6, a7, a11, a12, a14 ],
  definedby := [ -1, -2, [ 2, 1 ], 1, [ 3, 1 ], [ 3, 2 ], [ 5, 1 ],
                [ 5, 2 ], [ 6, 2 ] ],
  prime := 5,
  dimensions := [ 2, 2, 2, 3 ],
  epimorphism := [ 1, 2 ],
  powerRelators := [ g1^5/(a4), g2^5/(a4^4), a3^5, a4^5, a6^5, a7^5, a11^5,
                    a12^5, a14^5 ],
  commutatorRelators := [ Comm(g2, g1)/(a3),
                        Comm(a3, g1)/(a6), Comm(a3, g2)/(a7),
                        Comm(a6, g1)/(a11), Comm(a6, g2)/(a12),
                        Comm(a7, g1)/(a12), Comm(a7, g2)/(a14) ],
  definingCommutators := [ [ 2, 1 ], [ 3, 1 ], [ 3, 2 ], [ 5, 1 ],
                          [ 5, 2 ], [ 6, 1 ], [ 6, 2 ] ] ) );;
```

`PQP(r)`

`PQP` takes as argument a record r containing all information necessary to restore a PQP record Q . A PQP record Q is printed as function call to `PQP` with an argument describing Q . This is necessary because the internal power-commutator representation cannot be printed. Therefore all information about Q is encoded in a record r and Q is printed as `PQP(r)`.

`InitPQP(n, p)`

`InitPQP` creates a PQP record for an elementary abelian group of rank n and of order p^n for a prime p .

`FirstClassPQP(G, p)`

`FirstClassPQP` returns a PQP record for the exponent- p class 1 quotient of G .

`NextClassPQP(G, P)`

Let P be the PQP record for the exponent- p class c quotient of G . The function `NextClassPQP` returns a PQP record for the class $c + 1$ quotient of G , if such a quotient exists, and P otherwise. In latter case P is a maximal p -quotient of G . This is indicated by a comment if `InfoPQ1` is set to `Print`.

`Weight(P, w)`

Let P be a PQP record and w a word in the generators of P . The function `Weight` returns the weight of w with respect to the lower exponent- p central series defined by P .

`Factorization(P, w)`

Let P be a PQP record and w a word in the generators of P . The function `Factorization` returns a word in the weight 1 generators of P expressing w .

4.2 Internal Functions

The functions in this section can be used to construct and manipulate pc-presentations in GAP.

`Pcp(str, n, p, collector)`

`Pcp` creates a finite polycyclic presentation with n generators of an elementary abelian p -group of order p^n the type of which (power-commutator or power-conjugate) is determined by the parameter `collector`. It returns a pc description P of this presentation.

`collector` describes the collector which is used when computing normal words in P (see `NormalWordPcp`). Until now only "combinatorial" for combinatorial collection in p -groups using a power-commutator presentation is allowed, while future releases of GAP will also support a single collector with power-conjugate presentation. The collector of a pc description cannot be changed after creating a presentation with `Pcp`.

`str` describes the name-prefix used for the abstract generators of P . The names of n generators of P consist of the prefix `str` followed by numbers 1 to n . `GeneratorsPcp` retrieves the generators of P (see `GeneratorsPcp`).

Note that a power-commutator presentation can be changed with `DefineCommPcp` and `DefinePowerPcp`, while a power-conjugate presentation is changed using `DefineConjugatePcp` and `DefinePowerPcp`, which means that `Pcp` can be used as a start for further more complicated changes of a finite polycyclic presentation.

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> DefineCentralWeightsPcp( P, [ 1, 1, 2, 2, 2 ] );
gap> DefineCommPcp( P, 2, 1, a[3] * a[4] * a[5] );
gap> CommPcp( P, a[2]^2, a[1]^2 );
a[3]^2*a[4]^2*a[5]^2
gap> ShrinkPcp( P, [4] );
gap> CommPcp( P, a[2]^2, a[1]^2 );

a[3]^2*a[5]^2
```

`GeneratorsPcp(P)`

`GeneratorsPcp` returns the list L of abstract generators of P .

The elements of L are ordinary abstract words of length 1 with names described in `Pcp`. In order to multiply them according to the relations of P , the function `ProductPcp` has to be used.

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> a[1]^8 * a[3]^1 * a[2]^2;
a1^8*a3^1*a2^2
gap> NormalWordPcp( P, last );

a1*a2^2*a3^6
```

`ExtendCentralPcp(P, L, p)`

Let P be a `pcp` description with generators (g_1, \dots, g_n) , let L be a list $[s_1, \dots, s_m]$ of m strings and let p be a prime. Then `ExtendCentralPcp` appends m new generators h_1, \dots, h_m with names s_1, \dots, s_m to P , such that each generator h_i is of order p in P and commutes with every generator of P .

Note that if P is a `pcp` description with combinatorial collector then p must be equal to the common relative order of the g_i in P .

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> ExtendCentralPcp( P, [ "b1", "b2", "b3" ], 7 );
gap> GeneratorsPcp( P );

[ a1, a2, a3, a4, a5, b1, b2, b3 ]
```

`ShrinkPcp(P, L)`

Let P be a `pcp` description with generators (g_1, \dots, g_n) . Let L be a subset of $\{1, \dots, n\}$. Then `ShrinkPcp` removes the generators g_i for all $i \in L$ from P .

Note that if a right hand side (see "Finite Polycyclic Presentations") of a relation in P contains a generator g_i with $i \in L$ then this generator is removed from the relation. If a left hand side (see "Finite Polycyclic Presentations") of a relation in P contains a generator g_i with $i \in L$ then this relation is removed completely.

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> ExtendCentralPcp( P, [ "b1", "b2", "b3" ], 7 );
gap> GeneratorsPcp( P );
[ a1, a2, a3, a4, a5, b1, b2, b3 ]
gap> ShrinkPcp( P, [ 2, 5, 7 ] );
gap> GeneratorsPcp( P );

[ a1, a3, a4, b1, b3 ]
```

`AgGroupPcp(P)`

Let P be a pcp description. Then `AgGroupPcp` returns the finite polycyclic group described by the presentation P as Ag-group.

`DefinePowerPcp(P, i, x)`

Let P be a pcp description with generators (g_1, \dots, g_n) of relative orders o_1, \dots, o_n . Then `DefinePowerPcp` changes the power relation of g_i in P to $g_i^{o_i} = x$.

Note that x must be a normal word of P (see `NormalWordPcp`) and must obey the collector dependent requirements. So if a combinatorial collector is installed, the central weight of x in P must be higher than the weight of g_i in P , otherwise an error is raised. So, for instance, before changing the power relations of an elementary abelian group with combinatorial collector, the central weights of P have to be adjusted using the function `DefineCentralWeightsPcp`.

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> DefineCentralWeightsPcp( P, [ 1, 1, 2, 2, 2 ] );
gap> DefinePowerPcp( P, 1, a[3]^2 );
gap> PowerPcp( P, a[1], 8 );

a1*a3^2
```

`AddPowerPcp(P, i, x)`

Let P be a pcp description with generators (g_1, \dots, g_n) of relative orders o_1, \dots, o_n . Then `AddPowerPcp` modifies the power relation $g_i^{o_i} = w$ of g_i in P such that $g_i^{o_i} = s$ where s is the sum of w and x in P (see `SumPcp`).

Note that x must be a normal word of P (see `NormalWordPcp`) and the sum s must obey the collector dependent requirements. So if a combinatorial collector is installed, the central weight of s in P must be higher than the weight of g_i in P , otherwise an error is raised (see `DefinePowerPcp`).

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> DefineCentralWeightsPcp( P, [ 1, 1, 2, 2, 2 ] );
gap> DefinePowerPcp( P, 1, a[3]^2 );
gap> PowerPcp( P, a[1], 8 );
a1*a3^2
gap> AddPowerPcp( P, 1, a[4] );
gap> PowerPcp( P, a[1], 8 );

a1*a3^2*a4
```

`SubtractPowerPcp(P, i, x)`

Let P be a pcp description with generators (g_1, \dots, g_n) of relative orders o_1, \dots, o_n . Then `SubtractPowerPcp` modifies the power relation $g_i^{o_i} = w$ of the generators g_i in P such that $g_i^{o_i} = d$ where d is the difference of w and x in P (see `DifferencePcp`).

Note that x must be a normal word of P (see `NormalWordPcp`) and the difference d must obey the collector dependent requirements. So if a combinatorial collector is installed, the central weight of d in P must be higher than the weight of g_i in P , otherwise an error is raised (see `DefinePowerPcp`).

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> DefineCentralWeightsPcp( P, [ 1, 1, 2, 2, 2 ] );
gap> DefinePowerPcp( P, 1, a[3]^2 );
gap> PowerPcp( P, a[1], 8 );
a1*a3^2
gap> SubtractPowerPcp( P, 1, a[4] );
gap> PowerPcp( P, a[1], 8 );

a1*a3^2*a4^6
```

`DefineCommPcp(P, i, j, x)`

Let P be a pcp description of a power-commutator presentation with generators (g_1, \dots, g_n) and let $1 \leq j < i \leq n$. Then `DefineCommPcp` modifies the commutator relation of the generators g_i and g_j in P such that $Comm(g_i, g_j) = x$.

