Group Constructions

—

A GAP 4 Package

by

Hans Ulrich Besche,
and

Bettina Eick

Contents

1 Preface

The determination of all groups of a given order up to isomorphism is a central problem in finite group theory. It has been initiated in 1854 by A. Cayley who constructed the groups of order 4 and 6.

A large number of publications followed Cayley's work. For example, Hall and Senior determined the groups of order 2^n for $n \leq 6$, Neubüser listed the groups of order at most 100 except 64 and 96 and Laue added the groups of order 96, see [HS64], [Neu67] and [Lau82]. These determinations partially relied on the help of computers, but a general algorithm to construct groups had not been used. The resulting catalogue of groups of order at most 100 has been available in GAP 3.

Then Newman and O'Brien introduced an algorithm to determine groups of prime-power order, see [O'B90]. An implementation of this method is available in the ANUPQ share package of GAP. This method has been used to compute the groups of order 2^n for $n \leq 8$ and the groups of order 3^n for $n \leq 6$, see [O'B88], and the resulting groups are available in GAP. Moreover, the large number of groups of order 2^8 shows that algorithmic methods are the only sensible way for group determinations in this range.

In this share package we introduce practical methods to determine up to isomorphism all groups of a given order. The algorithms are described in [BE99a]. These methods have been used to construct the non-nilpotent groups of order at most 1000, see [BE99b]. The resulting catalogue of groups is available within the small groups library of GAP 4.

Our methods are not limited to groups of order at most 1000 and thus may be used to determine all or certain groups of higher order as well. However, it is not easy to say for which orders our methods are still practical and for which not. As a rule of thumb one can say that the number of primes and the size of the prime-powers contained in the factorisation of the given order determine the practicability of the algorithm; that is, the more primes are contained in the factorisation the more difficult the determination gets.

As an example, the construction of all non-nilpotent groups of order $192 = 2^6 \cdot 3$ takes 17 minutes on an PC 400 Mhz. This is a medium sized application of our methods. However, the construction of the groups of order $768 = 2^8 \cdot 3$ takes already rather long (a few days) and can be considered as a limit of our methods. On the other hand, the groups of order $5425 = 5^2 \cdot 7 \cdot 31$ can be determined in 5 sec. Moreover, if the determination of groups is restricted to groups with certain properties, then this might increase the efficiency of the construction process considerably. We include some example applications of our methods to illustrate this at the end of the manual.

Finally, we mention that the correctness of our algorithms is very hard to check for a user; in particular, since there are no other algorithms for the same purpose available, it might be difficult to verify that our methods compute all desired groups. Thus we note here that methods implemented in this share package have been used to compute large parts of the Small Groups library and this, in turn, has been checked by the authors as described in [BE99a] and [BE99b].

Comments and suggestions on this share package are very welcome. Please send them to

beick@tu-bs.de or hubesche@tu-bs.de.

Bug reports should also be e-mailed to either of these addresses.

2 Introduction to GrpConst

This package contains three methods to construct the groups of a certain type up to isomorphism.

The Frattini Extension Method:

This method can be used to determine all soluble groups of a given order. The practicability of the method depends clearly on the chosen order. Furthermore, the efficiency of the method might be increased by restricting the construction to groups with certain properties. This is easily possible for a number of properties; for example, it is useful and straightforward to compute non-nilpotent groups only. (See Chapter 4.)

The Cyclic Split Extension Method:

The cyclic split extension method can be used to list all groups of order $p^n \cdot q$ for different primes p and q which have a normal Sylow subgroup. These groups are also soluble, and hence might also be obtained by the Frattini extension method. However, the cyclic split extension method is more effective on this case. Note that this method relies on a list of groups of order p^n. Moreover, the efficiency of this method depends on an effective method to compute automorphism groups of p-groups. (See Chapter 5.)

The Upwards Extension Method:

The upwards extension method is the most general of the three methods. It can be used to construct all groups of a given order. However, it is the least efficient of the three methods and hence should only be used to construct non-soluble groups. Note that this method needs a list of perfect groups of order dividing the given one. (See Chapter 6.)

Furthermore, the package contains a wrap up function which combines the three methods to a general algorithm to construct the groups of a given order. (See Chapter 3.) Finally, there is an info class `InfoGrpCon` available for the functions of this share package with possible levels 1 up to 4.

3 Construction of All Groups

The following function can be used to determine up to isomorphism all groups of a given order. This method implements a combination of the more specific functions described below.

