

AutPGrp

Computing the Automorphism Group of a p -Group

1.12.0

17 May 2026

Bettina Eick
Eamonn O'Brien

Bettina Eick Email: beick@tu-bs.de
Homepage: <http://www.iaa.tu-bs.de/beick>
Address: Institut Analysis und Algebra
TU Braunschweig
Universitätsplatz 2
D-38106 Braunschweig
Germany

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

Abstract

The AutPGrp package introduces a new function to compute the automorphism group of a finite p -group. The underlying algorithm is a refinement of the methods described in O'Brien (1995). In particular, this implementation is more efficient in both time and space requirements and hence has a wider range of applications than the ANUPQ method. Our package is written in GAP code and it makes use of a number of methods from the GAP library such as the MeatAxe for matrix groups and permutation group functions. We have compared our method to the others available in GAP. Our package usually out-performs all but the method designed for finite abelian groups. We note that our method uses the small groups library in certain cases and hence our algorithm is more effective if the small groups library is installed.

Copyright

Bettina Eick and Eamonn O'Brien.

AutPGrp is free software; you can redistribute it and/or modify it under the terms of the [GNU General Public License](#) as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

Acknowledgements

We thank Alexander Hulpke for helping us with efficiency problems. Werner Nickel provided some functions from the GAP PQuotient which are used in this package.

Contents

1	Introduction	4
1.1	Introduction	4
2	The automorphism group method	5
2.1	The automorphism group method	5
3	The underlying function	7
3.1	The underlying function	7
4	Influencing the algorithm	10
4.1	Outline of the algorithm	10
4.2	The initialisation step	11
4.3	Stabilisers in matrix groups	11
4.4	Searching for a small generating set	12
4.5	An interactive version of the algorithm	12
5	Additional features of the package	15
5.1	Additional features of the package	15
	Index	16

Chapter 1

Introduction

1.1 Introduction

Given an arbitrary finite group, the computation of its automorphism group is a very difficult task. Pioneer work in this area was carried out by Felsch and Neubueser (1970), whose algorithm used the output of their subgroup lattice program. A technique developed by Neubueser in the early 1970s sought to compute the automorphism group viewed as a permutation group acting on unions of certain conjugacy classes of the group. A similar method was implemented by Hulpke (1997) in the GAP 4 library. Recently, Cannon and Holt (1999) presented a new algorithm which uses a *hybrid group* approach.

More efficient approaches are available to determine the automorphism group for groups satisfying certain properties. Following the work of Shoda (1928), Hulpke in 1997 implemented a practical method for finite abelian groups in the GAP 4 library. Wursthorn (1993) adapted modular group algebra techniques to compute the automorphism groups of p -groups; the GAP 3 share package *Sisyphe* includes an implementation. Smith (1994) introduced an algorithm for finite solvable groups which is available in the *AutAg* share package of GAP 3.

Moreover, the p -group generation method of Newman (1977) and O'Brien (1990) can be modified to compute the automorphism group of a finite p -group as outlined in O'Brien (1995). This algorithm is implemented in the ANU pq C program.

Here we introduce a new function to compute the automorphism group of a finite p -group. The underlying algorithm is a refinement of the methods described in O'Brien (1995). In particular, this implementation is more efficient in both time and space requirements and hence has a wider range of applications than the ANU pq method. Our package is written in GAP code and it makes use of a number of methods from the GAP library such as the *MeatAxe* for matrix groups and permutation group functions.

The GAP 4 package *ANUPQ*, which is an interface to most of the functionality of the ANU pq C program, uses the *AutPGrp* package to compute automorphism groups of p -groups.

We have compared our method to the others available in GAP. Our package usually out-performs all but the method designed for finite abelian groups. We note that our method uses the small groups library in certain cases and hence our algorithm is more effective if the small groups library is installed.

Chapter 2

The automorphism group method

2.1 The automorphism group method

The AutPGrp package installs a method for AutomorphismGroup for a finite p -group (see also Section (Reference: Groups of Automorphisms) in the GAP Reference Manual).