Note that P must be a pcp description with power-commutator relations. A pcp with combinatorial collector uses a power-commutator presentation, while a pcp with single collector uses a power-conjugate presentation. In that case `DefineConjugatePcp` has to be used.

Note that x must be a normal word of P (see `NormalWordPcp`) and must obey the collector dependent requirements. So if a combinatorial collector is installed, the central weight of x in P must be higher than or equal to the sum of the weights of g_i and g_j in P , otherwise an error is raised. So, for instance, before changing the commutator relations of an elementary abelian group with combinatorial collector, central weights of P have to be adjusted using the function `DefineCentralWeightsPcp`.

```

gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> DefineCentralWeightsPcp( P, [ 1, 1, 2, 2, 2 ] );
gap> DefineCommPcp( P, 2, 1, a[3]^2 );
gap> CommPcp( P, 2, 1 );
a3^2
gap> AddCommPcp( P, 1, a[4] );
gap> CommPcp( P, 2, 1 );

a3^2*a4

```

`AddCommPcp(P, i, j, x)`

Let P be a pcp description of a power-commutator presentation with generators (g_1, \dots, g_n) and let $1 \leq j < i \leq n$. Then `AddCommPcp` modifies the commutator relation $Comm(g_i, g_j) = w$ of P such that $Comm(g_i, g_j) = s$ where s is the sum of w and x in P (see `SumPcp`).

Note that P must be a pcp description with power-commutator relations. A pcp with combinatorial collector uses a power-commutator presentation, while a pcp with single collector uses a power-conjugate presentation. In that case the function `DefineConjugatePcp` has to be used.

Note that x must be a normal word of P (see `NormalWordPcp`) and the sum s must obey the collector dependent requirements. So if a combinatorial collector is installed, the central weight of s in P must be higher than or equal to the sum of the weights of g_i and g_j in P , otherwise an error is raised (see `DefineCommPcp`).

```

gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> DefineCentralWeightsPcp( P, [ 1, 1, 2, 2, 2 ] );
gap> DefineCommPcp( P, 2, 1, a[3]^2 );
gap> CommPcp( P, 2, 1 );
a3^2
gap> AddCommPcp( P, 1, a[4] );
gap> CommPcp( P, 2, 1 );

a3^2*a4

```

`SubtractCommPcp(P, i, j, x)`

Let P be a pcp description of a power-commutator presentation with generators (g_1, \dots, g_n) and let $1 \leq j < i \leq n$. Then the function `SubtractCommPcp` modifies the relation $Comm(g_i, g_j) = w$ of P such that $Comm(g_i, g_j) = d$ where d is the difference of w and x in P (see `DifferencePcp`).

Note that P must be a pcp description with power-commutator relations. A pcp with combinatorial collector uses a power-commutator presentation, while a pcp with single collector uses a power-conjugate presentation. In that case the function `DefineConjugatePcp` has to be used.

Note that x must be a normal word of P (see `NormalWordPcp`) and the difference d must obey the collector dependent requirements. So if a combinatorial collector is installed, the central weight of d in P must be higher than or equal to the sum of the weights of g_i and g_j in P , otherwise an error is raised (see `DefineCommPcp`).

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> DefineCentralWeightsPcp( P, [ 1, 1, 2, 2, 2 ] );
gap> DefineCommPcp( P, 2, 1, a[3]^2 );
gap> CommPcp( P, 2, 1 );
a3^2
gap> SubtractCommPcp( P, 1, a[4] );
gap> CommPcp( P, 2, 1 );

a3^2*a4^6
```

`DefineCentralWeightsPcp(P , W)`

Let P be a pcp description with generators (g_1, \dots, g_n) and combinatorial collector. Let W be a list of n positive integers w_1, \dots, w_n . Then `DefineCentralWeightsPcp` defines the central weight of g_i in P to be w_i .

Note that `Pcp` assigns central weights 1 to all generators when creating a new pcp description with combinatorial collector. Therefore, in order to change the presentation of such a description, the weights have to be changed prior to calling `DefinePowerPcp` or `DefineCommPcp`. These functions will not adjust the central weights automatically.

Note that the central weights must obey the weight condition described in `SetCollectorAgGroup`, otherwise an error is raised.

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> DefineCentralWeightsPcp( P, [ 1, 1, 2, 2, 2 ] );
gap> DefineCommPcp( P, 2, 1, a[3]^2 );
gap> CommPcp( P, 2, 1 );

a3^2
```

CentralWeightsPcp(*P*)

Let P be a pcp description with generators (g_1, \dots, g_n) and combinatorial collector. Then `CentralWeightsPcp` returns a list of the positive integers w_1, \dots, w_n where w_i is the central weight of g_i in P .

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> CentralWeightsPcp( P );
```

```
[ 1, 1, 1, 1, 1 ]
```

All of the following functions expect normal words of a pcp description P as arguments. They will not call `NormalWordPcp` to replace an argument by an equivalent normal word.

DifferencePcp(*P*, *u*, *v*)

`DifferencePcp` returns a normal word d of P having as exponent vector the difference of the exponent vectors of the normal words u and v in P .

Let G be the group described by P and let (g_1, \dots, g_n) be the generators of P with relative orders o_1, \dots, o_n . Then a normal word u of P is written as $g_1^{u_1} \dots g_n^{u_n}$ for integers u_i such that $0 \leq u_i < o_i$ and v is written as $g_1^{v_1} \dots g_n^{v_n}$ for integers v_i such that $0 \leq v_i < o_i$. Then `DifferencePcp` returns a normal word $d = g_1^{d_1} \dots g_n^{d_n}$ of P with integers d_i such that $0 \leq d_i < o_i$ and $d_i \equiv u_i - v_i \pmod{o_i}$.

An error is raised if u and v are not normal words of P .

DepthPcp(*P*, *g*)

`DepthPcp` returns the depth of a normal word g in P with respect to the pcp description P as an integer (see `Agwords`).

An error is raised if g is not a normal word of P .

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> DepthPcp( P, a[3]^6 * a[4]^5 );
```

```
3
```

ExponentsPcp(*P*, *g*)

`ExponentsPcp` returns the exponent vector of a normal word g of P with respect to the pcp description P as list of integers.

Let G be the group described by P and let (g_1, \dots, g_n) be the generators of P . Then a normal word g of P is written as $g_1^{v_1} \dots g_n^{v_n}$ for integers v_i fulfilling the condition described in “Collector Independent Functions”. The exponent vector of g with respect to P is the list $[v_1, \dots, v_n]$.

An error is raised if g is not a normal word of P .

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> ExponentsPcp( P, a[3]^6 * a[4]^5 );

[ 0, 0, 6, 5, 0 ]
```

`SumPcp(P, u, v)`

`SumPcp` returns a normal word s of P having as exponent vector the sum of the exponent vectors of the normal words u and v in P .

Let G be the group described by P and let (g_1, \dots, g_n) be the generators of P with relative orders o_1, \dots, o_n . Then a normal word u of P is written as $g_1^{u_1} \dots g_n^{u_n}$ for integers u_i such that $0 \leq u_i < o_i$ and v is written as $g_1^{v_1} \dots g_n^{v_n}$ for integers v_i such that $0 \leq v_i < o_i$. Then `SumPcp` returns a normal word $s = g_1^{s_1} \dots g_n^{s_n}$ of P with integers s_i such that $0 \leq s_i < o_i$ and $s_i \equiv u_i + v_i \pmod{o_i}$ where o_i .

An error is raised if u and v are not normal words of P .

`TailDepthPcp(P, g)`

`TailDepthPcp` returns the tail depth of a normal word g in P with respect to a pcp description P as an integer.

An error is raised if g is not a normal word of P .