Note that the chosen combination might not be the best possible for every application. Thus, if this function takes too long to construct the desired groups, then it might still be possible to determine these groups using the functions outlined in the following chapters. Moreover, the functions described in the following chapters provide more facilities and this might help to determine groups with certain properties more efficiently.

1 ► `ConstructAllGroups(` *order* `)` F

Usually the output of this function is a list of groups. The soluble groups in the list are given as pc groups and the others as permutation groups. However, in some cases the output might contain lists of groups as well. The groups is such a list could not be proved to be pairwise non-isomorphic by the algorithm, although this is likely to be the case, see Section 4.4 for further details.

```
gap> ConstructAllGroups( 60 );
[ <pc group of size 60 with 4 generators>,
  <pc group of size 60 with 4 generators>,
  <pc group of size 60 with 4 generators>,
  <pc group of size 60 with 4 generators>,
  <pc group of size 60 with 4 generators>,
  <pc group of size 60 with 4 generators>,
  <pc group of size 60 with 4 generators>,
  <pc group of size 60 with 4 generators>,
  <pc group of size 60 with 4 generators>,
  <pc group of size 60 with 4 generators>,
  <pc group of size 60 with 4 generators>,
  <pc group of size 60 with 4 generators>,
  A5 ]

gap> List( last2, IdGroup );
[ [ 60, 4 ], [ 60, 13 ], [ 60, 6 ], [ 60, 2 ], [ 60, 1 ], [ 60, 7 ],
  [ 60, 3 ], [ 60, 8 ], [ 60, 9 ], [ 60, 12 ], [ 60, 11 ], [ 60, 10 ],
  [ 60, 5 ] ]
```

4 The Frattini Extension Method

This is a method to construct up to isomorphism the soluble groups of a given order. The main function `FrattiniExtensionMethod` to construct groups is described in Section 4.1.

The construction process consists of two parts which can be addressed separately. In the first step a list of possible candidates for the Frattini factors of the desired groups is determined up to isomorphism. See Section 4.2 for the corresponding functions. In the second step the determined candidates are considered one after the other and for each candidate a list of extensions is computed. See Section 4.3 for the available functions.

4.1 The Main Frattini Extension Function

1 ▶ `FrattiniExtensionMethod(` *order* `)` F
▶ `FrattiniExtensionMethod(` *order*`,` *uncoded* `)` F
▶ `FrattiniExtensionMethod(` *order*`,` *flags* `)` F
▶ `FrattiniExtensionMethod(` *order*`,` *flags*`,` *uncoded* `)` F

First we describe the **input** of the function. The *order* is the size of the desired groups. The optional input *uncoded* is a boolean which determines the output format. If it is true, then pc groups are returned. Otherwise, if it is false or not given, then code records describing pc groups are returned (see `PcGroupCodeRec`).

The optional input *flags* is a record which is used to restrict the construction process to groups with certain properties only. This record consists of any of the following entries:

`nilpotent`
: must be `true`. Only nilpotent groups are constructed.

`nonnilpot`
: must be `true`. Only non-nilpotent groups are constructed.

`supersol`
: must be `true`. Only supersoluble groups are constructed.

`nonsupsol`
: must be `true`. Only non-supersoluble groups are constructed.

`pnormal`
: must be a list of primes. Only groups with normal Sylow p-subgroup for all p in the given list are constructed.

`nonpnorm`
: must be a list of primes. Only groups without normal Sylow p-subgroup for all p in the given list are constructed.

If a particular entry is not set, then no restriction on the groups is assumed. The default is an empty record of flags. Any combination of flags is possible. However, not all combinations make sense; For example, if `nilpotent` and `nonnilpotent` are both true, then the algorithm will return the empty list. If `nonnilpot` is true and `pnormal` is the list [3], then the non-nilpotent groups whose Sylow 3-subgroup is normal will be computed.

The **output** of the function is usually a list of pc groups or code records depending on *uncoded*. However, it may happen that the output list contains not only pc groups or codes, but also lists of pc groups or codes. This means that

the groups in such a sublist are probably non-isomorphic, but the algorithm did not do a final verification, since this would be time-consuming. If desired, then the user might do a verification using the function `DistinguishGroups` described below.