2.1.1 AutomorphismGroup

▷ AutomorphismGroup(G) (operation)

Returns: The automorphism group of G .

The input is a finite p -group G . If the filters IsPGroup, IsFinite and CanEasilyComputePcgs are set and true for G , the method selection of GAP 4 invokes this algorithm.

The output of the method is an automorphism group, whose generators are given in GroupHomomorphismByImages format in terms of their action on the underlying group G .

2.1.2 InfoAutGrp

▷ InfoAutGrp (info class)

This is a GAP InfoClass (Reference: InfoClass for a GAP package). By assigning an level in the range 1 to 4 via

Example

```
SetInfoLevel(InfoAutGrp, level);
```

varying levels of information on the progress of the computation, will be obtained.

Example

```
gap> LoadPackage("autpgrp", false);
true
gap> G := PcGroupCode(619031068735, 32); # SmallGroup( 32, 15 );
<pc group of size 32 with 5 generators>
gap> SetInfoLevel( InfoAutGrp, 1 );
gap> AutomorphismGroup(G);
#I step 1: 2^2 -- init automorphisms
#I step 2: 2^2 -- aut grp has size 2
#I step 3: 2^1 -- aut grp has size 32
#I final step: convert
<group of size 64 with 6 generators>
```

The algorithm proceeds by induction down the lower p -central series of G and the information corresponds to the steps of this induction. In the following example we observe that the method also accepts permutation groups as input, provided they satisfy the required filters.

Example

```
gap> G := DihedralGroup( IsPermGroup, 2^5 );;
gap> IsPGroup(G);
true
gap> CanEasilyComputePcgs(G);
true
gap> IsFinite(G);
true
gap> A := AutomorphismGroup(G);
#I step 1: 2^2 -- init automorphisms
#I step 2: 2^1 -- aut grp has size 2
#I step 3: 2^1 -- aut grp has size 8
#I step 4: 2^1 -- aut grp has size 32
#I final step: convert
<group of size 128 with 7 generators>
gap> A.1;
Pcgs([ ( 2,16)( 3,15)( 4,14)( 5,13)( 6,12)( 7,11)( 8,10),
( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16),
( 1, 3, 5, 7, 9,11,13,15)( 2, 4, 6, 8,10,12,14,16),
( 1, 5, 9,13)( 2, 6,10,14)( 3, 7,11,15)( 4, 8,12,16),
( 1, 9)( 2,10)( 3,11)( 4,12)( 5,13)( 6,14)( 7,15)( 8,16) ]) ->
[ ( 1, 2)( 3,16)( 4,15)( 5,14)( 6,13)( 7,12)( 8,11)( 9,10),
( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16),
( 1, 3, 5, 7, 9,11,13,15)( 2, 4, 6, 8,10,12,14,16),
( 1, 5, 9,13)( 2, 6,10,14)( 3, 7,11,15)( 4, 8,12,16),
( 1, 9)( 2,10)( 3,11)( 4,12)( 5,13)( 6,14)( 7,15)( 8,16) ]
gap> Order(A.1);
16
```

Chapter 3

The underlying function

3.1 The underlying function

Underlying the method installation for `AutomorphismGroup` is the function `AutomorphismGroupPGroup`. This function is intended for expert users who wish to influence the steps of the algorithm. Note also that `AutomorphismGroup` will always choose default values.

3.1.1 `AutomorphismGroupPGroup`

▷ `AutomorphismGroupPGroup(G [, $flag$])` (function)

The input is a finite p -group as above and an optional $flag$ which can be true or false. Here the filters for G need not be set, but they should be true for G . The possible values for $flag$ are considered later in Chapter 4. If $flag$ is not supplied, the algorithm proceeds similarly to the method installed for `AutomorphismGroup`, but it produces slightly more detailed output. The output of the function is a record which contains the following fields:

`glAutos`
a set of automorphisms which together with `agAutos` generate the automorphism group;

`glOrder`
an integer whose product with the `agOrders` gives the size of the automorphism group;

`agAutos`
a polycyclic generating sequence for a soluble normal subgroup of the automorphism group;

`agOrder`
the relative orders corresponding to `agAutos`;

`one`
the identity element of the automorphism group;

`group`
the underlying group G ;

`size`
the size of the automorphism group.

