ANUPQ

ANU p-Quotient

3.3.0

5 January 2023

Greg Gamble

Werner Nickel

Eamonn O'Brien

Greg Gamble
Email: Greg.Gamble@uwa.edu.au
Homepage: http://school.maths.uwa.edu.au/~gregg
Address: Department of Mathematics and Statistics
Curtin University
GPO Box U 1987
Perth WA 6845
Australia

Werner Nickel

Homepage: https://www2.mathematik.tu-darmstadt.de/~nickel/

Eamonn O'Brien
Email: obrien@math.auckland.ac.nz
Homepage: https://www.math.auckland.ac.nz/~obrien
Address: Department of Mathematics
University of Auckland
Private Bag 92019
Auckland
New Zealand

Copyright

Contents

Chapter 1

Introduction

1.1 Overview

The GAP 4 package ANUPQ provides an interface to the ANU pq C program written by Eamonn O'Brien, making the functionality of the C program available to GAP. Henceforth, we shall refer to the ANUPQ package when referring to the GAP interface, and to the ANU pq program or just pq when referring to that C program.

The pq program consists of implementations of the following algorithms:

1. A *p-quotient algorithm* to compute pc-presentations for p-factor groups of finitely presented groups.

2. A *p-group generation algorithm* to generate pc presentations of groups of prime power order.

3. A *standard presentation algorithm* used to compute a canonical pc-presentation of a p-group.

4. An algorithm which can be used to compute the *automorphism group* of a p-group.

   This part of the pq program is not accessible through the ANUPQ package. Instead, users are advised to consider the GAP 4 package AutPGrp by Bettina Eick and Eamonn O'Brien, which implements a better algorithm in GAP for the computation of automorphism groups of p-groups.

The current version of the ANUPQ package requires GAP 4.5, and version 1.5 of the AutPGrp package. All code that made the package compatible with earlier versions of GAP has been removed. If you must use an older GAP version and cannot upgrade, then you may try using an older ANUPQ version. However, you should not use versions of the ANUPQ package older than 2.2, since they are known to have bugs.

1.2 How to read this manual

It is not expected that readers of this manual will read it in a linear fashion from cover to cover; some sections contain material that is far too technical to be absorbed on a first reading.

Firstly, installers of the ANUPQ package will need to read Chapter 'Installing the ANUPQ Package', if they have not already gleaned these details from the `README` file.

Once the ANUPQ package is installed, users of the ANUPQ package will benefit most by first reading Chapter 'Mathematical Background and Terminology', which gives a brief description of the background and terminology used (this chapter also cites a number of references for further reading), and the introduction of Chapter 'Infrastructure' (skip the remainder of the chapter on a first reading).

Then the user/reader should pursue Chapter 'Non-interactive ANUPQ functions' in detail, delving into Chapter 'ANUPQ Options' as necessary for the options of the functions that are described. The user will become best acquainted with the ANUPQ package by trying the examples. This chapter describes the non-interactive functions of the ANUPQ package, i.e. "one-shot" functions that invoke the pq program in such a way that once GAP has got what it needs, the pq is allowed to exit. It is expected that most of the time, users will only need these functions.

Advanced users will want to explore Chapter 'Interactive ANUPQ functions' which describes all the interactive functions of the ANUPQ package; these are functions that extract information via a dialogue with a running pq process. Occasionally, a user needs the "next step"; the functions provided in this chapter make use of data from previous steps retained by the pq program, thus allowing the user to interact with the pq program like one can when one uses the pq program as a stand-alone (see `guide.dvi` in the `standalone-doc` directory).

After having read Chapters 'Non-interactive ANUPQ functions' and 'Interactive ANUPQ functions', cross-references will have taken the reader into Chapter 'ANUPQ Options'; by this stage, the reader need only read the introduction of Chapter 'ANUPQ Options'.

After the reader has developed some facility with the ANUPQ package, she should explore the examples described in Appendix 'Examples'.

If you run into trouble using the ANUPQ functions, some troubleshooting hints are given in Section 'Hints and Warnings regarding the use of Options'. If the troubleshooting hints don't help, Section 'Authors and Acknowledgements' below, gives contact details for the authors of the components of the ANUPQ package.

1.3 Authors and Acknowledgements

The C implementation of the ANU pq standalone was developed by Eamonn O'Brien.

An interactive interface using iostreams was developed with the assistance of Werner Nickel by Greg Gamble.

The GAP 4 version of this package was adapted from the GAP 3 version by Werner Nickel.

A new co-maintainer, Max Horn, joined the team in November, 2011.

The authors would like to thank Joachim Neubüser for his careful proof-reading and advice, and for formulating Chapter 'Mathematical Background and Terminology'.

We would also like to thank Bettina Eick who by her testing and provision of examples helped us to eliminate a number of bugs and who provided a number of valuable suggestions for extensions of the package beyond the GAP 3 capabilities.

If you find a bug, the last section of ANUPQ's `README` describes the information we need and where to send us a bug report; please take the time to read this (i.e. help us to help you).

Chapter 2

Mathematical Background and Terminology

In this chapter we will give a brief description of the mathematical notions used in the algorithms implemented in the ANU pq program that are made accessible from GAP through this package. For proofs and details we will point to relevant places in the published literature. Also we will try to give some explanation of terminology that may help to use the "low-level" interactive functions described in Section 'Low-level Interactive ANUPQ functions based on menu items of the pq program'. However, users who intend to use these functions are strongly advised to acquire a thorough understanding of the algorithms from the quoted literature. There is little or no checking done in these functions and naive use may result in incorrect results.

2.1 Basic notions

2.1.1 pc Presentations and Consistency

For details, see e.g. [NNN98].

Every finite p-group G has a presentation of the form:

$$\{a_1,\ldots,a_n \mid a_i^p = v_{ii}, 1 \le i \le n, [a_k,a_j] = v_{jk}, 1 \le j < k \le n\}.$$

where v_{jk} is a word in the elements $a_{k+1},\ldots,a_n$ for $1 \le j \le k \le n$.

This is called a *power-commutator* presentation (or *pc presentation* or *pcp*) of G, generators from such a presentation will be referred to as *pc generators*. In terms of such pc generators every element of G can be written in a "normal form" $a_1^{e_1}\ldots a_n^{e_n}$ with $0 \le e_i < p$. Moreover any given product of the generators can be brought into such a normal form using the defining relations in the above presentation as rewrite rules. Any such process is called *collection*. For the discussion of various collection methods see [LGS90] and [VL90a].

Every p-group of order p^n has such a pcp on n generators and conversely every such presentation defines a p-group. However a p-group defined by a pcp on n generators can be of smaller order p^m with $m < n$. A pcp on n generators that does in fact define a p-group of order p^n is called *consistent* in this manual, in line with most of the literature on the algorithms occurring here. A consistent pcp determines a *confluent rewriting system* (see `IsConfluent` (**Reference: IsConfluent**) of the GAP Reference Manual) for the group it defines and for this reason often (in particular in the GAP Reference Manual) such a pcp presentation is also called *confluent*.

Consistency of a pcp is tantamount to the fact that for any given word in the generators any two collections will yield the same normal form.

Consistency of a pcp can be checked by a finite set of *consistency conditions*, demanding that collection of the left hand side and of the right hand side of certain equations, starting with subproducts indicated by bracketing, will result in the same normal form. There are 3 types of such equations (that will be referred to in the manual):

$$\begin{array}{lcll} (a^n)a & = & a(a^n) & \text{(Type1)} \\ (b^n)a & = & b^{(n-1)}(ba), b(a^n) = (ba)a^{(n-1)} & \text{(Type2)} \\ c(ba) & = & (cb)a & \text{(Type3)} \end{array}$$

See [VL84] for a description of a sufficient set of consistency conditions in the context of the p-quotient algorithm.

2.1.2 Exponent-p Central Series and Weighted pc Presentations

For details, see [NNN98].

The (*descending* or *lower*) (*exponent-*)p*-central series* of an arbitrary group G is defined by

$$P_0(G) := G, P_i(G) := [G, P_{i-1}(G)]P_{i-1}(G)^p.$$

For a p-group G this series terminates with the trivial group. G has p*-class* c if c is the smallest integer such that $P_c(G)$ is the trivial group. In this manual, as well as in much of the literature about the pq- and related algorithms, the p-class is often referred to simply by *class*.

Let the p-group G have a consistent pcp as above. Then the subgroups

$$\langle 1 \rangle < \langle a_n \rangle < \langle a_n, a_{n-1} \rangle < \ldots < \langle a_n, \ldots, a_i \rangle < \ldots < G$$

form a central series of G. If this refines the p-central series, we can define the *weight function* w for the pc generators by $w(a_i) = k$, if a_i is contained in $P_{k-1}(G)$ but not in $P_k(G)$.

The pair of such a weight function and a pcp allowing it, is called a *weighted pcp*.

2.1.3 p-Cover, p-Multiplicator

For details, see [NNN98].

Let d be the minimal number of generators of the p-group G of p-class c. Then G is isomorphic to a factor group F/R of a free group F of rank d. We denote $[F,R]R^p$ by R^*. It can be proved (see e.g. [O'B90]) that the isomorphism type of $G^* := F/R^*$ depends only on G. G^* is called the p*-covering group* or p*-cover* of G, and R/R^* the p*-multiplicator* of G. The p-multiplicator is, of course, an elementary abelian p-group; its minimal number of generators is called the *(*p*-)multiplicator rank*.

2.1.4 Descendants, Capable, Terminal, Nucleus

For details, see [New77] and [O'B90].

Let again G be a p-group of p-class c and d the minimal number of generators of G. A p-group H is a *descendant* of G if the minimal number of generators of H is d and $H/P_c(H)$ is isomorphic to G. A descendant H of G is an *immediate descendant* if it has p-class $c+1$. G is called *capable* if it has immediate descendants; otherwise it is *terminal*.

Let $G^* = F/R^*$ again be the p-cover of G. Then the group $P_c(G^*)$ is called the *nucleus* of G. Note that $P_c(G^*)$ is contained in the p-multiplicator R/R^*.

It is proved (e.g. in [O'B90]) that the immediate descendants of G are obtained as factor groups of the p-cover by (proper) supplements of the nucleus in the (elementary abelian) p-multiplicator. These are also called *allowable*.

It is further proved there that every automorphism α of F/R extends to an automorphism α^* of the p-cover F/R^* and that the restriction of α^* to the multiplicator R/R^* is uniquely determined by α. Each *extended automorphism* α^* induces a permutation of the allowable subgroups. Thus the extended automorphisms determine a group P of *permutations* on the set A of allowable subgroups (The group P of permutations will appear in the description of some interactive functions). Choosing a representative S from each orbit of P on A, the set of factor groups F/S contains each (isomorphism type of) immediate descendant of G exactly once. For each immediate descendant, the procedure of computing the p-cover, extending the automorphisms and computing the orbits on allowable subgroups can be repeated. Iteration of this procedure can in principle be used to determine all descendants of a p-group.

2.1.5 Laws

Let $l(x_1,\ldots,x_n)$ be a word in the free generators $x_1,\ldots,x_n$ of a free group of rank n. Then $l(x_1,\ldots,x_n) = 1$ is called a *law* or *identical relation* in a group G if $l(g_1,\ldots,g_n) = 1$ for any choice of elements $g_1,\ldots,g_n$ in G. In particular, $x^e = 1$ is called an *exponent law*, $[[x,y],[u,v]] = 1$ the *metabelian law*, and $[\ldots[[x_1,x_2],x_2],\ldots,x_2] = 1$ an *Engel identity*.

2.2 The p-quotient Algorithm

For details, see [HN80], [NO96] and [VL84]. Other descriptions of the algorithm are given in [Sim94].

The pq algorithm successively determines the factor groups of the groups of the p-central series of a finitely presented (fp) group G. If a bound b for the p-class is given, the algorithm will determine those factor groups up to at most p-class b. If the p-central series terminates with a subgroup $P_k(G)$ with $k < b$, the algorithm will stop with that group. If no such bound is given, it will try to find the biggest such factor group.

$G/P_1(G)$ is the largest elementary abelian p-factor group of G and this can be found from the relation matrix of G using matrix diagonalisation modulo p. So it suffices to explain how $G/P_{i+1}(G)$ is found from G and $G/P_i(G)$ for some $i \geq 1$.

This is done, in principle, in two steps: first the p-cover of $G_i := G/P_i(G)$ is determined (which depends only on G_i, not on G) and then $G/P_{i+1}(G)$ as a factor group of this p-cover.

2.2.1 Finding the p-cover

A very detailed description of the first step is given in [NNN98], from which we just extract some passages in order to point to some terms occurring in this manual.

Let H be a p-group and $p^{d(b)}$ be the order of $H/P_b(H)$. So $d := d(1)$ is the minimal number of generators of H. A weighted pcp of H will be called *labelled* if for each generator a_k, $k > d$ one relation, having this generator as its right hand side, is marked as *definition* of this generator.

As described in [NNN98], a weighted labelled pcp of a p-group can be obtained stepping down its p-central series.

So let us assume that a weighted labelled pcp of G_i is given. A straightforward way of of writing down a (not necessarily consistent) pcp for its p-cover is to add generators, one for each relation which is not a definition, and modify the right hand side of each such relation by multiplying it on the right by one of the new generators -- a different generator for each such relation. Further relations are then added to make the new generators central and of order p. This procedure is called *adding tails*. A more formal description of it is again given in [NNN98].

It is important to realise that the "new" generators will generate an elementary abelian group, that is, in additive notation, a vector space over the field of p elements. As said, the pcp of the p-cover obtained in this way need not be consistent. Since the pcp of G_i was consistent, applying the consistency conditions to the pcp of the p-cover, in case the presentation obtained for p-cover is not consistent, will produce a set of equations between the new generators, that, written additively, are linear equations over the field of p elements and can hence be used to remove redundant generators until a consistent pcp is obtained.

In reality, to follow this straightforward procedure would be forbiddingly inefficient except for very small examples. There are many ways of a priori reducing the number of "new generators" to be introduced, using e.g. the weights attached to the generators, and the main part of [NNN98] is devoted to a detailed discussion with proofs of these possibilities.

2.2.2 Imposing the Relations of the fp Group

In order to obtain $G/P_{i+1}(G)$ from the pcp of the p-cover of $G_i = G/P_i(G)$, the defining relations from the original presentation of G must be imposed. Since G_i is a homomorphic image of G, these relations again yield relations between the "new generators" in the presentation of the p-cover of G_i.

2.2.3 Imposing Laws

While we have so far only considered the computation of the factor groups of a given fp group by the groups of its descending p-central series, the p-quotient algorithm allows a very important variant of this idea: laws can be prescribed that should be fulfilled by the p-factor groups computed by the algorithm. The key observation here is the fact that at each step down the descending p-central series it suffices to impose these laws only for a finite number of words. Again for efficiency of the method it is crucial to keep the number of such words small, and much of [NO96] and the literature quoted in this paper is devoted to this problem.

In this form, starting with a free group and imposing an exponent law (also referred to as an *exponent check*) the `pq` program has, in fact, found its most noted application in the determination of (restricted) Burnside groups (as reported in e.g. [HN80], [NO96] and [VL90b]).

Via a GAP program using the "local" interactive functions of the `pq` program made available through this interface also arbitrary laws can be imposed via the option `Identities` (see 6.2).

2.3 The p-group generation Algorithm, Standard Presentation, Isomorphism Testing

For details, see [New77] and [O'B90].

The p-group generation algorithm determines the immediate descendants of a given p-group G up to isomorphism. From what has been explained in Section 'Basic notions', it is clear that this amounts to the construction of the p-cover, the extension of the automorphisms of G to the p-cover

and the determination of representatives of the orbits of the action of these automorphisms on the set of supplements of the nucleus in the p-multiplicator.

The main practical problem here is the determination of these representatives. [O'B90] describes methods for this and the pq program allows choices according to whether space or time limitations must be met.

As well as the descendants of G, the pq program determines their automorphism groups from that of G (see [O'B95]), which is important for an iteration of the process; this has been used by Eamonn O'Brien, e.g. in the classification of the 2-groups that are now also part of the *Small Groups* library available through GAP.

A variant of the p-group generation algorithm is also used to define a *standard presentation* of a given p-group. This is done by constructing an isomorphic copy of the given group through a chain of descendants and at each step making a choice of a particular representative for the respective orbit of capable groups. In a fairly delicate process, subgroups of the p-multiplicator are represented by *echelonised matrices* and a first among the *labels for standard matrices* is chosen (this is described in detail in [O'B94]).

Finally, the standard presentation provides a way of testing if two given p-groups are isomorphic: the standard presentations of the groups are computed, for practical purposes *compacted* and the results compared for being identical, i.e. the groups are isomorphic if and only if their standard presentations are identical.

Chapter 3

Infrastructure

Most of the details in this chapter are of a technical nature; the user need only skim over this chapter on a first reading. Mostly, it is enough to know that

- you must do a `LoadPackage("anupq");` before you can expect to use a command defined by the ANUPQ package (details are in Section 'Loading the ANUPQ Package');
- partial results of ANUPQ commands and some other data are stored in the `ANUPQData` global variable (details are in Section 'The ANUPQData Record');
- doing `SetInfoLevel(InfoANUPQ, n);` for n greater than the default value 1 will give progressively more information of what is going on "behind the scenes" (details are in Section 'Setting the Verbosity of ANUPQ via Info and InfoANUPQ');
- in Section 'Utility Functions' we describe some utility functions and functions that run examples from the collection of examples of this package;
- in Section 'Attributes and a Property for fp and pc p-groups' we describe the attributes and property `NuclearRank`, `MultiplicatorRank` and `IsCapable`; and
- in Section 'Hints and Warnings regarding the use of Options' we describe some troubleshooting strategies. Also this section explains the utility of setting `ANUPQWarnOfOtherOptions := true;` (particularly for novice users) for detecting misspelt options and diagnosing other option usage problems.

3.1 Loading the ANUPQ Package

To use the ANUPQ package, as with any GAP package, it must be requested explicitly. This is done by calling

Example

```
gap> LoadPackage( "anupq" );
-------------------------------------------------------------------------
Loading  ANUPQ 3.3.0 (ANU p-Quotient)
by Greg Gamble (GAP code, http://school.maths.uwa.edu.au/~gregg),
   Werner Nickel (GAP code, https://www2.mathematik.tu-darmstadt.de/~nickel/), and
   Eamonn O'Brien (C code, https://www.math.auckland.ac.nz/~obrien).
maintained by:
```

```
    Greg Gamble (http://school.maths.uwa.edu.au/~gregg) and
    Max Horn (https://www.quendi.de/math).
  uses ANU pq binary (C code program) version: 1.9
  Homepage: https://gap-packages.github.io/anupq/
  Report issues at https://github.com/gap-packages/anupq/issues
  ------------------------------------------------------------------------
  true
```

Note that since the ANUPQ package uses the `AutomorphimGroupPGroup` function of the AutPGrp package and, in any case, often needs other AutPGrp functions when computing descendants, the user must ensure that the AutPGrp package is also installed, at least version 1.5. If the AutPGrp package is not installed, the ANUPQ package will `fail` to load.

Also, if GAP cannot find a working `pq` binary, the call to `LoadPackage` will return `fail`.

If you want to load the ANUPQ package by default, you can put the `LoadPackage` command into your `gap.ini` file (see Section **Reference: The gap.ini and gaprc files** in the GAP Reference Manual). By the way, the novice user of the ANUPQ package should probably also append the line

Example

```
ANUPQWarnOfOtherOptions := true;
```

to their `gap.ini` file, somewhere after the `LoadPackage( "anupq" );` command (see `ANUPQWarnOfOtherOptions` (3.6.1)).

3.2 The ANUPQData Record

This section contains fairly technical details which may be skipped on an initial reading.

3.2.1 ANUPQData

▷ `ANUPQData` (global variable)

is a GAP record in which the essential data for an ANUPQ session within GAP is stored; its fields are:

`binary`
: the path of the `pq` binary;

`tmpdir`
: the path of the temporary directory used by the `pq` binary and GAP (i.e. the directory in which all the `pq`'s temporary files are created) (also see `ANUPQDirectoryTemporary` (3.2.2) below);

`outfile`
: the full path of the default `pq` output file;

`SPimages`
: the full path of the file `GAP_library` to which the `pq` program writes its Standard Presentation images;

`version`
: the version of the current `pq` binary;

`ni` a data record used by non-interactive functions (see below and Chapter 'Non-interactive ANUPQ functions');

`io` list of data records for `PqStart` (see below and `PqStart` (5.1.1)) processes;

`topqlogfile`
name of file logged to by `ToPQLog` (see `ToPQLog` (3.4.7)); and

`logstream`
stream of file logged to by `ToPQLog` (see `ToPQLog` (3.4.7)).

Each time an interactive ANUPQ process is initiated via `PqStart` (see `PqStart` (5.1.1)), an identifying number *`ioIndex`* is generated for the interactive process and a record `ANUPQData.io[`*`ioIndex`*`]` with some or all of the fields listed below is created. Whenever a non-interactive function is called (see Chapter 'Non-interactive ANUPQ functions'), the record `ANUPQData.ni` is updated with fields that, if bound, have exactly the same purpose as for a `ANUPQData.io[`*`ioIndex`*`]` record.

`stream`
the IOStream opened for interactive ANUPQ process *`ioIndex`* or non-interactive ANUPQ function;

`group`
the group given as first argument to `PqStart`, `Pq`, `PqEpimorphism`, `PqDescendants` or `PqStandardPresentation` (or any synonymous methods);

`haspcp`
is bound and set to `true` when a pc presentation is first set inside the `pq` program (e.g. by `PqPcPresentation` or `PqRestorePcPresentation` or a higher order function like `Pq`, `PqEpimorphism`, `PqPCover`, `PqDescendants` or `PqStandardPresentation` that does a `PqPcPresentation` operation, but *not* `PqStart` which only starts up an interactive ANUPQ process);

`gens`
a list of the generators of the group `group` as strings (the same as those passed to the `pq` program);

`rels`
a list of the relators of the group `group` as strings (the same as those passed to the `pq` program);

`name`
the name of the group whose pc presentation is defined by a call to the `pq` program (according to the `pq` program `--` unless you have used the `GroupName` option (see e.g. `Pq` (4.1.1)) or applied the function `SetName` (see `SetName` (**Reference: Name**)) to the group, the "generic" name `"[grp]"` is set as a default);

`gpnum`
if not a null string, the "number" (i.e. the unique label assigned by the `pq` program) of the last descendant processed;

`class`
the largest lower exponent-p central class of a quotient group of the group (usually `group`) found by a call to the `pq` program;

`forder`
: the factored order of the quotient group of largest lower exponent-p central class found for the group (usually `group`) by a call to the `pq` program (this factored order is given as a list `[p,n]`, indicating an order of p^n);

`pcoverclass`
: the lower exponent-p central class of the p-covering group of a p-quotient of the group (usually `group`) found by a call to the `pq` program;

`workspace`
: the workspace set for the `pq` process (either given as a second argument to `PqStart`, or set by default to 10000000);

`menu`
: the current menu of the `pq` process (the `pq` program is managed by various menus, the details of which the user shouldn't normally need to know about -- the `menu` field remembers which menu the `pq` process is currently "in");

`outfname`
: is the file to which `pq` output is directed, which is always `ANUPQData.outfile`, except when option `SetupFile` is used with a non-interactive function, in which case `outfname` is set to `"PQ_OUTPUT"`;

`pQuotient`
: is set to the value returned by `Pq` (see `Pq` (4.1.1)) (the field `pQepi` is also set at the same time);

`pQepi`
: is set to the value returned by `PqEpimorphism` (see `PqEpimorphism` (4.1.2)) (the field `pQuotient` is also set at the same time);

`pCover`
: is set to the value returned by `PqPCover` (see `PqPCover` (4.1.3));

`SP`
: is set to the value returned by `PqStandardPresentation` or `StandardPresentation` (see `PqStandardPresentation` (5.3.4)) when called interactively, for process *i* (the field `SPepi` is also set at the same time);

`SPepi`
: is set to the value returned by `EpimorphismPqStandardPresentation` or `EpimorphismStandardPresentation` (see `EpimorphismPqStandardPresentation` (5.3.5)) when called interactively, for process *i* (the field `SP` is also set at the same time);

`descendants`
: is set to the value returned by `PqDescendants` (see `PqDescendants` (4.4.1));

`treepos`
: if set by a call to `PqDescendantsTreeCoclassOne` (see `PqDescendantsTreeCoclassOne` (A.4.1)), it contains a record with fields `class`, `node` and `ndes` being the information that determines the last descendant with a non-zero number of descendants processed;

`xgapsheet`
: if set by a call to `PqDescendantsTreeCoclassOne` (see `PqDescendantsTreeCoclassOne` (A.4.1)) during an XGAP session, it contains the XGAP `Sheet` on which the descendants tree is displayed; and

`nextX`
: if set by a call to `PqDescendantsTreeCoclassOne` (see `PqDescendantsTreeCoclassOne` (A.4.1)) during an XGAP session, it contains a list of integers, the *i*th entry of which is the *x*-coordinate of the next node (representing a descendant) for the *i*th class.