Moreover, it might be worth noting that the groups in such sublists of the output list are always reduced by the random isomorphism test (see the Section on `Random Isomorphism Testing` in the reference manual). Hence the probability that there are still isomorphisms between groups in this list is less than 2^{-100}.

```
gap> flags := rec( nonnilpot := true, pnormal := [3] );
rec( nonnilpot := true, pnormal := [ 3 ] )
gap> grps := FrattiniExtensionMethod( 24, flags, true );
[ <pc group with 4 generators>, <pc group with 4 generators>,
  <pc group with 4 generators>, <pc group with 4 generators>,
  <pc group with 4 generators>, <pc group with 4 generators>,
  <pc group with 4 generators> ]
gap> List( last, IdGroup );
[ [ 24, 1 ], [ 24, 5 ], [ 24, 8 ], [ 24, 6 ], [ 24, 7 ], [ 24, 4 ],
  [ 24, 14 ] ]

gap> FrattiniExtensionMethod( 8 );
[ rec( code := 323, order := 8, isFrattiniFree := false, first := [ 1, 1, 2 ],
      socledim := [ 1 ], extdim := [ 2, 2 ], isUnique := true ),
  rec( code := 34, order := 8, isFrattiniFree := false, first := [ 1, 1, 3 ],
      socledim := [ 1, 1 ], extdim := [ 2 ], isUnique := true ),
  rec( code := 36, order := 8, isFrattiniFree := false, first := [ 1, 1, 3 ],
      socledim := [ 1, 1 ], extdim := [ 2 ], isUnique := true ),
  rec( code := 2343, order := 8, isFrattiniFree := false,
      first := [ 1, 1, 3 ], socledim := [ 1, 1 ], extdim := [ 2 ],
      isUnique := true ),
  rec( code := 0, order := 8, isFrattiniFree := true, first := [ 1, 1, 4 ],
      socledim := [ 1, 1, 1 ], extdim := [  ], isUnique := true ) ]
```

4.2 The Construction of Frattini Free Groups

A finite group is called **Frattini free** if it has a trivial Frattini subgroup. As candidates for the Frattini factors of the groups of size *order*, we compute Frattini free groups of suitable size dividing *order*.

1 ▶ `FrattiniFactorCandidates(` *order*`,` *flags* `)` F
▶ `FrattiniFactorCandidates(` *order*`,` *flags*`,` *uncoded* `)` F

The input is similar to the input for the function `FrattiniExtensionMethod`.

The output is a list of candidates for the Frattini factors of the desired groups, i.e. the groups of size *order* possibly restricted by *flags*. By default the groups are returned as codes which may be changed using the boolean *uncoded*.

Note that the computed list is always reduced to isomorphism type representatives. Moreover, it might happen that some of the Frattini free groups are not realised as Frattini factors of a group of size *order*. However, in practice this is a very rare case.

Furthermore, note that for this part of the Frattini extension method the restriction to the positive properties `nilpotent`, `supersol` and `pnormal` in the flags record will reduce the amount of computation considerably, while the negative properties do not have such a major influence on the efficiency of this method.

```
gap> flags := rec( nonsupsol := true );
rec( nonsupsol := true )
gap> FrattiniFactorCandidates( 24, flags, true );
[ <pc group with 4 generators>, <pc group with 3 generators>,
  <pc group with 4 generators> ]
gap> List(last, IdGroup);
[ [ 24, 12 ], [ 12, 3 ], [ 24, 13 ] ]
```

4.3 The Determination of Frattini Extensions

A group H is a **Frattini extension** of a group G if there exists a normal subgroup N of H such that $H/N \cong G$ and $N \leq \phi(H)$ holds. Clearly, each finite group can be obtained as a Frattini extension of a Frattini free group.

1 ▶ `FrattiniExtensions( `*code/group*`, `*order*` )` F
▶ `FrattiniExtensions( `*code/group*`, `*order*`, `*uncoded*` )` F

Here the default input is a Frattini free group described by a code and the size *order* of the groups which shall be constructed. Alternatively, one can input a Frattini free group as pc group. Moreover, it is possible to give a list of codes or pc groups at once. The flag *uncoded* changes the output format to pc groups instead of codes as above.