```
gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> TailDepthPcp( P, a[3]^6 * a[4]^5 );
```

4

4.2.1 Composition Functions for Elements of Pcps

Words of a pcp description P can be passed as arguments to the following functions. In case such a word is not normal it is replaced by an equivalent normal

word by the function `NormalWordPcp` (see `NormalWordPcp`). Note that `NormalWordPcp` also uses the presentation and collector stored in P to compute a normal word equivalent to the word passed as argument.

Let P have generators (g_1, \dots, g_n) such that the relative order of g_i in P is o_i . Then a word w of P is a product in the generators of P and their inverses. A *normal word* w of P is a product in g_1, \dots, g_n such that $w = g_1^{e_1} \dots g_n^{e_n}$ for nonnegative integers $e_i < o_i$.

`CommPcp(P, g, h)`

`CommPcp(P, i, j)`

Let P be a pcp description with generators (g_1, \dots, g_n) . For the first type of call `CommPcp` returns a normal word equivalent to the commutator $Comm(g, h)$ in P for words g and h of P (see `NormalWordPcp`). Here g and h have to be (not necessarily normal) words in the generators of P .

For the second type of call P has to be a pc-presentation and $1 \leq j < i \leq n$. Then the function `CommPcp` returns the right hand side w_{ij} of the commutator relation $w_{ij} = Comm(g_i, g_j)$.

`ConjugatePcp(P, g, h)`

`ConjugatePcp(P, i, j)`

Let P be a pcp description with generators (g_1, \dots, g_n) . For the first type of call `ConjugatePcp` returns a normal word equivalent to the conjugate g^h of g under h in P for words g and h of P (see `NormalWordPcp`). Here g and h have to be (not necessarily normal) words in the generators of P .

For the second type of call P must be a pc-presentation and $1 \leq j < i \leq n$. Then `ConjugatePcp` returns the right hand side w'_{ij} of the conjugate relations of $w'_{ij} = g_j^{g_i}$

`LeftQuotientPcp(P, g, h)`

`LeftQuotientPcp` returns a normal word equivalent to left quotient $g^{-1}h$ in P for words g and h of P (see `NormalWordPcp`). Here g and h have to be (not necessarily normal) words in the generators of P .

```

gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> LeftQuotientPcp( P, a[2]^8 * a[4], a[3]^3 );

a2^6*a3^3*a4^6

```

NormalWordPcp(P , w)

Let P be a pcp description and let w be a word in the generators of P . Then NormalWordPcp returns an equivalent normal word g in P obtained from w using the collector and relations of P .

```

gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> NormalWordPcp( P, a[5]^8 * a[4]^5 * a[1]^1 );

a1^6*a4^5*a5

```

PowerPcp(P , g , e)

PowerPcp(P , i)

Let P be a pcp description with generators (g_1, \dots, g_n) . For the first type of call PowerPcp returns a normal word equivalent to the e^{th} power of g in P for a word g in P and an integer e (see NormalWordPcp). Here g and h have to be (not necessarily normal) words in the generators of P .

For the second type of call i must be an integer between 1 and n , then PowerPcp returns the right hand side w_{ii} of the power relation $w_{ii} = g_i^{o_i}$ where o_i is the relative order of g_i .

```

gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> PowerPcp( P, a[2]^2 * a[4], 3 );
a2^6*a4^3
gap> PowerPcp( P, 2 );

IdWord

```

ProductPcp(P , g , h)

ProductPcp returns a normal word equivalent to the product gh in P for words g and h of P (see NormalWordPcp). Here g and h have to be (not necessarily normal) words in the generators of P .

```

gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> ProductPcp( P, a[2]^8 * a[4], a[3]^3 );

a2*a3^3*a4

```

QuotientPcp(P , g , h)

QuotientPc returns a normal word equivalent to the quotient gh^{-1} in P for words g and h of P (see NormalWordPcp). Here g and h have to be (not necessarily normal) words in the generators of P .

```

gap> P := Pcp( "a", 5, 7, "combinatorial" );
<Pcp: 5 generators, combinatorial collector>
gap> a := GeneratorsPcp( P );
[ a1, a2, a3, a4, a5 ]
gap> QuotientPcp( P, a[2]^8 * a[4], a[3]^3 );

a2*a3^4*a4

```

5 Appendix 1

Some sample runs

At various stages in this work we have looked at sample presentations to allow us to compare the current performance of the PQA in GAP and the **C** standalone. This has allowed us to improve the performance of both programs. The most recent improvement has been a tuning of the **C** standalone by speeding up the most significant inner loop. This happened in the course of preparing the current figures. The given figures show how effective both programs are.

The following presentations are used as test examples:

$$G_1 = \{a, b \mid \quad\}$$

$$G_2 = \{a, b \mid a^p, b^p, [b, 5a]\}$$

$$G_3 = \{a, b, c_1, c_2, c_3, c_4 \mid c_1 := [b, a], c_2 := [c_1, a], c_3 := [c_2, a], c_4 := [c_3, a], [c_4, a]\}$$

$$G_4 = \{a, b, c_1, c_2, c_3, c_4 \mid c_1 := [b, a], c_2 := [c_1, a], c_3 := [c_2, a], c_4 := [c_3, a], \\ [c_1, b], [c_4, a]\}$$

The structure of the relations in a presentation influences the performance of the implementations of the PQA. Therefore we chose the presentations such that they contain different types of relations. Presentation G_1 contains no relations. Presentation G_2 contains a left normed commutator of length 5. Presentations G_3 and G_4 describe groups in which such a commutator relation is satisfied by the generators a and b , however in the presentation this relation is not written as a left normed commutator of length 5. Instead, additional generators are introduced so that the relations can be expressed as commutators of generators. The reason for this is to display the effect of the way GAP stores relations has on the performance of the GAP implementation as discussed in Section 3.2.

| Presentation | Prime | Class | Times (sec) | |
|--------------|-------|-------|-------------|--------|
| | | | GAP 3.3 | ANU PQ |
| G_1 | 2 | 12 | 380.2 | 138.0 |
| | 5 | 12 | 522.8 | 158.0 |
| | 7 | 12 | 542.0 | 210.0 |
| | 17 | 11 | 247.4 | 148.0 |
| G_2 | 5 | 12 | 107.3 | 16.5 |
| | 7 | 12 | 317.1 | 29.0 |
| | 17 | 11 | 1605.0 | 39.6 |
| G_3 | 5 | 12 | 42.1 | + |
| | 7 | 12 | 62.0 | + |
| | 17 | 11 | 57.4 | + |
| G_4 | 7 | 15 | 6.7 | 1.6 |
| | 17 | 17 | 60.3 | 31.5 |
| | 23 | 17 | 159.0 | 97.1 |
| | 31 | 17 | 460.0 | 320.3 |

The computations marked $+$ were not performed since the ANU PQ handles G_2 without expanding the commutator relation. Therefore the times for the ANU PQ on G_2 and G_3 are essentially the same.

Some comments are in order. The reason for reporting on the calculation of the class-11 17-quotient of the free group of rank 2 (G_1) is that the current combinatorial collector in GAP runs into an overflow in class 12. The **C** standalone does not because it uses the O'Brien & Vaughan-Lee collector which avoids overflows for all primes less than 1000.

There are further questions to explore: why is the GAP version less prime dependent? It is conceivable that once the overflow mentioned above has been dealt with there will be cases where the GAP version out-performs the **C** standalone.

6 Appendix 2

The listing

```
#####
##
##A pq.g GAP library Werner Nickel
##A & Alice Niemeyer
##
##A @(#)Id: pq.g,v 3.10 1992/08/19 11:24:41 fceller Rel $
##
##Y Copyright 1991, Mathematics Research Section, ANU, Canberra, Australia
##
## This file contains an implementation of the  $p$ -quotient algorithm as
## specified by Havas and Newman ("Application of computers to questions
## like those of Burnside" in Lecture Notes in Mathematics 806, Springer
## 1980). Many tricks and techniques used in this implementation are not
## published and we learnt them in many discussions from Mike Newman. He
## also explained and pointed out to us important sections of the FORTRAN
## code of the Canberra Nilpotent Quotient Program, which is the only
## (written) source of information on the implementation and the fine
## details of the  $p$ -Quotient Algorithm.
##
## This implementation is written for GAP 3.1 and Frank Celler's extension
## of the Pc-Module which makes it possible to handle central extensions of
##  $p$ -groups more efficiently than this was possible in GAP 2.6. The
## current internal code supporting the needs for the  $p$ -quotient algorithm
## was written by Frank Celler according to the observations made in many
## experimental versions of the  $p$ -Quotient Algorithm. At the time of
## writing the current implementation runs within a factor of two of the
## FORTRAN standalone.
##
## THE PQ-DATA-STRUCTURE RECORD:
##
## The  $p$ -Quotient Algorithm deals with finite presentations for
##  $p$ -quotients of groups. In the implementation, we store a finite presen-
## tation of a  $p$ -quotient  $Q$  of a group in a record which we will call
## the PQp record. It contains the following record fields :
##
## .generators A list of abstract generators which generate  $Q$ 
## .pcp An internal pcp-description for  $Q$ 
## .identity The identity element of the group  $Q$ 
## .dimensions A list, where dimensions[i] is the dimension
## of the  $i$ -th factor in the lower exponent- $p$ 
## central series calculated by the  $p$ -Quotient
## Algorithm
## .prime An integer, which is the prime  $p$ 
## .one A finite field element, the unit of  $\mathbb{GF}(p)$ 
## .maxgennr An integer used to create names for generators
## .nrgens An integer, the number of generators of the
## previously computed quotient, used to identify
## the last element in .generators which does not
## lie in the  $p$ -multiplier
## .nrnewgens An integer specifying how many new generators
## are still alive. It is used to determine
## quickly, whether the  $p$ -quotient is completed
##
## .defining A list of two sets :
```