3.2.2 ANUPQDirectoryTemporary

▷ `ANUPQDirectoryTemporary(`*dir*`)` (function)

calls the UNIX command `mkdir` to create *dir*, which must be a string, and if successful a directory object for *dir* is both assigned to `ANUPQData.tmpdir` and returned. The field `ANUPQData.outfile` is also set to be a file in `ANUPQData.tmpdir`, and on exit from GAP *dir* is removed. Most users will never need this command; by default, GAP typically chooses a "random" subdirectory of `/tmp` for `ANUPQData.tmpdir` which may occasionally have limits on what may be written there. `ANUPQDirectoryTemporary` permits the user to choose a directory (object) where one is not so limited.

3.3 Setting the Verbosity of ANUPQ via Info and InfoANUPQ

3.3.1 InfoANUPQ

▷ `InfoANUPQ` (info class)

The input to and the output from the `pq` program is, by default, not displayed. However the user may choose to see some, or all, of this input/output. This is done via the `Info` mechanism (see Section **Reference: Info Functions** in the GAP Reference Manual). For this purpose, there is the *InfoClass* `InfoANUPQ`. If the `InfoLevel` of `InfoANUPQ` is high enough each line of `pq` input/output is directed to a call to `Info` and will be displayed for the user to see. By default, the `InfoLevel` of `InfoANUPQ` is 1, and it is recommended that you leave it at this level, or higher. Messages that the user should presumably want to see and output from the `pq` program influenced by the value of the option `OutputLevel` (see the options listed in Section `Pq` (4.1.1)), other than timing and memory usage are directed to `Info` at `InfoANUPQ` level 1.

To turn off *all* `InfoANUPQ` messaging, set the `InfoANUPQ` level to 0.

There are five other user-intended `InfoANUPQ` levels: 2, 3, 4, 5 and 6.

Example

```
gap> SetInfoLevel(InfoANUPQ, 2);
```

enables the display of most timing and memory usage data from the `pq` program, and also the number of identity instances when the `Identities` option is used. (Some timing and memory usage data, particularly when profuse in quantity, is `Info`-ed at `InfoANUPQ` level 3 instead.) Note that the the GAP functions `time` and `Runtime` (see `Runtime` (**Reference: Runtime**) in the GAP Reference Manual) count the time spent by GAP and *not* the time spent by the (external) `pq` program.

Example
```
gap> SetInfoLevel(InfoANUPQ, 3);
```

enables the display of output of the nature of the first two `InfoANUPQ` that was not directly invoked by the user (e.g. some commands require GAP to discover something about the current state known to the pq program). The identity instances processed under the `Identities` option are also displayed at this level. In some cases, the pq program produces a lot of output despite the fact that the `OutputLevel` (see 6.2) is unset or is set to 0; such output is also `Info`-ed at `InfoANUPQ` level 3.

Example
```
gap> SetInfoLevel(InfoANUPQ, 4);
```

enables the display of all the commands directed to the pq program, behind a "`ToPQ>` " prompt (so that you can distinguish it from the output from the pq program). See Section 'Hints and Warnings regarding the use of Options' for an example of how this can be a useful troubleshooting tool.

Example
```
gap> SetInfoLevel(InfoANUPQ, 5);
```

enables the display of the pq program's prompts for input. Finally,

Example
```
gap> SetInfoLevel(InfoANUPQ, 6);
```

enables the display of all other output from the pq program, namely the banner and menus. However, the timing data printed when the pq program exits can never be observed.

3.4 Utility Functions

3.4.1 PqLeftNormComm

▷ `PqLeftNormComm(`*`elts`*`)` (function)

returns for a list of elements of some group (e.g. *`elts`* may be a list of words in the generators of a free or fp group) the left normed commutator of *`elts`*, e.g. if *`w1`*, *`w2`*, *`w3`* are such elements then `PqLeftNormComm( [`*`w1`*`, `*`w2`*`, `*`w3`*`] );` is equivalent to `Comm( Comm(` *`w1`*`, `*`w2`*` ), `*`w3`*` );`.

Note: *`elts`* must contain at least two elements.

3.4.2 PqGAPRelators

▷ `PqGAPRelators(`*`group, rels`*`)` (function)

returns a list of words that GAP understands, given a list *`rels`* of strings in the string representations of the generators of the fp group *`group`* prepared as a list of relators for the pq program.

Note: The pq program does not use `/` to indicate multiplication by an inverse and uses square brackets to represent (left normed) commutators. Also, even though the pq program accepts relations, all elements of *`rels`* *must* be in relator form, i.e. a relation of form *`w1`* `=` *`w2`* must be written as *`w1`*`*(`*`w2`*`)^-1`.

Here is an example:

Example

```
gap> F := FreeGroup("a", "b");
<free group on the generators [ a, b ]>
gap> PqGAPRelators(F, [ "a*b^2", "[a,b]^2*a", "([a,b,a,b,b]*a*b)^2*a" ]);
[ a*b^2, a^-1*b^-1*a*b*a^-1*b^-1*a*b*a, b^-1*a^-1*b^-1*a^-1*b*a*b^-1*a*b*a^
    -1*b*a^-1*b^-1*a*b*a*b^-1*a^-1*b^-1*a^-1*b*a*b^-1*a*b^-1*a^-1*b*a^-1*b^
    -1*a*b*a*b*a^-1*b*a*b^-1*a*b*a^-1*b*a^-1*b^-1*a*b*a*b^-1*a^-1*b^-1*a^
    -1*b*a*b^-1*a*b^-1*a^-1*b*a^-1*b^-1*a*b*a*b^2*a*b*a ]
```

3.4.3 PqParseWord

▷ `PqParseWord(`*`word, n`*`)` (function)

parses a *word*, a string representing a word in the pc generators `x1,...,xn`, through GAP. This function is provided as a rough-and-ready check of *word* for syntax errors. A syntax error will cause the entering of a `break`-loop, in which the error message may or may not be meaningful (depending on whether the syntax error gets caught at the GAP or kernel level).

Note: The reason the generators *must* be `x1,...,xn` is that these are the pc generator names used by the pq program (as distinct from the generator names for the group provided by the user to a function like `Pq` that invokes the pq program).

3.4.4 PqExample (no arguments)

▷ `PqExample()` (function)
▷ `PqExample(`*`example[, PqStart][, Display]`*`)` (function)
▷ `PqExample(`*`example[, PqStart][, filename]`*`)` (function)

With no arguments, or with single argument `"index"`, or a string *example* that is not the name of a file in the `examples` directory, an index of available examples is displayed.

With just the one argument *example* that is the name of a file in the `examples` directory, the example contained in that file is executed in its simplest form. Some examples accept options which you may use to modify some of the options used in the commands of the example. To find out which options an example accepts, use one of the mechanisms for displaying the example described below.

Some examples have both non-interactive and interactive forms; those that are non-interactive only have a name ending in `-ni`; those that are interactive only have a name ending in `-i`; examples with names ending in `.g` also have only one form; all other examples have both non-interactive and interactive forms and for these giving `PqStart` as second argument invokes `PqStart` initially and makes the appropriate adjustments so that the example is executed or displayed using interactive functions.

If `PqExample` is called with last (second or third) argument `Display` then the example is displayed without being executed. If the last argument is a non-empty string *filename* then the example is also displayed without being executed but is also written to a file with that name. Passing an empty string as last argument has the same effect as passing `Display`.

Note: The variables used in `PqExample` are local to the running of `PqExample`, so there's no danger of having some of your variables over-written. However, they are not completely lost either. They are saved to a record `ANUPQData.examples.vars`, i.e. if `F` is a variable used in the example then you will be able to access it after `PqExample` has finished as `ANUPQData.examples.vars.F`.

3.4.5 AllPqExamples

▷ `AllPqExamples()` (function)

returns a list of all currently available examples in default UNIX-listing (i.e. alphabetic) order.

3.4.6 GrepPqExamples

▷ `GrepPqExamples(`*`string`*`)` (function)

runs the UNIX command `grep` *`string`* over the ANUPQ examples and returns the list of examples for which there is a match. The actual matches are `Info`-ed at `InfoANUPQ` level 2.

3.4.7 ToPQLog

▷ `ToPQLog(`*`[filename]`*`)` (function)

With string argument *`filename`*, `ToPQLog` opens the file with name *`filename`* for logging; all commands written to the pq binary (that are `Info`-ed behind a "`ToPQ>` " prompt at `InfoANUPQ` level 4) are then also written to that file (but without prompts). With no argument, `ToPQLog` stops logging to whatever file was being logged to. If a file was already being logged to, that file is closed and the file with name *`filename`* is opened for logging.

3.5 Attributes and a Property for fp and pc p-groups

3.5.1 NuclearRank

▷ `NuclearRank(`*`G`*`)` (attribute)
▷ `MultiplicatorRank(`*`G`*`)` (attribute)
▷ `IsCapable(`*`G`*`)` (property)

return the nuclear rank of *`G`*, p-multiplicator rank of *`G`*, and whether *`G`* is capable (i.e. `true` if it is, or `false` if it is not), respectively.

These attributes and property are set automatically if *`G`* is one of the following:

- an fp group returned by `PqStandardPresentation` or `StandardPresentation` (see `PqStandardPresentation` (4.2.1));
- the image (fp group) of the epimorphism returned by an `EpimorphismPqStandardPresentation` or `EpimorphismStandardPresentation` call (see `EpimorphismPqStandardPresentation` (4.2.2)); or
- one of the pc groups of the list of descendants returned by `PqDescendants` (see `PqDescendants` (4.4.1)).

If *`G`* is an fp group or a pc p-group and not one of the above and the attribute or property has not otherwise been set for *`G`*, then `PqStandardPresentation` is called to set all three of `NuclearRank`, `MultiplicatorRank` and `IsCapable`, before returning the value of the attribute or property actually called. Such a group *`G`* must know in advance that it is a p-group; this is the case for the groups

returned by the functions Pq and PqPCover, and the image group of the epimorphism returned by PqEpimorphism. Otherwise, if you know the group to be a p-group, then this can be set by typing

```
SetIsPGroup( G, true );
```

or by invoking IsPGroup(G). Note that for an fp group G, the latter may result in a coset enumeration which might not terminate in a reasonable time.

Note: For G such that HasNuclearRank(G) = true, IsCapable(G) is equivalent to (the truth or falsity of) NuclearRank(G) = 0.

3.6 Hints and Warnings regarding the use of Options

On a first reading we recommend you skip this section and come back to it if and when you run into trouble.

Note: By "options" we refer to GAP options. The pq program also uses the term "option"; to distinguish the two usages of "option", in this manual we use the term *menu item* to refer to what the pq program refers to as an "option".

Options are passed to the ANUPQ interface functions in either of the two usual mechanisms provided by GAP, namely:

- options may be set globally using the function PushOptions (see Chapter **Reference: Options Stack** in the GAP Reference Manual); or
- options may be appended to the argument list of any function call, separated by a colon from the argument list (see Chapter **Reference: Function Calls** in the GAP Reference Manual), in which case they are then passed on recursively to any subsequent inner function call, which may in turn have options of their own.

Particularly, when one is using the interactive functions of Chapter 'Interactive ANUPQ functions', one should, in general, avoid using the global method of passing options. In fact, it is recommended that prior to calling PqStart the OptionsStack be empty. The essential problem with setting options globally using the function PushOptions is that options pushed onto OptionsStack, in this way, (generally) remain there until an explicit PopOptions() call is made.

In contrast, options passed in the usual way behind a colon following a function's arguments (see **Reference: Function Call With Options** in the GAP Reference Manual) are local, and disappear from OptionsStack after the function has executed successfully. If the function does *not* execute successfully, i.e. it runs into error and the user quits the resulting break loop (see Section **Reference: Break Loops** in the Reference Manual) rather than attempting to repair the problem and typing return; then, unless the error at the kernel level, the OptionsStack is reset. If an error is detected inside the kernel (hopefully, this should occur only rarely, if at all) then the options of that function will *not* be cleared from OptionsStack; in such cases:

Example

```
gap> ResetOptionsStack();
#I  Options stack is already empty
```

is usually necessary (see Chapter ResetOptionsStack (**Reference: ResetOptionsStack**) in the GAP Reference Manual), which recursively calls PopOptions() until OptionsStack is empty, or as in the above case warns you that the OptionsStack is already empty.

Note that a function, that is passed options after the colon, will also see any global options or any options passed down recursively from functions calling that function, unless those options are over-ridden by options passed via the function. Also, note that duplication of option names for different programs may lead to misinterpretations, and mis-spelled options will not be "seen".

The non-interactive functions of Chapter 'Non-interactive ANUPQ functions' that have Pq somewhere in their name provide an alternative method of passing options as additional arguments. This has the advantages that options can be abbreviated and mis-spelled options will be trapped.

3.6.1 ANUPQWarnOfOtherOptions

▷ ANUPQWarnOfOtherOptions (global variable)

is a global variable that is by default false. If it is set to true then any function provided by the ANUPQ function that recognises at least one option, will warn you of "other" options, i.e. options that the function does not recognise. These warnings are emitted at InfoWarning or InfoANUPQ level 1. This is useful for detecting mis-spelled options. Here is an example using the function Pq (first described in Chapter 'Non-interactive ANUPQ functions'):

Example

```
gap> SetInfoLevel(InfoANUPQ, 1);        # Set InfoANUPQ to default level
gap> ANUPQWarnOfOtherOptions := true;;
gap> # The following makes entry into break loops very ‘‘quiet’’ ...
gap> OnBreak := function() Where(0); end;;
gap> F := FreeGroup( "a", "b" );
<free group on the generators [ a, b ]>
gap> Pq( F : Prime := 2, Classbound := 1 );
#I  ANUPQ Warning: Options: [ "Classbound" ] ignored
#I  (invalid for generic function: ‘Pq’).
user interrupt at
moreOfline := ReadLine( iostream );
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
```

Here we mistyped ClassBound as Classbound, and after seeing the Info-ed warning that Classbound was ignored, we typed a *control*-C (that's the "user interrupt at" message) which took us into a break loop. Since the Pq command was not able to finish, the options Prime and Classbound, in particular, will still be on the OptionsStack:

Example

```
brk> OptionsStack;
[ rec( Prime := 2, Classbound := 1 ),
  rec( Prime := 2, Classbound := 1, PqEpiOrPCover := "pQuotient" ) ]
```

The option PqEpiOrPCover is a behind-the-scenes option that need not concern the user. On quitting the break-loop the OptionsStack is reset and a warning telling you this is emitted:

Example

```
brk> quit; # to get back to the ‘gap>’ prompt
#I  Options stack has been reset
```

Above, we altered OnBreak (see OnBreak (**Reference: OnBreak**) in the Reference manual) to reduce the back-tracing on entry into a break loop. We now restore OnBreak to its usual value.

Example

```
gap> OnBreak := Where;;
```

Notes

In cases where functions recursively call others with options (e.g. when using `PqExample` with options), setting `ANUPQWarnOfOtherOptions := true` may give rise to spurious "other" option detections.

It is recommended that the novice user set `ANUPQWarnOfOtherOptions` to `true` in their `gap.ini` file (see Section 'Loading the ANUPQ Package').

Other Troubleshooting Strategies

There are some other strategies which may have helped us to see our error above. The function `Pq` recognises the option `OutputLevel` (see 6.2); if this option is set to at least 1, the `pq` program provides information on each class quotient as it is generated:

Example

```
gap> ANUPQWarnOfOtherOptions := false;; # Set back to normal
gap> F := FreeGroup( "a", "b" );;
gap> Pq( F : Prime := 2, Classbound := 1, OutputLevel := 1 );
#I  Lower exponent-2 central series for [grp]
#I  Group: [grp] to lower exponent-2 central class 1 has order 2^2
#I  Group: [grp] to lower exponent-2 central class 2 has order 2^5
#I  Group: [grp] to lower exponent-2 central class 3 has order 2^10
#I  Group: [grp] to lower exponent-2 central class 4 has order 2^18
#I  Group: [grp] to lower exponent-2 central class 5 has order 2^32
#I  Group: [grp] to lower exponent-2 central class 6 has order 2^55
#I  Group: [grp] to lower exponent-2 central class 7 has order 2^96
#I  Group: [grp] to lower exponent-2 central class 8 has order 2^167
#I  Group: [grp] to lower exponent-2 central class 9 has order 2^294
#I  Group: [grp] to lower exponent-2 central class 10 has order 2^520
#I  Group: [grp] to lower exponent-2 central class 11 has order 2^932
#I  Group: [grp] to lower exponent-2 central class 12 has order 2^1679
[... output truncated ...]
```

After seeing the information for the class 2 quotient we may have got the idea that the `Classbound` option was not recognised and may have realised that this was due to a mis-spelling. The above will ordinarily cause the available space to be exhausted, necessitating user-intervention by typing *control*-C and `quit;` (to escape the break loop); otherwise `Pq` terminates when the class reaches 63 (the default value of `ClassBound`).

If you have some familiarity with "keyword" command input to the `pq` binary, then setting the level of `InfoANUPQ` to 4 would also have indicated a problem:

Example

```
gap> ResetOptionsStack(); # Necessary, if a break-loop was entered above
gap> SetInfoLevel(InfoANUPQ, 4);
gap> Pq( F : Prime := 2, Classbound := 1 );
#I  ToPQ> 7  #to (Main) p-Quotient Menu
#I  ToPQ> 1  #define group
#I  ToPQ> name [grp]
#I  ToPQ> prime 2
#I  ToPQ> class 63
#I  ToPQ> exponent 0
#I  ToPQ> output 0
```

```
#I  ToPQ> generators { a,b }
#I  ToPQ> relators   {  };
[... output truncated ...]
```

Here the line "`#I ToPQ> class 63`" indicates that a directive to set the classbound to 63 was sent to the `pq` program.

Chapter 4

Non-interactive ANUPQ functions

Here we describe all the non-interactive functions of the ANUPQ package; i.e. "one-shot" functions that invoke the pq program in such a way that once GAP has got what it needs, the pq program is allowed to exit. It is expected that most of the time users will only need these functions. The functions interface with three of the four algorithms (see Chapter 'Introduction') provided by the ANU pq C program, and are mainly grouped according to the algorithm of the pq program they relate to.

In Section 'Computing p-Quotients', we describe the functions that give access to the p-quotient algorithm.

Section 'Computing Standard Presentations' describe functions that give access to the standard presentation algorithm.

Section 'Testing p-Groups for Isomorphism' describe functions that implement an isomorphism test for p-groups using the standard presentation algorithm.

In Section 'Computing Descendants of a p-Group', we describe functions that give access to the p-group generation algorithm.

To use any of the functions one must have at some stage previously typed:

Example

```
gap> LoadPackage("anupq");
```

(the response of which we have omitted; see 'Loading the ANUPQ Package').

It is strongly recommended that the user try the examples provided. To save typing there is a `PqExample` equivalent for each manual example. We also suggest that to start with you may find the examples more instructive if you set the `InfoANUPQ` level to 2 (see `InfoANUPQ` (3.3.1)).

4.1 Computing p-Quotients

4.1.1 Pq

▷ `Pq(`*`F: options`*`)` (function)

returns for the fp or pc group *`F`*, the p-quotient of *`F`* specified by *`options`*, as a pc group. Following the colon, *`options`* is a selection of the options from the following list, separated by commas like record components (see Section **Reference: Function Call With Options** in the GAP Reference Manual). As a minimum the user *must* supply a value for the `Prime` option. Below we list the options recognised by `Pq` (see Chapter 'ANUPQ Options' for detailed descriptions).

- Prime := *p*
- ClassBound := *n*
- Exponent := *n*
- Relators := *rels*
- Metabelian
- Identities := *funcs*
- GroupName := *name*
- OutputLevel := *n*
- SetupFile := *filename*
- PqWorkspace := *workspace*

Notes: Pq may also be called with no arguments or one integer argument, in which case it is being used interactively (see Pq (5.3.1)); the same options may be used, except that SetupFile and PqWorkspace are ignored by the interactive Pq function.

See Section 'Attributes and a Property for fp and pc p-groups' for the attributes and property NuclearRank, MultiplicatorRank and IsCapable which may be applied to the group returned by Pq.

See also PqEpimorphism (PqEpimorphism (4.1.2)).

We now give a few examples of the use of Pq. Except for the addition of a few comments and the non-suppression of output (by not using duplicated semicolons) the next 3 examples may be run by typing: PqExample("Pq"); (see PqExample (3.4.4)).

Example

```
gap> LoadPackage("anupq");; # does nothing if ANUPQ is already loaded
gap> # First we get a p-quotient of a free group of rank 2
gap> F := FreeGroup("a", "b");; a := F.1;; b := F.2;;
gap> Pq( F : Prime := 2, ClassBound := 3 );
<pc group of size 1024 with 10 generators>
gap> # Now let us get a p-quotient of an fp group
gap> G := F / [a^4, b^4];
<fp group on the generators [ a, b ]>
gap> Pq( G : Prime := 2, ClassBound := 3 );
<pc group of size 256 with 8 generators>
gap> # Now let's get a different p-quotient of the same group
gap> Pq( G : Prime := 2, ClassBound := 3, Exponent := 4 );
<pc group of size 128 with 7 generators>
gap> # Now we'll get a p-quotient of another fp group
gap> # which we will redo using the 'Relators' option
gap> R := [ a^25, Comm(Comm(b, a), a), b^5 ];
[ a^25, a^-1*b^-1*a*b*a^-1*b^-1*a^-1*b*a^2, b^5 ]
gap> H := F / R;
<fp group on the generators [ a, b ]>
gap> Pq( H : Prime := 5, ClassBound := 5, Metabelian );
<pc group of size 78125 with 7 generators>
```

Now we redo the last example to show how one may use the Relators option. Observe that Comm(Comm(b, a), a) is a left normed commutator which must be written in square bracket notation for the pq program and embedded in a pair of double quotes. The function PqGAPRelators (see PqGAPRelators (3.4.2)) can be used to translate a list of strings prepared for the Relators option into GAP format. Below we use it. Observe that the value of R is the same as before.

Example

```
gap> F := FreeGroup("a", "b");;
gap> # 'F' was defined for 'Relators'. We use the same strings that GAP uses
gap> # for printing the free group generators. It is *not* necessary to
gap> # predefine: a := F.1; etc. (as it was above).
gap> rels := [ "a^25", "[b, a, a]", "b^5" ];
[ "a^25", "[b, a, a]", "b^5" ]
gap> R := PqGAPRelators(F, rels);
[ a^25, a^-1*b^-1*a*b*a^-1*b^-1*a^-1*b*a^2, b^5 ]
gap> H := F / R;
<fp group on the generators [ a, b ]>
gap> Pq( H : Prime := 5, ClassBound := 5, Metabelian,
>            Relators := rels );
<pc group of size 78125 with 7 generators>
```

In fact, above we could have just passed F (rather than H), i.e. we could have done:

Example

```
gap> F := FreeGroup("a", "b");;
gap> rels := [ "a^25", "[b, a, a]", "b^5" ];
[ "a^25", "[b, a, a]", "b^5" ]
gap> Pq( F : Prime := 5, ClassBound := 5, Metabelian,
>            Relators := rels );
<pc group of size 78125 with 7 generators>
```

The non-interactive Pq function also allows the options to be passed in two other ways; these alternatives have been included for those familiar with the GAP 3 version of the ANUPQ package; the preferred method of passing options is the one already described. Firstly, they may be passed in a record as a second argument; note that any boolean options must be set explicitly e.g.

Example

```
gap> Pq( H, rec( Prime := 5, ClassBound := 5, Metabelian := true ) );
<pc group of size 78125 with 7 generators>
```

It is also possible to pass them as extra arguments, where each option name appears as a string followed immediately by its value (if not a boolean option) e.g.

Example

```
gap> Pq( H, "Prime", 5, "ClassBound", 5, "Metabelian" );
<pc group of size 78125 with 7 generators>
```

The preceding two examples can be run from GAP via PqExample("Pq-ni"); (see PqExample (3.4.4)).

This method of passing options permits abbreviation; the only restriction is that the abbreviation must be unique. So "Pr" may be used for "Prime", "Class" or even just "C" for "ClassBound", etc.

The following example illustrates the use of the option Identities. We compute the largest finite Burnside group of exponent 5 that also satisfies the 3-Engel identity. Each identity is defined by a

function whose arguments correspond to the variables of the identity. The return value of each of those functions is the identity evaluated on the arguments of the function.