We do not return an automorphism group in the standard form because we wish to distinguish between `agAutos` and `glAutos`; the latter act non-trivially on the Frattini quotient of G . This hybrid-group description of the automorphism group permits more efficient computations with it. The following function converts the output of `AutomorphismGroupPGroup` to the output of `AutomorphismGroup`.

3.1.2 ConvertHybridAutGroup

▷ `ConvertHybridAutGroup(A)`

(function)

Returns: A record.

Let A be the automorphism group of a p -group G as computed by `AutomorphismGroupPGroup`. Then `ConvertHybridAutGroup` can compute a pc group isomorphic to the solvable part of A stored in the record component `A.agGroup`. This solvable part forms a subgroup of the automorphism group which contains at least the automorphisms centralizing the Frattini factor of G . The pc group facilitates various further computations with A .

Example

```
gap> LoadPackage("autpgrp", false);
true
gap> H := PcGroupCode(297368117289422176, 729); # SmallGroup (729, 34);
<pc group of size 729 with 6 generators>
gap> A := AutomorphismGroupPGroup(H);
#I step 1: 3^2 -- init automorphisms
#I step 2: 3^1 -- aut grp has size 8
#I step 3: 3^2 -- aut grp has size 72
#I step 4: 3^1 -- aut grp has size 5832
#I final step: convert
rec(
  agAutos :=
    [ Pcgs([ f1, f2, f3, f4, f5, f6 ]) -> [ f2, f1, f3^2, f5^2, f4^2, f6^2 ],
      Pcgs([ f1, f2, f3, f4, f5, f6 ]) -> [ f1, f2^2, f3^2*f5, f4^2*f6, f5,
        f6 ], Pcgs([ f1, f2, f3, f4, f5, f6 ]) ->
        [ f1^2, f2^2, f3*f4^2*f5^2*f6, f4^2*f6, f5^2*f6, f6 ],
      Pcgs([ f1, f2, f3, f4, f5, f6 ]) -> [ f1*f3, f2, f3*f5^2, f4*f6^2, f5,
        f6 ], Pcgs([ f1, f2, f3, f4, f5, f6 ]) ->
        [ f1, f2*f3, f3*f4, f4, f5*f6, f6 ],
      Pcgs([ f1, f2, f3, f4, f5, f6 ]) -> [ f1*f4, f2, f3*f6^2, f4, f5, f6 ],
      Pcgs([ f1, f2, f3, f4, f5, f6 ]) -> [ f1, f2*f4, f3, f4, f5, f6 ],
      Pcgs([ f1, f2, f3, f4, f5, f6 ]) -> [ f1*f5, f2, f3, f4, f5, f6 ],
      Pcgs([ f1, f2, f3, f4, f5, f6 ]) -> [ f1, f2*f5, f3*f6, f4, f5, f6 ],
      Pcgs([ f1, f2, f3, f4, f5, f6 ]) -> [ f1*f6, f2, f3, f4, f5, f6 ],
      Pcgs([ f1, f2, f3, f4, f5, f6 ]) -> [ f1, f2*f6, f3, f4, f5, f6 ] ],
  agOrder := [ 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3 ], glAutos := [ ],
  glOper := [ ], glOrder := 1, group := <pc group of size 729 with
    6 generators>, one := IdentityMapping( <pc group of size 729 with
    6 generators> ), size := 52488 )
gap> ConvertHybridAutGroup( A );
<group of size 52488 with 11 generators>
```

3.1.3 PcGroupAutPGroup

▷ PcGroupAutPGroup(*A*)

(function)

This function computes a pc presentation for the solvable part of the automorphism group *A* defined by *A*.agGroup. *A* is the output of the function AutomorphismGroupPGroup.