```

##      .defining[1]      - contains the triangle-indices of those
##                        commutators that define a new ( or, in
##                        the process of the $p$-cover algorithm
##                        also, pseudo ) generator
##      .defining[2]      - contains the indices of those generators
##                        whose $p$-th powers define a new ( or
##                        pseudo ) generator
##      .definedby        A list which contains the definition of the
##                        $k$-th generator in the $k$-th place. There
##                        are three different types of entries, namely
##                        lists, positive and negative integers :
##      [ j, i ]          - the generator is defined as commutator of
##                        the $j$-th and the $i$-th element in
##                        .generators
##      i                  - the generator is defined as $p$-th power
##                        of the $i$-th element in .generators
##      -i                 - the generator is defined as an image of
##                        the $i$-th generator in the presentation
##                        for $G$, consequently it must be of
##                        weight 1
##      .isdefinition      A list of two boolean lists
##      .isdefinition[1]   - the $(j-2)d+i$-th entry is true if the
##                        commutator of the $j$-th and the $i$-th
##                        generator is a definition, else false
##                        ($d$=.dimensions[1])
##      .isdefinition[2]   - the $i$-th entry is true if the $p$-th
##                        power of the $i$-th generator is a
##                        definition, else false
##      .epimorphism       A list containing an image in $Q$ for each
##                        generator of $G$. The image is one of the
##                        following:
##      i                   - the image is the $i$-th element of
##                        .generators
##      word                 - the image is word, which is a word in the
##                        generators of $Q$
##      .pInverse          A list containing the inverse of $i$ in $GF(p)$
##                        at position $i$.
##
##

```

```

#####
##
##F InfoPQ1( <arg> ) . . . . . package information
##F InfoPQ2( <arg> ) . . . . . package debug information
##

```

```

if not IsBound( InfoPQ1 ) then InfoPQ1 := Print; fi;
if not IsBound( InfoPQ2 ) then InfoPQ2 := Ignore; fi;

```

```

#####
##
##F PQpOps . . . . . presentation operations
##
PQpOps := rec();

```

```

#####
##
##F PQpOps.Print( <P> ) . . . . . print a PQp record
##
PQpOps.Print := function( P )

    local i, j, com, lst;

    Print("PQp( rec( \n");
    Print(" generators := ", P.generators, "\n" );
    Print(" definedby := ", P.definedby, "\n" );
    Print(" prime := ", P.prime, "\n" );

```

```

Print(" dimensions := ", P.dimensions, "\n" );
Print(" epimorphism := ", P.epimorphism, "\n");

Print(" powerRelators := [ " );
for i in [1 .. Length(P.generators)] do
  if PowerPcp(P.pcp, i) = IdWord then
    Print(P.generators[i], "~", P.prime);
  else
    Print(P.generators[i], "~", P.prime, "/"( "PowerPcp(P.pcp, i), " ) );
  fi;
  if i < Length(P.generators) then
    Print(" ");
  else
    Print(" ],\n");
  fi;
od;

com := false;
lst := [];
Print(" commutatorRelators := [ " );
for i in [2 .. Length(P.generators)] do
  for j in [1 .. i-1] do
    if CommPcp(P.pcp, i, j) <> IdWord then
      if com then
        Print(" ");
      else
        com := true;
      fi;
      Print("Comm(", P.generators[i], ", ", P.generators[j],
            ")/(" CommPcp(P.pcp, i, j), " )");
      Add(lst, [i,j]);
    fi;
  od;
od;
Print(" ],\n definingCommutators := ", lst, " )" );

end;

#####
##
## PQpOps.AgGroup . . . . . convert into an ag group
##
PQpOps.AgGroup := function( P )
  return AgGroupPcp(P.pcp);
end;

#####
##
## PQpOps.FpGroup . . . . . convert into an fp group
##
PQpOps.FpGroup := function( P )
  return FpGroup(AgGroupPcp(P.pcp));
end;

#####
##
## PQp( <R> ) . . . . . restore a PQp record
##
## 'PQp' takes as argument a GAP record containing all information necessary
## to restore the internal pcp-description, such as the GAP record created
## by the function 'Print'.
##
PQp := function(G)

  local      P, i, j, cw, ii, r, rr, k, w;

```

```

P := Pcp("a", Length(G.generators), G.prime, "combinatorial");
P := rec( generators      := GeneratorsPcp(P),
          pcp            := P,
          prime          := G.prime,
          one            := Z(G.prime)^0,
          nrgens         := Length(G.generators),
          maxgenr        := Length(G.generators)+1,
          nrnewgens      := Length(G.generators),
          defining       := [[ ], [ ]],
          definedby      := Copy(G.definedby),
          isdefinition   := [ BlistList([ ], [ ]),
                             BlistList([ ], [ ] ) ],
          dimensions     := Copy(G.dimensions),
          epimorphism    := Copy(G.epimorphism),
          isPQp          := true,
          operations     := PQpOps
        );

for i in [ 1 .. Length(G.generators) ] do
  Add(P.isdefinition[2], false);
od;
for i in [ 1 .. Length(P.generators)*P.dimensions[1] ] do
  Add(P.isdefinition[1], false);
od;
for k in [ 1 .. Length(P.definedby)] do
  if IsList(P.definedby[k]) then
    j := P.definedby[k][1];
    i := P.definedby[k][2];
    P.isdefinition[1][(j-2)*P.dimensions[1]+i] := true;
    AddSet(P.defining[1], TriangleIndex(j,i));
  elif P.definedby[k] > 0 then
    P.isdefinition[2][P.definedby[k]] := true;
    AddSet(P.defining[2], P.definedby[k]);
  fi;
od;

P.identity := P.generators[1]^0;

# compute inverses of 1 .. p-1 in GF(p) as suggested by M. Sch"onert
r := PrimitiveRootMod( P.prime ); rr := r;
i := 1/r mod P.prime; ii := i;
P.pInverse := [1];
while rr <> 1 do
  P.pInverse[rr] := ii;
  rr := rr * r mod P.prime;
  ii := ii * i mod P.prime;
od;

cw := [ ];
for i in [ 1 .. Length(G.dimensions) ] do
  for j in [ 1 .. G.dimensions[i] ] do
    cw[Length(cw)+1] := i;
  od;
od;
DefineCentralWeightsPcp(P.pcp, cw);

for i in [ 1 .. Length(G.generators) ] do
  w := G.powerRelators[i]^-1 * G.generators[i]^G.prime;
  DefinePowerPcp(P.pcp, i, MappedWord(w,G.generators,P.generators));
od;

for i in [ 1 .. Length(G.commutatorRelators) ] do
  w := G.commutatorRelators[i]^-1
    * Comm(G.generators[G.definingCommutators[i][1]],
          G.generators[G.definingCommutators[i][2]]);
  DefineCommPcp(P.pcp,
    G.definingCommutators[i][1],
    G.definingCommutators[i][2],

```

```

        MappedWord(w, G.generators, P.generators));
    od;

    for i in [ 1 .. Length(P.epimorphism) ] do
        if not IsInt(P.epimorphism[i]) then
            P.epimorphism[i] :=
                MappedWord(G.epimorphism[i],G.generators,P.generators);
        fi;
    od;
    return P;
end;

#####
##
## SavePQp( <file>, <P>, <N> ) . . . . . saves <P> to file
##
## 'SavePQp' prints the PQp record <P> to the file <file> with name <N>.
##
PQpOps.Save := function( file, P, N )

    local      i;

    PrintTo(file, "");
    for i in [ 1 .. Length(P.generators) ] do
        AppendTo(file, P.generators[i], " := AbstractGenerator(\"",
            P.generators[i],"\");\n");
    od;
    AppendTo(file, N, " := ", P, ";");

end;
SavePQp := PQpOps.Save;

#####
##
## InitPQp( <rank>, <prime> ) . . . . . initializes a PQp record
##
## 'InitPQp' creates a PQp record for an elementary abelian group of rank
## <rank> and order <prime>^<rank>.
##
InitPQp := function( rank, prime )

    local      i, ii, r, rr, P;

    P := rec();
    if rank > 0 then
        P.pcp      := Pcp( "g", rank, prime, "combinatorial" );
        P.generators := GeneratorsPcp( P.pcp );
        P.identity  := P.generators[1]^0;
    else
        P.generators := [];
    fi;
    P.dimensions := [];
    P.prime      := prime;
    P.maxgenr    := Length(P.generators)+1;
    P.nrgens     := 0;
    P.nrnewgens  := 0;
    P.one        := Z(P.prime)^0;
    P.defining   := [ [], [] ];
    P.definedby  := [];
    P.isdefinition := [ BlistList( [], [] ), BlistList( [], [] ) ];
    P.epimorphism := [];
    P.pInverse   := [1];
    P.unused     := 0;
    P.isPQp      := true;
    P.operations := PQpOps;

```

```

# initialise the .definedby entries for the first <rank> generators
for i in [ 1 .. Length(P.generators)] do
  P.definedby[i] := -i;
od;