Example

```
gap> F := FreeGroup(2);
<free group on the generators [ f1, f2 ]>
gap> Burnside5 := x->x^5;
function( x ) ... end
gap> Engel3 := function( x,y ) return PqLeftNormComm( [x,y,y,y] ); end;
function( x, y ) ... end
gap> Pq( F : Prime := 5, Identities := [ Burnside5, Engel3 ] );
#I  Class 1 with 2 generators.
#I  Class 2 with 3 generators.
#I  Class 3 with 5 generators.
#I  Class 3 with 5 generators.
<pc group of size 3125 with 5 generators>
```

The above example can be run from GAP via `PqExample( "B5-5-Engel3-Id" );` (see `PqExample` (3.4.4)).

4.1.2 PqEpimorphism

▷ `PqEpimorphism(`*F: options*`)` (function)

returns for the fp or pc group *F* an epimorphism from *F* onto the *p*-quotient of *F* specified by *options*; the possible options *options* and *required* option (`"Prime"`) are as for `Pq` (see `Pq` (4.1.1)). `PqEpimorphism` only differs from `Pq` in what it outputs; everything about what must/may be passed as input to `PqEpimorphism` is the same as for `Pq`. The same alternative methods of passing options to the non-interactive `Pq` function are available to the non-interactive version of `PqEpimorphism`.

Notes: `PqEpimorphism` may also be called with no arguments or one integer argument, in which case it is being used interactively (see `PqEpimorphism` (5.3.2)), and the options `SetupFile` and `PqWorkspace` are ignored by the interactive `PqEpimorphism` function.

See Section 'Attributes and a Property for fp and pc p-groups' for the attributes and property `NuclearRank`, `MultiplicatorRank` and `IsCapable` which may be applied to the image group of the epimorphism returned by `PqEpimorphism`.

Example

```
gap> F := FreeGroup (2, "F");
<free group on the generators [ F1, F2 ]>
gap> phi := PqEpimorphism( F : Prime := 5, ClassBound := 2 );
[ F1, F2 ] -> [ f1, f2 ]
gap> Image( phi );
<pc group of size 3125 with 5 generators>
```

Typing: `PqExample( "PqEpimorphism" );` runs the above example in GAP (see `PqExample` (3.4.4)).

4.1.3 PqPCover

▷ `PqPCover(`*F: options*`)` (function)

returns for the fp or pc group *F*, the *p*-covering group of the *p*-quotient of *F* specified by *options*, as a pc group, i.e. the *p*-covering group of the *p*-quotient `Pq( F : options )`. Thus the options that `PqPCover` accepts are exactly those expected for `Pq` (and hence as a minimum the user *must* supply a value for the `Prime` option; see `Pq` (4.1.1) for more details), except in the following special case.

If *F* is already a *p*-group, in the sense that `IsPGroup(F)` is `true`, then

`Prime`
: defaults to `PrimePGroup(F)`, if not supplied and `HasPrimePGroup(F) = true`; and

`ClassBound`
: defaults to `PClassPGroup(F)` if `HasPClassPGroup(F) = true` if not supplied, or to the usual default of 63, otherwise.

The same alternative methods of passing options to the non-interactive `Pq` function are available to the non-interactive version of `PqPCover`.

We now give a few examples of the use of `PqPCover`. These examples are just a subset of the ones we gave for `Pq` (see `Pq` (4.1.1)), except that in each instance the command `Pq` has been replaced with `PqPCover`. Essentially the same examples may be run by typing: `PqExample( "PqPCover" );` (see `PqExample` (3.4.4)).

Example

```
gap> F := FreeGroup("a", "b");; a := F.1;; b := F.2;;
gap> PqPCover( F : Prime := 2, ClassBound := 3 );
<pc group of size 262144 with 18 generators>
gap>
gap> # Now let's get a p-cover of a p-quotient of an fp group
gap> G := F / [a^4, b^4];
<fp group on the generators [ a, b ]>
gap> PqPCover( G : Prime := 2, ClassBound := 3 );
<pc group of size 16384 with 14 generators>
gap>
gap> # Now let's get a p-cover of a different p-quotient of the same group
gap> PqPCover( G : Prime := 2, ClassBound := 3, Exponent := 4 );
<pc group of size 8192 with 13 generators>
gap>
gap> # Now we'll get a p-cover of a p-quotient of another fp group
gap> # which we will redo using the 'Relators' option
gap> R := [ a^25, Comm(Comm(b, a), a), b^5 ];
[ a^25, a^-1*b^-1*a*b*a^-1*b^-1*a^-1*b*a^2, b^5 ]
gap> H := F / R;
<fp group on the generators [ a, b ]>
gap> PqPCover( H : Prime := 5, ClassBound := 5, Metabelian );
<pc group of size 48828125 with 11 generators>
gap>
gap> # Now we redo the previous example using the 'Relators' option
gap> F := FreeGroup("a", "b");;
gap> rels := [ "a^25", "[b, a, a]", "b^5" ];
[ "a^25", "[b, a, a]", "b^5" ]
gap> PqPCover( F : Prime := 5, ClassBound := 5, Metabelian,
>                  Relators := rels );
<pc group of size 48828125 with 11 generators>
```

4.2 Computing Standard Presentations

4.2.1 PqStandardPresentation

▷ `PqStandardPresentation(`*`F: options`*`)` (function)
▷ `StandardPresentation(`*`F: options`*`)` (method)

return the p-quotient specified by *options* of the fp or pc p-group F, as an *fp group* which has a standard presentation. Here *options* is a selection of the options from the following list (see Chapter 'ANUPQ Options' for detailed descriptions). Section 'Hints and Warnings regarding the use of Options' gives some important hints and warnings regarding option usage, and Section **Reference: Function Call With Options** in the GAP Reference Manual describes their "record"-like syntax.

- `Prime :=` *p*
- `pQuotient :=` *Q*
- `ClassBound :=` *n*
- `Exponent :=` *n*
- `Metabelian`
- `GroupName :=` *name*
- `OutputLevel :=` *n*
- `StandardPresentationFile :=` *filename*
- `SetupFile :=` *filename*
- `PqWorkspace :=` *workspace*

Unless F is a pc p-group, the user *must* supply either the option `Prime` or the option `pQuotient` (if both `Prime` and `pQuotient` are supplied, the prime p is determined by applying `PrimePGroup` (see `PrimePGroup` (**Reference: PrimePGroup**) in the Reference Manual) to the value of `pQuotient`).

The options for `PqStandardPresentation` may also be passed in the two other alternative ways described for `Pq` (see `Pq` (4.1.1)). `StandardPresentation` does not provide these alternative ways of passing options.

Notes: In contrast to the function `Pq` (see `Pq` (4.1.1)) which returns a pc group, `PqStandardPresentation` or `StandardPresentation` returns an fp group. This is because the output is mainly used for isomorphism testing for which an fp group is enough. However, the presentation is a polycyclic presentation and if you need to do any further computation with this group (e.g. to find the order) you can use the function `PcGroupFpGroup` (see `PcGroupFpGroup` (**Reference: PcGroupFpGroup**) in the GAP Reference Manual) to form a pc group.

If the user does not supply a p-quotient Q via the `pQuotient` option and the prime p is either supplied or F is a pc p-group, then a p-quotient Q is computed. If the user does supply a p-quotient Q via the `pQuotient` option, the package AutPGrp is called to compute the automorphism group of Q; an error will occur that asks the user to install the package AutPGrp if the automorphism group cannot be computed.

The attributes and property NuclearRank, MultiplicatorRank and IsCapable are set for the group returned by PqStandardPresentation or StandardPresentation (see Section 'Attributes and a Property for fp and pc p-groups').

We illustrate the method with the following examples.

Example

```
gap> F := FreeGroup( "a", "b" );; a := F.1;; b := F.2;;
gap> G := F / [a^25, Comm(Comm(b, a), a), b^5];
<fp group on the generators [ a, b ]>
gap> S := StandardPresentation( G : Prime := 5, ClassBound := 10 );
<fp group on the generators [ f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11,
  f12, f13, f14, f15, f16, f17, f18, f19, f20, f21, f22, f23, f24, f25, f26 ]>
gap> IsPcGroup( S );
false
gap> # if we need to compute with S we should convert it to a pc group
gap> Spc := PcGroupFpGroup( S );
<pc group of size 1490116119384765625 with 26 generators>
gap>
gap> H := F / [ a^625, Comm(Comm(Comm(Comm(b, a), a), a), a)/Comm(b, a)^5,
>               Comm(Comm(b, a), b), b^625 ];;
gap> StandardPresentation( H : Prime := 5, ClassBound := 15, Metabelian );
<fp group on the generators [ f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11,
  f12, f13, f14, f15, f16, f17, f18, f19, f20 ]>
gap>
gap> F4 := FreeGroup( "a", "b", "c", "d" );;
gap> a := F4.1;; b := F4.2;; c := F4.3;; d := F4.4;;
gap> G4 := F4 / [ b^4, b^2 / Comm(Comm (b, a), a), d^16,
>                 a^16 / (c * d), b^8 / (d * c^4) ];
<fp group on the generators [ a, b, c, d ]>
gap> K := Pq( G4 : Prime := 2, ClassBound := 1 );
<pc group of size 4 with 2 generators>
gap> StandardPresentation( G4 : pQuotient := K, ClassBound := 14 );
<fp group with 53 generators>
```

Typing: PqExample("StandardPresentation"); runs the above example in GAP (see PqExample (3.4.4)).

4.2.2 EpimorphismPqStandardPresentation

▷ EpimorphismPqStandardPresentation(*F: options*) (function)

▷ EpimorphismStandardPresentation(*F: options*) (method)

Each of the above functions accepts the same arguments and options as the function StandardPresentation (see StandardPresentation (4.2.1)) and returns an epimorphism from the fp or pc group F onto the finitely presented group given by a standard presentation, i.e. if S is the standard presentation computed for the p-quotient of F by StandardPresentation then EpimorphismStandardPresentation returns the epimorphism from F to the group with presentation S.

Note: The attributes and property NuclearRank, MultiplicatorRank and IsCapable are set for the image group of the epimorphism returned by EpimorphismPqStandardPresentation or EpimorphismStandardPresentation (see Section 'Attributes and a Property for fp and pc p-groups').

We illustrate the function with the following example.

Example

```
gap> F := FreeGroup(6, "F");
<free group on the generators [ F1, F2, F3, F4, F5, F6 ]>
gap> # For printing GAP uses the symbols F1, ... for the generators of F
gap> x := F.1;; y := F.2;; z := F.3;; w := F.4;; a := F.5;; b := F.6;;
gap> R := [x^3 / w, y^3 / w * a^2 * b^2, w^3 / b,
>          Comm (y, x) / z, Comm (z, x), Comm (z, y) / a, z^3 ];;
gap> Q := F / R;
<fp group on the generators [ F1, F2, F3, F4, F5, F6 ]>
gap> # For printing GAP also uses the symbols F1, ... for the generators of Q
gap> # (the same as used for F) ... but the gen'rs of Q and F are different:
gap> GeneratorsOfGroup(F) = GeneratorsOfGroup(Q);
false
gap> G := Pq( Q : Prime := 3, ClassBound := 3 );
<pc group of size 729 with 6 generators>
gap> phi := EpimorphismStandardPresentation( Q : Prime := 3,
>                                               ClassBound := 3 );
[ F1, F2, F3, F4, F5, F6 ] -> [ f1*f2^2*f3*f4^2*f5^2, f1*f2*f3*f5, f3^2,
  f4*f6^2, f5, f6 ]
gap> Source(phi); # This is the group Q (GAP uses F1, ... for gen'r symbols)
<fp group of size infinity on the generators [ F1, F2, F3, F4, F5, F6 ]>
gap> Range(phi);  # This is the group G (GAP uses f1, ... for gen'r symbols)
<fp group on the generators [ f1, f2, f3, f4, f5, f6 ]>
gap> AssignGeneratorVariables(G);
#I  Assigned the global variables [ f1, f2, f3, f4, f5, f6 ]
gap> # Just to see that the images of [F1, ..., F6] do generate G
gap> Group([ f1*f2^2*f3, f1*f2*f3*f4*f5^2*f6^2, f3^2, f4, f5, f6 ]) = G;
true
gap> Size( Image(phi) );
729
```

Typing: `PqExample( "EpimorphismStandardPresentation" );` runs the above example in GAP (see `PqExample` (3.4.4)). Note that `AssignGeneratorVariables` (see `AssignGeneratorVariables` (**Reference: AssignGeneratorVariables**)) has only been available since GAP 4.3.

4.3 Testing p-Groups for Isomorphism

4.3.1 IsPqIsomorphicPGroup

▷ `IsPqIsomorphicPGroup(`*G*`,` *H*`)` (function)
▷ `IsIsomorphicPGroup(`*G*`,` *H*`)` (method)

each return true if *G* is isomorphic to *H*, where both *G* and *H* must be pc groups of prime power order. These functions compute and compare in GAP the fp groups given by standard presentations for *G* and *H* (see `StandardPresentation` (4.2.1)).

Example

```
gap> G := Group( (1,2,3,4), (1,3) );
Group([ (1,2,3,4), (1,3) ])
gap> P1 := Image( IsomorphismPcGroup( G ) );
```

```
Group([ f1, f2, f3 ])
gap> P2 := SmallGroup( 8, 5 );
<pc group of size 8 with 3 generators>
gap> IsIsomorphicPGroup( P1, P2 );
false
gap> P3 := SmallGroup( 8, 4 );
<pc group of size 8 with 3 generators>
gap> IsIsomorphicPGroup( P1, P3 );
false
gap> P4 := SmallGroup( 8, 3 );
<pc group of size 8 with 3 generators>
gap> IsIsomorphicPGroup( P1, P4 );
true
```

Typing: `PqExample( "IsIsomorphicPGroup" );` runs the above example in GAP (see `PqExample` (3.4.4)).

4.4 Computing Descendants of a p-Group

4.4.1 PqDescendants

▷ `PqDescendants(`*`G: options`*`)` (function)

returns, for the pc group *G* which must be of prime power order with a confluent pc presentation (see `IsConfluent` (**Reference: IsConfluent for pc groups**) in the GAP Reference Manual), a list of descendants (pc groups) of *G*. Following the colon *options* a selection of the options listed below should be given, separated by commas like record components (see Section **Reference: Function Call With Options** in the GAP Reference Manual). See Chapter 'ANUPQ Options' for detailed descriptions of the options.

The automorphism group of each descendant *D* is also computed via a call to the `AutomorphismGroupPGroup` function of the AutPGrp package.

- `ClassBound := `*`n`*
- `Relators := `*`rels`*
- `OrderBound := `*`n`*
- `StepSize := `*`n`*, `StepSize := `*`list`*
- `RankInitialSegmentSubgroups := `*`n`*
- `SpaceEfficient`
- `CapableDescendants`
- `AllDescendants := false`
- `Exponent := `*`n`*
- `Metabelian`

- GroupName := *name*
- SubList := *sub*
- BasicAlgorithm
- CustomiseOutput := *rec*
- SetupFile := *filename*
- PqWorkspace := *workspace*

Notes: The function PqDescendants uses the automorphism group of *G* which it computes via the package AutPGrp. If this package is not installed an error may be raised. If the automorphism group of *G* is insoluble, the pq program will call GAP together with the AutPGrp package for certain orbit-stabilizer calculations. (So, in any case, one should ensure the AutPGrp package is installed.)

The attributes and property NuclearRank, MultiplicatorRank and IsCapable are set for each group of the list returned by PqDescendants (see Section 'Attributes and a Property for fp and pc p-groups').

The options *options* for PqDescendants may be passed in an alternative manner to that already described, namely you can pass PqDescendants a record as an argument, which contains as entries some (or all) of the above mentioned. Those parameters which do not occur in the record are set to their default values.

Note that you cannot set both OrderBound and StepSize.

In the first example we compute all descendants of the Klein four group which have exponent-2 class at most 5 and order at most 2^6.

Example

```
gap> F := FreeGroup( "a", "b" );; a := F.1;; b := F.2;;
gap> G := PcGroupFpGroup( F / [ a^2, b^2, Comm(b, a) ] );
<pc group of size 4 with 2 generators>
gap> des := PqDescendants( G : OrderBound := 6, ClassBound := 5 );;
gap> Length(des);
83
gap> List(des, Size);
[ 8, 8, 8, 16, 16, 16, 32, 16, 16, 16, 16, 16, 32, 32, 64, 64, 32, 32, 32,
  32, 32, 32, 32, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 32, 32, 32, 32,
  64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 32, 32, 32, 32, 32, 64, 64, 64,
  64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
  64, 64, 64, 64, 64, 64, 64 ]
gap> List(des, d -> Length( PCentralSeries( d, 2 ) ) - 1 );
[ 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
  3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4,
  4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
  4, 4, 4, 5, 5, 5, 5, 5 ]
```

Below, we compute all capable descendants of order 27 of the elementary abelian group of order 9.

Example

```
gap> F := FreeGroup( 2, "g" );
<free group on the generators [ g1, g2 ]>
gap> G := PcGroupFpGroup( F / [ F.1^3, F.2^3, Comm(F.1, F.2) ] );
<pc group of size 9 with 2 generators>
gap> des := PqDescendants( G : OrderBound := 3, ClassBound := 2,
```

```
>                               CapableDescendants );
[ <pc group of size 27 with 3 generators>,
  <pc group of size 27 with 3 generators> ]
gap> List(des, d -> Length( PCentralSeries( d, 3 ) ) - 1 );
[ 2, 2 ]
gap> # For comparison let us now compute all descendants
gap> PqDescendants( G : OrderBound := 3, ClassBound := 2);
[ <pc group of size 27 with 3 generators>,
  <pc group of size 27 with 3 generators>,
  <pc group of size 27 with 3 generators> ]
```

In the third example, we compute all capable descendants of the elementary abelian group of order 5^2 which have exponent-5 class at most 3, exponent 5, and are metabelian.

Example

```
gap> F := FreeGroup( 2, "g" );;
gap> G := PcGroupFpGroup( F / [ F.1^5, F.2^5, Comm(F.2, F.1) ] );
<pc group of size 25 with 2 generators>
gap> des := PqDescendants( G : Metabelian, ClassBound := 3,
>                               Exponent := 5, CapableDescendants );
[ <pc group of size 125 with 3 generators>,
  <pc group of size 625 with 4 generators>,
  <pc group of size 3125 with 5 generators> ]
gap> List(des, d -> Length( PCentralSeries( d, 5 ) ) - 1 );
[ 2, 3, 3 ]
gap> List(des, d -> Length( DerivedSeries( d ) ) );
[ 3, 3, 3 ]
gap> List(des, d -> Maximum( List( d, Order ) ) );
[ 5, 5, 5 ]
```

The examples "PqDescendants-1", "PqDescendants-2" and "PqDescendants-3" (in order) are essentially the same as the above three examples (see PqExample (3.4.4)).

4.4.2 PqSupplementInnerAutomorphisms

▷ PqSupplementInnerAutomorphisms(*D*) (function)

returns a generating set for a supplement to the inner automorphisms of *D*, in the form of a record with fields agAutos, agOrder and glAutos, as provided by the pq program. One should be very careful in using these automorphisms for a descendant calculation.

Note: In principle there must be a way to use those automorphisms in order to compute descendants but there does not seem to be a way to hand back these automorphisms properly to the pq program.

Example

```
gap> Q := Pq( FreeGroup(2) : Prime := 3, ClassBound := 1 );
<pc group of size 9 with 2 generators>
gap> des := PqDescendants( Q : StepSize := 1 );
[ <pc group of size 27 with 3 generators>,
  <pc group of size 27 with 3 generators>,
  <pc group of size 27 with 3 generators> ]
gap> S := PqSupplementInnerAutomorphisms( des[3] );
rec( agAutos := [  ], agOrder := [ 3, 2, 2, 2 ],
```

```
    glAutos := [ Pcgs([ f1, f2, f3 ]) -> [ f1*f2^2, f2, f3 ],
        Pcgs([ f1, f2, f3 ]) -> [ f1^2, f2, f3^2 ],
        Pcgs([ f1, f2, f3 ]) -> [ f1^2, f2, f3^2 ] ] )
  gap> A := AutomorphismGroupPGroup( des[3] );
  rec(
    agAutos := [ Pcgs([ f1, f2, f3 ]) -> [ f1^2, f2, f3^2 ],
        Pcgs([ f1, f2, f3 ]) -> [ f1*f2^2, f2, f3 ],
        Pcgs([ f1, f2, f3 ]) -> [ f1*f3, f2, f3 ],
        Pcgs([ f1, f2, f3 ]) -> [ f1, f2*f3, f3 ] ], agOrder := [ 2, 3, 3, 3 ],
    glAutos := [  ], glOper := [  ], glOrder := 1,
    group := <pc group of size 27 with 3 generators>,
    one := IdentityMapping( <pc group of size 27 with 3 generators> ),
    size := 54 )
```

Typing: `PqExample( "PqSupplementInnerAutomorphisms" );` runs the above example in GAP (see `PqExample` (3.4.4)).

Note that by also including `PqStart` as a second argument to `PqExample` one can see how it is possible, with the aid of `PqSetPQuotientToGroup` (see `PqSetPQuotientToGroup` (5.3.7)), to do the equivalent computations with the interactive versions of `Pq` and `PqDescendants` and a single `pq` process (recall `pq` is the name of the external C program).

4.4.3 PqList

▷ `PqList(`*`filename: [SubList := sub]`*`)` (function)

reads a file with name *`filename`* (a string) and returns the list *`L`* of pc groups (or with option `SubList` a sublist of *`L`* or a single pc group in *`L`*) defined in that file. If the option `SubList` is passed and has the value *`sub`*, then it has the same meaning as for `PqDescendants`, i.e. if *`sub`* is an integer then `PqList` returns *`L`*`[`*`sub`*`]`; otherwise, if *`sub`* is a list of integers `PqList` returns `Sublist(`*`L, sub`*`)`.

Both `PqList` and `SavePqList` (see `SavePqList` (4.4.4)) can be used to save and restore a list of descendants (see `PqDescendants` (4.4.1)).

4.4.4 SavePqList

▷ `SavePqList(`*`filename, list`*`)` (function)

writes a list of descendants *`list`* to a file with name *`filename`* (a string).

`SavePqList` and `PqList` (see `PqList` (4.4.3)) can be used to save and restore, respectively, the results of `PqDescendants` (see `PqDescendants` (4.4.1)).

Chapter 5

Interactive ANUPQ functions

Here we describe the interactive functions defined by the ANUPQ package, i.e. the functions that manipulate and initiate interactive ANUPQ processes. These are functions that extract information via a dialogue with a running pq process (process used in the UNIX sense). Occasionally, a user needs the "next step"; the functions provided in this chapter make use of data from previous steps retained by the pq program, thus allowing the user to interact with the pq program like one can when one uses the pq program as a stand-alone (see guide.dvi in the standalone-doc directory).

An interactive ANUPQ process is initiated by PqStart and terminated via PqQuit; these functions are described in ection 'Starting and Stopping Interactive ANUPQ Processes'.

Each interactive ANUPQ function that manipulates an already started interactive ANUPQ process, has a form where the first argument is the integer *i* returned by the initiating PqStart command, and a second form with one argument fewer (where the integer *i* is discovered by a default mechanism, namely by determining the least integer *i* for which there is a currently active interactive ANUPQ process). We will thus commonly say that "for the *i*th (or default) interactive ANUPQ process" a certain function performs a given action. In each case, it is an error if *i* is not the index of an active interactive process, or there are no current active interactive processes.

Notes: The global method of passing options (via PushOptions), should not be used with any of the interactive functions. In fact, the OptionsStack should be empty at the time any of the interactive functions is called.

On quitting GAP, PqQuitAll(); is executed, which terminates all active interactive ANUPQ processes. If GAP is killed without quitting, before all interactive ANUPQ processes are terminated, *zombie* processes (still living *child* processes whose *parents* have died), may result. Since zombie processes do consume resources, in such an event, the responsible computer user should seek out and terminate those zombie processes (e.g. on Linux: ps xw | grep pq gives you information on the pq processes corresponding to any interactive ANUPQ processes started in a GAP session; you can then do kill *N* for each number *N* appearing in the first column of this output).

5.1 Starting and Stopping Interactive ANUPQ Processes

5.1.1 PqStart (with group and workspace size)

▷ PqStart(*G, workspace: options*) (function)
▷ PqStart(*G: options*) (function)
▷ PqStart(*workspace: options*) (function)

▷ PqStart(: options) (function)

activate an iostream for an interactive ANUPQ process (i.e. PqStart starts up a pq process and opens a GAP iostream to "talk" to that process) and returns an integer *i* that can be used to identify that process. The argument *G* should be an *fp group* or *pc group* that the user intends to manipulate using interactive ANUPQ functions. If the function is called without specifying *G*, a group can be read in by using the function PqRestorePcPresentation (see PqRestorePcPresentation (5.6.3)). If PqStart is given an integer argument *workspace*, then the pq program is started up with a workspace (an integer array) of size *workspace* (i.e. $4 \times$ *workspace* bytes in a 32-bit environment); otherwise, the pq program sets a default workspace of 10000000.