Example

```
gap> H := PcGroupCode(297368117289422176, 729);; # SmallGroup (729, 34);
gap> A := AutomorphismGroupPGroup(H);;
#I step 1: 3^2 -- init automorphisms
#I step 2: 3^1 -- aut grp has size 8
#I step 3: 3^2 -- aut grp has size 72
#I step 4: 3^1 -- aut grp has size 5832
#I final step: convert
gap> B := PcGroupAutPGroup( A );
<pc group of size 52488 with 11 generators>
gap> I := InnerAutGroupPGroup( B );
Group([ f5, f4^2*f8, f6^2*f9^2, f11^2, f10^2, <identity> of ... ])
```

Chapter 4

Influencing the algorithm

A number of choices can be made by the user to influence the performance of `AutomorphismGroupPGroup`. Below we identify these choices and their default values used in `AutomorphismGroup` (2.1.1). We use the optional argument `flag` of `AutomorphismGroupPGroup` to invoke user-defined choices. The possible values for `flag` are

`flag = false`

the user-defined defaults are employed in the algorithm. See below for a list of possibilities.

`flag = true`

invokes the interactive version of the algorithm as described in Section 4.5.

In the next section we give a brief outline of the algorithm which is necessary to understand the possible choices. Then we introduce the choices the later sections of this chapter.

4.1 Outline of the algorithm

The basic algorithm proceeds by induction down the lower p -central series of a given p -group G ; that is, it successively computes $\text{Aut}(G_i)$ for the quotients $G_i = G/P_i(G)$ of the descending sequence of subgroups defined by $P_1(G) = G$ and $P_{i+1}(G) = [P_i(G), G]P_i(G)^p$ for $i \geq 1$. Hence, in the initial step of the algorithm, $\text{Aut}(G_2) = GL(d, p)$ where d is the rank of the elementary abelian group G_2 . In the inductive step it determines $\text{Aut}(G_{i+1})$ from $\text{Aut}(G_i)$. For this purpose we introduce an action of $\text{Aut}(G_i)$ on a certain elementary abelian p -group M (the p -multiplier of G_i). The main computation of the inductive step is the determination of the stabiliser in $\text{Aut}(G_i)$ of a subgroup U of M (the *allowable subgroup* for G_{i+1}). This stabiliser calculation is the bottle-neck of the algorithm.

Our package incorporates a number of refinements designed to simplify this stabiliser computation. Some of these refinements incur overheads and hence they are not always invoked. The features outlined below allow the user to select them.

Observe that the initial step of the algorithm returns $GL(d, p)$. But $\text{Aut}(G)$ may induce on G_2 a proper subgroup, say K , of $GL(d, p)$. Any intermediate subgroup of $GL(d, p)$ which contains K suffices for the algorithm and we supply two methods to construct a suitable subgroup: these use characteristic subgroups or invariants of normal subgroups of G . (See Section 4.2.)

In the inductive step an action of $\text{Aut}(G_i)$ on an elementary abelian group M is used. This action is computed as a matrix action on a vector space. To simplify the orbit-stabiliser computation of the

subspace U of M , we can construct the stabiliser of U by iteration over a sequence of $\text{Aut}(G_i)$ -invariant subspaces of M . (See Section 4.3.)

Orbit-stabiliser computations in finite solvable groups given by a polycyclic generating sequence are much more efficient than generic computations of this type. Thus our algorithm makes use of a large solvable normal subgroup S of $\text{Aut}(G_i)$. Further, it is useful if the generating set of $\text{Aut}(G_i)$ outside S is as small as possible. To achieve this we determine a permutation representation of $\text{Aut}(G_i)/S$ and use this to reduce the number of generators if possible. (See Section 4.4.)

4.2 The initialisation step

Assume we seek to compute the automorphism group of a p -group G having Frattini rank d . We first determine as small as possible a subgroup of $GL(d, p)$ whose extension can act on G .

The user can choose the initialisation routine by assigning `InitAutGroup` to any one of the following:

`InitAutomorphismGroupOver`

to use the minimal overgroups: We determine the minimal over-groups of the Frattini subgroup of G and compute invariants of these which must be respected by the automorphism group of G . We partition the minimal overgroups and compute the stabiliser in $GL(d, p)$ of this partition.