# compute inverses of 1 .. p-1 in GF(p) as suggested by M. Sch"onert
r := PrimitiveRootMod( P.prime ); rr := r;
i := 1/r mod P.prime;          ii := i;
while rr <> 1 do
  P.pInverse[rr] := ii;
  rr := rr * r mod P.prime;
  ii := ii * i mod P.prime;
od;

# initialise .epimorphism
for i in [1 .. rank] do P.epimorphism[i] := i; od;
return P;

end;

#####
##
##F AddGeneratorsPQp( <P>, <cl> ) . . . . . adds new/pseudo generators
##
## This function adds the new and pseudo generators to the PQp record <P>
## corresponding to weight <cl>. Call a generator $g$ a new generator if
## <cl> is the maximal class of the $p$-quotient and call it a pseudo
## generator otherwise. Whether <cl> is the maximal class is tested in the
## code by comparing <cl> to the length of '<P>.dimensions'.
##
## The rule for adding new/pseudo generators of weight <cl> is :
## 1) for a relation $[ b, a ] = w$ add a new/pseudo generator $g$ such that
## the relation becomes $[ b, a ] = wg$, if the relation is not a
## definition and $wt(a) = 1$ and $wt(b) = <cl>-1$. Call this the
## definition of the new/pseudo generator $g$.
## 2) for a relation $a^p = w$ add a new/pseudo generator $g$ such that the
## $wt(a) = 1$ or $a$ is defined as a $p$-th power and $wt(a) = <cl>-1$.
## Call this the definition of $g$.
##
AddGeneratorsPQp := function( P, cl )

  local i, j, x, k, l;

  # set l to the number of generators of weight less or equal cl
  l := 0;
  for i in [1 .. cl] do l := l + P.dimensions[i]; od;

  # Add new/pseudo generators to the commutators
  k := Length(P.definedby) + 1;
  for j in [ l-P.dimensions[cl]+1 .. l ] do

    # define pseudo generators for commutator relations
    for i in [1 .. Minimum(P.dimensions[1],j-1)] do

      # add new/pseudo generators to all rhs of relations, which are
      # not definitions
      x := TriangleIndex(j,i);
      if cl = Length(P.dimensions) then
        AddSet( P.defining[1], x );
        P.definedby[k] := [ j, i ];
        DefineCommPcp( P.pcp, j, i, P.generators[k] );
        k := k + 1;
      elif not x in P.defining[1] then
        AddSet( P.defining[1], x );
        P.definedby[k] := [ j, i ];
        AddCommPcp( P.pcp, j, i, P.generators[k] );
        k := k + 1;
      end if;
    end for;
  end for;
end function;

```

```

        fi;
    od;
od;

# add new/pseudo generators to the  $p^{\text{th}}$ -powers
for i in [1-P.dimensions[cl]+1 .. 1] do

    # Add a new/pseudo generator for each generator  $a_i$ , for which
    #  $a_i$  is not a definition and  $a_i$  is defined as  $p^{\text{th}}$  power
    if IsInt( P.definedby[i] ) then
        if cl = Length(P.dimensions) then
            AddSet( P.defining[2], i );
            P.definedby[k] := i;
            DefinePowerPcp( P.pcp, i, P.generators[k] );
            k := k + 1;
        elif not i in P.defining[2] then
            AddSet( P.defining[2], i );
            P.definedby[k] := i;
            AddPowerPcp( P.pcp, i, P.generators[k] );
            k := k + 1;
        fi;
    fi;
od;

end;

#####
##
##F DefineGeneratorsPQp( <P> ) . . . . . defines new/pseudo generators
##
## DefineGeneratorsPQp() creates the new and pseudo generators necessary for
## the computation of the  $p^{\text{th}}$ -cover of the group defined by the PQp record
## <P> and adds them to the internal power commutator presentation '<P>.pcp'
## of the PQp record by the function 'ExtendCentralPcp'.
##
DefineGeneratorsPQp := function( P )

    local l, i, nr_newgens, newgens, cl;

    # l will be the number of generators of weight less or equal <cl>
    l := Sum( P.dimensions );
    newgens := [];
    for cl in Reversed( [1 .. Length(P.dimensions)] ) do
        nr_newgens := 0;

        # The following for-loop can be avoided by storing the number of
        # generators per class defined by  $p^{\text{th}}$ -powers in the group
        # record. But that should not have a significant impact on the
        # performance of the PQ.
        for i in [ 1-P.dimensions[cl]+1 .. 1] do

            # Test if  $a_i$  was defined as a  $p^{\text{th}}$  power
            if IsInt(P.definedby[i]) then
                nr_newgens := nr_newgens + 1;
            fi;
        od;

        # Compute the number of commutators of the form [ cl, 1 ]. Note the
        # different handling in case cl = 1. If cl is the maximal class this
        # gives us the number of new generators.
        if cl = 1 then
            nr_newgens := nr_newgens + P.dimensions[1]*(P.dimensions[1]-1)/2;
        else
            nr_newgens := nr_newgens + P.dimensions[1]*P.dimensions[cl];
        fi;

        # Subtract from the number of generators those that have survived.
        # This is the number of pseudo generators.

```

```

    if cl < Length(P.dimensions) then
        nr_newgens := nr_newgens - P.dimensions[cl+1];
    fi;

    if cl = Length(P.dimensions) then
        P.nrnewgens := P.nrgens + nr_newgens;
        P.nrnewgensleft := nr_newgens;
    fi;

    # Create the new/pseudo generators.
    if cl < Length(P.dimensions) then
        for i in [ 1 .. nr_newgens ] do      # "p" as in "p"seudo
            Add( newgens, "p" );
        od;
    else
        for i in [ 1 .. nr_newgens ] do
            Add(newgens,ConcatenationString("a",StringInt(P.maxgennr)));
            P.maxgennr := P.maxgennr + 1;
        od;
    fi;
    l := l - P.dimensions[cl];
od;

# define some pseudo generators to lift the epimorphism
for i in [ 1 .. Length(P.epimorphism) ] do
    if not IsInt( P.epimorphism[i] ) then
        Add( newgens, "e" ); # "e" as in epimorphism
    fi;
od;
ExtendCentralPcp( P.pcp, newgens, P.prime );
P.generators := GeneratorsPcp( P.pcp );

return P;

end;

#####
##
##F TailsPQp( <P> ) . . . . . computes a covering presentation for <P>
##
## 'TailsPQp' computes a not necessarily consistent, covering presentation
## for <P>. For each class cl computed so far 'AddGeneratorsPQp' is called
## to add the new/pseudo generators created by a call to
## 'DefineGeneratorsPQp' for this class.
##
## 'AddGeneratorsPQp' modifies the relations of the form
## 1)  $[ b, a ] = w$  with  $\text{wt}(b) = cl$  and  $\text{wt}(a) = 1$ 
## 2)  $c^p = v$  with  $\text{wt}(c) = cl$  and  $c$  is either defined as  $p$ -th
## power or  $\text{wt}(c) = 1 (=cl)$ .
##
## A theoretical argument shows that for all other relations the word in the
## new/pseudo generators with which to modify each relation (called the
## 'tail' of the relation) can be computed. This is done in this function
## and the relations are modified accordingly.
##
TailsPQp := function( P )

    local    i, j, k, clnrgen, cl, g, x, y, z, endcl, enddim, Q;

    Q := P.pcp;
    clnrgen := Length(P.generators);

    P := DefineGeneratorsPQp(P);
    for cl in Reversed( [1 .. Length(P.dimensions)] ) do

        # add new/pseudo generators
        AddGeneratorsPQp(P, cl);

        # Compute the tails of the new/pseudo generators. First the tails of

```

```

# the $p^{th}$-powers are computed
for i in Reversed([clnrgen-P.dimensions[cl]+1 .. clnrgen]) do

  # compute tails for $p^{th}$-powers $a_i^p$ for which $a_i$ is
  # defined as a commutator
  if IsList(P.definedby[i]) then
    g := P.definedby[i][1];
    y := P.generators[ g ];
    z := P.generators[ P.definedby[i][2] ];

    # (y^p*z) / (y^(p-1) * (y*z))
    x := DifferencePcp( Q,
      ProductPcp( Q, PowerPcp( Q, g ), z),
      ProductPcp( Q,
        PowerPcp( Q, y, P.prime-1 ),
        ProductPcp( Q, y, z ) ) );
    if x <> P.identity then AddPowerPcp( Q, i, x ); fi;
  fi;
od;
clnrgen := clnrgen - P.dimensions[cl];