The only *options* currently recognised by PqStart are Prime, Exponent and Relators (see Chapter 'ANUPQ Options' for detailed descriptions of these options) and if provided they are essentially global for the interactive ANUPQ process, except that any interactive function interacting with the process and passing new values for these options will over-ride the global values.

5.1.2 PqQuit

▷ PqQuit(*i*) (function)
▷ PqQuit() (function)

closes the stream of the *i*th or default interactive ANUPQ process and unbinds its ANUPQData.io record.

Note: It can happen that the pq process, and hence the GAP iostream assigned to communicate with it, can die, e.g. by the user typing a Ctrl-C while the pq process is engaged in a long calculation. IsPqProcessAlive (see IsPqProcessAlive (5.2.3)) is provided to check the status of the GAP iostream (and hence the status of the pq process it was communicating with).

5.1.3 PqQuitAll

▷ PqQuitAll() (function)

is provided as a convenience, to terminate all active interactive ANUPQ processes with a single command. It is equivalent to executing PqQuit(*i*) for all active interactive ANUPQ processes *i* (see PqQuit (5.1.2)).

5.2 Interactive ANUPQ Process Utility Functions

5.2.1 PqProcessIndex

▷ PqProcessIndex(*i*) (function)
▷ PqProcessIndex() (function)

With argument *i*, which must be a positive integer, PqProcessIndex returns *i* if it corresponds to an active interactive process, or raises an error. With no arguments it returns the default active interactive process or returns fail and emits a warning message to Info at InfoANUPQ or InfoWarning level 1.

Note: Essentially, an interactive ANUPQ process *i* is "active" if ANUPQData.io[*i*] is bound (i.e. we still have some data telling us about it). Also see PqStart (5.1.1).

5.2.2 PqProcessIndices

▷ PqProcessIndices() (function)

returns the list of integer indices of all active interactive ANUPQ processes (see PqProcessIndex (5.2.1) for the meaning of "active").

5.2.3 IsPqProcessAlive

▷ IsPqProcessAlive(*i*) (function)
▷ IsPqProcessAlive() (function)

return true if the GAP iostream of the *i*th (or default) interactive ANUPQ process started by PqStart is alive (i.e. can still be written to), or false, otherwise. (See the notes for PqStart (5.1.1) and PqQuit (5.1.2).)

If the user does not yet have a gap> prompt then usually the pq program is still away doing something and an ANUPQ interface function is still waiting for a reply. Typing a Ctrl-C (i.e. holding down the Ctrl key and typing c) will stop the waiting and send GAP into a break-loop, from which one has no option but to quit;. The typing of Ctrl-C, in such a circumstance, usually causes the stream of the interactive ANUPQ process to die; to check this we provide IsPqProcessAlive (see IsPqProcessAlive).

The GAP iostream of an interactive ANUPQ process will also die if the pq program has a segmentation fault. We do hope that this never happens to you, but if it does and the failure is reproducible, then it's a bug and we'd like to know about it. Please read the README that comes with the ANUPQ package to find out what to include in a bug report and who to email it to.

5.3 Interactive Versions of Non-interactive ANUPQ Functions

5.3.1 Pq (interactive)

▷ Pq(*i*: *options*) (function)
▷ Pq(: *options*) (function)

return, for the fp or pc group (let us call it *F*), of the *i*th or default interactive ANUPQ process, the *p*-quotient of *F* specified by *options*, as a pc group; *F* must previously have been given (as first argument) to PqStart to start the interactive ANUPQ process (see PqStart (5.1.1)) or restored from file using the function PqRestorePcPresentation (see PqRestorePcPresentation (5.6.3)). Following the colon *options* is a selection of the options listed for the non-interactive Pq function (see Pq (4.1.1)), separated by commas like record components (see Section **Reference: Function Call With Options** in the GAP Reference Manual), except that the options SetupFile or PqWorkspace are ignored by the interactive Pq, and RedoPcp is an option only recognised by the interactive Pq i.e. the following options are recognised by the interactive Pq function:

- Prime := *p*

- ClassBound := *n*
- Exponent := *n*
- Relators := *rels*
- Metabelian
- Identities := *funcs*
- GroupName := *name*
- OutputLevel := *n*
- RedoPcp

Detailed descriptions of the above options may be found in Chapter 'ANUPQ Options'.

As a minimum the Pq function *must* have a value for the Prime option, though Prime need not be passed again in the case it has previously been provided, e.g. to PqStart (see PqStart (5.1.1)) when starting the interactive process.

The behaviour of the interactive Pq function depends on the current state of the pc presentation stored by the pq program:

1. If no pc presentation has yet been computed (the case immediately after the PqStart call initiating the process) then the quotient group of the input group of the process of largest lower exponent-p class bounded by the value of the ClassBound option (see 6.2) is returned.
2. If the current pc presentation of the process was determined by a previous call to Pq or PqEpimorphism, and the current call has a larger value ClassBound then the class is extended as much as is possible and the quotient group of the input group of the process of the new lower exponent-p class is returned.
3. If the current pc presentation of the process was determined by a previous call to PqPCover then a consistent pc presentation of a quotient for the current class is determined before proceeding as in 2.
4. If the RedoPcp option is supplied the current pc presentation is scrapped, all options must be re-supplied (in particular, Prime *must* be supplied) and then the Pq function proceeds as in 1.

See Section 'Attributes and a Property for fp and pc p-groups' for the attributes and property NuclearRank, MultiplicatorRank and IsCapable which may be applied to the group returned by Pq.

The following is one of the examples for the non-interactive Pq redone with the interactive version. Also, we set the option OutputLevel to 1 (see 6.2), in order to see the orders of the quotients of all the classes determined, and we set the InfoANUPQ level to 2 (see InfoANUPQ (3.3.1)), so that we catch the timing information.

Example

```
gap> F := FreeGroup("a", "b");; a := F.1;; b := F.2;;
gap> G := F / [a^4, b^4];
<fp group on the generators [ a, b ]>
gap> PqStart(G);
1
```

```
gap> SetInfoLevel(InfoANUPQ, 2); #To see timing information
gap> Pq(: Prime := 2, ClassBound := 3, OutputLevel := 1 );
#I  Lower exponent-2 central series for [grp]
#I  Group: [grp] to lower exponent-2 central class 1 has order 2^2
#I  Group: [grp] to lower exponent-2 central class 2 has order 2^5
#I  Group: [grp] to lower exponent-2 central class 3 has order 2^8
#I  Computation of presentation took 0.00 seconds
<pc group of size 256 with 8 generators>
```

5.3.2 PqEpimorphism (interactive)

▷ PqEpimorphism(*i*: *options*) (function)
▷ PqEpimorphism(: *options*) (function)

return, for the fp or pc group (let us call it *F*), of the *i*th or default interactive ANUPQ process, an epimorphism from *F* onto the *p*-quotient of *F* specified by *options*; *F* must previously have been given (as first argument) to PqStart to start the interactive ANUPQ process (see PqStart (5.1.1)). Since the underlying interactions with the pq program effected by the interactive PqEpimorphism are identical to those effected by the interactive Pq, everything said regarding the requirements and behaviour of the interactive Pq function (see Pq (5.3.1)) is also the case for the interactive PqEpimorphism.

Note: See Section 'Attributes and a Property for fp and pc p-groups' for the attributes and property NuclearRank, MultiplicatorRank and IsCapable which may be applied to the image group of the epimorphism returned by PqEpimorphism.

5.3.3 PqPCover (interactive)

▷ PqPCover(*i*: *options*) (function)
▷ PqPCover(: *options*) (function)

return, for the fp or pc group of the *i*th or default interactive ANUPQ process, the *p*-covering group of the *p*-quotient Pq(*i* : *options*) or Pq(: *options*), modulo the following:

1. If no pc presentation has yet been computed (the case immediately after the PqStart call initiating the process) and the group *F* of the process is already a *p*-group, in the sense that HasIsPGroup(*F*) and IsPGroup(*F*) is true, then

   Prime
   : defaults to PrimePGroup(*F*), if not supplied and HasPrimePGroup(*F*) = true; and

   ClassBound
   : defaults to PClassPGroup(*F*) if HasPClassPGroup(*F*) = true if not supplied, or to the usual default of 63, otherwise.

2. If a pc presentation has been computed and none of *options* is RedoPcp or if no pc presentation has yet been computed but 1. does not apply then PqPCover(*i* : *options*); is equivalent to:

```
Pq(i : options);
PqPCover(i);
```

3. If the RedoPcp option is supplied the current pc presentation is scrapped, and PqPCover proceeds as in 1. or 2. but without the RedoPcp option.

5.3.4 PqStandardPresentation (interactive)

▷ PqStandardPresentation(*[i]: options*) (function)
▷ StandardPresentation(*[i]: options*) (function)

return, for the *i*th or default interactive ANUPQ process, the *p*-quotient of the group *F* of the process, specified by *options*, as an *fp group* which has a standard presentation. Here *options* is a selection of the options from the following list (see Chapter 'ANUPQ Options' for detailed descriptions); this list is the same as for the non-interactive version of PqStandardPresentation except for the omission of options SetupFile and PqWorkspace (see PqStandardPresentation (4.2.1)).

- Prime := *p*
- pQuotient := *Q*
- ClassBound := *n*
- Exponent := *n*
- Metabelian
- GroupName := *name*
- OutputLevel := *n*
- StandardPresentationFile := *filename*

Unless *F* is a pc *p*-group, or the option Prime has been passed to a previous interactive function for the process to compute a *p*-quotient for *F*, the user *must* supply either the option Prime or the option pQuotient (if both Prime and pQuotient are supplied, the prime *p* is determined by applying PrimePGroup (see PrimePGroup (**Reference: PrimePGroup**) in the Reference Manual) to the value of pQuotient).

Taking one of the examples for the non-interactive version of StandardPresentation (see StandardPresentation (4.2.1)) that required two separate calls to the pq program, we now show how it can be done by setting up a dialogue with just the one pq process, using the interactive version of StandardPresentation:

Example

```
gap> F4 := FreeGroup( "a", "b", "c", "d" );;
gap> a := F4.1;; b := F4.2;; c := F4.3;; d := F4.4;;
gap> G4 := F4 / [ b^4, b^2 / Comm(Comm (b, a), a), d^16,
>                 a^16 / (c * d), b^8 / (d * c^4) ];
<fp group on the generators [ a, b, c, d ]>
gap> SetInfoLevel(InfoANUPQ, 1); #Only essential Info please
gap> PqStart(G4); #Start a new interactive process for a new group
2
gap> K := Pq( 2 : Prime := 2, ClassBound := 1 ); #'pq' process no. is 2
<pc group of size 4 with 2 generators>
gap> StandardPresentation( 2 : pQuotient := K, ClassBound := 14 );
<fp group with 53 generators>
```

Notes

In contrast to the function Pq (see Pq (4.1.1)) which returns a pc group, PqStandardPresentation or StandardPresentation returns an fp group. This is because the output is mainly used for isomorphism testing for which an fp group is enough. However, the presentation is a polycyclic presentation and if you need to do any further computation with this group (e.g. to find the order) you can use the function PcGroupFpGroup (see PcGroupFpGroup (**Reference: PcGroupFpGroup**) in the GAP Reference Manual) to form a pc group.

If the user does not supply a p-quotient Q via the pQuotient option, and the prime p is either supplied, stored, or F is a pc p-group, then a p-quotient Q is computed. (The value of the prime p is stored if passed initially to PqStart or to a subsequent interactive process.) Note that a stored value for pQuotient (from a prior call to Pq) does *not* have precedence over a value for the prime p. If the user does supply a p-quotient Q via the pQuotient option, the package AutPGrp is called to compute the automorphism group of Q; an error will occur that asks the user to install the package AutPGrp if the automorphism group cannot be computed.

If any of the interactive functions PqStandardPresentation, StandardPresentation, EpimorphismPqStandardPresentation or EpimorphismStandardPresentation has been called previously for an interactive process, a subsequent call to any of these functions for the same process returns the previously computed value. Note that all these functions compute both an epimorphism and an fp group and store the results in the SPepi and SP fields of the data record associated with the process. See the example for the interactive EpimorphismStandardPresentation (EpimorphismStandardPresentation (5.3.5)).

The attributes and property NuclearRank, MultiplicatorRank and IsCapable are set for the group returned by PqStandardPresentation or StandardPresentation (see Section 'Attributes and a Property for fp and pc p-groups').

5.3.5 EpimorphismPqStandardPresentation (interactive)

▷ EpimorphismPqStandardPresentation(*[i]: options*) (function)
▷ EpimorphismStandardPresentation(*[i]: options*) (method)

Each of the above functions accepts the same arguments and options as the interactive form of StandardPresentation (see StandardPresentation (5.3.4)) and returns an epimorphism from the fp or pc group F of the ith or default interactive ANUPQ process onto the finitely presented group given by a standard presentation, i.e. if S is the standard presentation computed for the p-quotient of F by StandardPresentation then EpimorphismStandardPresentation returns the epimorphism from F to the group with presentation S. The group F must have been given (as first argument) to PqStart to start the interactive ANUPQ process (see PqStart (5.1.1)).

Taking our earlier non-interactive example (see EpimorphismPqStandardPresentation (4.2.2)) and modifying it a little, we illustrate, as for the interactive StandardPresentation (see StandardPresentation (5.3.4)), how something that required two separate calls to the pq program can now be achieved with a dialogue with just one pq process. Also, observe that calls to one of the standard presentation functions (as mentioned in the notes of StandardPresentation (5.3.4)) computes and stores both an fp group with a standard presentation and an epimorphism; subsequent calls to a standard presentation function for the same process simply return the appropriate stored value.

Example

```
gap> F := FreeGroup(6, "F");;
gap> x := F.1;; y := F.2;; z := F.3;; w := F.4;; a := F.5;; b := F.6;;
```

```
gap> R := [x^3 / w, y^3 / w * a^2 * b^2, w^3 / b,
>          Comm (y, x) / z, Comm (z, x), Comm (z, y) / a, z^3 ];
[ F1^3*F4^-1, F2^3*F4^-1*F5^2*F6^2, F4^3*F6^-1, F2^-1*F1^-1*F2*F1*F3^-1,
  F3^-1*F1^-1*F3*F1, F3^-1*F2^-1*F3*F2*F5^-1, F3^3 ]
gap> Q := F / R;
<fp group on the generators [ F1, F2, F3, F4, F5, F6 ]>
gap> PqStart( Q );
3
gap> G := Pq( 3 : Prime := 3, ClassBound := 3 );
<pc group of size 729 with 6 generators>
gap> lev := InfoLevel(InfoANUPQ);; # Save current InfoANUPQ level
gap> SetInfoLevel(InfoANUPQ, 2); # To see computation times
gap> # It is not necessary to pass the 'Prime' option to
gap> # 'EpimorphismStandardPresentation' since it was previously
gap> # passed to 'Pq':
gap> phi := EpimorphismStandardPresentation( 3 : ClassBound := 3 );
#I  Class 1 3-quotient and its 3-covering group computed in 0.00 seconds
#I  Order of GL subgroup is 48
#I  No. of soluble autos is 0
#I    dim U = 1  dim N = 3  dim M = 3
#I    nice stabilizer with perm rep
#I  Computing standard presentation for class 2 took 0.00 seconds
#I  Computing standard presentation for class 3 took 0.01 seconds
[ F1, F2, F3, F4, F5, F6 ] -> [ f1*f2^2*f3*f4^2*f5^2, f1*f2*f3*f5, f3^2,
  f4*f6^2, f5, f6 ]
gap> # Image of phi should be isomorphic to G ...
gap> # let's check the order is correct:
gap> Size( Image(phi) );
729
gap> # 'StandardPresentation' and 'EpimorphismStandardPresentation'
gap> # behave like attributes, so no computation is done when
gap> # either is called again for the same process ...
gap> StandardPresentation( 3 : ClassBound := 3 );
<fp group of size 729 on the generators [ f1, f2, f3, f4, f5, f6 ]>
gap> # No timing data was Info-ed since no computation was done
gap> SetInfoLevel(InfoANUPQ, lev); # Restore previous InfoANUPQ level
```

A very similar (essential details are the same) example to the above may be executed live, by typing: `PqExample( "EpimorphismStandardPresentation-i" );`.

Note: The notes for `PqStandardPresentation` or `StandardPresentation` (see `PqStandardPresentation` (5.3.4)) apply also to `EpimorphismPqStandardPresentation` or `EpimorphismStandardPresentation` except that their return value is an *epimorphism onto* an fp group, i.e. one should interpret the phrase "returns an fp group" as "returns an epimorphism onto an fp group" etc.

5.3.6 PqDescendants (interactive)

▷ `PqDescendants(`*i: options*`)` (function)

▷ `PqDescendants(:` *options*`)` (function)

return for the pc group *G* of the *i*th or default interactive ANUPQ process, which must be of prime power order with a confluent pc presentation (see IsConfluent (**Reference: IsConfluent for pc groups**) in the GAP Reference Manual), a list of descendants (pc groups) of *G*. The group *G* is usually given as first argument to PqStart when starting the interactive ANUPQ process (see PqStart (5.1.1)). Alternatively, one may initiate the process with an fp group, use Pq interactively (see Pq (5.3.1)) to create a pc group and use PqSetPQuotientToGroup (see PqSetPQuotientToGroup (5.3.7)), which involves *no* computation, to set the pc group returned by Pq as the group of the process. Note that repeating a call to PqDescendants for the same interactive ANUPQ process simply returns the list of descendants originally calculated; a warning is emitted at InfoANUPQ level 1 reminding you of this should you do this.

After the colon, *options* a selection of the options listed for the non-interactive PqDescendants function (see PqDescendants (4.4.1)), should be given, separated by commas like record components (see Section **Reference: Function Call With Options** in the GAP Reference Manual), except that the options SetupFile or PqWorkspace are ignored by the interactive PqDescendants, i.e. the following options are recognised by the interactive PqDescendants function:

- ClassBound := *n*
- Relators := *rels*
- OrderBound := *n*
- StepSize := *n*, StepSize := *list*
- RankInitialSegmentSubgroups := *n*
- SpaceEfficient
- CapableDescendants
- AllDescendants := false
- Exponent := *n*
- Metabelian
- GroupName := *name*
- SubList := *sub*
- BasicAlgorithm
- CustomiseOutput := *rec*

Notes: The function PqDescendants uses the automorphism group of *G* which it computes via the package AutPGrp if the automorphism group of *G* is not already present. If AutPGrp is not installed an error may be raised. If the automorphism group of *G* is insoluble the pq program will call GAP together with the AutPGrp package for certain orbit-stabilizer calculations.

The attributes and property NuclearRank, MultiplicatorRank and IsCapable are set for each group of the list returned by PqDescendants (see Section 'Attributes and a Property for fp and pc p-groups').

Let us now repeat the examples previously given for the non-interactive PqDescendants, but this time with the interactive version of PqDescendants:

Example

```
gap> F := FreeGroup( "a", "b" );; a := F.1;; b := F.2;;
gap> G := PcGroupFpGroup( F / [ a^2, b^2, Comm(b, a) ] );
<pc group of size 4 with 2 generators>
gap> PqStart(G); #This will now be the 4th interactive process running
4
gap> des := PqDescendants( 4 : OrderBound := 6, ClassBound := 5 );;
gap> Length(des);
83
gap> List(des, Size);
[ 8, 8, 8, 16, 16, 16, 32, 16, 16, 16, 16, 16, 32, 32, 64, 64, 32, 32, 32,
  32, 32, 32, 32, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 32, 32, 32, 32,
  64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 32, 32, 32, 32, 32, 64, 64, 64,
  64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
  64, 64, 64, 64, 64, 64, 64 ]
gap> List(des, d -> Length( PCentralSeries( d, 2 ) ) - 1 );
[ 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
  3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4,
  4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
  4, 4, 4, 5, 5, 5, 5, 5 ]
```

In the second example we compute all capable descendants of order 27 of the elementary abelian group of order 9.

Example

```
gap> F := FreeGroup( 2, "g" );;
gap> G := PcGroupFpGroup( F / [ F.1^3, F.2^3, Comm(F.1, F.2) ] );
<pc group of size 9 with 2 generators>
gap> PqStart(G); #This will now be the 5th interactive process running
5
gap> des := PqDescendants( 5 : OrderBound := 3, ClassBound := 2,
>                              CapableDescendants );
[ <pc group of size 27 with 3 generators>,
  <pc group of size 27 with 3 generators> ]
gap> List(des, d -> Length( PCentralSeries( d, 3 ) ) - 1 );
[ 2, 2 ]
gap> # For comparison let us now compute all descendants
gap> # (using the non-interactive Pq function)
gap> PqDescendants( G : OrderBound := 3, ClassBound := 2);
[ <pc group of size 27 with 3 generators>,
  <pc group of size 27 with 3 generators>,
  <pc group of size 27 with 3 generators> ]
```

In the third example, we compute all capable descendants of the elementary abelian group of order 5^2 which have exponent-5 class at most 3, exponent 5, and are metabelian.

Example

```
gap> F := FreeGroup( 2, "g" );;
gap> G := PcGroupFpGroup( F / [ F.1^5, F.2^5, Comm(F.2, F.1) ] );
<pc group of size 25 with 2 generators>
gap> PqStart(G); #This will now be the 6th interactive process running
6
gap> des := PqDescendants( 6 : Metabelian, ClassBound := 3,
>                              Exponent := 5, CapableDescendants );
```

```
  [ <pc group of size 125 with 3 generators>,
    <pc group of size 625 with 4 generators>,
    <pc group of size 3125 with 5 generators> ]
gap> List(des, d -> Length( PCentralSeries( d, 5 ) ) - 1 );
[ 2, 3, 3 ]
gap> List(des, d -> Length( DerivedSeries( d ) ) );
[ 3, 3, 3 ]
gap> List(des, d -> Maximum( List( d, Order ) ) );
[ 5, 5, 5 ]
```

5.3.7 PqSetPQuotientToGroup

▷ PqSetPQuotientToGroup(*i*) (function)

▷ PqSetPQuotientToGroup() (function)

for the *i*th or default interactive ANUPQ process, set the p-quotient previously computed by the interactive Pq function (see Pq (5.3.1)) to be the group of the process. This function is supplied to enable the computation of descendants of a p-quotient that is already known to the pq program, via the interactive PqDescendants function (see PqDescendants (5.3.6)), thus avoiding the need to re-submit it and have the pq program recompute it.

Note: See the function PqPGSetDescendantToPcp (PqPGSetDescendantToPcp (5.9.4)) for a mechanism to make (the p-cover of) a particular descendants the current group of the process.

The following example of the usage of PqSetPQuotientToGroup, which is essentially equivalent to what is obtained by running PqExample("PqDescendants-1-i");, redoes the first example of PqDescendants (5.3.6) (which computes the descendants of the Klein four group).

Example

```
gap> F := FreeGroup( "a", "b" );
<free group on the generators [ a, b ]>
gap> procId := PqStart( F : Prime := 2 );
7
gap> Pq( procId : ClassBound := 1 );
<pc group of size 4 with 2 generators>
gap> PqSetPQuotientToGroup( procId );
gap> des := PqDescendants( procId : OrderBound := 6, ClassBound := 5 );;
gap> Length(des);
83
gap> List(des, Size);
[ 8, 8, 8, 16, 16, 16, 32, 16, 16, 16, 16, 16, 32, 32, 64, 64, 32, 32, 32,
  32, 32, 32, 32, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 32, 32, 32, 32,
  64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 32, 32, 32, 32, 32, 64, 64, 64,
  64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
  64, 64, 64, 64, 64, 64, 64 ]
gap> List(des, d -> Length( PCentralSeries( d, 2 ) ) - 1 );
[ 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
  3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4,
  4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
  4, 4, 4, 5, 5, 5, 5, 5 ]
```

5.4 Low-level Interactive ANUPQ functions based on menu items of the pq program

The pq program has 5 menus, the details of which the reader will not normally need to know, but if she wishes to know the details they may be found in the standalone manual: `guide.dvi`. Both `guide.dvi` and the pq program refer to the items of these 5 menus as "options", which do *not* correspond in any way to the options used by any of the GAP functions that interface with the pq program.

Warning: The commands provided in this section are intended to provide something like the interactive functionality one has when running the standalone, from within GAP. The pq standalone (in particular, its "advanced" menus) assumes some expertise of the user; doing the "wrong" thing can cause the program to crash. While a number of safeguards have been provided in the GAP interface to the pq program, these are *not* foolproof, and the user should exercise care and ensure pre-requisites of the various commands are met.