The partition of the minimal overgroups is computed using the function `PGFingerprint(G, U)`. This is the time-consuming part of this initialisation method. The user can overwrite the function `PGFingerprint`.

`InitAutomorphismGroupChar`

to use the characteristic subgroups: Compute a generating set for the stabiliser in $GL(d, p)$ of a chain of characteristic subgroups of G . In practice, we construct a characteristic chain by determining 2-step centralisers and omega subgroups of factors of the lower p -central series.

However, there are often other characteristic subgroups which are not found by these approaches. The user can overwrite the function `PGCharSubgroups(G)` to supply a set of characteristic subgroups.

`InitAutomorphismGroupFull`

to use the full $GL(d, p)$.

In the method for `AutomorphismGroup` (2.1.1) we use a default strategy: if the value $\frac{p^d-1}{p-1}$ is less than 1000, then we use the minimal overgroup approach, otherwise the characteristic subgroups are employed. An exception is made for homogeneous abelian groups where we initialise the algorithm with the full group $GL(d, p)$.

4.3 Stabilisers in matrix groups

Consider the i th inductive step of the algorithm. Here $A \leq \text{Aut}(G_i)$ acts as matrix group on the elementary abelian p -group M and we want to determine the stabiliser of a subgroup $U \leq M$.

We use the `MeatAxe` to compute a series of A -invariant subspaces through M such that each factor in the series is irreducible as A -module. Then we use this series to break the computation of $\text{Stab}_A(U)$ into several smaller orbit-stabiliser calculations.

Note that a theoretic argument yields an A -invariant subspace of M a priori: the nucleus N . This is always used to split the computation up. However, it may happen that $N = M$ and hence results in no improvement.

4.3.1 CHOP_MULT

▷ CHOP_MULT

(global variable)

The invariant series through M is computed and used if the global variable CHOP_MULT is set to true. Otherwise, the algorithm tries to determine $\text{Stab}_A(U)$ in one step. By default, CHOP_MULT is true.

4.4 Searching for a small generating set

After each step of the computation, we attempt to find a nice generating set for the automorphism group of the current factor.

If the automorphism group is soluble, we store a polycyclic generating set; otherwise, we store such a generating set for a large soluble normal subgroup S of the automorphism group A , and as few generators outside as possible. If $S = A$ and a polycyclic generating set for S is known, many steps of the algorithm proceed more rapidly.

4.4.1 NICE_STAB

▷ NICE_STAB

(global variable)

It may be both time-consuming and difficult to reduce the number of generators for A outside S . Note that if the initialisation of the algorithm is by `InitAutomorphismGroupOver`, then we always know a permutation representation for A/S . Occasionally the search for a small generating set is expensive. If this is observed, one could set the flag NICE_STAB to false and the algorithm no longer invokes this search.

4.5 An interactive version of the algorithm

The choice of initialisation and the choice of chopping of the p -multiplier can also be driven by an interactive version of the algorithm. We give an example.

```

Example
gap> G := PcGroupCode(2504972218445939562621471375, 256);; # SmallGroup( 2^8, 1000 );
gap> SetInfoLevel( InfoAutGrp, 3 );

gap> AutomorphismGroupPGroup( G, true );
#I step 1: 2^3 -- init automorphisms

choose initialisation (Over/Char/Full):      # we choose Full
#I   init automorphism group : Full
#I step 2: 2^3 -- aut grp has size 168
#I   computing cover
#I   computing matrix action
#I   computing stabilizer of U