# Next compute the tails of the commutators
endcl := P.nrgens;
for k in [ cl .. Length(P.dimensions) ] do
  endcl := endcl - P.dimensions[k];
od;

enddim := P.dimensions[1]+1;
k := 1;
while cl-k >= k+1 do
  for i in [enddim .. enddim+P.dimensions[k+1]] do
    if IsList(P.definedby[i]) then
      # the second generator is defined as commutator
      y := P.generators[ P.definedby[i][1] ];
      z := P.generators[ P.definedby[i][2] ];
      g := ProductPcp( Q, y, z );
      j := endcl;
      while j > i and j > endcl - P.dimensions[cl-k] do
        x := P.generators[j];
        # ((x*y)*z) / (x*(y*z))
        x := DifferencePcp( Q,
          ProductPcp( P.pcp,
            ProductPcp(Q,x,y), z),
            ProductPcp(Q,x,g) );
        if x <> P.identity then
          AddCommPcp( Q, j, i, x );
        fi;
        j := j - 1;
      od;
    elif P.definedby[i] > 0 then

      # The second generator is defined as $p$-th power
      # and is not one of the first generators (which are
      # defined by the epimorphism).
      z := P.definedby[i];
      y := P.generators[z];
      g := PowerPcp( Q, z );
      z := PowerPcp( Q,y,P.prime-1);
      j := endcl;
      while j > i and j > endcl - P.dimensions[cl-k] do
        x := P.generators[j];
        # ((x*y) * y^(p-1)) / (x*y^p)
        x := DifferencePcp( Q,
          ProductPcp( Q,
            ProductPcp(Q,x,y), z),
            ProductPcp(Q,x,g) );
        if x <> P.identity then
          AddCommPcp( Q, j, i, x );
        fi;
      od;
    end if;
  end for;
end while;

```

```

                j := j - 1;
            od;
        fi;
    od;
    endcl := endcl - P.dimensions[cl-k];
    enddim := enddim + P.dimensions[k+1];
    k      := k + 1;
od;
od;

return P;
end;

#####
##
## EchelonizePQp( <P>, <sys>, <w> ) . . . . . echelonize <w> along <sys>
##
## 'EchelonizePQp' takes the word <w> in the generators of <P> of highest
## weight and views it as a linear equation over $GF(p)$. This is possible,
## because the generators of highest weight are central and of order $p$.
## Furthermore, <sys> is a system of words in the generators of <P> of
## highest weight, also regarded as linear equations and in reduced form.
## 'EchelonizePQp' reduces <w> along the system of linear equations <sys>.
## If <w> is not the identity after the echelonisation, it is added to the
## system of equations.
##
EchelonizePQp := function( P, sys, w )

    local wd, t, i;

    w := TailReducedPcp( P.pcp, sys.ls, w );

    if w = P.identity then return 1; fi;

    wd := TailDepthPcp( P.pcp, w );
    w := PowerPcp( P.pcp, w, P.pInverse[ ExponentPcp(P.pcp,w,wd) ] );

    sys.ls[ wd ] := w;
    for i in sys.del do
        t := ExponentPcp( P.pcp, sys.ls[i], wd );
        if t > 0 then
            sys.ls[i] := DifferencePcp(P.pcp,sys.ls[i],PowerPcp(P.pcp,w,t));
        fi;
    od;
    Add( sys.del, wd );
    if wd <= P.nrnewgens then
        P.nrnewgensleft := P.nrnewgensleft - 1;
        if P.nrnewgensleft = 0 then return 0; fi;
    fi;

    return 1;

end;

#####
##
## ConsistencyPQp( <P> ) . . . . . determine inconsistencies
##
## 'ConsistencyPQp' evaluates all the relations of the consistency test.
## Each relation which is not satisfied yields a linear equation. The set of
## linear equations is returned. In the case that <P> is a consistent
## presentation the set of linear equations is empty. (cf. M.R. Vaughan-Lee:
## "An Aspect of the Nilpotent Quotient Algorithm", in Computational Group
## Theory, edited by Michael D. Atkinson)
##
ConsistencyPQp := function( P )

    local i, j, k, a, b, c,d, wtb, nrwt, linsys, wt, ai, ap;

```

```

linsys := rec( ls := [], del := [] );

# store in nrwt[i] the number of generators of weight less than i
nrwt := [0];
for i in [2 .. Length(P.dimensions)+1 ] do
  nrwt[i] := nrwt[i-1] + P.dimensions[i-1];
od;

# Loop through all possible weights a consistency relation can have.
for wt in Reversed( [ 3 .. Length(P.dimensions)+1 ] ) do
  InfoPQ2( "#I ConsistencyPQp: (a^p) * a = a * (a^p)\n" );

  # Check the relations $a^p*a = a*a^p$
  if wt mod 2 = 1 then
    # run through all $a$ with $2*wt(a) + 1 = wt$
    for i in [nrwt[(wt-1)/2]+1 .. nrwt[(wt-1)/2+1] ] do
      a := P.generators[i];
      ap := PowerPcp( P.pcp, i );
      a := DifferencePcp( P.pcp,
        ProductPcp( P.pcp, ap, a ),
        ProductPcp( P.pcp, a, ap ));
      if a <> P.identity then
        if EchelonizePQp( P, linsys, a ) = 0 then
          return 0;
        fi;
      fi;
    od;
  fi;
  InfoPQ2( "#I ConsistencyPQp: b*(a^p) = (b*a)*a^{(p-1)}\n" );

  # Check the consistency relation $b*(a^p) = b*a*(a^{(p-1)})$
  # for b > a.
  # wt = wt( b * a^p ) = wt(a) + wt(b) + 1 = wt
  # Loop through possible weights for a, namely wt(a) = i
  for i in [1 .. wt-2] do
    for k in [nrwt[i]+1 .. nrwt[i+1]] do
      a := P.generators[k];
      ap := PowerPcp( P.pcp, k );
      ai := PowerPcp( P.pcp, a, P.prime-1 );
      d := nrwt[wt-i];
      # wt(b) = wt-i-1, thus d-1 is the last choice for b
      if P.isdefinition[2][k] then
        # b > ap was done in TailsPQp, here we do b <= ap
        d := Minimum(Position(P.generators, ap), d);
      fi;
      # run through $b$ with $d >= b > a$ and $wt(b) = wt-i-1$
      for j in [Maximum(k+1, nrwt[wt-i-1]+1) .. d] do
        b := P.generators[j];
        b := DifferencePcp( P.pcp,
          ProductPcp( P.pcp, b, ap ),
          ProductPcp( P.pcp,
            ProductPcp( P.pcp, b, a ), ai ));
        if b <> P.identity then
          if EchelonizePQp( P, linsys, b ) = 0 then
            return 0;
          fi;
        fi;
      od;
    od;
  od;
  InfoPQ2( "#I ConsistencyPQp: (b^p)*a = b^{(p-1)}*b*a\n" );

  # Check the consistency relations $(b^p)*a = (b^{(p-1)})*b*a$
  # wt = wt(b^p*a) = wt(a) + wt(b) + 1 = wt(b) + 2
  # for b > a and wt(a) = 1
  # Loop through the generators of weight 1
  for i in [1 .. P.dimensions[1]] do
    a := P.generators[i];
    # run through all $b$ with $wt(b) = wt - 2$

```

```

for j in [Maximum(i+1,nrwt[wt-2]+1) .. nrwt[wt-1]] do
  if not P.isdefinition[1][(j-2)*P.dimensions[1]+i] then
    # if [b,a] is a definition of x, say then
    # b^p*a was used to compute the tail of x^p
    b := P.generators[j];
    b := DifferencePcp( P.pcp,
      ProductPcp( P.pcp, PowerPcp( P.pcp, j ), a),
      ProductPcp( P.pcp,
        PowerPcp( P.pcp,b,P.prime-1),
        ProductPcp(P.pcp,b,a)));
    if b <> P.identity then
      if EchelonizePQp( P, linsys, b ) = 0 then
        return 0;
      fi;
    fi;
  fi;
od;
od;
InfoPQ2( "#I ConsistencyPQp: (c*b)*a = c*(b*a)\n" );

# and relations $(c*b)*a = c*(b*a)$
for i in [1 .. P.dimensions[1]] do
  a := P.generators[i];
  # run through all $b$ with $wt(b) >= 1$ and
  # $wt(b) <= wt(c) = wt - wt(b) - wt(a) = wt - wt(b) - 1$,
  # which is the condition $2 * wt(b) <= wt - 1$
  wtb := 1;
  while 2 * wtb <= wt - 1 do
    for j in [Maximum(i+1,nrwt[wtb]+1) .. nrwt[wtb+1]] do
      b := P.generators[j];
      ap := ProductPcp(P.pcp,b,a);
      d := nrwt[wt-wtb];
      if P.isdefinition[1][(j-2)*P.dimensions[1]+i] then
        d := Minimum(d,
          Position(P.generators,CommPcp(P.pcp,j,i)));
      fi;
      # wt(c) = wt - wt(b) - 1
      for k in [Maximum(j+1,nrwt[wt-wtb-1]+1) .. d] do
        c := P.generators[k];
        c := DifferencePcp( P.pcp,
          ProductPcp(P.pcp,
            ProductPcp(P.pcp,c,b),a),
          ProductPcp(P.pcp, c, ap ) );
        if c <> P.identity then
          if EchelonizePQp( P, linsys, c ) = 0 then
            return 0;
          fi;
        fi;
      od;
    od;
    wtb := wtb + 1;
  od;
od;
od;
return linsys;
end;