5.5 General commands

The following commands either use a menu item from whatever menu is "current" for the pq program, or have general application and are not associated with just one menu item of the pq program.

5.5.1 PqNrPcGenerators

▷ `PqNrPcGenerators(`*i*`)` (function)
▷ `PqNrPcGenerators()` (function)

for the *i*th or default interactive ANUPQ process, return the number of pc generators of the lower exponent p-class quotient of the group currently determined by the process. This also applies if the pc presentation is not consistent.

5.5.2 PqFactoredOrder

▷ `PqFactoredOrder(`*i*`)` (function)
▷ `PqFactoredOrder()` (function)

for the *i*th or default interactive ANUPQ process, return an integer pair `[`*p*`, `*n*`]` where *p* is a prime and *n* is the number of pc generators (see `PqNrPcGenerators` (5.5.1)) in the pc presentation of the quotient group currently determined by the process. If this presentation is consistent, then p^n is the order of the quotient group. Otherwise (if tails have been added but the necessary consistency checks, relation collections, exponent law checks and redundant generator eliminations have not yet been done), p^n is an upper bound for the order of the group.

5.5.3 PqOrder

▷ `PqOrder(`*i*`)` (function)
▷ `PqOrder()` (function)

for the *i*th or default interactive ANUPQ process, return p^n where `[`*p*`, `*n*`]` is the pair as returned by `PqFactoredOrder` (see `PqFactoredOrder` (5.5.2)).

5.5.4 PqPClass

▷ `PqPClass(i)` (function)
▷ `PqPClass()` (function)

for the *i*th or default interactive ANUPQ process, return the lower exponent p-class of the quotient group currently determined by the process.

5.5.5 PqWeight

▷ `PqWeight(i, j)` (function)
▷ `PqWeight(j)` (function)

for the *i*th or default interactive ANUPQ process, return the weight of the *j*th pc generator of the lower exponent p-class quotient of the group currently determined by the process, or `fail` if there is no such numbered pc generator.

5.5.6 PqCurrentGroup

▷ `PqCurrentGroup(i)` (function)
▷ `PqCurrentGroup()` (function)

for the *i*th or default interactive ANUPQ process, return the group whose pc presentation is determined by the process as a GAP pc group (either a lower exponent p-class quotient of the start group or the p-cover of such a quotient).

Notes: See Section 'Attributes and a Property for fp and pc p-groups' for the attributes and property `NuclearRank`, `MultiplicatorRank` and `IsCapable` which may be applied to the group returned by `PqCurrentGroup`.

5.5.7 PqDisplayPcPresentation

▷ `PqDisplayPcPresentation(i:  [OutputLevel := lev])` (function)
▷ `PqDisplayPcPresentation(:  [OutputLevel := lev])` (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to display the pc presentation of the lower exponent p-class quotient of the group currently determined by the process.

Except if the last command communicating with the pq program was a p-group generation command (for which there is only a verbose output level), to set the amount of information this command displays you may wish to call `PqSetOutputLevel` first (see `PqSetOutputLevel` (5.5.8)), or equivalently pass the option `OutputLevel` (see 6.2).

Note: For those familiar with the pq program, `PqDisplayPcPresentation` performs menu item 4 of the current menu of the pq program.

5.5.8 PqSetOutputLevel

▷ `PqSetOutputLevel(i, lev)` (function)
▷ `PqSetOutputLevel(lev)` (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to set the output level of the pq program to *lev*.

Note: For those familiar with the pq program, PqSetOutputLevel performs menu item 5 of the main (or advanced) p-Quotient menu, or the Standard Presentation menu.

5.5.9 PqEvaluateIdentities

▷ PqEvaluateIdentities(*i: [Identities := funcs]*) (function)
▷ PqEvaluateIdentities(*: [Identities := funcs]*) (function)

for the *i*th or default interactive ANUPQ process, invoke the evaluation of identities defined by the Identities option, and eliminate any redundant pc generators formed. Since a previous value of Identities is saved in the data record of the process, it is unnecessary to pass the Identities if set previously.

Note: This function is mainly implemented at the GAP level. It does not correspond to a menu item of the pq program.

5.6 Commands from the Main p-Quotient menu

5.6.1 PqPcPresentation

▷ PqPcPresentation(*i: options*) (function)
▷ PqPcPresentation(*: options*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to compute the pc presentation of the quotient (determined by *options*) of the group of the process, which for process *i* is stored as ANUPQData.io[i].group.

The possible *options* are the same as for the interactive Pq (see Pq (5.3.1)) function, except for RedoPcp (which, in any case, would be superfluous), namely: Prime, ClassBound, Exponent, Relators, GroupName, Metabelian, Identities and OutputLevel (see Chapter 'ANUPQ Options' for a detailed description for these options). The option Prime is required unless already provided to PqStart.

Notes

The pc presentation is held by the pq program. In contrast to Pq (see Pq (5.3.1)), no GAP pc group is returned; see PqCurrentGroup (PqCurrentGroup (5.5.6)) if you need the corresponding GAP pc group.

PqPcPresentation(*i: options*); is roughly equivalent to the following sequence of low-level commands:

```
PqPcPresentation(i: opts); #class 1 call
for c in [2 .. class] do
    PqNextClass(i);
od;
```

where *opts* is *options* except with the ClassBound option set to 1, and *class* is either the maximum class of a p-quotient of the group of the process or the user-supplied value of the option ClassBound (whichever is smaller). If the Identities option has been set, both the first

PqPcPresentation class 1 call and the PqNextClass calls invoke PqEvaluateIdentities(*i*); as their final step.

For those familiar with the pq program, PqPcPresentation performs menu item 1 of the main p-Quotient menu.

5.6.2 PqSavePcPresentation

▷ PqSavePcPresentation(*i, filename*) (function)
▷ PqSavePcPresentation(*filename*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to save the pc presentation previously computed for the quotient of the group of that process to the file with name *filename*. If the first character of the string *filename* is not /, *filename* is assumed to be the path of a writable file relative to the directory in which GAP was started. A saved file may be restored by PqRestorePcPresentation (see PqRestorePcPresentation (5.6.3)).

Note: For those familiar with the pq program, PqSavePcPresentation performs menu item 2 of the main p-Quotient menu.

5.6.3 PqRestorePcPresentation

▷ PqRestorePcPresentation(*i, filename*) (function)
▷ PqRestorePcPresentation(*filename*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to restore the pc presentation previously saved to *filename*, by PqSavePcPresentation (see PqSavePcPresentation (5.6.2)). If the first character of the string *filename* is not /, *filename* is assumed to be the path of a readable file relative to the directory in which GAP was started.

Note: For those familiar with the pq program, PqRestorePcPresentation performs menu item 3 of the main p-Quotient menu.

5.6.4 PqNextClass

▷ PqNextClass(*i: [QueueFactor]*) (function)
▷ PqNextClass(*: [QueueFactor]*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to calculate the next class of ANUPQData.io[*i*].group.

PqNextClass accepts the option QueueFactor (see also 6.2) which should be a positive integer if automorphisms have been previously supplied. If the pq program requires a queue factor and none is supplied via the option QueueFactor a default of 15 is taken.

Notes

The single command: PqNextClass(*i*); is equivalent to executing

```
PqComputePCover(i);
PqCollectDefiningRelations(i);
PqDoExponentChecks(i);
PqEliminateRedundantGenerators(i);
```

If the Identities option is set the PqEliminateRedundantGenerators(*i*); step is essentially replaced by PqEvaluateIdentities(*i*); (which invokes its own elimination of redundant generators).

For those familiar with the pq program, PqNextClass performs menu item 6 of the main p-Quotient menu.

5.6.5 PqComputePCover

▷ PqComputePCover(*i*) (function)
▷ PqComputePCover() (function)

for the *i*th or default interactive ANUPQ processi, directi, the pq program to compute the p-covering group of ANUPQData.io[*i*].group. In contrast to the function PqPCover (see PqPCover (4.1.3)), this function does not return a GAP pc group.

Notes

The single command: PqComputePCover(*i*); is equivalent to executing

```
PqSetupTablesForNextClass(i);
PqTails(i, 0);
PqDoConsistencyChecks(i, 0, 0);
PqEliminateRedundantGenerators(i);
```

For those familiar with the pq program, PqComputePCover performs menu item 7 of the main p-Quotient menu.

5.7 Commands from the Advanced p-Quotient menu

5.7.1 PqCollect

▷ PqCollect(*i*, *word*) (function)
▷ PqCollect(*word*) (function)

for the *i*th or default interactive ANUPQ process, instruct the pq program to do a collection on *word*, a word in the current pc generators (the form of *word* required is described below). PqCollect returns the resulting word of the collection as a list of generator number, exponent pairs (the same form as the second allowed input form of *word*; see below).

The argument *word* may be input in either of the following ways:

1. *word* may be a string, where the *i*th pc generator is represented by x*i*, e.g. "x3*x2^2*x1". This way is quite versatile as parentheses and left-normed commutators -- using square brackets, in the same way as PqGAPRelators (see PqGAPRelators (3.4.2)) -- are permitted; *word* is checked for correct syntax via PqParseWord (see PqParseWord (3.4.3)).

2. Otherwise, *word* must be a list of generator number, exponent pairs of integers, i.e. each pair represents a "syllable" so that [[3, 1], [2, 2], [1, 1]] represents the same word as that of the example given for the first allowed form of *word*.

Note: For those familiar with the pq program, PqCollect performs menu item 1 of the Advanced p-Quotient menu.

5.7.2 PqSolveEquation

▷ `PqSolveEquation(i, a, b)` (function)
▷ `PqSolveEquation(a, b)` (function)

for the `i`th or default interactive ANUPQ process, direct the pq program to solve $a * x = b$ for x, where `a` and `b` are words in the pc generators. For the representation of these words see the description of the function `PqCollect` (`PqCollect` (5.7.1)).

Note: For those familiar with the pq program, `PqSolveEquation` performs menu item 2 of the Advanced p-Quotient menu.

5.7.3 PqCommutator

▷ `PqCommutator(i, words, pow)` (function)
▷ `PqCommutator(words, pow)` (function)

for the `i`th or default interactive ANUPQ process, instruct the pq program to compute the left normed commutator of the list *`words`* of words in the current pc generators raised to the integer power *`pow`*, and return the resulting word as a list of generator number, exponent pairs. The form required for each word of *`words`* is the same as that required for the *`word`* argument of `PqCollect` (see `PqCollect` (5.7.1)). The form of the output word is also the same as for `PqCollect`.

Note: For those familiar with the pq program, `PqCommutator` performs menu item 3 of the Advanced p-Quotient menu.

5.7.4 PqSetupTablesForNextClass

▷ `PqSetupTablesForNextClass(i)` (function)
▷ `PqSetupTablesForNextClass()` (function)

for the `i`th or default interactive ANUPQ process, direct the pq program to set up tables for the next class. As as side-effect, after `PqSetupTablesForNextClass(i)` the value returned by `PqPClass(i)` will be one more than it was previously.

Note: For those familiar with the pq program, `PqSetupTablesForNextClass` performs menu item 6 of the Advanced p-Quotient menu.

5.7.5 PqTails

▷ `PqTails(i, weight)` (function)
▷ `PqTails(weight)` (function)

for the `i`th or default interactive ANUPQ process, direct the pq program to compute and add tails of weight *`weight`* if *`weight`* is in the integer range `[2 .. PqPClass(i)]` (assuming `i` is the number of the process, even in the default case) or for all weights if *`weight`* `= 0`.

If *`weight`* is non-zero, then tails that introduce new generators for only weight *`weight`* are computed and added, and in this case and if *`weight`* `< PqPClass(i)`, it is assumed that the tails that introduce new generators for each weight from `PqPClass(i)` down to weight *`weight`* `+ 1` have already been added. You may wish to call `PqSetMetabelian` (see `PqSetMetabelian` (5.7.16)) prior to calling `PqTails`.

Notes

For its use in the context of finding the next class see PqNextClass (5.6.4); in particular, a call to PqSetupTablesForNextClass (see PqSetupTablesForNextClass (5.7.4)) needs to have been made prior to calling PqTails.

The single command: PqTails(*i*, *weight*); is equivalent to

```
PqComputeTails(i, weight);
PqAddTails(i, weight);
```

For those familiar with the pq program, PqTails uses menu item 7 of the Advanced p-Quotient menu.

5.7.6 PqComputeTails

▷ PqComputeTails(*i*, *weight*) (function)
▷ PqComputeTails(*weight*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to compute tails of weight *weight* if *weight* is in the integer range [2 .. PqPClass(*i*)] (assuming *i* is the number of the process, even in the default case) or for all weights if *weight* = 0. See PqTails (PqTails (5.7.5)) for more details.

Note: For those familiar with the pq program, PqComputeTails uses menu item 7 of the Advanced p-Quotient menu.

5.7.7 PqAddTails

▷ PqAddTails(*i*, *weight*) (function)
▷ PqAddTails(*weight*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to add the tails of weight *weight*, previously computed by PqComputeTails (see PqComputeTails (5.7.6)), if *weight* is in the integer range [2 .. PqPClass(*i*)] (assuming *i* is the number of the process, even in the default case) or for all weights if *weight* = 0. See PqTails (PqTails (5.7.5)) for more details.

Note: For those familiar with the pq program, PqAddTails uses menu item 7 of the Advanced p-Quotient menu.

5.7.8 PqDoConsistencyChecks

▷ PqDoConsistencyChecks(*i*, *weight*, *type*) (function)
▷ PqDoConsistencyChecks(*weight*, *type*) (function)

for the *i*th or default interactive ANUPQ process, do consistency checks for weight *weight* if *weight* is in the integer range [3 .. PqPClass(*i*)] (assuming *i* is the number of the process) or for all weights if *weight* = 0, and for type *type* if *type* is in the range [1, 2, 3] (see below) or for all types if *type* = 0. (For its use in the context of finding the next class see PqNextClass (5.6.4).)

The *type* of a consistency check is defined as follows. PqDoConsistencyChecks(*i*, *weight*, *type*) for *weight* in [3 .. PqPClass(*i*)] and the given value of *type* invokes the equivalent of the following PqDoConsistencyCheck calls (see PqDoConsistencyCheck (5.7.17)):

type = 1:
PqDoConsistencyCheck(*i*, *a*, *a*, *a*) checks 2 * PqWeight(*i*, *a*) + 1 = *weight*, for pc generators of index *a*.

type = 2:
PqDoConsistencyCheck(*i*, *b*, *b*, *a*) checks for pc generators of indices *b*, *a* satisfyingx both *b* > *a* and PqWeight(*i*, *b*) + PqWeight(*i*, *a*) + 1 = *weight*.

type = 3:
PqDoConsistencyCheck(*i*, *c*, *b*, *a*) checks for pc generators of indices *c*, *b*, *a* satisfying *c* > *b* > *a* and the sum of the weights of these generators equals *weight*.

Notes

PqWeight(*i*, *j*) returns the weight of the *j*th pc generator, for process *i* (see PqWeight (5.5.5)).

It is assumed that tails for the given weight (or weights) have already been added (see PqTails (5.7.5)).

For those familiar with the pq program, PqDoConsistencyChecks performs menu item 8 of the Advanced p-Quotient menu.

5.7.9 PqCollectDefiningRelations

▷ PqCollectDefiningRelations(*i*) (function)
▷ PqCollectDefiningRelations() (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to collect the images of the defining relations of the original fp group of the process, with respect to the current pc presentation, in the context of finding the next class (see PqNextClass (5.6.4)). If the tails operation is not complete then the relations may be evaluated incorrectly.

Note: For those familiar with the pq program, PqCollectDefiningRelations performs menu item 9 of the Advanced p-Quotient menu.

5.7.10 PqCollectWordInDefiningGenerators

▷ PqCollectWordInDefiningGenerators(*i*, *word*) (function)
▷ PqCollectWordInDefiningGenerators(*word*) (function)

for the *i*th or default interactive ANUPQ process, take a user-defined word *word* in the defining generators of the original presentation of the fp or pc group of the process. Each generator is mapped into the current pc presentation, and the resulting word is collected with respect to the current pc presentation. The result of the collection is returned as a list of generator number, exponent pairs.

The *word* argument may be input in either of the two ways described for PqCollect (see PqCollect (5.7.1)).

Note: For those familiar with the pq program, PqCollectDefiningGenerators performs menu item 23 of the Advanced p-Quotient menu.

5.7.11 PqCommutatorDefiningGenerators

▷ `PqCommutatorDefiningGenerators(i, words, pow)` (function)
▷ `PqCommutatorDefiningGenerators(words, pow)` (function)

for the *i*th or default interactive ANUPQ process, take a list *words* of user-defined words in the defining generators of the original presentation of the fp or pc group of the process, and an integer power *pow*. Each generator is mapped into the current pc presentation. The list *words* is interpreted as a left-normed commutator which is then raised to *pow* and collected with respect to the current pc presentation. The result of the collection is returned as a list of generator number, exponent pairs.

Note For those familiar with the pq program, `PqCommutatorDefiningGenerators` performs menu item 24 of the Advanced p-Quotient menu.

5.7.12 PqDoExponentChecks

▷ `PqDoExponentChecks(i: [Bounds := list])` (function)
▷ `PqDoExponentChecks(: [Bounds := list])` (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to do exponent checks for weights (inclusively) between the bounds of `Bounds` or for all weights if `Bounds` is not given. The value *list* of `Bounds` (assuming the interactive process is numbered *i*) should be a list of two integers *low*, *high* satisfying $1 \leq low \leq high \leq$ `PqPClass(i)` (see `PqPClass` (5.5.4)). If no exponent law has been specified, no exponent checks are performed.

Note: For those familiar with the pq program, `PqDoExponentChecks` performs menu item 10 of the Advanced p-Quotient menu.

5.7.13 PqEliminateRedundantGenerators

▷ `PqEliminateRedundantGenerators(i)` (function)
▷ `PqEliminateRedundantGenerators()` (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to eliminate redundant generators of the current p-quotient.

Note: For those familiar with the pq program, `PqEliminateRedundantGenerators` performs menu item 11 of the Advanced p-Quotient menu.

5.7.14 PqRevertToPreviousClass

▷ `PqRevertToPreviousClass(i)` (function)
▷ `PqRevertToPreviousClass()` (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to abandon the current class and revert to the previous class.

Note: For those familiar with the pq program, `PqRevertToPreviousClass` performs menu item 12 of the Advanced p-Quotient menu.

5.7.15 PqSetMaximalOccurrences

▷ `PqSetMaximalOccurrences(i, noccur)` (function)
▷ `PqSetMaximalOccurrences(noccur)` (function)

for the `i`th or default interactive ANUPQ process, direct the `pq` program to set maximal occurrences of the weight 1 generators in the definitions of pcp generators of the group of the process. This can be used to avoid the definition of generators of which one knows for theoretical reasons that they would be eliminated later on.

The argument `noccur` must be a list of non-negative integers of length the number of weight 1 generators (i.e. the rank of the class 1 p-quotient of the group of the process). An entry of `0` for a particular generator indicates that there is no limit on the number of occurrences for the generator.

Note: For those familiar with the `pq` program, `PqSetMaximalOccurrences` performs menu item 13 of the Advanced p-Quotient menu.

5.7.16 PqSetMetabelian

▷ `PqSetMetabelian(i)` (function)
▷ `PqSetMetabelian()` (function)

for the `i`th or default interactive ANUPQ process, direct the `pq` program to enforce metabelian-ness.

Note: For those familiar with the `pq` program, `PqSetMetabelian` performs menu item 14 of the Advanced p-Quotient menu.

5.7.17 PqDoConsistencyCheck

▷ `PqDoConsistencyCheck(i, c, b, a)` (function)
▷ `PqDoConsistencyCheck(c, b, a)` (function)
▷ `PqJacobi(i, c, b, a)` (function)
▷ `PqJacobi(c, b, a)` (function)

for the `i`th or default interactive ANUPQ process, direct the `pq` program to do the consistency check for the pc generators with indices `c`, `b`, `a` which should be non-increasing positive integers, i.e. $c \geq b \geq a$.

There are 3 types of consistency checks:

$$\begin{array}{lcll} (a^n)a & = & a(a^n) & \text{(Type 1)} \\ (b^n)a & = & b^{(n-1)}(ba), b(a^n) = (ba)a^{(n-1)} & \text{(Type 2)} \\ c(ba) & = & (cb)a & \text{(Type 3)} \end{array}$$

The reason some people talk about Jacobi relations instead of consistency checks becomes clear when one looks at the consistency check of type 3:

$$\begin{array}{rcl} c(ba) & = & ac[c,a]b[b,a] = acb[c,a][c,a,b][b,a] = \ldots \\ (cb)a & = & bc[c,b]a = ab[b,a]c[c,a][c,b][c,b,a] \\ & = & abc[b,a][b,a,c][c,a][c,b][c,b,a] = \ldots \end{array}$$

Each collection would normally carry on further. But one can see already that no other commutators of weight 3 will occur. After all terms of weight one and weight two have been moved to the left we end up with:

$$\begin{aligned} & abc[b,a][c,a][c,b][c,a,b]\dots \\ = \; & abc[b,a][c,a][c,b][c,b,a][b,a,c]\dots \end{aligned}$$

Modulo terms of weight 4 this is equivalent to

$$[c,a,b][b,c,a][a,b,c] = 1$$

which is the Jacobi identity.

See also `PqDoConsistencyChecks` (`PqDoConsistencyChecks` (5.7.8)).

Note: For those familiar with the `pq` program, `PqDoConsistencyCheck` and `PqJacobi` perform menu item 15 of the Advanced p-Quotient menu.

5.7.18 PqCompact

▷ `PqCompact(i)` (function)
▷ `PqCompact()` (function)

for the *i*th or default interactive ANUPQ process, direct the `pq` program to do a compaction of its work space. This function is safe to perform only at certain points in time.

Note: For those familiar with the `pq` program, `PqCompact` performs menu item 16 of the Advanced p-Quotient menu.

5.7.19 PqEchelonise

▷ `PqEchelonise(i)` (function)
▷ `PqEchelonise()` (function)

for the *i*th or default interactive ANUPQ process, direct the `pq` program to echelonise the word most recently collected by `PqCollect` or `PqCommutator` against the relations of the current pc presentation, and return the number of the generator made redundant or `fail` if no generator was made redundant. A call to `PqCollect` (see `PqCollect` (5.7.1)) or `PqCommutator` (see `PqCommutator` (5.7.3)) needs to be performed prior to using this command.

Note: For those familiar with the `pq` program, `PqEchelonise` performs menu item 17 of the Advanced p-Quotient menu.

5.7.20 PqSupplyAutomorphisms

▷ `PqSupplyAutomorphisms(i, mlist)` (function)
▷ `PqSupplyAutomorphisms(mlist)` (function)

for the *i*th or default interactive ANUPQ process, supply the automorphism data provided by the list *mlist* of matrices with non-negative integer coefficients. Each matrix in *mlist* describes one automorphism in the following way.

- The rows of each matrix correspond to the pc generators of weight one.

- Each row is the exponent vector of the image of the corresponding weight one generator under the respective automorphism.

Note: For those familiar with the pq program, PqSupplyAutomorphisms uses menu item 18 of the Advanced p-Quotient menu.

5.7.21 PqExtendAutomorphisms

▷ PqExtendAutomorphisms(*i*) (function)
▷ PqExtendAutomorphisms() (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to extend automorphisms of the p-quotient of the previous class to the p-quotient of the present class.

Note: For those familiar with the pq program, PqExtendAutomorphisms uses menu item 18 of the Advanced p-Quotient menu.

5.7.22 PqApplyAutomorphisms

▷ PqApplyAutomorphisms(*i, qfac*) (function)
▷ PqApplyAutomorphisms(*qfac*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to apply automorphisms; *qfac* is the queue factor e.g. 15.

Note: For those familiar with the pq program, PqCloseRelations performs menu item 19 of the Advanced p-Quotient menu.

5.7.23 PqDisplayStructure

▷ PqDisplayStructure(*i: [Bounds := list]*) (function)
▷ PqDisplayStructure(*: [Bounds := list]*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to display the structure for the pcp generators numbered (inclusively) between the bounds of Bounds or for all generators if Bounds is not given. The value *list* of Bounds (assuming the interactive process is numbered *i*) should be a list of two integers *low*, *high* satisfying $1 \leq low \leq high \leq$ PqNrPcGenerators(*i*) (see PqNrPcGenerators (5.5.1)). PqDisplayStructure also accepts the option OutputLevel (see 6.2).

Explanation of output

New generators are defined as commutators of previous generators and generators of class 1 or as p-th powers of generators that have themselves been defined as p-th powers. A generator is never defined as p-th power of a commutator.