The output of this function is similar to the output of the function `FrattiniExtensionMethod`.

```
gap> G := SmallGroup( 24, 12 );
<pc group of size 24 with 4 generators>
gap> FrattiniSubgroup(G);
Group([  ])
gap> FrattiniExtensions( G, 48, true );
[ <pc group with 5 generators>, <pc group with 5 generators>,
  <pc group with 5 generators> ]
gap> List( last, IdGroup);
[ [ 48, 29 ], [ 48, 30 ], [ 48, 28 ] ]

gap> cand := FrattiniFactorCandidates( 6, rec() );
[ rec( code := 25, order := 6, isFrattiniFree := true, first := [ 1, 2, 3 ],
      socledim := [ 1 ], extdim := [  ], isUnique := true ),
  rec( code := 1, order := 6, isFrattiniFree := true, first := [ 1, 1, 3 ],
      socledim := [ 1, 1 ], extdim := [  ], isUnique := true ) ]
gap> FrattiniExtensions( cand, 12 );
[ rec( code := 6442, order := 12, isFrattiniFree := false,
      first := [ 1, 2, 3 ], socledim := [ 1 ], extdim := [ 2 ],
      isUnique := true ),
  rec( code := 266, order := 12, isFrattiniFree := false,
      first := [ 1, 1, 3 ], socledim := [ 1, 1 ], extdim := [ 2 ],
      isUnique := true ) ]
```

4.4 Verifying non-isomorphism

The output of the functions `FrattiniExtensionMethod` or `FrattiniExtensions` might contain sublists of groups. That means, that the groups contained in sublists could not be distinguished up to isomorphism by the Frattini extension method. However, the groups have gone through the random isomorphism test and hence it is likely that they are not isomorphic.

Here we provide a tool that can be used to try to prove that these groups are non-isomorphic. This is not done automatically within the Frattini extension method, since it might be time consuming and many users might not be interested in a complete verification of non-isomorphism.

To distinguish groups we compute invariants of the given groups. Clearly, if the invariants differ, then we obtain that the corresponding groups are not isomorphic. However, the converse is not true and hence we might not succeed to distinguish all non-isomorphic groups in a given list. See [BE99a] for a description of the used invariants.

1 ► `DistinguishGroups(` *list*`,` *bool* `)` F

The function `DistinguishGroups` takes as input *list* a list as described for the output of `FrattiniExtensions`. It returns a similar list, where the sublists contained in *list* are split up.

There are two levels to operate the function `DistinguishGroups` which are controlled by the second input parameter *bool* of the function. If *bool* is `false`, then only few invariants are computed, if it is `true`, then we try also the more complicated invariants. Clearly, if *bool* is `false`, then the result is obtained faster, but if *bool* is `true`, then we might distinguish more groups.

If `DistinguishGroups` fails to split up the input list completely, then a user might use the general purpose function `IsomorphismGroups` to prove the non-isomorphism between the remaining groups. However, this might be a time consuming computation.

5 The Cyclic Split Extension Method

This is a method to construct up to isomorphism the groups of order $p^n \cdot q$ for different primes p and q which have a normal Sylow subgroup. We first describe the main function for this method and then functions for a slightly more low level access to the algorithms.

Note that all functions described in this chapter rely on an efficient method for `AutomorphismGroup` for p-groups. Such a method is provided in the package AutPGrp. Thus it is useful to install and load this share package before using the functions described in this chapter.

5.1 The Main Function

1 ▶ `CyclicSplitExtensionMethod( p, n, q )` F
▶ `CyclicSplitExtensionMethod( p, n, q, uncoded )` F

Clearly, each of the computed groups is a split extension of a group of order p^n and the cyclic group of order q. The output is a record with three entries *up*, *down* and *both*. Each of these contains a list of groups, *both* the nilpotent groups, *up* the remaining groups with a normal Sylow p-subgroup and *down* the remaining groups with normal Sylow q-subgroup.

As in Chapter 4 all groups are described as codes. Setting *uncoded* to true, the function will return pc groups instead.

If one wants to construct the groups of order $p^n \cdot q$ for fixed p and several primes q, it is more efficient to do this in one go. Thus it is possible to hand a list of primes for the input q.

```
gap> CyclicSplitExtensionMethod( 2,2,7, true );
rec( up := [  ],
  down  := [ <pc group of size 28 with 3 generators>,
             <pc group of size 28 with 3 generators> ],
  both  := [ <pc group of size 28 with 3 generators>,
             <pc group of size 28 with 3 generators> ] )

gap> CyclicSplitExtensionMethod( 2,2,[3,5], true );
rec( up := [ <pc group of size 12 with 3 generators> ],
  down  := [ <pc group of size 12 with 3 generators>,
             <pc group of size 20 with 3 generators>,
             <pc group of size 20 with 3 generators>,
             <pc group of size 12 with 3 generators>,
             <pc group of size 20 with 3 generators> ],
  both  := [ <pc group of size 12 with 3 generators>,
             <pc group of size 20 with 3 generators>,
             <pc group of size 12 with 3 generators>,
             <pc group of size 20 with 3 generators> ] )
```

Note that the function `CyclicSplitExtensionMethod` requires that the groups of order p^n are given within the SmallGroups Library.