#####
##
## ElimTailsPQp( <P>, <sys> ) . . . . . eliminate superfluous generators
##
## 'ElimTailsPQp' takes two parameters <P> and <sys>. <sys> is a list of
## words in the generators of <P> of highest weight. Each word specifies a
## relation between a superfluous generator and other generators of highest
## weight, by which the superfluous one is to be replaced. 'ElimTailsPQp'
## eliminates all occurrences of superfluous generators.

```

```

##
ElimTailsPQp := function( P, sys )

  local      words, del, wasDefComm, wasDefPow, defby,
             i, j, k, r, Q, tmp;

  if sys.ls <> [] then
    del := sys.del;
    words := sys.ls;
  fi;

  if sys.ls = [] or del = [] then
    P.unused := 0;
    Add( P.dimensions, Length( P.generators ) - P.nrgens );
    P.nrgens := Length( P.generators );
    i := P.dimensions[1];
    j := P.dimensions[Length(P.dimensions)];

    # Enlarge the boolean list .isdefinition[1] and .isdefinition[2]
    for k in [1..j] do Add( P.isdefinition[2], false ); od;
    for k in [ 1..j*i ] do Add(P.isdefinition[1], false); od;

    # Update the entries in the boolean list .isdefinition[1]
    # and .isdefinition[2]
    for k in [P.nrgens-j+1 .. P.nrgens] do
      if IsList(P.definedby[k]) then
        P.isdefinition[1][(P.definedby[k][1]-2)*i+P.definedby[k][2]]
          := true;
        elif P.definedby[k] > 0 then
          P.isdefinition[2][P.definedby[k]] := true;
        fi;
      od;

      return P;
    fi;

  if Length( del ) = Length( P.generators ) then
    P := InitPQp( 0, P.prime );
    return( P );
  fi;
  Q := P.pcp;

  # At first run through the sequence of tails and delete all the
  # definitions of new/pseudo generators
  wasDefComm := [];
  wasDefPow := [];
  for i in del do
    defby := P.definedby[ i ];
    if IsList(defby) then

      # Use 'SubtractCommPcp', since del[i] is a tail generator
      SubtractCommPcp( Q, defby[1], defby[2], words[i] );
      AddSet( wasDefComm, TriangleIndex(defby[1],defby[2]));
    elif defby > 0 then
      SubtractPowerPcp( Q, defby, words[i] );
      AddSet( wasDefPow, defby );
    else
      defby := -defby;
      if IsInt( P.epimorphism[ defby ] ) then
        P.epimorphism[defby] :=
          DifferencePcp( P.pcp, P.generators[i],words[i]);
      else
        P.epimorphism[defby] :=
          DifferencePcp( P.pcp, P.epimorphism[defby],words[i]);
      fi;
    fi;
  od;

  # In this loop we check all the commutator relations and loop through all

```

```

# generators of the previous step
for i in [1 .. P.nrgens] do
  # loop through all generators such that the
  # Triangle Index is defined
  for j in [ 1 .. i - 1 ] do
    if not TriangleIndex(i,j) in P.defining[1] then

      # The commutator does not define a new generator thus all
      # occurrences of generators to be eliminated on the right
      # hand side of this commutator relation have to be replaced
      # by corresponding words in the remaining generators.
      tmp := CommPcp( Q, i, j );
      if tmp <> P.identity then
        r := TailReducedPcp( Q, words, tmp );
        DefineCommPcp( Q, i, j, r );
      fi;
    fi;
  od;
od;

P.defining[1] := Difference( P.defining[1], wasDefComm );

# In this loop we check all the power relations and loop through all the
# generators
for i in [1 .. P.nrgens] do

  # Does the power define a generator ?
  if not i in P.defining[2] then

    # The power does not define a new generator thus all occurrences
    # of generators to be eliminated on the right hand side of this
    # power relation have to be replaced by corresponding words in
    # the remaining generators.
    tmp := PowerPcp( Q, i );
    if tmp <> P.identity then
      r := TailReducedPcp( Q, words, tmp );
      DefinePowerPcp( Q, i, r );
    fi;
  fi;
od;

P.defining[2] := Difference( P.defining[2], wasDefPow );

defby := [];
k := 1;
for i in [ 1 .. Length(P.definedby) ] do
  if not i in del then
    defby[k] := P.definedby[i];
    k := k + 1;
  fi;
od;
P.definedby := defby;

P.unused := Length( del );
Add( P.dimensions, Length( P.generators ) - P.nrgens - P.unused );
P.nrgens := Length( P.generators ) - Length( del );

ShrinkPcp( Q, del );
P.generators := GeneratorsPcp(Q);

i := P.dimensions[1];
j := P.dimensions[Length(P.dimensions)];

# Enlarge the boolean list .isdefinition[1] and .isdefinition[2]
for k in [1..j] do Add( P.isdefinition[2], false ); od;
for k in [ 1..j*i ] do Add(P.isdefinition[1], false); od;

# Update the entries in the boolean lists <P>.isdefinition[1] and
# <P>.isdefinition[2]

```

```

for k in [P.nrgens-j+1 .. P.nrgens] do
  if IsList(P.definedby[k]) then
    P.isdefinition[1][(P.definedby[k][1]-2)*i+P.definedby[k][2]]
      := true;
  elif P.definedby[k] > 0 then
    P.isdefinition[2][P.definedby[k]] := true;
  fi;
od;

for k in [ 1 .. Length(P.epimorphism) ] do
  if not IsInt( P.epimorphism[k] ) then

    # Convert the data structure
    P.epimorphism[k] := SumPcp( Q, P.epimorphism[k], P.identity );
  else
    i := 0;
    for j in [ 1 .. P.epimorphism[k] ] do
      if not j in del then
        i := i + 1;
      fi;
    od;
    P.epimorphism[k] := i;
  fi;
od;
return P;

end;

#####
##
## LiftHomomorphismPQp( <G>, <P>, <linsys> ) . lift homomorphism to p-cover
##
## 'LiftHomomorphismPQp' attempts to lift the homomorphism from the given
## finitely presented group <G> onto the p-cover of a previously calculated
## $p$-quotient. In doing so it obtains relations between the generators of
## the $p$-multiplier in order to satisfy the relations of the finitely
## presented group. It uses these relations to eliminate some (or possibly
## all) of the generators of the $p$-multiplier.
##
LiftHomomorphismPQp := function( G, P, linsys )

  local i, s, images, k, x;

  # determine the number of pseudo generators needed here
  k := 0;
  for i in [ 1 .. Length(P.epimorphism) ] do
    if not IsInt( P.epimorphism[i] ) then
      k := k+1;
    fi;
  od;

  k := Length(P.generators) - k + 1;

  # add pseudo generators to the images of the homomorphism
  for i in [ 1 .. Length(P.epimorphism) ] do
    if not IsInt( P.epimorphism[i] ) then
      P.epimorphism[i] :=
        SumPcp( P.pcp, P.epimorphism[i], P.generators[k] );
      P.definedby[k] := -i;
      k := k+1;
    fi;
  od;

  # temporarily write all images as group elements
  images := Copy( P.epimorphism );
  for i in [ 1 .. Length(images) ] do
    if IsInt(images[i]) then
      images[i] := P.generators[images[i]];
    fi;
  od;
end;

```

```

    fi;
  od;

  # compute the images of the relations of P under
  # the epimorphism
  for i in [ 1 .. Length(G.relators) ] do
    s := MappedWord( G.relators[i], G.generators, images );
    s := NormalWordPcp( P.pcp, s );
    s := PowerPcp( P.pcp, s, G.exponents[i] );
    EchelonizePQp( P, linsys, s );
  od;

  return [linsys,P];

end;

#####
##
##F CleanUpPQp( <P> ) . . . . . clean up <P>
##
CleanUpPQp := function( P )

  local i, defby;

  # Delete all the new and pseudo generators introduced
  for i in [ P.nrgens+1 .. Length(P.definedby) ] do
    defby := P.definedby[i];
    if IsList(defby) then
      RemoveSet( P.defining[1], TriangleIndex(defby[1], defby[2]) );
      P.isdefinition[1][ (defby[1]-2)*P.dimensions[1]+defby[2] ]
        := false;
    else
      if defby < 0 then defby := -defby; fi;
      RemoveSet( P.defining[2], defby );
      P.isdefinition[2][defby] := false;
    fi;
  od;
  P.definedby := Sublist( P.definedby, [ 1 .. P.nrgens ] );
  ShrinkPcp( P.pcp, [P.nrgens+1 .. Length(P.generators)] );
  P.generators := GeneratorsPcp(P.pcp);
  P.nrgens := Length(P.generators);
  P.nrnewgens := 0;

end;

#####
##
##F FirstClassPQp( <G>, <p> ) . . . . . computes the p-abelian quotient
##
## 'FirstClassPQp' returns a PQp record for the exponent- $p$ -class 1
## quotient of <G>.
##
FirstClassPQp := function( G, p )

  local erg, P;