Therefore, there are two cases: all the numbers on the righthand side are either the same or they differ. Below, gi refers to the *i*th defining generator.

- If the righthand side numbers are all the same, then the generator is a p-th power (of a p-th power of a p-th power, etc.). The number of repeated digits say how often a p-th power has to be taken.

In the following example, the generator number 31 is the eleventh power of generator 17 which in turn is an eleventh power and so on:

\begintt #I 31 is defined on 17^11 = 1 1 1 1 1 \endtt So generator 31 is obtained by taking the eleventh power of generator 1 five times.

- If the numbers are not all the same, the generator is defined by a commutator. If the first two generator numbers differ, the generator is defined as a left-normed commutator of the weight one generators, e.g.

  \begintt #I 19 is defined on [11, 1] = 2 1 1 1 1 \endtt Here, generator 19 is defined as the commutator of generator 11 and generator 1 which is the same as the left-normed commutator `[x2, x1, x1, x1, x1]`. One can check this by tracing back the definition of generator 11 until one gets to a generator of class 1.

- If the first two generator numbers are identical, then the left most component of the left-normed commutator is a p-th power, e.g.

  \begintt #I 25 is defined on [14, 1] = 1 1 2 1 1 \endtt

  In this example, generator 25 is defined as commutator of generator 14 and generator 1. The left-normed commutator is

  $$[(x1^{11})^{11}, x2, x1, x1]$$

  Again, this can be verified by tracing back the definitions.

Note: For those familiar with the pq program, `PqDisplayStructure` performs menu item 20 of the Advanced p-Quotient menu.

5.7.24 PqDisplayAutomorphisms

▷ `PqDisplayAutomorphisms(`*`i`*`:  [Bounds := list])` (function)
▷ `PqDisplayAutomorphisms(:  [Bounds := list])` (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to display the automorphism actions on the pcp generators numbered (inclusively) between the bounds of `Bounds` or for all generators if `Bounds` is not given. The value *list* of `Bounds` (assuming the interactive process is numbered *i*) should be a list of two integers *low*, *high* satisfying $1 \leq low \leq high \leq$ `PqNrPcGenerators(`*`i`*`)` (see `PqNrPcGenerators` (5.5.1)). `PqDisplayStructure` also accepts the option `OutputLevel` (see 6.2).

Note: For those familiar with the pq program, `PqDisplayAutomorphisms` performs menu item 21 of the Advanced p-Quotient menu.

5.7.25 PqWritePcPresentation

▷ `PqWritePcPresentation(`*`i, filename`*`)` (function)
▷ `PqWritePcPresentation(`*`filename`*`)` (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to write a pc presentation of a previously-computed quotient of the group of that process, to the file with name *filename*. Here the group of a process is the one given as first argument when `PqStart` was called to initiate

that process (for process *i* the group is stored as ANUPQData.io[*i*].group). If the first character of the string *filename* is not /, *filename* is assumed to be the path of a writable file relative to the directory in which GAP was started. If a pc presentation has not been previously computed by the pq program, then pq is called to compute it first, effectively invoking PqPcPresentation (see PqPcPresentation (5.6.1)).

Note: For those familiar with the pq program, PqPcWritePresentation performs menu item 25 of the Advanced p-Quotient menu.

5.8 Commands from the Standard Presentation menu

5.8.1 PqSPComputePcpAndPCover

▷ PqSPComputePcpAndPCover(*i: options*) (function)
▷ PqSPComputePcpAndPCover(*: options*) (function)

for the *i*th or default interactive ANUPQ process, directs the pq program to compute for the group of that process a pc presentation up to the p-quotient of maximum class or the value of the option ClassBound and the p-cover of that quotient, and sets up tabular information required for computation of a standard presentation. Here the group of a process is the one given as first argument when PqStart was called to initiate that process (for process *i* the group is stored as ANUPQData.io[*i*].group).

The possible *options* are Prime, ClassBound, Relators, Exponent, Metabelian and OutputLevel (see Chapter 'ANUPQ Options' for detailed descriptions of these options). The option Prime is normally determined via PrimePGroup, and so is not required unless the group doesn't know it's a p-group and HasPrimePGroup returns false.

Note: For those familiar with the pq program, PqSPComputePcpAndPCover performs option 1 of the Standard Presentation menu.

5.8.2 PqSPStandardPresentation

▷ PqSPStandardPresentation(*i[, mlist]: [options]*) (function)
▷ PqSPStandardPresentation(*[mlist]: [options]*) (function)

for the *i*th or default interactive ANUPQ process, inputs data given by *options* to compute a standard presentation for the group of that process. If argument *mlist* is given it is assumed to be the automorphism group data required. Otherwise it is assumed that a call to either Pq (see Pq (5.3.1)) or PqEpimorphism (see PqEpimorphism (5.3.2)) has generated a p-quotient and that GAP can compute its automorphism group from which the necessary automorphism group data can be derived. The group of the process is the one given as first argument when PqStart was called to initiate the process (for process *i* the group is stored as ANUPQData.io[*i*].group and the p-quotient if existent is stored as ANUPQData.io[*i*].pQuotient). If *mlist* is not given and a p-quotient of the group has not been previously computed a class 1 p-quotient is computed.

PqSPStandardPresentation accepts three options, all optional:

- ClassBound := *n*
- PcgsAutomorphisms

- `StandardPresentationFile := filename`

If `ClassBound` is omitted it defaults to 63.

Detailed descriptions of the above options may be found in Chapter 'ANUPQ Options'.

Note: For those familiar with the pq program, `PqSPPcPresentation` performs menu item 2 of the Standard Presentation menu.

5.8.3 PqSPSavePresentation

▷ `PqSPSavePresentation(i, filename)` (function)
▷ `PqSPSavePresentation(filename)` (function)

for the `i`th or default interactive ANUPQ process, directs the pq program to save the standard presentation previously computed for the group of that process to the file with name `filename`, where the group of a process is the one given as first argument when `PqStart` was called to initiate that process. If the first character of the string `filename` is not `/`, `filename` is assumed to be the path of a writable file relative to the directory in which GAP was started.

Note: For those familiar with the pq program, `PqSPSavePresentation` performs menu item 3 of the Standard Presentation menu.

5.8.4 PqSPCompareTwoFilePresentations

▷ `PqSPCompareTwoFilePresentations(i, f1, f2)` (function)
▷ `PqSPCompareTwoFilePresentations(f1, f2)` (function)

for the `i`th or default interactive ANUPQ process, direct the pq program to compare the presentations in the files with names `f1` and `f2` and returns `true` if they are identical and `false` otherwise. For each of the strings `f1` and `f2`, if the first character is not a `/` then it is assumed to be the path of a readable file relative to the directory in which GAP was started.

Notes

The presentations in files `f1` and `f2` must have been generated by the pq program but they do *not* need to be *standard* presentations. If If the presentations in files `f1` and `f2` *have* been generated by `PqSPStandardPresentation` (see `PqSPStandardPresentation` (5.8.2)) then a `false` response from `PqSPCompareTwoFilePresentations` says the groups defined by those presentations are *not* isomorphic.

For those familiar with the pq program, `PqSPCompareTwoFilePresentations` performs menu item 6 of the Standard Presentation menu.

5.8.5 PqSPIsomorphism

▷ `PqSPIsomorphism(i)` (function)
▷ `PqSPIsomorphism()` (function)

for the `i`th or default interactive ANUPQ process, direct the pq program to compute the isomorphism mapping from the p-group of the process to its standard presentation. This function provides a description only; for a GAP object, use `EpimorphismStandardPresentation` (see `EpimorphismStandardPresentation` (5.3.5)).

Note: For those familiar with the pq program, PqSPIsomorphism performs menu item 8 of the Standard Presentation menu.

5.9 Commands from the Main p-Group Generation menu

Note that the p-group generation commands can only be applied once the pq program has produced a pc presentation of some quotient group of the "group of the process".

5.9.1 PqPGSupplyAutomorphisms

▷ PqPGSupplyAutomorphisms(*i[, mlist]: options*) (function)
▷ PqPGSupplyAutomorphisms(*[mlist]: options*) (function)

for the *i*th or default interactive ANUPQ process, supply the pq program with the automorphism group data needed for the current quotient of the group of that process (for process *i* the group is stored as ANUPQData.io[*i*].group). For a description of the format of *mlist* see PqSupplyAutomorphisms (5.7.20). The options possible are NumberOfSolubleAutomorphisms and RelativeOrders. (Detailed descriptions of these options may be found in Chapter 'ANUPQ Options'.)

If *mlist* is omitted, the automorphism data is determined from the group of the process which must have been a p-group in pc presentation.

Note: For those familiar with the pq program, PqPGSupplyAutomorphisms performs menu item 1 of the main p-Group Generation menu.

5.9.2 PqPGExtendAutomorphisms

▷ PqPGExtendAutomorphisms(*i*) (function)
▷ PqPGExtendAutomorphisms() (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to compute the extensions of the automorphisms of the p-quotient of the previous class to the p-quotient of the current class. You may wish to set the InfoLevel of InfoANUPQ to 2 (or more) in order to see the output from the pq program (see InfoANUPQ (3.3.1)).

Note: For those familiar with the pq program, PqPGExtendAutomorphisms performs menu item 2 of the main or advanced p-Group Generation menu.

5.9.3 PqPGConstructDescendants

▷ PqPGConstructDescendants(*i: options*) (function)
▷ PqPGConstructDescendants(*: options*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to construct descendants prescribed by *options*, and return the number of descendants constructed (compare function PqDescendants (4.4.1) which returns the list of descendants). The options possible are ClassBound, OrderBound, StepSize, PcgsAutomorphisms, RankInitialSegmentSubgroups, SpaceEfficient, CapableDescendants, AllDescendants, Exponent, Metabelian,

BasicAlgorithm, CustomiseOutput. (Detailed descriptions of these options may be found in Chapter 'ANUPQ Options'.)

PqPGConstructDescendants requires that the pq program has previously computed a pc presentation and a p-cover for a p-quotient of some class of the group of the process.

Note: For those familiar with the pq program, PqPGConstructDescendants performs menu item 5 of the main p-Group Generation menu.

5.9.4 PqPGSetDescendantToPcp (with class)

▷ PqPGSetDescendantToPcp(*i, cls, n*) (function)
▷ PqPGSetDescendantToPcp(*cls, n*) (function)
▷ PqPGSetDescendantToPcp(*i: [Filename := name]*) (function)
▷ PqPGSetDescendantToPcp(*: [Filename := name]*) (function)
▷ PqPGRestoreDescendantFromFile(*i, cls, n*) (function)
▷ PqPGRestoreDescendantFromFile(*cls, n*) (function)
▷ PqPGRestoreDescendantFromFile(*i: [Filename := name]*) (function)
▷ PqPGRestoreDescendantFromFile(*: [Filename := name]*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to restore group *n* of class *cls* from a temporary file, where *cls* and *n* are positive integers, or the group stored in *name*. PqPGSetDescendantToPcp and PqPGRestoreDescendantFromFile are synonyms; they make sense only after a prior call to construct descendants by say PqPGConstructDescendants (see PqPGConstructDescendants (5.9.3)) or the interactive PqDescendants (see PqDescendants (5.3.6)). In the Filename option forms, the option defaults to the last filename in which a presentation was stored by the pq program.

Notes

Since the PqPGSetDescendantToPcp and PqPGRestoreDescendantFromFile are intended to be used in calculation of further descendants the pq program computes the p-cover of the restored descendant. Hence, PqCurrentGroup used immediately after one of these commands returns the p-cover of the restored descendant rather than the descendant itself.

For those familiar with the pq program, PqPGSetDescendantToPcp and PqPGRestoreDescendantFromFile perform menu item 3 of the main or advanced p-Group Generation menu.

5.10 Commands from the Advanced p-Group Generation menu

The functions below perform the component algorithms of PqPGConstructDescendants (see PqPGConstructDescendants (5.9.3)). You can get some idea of their usage by trying PqExample("Nott-APG-Rel-i");. You can get some idea of the breakdown of PqPGConstructDescendants into these functions by comparing the previous output with PqExample("Nott-PG-Rel-i");.

These functions are intended for use only by "experts"; please contact the authors of the package if you genuinely have a need for them and need any amplified descriptions.

5.10.1 PqAPGDegree

▷ PqAPGDegree(*i*, *step*, *rank*: [*Exponent* := *n*]) (function)
▷ PqAPGDegree(*step*, *rank*: [*Exponent* := *n*]) (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to invoke menu item 6 of the Advanced p-Group Generation menu. Here the step-size *step* and the rank *rank* are positive integers and are the arguments required by the pq program. See 6.2 for the one recognised option Exponent.

5.10.2 PqAPGPermutations

▷ PqAPGPermutations(*i*: *options*) (function)
▷ PqAPGPermutations(: *options*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq program to perform menu item 7 of the Advanced p-Group Generation menu. Here the options *options* recognised are PcgsAutomorphisms, SpaceEfficient, PrintAutomorphisms and PrintPermutations (see Chapter 'ANUPQ Options' for details).

5.10.3 PqAPGOrbits

▷ PqAPGOrbits(*i*: *options*) (function)
▷ PqAPGOrbits(: *options*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq to perform menu item 8 of the Advanced p-Group Generation menu.

Here the options *options* recognised are PcgsAutomorphisms, SpaceEfficient and CustomiseOutput (see Chapter 'ANUPQ Options' for details). For the CustomiseOutput option only the setting of the orbit is recognised (all other fields if set are ignored).

5.10.4 PqAPGOrbitRepresentatives

▷ PqAPGOrbitRepresentatives(*i*: *options*) (function)
▷ PqAPGOrbitRepresentatives(: *options*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq to perform item 9 of the Advanced p-Group Generation menu.

The options *options* may be any selection of the following: PcgsAutomorphisms, SpaceEfficient, Exponent, Metabelian, CapableDescendants (or AllDescendants), CustomiseOutput (where only the group and autgroup fields are recognised) and Filename (see Chapter 'ANUPQ Options' for details). If Filename is omitted the reduced p-cover is written to the file "redPCover" in the temporary directory whose name is stored in ANUPQData.tmpdir.

5.10.5 PqAPGSingleStage

▷ PqAPGSingleStage(*i*: *options*) (function)
▷ PqAPGSingleStage(: *options*) (function)

for the *i*th or default interactive ANUPQ process, direct the pq to perform option 5 of the Advanced p-Group Generation menu.

The possible options are StepSize, PcgsAutomorphisms, RankInitialSegmentSubgroups, SpaceEfficient, CapableDescendants, AllDescendants, Exponent, Metabelian, BasicAlgorithm and CustomiseOutput. (Detailed descriptions of these options may be found in Chapter 'ANUPQ Options'.)

5.11 Primitive Interactive ANUPQ Process Read/Write Functions

For those familiar with using the pq program as a standalone we provide primitive read/write tools to communicate directly with an interactive ANUPQ process, started via PqStart. For the most part, it is up to the user to translate the output strings from pq program into a form useful in GAP.

5.11.1 PqRead

▷ PqRead(*i*) (function)
▷ PqRead() (function)

read a complete line of ANUPQ output, from the *i*th or default interactive ANUPQ process, if there is output to be read and returns fail otherwise. When successful, the line is returned as a string complete with trailing newline, colon, or question-mark character. Please note that it is possible to be "too quick" (i.e. the return can be fail purely because the output from ANUPQ is not there yet), but if PqRead finds any output at all, it waits for a complete line. PqRead also writes the line read via Info at InfoANUPQ level 2. It doesn't try to distinguish banner and menu output from other output of the pq program.

5.11.2 PqReadAll

▷ PqReadAll(*i*) (function)
▷ PqReadAll() (function)

read and return as many *complete* lines of ANUPQ output, from the *i*th or default interactive ANUPQ process, as there are to be read, *at the time of the call*, as a list of strings with any trailing newlines removed and returns the empty list otherwise. PqReadAll also writes each line read via Info at InfoANUPQ level 2. It doesn't try to distinguish banner and menu output from other output of the pq program. Whenever PqReadAll finds only a partial line, it waits for the complete line, thus increasing the probability that it has captured all the output to be had from ANUPQ.

5.11.3 PqReadUntil

▷ PqReadUntil(*i, IsMyLine*) (function)
▷ PqReadUntil(*IsMyLine*) (function)
▷ PqReadUntil(*i, IsMyLine, Modify*) (function)
▷ PqReadUntil(*IsMyLine, Modify*) (function)

read complete lines of ANUPQ output, from the *i*th or default interactive ANUPQ process, "chomps" them (i.e. removes any trailing newline character), emits them to `Info` at `InfoANUPQ` level 2 (without trying to distinguish banner and menu output from other output of the `pq` program), and applies the function *Modify* (where *Modify* is just the identity map/function for the first two forms) until a "chomped" line *line* for which *IsMyLine*`(` *Modify*`(`*line*`) )` is true. `PqReadUntil` returns the list of *Modify*-ed "chomped" lines read.

Notes: When provided by the user, *Modify* should be a function that accepts a single string argument.

IsMyLine should be a function that is able to accept the output of *Modify* (or take a single string argument when *Modify* is not provided) and should return a boolean.

If *IsMyLine*`(` *Modify*`(`*line*`) )` is never true, `PqReadUntil` will wait indefinitely.

5.11.4 PqWrite

▷ `PqWrite(`*i*`,` *string*`)` (function)
▷ `PqWrite(`*string*`)` (function)

write *string* to the *i*th or default interactive ANUPQ process; *string* must be in exactly the form the ANUPQ standalone expects. The command is echoed via `Info` at `InfoANUPQ` level 3 (with a "`ToPQ> `" prompt); i.e. do `SetInfoLevel(InfoANUPQ, 3);` to see what is transmitted to the `pq` program. `PqWrite` returns `true` if successful in writing to the stream of the interactive ANUPQ process, and `fail` otherwise.

Note: If `PqWrite` returns `fail` it means that the ANUPQ process has died.

Chapter 6

ANUPQ Options

6.1 Overview

In this chapter we describe in detail all the options used by functions of the ANUPQ package. Note that by "options" we mean GAP options that are passed to functions after the arguments and separated from the arguments by a colon as described in Chapter **Reference: Function Calls** in the Reference Manual. The user is strongly advised to read Section 'Hints and Warnings regarding the use of Options'.

6.1.1 AllANUPQoptions

▷ `AllANUPQoptions()` (function)

lists all the GAP options defined for functions of the ANUPQ package:

Example

```
gap> AllANUPQoptions();
[ "AllDescendants", "BasicAlgorithm", "Bounds", "CapableDescendants",
  "ClassBound", "CustomiseOutput", "Exponent", "Filename", "GroupName",
  "Identities", "Metabelian", "NumberOfSolubleAutomorphisms", "OrderBound",
  "OutputLevel", "PcgsAutomorphisms", "PqWorkspace", "Prime",
  "PrintAutomorphisms", "PrintPermutations", "QueueFactor",
  "RankInitialSegmentSubgroups", "RedoPcp", "RelativeOrders", "Relators",
  "SetupFile", "SpaceEfficient", "StandardPresentationFile", "StepSize",
  "SubList", "TreeDepth", "pQuotient" ]
```

The following global variable gives a partial breakdown of where the above options are used.

6.1.2 ANUPQoptions

▷ `ANUPQoptions` (global variable)

is a record of lists of names of admissible ANUPQ options, such that each field is either the name of a "key" ANUPQ function or `other` (for a miscellaneous list of functions) and the corresponding value is the list of option names that are admissible for the function (or miscellaneous list of functions).

Also, from within a GAP session, you may use GAP's help browser (see Chapter **Reference: The Help System** in the GAP Reference Manual); to find out about any particular ANUPQ option, simply type: "`?option option`", where *option* is one of the options listed above without any quotes, e.g.

Example

```
gap> ?option Prime
```

will display the sections in this manual that describe the `Prime` option. In fact the first 4 are for the functions that have `Prime` as an option and the last actually describes the option. So follow up by choosing

Example

```
gap> ?5
```

This is also the pattern for other options (the last section of the list always describes the option; the other sections are the functions with which the option may be used).

In the section following we describe in detail all ANUPQ options. To continue onto the next section on-line using GAP's help browser, type:

Example

```
gap> ?>
```

6.2 Detailed descriptions of ANUPQ Options

`Prime := p`
: Specifies that the p-quotient for the prime p should be computed.

`ClassBound := n`
: Specifies that the p-quotient to be computed has lower exponent-p class at most n. If this option is omitted a default of 63 (which is the maximum possible for the pq program) is taken, except for `PqDescendants` (see `PqDescendants` (4.4.1)) and in a special case of `PqPCover` (see `PqPCover` (4.1.3)). Let F be the argument (or start group of the process in the interactive case) for the function; then for `PqDescendants` the default is `PClassPGroup(F) + 1`, and for the special case of `PqPCover` the default is `PClassPGroup(F)`.

`pQuotient := Q`
: This option is only available for the standard presentation functions. It specifies that a p-quotient of the group argument of the function or group of the process is the pc p-group Q, where Q is of class *less than* the provided (or default) value of `ClassBound`. If `pQuotient` is provided, then the option `Prime` if also provided, is ignored; the prime p is discovered by computing `PrimePGroup(Q)`.

`Exponent := n`
: Specifies that the p-quotient to be computed has exponent n. For an interactive process, `Exponent` defaults to a previously supplied value for the process. Otherwise (and non-interactively), the default is 0, which means that no exponent law is enforced.

`Relators := rels`
: Specifies that the relators sent to the pq program should be *rels* instead of the relators of the argument group F (or start group in the interactive case) of the calling function; *rels* should be a list of *strings* in the string representations of the generators of F, and F must be an *fp group*

(even if the calling function accepts a pc group). This option provides a way of giving relators to the pq program, without having them pre-expanded by GAP, which can sometimes effect a performance loss of the order of 100 (see Section 'The Relators Option').

Notes

1. The pq program does not use / to indicate multiplication by an inverse and uses square brackets to represent (left normed) commutators. Also, even though the pq program accepts relations, all elements of *rels must* be in relator form, i.e. a relation of form *w1* = *w2* must be written as *w1**(*w2*)^-1 and then put in a pair of double-quotes to make it a string. See the example below.
2. To ensure there are no syntax errors in *rels*, each relator is parsed for validity via PqParseWord (see PqParseWord (3.4.3)). If they are ok, a message to say so is Info-ed at InfoANUPQ level 2.

Metabelian
: Specifies that the largest metabelian p-quotient subject to any other conditions specified by other options be constructed. By default this restriction is not enforced.

GroupName := *name*
: Specifies that the pq program should refer to the group by the name *name* (a string). If GroupName is not set and the group has been assigned a name via SetName (see **Reference: Name**) it is set as the name the pq program should use. Otherwise, the "generic" name "[grp]" is set as a default.

Identities := *funcs*
: Specifies that the pc presentation should satisfy the laws defined by each function in the list *funcs*. This option may be called by Pq, PqEpimorphism, or PqPCover (see Pq (4.1.1)). Each function in the list *funcs* must return a word in its arguments (there may be any number of arguments). Let *identity* be one such function in *funcs*. Then as each lower exponent p-class quotient is formed, instances *identity*$(w1,\ldots,wn)$ are added as relators to the pc presentation, where $w1,\ldots,wn$ are words in the pc generators of the quotient. At each class the class and number of pc generators is Info-ed at InfoANUPQ level 1, the number of instances is Info-ed at InfoANUPQ level 2, and the instances that are evaluated are Info-ed at InfoANUPQ level 3. As usual timing information is Info-ed at InfoANUPQ level 2; and details of the processing of each instance from the pq program (which is often quite *voluminous*) is Info-ed at InfoANUPQ level 3. Try the examples "B2-4-Id" and "11gp-3-Engel-Id" which demonstrate the usage of the Identities option; these are run using PqExample (see PqExample (3.4.4)). Take note of Note 1. below in relation to the example "B2-4-Id"; the companion example "B2-4" generates the same group using the Exponent option. These examples are discussed at length in Section 'The Identities Option and PqEvaluateIdentities Function'.

Notes

1. Setting the InfoANUPQ level to 3 or more when setting the Identities option may slow down the computation considerably, by overloading GAP with io operations.
2. The Identities option is implemented at the GAP level. An identity that is just an exponent law should be specified using the Exponent option (see option Exponent (6.2)), which is implemented at the C level and is highly optimised and so is much more efficient.

3. The number of instances of each identity tends to grow combinatorially with the class. So *care* should be exercised in using the `Identities` option, by including other restrictions, e.g. by using the `ClassBound` option (see `option ClassBound` (6.2)).