```

```

#I    dim U = 3  dim N = 6  dim M = 6

chop M/N and N: (y/n):                # we choose n
#I    induce autos and add central autos
#I    step 3: 2^2 -- aut grp has size 12288
#I    computing cover
#I    computing matrix action
#I    computing stabilizer of U
#I    dim U = 6  dim N = 5  dim M = 8

chop M/N and N: (y/n):                # we choose y
#I    induce autos and add central autos
#I    final step: convert
rec(
  glAutos := [ Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) -> [ f1, f2*f3, f3,
    f4, f5, f6*f7, f7, f8 ],
    Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
    [ f1*f3*f5*f6, f2*f3, f3, f4, f5*f8, f6*f7, f7, f8 ],
    Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
    [ f1*f3, f2*f4, f3, f4*f7, f5*f7, f6*f7*f8, f7, f8 ] ], glOrder := 4,
  agAutos :=
    [ Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) -> [ f1*f4, f2, f3, f4*f8, f5,
      f6, f7, f8 ], Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
      [ f1, f2*f4, f3, f4*f7, f5, f6*f7*f8, f7, f8 ],
      Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
      [ f1*f5, f2, f3, f4, f5, f6, f7, f8 ],
      Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
      [ f1, f2*f5, f3, f4, f5, f6, f7, f8 ],
      Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
      [ f1, f2, f3*f5, f4, f5, f6, f7, f8 ],
      Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
      [ f1*f6, f2, f3, f4, f5*f7*f8, f6, f7, f8 ],
      Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
      [ f1, f2*f6, f3, f4*f7*f8, f5, f6, f7, f8 ],
      Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
      [ f1*f8, f2, f3, f4, f5, f6, f7, f8 ],
      Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
      [ f1, f2*f8, f3, f4, f5, f6, f7, f8 ],
      Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
      [ f1, f2, f3*f8, f4, f5, f6, f7, f8 ],
      Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
      [ f1*f7, f2, f3, f4, f5, f6, f7, f8 ],
      Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
      [ f1, f2*f7, f3, f4, f5, f6, f7, f8 ],
      Pcgs([ f1, f2, f3, f4, f5, f6, f7, f8 ]) ->
      [ f1, f2, f3*f7, f4, f5, f6, f7, f8 ] ],
  agOrder := [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ],
  one := IdentityMapping( <pc group of size 256 with 8 generators> ),
  group := <pc group of size 256 with 8 generators>, size := 32768 )

```

Two points are worthy of comment. First, the interactive version of the algorithm permits the user to make a suitable choice in each step of the algorithm instead of making one choice at the

beginning. Secondly, the output of the `Info` function shows the ranks of the p -multiplier and allowable subgroup, and thus allow the user to observe the scale of difficulty of the computation.

Chapter 5

Additional features of the package

5.1 Additional features of the package

As an additional feature of this package we provide some functions to count extensions of p -groups and Lie algebras over $GF(p)$. These functions have been used in counting the 2-groups of size 2^{10} .

5.1.1 NumberOfPClass2PGroups (for 3 arguments)

▷ `NumberOfPClass2PGroups(n , p , k)` (operation)

This function determines the number of n -generator p -groups of p -class 2 with Frattini subgroup of order 2^k .

5.1.2 NumberOfPClass2PGroups (for 2 arguments)

▷ `NumberOfPClass2PGroups(n , p)` (operation)

This function returns a list of numbers of n -generator p -groups of p -class 2 with Frattini subgroup of order 2^k for k in $1, \dots, n(n+1)/2$.

5.1.3 NumberOfClass2LieAlgebras (for 3 arguments)

▷ `NumberOfClass2LieAlgebras(n , p , k)` (operation)

This function determines the number of n -generator Lie algebras of class 2 over $GF(p)$ with derived Lie subalgebra of dimension k .

5.1.4 NumberOfClass2LieAlgebras (for 2 arguments)

▷ `NumberOfClass2LieAlgebras(n , p)` (operation)

This function returns a list of numbers of n -generator Lie algebras of class 2 over $GF(p)$ with derived Lie subalgebra of dimension k for k in $1, \dots, n(n-1)/2$.

Index

AutomorphismGroup, [5](#)
AutomorphismGroupPGroup, [7](#)

CHOP_MULT, [12](#)
ConvertHybridAutGroup, [8](#)

InfoAutGrp, [5](#)

NICE_STAB, [12](#)
NumberOfClass2LieAlgbras
 for 2 arguments, [15](#)
NumberOfClass2LieAlgebras
 for 3 arguments, [15](#)
NumberOfPClass2PGroups
 for 2 arguments, [15](#)
 for 3 arguments, [15](#)

PcGroupAutPGroup, [9](#)