  # add '<G>.exponents' and 'G.relators' if missing
  if not IsBound(G.relators) then
    G.relators := [];
  fi;
  if not IsBound(G.exponents) then
    G.exponents := List(G.relators, x -> 1);
  fi;

  P := InitPQp(Length(G.generators), p);
  erg := ConsistencyPQp(P);
  if erg = 0 then
    CleanUpPQp(P);
    return P;
  fi;
end;

```

```

fi;
erg := LiftHomomorphismPQp(G, P, erg);
P := ElimTailsPQp( erg[2], erg[1] );
return P;

end;

#####
##
## PQquotient( <G>, <p>, <cl> ) . . . . . computes a $p$-quotient of <G>
##
## 'PQquotient' computes the class <cl> $p$-quotient of the group <G> given
## by generators and relations. If <cl> is 0 it either computes the largest
## $p$-quotient, if there is a largest, or runs forever or until one of the
## following is satisfied:
##
##          a) GAP runs into a segmentation fault
##          b) GAP runs into a bus error
##          c) GAP runs out of memory
##          d) GAP is killed by an annoyed user
##          e) The machine crashes
##          f) The machine is rebooted
##          g) The sky falls down
##
PQquotient := function(G, p, cl)

  local i, forever, P, erg, t;

  # Check the arguments
  if not IsPrime(p) then
    Error("<p> must be a prime");
  fi;
  if not IsInt( cl ) or cl < 0 then
    Error("<cl> must be a non-negative integer");
  fi;

  # add '<G>.exponents' and '<G>.relators' if missing
  if not IsBound(G.relators) then
    G.relators := [];
  fi;
  if not IsBound(G.exponents) then
    G.exponents := List(G.relators, x -> 1);
  fi;

  # Who wants to run forever?
  forever := false;
  if cl = 0 then forever := true; fi;

  # initialise <P>
  P := FirstClassPQp( G, p );
  erg := rec( ls := [], del := [] );
  if Length(P.generators) = 0 then return P; fi;
  t := Runtime();
  InfoPQ1( "#I PQquotient: class 1 : ", P.dimensions[1], "\n" );

  # main loop
  i := 2;
  while i <= cl or forever do
    InfoPQ1( "#I PQquotient: Runtime : ", Runtime()-t, "\n" );
    P := TailsPQp(P);
    erg := ConsistencyPQp(P);
    if erg = 0 then
      CleanupPQp(P);
      InfoPQ1( "#I PQquotient: <G> has exponent-p-class :",
              Length(P.dimensions), "\n",
              "Runtime : ", Runtime()-t, "\n" );
      return P;
    fi;
    erg := LiftHomomorphismPQp(G, P, erg);

```

```

P := ElimTailsPQp( erg[2], erg[1] );

if P.dimensions[ Length( P.dimensions ) ] = 0 then
  P.dimensions := Sublist(P.dimensions, [1..Length(P.dimensions)-1]);
P.nrnewgens := 0;
InfoPQ1( "#I PQquotient: <G> has exponent-p-class :",
        Length(P.dimensions), "\n", "Runtime : ",
        Runtime()-t, "\n" );
return P;
fi;

InfoPQ1( "#I PQquotient: class ", i, " : ",
        P.dimensions[Length(P.dimensions)], "\n" );
i := i+1;
od;
InfoPQ1( "#I PQquotient: Runtime : ", Runtime()-t, "\n" );
return P;

end;

pQuotient := PQuotient;
PrimeQuotient := PQuotient;

#####
##
## NextClassPQp( <G>, <P> ) . . . . . compute the next class
##
## Let <P> be a PQp record for the class $c$-quotient of <G>. 'NextClassPQp'
## returns a PQp record for the class $c+1$ quotient of <G>, if it exists
## and <P> otherwise.
##
NextClassPQp := function( G, P )

  local P, erg;

  P := TailsPQp(P);
  erg := ConsistencyPQp(P);
  if erg = 0 then
    InfoPQ1("#I NextClassPQp: <G> has no larger p-quotient\n");
    CleanUpPQp(P);
    return P;
  fi;
  erg := LiftHomomorphismPQp(G, P, erg);
  P := ElimTailsPQp( erg[2], erg[1] );

  if P.dimensions[ Length( P.dimensions ) ] = 0 then
    P.dimensions := Sublist( P.dimensions, [1..Length(P.dimensions)-1]);
    P.nrnewgens := 0;
  fi;
  return P;

end;

#####
##
## Weight( <P>, <w> ) . . . . . compute the weight of <w> in <P>
##
## Let <P> be a PQp record and <w> a word in the generators of <P>. 'Weight'
## returns the weight of <w>.
##
Weight := function( P, w )
  return P.operations.Weight(P, w);
end;

PQpOps.Weight := function ( P, w )

  local i, d, wt;

```

```

d := DepthPcp(P.pcp, w);
i := 1;
wt := 1;
while i < d do
  i := i + P.dimensions[wt];
  wt := wt + 1;
od;
return wt-1;

end;

#####
##
## PQpOps.Factorization( <P>, <w> ) . . . express w in the generators of <P>
##
## Let <P> be a PQp record and <w> a word in the generators of <P>.
## 'PQpOps.Factorization' returns a word expressing <w> in the weight 1
## generators.
##
PQpOps.Factorization := function( P, w )

  local i, gens, wt;

  # check '<P>.abstractGenerators'
  if IsBound(P.abstractGenerators) then
    if Length(P.abstractGenerators) <> Length(P.generators) then
      Error("not enough abstract generators");
    fi;
  else
    P.abstractGenerators := P.generators;
  fi;

  # Build a list gens which contains the word defining generator i in the
  # i-th position. Note that we use the operators * and ^, since we do not
  # want to collect the word in the group <P>.
  gens := [];
  for i in [1 .. Length(P.generators) ] do
    if IsInt( P.definedby[i] ) then
      if P.definedby[i] < 0 then
        gens[i] := P.generators[i];
      else
        gens[i] := P.generators[P.definedby[i]]^P.prime;
      fi;
    else
      gens[i] := P.generators[P.definedby[i][1]]^-1
        * P.generators[P.definedby[i][2]]^-1
        * P.generators[P.definedby[i][1]]
        * P.generators[P.definedby[i][2]];
    fi;
  od;

  wt := TailDepthPcp( P.pcp,w );
  while wt > 1 do
    w := MappedWord( w, P.abstractGenerators, gens );
    wt := wt - 1;
  od;

  return w;

end;

```

Bibliography

William A. Alford and George Havas and M.F. Newman (1975), “Groups of exponent four”, *Notices Amer. Math. Soc.*, **22**, A.301.

Judith A. Ascione (1979), *On 3-groups of second maximal class*, PhD thesis. Australian National University.

Thomas Bischops (1989), *Collectoren im Programmsystem GAP*, Diplomarbeit an der RWTH Aachen: Lehrstuhl D für Mathematik.

John J. Cannon (1984), “An Introduction to the Group Theory Language, Cayley”, *Computational Group Theory*, (Durham, 1982), pp. 145–183. Academic Press, London, New York.

George Havas and M.F. Newman (1980), “Application of computers to questions like those of Burnside”, *Burnside Groups*, Lecture Notes in Math., **806**, (Bielefeld, 1977), pp. 211–230. Springer-Verlag, Berlin, Heidelberg, New York.

B. Huppert (1967), *Endliche Gruppen I*. Grundlehren Math. Wiss., **134**, Springer-Verlag, Berlin, Heidelberg, New York.

C.R. Leedham-Green and L.H. Soicher (1990), “Collection from the left and other strategies”, *J. Symbolic Comput.*, **9**, 665–675.

E.A. O’Brien (1990), “The p -group generation algorithm”, *J. Symbolic Comput.*, **9**, 677–698.

E.A. O’Brien (1991), “The groups of order 256”, *J. Algebra*, **143**(1), 219–235.

- Martin Schönert (1987), *Konzeption und Implementation des Programmiersystems GAP für die Algorithmische Gruppentheorie*. Diplomarbeit an der RWTH Aachen: Lehrstuhl D für Mathematik.
- Martin Schönert *et al.* (1993), *GAP – Groups, Algorithms and Programming*. RWTH Aachen: Lehrstuhl D für Mathematik.
- Charles C. Sims (to appear), *Computing with finitely presented groups*. Cambridge University Press.
- M.R. Vaughan-Lee (1984), “An Aspect of the Nilpotent Quotient Algorithm”, *Computational Group Theory*, (Durham, 1982), pp. 76–83. Academic Press, London, New York.
- M.R. Vaughan-Lee (1990), “Collection from the left”, *J. Symbolic Comput.*, **9**, 725–733.
- J.W. Wamsley (1974), “Computation in nilpotent groups (theory)”, *Proc. Second Internat. Conf. Theory of Groups*, Lecture Notes in Math., **372**, (Canberra, 1973), pp. 691–700. Springer-Verlag.