`OutputLevel := n`
: Specifies the level of "verbosity" of the information output by the ANU `pq` program when computing a pc presentation; *`n`* must be an integer in the range 0 to 3. `OutputLevel := 0` displays at most one line of output and is the default; `OutputLevel := 1` displays (usually) slightly more output and `OutputLevel`s of 2 and 3 are two levels of verbose output. To see these messages from the `pq` program, the `InfoANUPQ` level must be set to at least 1 (see `InfoANUPQ` (3.3.1)). See Section 'Hints and Warnings regarding the use of Options' for an example of how `OutputLevel` can be used as a troubleshooting tool.

`RedoPcp`
: Specifies that the current pc presentation (for an interactive process) stored by the `pq` program be scrapped and clears the current values stored for the options `Prime`, `ClassBound`, `Exponent` and `Metabelian` and also clears the `pQuotient`, `pQepi` and `pCover` fields of the data record of the process.

`SetupFile := filename`
: Non-interactively, this option directs that `pq` should not be called and that an input file with name *`filename`* (a string), containing the commands necessary for the ANU `pq` standalone, be constructed. The commands written to *`filename`* are also `Info`-ed behind a "`ToPQ> `" prompt at `InfoANUPQ` level 4 (see `InfoANUPQ` (3.3.1)). Except in the case following, the calling function returns `true`. If the calling function is the non-interactive version of one of `Pq`, `PqPCover` or `PqEpimorphism` and the group provided as argument is trivial given with an empty set of generators, then no setup file is written and `fail` is returned (the `pq` program cannot do anything useful with such a group). Interactively, `SetupFile` is ignored.

  Note: Since commands emitted to the `pq` program may depend on knowing what the "current state" is, to form a setup file some "close enough guesses" may sometimes be necessary; when this occurs a warning is `Info`-ed at `InfoANUPQ` or `InfoWarning` level 1. To determine whether the "close enough guesses" give an accurate setup file, it is necessary to run the command without the `SetupFile` option, after either setting the `InfoANUPQ` level to at least 4 (the setup file script can then be compared with the "`ToPQ> `" commands that are `Info`-ed) or setting a `pq` command log file by using `ToPQLog` (see `ToPQLog` (3.4.7)).

`PqWorkspace := workspace`
: Non-interactively, this option sets the memory used by the `pq` program. It sets the maximum number of integer-sized elements to allocate in its main storage array. By default, the `pq` program sets this figure to 10000000. Interactively, `PqWorkspace` is ignored; the memory used in this case may be set by giving `PqStart` a second argument (see `PqStart` (5.1.1)).

`PcgsAutomorphisms`

`PcgsAutomorphisms := false`
: Let *`G`* be the group associated with the calling function (or associated interactive process). Passing the option `PcgsAutomorphisms` without a value (or equivalently setting it to `true`), specifies that a polycyclic generating sequence for the automorphism group (which must be *soluble*)

of `G`, be computed and passed to the pq program. This increases the efficiency of the computation; it also prevents the pq from calling GAP for orbit-stabilizer calculations. By default, `PcgsAutomorphisms` is set to the value returned by `IsSolvable( AutomorphismGroup( G ) )`, and uses the package AutPGrp to compute `AutomorphismGroup( G )` if it is installed. This flag is set to `true` or `false` in the background according to the above criterion by the function `PqDescendants` (see `PqDescendants` (4.4.1) and `PqDescendants` (5.3.6)).

Note: If `PcgsAutomorphisms` is used when the automorphism group of `G` is insoluble, an error message occurs.

`OrderBound := n`
Specifies that only descendants of size at most p^n, where `n` is a non-negative integer, be generated. Note that you cannot set both `OrderBound` and `StepSize`.

`StepSize := n`

`StepSize := list`
For a positive integer `n`, `StepSize` specifies that only those immediate descendants which are a factor p^n bigger than their parent group be generated.

For a list `list` of positive integers such that the sum of the length of `list` and the exponent-p class of `G` is equal to the class bound defined by the option `ClassBound`, `StepSize` specifies that the integers of `list` are the step sizes for each additional class.

`RankInitialSegmentSubgroups := n`
Sets the rank of the initial segment subgroup chosen to be `n`. By default, this has value 0.

`SpaceEfficient`
Specifies that the pq program performs certain calculations of p-group generation more slowly but with greater space efficiency. This flag is frequently necessary for groups of large Frattini quotient rank. The space saving occurs because only one permutation is stored at any one time. This option is only available if the `PcgsAutomorphisms` flag is set to `true` (see `option PcgsAutomorphisms` (6.2)). For an interactive process, `SpaceEfficient` defaults to a previously supplied value for the process. Otherwise (and non-interactively), `SpaceEfficient` is by default `false`.

`CapableDescendants`
By default, *all* (i.e. capable and terminal) descendants are computed. If this flag is set, only capable descendants are computed. Setting this option is equivalent to setting `AllDescendants := false` (see `option AllDescendants` (6.2)), except if both `CapableDescendants` and `AllDescendants` are passed, `AllDescendants` is essentially ignored.

`AllDescendants := false`
By default, *all* descendants are constructed. If this flag is set to `false`, only capable descendants are computed. Passing `AllDescendants` without a value (which is equivalent to setting it to `true`) is superfluous. This option is provided only for backward compatibility with the GAP 3 version of the ANUPQ package, where by default `AllDescendants` was set to `false` (rather than `true`). It is preferable to use `CapableDescendants` (see `option CapableDescendants` (6.2)).

`TreeDepth := class`
: Specifies that the descendants tree developed by `PqDescendantsTreeCoclassOne` (see `PqDescendantsTreeCoclassOne` (A.4.1)) should be extended to class `class`, where `class` is a positive integer.

`SubList := sub`
: Suppose that `L` is the list of descendants generated, then for a list `sub` of integers this option causes `PqDescendants` to return `Sublist( L, sub )`. If an integer `n` is supplied, `PqDescendants` returns `L[n]`.

`NumberOfSolubleAutomorphisms := n`
: Specifies that the number of soluble automorphisms of the automorphism group supplied by `PqPGSupplyAutomorphisms` (see `PqPGSupplyAutomorphisms` (5.9.1)) in a p-group generation calculation is `n`. By default, `n` is taken to be 0; `n` must be a non-negative integer. If $n \geq 0$ then a value for the option `RelativeOrders` (see 6.2) must also be supplied.

`RelativeOrders := list`
: Specifies the relative orders of each soluble automorphism of the automorphism group supplied by `PqPGSupplyAutomorphisms` (see `PqPGSupplyAutomorphisms` (5.9.1)) in a p-group generation calculation. The list `list` must consist of `n` positive integers, where `n` is the value of the option `NumberOfSolubleAutomorphisms` (see 6.2). By default `list` is empty.

`BasicAlgorithm`
: Specifies that an algorithm that the pq program calls its "default" algorithm be used for p-group generation. By default this algorithm is *not* used. If this option is supplied the settings of options `RankInitialSegmentSubgroups`, `AllDescendants`, `Exponent` and `Metabelian` are ignored.

`CustomiseOutput := rec`
: Specifies that fine tuning of the output is desired. The record `rec` should have any subset (or all) of the the following fields:

  `perm := list`
  : where `list` is a list of booleans which determine whether the permutation group output for the automorphism group should contain: the degree, the extended automorphisms, the automorphism matrices, and the permutations, respectively.

  `orbit := list`
  : where `list` is a list of booleans which determine whether the orbit output of the action of the automorphism group should contain: a summary, and a complete listing of orbits, respectively. (It's possible to have *both* a summary and a complete listing.)

  `group := list`
  : where `list` is a list of booleans which determine whether the group output should contain: the standard matrix of each allowable subgroup, the presentation of reduced p-covering groups, the presentation of immediate descendants, the nuclear rank of descendants, and the p-multiplicator rank of descendants, respectively.

  `autgroup := list`
  : where `list` is a list of booleans which determine whether the automorphism group output should contain: the commutator matrix, the automorphism group description of descendants, and the automorphism group order of descendants, respectively.

`trace := val`
: where *val* is a boolean which if `true` specifies algorithm trace data is desired. By default, one does not get algorithm trace data.

Not providing a field (or mis-spelling it!), specifies that the default output is desired. As a convenience, `1` is also accepted as `true`, and any value that is neither `1` nor `true` is taken as `false`. Also for each *list* above, an unbound list entry is taken as `false`. Thus, for example

Example

```
CustomiseOutput := rec(group := [,,1], autgroup := [,1])
```

specifies for the group output that only the presentation of immediate descendants is desired, for the automorphism group output only the automorphism group description of descendants should be printed, that there should be no algorithm trace data, and that the default output should be provided for the permutation group and orbit output.

`StandardPresentationFile := filename`
: Specifies that the file to which the standard presentation is written has name *filename*. If the first character of the string *filename* is not `/`, *filename* is assumed to be the path of a writable file relative to the directory in which GAP was started. If this option is omitted it is written to the file with the name generated by the command `Filename( ANUPQData.tmpdir, "SPres" );`, i.e. the file with name `"SPres"` in the temporary directory in which the pq program executes.

`QueueFactor := n`
: Specifies a queue factor of *n*, where *n* must be a positive integer. This option may be used with `PqNextClass` (see `PqNextClass` (5.6.4)).

The queue factor is used when the pq program uses automorphisms to close a set of elements of the p-multiplicator under their action.

The algorithm used is a spinning algorithm: it starts with a set of vectors in echelonized form (elements of the p-multiplicator) and closes the span of these vectors under the action of the automorphisms. For this each automorphism is applied to each vector and it is checked if the result is contained in the span computed so far. If not, the span becomes bigger and the vector is put into a queue and the automorphisms are applied to this vector at a later stage. The process terminates when the automorphisms have been applied to all vectors and no new vectors have been produced.

For each new vector it is decided, if its processing should be delayed. If the vector contains too many non-zero entries, it is put into a second queue. The elements in this queue are processed only when there are no elements in the first queue left.

The queue factor is a percentage figure. A vector is put into the second queue if the percentage of its non-zero entries exceeds the queue factor.

`Bounds := list`
: Specifies a lower and upper bound on the indices of a list, where *list* is a pair of positive non-decreasing integers. See `PqDisplayStructure` (5.7.23) and `PqDisplayAutomorphisms` (5.7.24) where this option may be used.

`PrintAutomorphisms := list`
: Specifies that automorphism matrices be printed.

`PrintPermutations := list`
: Specifies that permutations of the subgroups be printed.

`Filename := string`
: Specifies that an output or input file to be written to or read from by the `pq` program should have the name *`string`*.

Chapter 7

Installing the ANUPQ Package

The ANU pq program is written in C and the package can be installed under UNIX and in environments similar to UNIX. In particular it is known to work on Linux and Mac OS X, and also on Windows equipped with cygwin.

The current version of the ANUPQ package requires GAP 4.9, and version 1.2 of the AutPGrp package. However, we recommend using at least GAP 4.6 and AutPGrp 1.5.

To install the ANUPQ package, move the file `anupq-`*`XXX`*`.tar.gz` for some version number *`XXX`* into the `pkg` directory in which you plan to install ANUPQ. Usually, this will be the directory `pkg` in the hierarchy of your version of GAP; it is however also possible to keep an additional `pkg` directory in your private directories. The only essential difference with installing ANUPQ in a `pkg` directory different to the GAP home directory is that one must start GAP with the `-l` switch (see Section **Reference: Command Line Options**), e.g. if your private `pkg` directory is a subdirectory of `mygap` in your home directory you might type:

```
gap -l ";myhomedir/mygap"
```

where *`myhomedir`* is the path to your home directory, which may be replaced by a tilde. The empty path before the semicolon is filled in by the default path of the GAP home directory.

Then, in your chosen `pkg` directory, unpack `anupq-`*`XXX`*`.tar.gz` by

```
tar xf anupq-<XXX>.tar.gz
```

Change to the newly created `anupq` directory. Now you need to call `configure`. If you installed ANUPQ into the main `pkg` directory, call

```
./configure
```

If you installed ANUPQ in another directory than the usual 'pkg' subdirectory, instead call

```
./configure --with-gaproot=<path>
```

where *`path`* is the path to the GAP home directory. (You can also call

```
./configure --help
```

for further options.)

What this does is look for a file `sysinfo.gap` in the root directory of GAP in order to determine an architecture name for the subdirectory of `bin` in which to put the compiled `pq` binary. This only makes sense if GAP was compiled for the same architecture that `pq` will be. If you have a shared file system mounted across different architectures, then you should run `configure` and `make` for ANUPQ for each architecture immediately after compiling GAP on the same architecture.

If you had to install the package in your own directory but wish to use the system GAP then you will need to find out what *path* is. To do this, start up GAP and find out what GAP's root path is from finding the value of the variable `GAPInfo.RootPaths`, e.g.

Example

```
gap> GAPInfo.RootPaths;
[ "/usr/local/lib/gap4r4/" ]
```

would tell you to use `/usr/local/lib/gap4r4` for *path*.

The `configure` command will fetch the architecture type for which GAP has been compiled last and create a `Makefile`. You can now simply call

```
make
```

to compile the binary and to install it in the appropriate place.

The path of GAP (see *Note* below) used by the `pq` binary (the value `GAP` is set to in the `make` command) may be over-ridden by setting the environment variable `ANUPQ_GAP_EXEC`. These values are only of interest when the `pq` program is run as a standalone; however, the `testPq` script assumes you have set one of these correctly (see Section 'Testing your ANUPQ installation'). When the `pq` program is started from GAP communication occurs via an iostream, so that the `pq` binary does not actually need to know a valid path for GAP is this case.

Note. By "path of GAP" we mean the path of the command used to invoke GAP (which should be a script, e.g. the `gap.sh` script generated in the `bin` directory for the version of GAP when GAP was compiled). The usual strategy is to copy the `gap.sh` script to a standard location, e.g. `/usr/local/bin/gap`. It is a mistake to copy the GAP executable `gap` (in a directory with name of form `bin/`*`compile-platform`*) to the standard location, since direct invocation of the executable results in GAP starting without being able to find its own library (a fatal error).

7.1 Testing your ANUPQ installation

Now it is time to test the installation. After doing `configure` and `make` you will have a `testPq` script. The script assumes that, if the environment variable `ANUPQ_GAP_EXEC` is set, it is a correct path for GAP, or otherwise that the `make` call that compiled the `pq` program set `GAP` to a correct path for GAP (see Section 'Running the pq program as a standalone' for more details). To run the tests, just type:

```
./testPq
```

Some of the tests the script runs take a while. Please be patient. The script checks that you not only have a correct GAP (at least version 4.4) installation that includes the AutPGrp package, but that the ANUPQ package and its `pq` binary interact correctly. You should see something like the following output:

Example

```
Made dir: /tmp/testPq
Testing installation of ANUPQ Package (version 3.1)

The first two tests check that the pq C program compiled ok.
Testing the pq binary ... OK.
Testing the pq binary's stack size ... OK.
The pq C program compiled ok! We test it's the right one below.

The next tests check that you have the right version of GAP
for the ANUPQ package and that GAP is finding
the right versions of the ANUPQ and AutPGrp packages.

Checking GAP ...
 pq binary made with GAP set to: /usr/local/bin/gap
 Starting GAP to determine version and package availability ...
  GAP version (4.6.5) ... OK.
  GAP found ANUPQ package (version 3.1) ... good.
  GAP found pq binary (version 1.9) ... good.
  GAP found AutPGrp package (version 1.5) ... good.
 GAP is OK.

Checking the link between the pq binary and GAP ... OK.
Testing the standard presentation part of the pq binary ... OK.
Doing p-group generation (final GAP/ANUPQ) test ... OK.
Tests complete.
Removed dir: /tmp/testPq
Enjoy using your functional ANUPQ package!
```

7.2 Running the pq program as a standalone

When the pq program is run as a standalone it sometimes needs to call GAP to compute stabilisers of subgroups; in doing so, it first checks the value of the environment variable ANUPQ_GAP_EXEC, and uses that, if set, or otherwise the value of GAP it was compiled with, as the path for GAP. If you ran testPq (see Section 'Testing your ANUPQ installation') and you got both GAP is OK and the link between the pq binary and GAP is OK, you should be fine. Otherwise heed the recommendations of the error messages you get and run the testPq until all tests are passed.

It is especially important that the GAP, whose path you gave, should know where to find the ANUPQ and AutPGrp packages. To ensure this the path should be to a shell script that invokes GAP. If you needed to install the needed packages in your own directory (because, say, you are not a system administrator) then you should create your own shell script that runs GAP with a correct setting of the -l option and set the path used by the pq binary to the path of that script. To create the script that runs GAP it is easiest to copy the system one and edit it, e.g. start by executing the following UNIX commands (skip the second step if you already have a bin directory; you@unix> is your UNIX prompt):

```
you@unix> cd
you@unix> mkdir bin
you@unix> cd bin
you@unix> which gap
```

```
/usr/local/bin/gap
you@unix> cp /usr/local/bin/gap mygap
you@unix> chmod +x mygap
```

At the second-last step use the path of GAP returned by `which gap`. Now hopefully you will have a copy of the script that runs the system GAP in `mygap`. Now use your favourite editor to edit the `-l` part of the last line of `mygap` which should initially look something like:

```
exec $GAP_DIR/bin/$GAP_PRG -m $GAP_MEM -o 970m -l $GAP_DIR $*
```

so that it becomes (the tilde is a UNIX abbreviation for your home directory):

```
exec $GAP_DIR/bin/$GAP_PRG -m $GAP_MEM -o 970m -l "$GAP_DIR;~/gapstuff" $*
```

assuming that your personal GAP `pkg` directory is a subdirectory of `gapstuff` in your home directory. Finally, to let the `pq` program know where GAP is and also know where your `pkg` directory is that contains ANUPQ, set the environment variable `ANUPQ_GAP_EXEC` to the complete (i.e. absolute) path of your `mygap` script (do not use the tilde abbreviation).

Appendix A

Examples

There are a large number of examples provided with the ANUPQ package. These may be executed or displayed via the function PqExample (see PqExample (3.4.4)). Each example resides in a file of the same name in the directory examples. Most of the examples are translations to GAP of examples provided for the pq standalone by Eamonn O'Brien; the standalone examples are found in directories standalone/examples (p-quotient and p-group generation examples) and standalone/isom (standard presentation examples). The first line of each example indicates its origin. All the examples seen in earlier chapters of this manual are also available as examples, in a slightly modified form (the example which one can run in order to see something very close to the text example "live" is always indicated near -- usually immediately after -- the text example). The format of the (PqExample) examples is such that they can be read by the standard Read function of GAP, but certain features and comments are interpreted by the function PqExample to do somewhat more than Read does. In particular, any function without a -i, -ni or .g suffix has both a non-interactive and interactive form; in these cases, the default form is the non-interactive form, and giving PqStart as second argument generates the interactive form.

Running PqExample without an argument or with a non-existent example Infos the available examples and some hints on usage:

Example

```
gap> PqExample();
#I                  PqExample Index (Table of Contents)
#I                  ----------------------------------
#I  This table of possible examples is displayed when calling 'PqExample'
#I  with no arguments, or with the argument: "index" (meant in the  sense
#I  of ''list''), or with a non-existent example name.
#I
#I  Examples that have a name ending in '-ni' are  non-interactive  only.
#I  Examples that have a  name  ending  in  '-i'  are  interactive  only.
#I  Examples with names ending in '.g' also have  only  one  form.  Other
#I  examples have both a non-interactive and an  interactive  form;  call
#I  'PqExample' with 2nd argument 'PqStart' to get the  interactive  form
#I  of the example. The substring 'PG' in an  example  name  indicates  a
#I  p-Group Generation example, 'SP' indicates  a  Standard  Presentation
#I  example, 'Rel' indicates it uses  the  'Relators'  option,  and  'Id'
#I  indicates it uses the 'Identities' option.
#I
#I  The following ANUPQ examples are available:
#I
```

```
#I   p-Quotient examples:
#I    general:
#I     "Pq"                    "Pq-ni"                 "PqEpimorphism"
#I     "PqPCover"              "PqSupplementInnerAutomorphisms"
#I    2-groups:
#I     "2gp-Rel"               "2gp-Rel-i"             "2gp-a-Rel-i"
#I     "B2-4"                  "B2-4-Id"               "B2-8-i"
#I     "B4-4-i"                "B4-4-a-i"              "B5-4.g"
#I    3-groups:
#I     "3gp-Rel-i"             "3gp-a-Rel"             "3gp-a-Rel-i"
#I     "3gp-a-x-Rel-i"         "3gp-maxoccur-Rel-i"
#I    5-groups:
#I     "5gp-Rel-i"             "5gp-a-Rel-i"           "5gp-b-Rel-i"
#I     "5gp-c-Rel-i"           "5gp-metabelian-Rel-i" "5gp-maxoccur-Rel-i"
#I     "F2-5-i"                "B2-5-i"                "R2-5-i"
#I     "R2-5-x-i"              "B5-5-Engel3-Id"
#I    7-groups:
#I     "7gp-Rel-i"
#I    11-groups:
#I     "11gp-i"                "11gp-Rel-i"            "11gp-a-Rel-i"
#I     "11gp-3-Engel-Id"       "11gp-3-Engel-Id-i"
#I
#I   p-Group Generation examples:
#I    general:
#I     "PqDescendants-1"       "PqDescendants-2"       "PqDescendants-3"
#I     "PqDescendants-1-i"
#I    2-groups:
#I     "2gp-PG-i"              "2gp-PG-2-i"            "2gp-PG-3-i"
#I     "2gp-PG-4-i"            "2gp-PG-e4-i"
#I     "PqDescendantsTreeCoclassOne-16-i"
#I    3-groups:
#I     "3gp-PG-i"              "3gp-PG-4-i"            "3gp-PG-x-i"
#I     "3gp-PG-x-1-i"          "PqDescendants-treetraverse-i"
#I     "PqDescendantsTreeCoclassOne-9-i"
#I    5-groups:
#I     "5gp-PG-i"              "Nott-PG-Rel-i"         "Nott-APG-Rel-i"
#I     "PqDescendantsTreeCoclassOne-25-i"
#I    7,11-groups:
#I     "7gp-PG-i"              "11gp-PG-i"
#I
#I   Standard Presentation examples:
#I    general:
#I     "StandardPresentation" "StandardPresentation-i"
#I     "EpimorphismStandardPresentation"
#I     "EpimorphismStandardPresentation-i"           "IsIsomorphicPGroup-ni"
#I    2-groups:
#I     "2gp-SP-Rel-i"          "2gp-SP-1-Rel-i"        "2gp-SP-2-Rel-i"
#I     "2gp-SP-3-Rel-i"        "2gp-SP-4-Rel-i"        "2gp-SP-d-Rel-i"
#I     "gp-256-SP-Rel-i"       "B2-4-SP-i"             "G2-SP-Rel-i"
#I    3-groups:
#I     "3gp-SP-Rel-i"          "3gp-SP-1-Rel-i"        "3gp-SP-2-Rel-i"
#I     "3gp-SP-3-Rel-i"        "3gp-SP-4-Rel-i"        "G3-SP-Rel-i"
```

```
#I    5-groups:
#I     "5gp-SP-Rel-i"          "5gp-SP-a-Rel-i"        "5gp-SP-b-Rel-i"
#I     "5gp-SP-big-Rel-i"      "5gp-SP-d-Rel-i"        "G5-SP-Rel-i"
#I     "G5-SP-a-Rel-i"         "Nott-SP-Rel-i"
#I    7-groups:
#I     "7gp-SP-Rel-i"          "7gp-SP-a-Rel-i"        "7gp-SP-b-Rel-i"
#I    11-groups:
#I     "11gp-SP-a-i"           "11gp-SP-a-Rel-i"       "11gp-SP-a-Rel-1-i"
#I     "11gp-SP-b-i"           "11gp-SP-b-Rel-i"       "11gp-SP-c-Rel-i"
#I
#I  Notes
#I  -----
#I  1. The example (first) argument of 'PqExample' is a string; each
#I     example above is in double quotes to remind you to include them.
#I  2. Some examples accept options. To find out whether a particular
#I     example accepts options, display it first (by including 'Display'
#I     as last argument) which will also indicate how 'PqExample'
#I     interprets the options, e.g. 'PqExample("11gp-SP-a-i", Display);'.
#I  3. Try 'SetInfoLevel(InfoANUPQ, <n>);' for some <n> in [2 .. 4]
#I     before calling PqExample, to see what's going on behind the scenes.
#I
```

If on your terminal you are unable to scroll back, an alternative to typing `PqExample();` to see the displayed examples is to use on-line help, i.e. you may type:

Example

```
gap> ?anupq:examples
```

which will display this appendix in a GAP session. If you are not fussed about the order in which the examples are organised, `AllPqExamples();` lists the available examples relatively compactly (see `AllPqExamples` (3.4.5)).

In the remainder of this appendix we will discuss particular aspects related to the `Relators` (see 6.2) and `Identities` (see 6.2) options, and the construction of the Burnside group $B(5,4)$.