5.2 The Underlying Functions

It is possible to construct the cyclic extensions of a single group of order p^n only. The output is as above.

1 ▶ `CyclicSplitExtensions( G, q )` F
▶ `CyclicSplitExtensions( G, q, uncoded )` F

Moreover, the computation of the record entry *up* and the record entry *down* can be separated by using the following functions.

2 ▶ `CyclicSplitExtensionsUp( G, q )` F
▶ `CyclicSplitExtensionsUp( G, q, uncoded )` F

3 ▶ `CyclicSplitExtensionsDown( G, q )` F
▶ `CyclicSplitExtensionsDown( G, q, uncoded )` F

The input for these functions is the same as above. The first function returns a list of groups with one normal subgroup of order p^n and the second a list of groups with one normal subgroup of order q.

```
gap> G := SmallGroup( 16, 10 );;
gap> CyclicSplitExtensionsUp( G, 3, true );
[ <pc group with 5 generators> ]

gap> G := SylowSubgroup( SymmetricGroup(4), 2);
Group([ (1,2), (3,4), (1,3)(2,4) ])
gap> CyclicSplitExtensionsDown( G, 3 );
[ rec( code := 6562689, order := 24 ),
  rec( code := 2837724033, order := 24 ) ]
```

6 The Upwards Extension Method

This is a method to construct up to isomorphism the finite groups of a given order. For this purpose it will loop over all possible perfect groups and construct upwards extensions by soluble groups. This, in turn, is done by iterated cyclic extensions.

Since this method is less efficient than the above two methods, it will usually only be used for the determination of non-soluble groups.

1 ► `UpwardsExtensions( G, s )` F

Let G be a permutation group and s a positive integer. This function returns a list corresponding to `DivisorsInt(s)`. Let t be the i-th divisor of s. Then the i-th entry in the output is a list of all extensions of G by a soluble group of order t up to isomorphism. The returned groups are permutation groups again.

Typically, this function is applied to perfect groups G, which may be obtained from the perfect groups catalogue in GAP (see the Section on `Finite perfect groups` in the reference manual).

The most time-consuming part of the computation in `UpwardsExtensions` is the isomorphism test. The following function does no reduction to isomorphism type representatives and hence is much more efficient.

2 ► `CyclicExtensions( G, p )` F

Here G should be a permutation group and p a prime. This function computes a list of permutation groups containing the upwards extensions of G by the cyclic group of order p, but not reduced to isomorphism type representatives.

There is an info class `InfoUpExt` available with values from 1 to 3.

```
gap> G := PerfectGroup( IsPermGroup, 120, 1 );
A5 2^1
gap> c := CyclicExtensions( G, 2 );;
gap> List( c, IdGroup );
[ [ 240, 94 ], [ 240, 93 ], [ 240, 90 ], [ 240, 89 ] ]
gap> H := c[1];
<permutation group of size 240 with 2 generators>
gap> CyclicExtensions( H, 2 );;
gap> List(last, IdGroup);
[ [ 480, 960 ], [ 480, 955 ], [ 480, 222 ], [ 480, 222 ], [ 480, 953 ],
  [ 480, 953 ], [ 480, 957 ], [ 480, 957 ], [ 480, 949 ], [ 480, 950 ],
  [ 480, 219 ], [ 480, 219 ] ]

gap> u := UpwardsExtensions( G, 4 );;
gap> List( u, Length );
[ 1, 4, 14 ]
gap> List( u[3], IdGroup);
[ [ 480, 960 ], [ 480, 959 ], [ 480, 950 ], [ 480, 222 ], [ 480, 221 ],
  [ 480, 947 ], [ 480, 949 ], [ 480, 219 ], [ 480, 948 ], [ 480, 218 ],
  [ 480, 955 ], [ 480, 957 ], [ 480, 953 ], [ 480, 946 ] ]
```

In case that we want to extend a perfect group with trivial centre, then there is a better algorithm available. This is implemented as well and can be used with the following functions.