A.1 The Relators Option

The `Relators` option was included because computations involving words containing commutators that are pre-expanded by GAP before being passed to the `pq` program may run considerably more slowly, than the same computations being run with GAP pre-expansions avoided. The following examples demonstrate a case where the performance hit due to pre-expansion of commutators by GAP is a factor of order 100 (in order to see timing information from the `pq` program, we set the `InfoANUPQ` level to 2).

Firstly, we run the example that allows pre-expansion of commutators (the function `PqLeftNormComm` is provided by the ANUPQ package; see `PqLeftNormComm` (3.4.1)). Note that since the two commutators of this example are *very* long (taking more than an page to print), we have edited the output at this point.

Example

```
gap> SetInfoLevel(InfoANUPQ, 2); #to see timing information
gap> PqExample("11gp-i");
#I  #Example: "11gp-i" . . . based on: examples/11gp
```

```
#I  F, a, b, c, R, procId are local to ‘PqExample’
gap> F := FreeGroup("a", "b", "c"); a := F.1; b := F.2; c := F.3;
<free group on the generators [ a, b, c ]>
a
b
c
gap> R := [PqLeftNormComm([b, a, a, b, c])^11,
>          PqLeftNormComm([a, b, b, a, b, c])^11, (a * b)^11];;
gap> procId := PqStart(F/R : Prime := 11);
1
gap> PqPcPresentation(procId : ClassBound := 7,
>                              OutputLevel := 1);
#I  Lower exponent-11 central series for [grp]
#I  Group: [grp] to lower exponent-11 central class 1 has order 11^3
#I  Group: [grp] to lower exponent-11 central class 2 has order 11^8
#I  Group: [grp] to lower exponent-11 central class 3 has order 11^19
#I  Group: [grp] to lower exponent-11 central class 4 has order 11^42
#I  Group: [grp] to lower exponent-11 central class 5 has order 11^98
#I  Group: [grp] to lower exponent-11 central class 6 has order 11^228
#I  Group: [grp] to lower exponent-11 central class 7 has order 11^563
#I  Computation of presentation took 27.04 seconds
gap> PqSavePcPresentation(procId, ANUPQData.outfile);
#I  Variables used in ‘PqExample’ are saved in ‘ANUPQData.example.vars’.
```

Now we do the same calculation using the `Relators` option. In this way, the commutators are passed directly as strings to the pq program, so that GAP does not “see” them and pre-expand them.

Example

```
gap> PqExample("11gp-Rel-i");
#I  #Example: "11gp-Rel-i" . . . based on: examples/11gp
#I  #(equivalent to "11gp-i" example but uses ‘Relators’ option)
#I  F, rels, procId are local to ‘PqExample’
gap> F := FreeGroup("a", "b", "c");
<free group on the generators [ a, b, c ]>
gap> rels := ["[b, a, a, b, c]^11", "[a, b, b, a, b, c]^11", "(a * b)^11"];
[ "[b, a, a, b, c]^11", "[a, b, b, a, b, c]^11", "(a * b)^11" ]
gap> procId := PqStart(F : Prime := 11, Relators := rels);
2
gap> PqPcPresentation(procId : ClassBound := 7,
>                              OutputLevel := 1);
#I  Relators parsed ok.
#I  Lower exponent-11 central series for [grp]
#I  Group: [grp] to lower exponent-11 central class 1 has order 11^3
#I  Group: [grp] to lower exponent-11 central class 2 has order 11^8
#I  Group: [grp] to lower exponent-11 central class 3 has order 11^19
#I  Group: [grp] to lower exponent-11 central class 4 has order 11^42
#I  Group: [grp] to lower exponent-11 central class 5 has order 11^98
#I  Group: [grp] to lower exponent-11 central class 6 has order 11^228
#I  Group: [grp] to lower exponent-11 central class 7 has order 11^563
#I  Computation of presentation took 0.27 seconds
gap> PqSavePcPresentation(procId, ANUPQData.outfile);
#I  Variables used in ‘PqExample’ are saved in ‘ANUPQData.example.vars’.
```

A.2 The Identities Option and PqEvaluateIdentities Function

Please pay heed to the warnings given for the `Identities` option (see 6.2); it is written mainly at the GAP level and is not particularly optimised. The `Identities` option allows one to compute p-quotients that satisfy an identity. A trivial example better done using the `Exponent` option, but which nevertheless demonstrates the usage of the `Identities` option, is as follows:

Example

```
gap> SetInfoLevel(InfoANUPQ, 1);
gap> PqExample("B2-4-Id");
#I  #Example: "B2-4-Id" . . . alternative way to generate B(2, 4)
#I  #Generates B(2, 4) by using the 'Identities' option
#I  #... this is not as efficient as using 'Exponent' but
#I  #demonstrates the usage of the 'Identities' option.
#I  F, f, procId are local to 'PqExample'
gap> F := FreeGroup("a", "b");
<free group on the generators [ a, b ]>
gap> # All words w in the pc generators of B(2, 4) satisfy f(w) = 1
gap> f := w -> w^4;
function( w ) ... end
gap> Pq( F : Prime := 2, Identities := [ f ] );
#I  Class 1 with 2 generators.
#I  Class 2 with 5 generators.
#I  Class 3 with 7 generators.
#I  Class 4 with 10 generators.
#I  Class 5 with 12 generators.
#I  Class 5 with 12 generators.
<pc group of size 4096 with 12 generators>
#I  Variables used in 'PqExample' are saved in 'ANUPQData.example.vars'.
gap> time;
1400
```

Note that the `time` statement gives the time in milliseconds spent by GAP in executing the `PqExample("B2-4-Id");` command (i.e. everything up to the `Info`-ing of the variables used), but over 90% of that time is spent in the final `Pq` statement. The time spent by the `pq` program, which is negligible anyway (you can check this by running the example while the `InfoANUPQ` level is set to 2), is not counted by `time`.

Since the identity used in the above construction of $B(2,4)$ is just an exponent law, the "right" way to compute it is via the `Exponent` option (see 6.2), which is implemented at the C level and *is* highly optimised. Consequently, the `Exponent` option is significantly faster, generally by several orders of magnitude:

Example

```
gap> SetInfoLevel(InfoANUPQ, 2); # to see time spent by the 'pq' program
gap> PqExample("B2-4");
#I  #Example: "B2-4" . . . the ''right'' way to generate B(2, 4)
#I  #Generates B(2, 4) by using the 'Exponent' option
#I  F, procId are local to 'PqExample'
gap> F := FreeGroup("a", "b");
<free group on the generators [ a, b ]>
gap> Pq( F : Prime := 2, Exponent := 4 );
#I  Computation of presentation took 0.00 seconds
<pc group of size 4096 with 12 generators>
```

```
#I  Variables used in ‘PqExample’ are saved in ‘ANUPQData.example.vars’.
gap> time; # time spent by GAP in executing ‘PqExample("B2-4");’
50
```

The following example uses the Identities option to compute a 3-Engel group for the prime 11. As is the case for the example "B2-4-Id", the example has both a non-interactive and an interactive form; below, we demonstrate the interactive form.

Example

```
gap> SetInfoLevel(InfoANUPQ, 1); # reset InfoANUPQ to default level
gap> PqExample("11gp-3-Engel-Id", PqStart);
#I  #Example: "11gp-3-Engel-Id" . . . 3-Engel group for prime 11
#I  #Non-trivial example of using the ‘Identities’ option
#I  F, a, b, G, f, procId, Q are local to ‘PqExample’
gap> F := FreeGroup("a", "b"); a := F.1; b := F.2;
<free group on the generators [ a, b ]>
a
b
gap> G := F/[ a^11, b^11 ];
<fp group on the generators [ a, b ]>
gap> # All word pairs u, v in the pc generators of the 11-quotient Q of G
gap> # must satisfy the Engel identity: [u, v, v, v] = 1.
gap> f := function(u, v) return PqLeftNormComm( [u, v, v, v] ); end;
function( u, v ) ... end
gap> procId := PqStart( G );
3
gap> Q := Pq( procId : Prime := 11, Identities := [ f ] );
#I  Class 1 with 2 generators.
#I  Class 2 with 3 generators.
#I  Class 3 with 5 generators.
#I  Class 3 with 5 generators.
<pc group of size 161051 with 5 generators>
gap> # We do a ‘‘sample’’ check that pairs of elements of Q do satisfy
gap> # the given identity:
gap> f( Random(Q), Random(Q) );
<identity> of ...
gap> f( Q.1, Q.2 );
<identity> of ...
#I  Variables used in ‘PqExample’ are saved in ‘ANUPQData.example.vars’.
```

The (interactive) call to Pq above is essentially equivalent to a call to PqPcPresentation with the same arguments and options followed by a call to PqCurrentGroup. Moreover, the call to PqPcPresentation (as described in PqPcPresentation (5.6.1)) is equivalent to a “class 1” call to PqPcPresentation followed by the requisite number of calls to PqNextClass, and with the Identities option set, both PqPcPresentation and PqNextClass “quietly” perform the equivalent of a PqEvaluateIdentities call. In the following example we break down the Pq call into its low-level equivalents, and set and unset the Identities option to show where PqEvaluateIdentities fits into this scheme.

Example

```
gap> PqExample("11gp-3-Engel-Id-i");
#I  #Example: "11gp-3-Engel-Id-i" . . . 3-Engel grp for prime 11
#I  #Variation of "11gp-3-Engel-Id" broken down into its lower-level component
```

```
#I  #command parts.
#I  F, a, b, G, f, procId, Q are local to ‘PqExample’
gap> F := FreeGroup("a", "b"); a := F.1; b := F.2;
<free group on the generators [ a, b ]>
a
b
gap> G := F/[ a^11, b^11 ];
<fp group on the generators [ a, b ]>
gap> # All word pairs u, v in the pc generators of the 11-quotient Q of G
gap> # must satisfy the Engel identity: [u, v, v, v] = 1.
gap> f := function(u, v) return PqLeftNormComm( [u, v, v, v] ); end;
function( u, v ) ... end
gap> procId := PqStart( G : Prime := 11 );
4
gap> PqPcPresentation( procId : ClassBound := 1);
gap> PqEvaluateIdentities( procId : Identities := [f] );
#I  Class 1 with 2 generators.
gap> for c in [2 .. 4] do
>      PqNextClass( procId : Identities := [] ); #reset ‘Identities’ option
>      PqEvaluateIdentities( procId : Identities := [f] );
>    od;
#I  Class 2 with 3 generators.
#I  Class 3 with 5 generators.
#I  Class 3 with 5 generators.
gap> Q := PqCurrentGroup( procId );
<pc group of size 161051 with 5 generators>
gap> # We do a ‘‘sample’’ check that pairs of elements of Q do satisfy
gap> # the given identity:
gap> f( Random(Q), Random(Q) );
<identity> of ...
gap> f( Q.1, Q.2 );
<identity> of ...
#I  Variables used in ‘PqExample’ are saved in ‘ANUPQData.example.vars’.
```

A.3 A Large Example

An example demonstrating how a large computation can be organised with the ANUPQ package is the computation of the Burnside group $B(5,4)$, the largest group of exponent 4 generated by 5 elements. It has order 2^{2728} and lower exponent-p central class 13. The example `"B5-4.g"` computes $B(5,4)$; it is based on a pq standalone input file written by M. F. Newman.

To be able to do examples like this was part of the motivation to provide access to the low-level functions of the standalone program from within GAP.

Please note that the construction uses the knowledge gained by Newman and O'Brien in their initial construction of $B(5,4)$, in particular, insight into the commutator structure of the group and the knowledge of the p-central class and the order of $B(5,4)$. Therefore, the construction cannot be used to prove that $B(5,4)$ has the order and class mentioned above. It is merely a reconstruction of the group. More information is contained in the header of the file `examples/B5-4.g`.

Example

```
procId := PqStart( FreeGroup(5) : Exponent := 4, Prime := 2 );
Pq( procId : ClassBound := 2 );
```

```
PqSupplyAutomorphisms( procId,
      [
        [ [ 1, 1, 0, 0, 0],      # first automorphism
          [ 0, 1, 0, 0, 0],
          [ 0, 0, 1, 0, 0],
          [ 0, 0, 0, 1, 0],
          [ 0, 0, 0, 0, 1] ],

        [ [ 0, 0, 0, 0, 1],      # second automorphism
          [ 1, 0, 0, 0, 0],
          [ 0, 1, 0, 0, 0],
          [ 0, 0, 1, 0, 0],
          [ 0, 0, 0, 1, 0] ]
                           ] );;

Relations :=
  [ [],          ## class 1
    [],          ## class 2
    [],          ## class 3
    [],          ## class 4
    [],          ## class 5
    [],          ## class 6
    ## class 7
    [ [ "x2","x1","x1","x3","x4","x4","x4" ] ],
    ## class 8
    [ [ "x2","x1","x1","x3","x4","x5","x5","x5" ] ],
    ## class 9
    [ [ "x2","x1","x1","x3","x4","x4","x5","x5","x5" ],
      [ "x2","x1","x1","x2","x3","x4","x5","x5","x5" ],
      [ "x2","x1","x1","x3","x3","x4","x5","x5","x5" ] ],
    ## class 10
    [ [ "x2","x1","x1","x2","x3","x3","x4","x5","x5","x5" ],
      [ "x2","x1","x1","x3","x3","x4","x4","x5","x5","x5" ] ],
    ## class 11
    [ [ "x2","x1","x1","x2","x3","x3","x4","x4","x5","x5","x5" ],
      [ "x2","x1","x1","x2","x3","x1","x3","x4","x2","x4","x3" ] ],
    ## class 12
    [ [ "x2","x1","x1","x2","x3","x1","x3","x4","x2","x5","x5","x5" ],
      [ "x2","x1","x1","x3","x2","x4","x3","x5","x4","x5","x5","x5" ] ],
    ## class 13
    [ [ "x2","x1","x1","x2","x3","x1","x3","x4","x2","x4","x5","x5","x5"
        ] ]
];

for class in [ 3 .. 13 ] do
    Print( "Computing class ", class, "\n" );
    PqSetupTablesForNextClass( procId );

    for w in [ class, class-1 .. 7 ] do

        PqAddTails( procId, w );
        PqDisplayPcPresentation( procId );
```

```
        if Relations[ w ] <> [] then
            # recalculate automorphisms
            PqExtendAutomorphisms( procId );

            for r in Relations[ w ] do
                Print( "Collecting ", r, "\n" );
                PqCommutator( procId, r, 1 );
                PqEchelonise( procId );
                PqApplyAutomorphisms( procId, 15 ); #queue factor = 15
            od;

            PqEliminateRedundantGenerators( procId );
        fi;
        PqComputeTails( procId, w );
    od;
    PqDisplayPcPresentation( procId );

    smallclass := Minimum( class, 6 );
    for w in [ smallclass, smallclass-1 .. 2 ] do
        PqTails( procId, w );
    od;
    # recalculate automorphisms
    PqExtendAutomorphisms( procId );
    PqCollect( procId, "x5^4" );
    PqEchelonise( procId );
    PqApplyAutomorphisms( procId, 15 ); #queue factor = 15
    PqEliminateRedundantGenerators( procId );
    PqDisplayPcPresentation( procId );
od;
```

A.4 Developing descendants trees

In the following example we will explore the 3-groups of rank 2 and 3-coclass 1 up to 3-class 5. This will be done using the p-group generation machinery of the package. We start with the elementary abelian 3-group of rank 2. From within GAP, run the example "PqDescendants-treetraverse-i" via PqExample (see PqExample (3.4.4)).

Example

```
gap> G := ElementaryAbelianGroup( 9 );
<pc group of size 9 with 2 generators>
gap> procId := PqStart( G );
5
gap> #
gap> #  Below, we use the option StepSize in order to construct descendants
gap> #  of coclass 1. This is equivalent to setting the StepSize to 1 in
gap> #  each descendant calculation.
gap> #
gap> #  The elementary abelian group of order 9 has 3 descendants of
gap> #  3-class 2 and 3-coclass 1, as the result of the next command
gap> #  shows.
gap> #
```

```
gap> PqDescendants( procId : StepSize := 1 );
[ <pc group of size 27 with 3 generators>,
  <pc group of size 27 with 3 generators>,
  <pc group of size 27 with 3 generators> ]
gap> #
gap> #  Now we will compute the descendants of coclass 1 for each of the
gap> #  groups above. Then we will compute the descendants  of coclass 1
gap> #  of each descendant and so  on.  Note  that the  pq program keeps
gap> #  one file for each class at a time.  For example, the descendants
gap> #  calculation for  the  second  group  of class  2  overwrites the
gap> #  descendant file  obtained  from  the  first  group  of  class 2.
gap> #  Hence,  we have to traverse the descendants tree  in depth first
gap> #  order.
gap> #
gap> PqPGSetDescendantToPcp( procId, 2, 1 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
2
gap> PqPGSetDescendantToPcp( procId, 3, 1 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
2
gap> PqPGSetDescendantToPcp( procId, 4, 1 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
2
gap> #
gap> #  At this point we stop traversing the ‘‘left most’’ branch of the
gap> #  descendants tree and move upwards.
gap> #
gap> PqPGSetDescendantToPcp( procId, 4, 2 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
#I  group restored from file is incapable
0
gap> PqPGSetDescendantToPcp( procId, 3, 2 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
#I  group restored from file is incapable
0
gap> #
gap> #  The computations above indicate that the descendants subtree under
gap> #  the first descendant of the elementary abelian group of order 9
gap> #  will have only one path of infinite length.
gap> #
gap> PqPGSetDescendantToPcp( procId, 2, 2 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
4
gap> #
gap> #  We get four descendants here, three of which will turn out to be
gap> #  incapable, i.e., they have no descendants and are terminal nodes
```

```
gap> #  in the descendants tree.
gap> #
gap> PqPGSetDescendantToPcp( procId, 2, 3 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
#I  group restored from file is incapable
0
gap> #
gap> #  The third descendant of class three is incapable.  Let us return
gap> #  to the second descendant of class 2.
gap> #
gap> PqPGSetDescendantToPcp( procId, 2, 2 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
4
gap> PqPGSetDescendantToPcp( procId, 3, 1 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
#I  group restored from file is incapable
0
gap> PqPGSetDescendantToPcp( procId, 3, 2 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
#I  group restored from file is incapable
0
gap> #
gap> #  We skip the third descendant for the moment ...
gap> #
gap> PqPGSetDescendantToPcp( procId, 3, 4 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
#I  group restored from file is incapable
0
gap> #
gap> #  ... and look at it now.
gap> #
gap> PqPGSetDescendantToPcp( procId, 3, 3 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
6
gap> #
gap> #  In this branch of the descendant tree we get 6 descendants of class
gap> #  three.  Of those 5 will turn out to be incapable and one will have
gap> #  7 descendants.
gap> #
gap> PqPGSetDescendantToPcp( procId, 4, 1 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
#I  group restored from file is incapable
0
gap> PqPGSetDescendantToPcp( procId, 4, 2 );
gap> PqPGExtendAutomorphisms( procId );
```

```
gap> PqPGConstructDescendants( procId : StepSize := 1 );
7
gap> PqPGSetDescendantToPcp( procId, 4, 3 );
gap> PqPGExtendAutomorphisms( procId );
gap> PqPGConstructDescendants( procId : StepSize := 1 );
#I  group restored from file is incapable
0
```

To automate the above procedure to some extent we provide:

A.4.1 PqDescendantsTreeCoclassOne

▷ `PqDescendantsTreeCoclassOne(`*i*`)` (function)
▷ `PqDescendantsTreeCoclassOne()` (function)

for the *i*th or default interactive ANUPQ process, generate a descendant tree for the group of the process (which must be a pc p-group) consisting of descendants of p-coclass 1 and extending to the class determined by the option `TreeDepth` (or 6 if the option is omitted). In an XGAP session, a graphical representation of the descendants tree appears in a separate window. Subsequent calls to `PqDescendantsTreeCoclassOne` for the same process may be used to extend the descendant tree from the last descendant computed that itself has more than one descendant. `PqDescendantsTreeCoclassOne` also accepts the options `CapableDescendants` (or `AllDescendants`) and any options accepted by the interactive `PqDescendants` function (see `PqDescendants` (5.3.6)).

Notes

1. `PqDescendantsTreeCoclassOne` first calls `PqDescendants`. If `PqDescendants` has already been called for the process, the previous value computed is used and a warning is `Info`-ed at `InfoANUPQ` level 1.

2. As each descendant is processed its unique label defined by the `pq` program and number of descendants is `Info`-ed at `InfoANUPQ` level 1.

3. `PqDescendantsTreeCoclassOne` is an "experimental" function that is included to demonstrate the sort of things that are possible with the p-group generation machinery.

Ignoring the extra functionality provided in an XGAP session, `PqDescendantsTreeCoclassOne`, with one argument that is the index of an interactive ANUPQ process, is approximately equivalent to:

```
PqDescendantsTreeCoclassOne := function( procId )
    local des, i;

    des := PqDescendants( procId : StepSize := 1 );
    RecurseDescendants( procId, 2, Length(des) );
end;
```

where `RecurseDescendants` is (approximately) defined as follows:

```
RecurseDescendants := function( procId, class, n )
    local i, nr;
```

```
    if class > ValueOption("TreeDepth") then return; fi;

    for i in [1..n] do
        PqPGSetDescendantToPcp( procId, class, i );
        PqPGExtendAutomorphisms( procId );
        nr := PqPGConstructDescendants( procId : StepSize := 1 );
        Print( "Number of descendants of group ", i,
               " at class ", class, ": ", nr, "\n" );
        RecurseDescendants( procId, class+1, nr );
    od;
    return;
end;
```

The following examples (executed via `PqExample`; see `PqExample` (3.4.4)), demonstrate the use of `PqDescendantsTreeCoclassOne`:

`"PqDescendantsTreeCoclassOne-9-i"`
: approximately does example `"PqDescendants-treetraverse-i"` again using `PqDescendantsTreeCoclassOne`;

`"PqDescendantsTreeCoclassOne-16-i"`
: uses the option `CapableDescendants`; and

`"PqDescendantsTreeCoclassOne-25-i"`
: calculates all descendants by omitting the `CapableDescendants` option.

The numbers 9, 16 and 25 respectively, indicate the order of the elementary abelian group to which `PqDescendantsTreeCoclassOne` is applied for these examples.

References

[HN80] George Havas and M. F. Newman. Application of computers to questions like those of Burnside. In *Burnside groups (Proc. Workshop, Univ. Bielefeld, Bielefeld, 1977)*, volume 806 of *Lecture Notes in Math.*, pages 211–230. Springer, Berlin, 1980. 9, 10

[LGS90] C. R. Leedham-Green and L. H. Soicher. Collection from the left and other strategies. *J. Symbolic Comput.*, 9(5-6):665–675, 1990. Computational group theory, Part 1. 7

[New77] M. F. Newman. Determination of groups of prime-power order. In *Group theory (Proc. Miniconf., Australian Nat. Univ., Canberra, 1975)*, pages 73–84. Lecture Notes in Math., Vol. 573. Springer, Berlin, 1977. 8, 10

[NNN98] M. F. Newman, Werner Nickel, and Alice C. Niemeyer. Descriptions of groups of prime-power order. *J. Symbolic Comput.*, 25(5):665–682, 1998. 7, 8, 9, 10

[NO96] M. F. Newman and E. A. O'Brien. Application of computers to questions like those of Burnside. II. *Internat. J. Algebra Comput.*, 6(5):593–605, 1996. 9, 10

[O'B90] E. A. O'Brien. The p-group generation algorithm. *J. Symbolic Comput.*, 9(5-6):677–698, 1990. Computational group theory, Part 1. 8, 9, 10, 11

[O'B94] E. A. O'Brien. Isomorphism testing for p-groups. *J. Symbolic Comput.*, 17(2):131, 133–147, 1994. 11

[O'B95] E. A. O'Brien. Computing automorphism groups of p-groups. In *Computational algebra and number theory (Sydney, 1992)*, volume 325 of *Math. Appl.*, pages 83–90. Kluwer Acad. Publ., Dordrecht, 1995. 11

[Sim94] Charles C. Sims. *Computation with finitely presented groups*, volume 48 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 1994. 9

[VL84] M. R. Vaughan-Lee. An aspect of the nilpotent quotient algorithm. In *Computational group theory (Durham, 1982)*, pages 75–83. Academic Press, London, 1984. 8, 9

[VL90a] M. R. Vaughan-Lee. Collection from the left. *J. Symbolic Comput.*, 9(5-6):725–733, 1990. Computational group theory, Part 1. 7

[VL90b] Michael Vaughan-Lee. *The restricted Burnside problem*, volume 5 of *London Mathematical Society Monographs. New Series*. The Clarendon Press Oxford University Press, New York, 1990. Oxford Science Publications. 10

Index