3 ► `UpwardsExtensionsNoCentre( G, s )` F

Let G be a perfect permutation group with trivial centre and s a positive integer. This function returns a list of all extensions of G by a soluble group of order s up to isomorphism. The returned groups are permutation groups again. Note that, in difference to `UpwardsExtensions` this function does not return the extensions by groups of order dividing s. Moreover, the implementation of the function requires that all soluble groups of order s are available as `SmallGroups`. The implementation then uses the following function to determine groups.

4 ► `ExtensionsByGroupNoCentre( G, H )` F

Let G be a perfect permutation group with trivial centre and H a soluble group. This functions returns all extensions of G by H up to isomorphism.

7 Examples with Runtimes

In this chapter we outline some examples of applications of the methods described above. The examples are meant to give an idea of the possible applications of the package. Thus we included runtimes for all examples, but omitted the output in some cases, since it would be too long to be printed. The runtimes have been obtained on a 400 Mhz PC running under Linux.

```
gap> ConstructAllGroups( 60 );; time;
4080
```

In the following examples we observe that the restriction to certain groups is often helpful. Note that nilpotent groups can often be obtained as direct product of p-groups which, in turn, might better be constructed by p-group generation methods.

```
gap> FrattiniExtensionMethod( 5^3 * 7 * 31, true );;
gap> time;
13670

gap> flags := rec( nonnilpot := true );;
gap> FrattiniExtensionMethod( 5^3 * 7 * 31, flags, true );;
gap> time;
8400

gap> flags := rec( nonsupsol := true );;
gap> FrattiniExtensionMethod( 5^3 * 7 * 31, flags, true );;
gap> time;
3640

gap> flags := rec( nonpnorm := [31] );;
gap> FrattiniExtensionMethod( 5^3 * 7 * 31, flags, true );;
gap> time;
1740
```

Next we consider groups of an order whose factorision contains a large prime. Note that the Small Groups library contains a generic method to construct the groups whose order is the product of at most 3 primes. This method is used in `ConstructAllGroups` which is therefore much more efficient in the next example.

```
gap> FrattiniExtensionMethod( 10007 * 2, true );
[ <pc group of size 20014 with 2 generators>,
  <pc group of size 20014 with 2 generators> ]
gap> time;
87950
```

```
gap> flags := rec( nonnilpot := true );;
gap> FrattiniExtensionMethod( 10007 * 2, flags, true  );
[ <pc group of size 20014 with 2 generators> ]
gap> time;
48950

gap> ConstructAllGroups( 10007 * 2 );
[ <pc group of size 20014 with 2 generators>,
  <pc group of size 20014 with 2 generators> ]
gap> time;
30
```

Finally we consider an order which factorises in seven primes and contains a moderately large prime power. Note that there are 943 non-nilpotent groups of order $288 = 2^5 \cdot 3^2$ while there are only 90 such groups without normal Sylow subgroup.

```
gap> flags := rec( nonnilpot := true );;
gap> FrattiniExtensionMethod( 2^5 * 3^2, flags, true );;
gap> time;
656630

gap> flags := rec( nonpnorm := [2,3] );;
gap> FrattiniExtensionMethod( 2^5 * 3^2, flags, true );;
gap> time;
58180
```

Bibliography

[BE99a] Hans Ulrich Besche and Bettina Eick. Construction of finite groups. *J. Symb. Comput.*, 27:387 – 404, 1999.

[BE99b] Hans Ulrich Besche and Bettina Eick. The groups of order at most 1000 except 512 and 768. *J. Symb. Comput.*, 27:405 – 413, 1999.

[HS64] Marshall Hall, Jr. and James K. Senior. *The Groups of Order* 2^n *(*$n \leq 6$*)*. Macmillan Company, 1964.

[Lau82] Reinhard Laue. *Zur Konstruktion und Klassifikation endlicher auflösbarer Gruppen*, volume 9 of *Bayreuther Mathematische Schriften*. 1982.

[Neu67] Joachim Neubüser. *Die Untergruppenverbände der Gruppen der Ordnungen* $\leq$ 100 *mit Ausnahme der Ordnungen 64 und 96*. Habilitationsschrift, Universität Kiel, 1967.

[O'B88] Eamonn A. O'Brien. *The groups of order dividing* 256. PhD thesis, Australian National University, 1988.

[O'B90] Eamonn A. O'Brien. The p-group generation algorithm. *J. Symb. Comput.*, 9:677 – 698, 1990.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "`PermutationCharacter`" comes before "permutation group".